PipeWeaver: Addressing Data Dynamicity in Large Multimodal Model Training with Dynamic Interleaved Pipeline

Zhenliang Xue[1] Hanpeng Hu[2] Xing Chen[2] Yimin Jiang[*] Yixin Song[1,3]
Zeyu Mi[1,3] Yibo Zhu[2] Daxin Jiang[2] Yubin Xia[1] Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]StepFun [3]Zenergize AI [*]Unaffiliated

Abstract

Large multimodal models (LMMs) have demonstrated excellent capabilities in both understanding and generation tasks with various modalities. While these models can accept flexible combinations of input data, their training efficiency suffers from two major issues: pipeline stage imbalance caused by heterogeneous model architectures, and training data dynamicity stemming from the diversity of multimodal data.

In this paper, we present PipeWeaver, a dynamic pipeline scheduling framework designed for LMM training. The core of PipeWeaver is *dynamic interleaved pipeline*, which searches for pipeline schedules dynamically tailored to current training batches. PipeWeaver addresses issues of LMM training with two techniques: adaptive modality-aware partitioning and efficient pipeline schedule search within a hierarchical schedule space. Meanwhile, PipeWeaver utilizes SEMU (Step Emulator), a training simulator for multimodal models, for accurate performance estimations, accelerated by spatial-temporal subgraph reuse to improve search efficiency. Experiments show that PipeWeaver can enhance LMM training efficiency by up to 97.3% compared to state-of-the-art systems, and demonstrate excellent adaptivity to LMM training's data dynamicity.

1 Introduction

Transformer-based models [11, 33, 48, 56] have demonstrated remarkable capabilities in multimodal understanding, reasoning, and generation, establishing themselves as foundational architectures for next-generation large multimodal models (LMMs) [1, 3, 10, 20, 32, 33, 36, 44]. Modern LMMs integrate multiple *modality modules*, including encoders, decoders, and the backbone model connected via modality adapters. This architecture enables flexible and seamless processing of interleaved modality data (e.g., text, images, video), thereby supporting diverse task paradigms, including multimodal document understanding [8] and multi-turn dialogs [35].

However, this flexibility makes training LMMs a challenging task due to the irregular and fluctuating execution latencies across different modality modules and data batches, introducing the unique challenge of *dynamic imbalance*. This problem arises from two interrelated factors:

- **Pipeline Stage Imbalance:** LMMs exhibit architectural heterogeneity due to diverse modality modules with distinct parameter shapes and operator types (e.g., attention, GEMM, convolution), which creates skewed computational workloads and varying memory access patterns, complicating the partitioning of model execution into balanced pipeline stages. As a result, even with exhaustive enumeration of all possible splits of model layers, the "optimal" partitioning still incurs a 22.8% pipeline bubble overhead for a 37B-parameter LMM, mainly due to the significant discrepancy between heterogeneous layers that hinders perfectly balanced pipeline stage partitioning.
- **Training Data Dynamicity:** This inherent heterogeneity is further compounded by the diversity of multimodal training data, which consists of large-scale, diverse datasets with highly variable modality distributions across data batches. This data dynamicity makes one-fits-all pipeline schedules infeasible for LMM training. Despite data packing techniques [26, 52] designed to produce more balanced datasets, computational imbalance persists, with the largest data sample incurring 4.15× greater computational load than the smallest in our experiment. Such disparities induce substantial workload fluctuations between data batches, necessitating adaptive pipeline scheduling.

Current distributed training systems fail to address the dynamic imbalance problem, as they either lack awareness of pipeline stage imbalance during partitioning or employ static pipeline schedules that ignore training data dynamicity. Megatron-LM [42], the state-of-the-art framework for training unimodal large language models, focuses solely on model architectures comprised of repeated homogeneous transformer layers, making its design inadequate for heterogeneous modality modules. Recent training systems for multimodal models like Spindle [51] and Optimus [15], rely on static pipeline schedules specialized for fixed sets of training tasks, overlooking the dynamic nature of multimodal data and resulting in suboptimal pipeline performance.

A promising solution to this issue is *dynamically searching for pipeline schedules tailored to the content of subsequent data batches*. However, dynamic pipeline scheduling faces a critical trade-off between search efficiency and estimation accuracy. First, the search process must complete within a tight time budget matching the latency of one training iteration (e.g., 10 seconds). This demands a carefully designed search space for rapid pipeline schedule exploration. Second, the search process requires frequent performance estimations for candidate pipeline schedules. Existing training schedulers employ oversimplified coarse-grained cost models to reduce estimation overhead. For example, Alpa's cost model assumes uniform microbatch latencies [58], ignoring dynamic imbalance and yielding inaccurate predictions for LMM workloads. While white-box modeling simulators [14, 16, 49] can achieve accurate performance estimation through detailed operator-level simulation, they necessitate full-model re-simulation whenever workloads are updated. This process incurs prohibitive overhead (e.g., minutes per simulation), significantly slowing down the search process.

In this paper, we propose the *dynamic interleaved pipeline* and a dynamic multimodal pipeline scheduling framework named PipeWeaver. Our key observation is that modality-specific pipeline stage partitioning and microbatch splitting dissect mixed-modal training data into isolated single-modal components, which enables independent tuning to achieve globally balanced execution latencies across both modality modules and data microbatches. Based on this observation, we introduce *modality-aware partitioning*, which individually splits pipeline executions and data microbatches of each modality module into balanced pipeline segments (i.e., sequences of pipeline stages, defined in §5) to tackle the issue of pipeline stage imbalance.

To address training data dynamicity, PipeWeaver searches for efficient pipeline schedules that adaptively interleave pipeline segments from different modality modules according to the content of current training batch. We design a *hierarchical schedule space* with efficient exploration of pipeline schedules via tailored search algorithms. Leveraging the inherent hierarchy of modality modules, pipeline stages, and model layers, we develop specialized search strategies that align with the scheduling properties of each level. Concurrently, we implement a multimodal training simulator named SEMU (Step Emulator) that accelerates performance estimation during pipeline schedule exploration through *spatial-temporal subgraph reuse*. This mechanism identifies recurrent spatial computation patterns by clustering adjacent operators (e.g., nodes within a pipeline stage) into reusable subgraphs, which can be reused across different simulator invocations, effectively enhancing search efficiency while preserving estimation accuracy.

We implement PipeWeaver on top of Megatron-LM, the state-of-the-art distributed training framework for transformer-based large language models. In addition to the scheduling algorithms, we augment Megatron-LM with a reconfigurable pipeline mechanism to support the deployment of dynamic interleaved pipeline. Experiments demonstrate PipeWeaver's superior performance to baseline training systems with up to 97.3% performance improvement. Furthermore, PipeWeaver shows excellent adaptivity to dynamic workloads of LMM training, and manages to maintain maximal utilization of hardware resources throughout the training process.

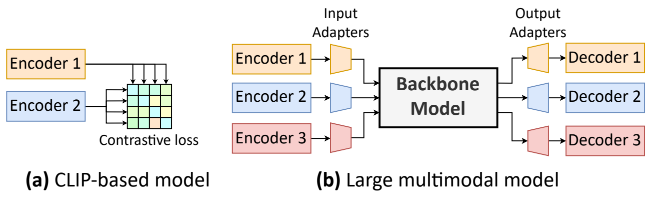


Figure 1. Comparison between CLIP-based models and LMMs.

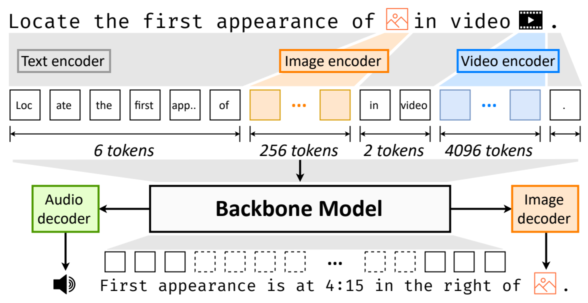


Figure 2. An example of LMM that uses a large language model as the backbone. The user prompt contains an image and a video clip, and is converted into tokens by corresponding modality encoders, which are further processed by the backbone model to produce the response in multimodal text or speech audio.

Contributions. We make the following contributions in designing PipeWeaver and SEMU:

- We propose dynamic interleaved pipeline to address the issue of dynamic imbalance, and present PipeWeaver, a dynamic pipeline scheduling framework for LMM training.
- We design a hierarchical schedule space for rapid pipeline schedule search, adaptively optimizing the training pipeline under dynamic workloads.
- We implement SEMU for fast and accurate performance estimations of multimodal training workloads.
- We evaluate PipeWeaver against three state-of-the-art training systems on five LMM models, demonstrating improvement up to 97.3% in training efficiency.

2 Dynamic Imbalance Characterization

2.1 Large Multimodal Models

Multimodal models have long sought to bridge heterogeneous modalities by learning unified semantic representations [43]. Early CLIP-based multimodal models [40] pioneered the dual-encoder architecture (e.g., ViT for images, BERT for text) and employed contrastive loss [6] to align cross-modal embeddings in shared semantic spaces.

The advent of large multimodal models (LMM) marks a paradigm shift that extends beyond representation alignment to encompass multimodal generation capabilities. This evolution is driven by the integration of large-scale transformer-based architectures, such as large language models (LLMs), Vision Transformers (ViT), and diffusion transformers (DiT). LMMs integrate encoders with the backbone model using modality adapters, as illustrated in Fig.1. The backbone model subsequently generates output representations either through

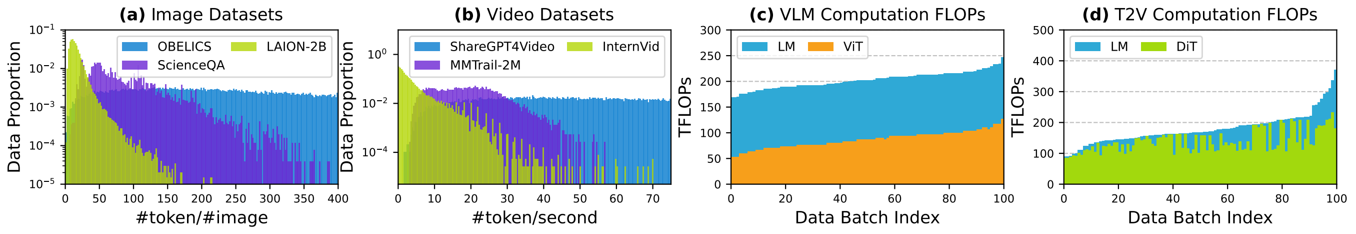


Figure 3. (a-b) The distribution of text tokens per image in OBELICS [27], LAION-2B [41], and ScienceQA [31] image datasets, and text tokens per second in ShareGPT4Video [5], InternVid [50], and MMTrail-2M [7] video datasets. The Y-axis shows normalized data proportions. **(c–d)** Computational requirements across 100 data batches for VLM [33] and T2V models after data packing. The X-axis represents the data batch index sorted in ascending order by total computational cost, while the Y-axis indicates floating-point operations measured in TFLOPs.

autoregressive processes or diffusion mechanisms. These representations are then converted into human-perceivable formats, including text, audio, and images, via specialized modality decoders. The scalable and generative architecture of LMM significantly broadens the spectrum of supported tasks compared to CLIP-based models. For instance, in an LMM (Fig.2), users can interleave text, images, and video within a single query and iteratively refine their queries by appending follow-up questions to create multi-turn dialogs [10, 22, 35]. The LMM can respond in text or speech [20], and generate images using diffusion modules [28, 32, 37].

Large transformer-based models are typically trained with hierarchical distributed schemes such as data parallelism (DP), pipeline parallelism (PP), and tensor parallelism (TP) [42]. Pipeline parallelism partitions model layers into multiple model chunks placed on different machines (i.e., pipeline ranks). Each model chunk can perform both forward and backward stage computations. Data batches are split into smaller microbatches and passed between pipeline stages with point-to-point (P2P) communications.

2.2 Sources of Dynamic Imbalance

Due to the heterogeneity of multimodal model architectures and the diversity of training data, the training process for LMMs differs significantly from that for unimodal models in *pipeline stage imbalance* and *training data dynamicity*.

Pipeline Stage Imbalance. Achieving optimal pipeline efficiency requires balanced stage partitioning to minimize pipeline bubbles, yet inherent computational disparities between modality modules introduce fundamental challenges. For example, given a 37B vision language model (VLM) with a 5B ViT vision encoder (64 layers) and a 32B language model (64 layers), and a data batch with 8192 text tokens and 8 images, each ViT layer processes 8 images in 6.75ms, while each LM layer requires 10.5ms for 8192 tokens. Partitioning across 16 pipeline stages by exhaustively exploring all possible layer combinations results in stage latencies ranging from 63ms to 73.5ms, i.e., 16.7% variation. This imbalance introduces 22.8% additional pipeline bubbles and increases end-to-end training latency by 5%, highlighting the inherent difficulty of identifying the perfectly balanced partitioning.

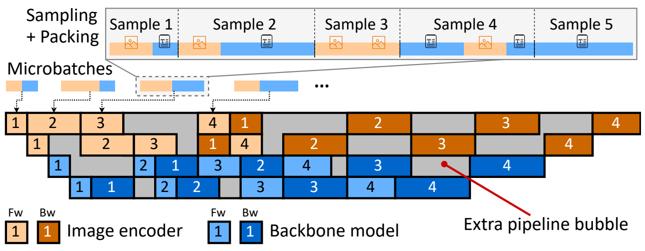


Figure 4. Illustration for the impact of dynamic imbalance.

Training Data Dynamicity. This challenge is exacerbated by the dynamic heterogeneity of multimodal training data, where compositional variability across batches induces fluctuating computational demands. Multimodal data are comprised of various datasets, e.g., image-description pairs [4, 41], video with captions [5, 7, 50, 54] and interleaved image-text content [27, 61], etc. A data sample typically consists of images and video clips accompanied by description texts. Multimodal datasets exhibit significant diversity in modality ratios (as shown in Fig.3a–b). Taking image datasets as an example, the LAION-2B [41] dataset is made from images paired with short captions, with a small text-image ratio (16.4 tokens/image), while the OBELICS [27] dataset contains full-length multimodal documents with diverse text-image ratios (0.4 to 3115 tokens/image).

During training, data samples can be packed into larger batches for better computation efficiency [26, 52]. For vision language models, data samples are usually packed up to the context length of the model (e.g., 8192 tokens). For text-to-video (T2V) diffusion models, video clips with similar durations and aspect ratios can be grouped together and processed in one batch [32, 37]. However, data packing fails to achieve balanced workloads when dealing with multiple modalities due to their divergent distributions (as shown in Fig.3c–d), creating an intractable trade-off where balancing one modality may exacerbate imbalance in others.

2.3 Negative Impact on Pipeline Parallelism

These dual factors of stage imbalance and data dynamicity result in the *dynamic imbalance* problem, which fundamentally undermines the conventional distributed training pipeline. As shown in Fig.4, dynamic imbalance introduces signifi-

Model Setup	Time (s)	PFLOPs	MFU
LM 7B	4.068	12.8	0.400
ViT 2B + LM 5B (static data)	4.567	12.7	0.351
ViT 2B + LM 5B (dynamic data)	6.789	12.8	0.239

Table 1. Training performance of 7B models on 8 GPUs (TP=2, PP=4). "PFLOPs" is the number of floating-point operations of one iteration in petaFLOPs, which is controlled to a fixed value. "MFU" is the model FLOPs utilization.

cant pipeline bubbles in Megatron-LM's 1F1B pipeline and severely degrades throughput.

To quantify this, we compare two setups with the same number of parameters: unimodal LM (7B) and vision LM (ViT 2B + LM 5B). Under identical 1F1B pipeline configurations (balanced parameter partitioning, fixed computational budgets), VLM incurs 12.5% overhead on static data due to stage imbalance. With real-world dynamic data, overhead escalates to 40.3% (Table 1). These findings demonstrate the necessity for adaptive pipeline scheduling in LMM training. Notably, data characteristics can be prefetched from the training dataset, enabling asynchronous pipeline scheduling for upcoming data batches.

3 Approach Overview

This section first presents the problem formulation and then identifies the challenges. It finally presents an overview of our dynamic interleaved pipeline approach (§3.3).

3.1 Pipeline Scheduling Problem Formulation

An intuitive approach to tackle the dynamic imbalance problem is to design a dynamically adaptable pipeline schedule. Suppose we have obtained computation latency lat_u and memory consumption differences Δ_u for every stage u. Let s_u, t_u denote the start and end timestamps of stage u, with $t_u = s_u + \mathrm{lat}_u$. For stage u, v, enforce precedence $t_u \le s_v$ if v succeeds u. For pipeline rank p with assigned stage set $\mathbb{S}_p$, let $o_{u,v} \in \{0, 1\}$ $(u, v \in \mathbb{S}_p)$ to indicate whether stage u is executed before stage v. Our optimization target is to minimize the pipeline latency $\max\{t_u : u \in \mathbb{S}_p, 1 \le p \le P\}$, s.t.

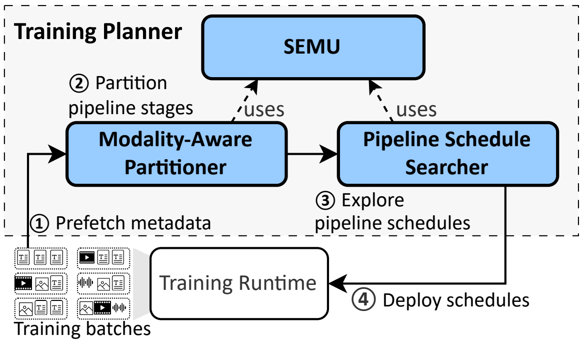
$$\textstyle\sum_{u\in\mathbb{S}_p} o_{u,v} \cdot \Delta_u \le \mathbb{C}_p, \quad \forall v \in \mathbb{S}_p$$

where P is pipeline parallelism size and $\mathbb{C}_p$ is the per-rank memory constraint. A naive approach to solve this problem is leveraging off-the-shelf integer linear programming (ILP) solvers [23, 45], and we take it as a baseline approach [17].

3.2 Challenges

The baseline approach, however, is impractical when directly applied to LMM training, mainly due to the lack of a precise cost model under a multimodal training scenario and tremendous complexity of ILP solving.

Cost Model Accuracy. Accurate estimation of stage latencies (lat_u) and memory consumptions (Δ_u) is critical for effective pipeline scheduling, especially for LMM training with



Training Planner
SEMU
② Partition pipeline stages
uses
uses
Modality-Aware Partitioner
Pipeline Schedule Searcher
① Prefetch metadata
③ Explore pipeline schedules
Training Runtime
④ Deploy schedules
Training batches

Figure 5. Overall workflow of PipeWeaver's training planner.

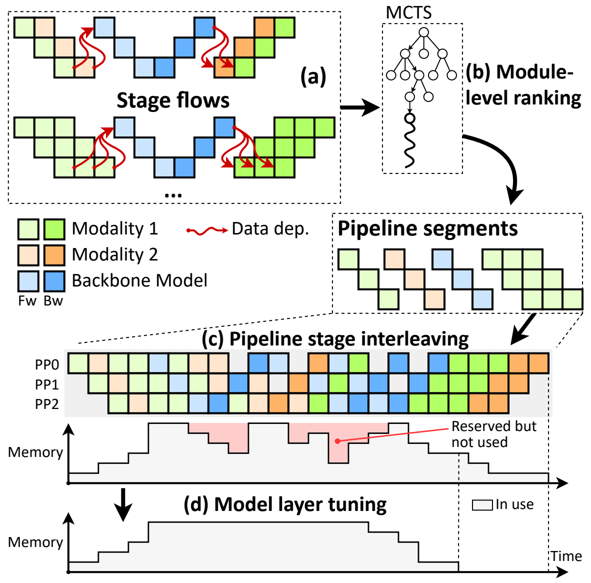
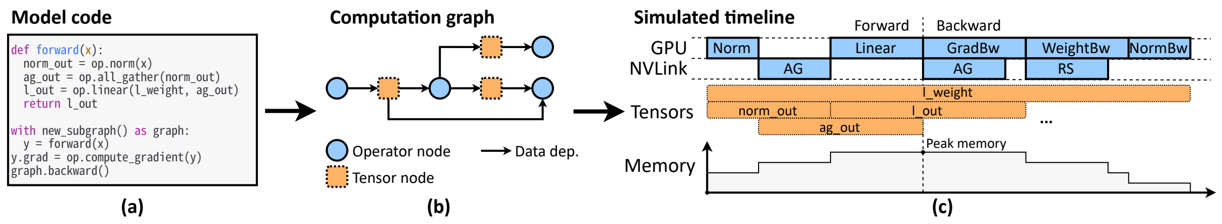


Figure 6. Workflow of PipeWeaver's pipeline schedule searcher. **(a)** Simulated pipeline stage flows of microbatches. **(b–d)** Three levels of search algorithms for optimizing pipeline schedules.

data dynamicity. Inaccurate estimation can result in suboptimal and invalid schedules (e.g., with out-of-memory errors). Existing systems like Alpa [58] employ coarse-grained heuristics, such as modeling 1F1B pipeline latency as $\sum_{i=1}^{n} t_i + m \cdot \max\{t_i\}$ (n stages, m microbatches), which assumes uniform microbatch latencies. These models fail under multimodal workloads, where heterogeneous data shapes and modality ratios induce dynamic computational demands.

Search Efficiency. Dynamic pipeline scheduling requires that search latencies are under typical training time (about 10–60 seconds, §8.1). However, the baseline ILP approach fails to meet these real-time constraints, taking over 50 minutes to optimize schedules [17]. This inefficiency stems from the exponentially growing search space of pipeline schedules, which increases with the scale of the training setup. For example, the number of possible schedules escalates from $32! \approx 2.6 \times 10^{35}$ for 32 microbatches to 1.3×10^{89} for 64 microbatches. Such exponential growth renders exhaustive optimization impractical for large-scale LMM training.


```
def forward(x):
  norm_out = op.norm(x)
  ag_out = op.all_gather(norm_out)
  l_out = op.linear(l_weight, ag_out)
  return l_out

with new_subgraph() as graph:
  y = forward(x)
y.grad = op.compute_gradient(y)
graph.backward()
```

Figure 7. An example simulation for a module with one `Norm` layer and one `Linear` layer in SEMU. **(a)** The simulation model code. **(b)** The computation graph consists of operator and tensor nodes. **(c)** The simulated timeline. The backward computation for `Linear` layer is represented by two operator nodes: gradient backward (`GradBw`) and weight backward (`WeightBw`).

3.3 Dynamic Interleaved Pipeline

We propose PipeWeaver, a flexible and dynamic scheduling framework for multimodal training pipelines. At the core of PipeWeaver is a training planner comprising three key components: multimodal training simulator (SEMU), modality-aware partitioner, and pipeline schedule searcher. As depicted in Fig.5, the framework operates through an iterative four-stage process:

1. **Metadata Prefetching:** The training planner proactively retrieves metadata for the next iteration's training microbatches, including text token counts, image quantities, and video durations.
2. **Adaptive Stage Partitioning:** Leveraging the prefetched metadata, the modality-aware partitioner divides modality modules and data microbatches and generates simulated pipeline executions on SEMU.
3. **Pipeline Schedule Searching:** The pipeline schedule searcher systematically evaluates potential pipeline schedules using SEMU and decides the execution order of all stages on each pipeline rank.
4. **Runtime Deployment:** The optimal schedule identified through this process is subsequently compiled into execution plans and deployed to the training runtime.

Both the modality-aware partitioner and schedule searcher rely on SEMU for accurate performance prediction of candidate schedules. This integrated approach enables PipeWeaver to efficiently navigate the complex optimization space of multimodal training.

Multimodal Training Simulator (SEMU, §4) employs directed acyclic computation graphs (DAGs) to represent complex LMM training workloads. During pipeline schedule searching, the simulator undergoes repeated invocations to update performance estimations of candidate pipeline configurations. This operational characteristic makes simulation efficiency a critical performance factor. To fulfill this requirement, we propose *spatial-temporal subgraph reuse* (§4.2) that identifies recurrent computational patterns by clustering adjacent operators (such as those within a pipeline stage) into reusable subgraphs. These encapsulated subgraphs can then be efficiently reused across different invocations, effectively reducing simulation complexity.

Modality-Aware Partitioner (§5) is responsible for splitting microbatches across different modality modules during training. Prior to the beginning of training, PipeWeaver partitions each modality module over all pipeline ranks by evenly distributing model layers to individual ranks. During training, the modality-aware partitioner further divides data microbatches into modality-specific sub-microbatches to balance computational loads among pipeline stages handling different modalities. The partitioner builds the simulated pipeline executions for subsequent schedule searching.

Pipeline Schedule Searcher (§6) identifies optimal pipeline schedules based on simulated execution traces from the modality-aware partitioner. As shown in Fig.6, it operates through a three-tiered hierarchical search mechanism. First, it establishes execution precedence among pipeline stages by assigning priority weights to modality modules within each microbatch (§6.1). Next, it heuristically interleaves forward and backward stages to minimize pipeline bubbles, while preserving predetermined priority relationships (§6.2). Finally, as data dynamicity introduces significant fluctuations in memory consumption, the schedule searcher further optimizes memory optimization configurations for all model layers to stabilize memory consumption patterns throughout training iterations (§6.3), enhancing resource utilization and training efficiency.

4 SEMU

4.1 Graph Representations

SEMU adopts operator-level analytical modeling that simulates the exeution of GPU kernels, avoiding the inefficient tracing or profiling methods used in most training simulators [19, 30, 49, 53]. This lightweight approach enables fast reconstruction of training processes, making it particularly suited for LMMs' highly dynamic workloads. SEMU decouples the simulation code from the actual training code (Fig.7a) and offers a PyTorch-like interface that simplifies architectural description of LMMs.

DAGs consist of two types of nodes: operator nodes representing low-level GPU operations (e.g., matrix multiplication and collective communication) and tensor nodes corresponding to data buffers such as model parameters and intermediate activation tensors (Fig.7b). Each node is assigned to a specific device (e.g., GPU, CPU, or NVLink) and connected via dependency edges. SEMU employs an analytical model for operator latency estimation. Each operator node is characterized by three metrics: floating-point operation count (N_{fop}), memory access volume (N_{mem}), and network data transfer size (N_{net}). Using the associated device's computational capacity (F in FLOPS), memory bandwidth (B_{mem} in bytes/s), and network throughput (B_{net} in bytes/s) [1], SEMU calculates the latency as the maximum value of N_{fop}/F, $N_{\text{mem}}/B_{\text{mem}}$, and $N_{\text{net}}/B_{\text{net}}$. This baseline estimation can be refined through efficiency scaling factors (α_{fop}, α_{mem}, and α_{net}) applied to the respective device capabilities. For specialized use cases, users may override the default cost model with custom latency estimation rules.

Start and end timestamps for all operator nodes are populated in topological order of the DAG. After that, SEMU determines the lifetime for each tensor node by examining all nodes that reference it. With these lifetime information, SEMU constructs memory consumption timelines (Fig.7c) to infer peak memory usages for all devices.

4.2 Spatial-Temporal Subgraph Reuse

SEMU will be invoked multiple times during pipeline schedule searching, making simulation efficiency critical for rapid schedule space exploration. We identify two key characteristics of LMM training workloads that enable performance enhancements in simulation. First, in the spatial dimension, repetitive patterns are prevalent in LMM training and can be consolidated into a single representative instance to eliminate redundant simulation. For example, GPU workers in tensor parallelism perform completely symmetric computations that can be folded into a unified simulation. Second, in the temporal dimension, intermediate simulation results can be cached and reused across consecutive simulator invocations to reduce overhead. This applies particularly to computational graphs within individual model layers or pipeline stages, where partial simulations remain consistent across iterations.

SEMU leverages spatial and temporal optimization opportunities by capturing repetitive patterns with *reusable subgraphs*. These subgraphs group neighboring operator nodes for caching and are reused across invocations. Users can encapsulate structural components of models, such as pipeline stages and model layers, into subgraphs to enable cross-invocation reusability. To accelerate simulation for symmetric workloads like tensor parallelism, subgraphs support replication that requires only a single simulation pass across duplicated instances. Additionally, subgraphs can be nested to accommodate the hierarchical architecture of LMMs, enabling flexible representation of computational graphs.

[1]SEMU unifies computing and communication devices by setting computing devices' network throughput to `nil` and vice versa. An operator nodes with non-zero N_{net} on a computing device will trigger a runtime error.

SEMU further offers a checkpoint-restore mechanism to facilitate subgraph reuse. Users can simulate specific portions of training workloads (e.g., individual pipeline microbatches), save simulation states into checkpoints, and resume the simulation with new dependency edges between subgraphs. SEMU automatically identifies unaffected subgraphs that remain intact after edge additions, reusing their results from the previous checkpoint. These reused subgraphs are consolidated into single nodes before continuing the simulation process.

5 Modality-Aware Partitioner

Prior to detailing the design of modality-aware partitioner, we first define several key concepts. A *modality module* refers to a specialized model component dedicated to processing and interpreting data from a single modality, with the backbone model itself being categorized as a special modality module as well. A *stage flow* is a set of interconnected pipeline stages operating on the same microbatch, which consists of multiple *pipeline segments*. Each segment comprises P (pipeline parallelism size) consecutive stages distributed across all pipeline ranks. These segments are further classified into forward and backward segments based on their computation types.

We begin by investigating how model chunk partitioning and microbatch construction affect pipeline stage scheduling through analysis of optimal schedules generated by the ILP algorithm described in §3.1. This examination reveals several insights that inform our pipeline scheduling design:

① **Modality-Aware Stage Segregation:** Model stages from different modalities should not reside within the same pipeline segment. As illustrated in Fig.8a and Fig.8b, multimodal pipelines can adopt two partitioning strategies: (1) *mixed partitioning*: multiple modality modules share a pipeline segment; (2) *separated partitioning*: each modality module occupies dedicated pipeline segments. Stages from distinct modalities exhibit divergent sensitivities to dynamicity of data modality distributions. For example, increasing the number of images in a microbatch substantially prolongs vision encoder stage execution, while minimally affecting backbone model stages. Consequently, colocating stages from different modalities within a single pipeline segment inevitably creates execution time disparities across pipeline ranks, inducing pipeline bubbles when processing dynamic multimodal data. Our results on the ViT 2B + LM 5B model (§2.2) show that even the optimal mixed partitioning schedule underperforms separated partitioning by 13.1%, validating the necessity of modality-aware stage segregation.

② **Modality-Aware Data Batching:** Sub-microbatch sizes specific to modality modules are preferred to balance pipeline stage latencies across modalities. For instance, given a backbone model processing a microbatch with B data samples and N_{img} images, the vision encoder can employ a smaller

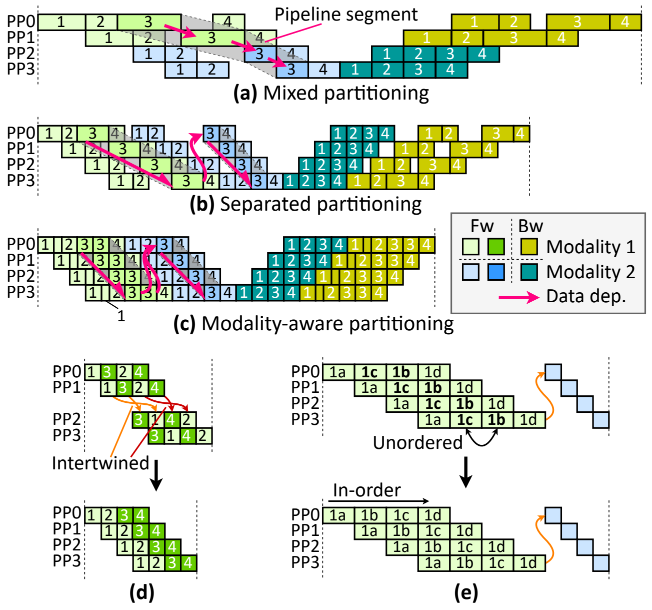


Figure 8. Examples of observations in §5 and §6.1. Numbers within boxes indicate different microbatches, while alphabetical characters distinguish sub-microbatches. **(a–b)** Examples for mixed and separated partitioning. **(c)** Modality-aware data batching further splits microbatches into sub-microbatches for each modality. **(d)** Example of ordering consistency, i.e., stage chains are not intertwined with each other. **(e)** Relative ordering inside a pipeline segment group does not impact pipeline efficiency.

sub-microbatch size of $N_{\text{img}}/2$. This strategy splits the encoder computation into two equally sized sub-microbatches, thereby distributing computational workloads to approximate the backbone model's execution time. Such modality-aware batching reduces inter-modality latency discrepancies, enhancing overall pipeline efficiency.

③ **Ordering Consistency:** Optimal pipeline schedules manifest a tendency of consistent relative stage ordering across pipeline ranks. For any two stages S_1 and S_2 of the same computation types (forward or backward), if S_1 precedes S_2 at the i-th pipeline rank, optimal scheduling consistently positions S_1 before S_2 at the $(i+1)$-th rank, as shown in Fig.8d. The primary motivation for this behavior stems from the scheduling inefficiency of inconsistent ordering that introduces extraneous pipeline bubbles.

Based on these observations, we propose *modality-aware partitioning*, which decomposes different modality modules into multiple pipeline segments at modality module level, and splits microbatches into modality-specific sub-microbatches at data level. Furthermore, to maintain the ordering consistency property, PipeWeaver ensures that any two pipeline segments belonging to the same computation type (forward or backward) do not intertwine with each other.

Modality Module Level. PipeWeaver determines two parameters for each modality module before training: the sub-microbatch size and the number of pipeline segments per sub-microbatch.

- **Modality-Specific Sub-Microbatch Size:** PipeWeaver independently selects the minimum viable sub-microbatch size B_i for the i-th modality module through profiling. While smaller sub-microbatch sizes enable finer-grained pipeline partitioning and improved scheduling efficiency, excessively small sizes risk GPU underutilization (Fig.10). To balance these factors, PipeWeaver measures the actual computation latency of each modality module under different sub-microbatch sizes. The optimal size is determined as the smallest sub-microbatch that maintains at least 95% of the maximum observed GPU computational efficiency across all tested sub-microbatch sizes.
- **Number of Pipeline Segments:** Following sub-microbatch size determination, PipeWeaver partitions each modality module's stage flow into pipeline segments to achieve global latency balancing across all modalities. Given n modality modules with profiled computation latencies $T_1, T_2, ..., T_n$ sorted in ascending order, PipeWeaver assigns pipeline segment counts proportional to these latencies. Specifically, the fastest module (T_1) forms one pipeline segment, and the i-th module is partitioned into $K_i = \lfloor T_i/T_1 \rfloor$ segments. Segments are sequentially connected within each modality module. For a modality module with L_i layers divided into K_i segments under pipeline parallelism degree P, PipeWeaver distributes layers across $P \cdot K_i$ model chunks, each of which contains $L_i/(P \cdot K_i)$ consecutive layers.

Data Level. PipeWeaver constructs sub-microbatches for each modality module based on the content of current training batches. For a microbatch assigned to the i-th modality module with batch size N_i, PipeWeaver splits it into $M_i = \lceil N_i/B_i \rceil$ evenly partitioned sub-microbatches. This partitioning generates $M_i \cdot K_i$ pipeline segments for the corresponding modality module. We define a *pipeline segment group* as the set of all pipeline segments derived from the same microbatch within a specific modality module. Finally, SEMU constructs simulated stage flows for all microbatches, which will be used by the pipeline schedule searcher.

6 Pipeline Schedule Searcher

6.1 Modality-Module-Level Ranking

PipeWeaver first determines the relative ordering between pipeline segments by assigning scheduling priority values. We observe that for any two pipeline segments from the same pipeline segment group, their relative order does not impact the end-to-end performance, as depicted in Fig.8e. Therefore we can assign the same priority value to segments within a pipeline segment group to reduce the search space.

To efficiently explore all possible priority assignments, PipeWeaver employs Monte-Carlo tree search (MCTS) [9].

Algorithm 1 Modality-Module-Level Ranking with MCTS

```
while not (time budget or search space exhausted) do
    $x \leftarrow$ root
    while $x$ is not a leaf node do                ▷ Node selection
        $x \leftarrow \text{argmax}_v\{s_v^\alpha + \beta\sqrt{(\log N_x)/N_v}\}$
    end while
    Expand a new node $u$ from $x$ and $x \leftarrow u$
    $\text{score}_{\max} \leftarrow 0$
    for $i \leftarrow 1$ to $N_{\text{tries}}$ do                ▷ Random rollout
        Generate random priority assignment $A$ from $x$
        Rollout $A$ using PipeWeaver's interleaving
        score $\leftarrow$ Percentage of non-bubble time
        $\text{score}_{\max} \leftarrow \max\{\text{score}_{\max}, \text{score}\}$
    end for
    repeat                ▷ Score backpropagation
        $s_x \leftarrow \max\{s_x, \text{score}_{\max}\}$
        $x \leftarrow x$.parent
    until $x$ is the tree root
end while
```

Search Tree Construction. Given n pipeline segment groups $G_1, G_2, ..., G_n$, MCTS aims to assign distinct priority values from $\{1, 2, ..., n\}$ to these groups. MCTS iteratively constructs a search tree where each node at depth d represents a priority assignment for G_d. Consequently, any path from the root to a depth-d node constitutes a partial priority assignment for the first d groups. Each node maintains the highest performance score observed among its descendant nodes, enabling efficient traversal during the search process.

Search Loop. The search process (Algorithm 1) operates through four phases:

1. **Node Selection:** Starting from the root x, MCTS navigates downward by selecting child nodes v with the highest upper confidence bound (UCB) score until reaching a leaf (line 2–5). The UCB score balances exploration and exploitation with $s_v^\alpha + \beta\sqrt{(\log N_x)/N_v}$, where N_x and N_v denote visit counts, and α, β are tunable hyperparameters.
2. **Tree Expansion:** A new node for the priority assignment to the next pipeline segment group is added (line 6).
3. **Random Rollout:** Multiple rollouts (e.g., 10 trials) assign random priorities to subsequent groups, generating the final pipeline segment rankings. Each ranking undergoes pipeline stage interleaving (§6.2) and model layer tuning (§6.3) to compute performance scores (line 10).
4. **Score Backpropagation:** The best rollout score propagates upward to update ancestor nodes (line 14–17).

The loop continues iterating until either the time budget is exhausted or the entire search space has been explored. Throughout the search process, dependencies between pipeline segments are enforced to eliminate invalid priority assignments, i.e., each segment must maintain a priority value smaller than that of its preceding segments.

6.2 Pipeline Stage Interleaving

Priority values determine the relative ordering of pipeline segments sharing the same computation type. PipeWeaver employs a heuristic dual-queue approach to further interleave forward and backward pipeline stages adaptively, constructing a compact pipeline schedule.

Initialization. For each pipeline rank, PipeWeaver maintains three components: (1) the end time of the last scheduled pipeline stage (denoted as t_{last}, initially zero); (2) two priority queues (Q_{fw}, Q_{bw}) storing forward and backward pipeline stages in descending order by priority values; and (3) the minimum start time ($t_{\min}$) of stages in Q_{fw} and Q_{bw}. Each stage is assigned a minimum schedulable start time (t_{start}), initialized to zero for stages without predecessors or infinity otherwise.

Iterative Scheduling. In each iteration, PipeWeaver selects a pipeline rank and schedules one stage from it. First, the pipeline rank with the smallest $t_{\min}$ is chosen. Second, we compare the minimum start times of stages in Q_{fw} (t_{fw}) and Q_{bw} (t_{bw}) against t_{last}. If both t_{fw} and t_{bw} are smaller than t_{last}, scheduling either type would avoid pipeline bubbles. In this case, we alternate between forward and backward stages based on the computation type of the last scheduled stage on the rank, emulating Megatron-LM's memory-efficient "one-forward-one-backward" pattern [26, 42]. Otherwise, the pipeline stage with a smaller t_{start} is chosen to minimize the pipeline bubble between $t_{\min}$ and the selected stage's start time.

The selected stage is dequeued and scheduled on the pipeline rank. Subsequently, t_{last}, $t_{\min}$, and t_{start} values for all successor stages are updated to reflect the new schedule.

Memory Constraints. Throughout scheduling, PipeWeaver tracks real-time memory consumption for all pipeline ranks and temporarily disables forward queues for pipeline ranks exceeding memory capacity to prevent memory overflow.

6.3 Model Layer Tuning

The dynamic nature of data in LMM training induces substantial memory usage fluctuations in conventional distributed training systems, which forces systems to reserve memory for worst-case scenarios, resulting in suboptimal memory utilization. Furthermore, memory allocation patterns are governed by tensor lifecycles: tensors generated in forward stages are subsequently consumed in backward stages. The intertwined lifecycles between these tensors preclude independent optimization of individual model layers. To address this issue, PipeWeaver dynamically selects appropriate memory optimization strategies (e.g., activation checkpointing and offloading) across all model layers. Our optimization aims to minimize end-to-end training time while maintaining memory consumption within a predefined capacity.

Candidate Generation. For each model layer, PipeWeaver enumerates all possible strategies and estimates their exe-

Action Type	Description
`forward_stage`	Execute forward pipeline stage.
`backward_stage`	Execute backward pipeline stage.
`isend`	Launch P2P send kernel.
`wait_isend`	Wait for P2P send kernel to complete and release the sending buffer.
`irecv`	Launch P2P receive kernel.
`wait_irecv`	Wait for P2P receive kernel to complete.

Table 2. Types of actions in execution plans defined by PipeWeaver.

cution time and memory consumption using SEMU. Next, PipeWeaver treats each forward stage and its corresponding backward stage as a *stage pair*, and strategically selects up to K optimization strategy candidates for each stage pair from numerous strategy combinations of model layers in this stage pair. PipeWeaver first identifies the fastest and the most memory-efficient candidates. Then it divides the memory usage range between these two extremes into $K-2$ buckets and searches for the most time-efficient candidate within each bucket through a multiple-choice knapsack algorithm [2].

ILP Formulation. Following pipeline stage interleaving, PipeWeaver employs an ILP solver [45] to identify optimal scheduling candidates. Given n stage pairs, each with K candidate strategies, the i-th stage pair's start and end timestamps are s_i and t_i. For the j-th candidate strategy of this stage pair, let $\text{lat}_{i,j}$ denote its latency and $\text{mem}_{i,j}$ represent its memory consumption. PipeWeaver creates nK binary variables $o_{i,j} \in \{0,1\}$ to indicate whether the i-th stage pair selects its j-th candidate. The ILP is formulated as the following optimization problem:

- **Goal:** Minimize total latency $\sum_i^n \sum_j^K o_{i,j} \cdot \text{lat}_{i,j}$.
- **Selection Constraints:** Each stage pair selects one scheme, i.e., $\sum_j^K o_{i,j} = 1$ ($\forall 1 \le i \le n$).
- **Memory Constraints:** At any time s_k, the memory limit M is satisfied, i.e., $\sum_i^n [s_i \le s_k \le t_i] \sum_j^K o_{i,j} \cdot \text{mem}_{i,j} \le M$ ($\forall 1 \le k \le n$), where $[\cdot]$ is the Iverson bracket [24].

Optimizations. Even though the problem scale has been reduced significantly compared to the baseline ILP approach (§3.1), solving a single ILP instance still requires several seconds. We propose two optimizations to achieve efficient ILP solving (<10 milliseconds on a single CPU core). First, we initialize $o_{i,j}$ with a greedy solution to warm up the ILP solving. Second, we allow a small optimality gap (e.g., ≤5%) for early termination, as closing the final 5% gap incurs diminishing returns but prohibitive computational costs.

7 Implementation

PipeWeaver is implemented on top of Megatron-LM [42], the state-of-the-art training framework for transformer-based large language models. PipeWeaver features a central controller that interacts with Megatron-LM's core runtime. The controller iteratively prefetches the metadata, performs pipeline schedule searching, and deploys the generated schedules.

7.1 Metadata Prefetching

PipeWeaver employs pinned host memory buffers [39] in DRAM to prefetch microbatches for upcoming training iterations via a wrapped dataloader. Training metadata, such as the number of text tokens and images, is inferred from the prefetched tensor shapes. To hold both training data of the current and the next iterations, PipeWeaver uses a double buffering method that preallocates two sets of data buffers. Upon completing an iteration, PipeWeaver swaps data buffers, enabling seamless reuse of prefetched data for subsequent steps while maintaining pipeline continuity.

7.2 Pipeline Schedule Searching

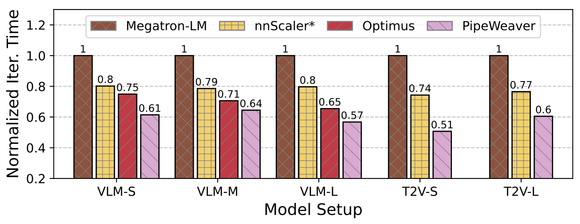
PipeWeaver runs SEMU and search algorithms on CPUs. PipeWeaver first simulates computation graphs of microbatches individually based on the prefetched training metadata, and then checkpoints simulation states for subsequent search phases. During pipeline schedule search, multiple search workers are spawned to explore the schedule space in parallel. Search workers share the global MCTS search statistics and operate on replicated copies of the simulation state. After each search round, global MCTS statistics are atomically updated via a mutex-protected operation and then worker-local states are restored to the checkpoint. Contention for the MCTS mutex remains minimal, as workers perform multiple policy rollouts between synchronizations, thereby amortizing synchronization overhead. To avoid interference with training processes, PipeWeaver allocates no more than 50% of available CPU cores to search algorithms (e.g., 64 out of 128 cores on a machine with 8 H800 GPUs).

7.3 Execution Plan Deployment

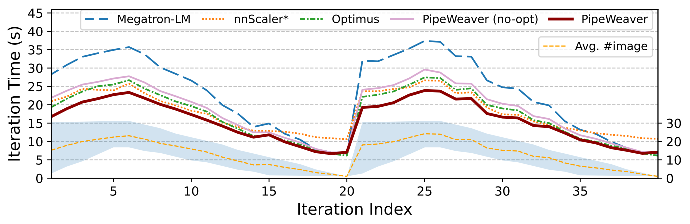
To deploy a simulated pipeline schedule to real GPU clusters, we need to convert it into a physical execution plan that describes computation and communication patterns.

Schedule Compilation. PipeWeaver defines a set of actions that constitute a pipeline execution plan, as listed in Table 2. Pipeline stages are translated to `forward_stage` or `backward_stage` according to stage types, with optimization strategies determined by model layer tuning. For P2P communications, PipeWeaver seeks to overlap them with stage computations through the usage of asynchronous communication kernels (`isend`, `irecv`). PipeWeaver inserts P2P launch and wait actions (`wait_isend`, `wait_irecv`) according to P2P operation's start and end timestamps in the simulated timeline, which is similar to DynaPipe [26].

Runtime Modifications. We implement a new pipeline schedule scheme in Megatron-LM's `schedules` module. Each pipeline worker receives a list of actions from the central controller via RPC communications and executes these actions in order. We group consecutive P2P kernels into a batched operation [38] to enhance communication performance.



(a) Average performance



(b) Dynamic workloads

Figure 9. End-to-end performance. **(a)** Average performance of 100 iteration on real datasets across five model setups. **(b)** End-to-end latencies of 40 consecutive iterations. The orange dashed line depicts the average number of images.

8 Evaluation

Models. We conduct comprehensive evaluations of PipeWeaver across two major LMM architectures: vision-language models (VLMs) and text-to-video models (T2V). The VLM implementations integrate ViT-based image encoders (5B and 22B parameters) [12, 13] with language model backbones (Llama3 8B [33], Qwen2 32B/72B [55]), while T2V architectures combine language model encoders with DiT-based diffusion video decoders (5B and 30B parameters) [37]. We choose five distinct model combinations ranging from 12B to 94B parameters, as detailed in Table 3 and Table 4.

Datasets. We employ a combination of diverse open-source datasets [5, 7, 27, 31, 41, 50], comprising pure image-text pairs, interleaved image-text documents, and video-caption pairs. For VLM models, we scale all images to 768px resolution, with each image being encoded by the ViT vision encoder into 169 patch tokens. Multiple image-text data samples are packed into sequences of 8192 tokens, resulting in a maximum image capacity of $\lfloor 8192/169 \rfloor = 48$ per sequence. For T2V models, we adopt MovieGen's [37] configuration by transcoding videos to 16 FPS with a maximum duration of 16 seconds. When processing short videos, we group up to 8 video clips per microbatch while maintaining the total microbatch duration below 16 seconds.

Baselines. We benchmark PipeWeaver against three state-of-the-art baseline systems:

- **Megatron-LM** [42] is a widely-used unimodal LLM training framework. We use interleaved pipeline parallelism (VPP) and partition LMM layers into model chunks with approximately balanced parameter distribution.
- **nnScaler** [29] automates parallelization for deep neural network training. Since generating a single parallelization plan takes several minutes and requires restarting the training process for plan updates, We pre-generate a static parallelization plan before training with a representative training workload. For fair cross-framework comparison, we implement nnScaler's model chunk partitioning schemes and layer memory optimizations in Megatron-LM, with performance metrics labeled as "nnScaler*".
- **Optimus** [15] proposes coarse-grained and fine-grained bubble scheduling to optimize multimodal LLM (MLLM) training with multiple encoders. The coarse-grained strategy sequences all modality encoder computations before backbone model execution at the pipeline level, while the fine-grained method fills TP communication bubbles with encoder computations and is orthogonal to our pipeline design. For focused pipeline scheduling comparisons, we implement Optimus' coarse-grained bubble scheduling in PipeWeaver. Due to the lack of support for diffusion decoders, we exclude Optimus from T2V model evaluations.

Name	# of Layers	Embed Dim	FFN Hidden Dim	# of Attn. Heads	# of Attn. Groups
ViT 5B [12]	63	1792	15360	16	16
ViT 22B [12]	48	6144	24576	48	48
Llama3 8B [33]	32	4096	14336	32	8
Qwen2 32B [56]	64	5120	27648	40	8
Qwen2 72B [56]	80	8192	29568	64	8
DiT 5B [37]	28	3584	10240	28	28
DiT 30B [37]	48	6144	24576	48	48

Table 3. Model specifications used in the evaluations.

Name	Model Setup	TP	PP	#GPU
VLM-S	ViT 5B + Llama3 8B	4	4	16
VLM-M	ViT 5B + Qwen2 32B	8	4	32
VLM-L	ViT 22B + Qwen2 72B	8	8	64
T2V-S	Llama3 8B + DiT 5B	4	4	16
T2V-L	Qwen2 32B + DiT 30B	8	8	64

Table 4. Model combinations used in the evaluation.

Testbed. All real-world experiments were conducted on a cluster of 64 H800 GPUs, comprising 8 machines. Each machine is equipped with 128 CPUs, 256GB host memory, and 8 H800 GPUs interconnected via 200 GB/s NVLink. Machines are connected by 8×200Gbps RoCEv2 network based on rail-optimized topology.

Metrics. We adopt training iteration time and model FLOPs utilization (MFU) as performance metrics, with all reported values averaged across ten independent runs.

8.1 End-to-End Performance

We first conduct experiments comparing the average end-to-end performance of PipeWeaver against baseline systems us-

Techniques	Iter. Time (s)	Δ%
Vanilla Megatron-LM	26.13	0.0%
+ Modality-aware partitioner (§5)	22.27	17.3%
+ Pipeline stage interleaving (§6.2)	18.81	38.9%
+ Modality-module-level ranking (§6.1)	17.61	48.3%
+ Model layer tuning (§6.3)	16.05	62.8%

Table 5. Quantitative impact of PipeWeaver's optimizations.

ing real datasets and five model configurations summarized in Table 4 As demonstrated in Fig.9a, PipeWeaver achieves training throughput improvements of 15.6%–76.2% over three baseline systems in VLM model setups, and 36.6%–97.3% over two baselines in T2V model configurations, demonstrating consistent performance gains across diverse model architectures and parameter scales.

To analyze PipeWeaver's performance characteristics under dynamic workloads against other baselines, we investigate the VLM-S model with manual control of image quantity bounds during training iterations. We monitor 40 iterations showing two "rise-and-fall" patterns in image counts. Each pattern consists of: (1) gradually increasing the lower bound from 0 to 16 while maintaining an upper bound of 32 (iterations 1–5), achieving a peak average of 22 images, followed by (2) progressively reducing both bounds to zero (iterations 6–20).

Fig.9b reveals PipeWeaver's consistent superior performance across all systems. During high-image-count phases (iterations 1–10), Megatron-LM suffers significant computational imbalance across modality modules and data microbatches, exhibiting a 52.9% slowdown compared to PipeWeaver at iteration 6. Although both nnScaler and Optimus partially mitigate dynamic imbalance effects, they are still 10.4% slower than PipeWeaver. As both image counts and bound gaps decrease (iterations 11–20), training workloads converge toward pure language tasks, narrowing the performance gap between PipeWeaver and baseline systems. Notably, nnScaler's restriction to 1F1B scheduling mandates all modality modules to be partitioned inside one pipeline segment, which creates significant pipeline imbalance when image encoder workloads diminish, resulting in 50.5% performance degradation during iterations 15–20.

8.2 Performance Ablation

Performance Breakdown. Using the VLM-S model setup, we incrementally integrate four key components (modality-aware partitioner, pipeline stage interleaving, modality-module-level ranking, and model layer tuning) onto Megatron-LM. As demonstrated in Table 5, the modality-aware partitioner alone delivers a 17.3% performance improvement over the baseline Megatron-LM, highlighting its critical role in LMM training. Among optimizations of pipeline schedule searcher, pipeline stage interleaving provides a substantial 21.6% performance boost compared to Megatron-LM's default pipelining. Subsequent modality-module-level ranking and model layer tuning enhance performance by an additional 23.9% through intelligent reordering of pipeline segments and adaptive memory optimization strategy selection. In Fig.9b, we visually contrast improvements from modality-aware partitioner and pipeline schedule searcher over dynamic workloads, separated by the line labeled "PipeWeaver (no-opt)" that excludes pipeline schedule searcher's optimizations.

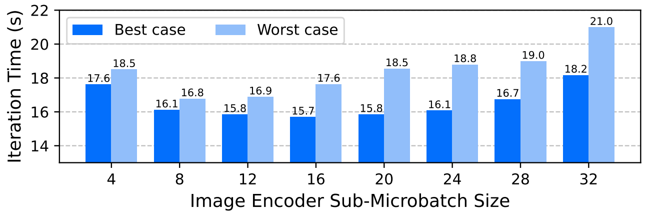


Figure 10. Impact of sub-microbatch sizes.

Impact of Sub-Microbatch Sizes. We investigate the influence of modality-specific sub-microbatch sizes on pipeline scheduling and GPU execution efficiency using the VLM-S model, by testing image sub-microbatch sizes ranging from 4 to 32 and deriving the best and the worst[2] pipeline schedules.

Our analysis of Fig.10 reveals two key findings. First, smaller sub-microbatch sizes (4–12) significantly reduce the performance gap between best and worst schedules from 15.4% to 5.1%. This narrowed variance indicates reduced sensitivity to schedule configurations, thereby lowering the difficulty in achieving optimal pipeline schedules. Second, extremely small sub-microbatch sizes (<8) demonstrate diminishing returns due to underutilized GPU computational capacity, resulting in a 12.1% increase in iteration time compared to medium-sized batches. Through empirical validation, we identify a sub-microbatch size of 12 as the optimal balance point between pipeline schedule flexibility and hardware utilization efficiency.

Impact of Model Layer Tuning. We analyze the memory usage timeline of the first pipeline rank during VLM-M model training. As shown in Fig.11, baseline systems fail to fully utilize available GPU memory. Megatron-LM exhibits significant memory fluctuations during the steady 1F1B pipeline phase, while Optimus manifests a gradual memory increase due to executing all modality encoder computations and storing substantial activation memory prior to the backbone model. This results in 25.3% higher peak memory consumption compared to Megatron-LM. In contrast, PipeWeaver maintains consistently low memory usage throughout training iterations by partitioning microbatches into smaller modality-specific sub-microbatches, enabling finer-grained interleaving of forward and backward pipeline stages. Model layer tuning further intelligently selects memory optimization configurations to achieve full utilization of available GPU memory, achieving 52.9% fewer memory

[2]We obtain the worst pipeline schedule by inverting the optimization goal of modality-module-level ranking to maximizing iteration time.

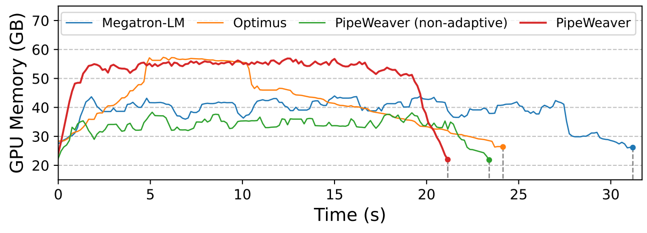


Figure 11. Memory usage timelines of the first pipeline rank in VLM-M training. The "PipeWeaver (non-adaptive)" variant disables model layer tuning and does not seek to utilize all available GPU memory.

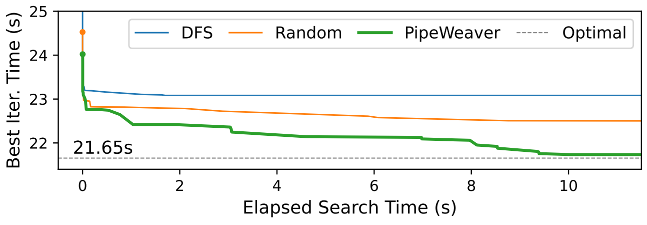


Figure 12. Search progresses of different exploration strategies.

fluctuations than Megatron-LM and delivering 12.2% performance gains over Optimus under equivalent peak memory conditions.

8.3 Planner Evaluation

Search Efficiency. To demonstrate the efficiency of PipeWeaver's pipeline schedule search, we track the progression of current best pipeline schedule performance versus elapsed search time, comparing it with two variants using depth-first search (DFS) and random exploration instead of the MCTS algorithm utilized in modality-module-level ranking. Using the largest VLM-L model as the target workload, we run all search algorithm variants on 64 CPU cores. Fig.12 shows PipeWeaver achieves near-optimal pipeline schedule performance within 10 seconds, which can be overlapped with VLM-L's typical 20-second training iteration duration. In contrast, DFS and random exploration strategies fail to quickly identify optimal pipeline schedules due to their lack of score-guided search optimization like MCTS.

Simulation Accuracy. We assess the accuracy of SEMU through a grid-search experiment for VLM-M across 64 GPUs, and compare simulated results against actual GPU executions. We systematically evaluate all valid combinations of DP, PP, and TP sizes where all values are powers of two and TP ≤ 8. As illustrated in Fig.13, the optimal parallelism configuration is DP8, TP2, and PP4, achieving a MFU of 29.7%. Although PipeWeaver's default simulation settings initially exhibit relative errors up to 10% compared to ground-truth measurements, SEMU still successfully predicts the optimal parallel configuration. Through calibration via offline microbenchmarks that align efficiency scale factors for matrix multiplications and collective communication operations (§4.1), SEMU achieves an average simulation accuracy of 97.6%.

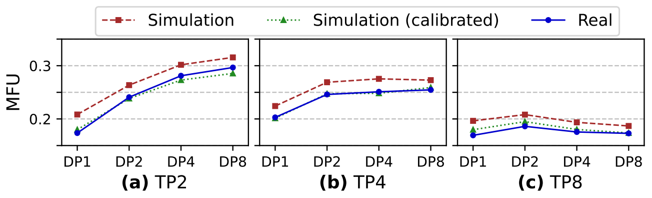


Figure 13. Comparison between pre- and post-calibration simulation results versus actual GPU executions.

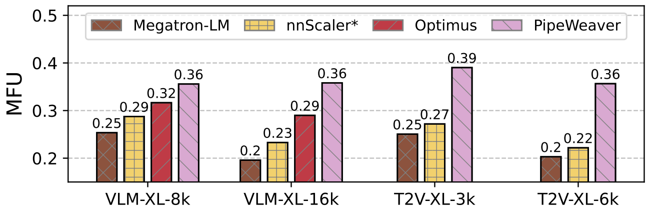


Figure 14. Large scale simulations on H100 clusters.

Name	Model Setup	DP	TP	PP	#GPU
VLM-XL-8k	ViT 22B + GPT 175B	128	8	8	8192
VLM-XL-16k	ViT 22B + GPT 175B	128	8	16	16384
T2V-XL-3k	Qwen2 72B + DiT 30B	96	8	4	3072
T2V-XL-6k	Qwen2 72B + DiT 30B	96	8	8	6144

Table 6. Model combinations used in large-scale simulations.

8.4 Large-Scale Simulation

To validate PipeWeaver's effectiveness in large-scale H100 GPU cluster environments, we conducted two experiments using SEMU with substantial large multimodal models (VLM-XL and T2V-XL) to evaluate its theoretical improvements over baseline approaches. Detailed configurations for both models and GPU clusters are presented in Table 6.

Results. Experimental results in Fig.14 demonstrate that PipeWeaver achieves remarkable MFU scores of 0.36 for VLM-XL and 0.39 for T2V-XL. The system consistently outperforms baseline methods by up to 82.8%, particularly with larger pipeline parallelism sizes, as larger PP dimensions introduce more complex pipeline structures that require meticulous orchestration between stages by the pipeline schedule searcher.

9 Related Work

Automated Parallelization. Several systems enable automated parallelization for training deep learning models. Notable examples include Alpa [58], nnScaler [29] and Unity [47]. These systems perform full-scale model planning across DP, PP, and TP dimensions. While they employ exhaustive search algorithms to generate near-optimal training configurations, the planning process often requires several minutes to complete [29]. This substantial time overhead renders such approaches impractical for LMM training scenarios that demands dynamic replanning.

Training Simulators. Recent research has explored simulator-based approaches to optimize DNN training without requiring physical GPU clusters [14, 16, 18, 19, 30, 49, 53, 60]. However, most existing simulators prioritize simulation accuracy over efficiency. For instance, SimAI [49] employs the event-driven ns3 network simulator [34] for precise network modeling, but this comes at significant computational cost: simulating a 128-GPU workload requires over two hours [16]. This efficiency hinders their employment in dynamic planning of LMM training. In contrast, SEMU employs spatial-temporal subgraph reuse to effectively resolve this issue while maintaining simulation accuracy.

Training Systems for Multimodal Models. Many system optimizations [15, 21, 25, 46, 51, 57, 59] have been developed to address the architectural and training data heterogeneity inherent in multimodal model training. These approaches primarily focus on resolving data imbalance across DP groups, coordinating heterogeneous modality modules, and optimizing job scheduling for diverse training workloads. While effective in their respective domains, these optimization techniques remain largely complementary to PipeWeaver's dynamic pipeline mechanism.

10 Conclusion

Efficient large multimodal model training is challenging due to pipeline stage imbalance and training data dynamicity. In this paper, we propose PipeWeaver to address dynamic imbalance of LMM training with adaptive modality-aware partitioning and efficient pipeline schedule search. Our experimental results demonstrate that PipeWeaver achieves remarkable performance improvements, outperforming existing state-of-the-art training systems by up to 97.3% in training throughput. PipeWeaver has been used in many of our internal production workload, and we are planning to open-source the code soon.

References

[1] Anthropic. The Claude 3 model family: Opus, Sonnet, Haiku. https://www.anthropic.com/news/claude-3-family, 2024.

[2] R. D. Armstrong, D. S. Kung, P. Sinha, and A. A. Zoltners. A computational study of a multiple-choice knapsack algorithm. *ACM Trans. Math. Softw.*, 9(2):184–198, June 1983.

[3] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-VL technical report. arXiv, 2025.

[4] Minwoo Byeon, Beomhee Park, Haecheon Kim, Sungjun Lee, Woonhyuk Baek, and Saehoon Kim. COYO-700M: Image-text pair dataset. https://github.com/kakaobrain/coyo-dataset, 2022.

[5] Lin Chen, Xilin Wei, Jinsong Li, Xiaoyi Dong, Pan Zhang, Yuhang Zang, Zehui Chen, Haodong Duan, Bin Lin, Zhenyu Tang, Li Yuan, Yu Qiao, Dahua Lin, Feng Zhao, and Jiaqi Wang. ShareGPT4Video: Improving video understanding and generation with better captions. arXiv, 2024.

[6] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[7] Xiaowei Chi, Yatian Wang, Aosong Cheng, Pengjun Fang, Zeyue Tian, Yingqing He, Zhaoyang Liu, Xingqun Qi, Jiahao Pan, Rongyu Zhang, Mengfei Li, Ruibin Yuan, Yanbing Jiang, Wei Xue, Wenhan Luo, Qifeng Chen, Shanghang Zhang, Qifeng Liu, and Yike Guo. MMTrail: A multimodal trailer video dataset with language and music descriptions. arXiv, 2024.

[8] Yew Ken Chia, Liying Cheng, Hou Pong Chan, Chaoqun Liu, Maojia Song, Sharifah Mahani Aljunied, Soujanya Poria, and Lidong Bing. M-Longdoc: A benchmark for multimodal super-long document understanding and a retrieval-aware tuning framework. arXiv, 2024.

[9] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games*, 2007.

[10] DeepMind. Gemma 3. https://blog.google/technology/developers/gemma-3/, 2025.

[11] DeepSeek-AI. DeepSeek-V3 technical report. arXiv, 2025.

[12] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, Rodolphe Jenatton, Lucas Beyer, Michael Tschannen, Anurag Arnab, Xiao Wang, Carlos Riquelme, Matthias Minderer, Joan Puigcerver, Utku Evci, Manoj Kumar, Sjoerd van Steenkiste, Gamaleldin F. Elsayed, Aravindh Mahendran, Fisher Yu, Avital Oliver, Fantine Huot, Jasmijn Bastings, Mark Patrick Collier, Alexey Gritsenko, Vighnesh Birodkar, Cristina Vasconcelos, Yi Tay, Thomas Mensink, Alexander Kolesnikov, Filip Pavetić, Dustin Tran, Thomas Kipf, Mario Lučić, Xiaohua Zhai, Daniel Keysers, Jeremiah Harmsen, and Neil Houlsby. Scaling vision transformers to 22 billion parameters. arXiv, 2023.

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[14] Jiangfei Duan, Xiuhong Li, Ping Xu, Xingcheng Zhang, Shengen Yan, Yun Liang, and Dahua Lin. Proteus: Simulating the performance of distributed DNN training. In *IEEE Transactions on Parallel & Distributed Systems*, 2024.

[15] Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. Optimus: Accelerating large-scale multi-modal LLM training by bubble exploitation. arXiv, 2024.

[16] Yicheng Feng, Yuetao Chen, Kaiwen Chen, Jingzong Li, Tianyuan Wu, Peng Cheng, Chuan Wu, Wei Wang, Tsung-Yi Ho, and Hong Xu. Echo: Simulating distributed training at scale. arXiv, 2024.

[17] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. ReCycle: Resilient training of large DNNs using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 211–228, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Fei Gui, Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Ran Zhang, Hongbing Yang, and Dian Xiong. Accelerating design space exploration for LLM training systems with multi-experiment parallel simulation. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025.

[19] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. dPRO: A generic performance diagnosis and optimization toolkit for expediting distributed DNN training. In *Proceedings of Machine Learning and Systems*, 2022.

[20] Ailin Huang, Boyong Wu, Bruce Wang, Chao Yan, Chen Hu, Chengli Feng, Fei Tian, Feiyu Shen, Jingbei Li, Mingrui Chen, Peng Liu, Ruihang Miao, Wang You, Xi Chen, Xuerui Yang, Yechang Huang, Yuxiang Zhang, Zheng Gong, Zixin Zhang, Hongyu Zhou, Jianjian Sun, Brian Li, Chengting Feng, Changyi Wan, Hanpeng Hu, Jianchang Wu,

Jiangjie Zhen, Ranchen Ming, Song Yuan, Xuelin Zhang, Yu Zhou, Bingxin Li, Buyun Ma, Hongyuan Wang, Kang An, Wei Ji, Wen Li, Xuan Wen, Xiangwen Kong, Yuankai Ma, Yuanwei Liang, Yun Mou, Bahtiyar Ahmidi, Bin Wang, Bo Li, Changxin Miao, Chen Xu, Chenrun Wang, Dapeng Shi, Deshan Sun, Dingyuan Hu, Dula Sai, Enle Liu, Guanzhe Huang, Gulin Yan, Heng Wang, Haonan Jia, Haoyang Zhang, Jiahao Gong, Junjing Guo, Jiashuai Liu, Jiahong Liu, Jie Feng, Jie Wu, Jiaoren Wu, Jie Yang, Jinguo Wang, Jingyang Zhang, Junzhe Lin, Kaixiang Li, Lei Xia, Li Zhou, Liang Zhao, Longlong Gu, Mei Chen, Menglin Wu, Ming Li, Mingxiao Li, Mingliang Li, Mingyao Liang, Na Wang, Nie Hao, Qiling Wu, Qinyuan Tan, Ran Sun, Shuai Shuai, Shaoliang Pang, Shiliang Yang, Shuli Gao, Shanshan Yuan, Siqi Liu, Shihong Deng, Shilei Jiang, Sitong Liu, Tiancheng Cao, Tianyu Wang, Wenjin Deng, Wuxun Xie, Weipeng Ming, Wenqing He, Wen Sun, Xin Han, Xin Huang, Xiaomin Deng, Xiaojia Liu, Xin Wu, Xu Zhao, Yanan Wei, Yanbo Yu, Yang Cao, Yangguang Li, Yangzhen Ma, Yanming Xu, Yaoyu Wang, Yaqiang Shi, Yilei Wang, Yizhuang Zhou, Yinmin Zhong, Yang Zhang, Yaoben Wei, Yu Luo, Yuanwei Lu, Yuhe Yin, Yuchu Luo, Yuanhao Ding, Yuting Yan, Yaqi Dai, Yuxiang Yang, Zhe Xie, Zheng Ge, Zheng Sun, Zhewei Huang, Zhichao Chang, Zhisheng Guan, Zidong Yang, Zili Zhang, Binxing Jiao, Daxin Jiang, Heung-Yeung Shum, Jiansheng Chen, Jing Li, Shuchang Zhou, Xiangyu Zhang, Xinhao Zhang, and Yibo Zhu. Step-Audio: Unified understanding and generation in intelligent speech interaction. arXiv, 2025.

[21] Jun Huang, Zhen Zhang, Shuai Zheng, Feng Qin, and Yida Wang. DISTMM: Accelerating distributed multimodal model training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1157–1171, Santa Clara, CA, April 2024. USENIX Association.

[22] Minbin Huang, Yanxin Long, Xinchi Deng, Ruihang Chu, Jiangfeng Xiong, Xiaodan Liang, Hong Cheng, Qinglin Lu, and Wei Liu. DialogGen: Multi-modal interactive dialogue system for multi-turn text-to-image generation. arXiv, 2024.

[23] Q. Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. arXiv, 2015.

[24] Kenneth E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), page 345–351, New York, NY, USA, 1962. Association for Computing Machinery.

[25] Insu Jang, Runyu Lu, Nikhil Bansal, Ang Chen, and Mosharaf Chowdhury. Cornstarch: Distributed multimodal training must be multimodality-aware. arXiv, 2025.

[26] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. DynaPipe: Optimizing multi-task training through dynamic pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 542–559, New York, NY, USA, 2024. Association for Computing Machinery.

[27] Hugo Laurençon, Lucile Saulnier, Léo Tronchon, Stas Bekman, Amanpreet Singh, Anton Lozhkov, Thomas Wang, Siddharth Karamcheti, Alexander M. Rush, Douwe Kiela, Matthieu Cord, and Victor Sanh. OBELICS: An open web-scale filtered dataset of interleaved image-text documents. arXiv, 2023.

[28] Zhimin Li, Jianwei Zhang, Qin Lin, Jiangfeng Xiong, Yanxin Long, Xinchi Deng, Yingfang Zhang, Xingchao Liu, Minbin Huang, Zedong Xiao, Dayou Chen, Jiajun He, Jiahao Li, Wenyue Li, Chen Zhang, Rongwei Quan, Jianxiang Lu, Jiabin Huang, Xiaoyan Yuan, Xiaoxiao Zheng, Yixuan Li, Jihong Zhang, Chao Zhang, Meng Chen, Jie Liu, Zheng Fang, Weiyan Wang, Jinbao Xue, Yangyu Tao, Jianchen Zhu, Kai Liu, Sihuan Lin, Yifu Sun, Yun Li, Dongdong Wang, Mingtao Chen, Zhichao Hu, Xiao Xiao, Yan Chen, Yuhong Liu, Wei Liu, Di Wang, Yong Yang, Jie Jiang, and Qinglin Lu. Hunyuan-DiT: A powerful multi-resolution diffusion transformer with fine-grained chinese understanding. arXiv, 2024.

[29] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[30] Guandong Lu, Runzhe Chen, Yakai Wang, Yangjie Zhou, Rui Zhang, Zheng Hu, Yanming Miao, Zhifang Cai, Li Li, Jingwen Leng, and Minyi Guo. DistSim: A performance model of large-scale hybrid distributed DNN training. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, 2023.

[31] Pan Lu, Swaroop Mishra, Tony Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. Learn to explain: Multimodal reasoning via thought chains for science question answering. In *The 36th Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[32] Guoqing Ma, Haoyang Huang, Kun Yan, Liangyu Chen, Nan Duan, Shengming Yin, Changyi Wan, Ranchen Ming, Xiaoniu Song, Xing Chen, Yu Zhou, Deshan Sun, Deyu Zhou, Jian Zhou, Kaijun Tan, Kang An, Mei Chen, Wei Ji, Qiling Wu, Wen Sun, Xin Han, Yanan Wei, Zheng Ge, Aojie Li, Bin Wang, Bizhu Huang, Bo Wang, Brian Li, Changxing Miao, Chen Xu, Chenfei Wu, Chenguang Yu, Dapeng Shi, Dingyuan Hu, Enle Liu, Gang Yu, Ge Yang, Guanzhe Huang, Gulin Yan, Haiyang Feng, Hao Nie, Haonan Jia, Hanpeng Hu, Hanqi Chen, Haolong Yan, Heng Wang, Hongcheng Guo, Huilin Xiong, Huixin Xiong, Jiahao Gong, Jianchang Wu, Jiaoren Wu, Jie Wu, Jie Yang, Jiashuai Liu, Jiashuo Li, Jingyang Zhang, Junjing Guo, Junzhe Lin, Kaixiang Li, Lei Liu, Lei Xia, Liang Zhao, Liguo Tan, Liwen Huang, Liying Shi, Ming Li, Mingliang Li, Muhua Cheng, Na Wang, Qiaohui Chen, Qinglin He, Qiuyan Liang, Quan Sun, Ran Sun, Rui Wang, Shaoliang Pang, Shiliang Yang, Sitong Liu, Siqi Liu, Shuli Gao, Tiancheng Cao, Tianyu Wang, Weipeng Ming, Wenqing He, Xu Zhao, Xuelin Zhang, Xianfang Zeng, Xiaojia Liu, Xuan Yang, Yaqi Dai, Yanbo Yu, Yang Li, Yineng Deng, Yingming Wang, Yilei Wang, Yuanwei Lu, Yu Chen, Yu Luo, Yuchu Luo, Yuhe Yin, Yuheng Feng, Yuxiang Yang, Zecheng Tang, Zekai Zhang, Zidong Yang, Binxing Jiao, Jiansheng Chen, Jing Li, Shuchang Zhou, Xiangyu Zhang, Xinhao Zhang, Yibo Zhu, Heung-Yeung Shum, and Daxin Jiang. Step-Video-T2V technical report: The practice, challenges, and future of video foundation model. arXiv, 2025.

[33] Meta. The Llama 3 herd of models. arXiv, 2024.

[34] ns3 maintainers. ns3: A discrete-event network simulator for internet systems. https://www.nsnam.org.

[35] OpenAI. Introducing 4o image generation. https://openai.com/index/introducing-4o-image-generation/.

[36] OpenAI. GPT-4o system card. https://openai.com/index/gpt-4o-system-card/, 2024.

[37] Adam Polyak, Amit Zohar, Andrew Brown, Andros Tjandra, Animesh Sinha, Ann Lee, Apoorv Vyas, Bowen Shi, Chih-Yao Ma, Ching-Yao Chuang, David Yan, Dhruv Choudhary, Dingkang Wang, Geet Sethi, Guan Pang, Haoyu Ma, Ishan Misra, Ji Hou, Jialiang Wang, Kiran Jagadeesh, Kunpeng Li, Luxin Zhang, Mannat Singh, Mary Williamson, Matt Le, Matthew Yu, Mitesh Kumar Singh, Peizhao Zhang, Peter Vajda, Quentin Duval, Rohit Girdhar, Roshan Sumbaly, Sai Saketh Rambhatla, Sam Tsai, Samaneh Azadi, Samyak Datta, Sanyuan Chen, Sean Bell, Sharadh Ramaswamy, Shelly Sheynin, Siddharth Bhattacharya, Simran Motwani, Tao Xu, Tianhe Li, Tingbo Hou, Wei-Ning Hsu, Xi Yin, Xiaoliang Dai, Yaniv Taigman, Yaqiao Luo, Yen-Cheng Liu, Yi-Chiao Wu, Yue Zhao, Yuval Kirstain, Zecheng He, Zijian He, Albert Pumarola, Ali Thabet, Artsiom Sanakoyeu, Arun Mallya, Baishan Guo, Boris Araya, Breena Kerr, Carleigh Wood, Ce Liu, Cen Peng, Dimitry Vengertsev, Edgar Schonfeld, Elliot Blanchard, Felix Juefei-Xu, Fraylie Nord, Jeff Liang, John Hoffman, Jonas Kohler, Kaolin Fire, Karthik Sivakumar, Lawrence Chen, Licheng Yu, Luya Gao, Markos

Georgopoulos, Rashel Moritz, Sara K. Sampson, Shikai Li, Simone Parmeggiani, Steve Fine, Tara Fowler, Vladan Petrovic, and Yuming Du. Movie Gen: A cast of media foundation models. arXiv, 2024.

[38] PyTorch. `torch.distributed.batch_isend_irecv`. https://pytorch.org/docs/stable/distributed.html#torch.distributed.batch_isend_irecv.

[39] PyTorch. `torch.Tensor.pin_memory`. https://pytorch.org/docs/stable/generated/torch.Tensor.pin_memory.html.

[40] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. arXiv, 2021.

[41] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, Patrick Schramowski, Srivatsa Kundurthy, Katherine Crowson, Ludwig Schmidt, Robert Kaczmarczyk, and Jenia Jitsev. LAION-5B: an open large-scale dataset for training next generation image-text models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[42] 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, 2020.

[43] Shezheng Song, Xiaopeng Li, Shasha Li, Shan Zhao, Jie Yu, Jun Ma, Xiaoguang Mao, and Weimin Zhang. How to bridge the gap between modalities: Survey on multimodal large language model. arXiv, 2025.

[44] Xin Tan, Yuetao Chen, Yimin Jiang, Xing Chen, Kun Yan, Nan Duan, Yibo Zhu, Daxin Jiang, and Hong Xu. DSV: Exploiting dynamic sparsity to accelerate large-scale video DiT trainings. *arXiv preprint arXiv:2502.07590*, 2025.

[45] HiGHS Team. HiGHS: Linear optimization software. https://github.com/ERGO-Code/HiGHS.

[46] Ye Tian, Zhen Jia, Ziyue Luo, Yida Wang, and Chuan Wu. DiffusionPipe: Training large diffusion models with efficient pipelines. In *Proceedings of Machine Learning and Systems*, 2024.

[47] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[49] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. SimAI: Unifying architecture design and performance tunning for large-scale large language model training with scalability and precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025.

[50] Yi Wang, Yinan He, Yizhuo Li, Kunchang Li, Jiashuo Yu, Xin Ma, Xinhao Li, Guo Chen, Xinyuan Chen, Yaohui Wang, Conghui He, Ping Luo, Ziwei Liu, Yali Wang, Limin Wang, and Yu Qiao. InternVid: A large-scale video-text dataset for multimodal understanding and generation. arXiv, 2024.

[51] Yujie Wang, Shenhan Zhu, Fangcheng Fu, Xupeng Miao, Jie Zhang, Juan Zhu, Fan Hong, Yong Li, and Bin Cui. Spindle: Efficient distributed training of multi-task large models via wavefront scheduling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1139–1155, New York, NY, USA, 2025. Association for Computing Machinery.

[52] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, and Yufei Ding. WLB-LLM: Workload-balanced 4D parallelism for large language model training. arXiv, 2025.

[53] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[54] Jun Xu, Tao Mei, Ting Yao, and Yong Rui. MSR-VTT: A large video description dataset for bridging video and language. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5288–5296, 2016.

[55] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report. arXiv, 2024.

[56] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. arXiv, 2025.

[57] Zili Zhang, Yinmin Zhong, Ranchen Ming, Hanpeng Hu, Jianjian Sun, Zheng Ge, Yibo Zhu, and Xin Jin. DistTrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models. arXiv, 2024.

[58] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[59] Yijie Zheng, Bangjun Xiao, Lei Shi, Xiaoyang Li, Faming Wu, Tianyu Li, Xuefeng Xiao, Yang Zhang, Yuxuan Wang, and Shouda Liu. Orchestrate multimodal data with batch post-balancing to accelerate multimodal large language model training. arXiv, 2025.

[60] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[61] Wanrong Zhu, Jack Hessel, Anas Awadalla, Samir Yitzhak Gadre, Jesse Dodge, Alex Fang, Youngjae Yu, Ludwig Schmidt, William Yang Wang, and Yejin Choi. Multimodal C4: An open, billion-scale corpus of images interleaved with text. arXiv, 2023.