Efficient Multimodal Model Training at Scale

by

Insu Jang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2026

Doctoral Committee:

Associate Professor Mosharaf Chowdhury, Chair
Professor Aditya Akella, The University of Texas at Austin
Professor Karthik Duraisamy
Associate Professor Ryan Huang

To my family

Insu Jang

insujang@umich.edu

ORCID iD: 0009-0007-5206-2333

ACKNOWLEDGEMENTS

Pursuing a PhD has been one of the most challenging and rewarding experiences of my life. It has tested my persistence, shaped the way I think, and taught me to approach difficult problems with patience, curiosity, and care.

This journey would not have been possible without the guidance, support, and encouragement of my advisor, Professor Mosharaf Chowdhury. I am deeply grateful for the time, insight, and trust that helped me grow as a researcher. Through our discussions, I learned not only how to conduct research, but also how to ask meaningful questions, reason carefully about ideas, and pursue high standards. My early years as a PhD student were especially difficult, and I am grateful for Mosharaf's patience and support during that time. He encouraged me to build confidence in my own abilities and to find my footing as an independent researcher. It was also Mosharaf who suggested that I study systems for machine learning (ML), a field I had never expected to pursue before starting my PhD, but one that I eventually came to love. Without his guidance, I would not have been able to reach my current accomplishments, and my life would have taken a very different path. I am truly grateful for his support and deeply honored to have him as my advisor.

I would also like to give my sincere thanks to my friends and colleagues at SymbioticLab – Peifeng Yu, Jie You, Hasan Maruf, Fan Lai, Yiwen Zhang, Jiachen Liu, Jae-Won Chung, Shiqi He, Jeff Ma, Runyu Lu, Kevin Xue, and Ruofan Wu. It was a pleasure to work with them, and I am grateful for the many discussions, shared challenges, and moments of encouragement that helped me along the way. In particular, Jae-Won joined the lab together with me, and since then he has been a great friend and mentor, and I learned a lot about how to be a good collaborative researcher from him.

Finally, my deepest thanks go to my family, my wife Minkyoung Cho and my dog Dduuboo. The emotional support and encouragement from them have been invaluable, and I truly appreciate their love and support. I would not have been able to complete this journey without their support.

I will always miss my life at Ann Arbor and University of Michigan. While it was hot in Summer and cold in Winter, it was already beautiful, calm, and peaceful. Nowhere else will match the peace I felt in Ann Arbor.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF APPENDICES

APPENDIX

ABSTRACT

As large language models (LLMs) evolve into multimodal foundation models, distributed training across massive GPU clusters has become indispensable. Because distributed training necessitates frequent, collective state synchronizations across thousands of devices, any imbalance in execution time directly translates to systemic idle time, as the entire cluster must stall and wait for the slowest device. Consequently, maximizing training throughput requires consistently load-balancing the system across all participating accelerators.

However, achieving and maintaining this optimal balance is severely disrupted by two fundamental challenges in large-scale multimodal training: *resource imbalances* and *workload imbalances*. On the resource side, operating at the scale of thousands of GPUs inherently increases hardware failure events. When failures occur, they dynamically alter the cluster topology, creating variability in the total number of available GPUs and heterogeneity in the number of active GPUs per model replica. On the workload side, distributing diverse modalities to be executed in parallel across the system inherently causes severe workload imbalance. Because different modalities impose vastly different computational footprints, distributing their execution across GPUs introduces profound workload heterogeneity. Furthermore, as the relative mixture of these modalities changes from batch to batch, it creates unpredictable workload variability throughout the training process. Together, this intertwined heterogeneity and variability – spanning both the underlying hardware resources and the multimodal training workloads – make it exceedingly difficult to prevent straggler effects and sustain high training efficiency.

This dissertation studies how to systematically address heterogeneity and variability in both resources and workloads to make distributed multimodal training highly efficient and robust. We have developed three solutions to provide fault-tolerance and workload balancing for multimodal training. First, we propose Oobleck, a fault-tolerant hybrid-parallel training framework that uses heterogeneous pipeline templates to address resource variability. A pipeline template is a specification of a model replica for a given number of GPUs, and Oobleck uses a composition of heterogeneous pipeline templates to utilize all available GPUs. When failures happen, Oobleck can reinstantiate pipelines from the pipeline templates and copy missing model states from other replicas to recover from failures. This apporach allows Oobleck to achieve efficient workload rebalancing without checkpointing.

Second, we present Cornstarch, a distributed multimodal training framework that addresses intra-batch multimodal workload heterogeneity and variability. Different from existing distributed multimodal training frameworks that focus only on first-order model and data heterogeneity, Cornstarch discovers latent higher-order heterogeneity and variability in two parallelization dimensions: pipeline parallelism and context parallelism. More specifically, Cornstarch considers the frozen status of model components to balance the pipeline stages in pipeline parallelism, and it analyzes variable imbalance of workload distribution in context parallelism.

Third, we introduce Entrain that balances inter-batch multimodal workload variability. The relative workload ratio of heterogeneous modalities varies across batches, thus optimizing the parallel configuration for one batch may not be optimal for another one. A natural intuition would be to dynamically adapt the parallel configuration for each batch. However, Entrain shows a counterintuitive result that a single, static parallel configuration suffices for optimal load balancing with macroscopic batch-level profiling. While at macroscopic scale, the workload ratio between modalities converges to a stable constant, variability persists at the microscopic scale, which is exposed when a batch is split into microbatches in pipeline parallelism. Entrain addresses this with a hierarchical microbatch assignment and deferral optimization to stabilize variability across microbatches.

Together, these works provide comprehensive solutions to address heterogeneity and variability for highly efficient and robust distributed multimodal training.

CHAPTER 1

Introduction

Since the introduction of the Transformer architecture [145], deep neural networks (DNNs) and Large Language Models (LLMs) have become the de-facto standard for machine learning tasks. Modern state-of-the-art models now routinely encompass hundreds of billions or even trillions of parameters [106, 107, 39, 40, 41, 5, 139, 26, 140, 38], trained across massive, ever-growing datasets [133, 152, 112]. Beyond simply scaling parameter counts, a major paradigm shift in contemporary AI is the evolution toward natively multimodal foundation models. Rather than processing text in isolation, recent architectures [107, 25, 41, 162, 140, 38, 157] are increasingly designed from the ground up to concurrently understand, reason about, and generate content across diverse modalities. By seamlessly integrating various modalities, these natively multimodal foundation models unlock unprecedented capabilities for complex, human-like perception and interaction. Because these massive, multimodal foundation models far exceed the memory capacity and computational capabilities of any single accelerator, large-scale distributed training across thousands of GPUs has become mandatory.

1.1 Distributed Multimodal Training

To fit these models into memory and maintain reasonable training times, the adoption of combination of several parallelism strategies [102, 76, 81] has become an inevitable necessity and is now widely adopted in state-of-the-art training pipelines [17, 139, 38, 161]. This involves a combination of Data Parallelism (DP) to distribute the massive datasets across replicas [122, 123, 176], Context Parallelism (CP) to handle the exceptionally long sequence lengths characteristic of multimodal inputs [151, 164, 43, 54, 33], Tensor Parallelism (TP) to partition individual mathematical operations within a layer [102, 147, 99], Pipeline Parallelism (PP) to divide the model's sequential layers across multiple devices [115, 100, 101, 30]. These four dimensions are collectively referred to as 4D parallelism, which can be applied

to any Transformer-based model architecture, while there are other parallelism strategies specific to certain model architectures such as Expert Parallelism (EP) that distributes expert blocks in Mixture-of-Experts (MoE) models [121, 131, 52] or parallelisms for Diffusion models [32, 75, 31, 90, 158]. However, while hybrid parallelism makes training such unprecedented scale models possible, it also introduces complex requirements for workload and resource balancing.

A key characteristic of scaling up large multimodal foundation models is the need for balanced workload distribution during distributed training. Because the training process is dispersed across thousands of GPUs using various parallelism strategies, these devices must routinely communicate to exchange intermediate activations, gradients, or model parameters. This coordination creates collective synchronization points throughout the execution pipeline. At each synchronization point, the overall progression of the cluster is determined by the slowest participating device, commonly referred to as a straggler. If the workload is unevenly distributed across the GPUs, devices that finish their computation early must stall and wait until the straggler completes its assigned task. Consequently, achieving high efficiency in large-scale distributed training relies heavily on maintaining a balanced workload distribution to minimize these straggler effects.

1.2 Main Challenges: Resource and Workload Variance

Achieving and maintaining the optimal balance required by distributed multimodal training is severely disrupted by two fundamental areas of variance: resource variance and workload variance. These two pillars continuously inject heterogeneity and variability into the system, systematically creating stragglers and bottlenecking the entire cluster.

Resource Variance. Operating at the massive scale of thousands of GPUs inherently increases the frequency of hardware failures. Even though the individual failure rate or Mean Time Between Failures (MTBF) for a single GPU remains constant, the aggregate probability of encountering a failure somewhere in the system grows proportionally with the size of the cluster. This constant threat of failure is uniquely devastating in large-scale distributed training because its impact extends far beyond the immediately affected hardware. When a single GPU fails, it directly paralyzes only the highly localized cohort it belongs to, such as a single Tensor Parallel (TP) group or a specific pipeline stage. And then, because all execution dimensions in the 4D parallelism hierarchy are bound by rigorous, collective synchronizations, the stall caused by this single failed device rapidly propagates to larger execution groups, and eventually brings the entire thousands of GPUs training cluster to a complete halt.

As the system restarts and attempts to resume training, these localized failures permanently alter the underlying cluster topology. First, they introduce unpredictable *resource variability* by constantly shifting the total number of available GPUs over the course of the training run. Second, because GPUs are distributed across the rigid grid structures of multiple parallel dimensions, failures that affect only a very localized area inherently break the symmetry of the system, actively creating *resource heterogeneity*. With a cluster topology that is no longer perfectly symmetric, mapping the remaining hardware onto rigid parallel dimensions inevitably leaves the system attempting to run symmetric hybrid-parallel workloads on an asymmetric hardware backend. This results in model replicas operating with unequal numbers of active GPUs or unevenly degraded computational capabilities compared to others.

Workload Variance. Beyond the physical infrastructure, the multimodal nature of the workload itself introduces profound imbalances. Distributing diverse modalities to be executed in parallel across the system inherently causes *workload heterogeneity*. This heterogeneity arises fundamentally from both the data and the model. Processing different modalities requires handling disparate data structures (e.g., dense, high-resolution continuous image patches versus discrete text tokens) using entirely distinct model architectures (e.g., Vision Transformers for images versus Large Language Models for text). These architectural and data-level differences impose vastly unequal computational, memory, and communication footprints across the participating GPUs.

Furthermore, multimodal training is highly dynamic, leading to unpredictable *workload variability* throughout the training process. The relative computational ratio between these modalities constantly fluctuates from batch to batch depending on the exact composition of the sampled training data. Additionally, the complex ways these modalities interact during execution – such as non-causal attention patterns dynamically shifting focus between text and visual inputs – further perturb the distribution of work over time. Consequently, a distributed execution plan that perfectly balances the workload for one training iteration may become severely imbalanced in the next, making it exceedingly difficult to sustain optimal throughput without continuous adaptation.

Together, this intertwined heterogeneity and variability – spanning both the underlying hardware resources and the multimodal training workloads – makes it exceedingly difficult to prevent systemic idle time and sustain high training efficiency.

1.3 Thesis Statement and Contributions

Thesis Statement. *This dissertation proposes a comprehensive system architecture that both statically orchestrates and dynamically adapts to the intertwined heterogeneity and variability of hardware resources and multimodal workloads. By challenging long-held assumptions and rules of thumb in distributed system design, this work effectively neutralizes systemic straggler effects to maximize end-to-end training throughput.*

We have developed three solutions to provide fault-tolerance and workload balancing for distributed multimodal training.

Oobleck [57]. To provide fault-tolerant distributed training, a straightforward approach is to use checkpointing that periodically saves the intermediate training states to the persistent storage [6]. When failures happen, the entire training job is restarted and loads the last checkpoint to resume training. This design is simple and easy to implement, but it requires a lot of overhead to save and load the checkpoint [102]. Additionally, when hybrid parallelism is used, it is not trivial to find a new optimal configuration for the remaining GPUs after failures. Production environments, therefore, often resort to using backup GPUs to cover the failures to maintain the initial parallel configuration, which forces the backup GPUs to idle until the failures happen. Oobleck is a breakthrough work that solves these problems by introducing pipeline templates and utilizing model state redundancy across the replicas. Pipeline templates are a set of specifications of a model replica for a given number of GPUs, and a whole parallel configuration is generated from a composition of these templates. Composition of pipeline templates is a key idea that allows Oobleck to utilize all available GPUs throughout the duration of training even when the number of GPUs changes due to failure and recovery events. When failures happen, Oobleck reinstantiate pipelines from the pipeline templates and copy missing model states from other replicas, instead of loading the checkpoint. Heterogeneity and imbalance of computational capabilities between the replicas due to failures is resolved by redistributing the batch proportionally to the computational capabilities of the replicas.

Cornstarch [56]. To support the era of upcoming multimodal foundational models, a few works have attempted to address the heterogeneity in multimodal training, by focusing on model and data heterogeneity [49, 173, 34, 163]. While it remains important to address the heterogeneity in multimodal training, they are incremental improvements over the existing works that focus on unimodal training by distributing the model and data in a different way to adjust for the heterogeneity [102]. Cornstarch transcends such first-order heterogeneity and addresses higher-order heterogeneity and variability in multimodal training. First, multimodal training typically starts by integrating independently pretrained unimodal foun-

dation models (e.g., a pretrained Vision Transformer combined with a pretrained LLM) rather than training the entire unified architecture from scratch. To seamlessly unify these disparate components, the training process proceeds through multiple distinct stages, each focusing on specific learning objectives such as modality alignment or instruction tuning. Depending on the objective of the current stage, the model components are either frozen or trainable. This partial freezing of the model drastically alters the computational footprint of different layers, introducing severe, previously unconsidered workload imbalance across pipeline stages that traditional unimodal parallelization strategies fail to address. Cornstarch considers the frozen status of model components to balance the pipeline stages in pipeline parallelism. Second, multimodal training generates highly dynamic attention patterns that are not causal, which introduces variable imbalance of workload distribution in context parallelism. While causal attention patterns are always static that can easily be distributed, non-causal attention patterns are highly dynamic depending on the content and the location of the input data,which makes efficient workload distribution challenging. Cornstarch introduces a novel workload distribution algorithm that balances the computational cost of non-causal attention patterns at both inter-GPU and intra-GPU granularity.

Entrain [55]. The heterogeneity and variability in multimodal data distribution does not only exist within a single batch, but also across different batches. Typical analysis of heterogeneity of multimodal data shows the distribution of each modality independently [173, 163], however, multiple modalities are highly entangled within a single sample, which introduces another layer of heterogeneity and variability. Intuitively, it would make sense to dynamically adapt the parallel configuration for each batch to balance the workload distribution. Counter-intuitively, even with variability of the ratio of modalities across samples, we figure out that macroscopic-scale large size of batch can stabilize the ratio of modalities in a batch, which allows to derive a single parallel configuration that considers the ratio of modalities in a batch is sufficient for optimal load balancing throughout the training process. When pipeline parallelism is used, however, a batch is split into multiple microbatches, which re-exposes variability of the ratio fo modalities across microbatches. Dynamically adapting the parallel configuration for each microbatch is prohibitively expensive, therefore, Entrain introduces deferred workload optimization to balance the workload distribution.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows.

- Chapter 2 details existing approaches for fault-tolerance and introduces Oobleck along with pipeline templates. We mathematically prove the capability of Oobleck's tolerance to

failures and the capability of the composition of pipeline templates to utilize all available GPUs. We also provide detailed mechanisms of redistributing the batch to adjust the workload balance.

- Chapter 3 presents Cornstarch, a distributed multimodal training framework that analyzes computational cost of model components with frozen status precisely for balanced pipeline parallelism, and introduces a workload-aware context parallelism that balances the distribution of non-causal attention computation intra- and inter-GPU.
- Chapter 4 introduces Entrain, which addresses the heterogeneity and variability of multimodal data distribution across batches and microbatches. We show that a macroscopic-scale large batch size stabilizes the modality ratio to enable a single parallel configuration, and present a deferred workload optimization technique to maintain balanced workload distribution across highly variable microbatches.
- Chapter 5 concludes the dissertation by summarizing the contributions and discussing future directions.

CHAPTER 2

Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates

This chapter focuses on resilient and fault-tolerant distributed large model training and introduces Oobleck. As a core contribution of Oobleck, we introduce the concept of pipeline templates that provide resilient and high-throughput distributed training. Pipeline templates are designed to cover any number of available GPUs at any time during training, and each pipeline template is an optimal configuration for a given number of GPUs. Balancing across pipeline templates is achieved by batch distribution, where the iteration time of each pipeline template is precisely estimated to minimize the overall straggler problem.

The rest of the chapter is organized as follows. Section 2.1 provides the introduction to Oobleck. Section 2.2 introduces the background knowledge of hybrid parallelism and fault tolerance in distributed training. Section 2.3 describes the design of Oobleck, followed by algorithms (Section 2.4), dynamic reconfiguration (Section 2.5), and implementation (Section 2.6). Evaluation results are presented in Section 2.7. We then survey related work in Section 2.8 and conclude the chapter in Section 2.9.

2.1 Introduction

DNN models continue to become larger [146]. Many recent advances in deep learning have been attributed to significant increases in model size to hundreds of billions of parameters and training on ever-growing datasets [106, 132]. Recent studies suggest that a trillion-parameter model would require at least 2TB of memory simply to store model parameters, and tens or hundreds of TB for training [122, 127, 123, 80, 60]. Naturally, scaling large model training has received intense attention over the past few years [101, 30, 132, 171, 156]. Distributed *hybrid-parallel training* [102, 177] that combines model and data parallelism has emerged as the primary approach to training such large models.

Unfortunately, the likelihood of experiencing failures also increases with the scale and duration of training [59, 44, 153]. The effect is further amplified by the synchronous nature of DNN training, which causes all participating devices to idle until the failed one has recovered, causing massive underutilization. Indeed, teams from Meta, HuggingFace, and LAION report significant underutilization from failures when training large models [171, 156, 8]. Failure rates are even higher for training jobs that use spot instances in the cloud [141, 6].

Existing frameworks have little systematic support for fault tolerance during hybrid-parallel training. Ensuring continuous operation in the presence of failures fundamentally requires redundancy in one form or another. Model state redundancy in data-parallel training is the only form of "free" redundancy, because each worker already has a copy of the model states. Most solutions harness the inherent redundancy provided by data parallelism and utilize its embarrassingly parallel nature to elastically change the number of GPUs while dynamically changing the global batch size [53, 114, 73, 172]. However, they are unable to extend these benefits to hybrid parallelism and are limited only to data-parallel training.

In contrast, fault tolerance approaches tailored toward hybrid parallelism struggle to leverage any inherent redundancy. Instead, they introduce additional redundancy in various forms; e.g., having a pool of standby GPUs [156], using checkpoints to reconfigure and restart [6], and performing redundant computations in anticipation of a possible failure [141]. Essentially, they consider overhead during training vs. overhead to recover from failure(s), and choose one of the two extremes (§2.2.2): if failures are infrequent, amortized overhead for reconfiguration would also be low [6]; if failures are more frequent, then incurring some overhead during training may be more preferable than spending significant time in recovery [141].

In this paper, we present Oobleck, a fault-tolerant hybrid-parallel training framework. It provides high training throughput, guaranteed fault tolerance, and fast recovery *without* introducing additional overhead.

Pipeline templates are at the core of Oobleck's design. A pipeline template is a specification of pipeline execution for a given number of nodes. They are designed during the planning phase by Oobleck's template generator and reused during execution by Oobleck's execution engine. All pipeline templates are logically equivalent yet physically heterogeneous; each has a different number of nodes and associated configurations that can be used to instantiate a pipeline for a given model. Oobleck uses one or more pipeline templates to create pipeline replicas to exploit the inherent model states redundancy across the replicas. Pipelines affected by failures can reconstruct model states by copying missing layers from other replicas without having to restart from a checkpoint.

More specifically, given a training job starting with the number of maximum simultaneous

failures to tolerate f, Oobleck's execution engine instantiates at least $f+1$ heterogeneous pipelines from the generated templates. The fixed global batch is distributed proportionally to the computing capability of heterogeneous pipelines such that all pipeline replicas train roughly at the same rate. Upon failures, Oobleck avoids demanding analysis of finding a new optimal configuration by simply reinstantiating pipelines from the precomputed pipeline templates while achieving maximum node utilization. This is always possible for f or fewer failures because Oobleck provably guarantees that a combination of pipelines generated from those precomputed templates can fully utilize all the remaining nodes.

We have implemented Oobleck on top of PyTorch and HuggingFace Transformers [155] using components from DeepSpeed [125] and Merak [68]. We evaluate Oobleck and compare its performance against Bamboo and Varuna across large models like GPT-3 with billions of parameters. Oobleck outperforms the state-of-the-art solutions by up to 13.9× as we consider different frequencies of failures, spot instance traces, and models of different sizes and computation complexity.

Overall, we make the following contributions in this paper.

- We present Oobleck, a novel framework for resilient distributed training that provides guaranteed fault tolerance and maximizes throughput.
- Oobleck introduces pipeline templates to (re)instantiate pipelines. Pipeline templates allow quick failure recovery and utilization of all available GPUs.
- We implement and evaluate Oobleck with several large models, e.g., variants of GPT-3, to demonstrate large improvements in terms of throughput and failure recovery.

Oobleck is open-source and available on GitHub.[1].

2.2 Background and Motivation

In this section, we briefly introduce hybrid parallelism that is commonly used for large model training. We also discuss existing fault tolerance strategies for hybrid-parallel training and highlight their limitations.

2.2.1 Hybrid Parallelism

As DNN models continue to grow in size and are trained on increasingly larger datasets [171, 156, 132], using just *data parallelism* or *model parallelism* is often not enough to efficiently train a DNN. Data parallelism splits and distributes input to multiple GPUs, but it requires

[1] https://github.com/SymbioticLab/Oobleck

each GPU to hold the entire model [53]. Model parallelism accommodates large model training by splitting the model across multiple GPUs – for example, *pipeline parallelism* splits the model into groups of layers called stages, and *tensor parallelism* slices each model layer into several tensor chunks. However, the former has pipeline bubble overheads that decrease compute utilization as the pipeline grows deeper [132]. The latter suffers from high communication cost that cannot be hidden in computation, because it requires several all-reduce operations in the critical path of both forward pass and back-propagation [122]. Consequently, a combination of data and model parallelism techniques – aka *hybrid parallelism* – is used in practice to train large DNN models [132, 156, 30, 102, 100, 101].

2.2.2 Fault Tolerance in Distributed Training

Failures are the norm in distributed systems, and distributed DNN training is no exception. The probability of experiencing one or more failures increases with the increasing number of GPUs and the duration of training. For instance, a Meta AI team suffered approximately 100+ hardware failures and had to do 100+ major restarts during OPT-175B training [171]. Because of the synchronous nature of distributed training, the cost of even one failure is *multiplied*: all the GPUs must idle until the impact of failure has been mitigated. The impact of this phenomenon was recently highlighted in detail by a LAION team when training CLIP models [8] as well as a BigScience team during BLOOM training [156].

Several recent works have focused on fault-tolerant data-parallel training via dynamically changing the global batch size [53, 114, 159, 73]. Fault-tolerant hybrid-parallel training is more challenging because the model is distributed across multiple GPUs. There are two primary approaches.

1. *Checkpointing:* Checkpointing is a popular mechanism to persistently store training progress. For example, the BigScience team training the BLOOM model [156] and the Meta AI team training the OPT model [171] used this recently. However, manual reconfiguration after identifying and replacing the failed GPUs with spare ones is time-consuming. Varuna [6] introduced job morphing to dynamically reconfigure training jobs to achieve the best performance with the remaining resources after restarting from the most recent checkpoint. While this does not introduce significant fixed overhead, recovery time can be unsustainable when the failure rate is high [141].
2. *Redundant computation (RC):* To avoid reconfiguration and restart overheads, Bamboo [141] recently introduced redundant computation (RC) where each pipeline stage is redundantly computed in two subsequent nodes. When a node fails, the backup node computes the forward and backward passes of the failed node. RC introduces fixed com-

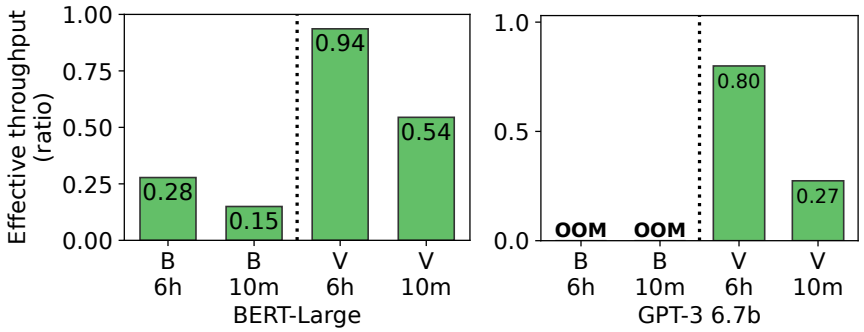


Figure 2.1: Effective time spent in training for Bamboo (B) and Varuna (V) running BERT-large and GPT-3 6.7b models for different frequency of failures (6h and 10m). An optimistic upper-bound of the optimal is 1.00, when training remains unaffected by failure(s).

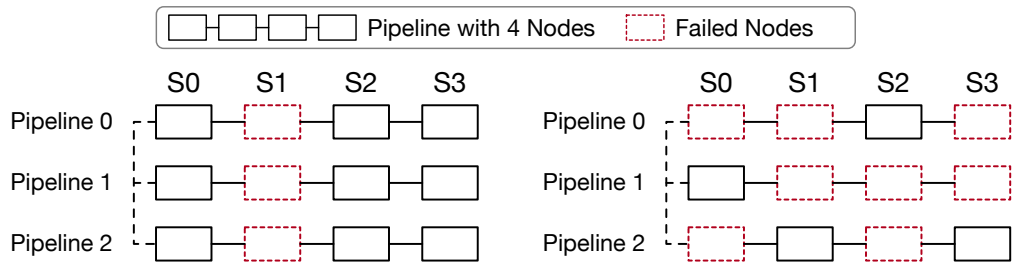
putational overhead due to redundancy in computing and memory overhead of holding redundant states in each node. Note that reconfiguration and restart from a checkpoint is still necessary if two adjacent nodes fail.

2.2.3 Limitations of the State-of-the-Art

State-of-the-art approaches for fault-tolerant hybrid-parallel training do not provide any systematic fault tolerance guarantees, and they have large overheads, especially when models become larger.

Figure 2.1 shows the time effectively spent in training (i.e., the time that leads to training throughput) using Varuna and Bamboo when training BERT-Large and GPT3-6.7B – with 340 million and 6.7 billion parameters, respectively – when one failure happens every six hours and every 10 minutes on average. Detailed experimental setup is in Section 2.7.1. Varuna provides higher training throughput compared to Bamboo, but it has noticeable job restart overhead. Although some recent proposals improved checkpointing overhead [98, 29], loading checkpoints upon restarts is still in the critical path. When failures become more frequent, restarting overheads dominate. Additionally, performance degradation after a failure is not proportional to the number of failures, but worse. This is because Varuna's hybrid parallelism uses a grid topology; one GPU failure breaks the grid of GPUs, leaving some of them idle.

Bamboo, in contrast, reduces checkpointing and restart overheads, but RC in Bamboo introduces significant performance overhead, even when some portion of the overhead is hidden in pipeline bubbles. Specifically, its forward RC redundantly computes the next stage all the time, lowering throughput even in the absence of failures. Backward RC takes place only after failure(s), but it adds additional overhead to some pipelines' iteration times,



(a) Failures happen to nodes with the same stage (S1 lost, not recoverable)

(b) Failures in random places (all stages alive, recoverable)

Figure 2.2: An example of Oobleck's fault tolerance guarantees with $f = 2$. S refers to a pipeline stage. (a) In the worst case, we lose model states (a stage) if more than f nodes fail. (b) In the general case, however, more than f node failures can be tolerated.

making them stragglers and inflating the iteration time of synchronous training. Worse, Bamboo also needs to restart with a full reconfiguration from a checkpoint for as few as two failures when two adjacent nodes fail.

Finally, both approaches perform poorly for larger models, especially when failures are frequent. Bamboo runs out of memory and Varuna spends most of the time preparing to train. We aim to design a solution that works well regardless of the frequency of failures both in terms of the fault tolerance guarantee it provides and the throughput it achieves.

2.3 Oobleck Overview

2.3.1 Pipeline Templates

Oobleck introduces *pipeline templates*, each of which is a pipeline specification that defines how many nodes should be assigned to a pipeline, how many stages to create, and how to map model layers in stages to GPUs. All pipelines *instantiated* by Oobleck are from precomputed pipeline templates. In practice, Oobleck instantiates multiple (possibly heterogeneous) pipelines from a set of heterogeneous pipeline templates to fully utilize an arbitrary number of nodes even when they do not form a grid. Decoupling "planning" (pipeline template generation) from "execution" (pipeline instantiation) enables fast failure recovery; a pipeline with lost node(s) is replaced with a new pipeline instantiated from another pipeline template that requires a fewer number of nodes.

2.3.2 Fault Tolerance Guarantees

Oobleck guarantees fault tolerance without restart for up to f simultaneous *pipeline* failures, because in the worst case, f node failures are enough to cause f pipelines to fail. Consider Figure 2.2, where there are three pipeline replicas each with four stages – i.e., each stage has three replicas. We can tolerate at most two simultaneous node failures in the worst case, because if three failures take out all three replicas of any stage (stage 1 in Figure 2.2a), the pipelines cannot be recovered. In general, however, Oobleck can tolerate in excess of f node failures, provided that a minimum of one copy of the entire model states is retained across the pipelines. For example, even after eight node failures in Figure 2.2b, one copy of each stage still remains alive; hence, it is recoverable.

2.3.3 System Components

Oobleck extends existing ML training frameworks in two primary aspects (Figure 2.3). First, it has *a pipeline template generator* to generate a set of heterogeneous pipeline templates that can be used by the execution engine for pipeline instantiation. Pipeline templates are created only once and never change during the entire training.

Second, Oobleck has a *distributed execution engine* that enables efficient heterogeneous pipeline execution. It instantiates pipelines from the given set of pipeline templates considering the user's fault tolerance threshold (f) and batch information (global batch and microbatch size). It creates at least $f + 1$ (possibly heterogeneous) pipeline replicas so that at least one copy of the model exists anytime during training for up to f simultaneous failures. The batch distributor calculates the number of microbatches for each pipeline that balances execution latency between heterogeneous pipelines. Pipeline instantiation and batch distribution happens whenever a node fails or is added. The node change monitor detects node failure(s) and node additions; then the execution engine dynamically reconfigures using precomputed pipeline templates.

2.3.4 Training Lifecycle

Oobleck users submit training jobs with a fault tolerance threshold f, a model and dataset to train on N homogeneous nodes (received from a GPU cluster manager [59, 153]), and batch size information ① (Figure 2.3). Oobleck's pipeline template generator first creates a set of pipeline templates ②. The distributed execution engine instantiates pipelines from the templates and deploys them on the cluster ③.

When node failure(s) happen ④, if we have a complete model replica, Oobleck does not

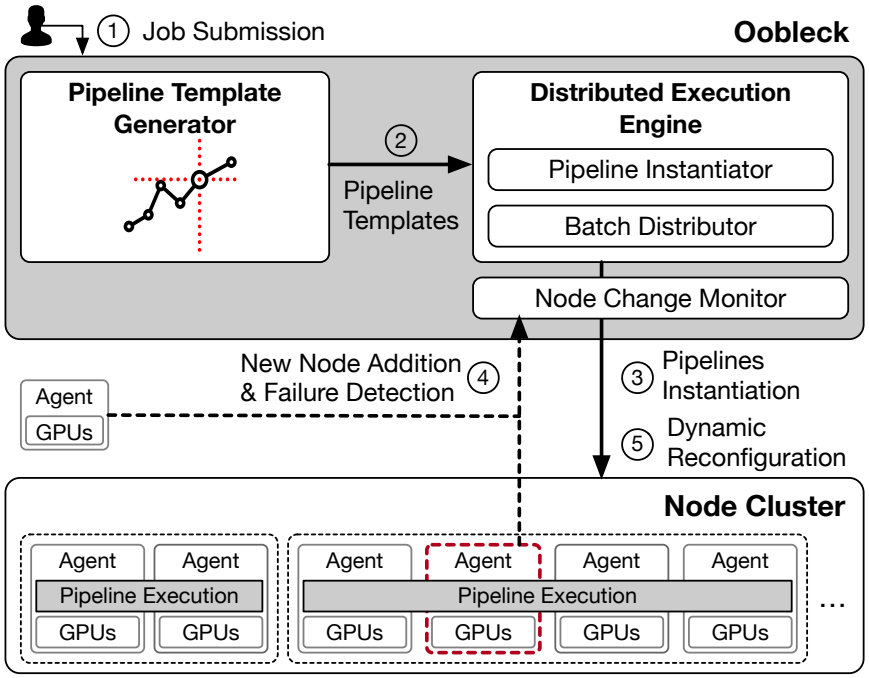


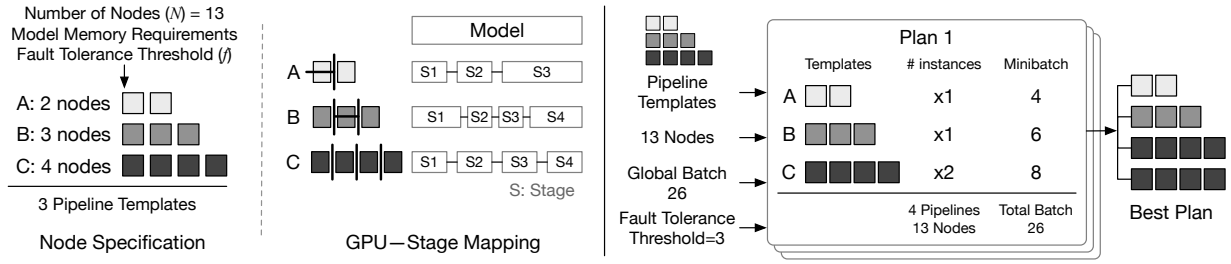
Figure 2.3: Oobleck system overview.

restart but reconfigures the pipelines ⑤. The execution engine reinstantiates pipelines from the templates to make sure all nodes are used. During pipeline reinstantiation, nodes share information about the ownership of model states and copy missing model states from others. After reconfiguration and model states copying are done, nodes resume training. A job runs until it reaches the target accuracy, a user terminates it, or Oobleck cannot maintain $f + 1$ pipeline replicas. If the cluster cannot hold $f + 1$ replicas, Oobleck stores the progress, informs the user, and exits. Thereafter, the user can decide to restart training from a recent checkpoint once enough nodes have recovered to maintain $f + 1$ replicas.

2.4 Oobleck Planning Algorithm

Oobleck tolerates f simultaneous failures by instantiating r ($\geq f+1$) heterogeneous pipeline replicas of the same model. Each of these logically equivalent pipeline replicas performs hybrid-parallel training. Unlike existing solutions that force a single homogeneous hybrid-parallel configuration over a rigid grid (# GPUs per pipeline stage $\times$ # pipeline stages $\times$ # pipeline replicas) [6], Oobleck's *heterogeneous pipeline execution* can utilize all available GPUs.

Because the number of available nodes can vary over time due to failures, Oobleck requires an effective mechanism to derive *all* possible configurations of heterogeneous pipelines that



(a) Generating pipeline templates (§2.4.1) (b) Pipeline instantiation (§2.4.2)

Figure 2.4: Oobleck's planning algorithm overview. First, it generates a set of pipeline templates, a combination of which can utilize all available nodes. A template is a specification of pipelines, how many nodes are assigned and how GPUs in the nodes should be mapped to pipeline stages. Then, pipelines are instantiated following the fastest (best) plan after checking all possible plans. A plan includes how many pipelines should be instantiated from each pipeline template given the number of nodes and how batch size should be distributed to the pipelines.

can utilize all available GPUs at any point in time. Oobleck's pipeline template generator computes a fixed set of pipeline templates at the beginning of the training job for the entire training (§2.4.1). The pipeline execution engine instantiates zero or more copies of each of the templates (i.e., a collection of heterogeneous pipeline replicas) to utilize all currently available nodes (§2.4.2).

2.4.1 Generating Pipeline Templates

Each pipeline template created by Oobleck is a set of specifications that defines how many nodes to use, and how the given GPUs and model layers are mapped to make pipeline stages use all those nodes. Figure 2.4a illustrates this process. In this example, we generate a set of pipeline templates. We first determine the number of heterogeneous pipeline templates and their node specifications (number of nodes) needed to utilize all available nodes, given the initial number of nodes N, the amount of memory required to train a model, and the fault tolerance threshold f (§2.4.1.1). In this case, three heterogeneous pipeline templates with 2, 3, and 4 nodes have been chosen. Then, for each template, we partition the model and map the available GPUs to them to create pipeline stages that minimize the iteration time (§2.4.1.2).

2.4.1.1 Node Specification.

Because pipeline templates never change during training and generating arbitrarily many of them is expensive, we must determine how many pipeline templates are needed, and then

how many nodes each of the templates should use, so that some combination of them can always utilize any number of available nodes, even when we have fewer number of nodes than at the beginning after failures.

This can be formulated as the Frobenius problem [124], which finds the Frobenius number g, the largest number that cannot be represented as a linear combination of integers. Meaning, any number of available nodes after failures $N' > g$ can be expressed as a linear combination of the given pipeline templates, each with a specified integer number of nodes.

If we represent the number of pipeline templates as p and the number of nodes for the i-th pipeline template be ordered values $n_i (0 < n_i \leq N)$ where $n_i < n_{i+1}$, we can guarantee that any feasible $N' \geq (f+1)n_0$ is always larger than g when the following conditions are met [129].

1. $p > n_0 - 1$.
2. n_i are consecutive integers $(n_i + 1 = n_{i+1})$.

See Appendix A.1 for a proof.

We set the lower bound of N' as $(f+1)n_0$: the smallest number of nodes required to maintain $f+1$ replicas of the model, because n_0 is the smallest number of nodes for a single pipeline. Any smaller N' cannot respect the fault tolerance threshold f.

Choice of n_0 and p. There are several choices for a set of pipeline templates that satisfy the conditions depending on the values of n_0 and p. We choose the smallest possible n_0 and the largest p. We select the minimum n_0 because shallow pipeline execution (smaller n_0) typically takes less time for the same amount of computation [132]. Although a large p does not directly benefit planning, it helps reduce reconfiguration overhead. The largest p can be calculated from the largest possible value for n_{p-1} (referred to as $n_{p-1}^{\max}$). When all but one of the $f+1$ replicas use n_0 nodes and the last one uses all the remaining nodes, $n_{p-1}^{\max} = N - fn_0$. We now have p to be the length of the range from n_0 to $n_{p-1}^{\max}$.

2.4.1.2 GPU – Stage Mapping.

Given the number of nodes in each pipeline template, we must determine how to best use them by finding the number of pipeline stages, partitioning the model layers to the stages, and mapping the nodes to those stages. We propose a divide and conquer algorithm to find the mapping that minimizes the iteration time. This algorithm divides the model into pipeline stages and the nodes into a set of GPUs at the same time, and then maps each of them so that we utilize all GPUs in the given nodes. It then iterates over all possible combinations of GPU–stage mapping and finds the one that minimizes the iteration time.

Let $T(S', u, v, d)$ be the minimum iteration time for layers $(l_u, l_{u+1}, \ldots, l_{v-1})$ partitioned

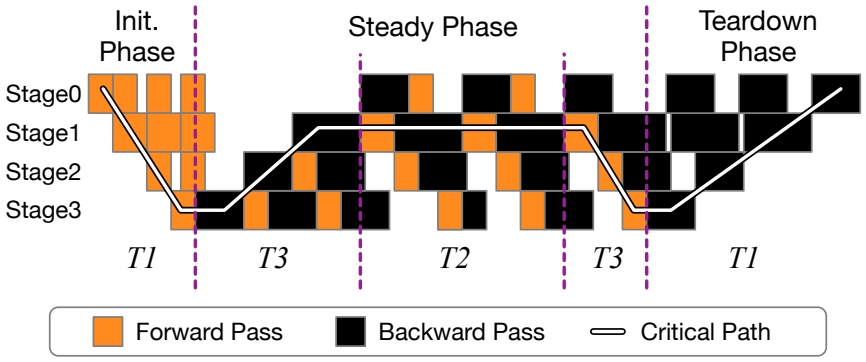


Figure 2.5: 1F1B pipeline execution breakdown ($T1, T2, T3$)

into S' stages and running on d GPUs. A pipeline for the entire model using all GPUs in the pipeline template then has the minimum iteration time $T(S, 0, L, n \cdot M)$, where the model has L layers, and there are n number of nodes in the pipeline template, each of which has M GPUs.

To calculate the minimum iteration time of a pipeline, the algorithm considers its critical path and breaks T down into three terms $T1$, $T2$, and $T3$ (Figure 2.5). $T1$ represents 1F1B initialization and teardown phases of pipeline execution, which include one forward and one backward for all stages. The steady phase in the middle has one forward and one backward pass alternating. The critical path may still include forward and backward passes of other stages than the slowest one on each end, similar to $T1$. We thus split the steady phase into $T2$ and $T3$. $T2$ includes the slowest stage alternating forward and backward, and $T3$ is the remaining part.

Divide. In the division phase, we divide the model and the nodes at the same time. Division continues until we cannot partition either GPUs or model layers, or the number of partitions matches the desired number of stages. If multiple GPUs are assigned to a pipeline stage, tensor parallelism is used to accelerate it. Figure 2.6 illustrates such a division and mapping process. After both sub-problems are conquered, the algorithm combines their results to calculate the execution time of a multi-stage pipeline created by connecting two sub-problems. From the definitions of $T1$, $T2$, and $T3$, the division and combination process can be defined

as recursive structures for each term:

$$T1_{s,k,m}(S', u, v, d) = T1_{s,k,m}(s, u, k, m) + T1_{s,k,m}(S' - s, k + 1, v, d - m) \tag{2.1}$$

$$T2_{s,k,m}(S', u, v, d \mid k^*) = (N_b - S' + k^* - 1)(F_{s_{k^*},m} + B_{s_{k^*},m}) \tag{2.2}$$

$$T3_{s,k,m}(S', u, v, d \mid k^*) = \begin{cases} T3_{s,k,m}(s, u, k, m \mid k_1^*) \\ +T1_{s,k,m}(S' - s, k + 1, v, d - m) & \text{if } k^* = k_1^* \\ T3_{s,k,m}(S' - s, k + 1, v, d - m \mid k_2^*) & \text{otherwise} \end{cases} \tag{2.3}$$

We iterate over s, k, and m globally, and find a (s, k, m) that minimizes $T1_{s,k,m} + T2_{s,k,m} + T3_{s,k,m}$. Each $T1_{s,k,m}$, $T2_{s,k,m}$, and $T3_{s,k,m}$ is the solution of $T1$, $T2$, and $T3$, respectively.

k^* denotes the index of the slowest stage, derived from either k_1^* or k_2^*, the slowest stage indices of the two sub-problems. $T2$ depends on the number of microbatches (N_b) deployed to the pipeline, which is not yet determined. From prior observations that the pipeline bubble overhead is negligible with $N_b \geq 4S'$ [50], we temporarily use $N_b = 4S'$ in planning. The structure of $T3$ is special as it includes $T1$ in Equation 2.3. T3 is an accumulation of forward and backward time for all the following stages after the slowest ($\sum_{k=k^*}^{S'-1}(F + B)$). If s_{k^*} is in the first half sub-problem (i.e. $s_{k^*} == s_{k_1^*}$), it can be broken down to $\sum_{k=k^*}^{s}(F + B) + \sum_{k=s+1}^{S'-1}(F + B)$, each of which represents $T3$ of the first half and $T1$ of the second half, respectively.

Conquer. When a problem has just one stage, we can easily calculate the execution time of running a stage s with $l_u, \ldots, l_{v-1}$ layers on d GPUs:

$$\begin{aligned} T1(1, u, v, d) =& F_{s,d} + B_{s,d} = \sum_{k=u}^{v-1}(F_{l_k,d} + B_{l_k,d}) \\ T2(1, u, v, d) =& 2(F_{s,d} + B_{s,d}) \\ T3(1, u, v, d) =& F_{s,d} + B_{s,d} \end{aligned} \tag{2.4}$$

There is one requirement for d GPUs running a single stage: *all d GPUs should be in the same node.* It is reasonable because if GPUs span several nodes, the cross-node network becomes a bottleneck and lowers the utilization of high-throughput intra-node network in collective communications done in intra-layer parallel execution. We simply mark all T values as ∞ if cross-node GPUs are given.

Choosing the number of stages S. We do not know which S provides the minimum iteration time. Therefore, we iterate over possible values of S in $(n, n + 1, \ldots, L)$. Because we partition the model at layer granularity, the number of stages cannot exceed the number

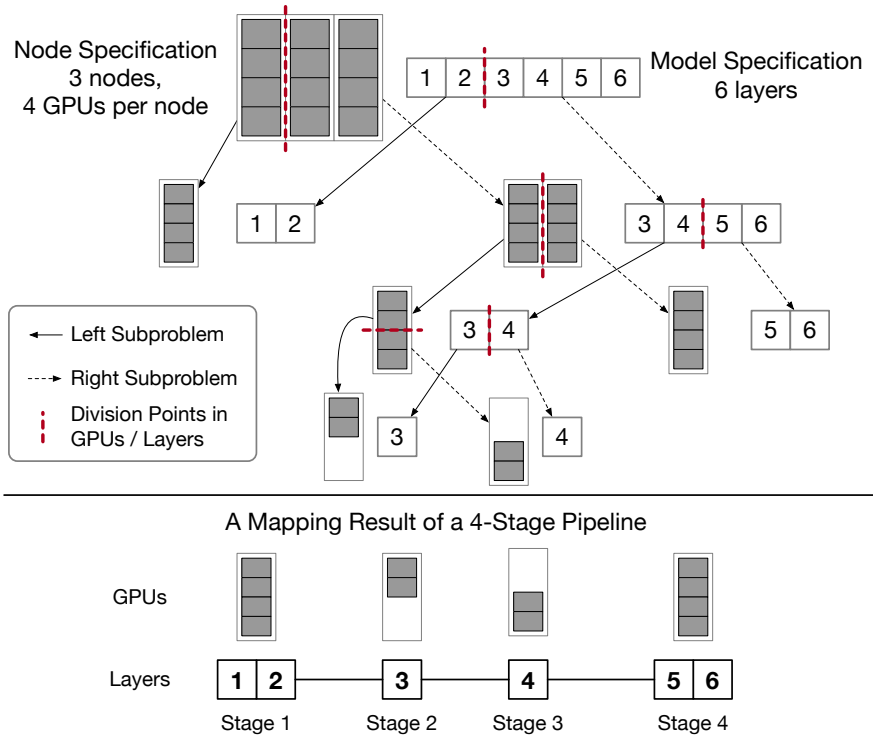


Figure 2.6: A toy example of division process for a 4-stage pipeline template with 3 nodes (template B in Figure 2.4) and a model with 6 layers. The model and the GPUs in nodes are divided into two sub-problems together. When division is done, each group of partitioned GPUs and layers form a stage. The algorithm iterates all combinations of layer partitioning and GPU partitioning to find the minimum T.

of layers L. The minimum is derived from the constraint that a single stage cannot be assigned to two or more nodes. If S becomes less than n, it breaks the constraint and some stages should have at least two nodes assigned according to the pigeonhole principle.

Time complexity of the naive implementation. The recursive stage division happens in $O(L)$. For every division, stages and layers are partitioned and they are assigned to two device sub-clusters. Stage and layer partitioning have $O(L)$ choices. Partitioning nodes is done in $O(n)$, but GPUs within a single node can further be partitioned, adding $O(M)$. Time complexity of layer partitioning and device assignment for a given number of stage is $O(LnM)$. Divide and conquer happens for each feasible S, and iterating over S values is $O(L-n)$. Therefore, the overall algorithm time complexity per pipeline template is $O\left((L-n)L^3nM\right)$.

Using memoization to reduce complexity. We cache all intermediate results to accelerate the divide and conquer algorithm. It boosts not only getting the mapping of one pipeline template but also helps in deriving the mapping of the other pipeline templates.

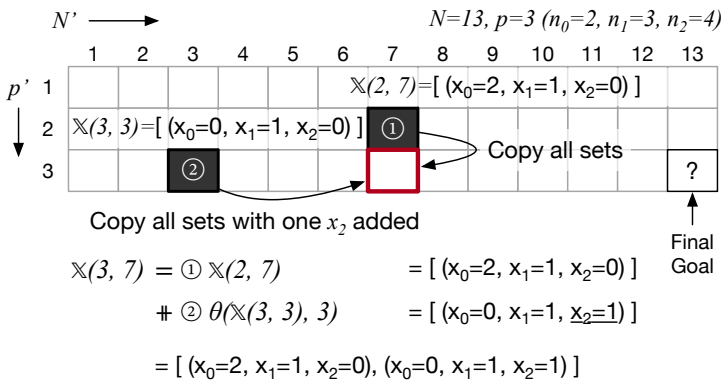


Figure 2.7: The dynamic programming algorithm finding all list of feasible **X**s. Underlined $x_2 = 1$ is added by $\theta()$ function.

In fact, running the algorithm for the largest pipeline template (with n_{p-1} nodes) is enough to calculate intermediate caches required for building the mapping of all the other pipeline templates. With all intermediate caches present, calculating the mapping of another smaller pipeline template can be done in $O(Ln)$.

2.4.2 Pipeline Instantiation

Given a set of pipeline templates, we know by construction (§2.4.1.1) that there exists a combination of them that Oobleck can instantiate to utilize all available nodes. However, such a combination of heterogeneous pipelines may not be unique. So we first find all such feasible combinations (§2.4.2.1). Oobleck chooses a plan with the highest estimated throughput among all the feasible combinations (§2.4.2.2).

2.4.2.1 Enumerating All Instantiation Options.

Although we know that we can utilize all available nodes with the set of pipeline templates, the number of pipelines to be instantiated from each pipeline template is undetermined. Worse, there may be several pipeline configurations that use all nodes from the same pipeline template set. For example, 13 nodes can be represented as the plan 1 in Figure 2.4b ($1 \cdot n_0 + 1 \cdot n_1 + 2 \cdot n_2$), but also as ($0 \cdot n_0 + 3 \cdot n_1 + 1 \cdot n_2$), and more. We therefore enumerate all feasible pipeline sets for currently available nodes and pick one that maximizes training throughput.

Let $\mathbb{X}(p, N)$ be a list of all feasible pipeline sets $[\mathbf{X}_0, \mathbf{X}_1, ...]$. Each $\mathbf{X}_i$ is a set of the number of pipelines to be instantiated $(x_0, x_1, \ldots, x_{p-1})$, using p number of heterogeneous pipeline templates with the number of nodes specification $(n_0, n_1, \ldots, n_{p-1})$, so that all N

nodes are used by pipelines. A feasible $\mathbf{X}_i$ satisfies the following requirements:

1. $N = x_0 n_0 + x_1 n_1 + \cdots + x_{p-1} n_{p-1}$ (All nodes are used).
2. $\sum_{j=0}^{p-1} x_j \geq f + 1$ (Number of pipelines is at least $f + 1$).

We exploit dynamic programming for the coin change problem to find $\mathbb{X}$. The coin change problem finds a combination of coins that add up to the given amount of money [9]. It is an equivalent problem to Requirement 1 above if we replace denominations of each coin with n_i, and the given amount of money with N. We formulate the dynamic programming structure as:

$$\mathbb{X}(p', N') = \mathbb{X}(p' - 1, N') +\!\!+ \theta(\mathbb{X}(p', N' - n_{p'}), p') \tag{2.5}$$

where $+\!\!+$ means concatenating two lists, and $\theta(\mathbb{X}, p')$ is a function that increases $x_{p'}$ by 1 in every $\mathbf{X}_i$s in $\mathbb{X}$.

Figure 2.7 shows the execution of the dynamic programming algorithm. The two terms in Equation 2.5 are associated with each black boxes. $\mathbb{X}$ in the red box should include all $\mathbf{X}_i$s that use all seven nodes in instantiating pipelines using the three different pipeline templates ($n_0 = 2, n_1 = 3, n_2 = 4$). $\mathbb{X}$ in ①, all $\mathbf{X}_i$s use seven nodes and are already feasible for $\mathbb{X}(3, 7)$. We just copy them. $\mathbb{X}$ in ②, however, only uses three nodes in total. By adding one four-node pipeline (increasing x_2 by 1), all $\mathbf{X}_i$s use seven nodes and become feasible for $\mathbb{X}(3, 7)$.

The dynamic programming is done in $O(Np)$ filling all table elements. $\mathbb{X}$ in the bottom-right corner of the table contains all feasible $\mathbf{X}_i$s. To satisfy Requirement 2, we filter the list and obtain sets with $\sum_{j=0}^{p-1} x_j \geq f + 1$.

2.4.2.2 Calculating Throughput with Batch Distribution.

Oobleck's execution engine needs to choose from several feasible $\mathbf{X}_i$s. We calculate the overall throughput for each $\mathbf{X}_i$ and choose the one that maximizes the throughput. To calculate throughput, we need to determine the batch size of each pipeline. While the global batch size is given by the user, it is Oobleck's responsibility to distribute them across heterogeneous pipelines to maximize overall throughput. It is crucial to assign work proportional to the amount of computing power of each pipeline; otherwise, the overall throughput will be decreased due to stragglers. We refer to this as *batch distribution*. Given the global batch size B and microbatch size b, batch distribution calculates the number of microbatches for each pipeline that minimizes stragglers.

Let $N_{b,i}$ be the number of microbatches for i-th pipeline ($0 \leq i < x,\ x = \sum_{j=0}^{p-1} x_j$) and T_i be the iteration time of the pipeline with a single microbatch of size b. Minibatch size for i-th pipeline can be calculated as $N_{b,i} \times b$. By adjusting $N_{b,i}$, we minimize variance between different pipelines' batch processing times. We formulate it as an integer optimization

problem:

$$\begin{aligned} \min \quad & \sum_{i=0}^{x-1} (N_{b,i} T_i - \overline{N_b T})^2 \\ \text{s.t.} \quad & \sum_{i=0}^{p-1} N_{b,i} b x_i = B \\ & N_{b,i} \in \mathbb{N} \end{aligned} \tag{2.6}$$

where $\overline{N_b T}$ is the average iteration time of all $(0 \leq i < x)$ pipelines. Any integer nonlinear optimization solver can be used to get $N_{b,i}$ and thus minibatch size for each pipeline.

Note that the optimization may fail to redistribute batch properly, primarily when the global batch size is too small and cannot be split to integers. Oobleck does not change the global batch size arbitrarily in such cases. Instead, it recommends an adjusted global batch size close to the original one but distributable.

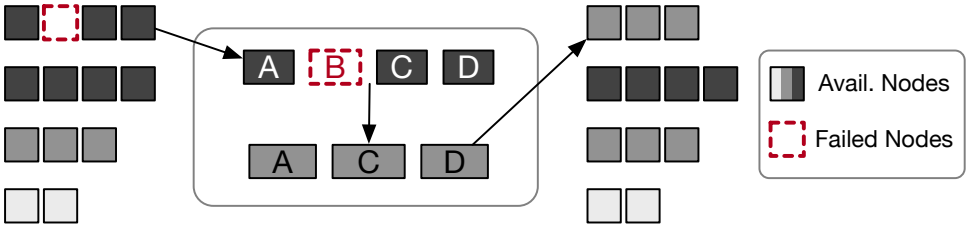
2.5 Dynamic Reconfiguration

Upon a node failure, the pipeline it was assigned to becomes incomplete and has missing model states; therefore, training halts in that pipeline. Pipelines affected by failures are replaced with new pipelines created via pipeline reinstantiation using precomputed pipeline templates (§2.5.1). After reinstantiating pipelines, the nodes copy missing layers from unaffected pipeline replicas. Oobleck also redistributes batch in response to the pipeline configuration change (§2.5.2). Provided that we have copies of the model states, Oobleck can recover from failures until we have fewer than $(f+1)n_0$ nodes.

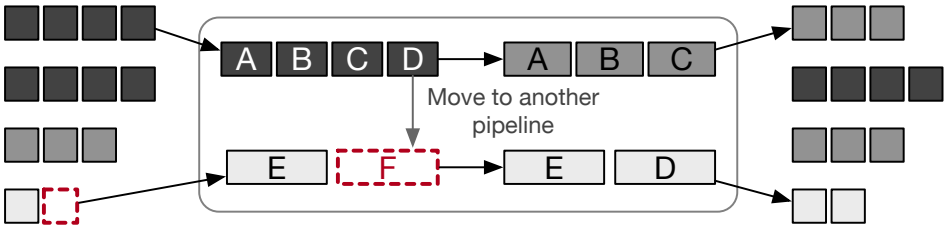
2.5.1 Pipeline Reinstantiation

Oobleck instantiates a new pipeline from one of the pipeline templates, replacing the existing one affected by failures. Given our limited number of pipeline templates, there might not be a suitable pipeline template for the remaining number of nodes. Thus, pipeline reinstantiation is done in three steps: simple reinstantiation, borrowing nodes, and merging pipelines.

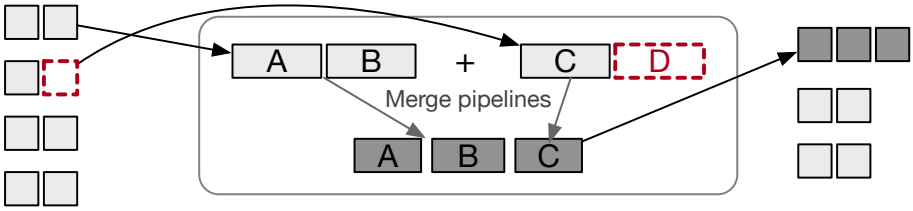
For each pipeline, Oobleck first checks if there is an instantiable pipeline template with remaining nodes; if so, Oobleck simply reinstantiates it and replaces the old one (Figure 2.8a). If there is no instantiable pipeline template with remaining nodes, Oobleck tries to *borrow nodes* from other pipelines until we have enough nodes to instantiate the smallest pipeline template (Figure 2.8b). Pipelines that yield their nodes should also be reinstantiated with fewer nodes.



(a) A node failure in a 4-node pipeline. We have a 3-node pipeline template, thus a new pipeline with 3 nodes is instantiated, which replaces the existing one.



(b) A node failure in a 2-node pipeline. Since there is no template for one node, it gets another node from another pipeline to keep the 2-node pipeline. Two affected pipelines reinstantiate or reconfigure themselves.



(c) A node failure in a 2-node pipeline. Because it cannot borrow a node from any other pipeline, it is merged with another pipeline.

Figure 2.8: Three steps of pipeline reinstantiation. After reinstantiation is done, missing layers in a new pipeline are copied from other pipelines.

After many reconfigurations and node borrowings, all pipelines may not be able to yield their nodes. When failures happen at this moment, the pipeline affected by failures cannot be reinstantiated due to a lack of nodes. In such a case, Oobleck *merges pipelines* to create a bigger pipeline (Figure 2.8c). It is guaranteed that we have an instantiable pipeline template for a merged pipeline if it has at least n_0 nodes (the minimum number of nodes to maintain one single pipeline). See Appendix A.2 for a proof.

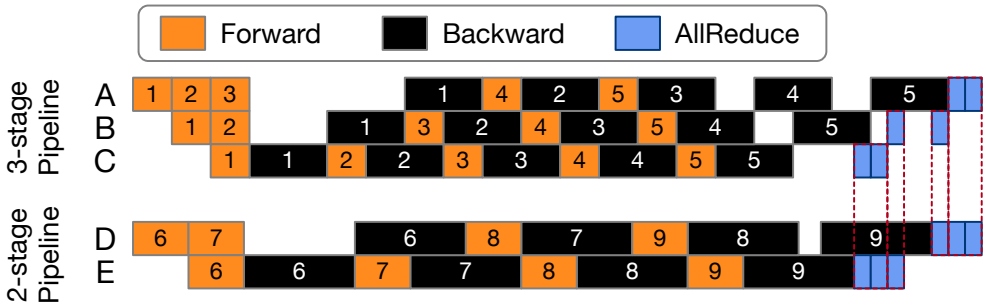


Figure 2.9: Heterogeneous pipeline execution with one 3-stage pipeline and one 2-stage pipeline. Allreduce synchronization happens at the end of iteration and done in layer granularity to communicate between multiple GPUs.

2.5.2 Batch Redistribution

After pipeline reinstantiation, execution configuration has been changed and distributed batches no longer ensure balanced execution. Oobleck runs Equation 2.6 again given the current set of the number of pipeline instances and continues training with a newly calculated batch size. Each pipeline may have more batches to compute, but the global batch size remains constant.

2.6 Implementation

We implement Oobleck in Python using PyTorch [111] and HuggingFace Transformers [155] using components from Merak [68] in using PyTorch fx symbolic tracer [126] to create pipelines and from DeepSpeed [125] to run them. For the implementation of 3D parallelism, Oobleck has integrated PyTorch Fully Sharded Data Parallel (FSDP) [176] as a replacement for tensor parallelism in each stage [122], pipeline parallelism, and data parallelism. We have used Pyomo library and its Mixed-Integer Nonlinear Decomposition Toolbox (MindtPy) solver for non-linear integer optimization [45, 12]. Here, we elaborate on the key challenges addressed in implementing Oobleck.

2.6.1 Model Synchronization Between Heterogeneous Pipelines

Model gradient synchronization between pipelines in hybrid parallelism is typically done at pipeline stage granularity. However, because heterogeneous pipelines in Oobleck have different stage configurations, stage-wise synchronization does not work. Oobleck instead breaks down stages into layers and synchronizes them individually, similar to PyTorch bucketing [111]. Figure 2.9 illustrates an example of a 6-layer model execution with two het-

erogeneous pipelines. Stage E in the 2-stage pipeline has 3 layers to synchronize which are stored in stage B and C in the 3-stage pipeline. Oobleck performs synchronization for each individual layer with potentially different peer nodes.

Data synchronization in smaller data might have performance issue because it might not fully saturate the network. We overlap communication with computation to offset increased communication latency [111].

2.6.2 Detecting Node Failures

Oobleck uses NCCL for communication between GPUs. However, NCCL cannot detect unexpected communication channel disconnection and hangs when tries to communicate with the failed node until a timer expires. To detect a node failure immediately, we launch a CPU process on each node and establish a TCP connection to a centralized CPU process. When a node dies, a socket disconnection event is triggered and broadcasted for reconfiguration.

2.7 Evaluation

We evaluate the effectiveness of Oobleck on large DNN models with 340M to 6.7B parameters and compare it against both Bamboo and Varuna. We summarize the results as follows:

- Oobleck outperforms the state-of-the-art solutions by up to 13.9× when nodes fail more frequently and matches them as failures become less frequent (§2.7.2).
- Oobleck's benefits extend to real-world settings where nodes are out and join back following spot instance traces. It outperforms the rest by up to 9.1× on average (§2.7.3).
- Ablation studies show that Oobleck's one-time planning overhead is low and it has high GPU utilization (§2.7.4).

2.7.1 Experimental Setup

Cluster setup. We evaluate Oobleck using 30 NVIDIA A40 GPUs with 40GB GPU memory each. The GPUs are connected to each other via a 200Gbps Mellanox ConnectX-6 InfiniBand adaptor for communication.

Varuna requires a remote object storage to store checkpoints for fault tolerance. We deploy a distributed object storage that consists of 6 nodes with two Intel Xeon Gold 6330 CPUs with 28 cores each, 512GB CPU memory, a 4TB PCIe 4.0 NVMe drive, and a 200Gbps Mellanox ConnectX-6 InfiniBand adaptor, respectively. We use MinIO for distributed object storage software [97].

Table 2.1: Model and batch configurations. * in Bamboo indicates the largest possible microbatch runnable in our evaluation environment. X means not runnable even with 1 microbatch size.

	# Params	Global Batch	Microbatch Size		
			Bamboo	Varuna	Oobleck
BERT-Large [27]	340M	8192	4*	32	32
GPT-2 [120]	345M	8192	1*	32	32
GPT-3 Medium [11]	350M	8192	X	16	16
GPT-3 2.7b [11]	2.7B	1024	X	2	2
GPT-3 6.7b [11]	6.7B	1024	X	2	2

Table 2.2: Throughput (samples/s) with different frequency of failures. Bamboo was not able to run any GPT-3 model due to lack of memory (OOM).

Freq.	System	BERT-Large	GPT-2	GPT-3 Med	GPT-3 2.7b	GPT-3 6.7b
6h	Bamboo	77.04	17.47			
	Varuna	260.11	86.51	29.50	7.27	4.14
	Oobleck	287.10	85.59	29.30	7.29	4.33
1h	Bamboo	75.60	17.13			
	Varuna	246.03	85.17	28.49	6.53	2.69
	Oobleck	286.28	85.42	29.21	7.23	4.22
10m	Bamboo	69.84	16.01			
	Varuna	173.52	76.39	20.19	1.76	0.26
	Oobleck	282.11	84.80	28.70	6.89	3.55

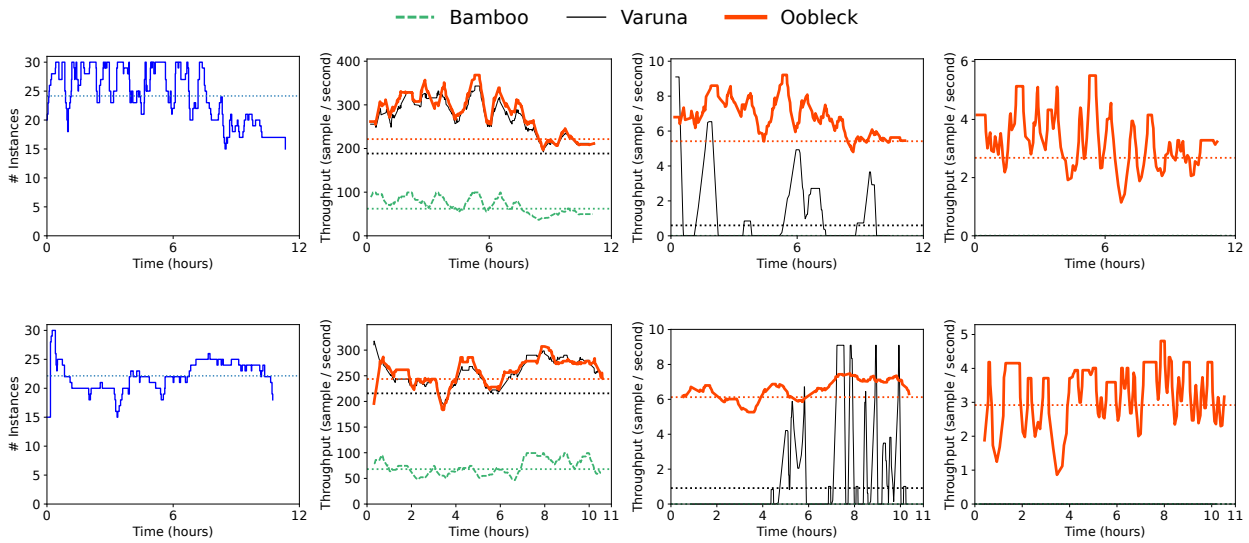
Baselines. We compare Oobleck to the following baselines:

- *Varuna* [6]: A resilient training framework based on automated parallel configuration and checkpoints [94].
- *Bamboo* [141]: A resilient training framework based on redundant computation without full restart [137].

Neither of them supports 3D parallelism; hence, Oobleck uses one GPU per node configuration to avoid its planner generating plans that cannot be implemented in our baselines.

Both works focus on utilizing spot instances, while Oobleck supports general fault tolerance including preemptions in spot instance environments. Spot instance environments have a unique mechanism of *preemption notification*; the system is notified prior to actual preemption happening. In our spot instance-based evaluation, all three frameworks leverage early warning. In general, however, there is no such notification.

For Varuna, we periodically perform synchronous checkpointing for every 10 iterations



(a) Node availability (b) BERT-Large (c) GPT-3 2.7b (d) GPT-3 6.7b

Figure 2.10: Throughput changes in spot instances environment, EC2 `P3` instances (top) and GCP `a2-highgpu-1g` instances (bottom), with various models. Note the different Y-axes scales for different models. Horizontal dotted lines represent average throughput. Bamboo could not run any GPT-3 models, while Varuna failed for GPT-3 6.7b.

following their continuous checkpointing policy [6].

Workloads. Table 2.1 lists model configurations. We adopt GPT-2 and BERT-Large from Bamboo and Varuna, and add three different configurations of GPT-3 from OpenAI [11] to verify its scalability. Although our evaluation only shows transformer models for comparison against two predecessors, Oobleck's design is not limited to transformer language models and can support other DNN models. For all evaluations, we use the Wikitext dataset [92] and TF32 precision.

We also list batch size configurations for each framework in Table 2.1. The reasons behind the discrepancy in batch sizes are twofold. First, Bamboo needs to store additional model states for redundant computation, requiring 2× memory. Second, Bamboo does not use activation checkpointing,[2] while Varuna and Oobleck do.

2.7.2 Throughput Under Controlled Failures

We first evaluate the average throughput of Bamboo, Varuna, and Oobleck on various failure scenarios. We set the frequency of failures from once every 6 hours (low rate) to once every

[2]This is because Bamboo's design choice stems from imbalanced memory consumption due to different amount of activations across stages. Activation checkpointing [66] drastically reduces memory consumption by activations and it conflicts with Bamboo's design.

10 minutes (high rate) to cover a wide spectrum of environments [141, 6, 70, 22, 130]. We monotonically reduce the number of available nodes without node recovery and measure average throughput until less than half of the nodes (15 nodes) remain.

Table 2.2 shows the average throughput for different frequencies of failures. Oobleck outperforms or matches other frameworks for every model in every scenario. Due to static overhead coming from redundant computation, Bamboo's throughput, while stable over different failure frequencies, is consistently low. Also, because they need to hold a large portion of GPU memory for an additional copy of model states for redundant computation and activations, Bamboo cannot train large models due to out-of-memory (OOM) errors.

Bamboo's gap from Varuna was surprising. We believe that it is due to differences in our evaluation environments. We use 200Gbps high-performance networking and ample NVMe storage, while the original evaluation used Amazon EC2 `p3.2xlarge` and `p3.8xlarge` instances with up to 10Gbps network and S3 object storage. High storage throughput in our setup significantly sped up Varuna. Furthermore, we use different batch configurations for Bamboo and Varuna so that each can run at maximum resource utilization; in contrast, Bamboo's evaluation used the same configuration for both, which throttled Varuna's potential throughput.

Overall, Varuna performs comparably to us when either the model is small or failures happen infrequently. For larger models and/or more frequent failures, the higher overhead of loading and saving checkpoints drastically decrease its throughput (13.9× for GPT3-6.7B).

2.7.3 Throughput in Spot Instance Traces

Next, we borrow real traces of node availability changes of spot instances from the Bamboo repository [137] and use their tools to replay the trace for 12 hours [141]. Events in the trace had been gathered from Amazon EC2 `P3` spot instances (`p3.2xlarge` and `p3.8xlarge`) and Google Cloud Platform (GCP) `a2-highgpu-1g` spot instances. Node preemption events happen every 7.7 minutes and 10.3 minutes, on average, for EC2 and GCP spot instances, respectively. Unlike experiments earlier where the number of available nodes monotonically decreases (§2.7.2), these traces include node addition events too. The actual experiments took place in our cluster where we simulated the availability events.

Figure 2.10 represents throughput changes for some models. See Appendix A.3 for results from other models; omitted models are similar to the BERT-Large model. Note that each data point in lines is an average throughput of a short time window for visibility. As such, it may not represent 0 throughput, which happens during reconfiguration or full restart. BERT-Large, the smallest model, has the least amount of checkpointing overhead; as such,

Table 2.3: Oobleck planning latency (in seconds) with various numbers of layers and nodes. BERT-Large, GPT-2, and GPT-3 Medium have 24 layers, while GPT-3 2.7b and 6.7b have 32 layers.

# Nodes	# GPUs Per Node	# Layers 24	32	64	96
8	1	0.28	0.71	9.65	68.50
	4	0.41	1.15	11.58	74.56
	8	0.54	1.50	20.98	109.76
16	1	3.37	7.45	66.35	540.36
	4	4.56	10.41	108.10	649.67
	8	4.90	11.78	176.04	1,213.63
24	1	11.35	30.11	262.47	1,477.54
	4	14.78	45.80	472.53	2,153.84
	8	15.59	49.25	520.08	3,297.92

the performance of Varuna is similar to Oobleck. However, as models become larger, even in our high-performance storage setup, Varuna takes increasingly longer to store and load checkpoints. Also, it starts suffering from fallbacks because it fails to finish checkpointing within the preemption grace period, decreasing its throughput more drastically. For example, for GPT-3 with 6.7 billion parameters, Varuna hung over the entire time and could not make training progress. Frequent changes in node availability trigger Varuna's full reconfiguration more frequently, wasting resources for saving and loading checkpoints which decreases its throughput. Bamboo cannot run any model larger than GPT-2.

2.7.4 Ablation Study

2.7.4.1 Overhead of Oobleck Planning

We run the planning algorithm (§2.4) to create a single pipeline template using various model specifications (number of layers) and node specifications (number of nodes and GPUs per node) to see its scalability. Table 2.3 shows the planning algorithm latency with various numbers of layers and nodes. Considering the estimated end-to-end training time of large models using hundreds of GPUs is ∼100 days [102], the planning overhead is marginal ($< 0.1\%$). Even if more GPUs are used for training (i.e., thousands of GPUs), the number of nodes for each pipeline template does not increase significantly. This is because Oobleck simply instantiates more of the smaller pipelines and utilizes data parallelism. Also, once a pipeline template is generated, the creation of subsequent templates can drastically be

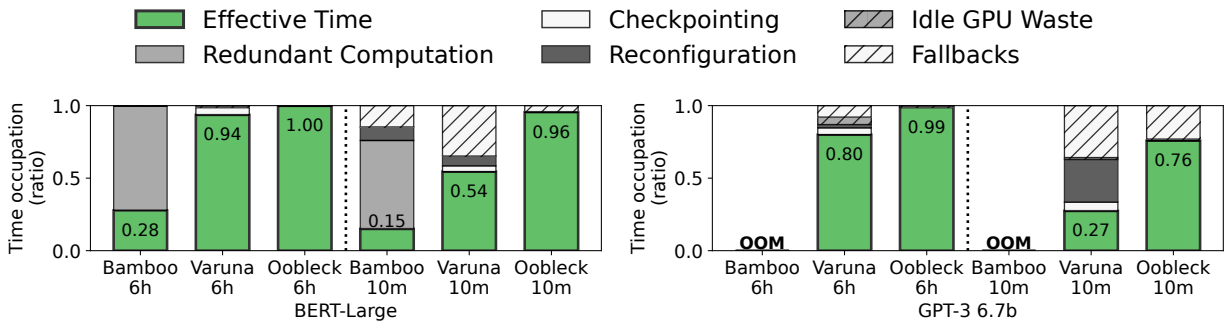


Figure 2.11: Time occupation breakdown of Bamboo, Varuna, and Oobleck running BERT-Large and GPT-3 6.7b model.

accelerated, adding negligible time, thanks to the usage of memoization and intermediate caches (§2.4.1.2).

2.7.4.2 Throughput Breakdown

Figure 2.11 shows the impact of each overhead on training throughput. Varuna's overheads include restarting overhead (reinitialization and loading a checkpoint), saving checkpoints, and throughput loss due to idle GPUs and fallbacks. Bamboo has significant overhead from redundant computation and reconfiguration overhead for data copy. Redundant computation in Bamboo is shown to have more than 50% overhead because it includes several indirect components that cannot clearly be separated – for example, pipeline bubble due to an increased number of pipeline stages to store redundant model states and imbalanced pipeline stages for balancing memory. Oobleck only has a small copying overhead for missing layers after pipeline reinstantiation.

All the frameworks experience fallback overhead, losing some training progress due to failures happening in the middle of iteration. It is more severe in Varuna because it has to fall back to the last checkpoint, while Bamboo and Oobleck lose at most one iteration. Varuna suffers significantly from much such waste occupying up to 75% of wall clock time, while Oobleck can achieve effective throughput of at least 75% of the no-failure scenario.

2.7.4.3 Impact of Checkpointing Overhead

Overhead of fault tolerance in Varuna mostly comes from serialized checkpointing and full restart. CheckFreq [98] recently introduced checkpointing optimization by pipelining checkpointing with computation, and it can improve the throughput of checkpoint-based training. Here, we go further by *completely* removing the overhead of checkpointing and analyzing the

Table 2.4: Throughput (samples/s) for Varuna, Varuna with no checkpointing overhead, and Oobleck running BERT-Large and GPT-3 6.7b.

	BERT-Large			GPT-3 6.7b		
Failure Frequency	6h	1h	10m	6h	1h	10m
Varuna	260.11	246.03	173.52	4.14	2.69	0.26
Varuna (no ckpt)	275.88	273.85	264.16	4.61	4.29	1.98
Oobleck	287.10	286.28	282.11	4.33	4.22	3.55

impact of failures only. Because of lower checkpointing overhead, we also increase the frequency of checkpointing from every 10 iterations to every 2 iterations. We define full restart overhead as framework initialization plus loading the last checkpoint overhead. While checkpointing overhead during training can be hidden by overlapping it with computation, the overhead of loading a checkpoint cannot be overlapped with computation; this is because computation cannot begin until the entire checkpoint is loaded.

Table 2.4 compares Varuna, Varuna with no checkpointing overhead, and Oobleck running the BERT-Large model and GPT-3 6.7b model. Although Varuna could increase its throughput, it still suffers from up to 60% overhead for the higher frequency of failures.

2.8 Related Work

Elastic training. Horovod Elastic [48] and TorchElastic [114] restart training upon failure and recovery. CoDDL [53] balances resource efficiency and short job priority in elastic resource sharing problems. Aryl [73] enables elastic resource sharing between inference and training workloads. Pollux [116] considers both resource utilization and statistical efficiency of training jobs when adaptively allocating resources. These works are all limited to elastic resource sharing for data-parallel training of small models that fit within a single GPU.

Distributed training with spot instances. Varuna [6] uses hybrid parallelism for distributed training with cheaper spot cloud instances. It reconfigures training when one or more failures happen. Bamboo [141] introduces redundant computation (RC) in pipeline parallelism to provide resilience in the presence of frequent preemptions of training with spot instances. Oobleck matches or significantly outperforms them for a wide range of model sizes and failure frequencies.

Large model training. Numerous proposals in recent years have attempted to optimize large model training through diverse mechanisms [102, 101, 50, 30, 138, 110, 78, 177, 122, 127, 123, 132]. However, they do not provide fault tolerance out-of-the-box and are orthogonal

to Oobleck.

2.9 Conclusion

As the scale of foundational model training continues to grow, the need for fault-tolerant distributed training becomes more pressing. This chapter introduced Oobleck, a resilient distributed large model training framework with guaranteed fault tolerance. Oobleck co-designs planning and execution for fast failure recovery and high throughput by introducing pipeline templates that are carefully designed during planning and reused during training execution. It achieves efficient failure recovery by reinstantiating pipeline(s) from the pipeline templates and copying missing model states from pipeline replicas without requiring a full restart from checkpoints. Oobleck outperforms state-of-the-art fault-tolerant distributed training solutions Bamboo and Varuna by up to 13.9×.

CHAPTER 3

Cornstarch: Efficient Distributed Training of Multimodal Large Language Models

The previous chapter addressed *resource variance* – fluctuations in the number of available GPUs during training – through pipeline templates that are model-agnostic and applicable to any model architecture. However, achieving high training throughput requires looking beyond the compute resources; the model architecture and training data themselves introduce another equally important source of inefficiency. *Workload variance* arises from the inherent heterogeneity of multimodal models, whose constituent components – e.g., pretrained modality encoders, projectors, and language models – differ substantially in architecture and the structure of the data they process. From this chapter, we move our focus to address workload variance in multimodal training.

3.1 Introduction

Multimodal large language models (MLLMs) aim to extend LLMs' reasoning capabilities to perform complex tasks across various data modalities, such as images and audio [149, 82, 83, 15, 160, 119, 18, 4, 14, 178, 169, 143, 87, 86, 1]. While MLLMs can be trained from scratch like traditional LLMs, they are more commonly constructed by integrating *pretrained* modality-specific encoders with language models [86, 174]. Each modality's input is first processed by its corresponding encoder, then projected into a shared text embedding space through learnable projection layers, and finally processed by the LLM with text tokens.

The larger size of MLLMs and the need for more data processing power make distributed MLLM training essential. However, heterogeneity in model and data makes balanced MLLM workload distribution more challenging and tackled by recent works [49, 173, 34, 58]. We observe that beyond the first-order disparities in model and data heterogeneity, there are two additional MLLM-specific distributed training challenges that have significant performance implications. First, MLLM training with *frozen* versus *trainable* models results in different

computational costs across modules. Model partitioning strategies that do not account for the frozen status of components can lead to suboptimal performance. Second, cross-modality interactions introduce non-causal attention patterns to enable more precise computation of their relationships [28, 148]. While distributing causal attention patterns in LLMs has been extensively studied [151, 85], *efficient distribution of non-causal attention patterns* remains an open challenge.

In this paper, we introduce Cornstarch, an efficient distributed MLLM training framework. Cornstarch transcends the first-order model and data heterogeneity-aware parallelization and uncovers latent higher-order heterogeneities in MLLMs that have not been considered in previous works [49, 173, 34, 58].

Frozen status-aware pipeline parallelism (§3.3.1). Cornstarch introduces a way to consider the frozen status of MLLM components in model partitioning. We observe that the frozen status of MLLM components can significantly affect the pipeline stage balancing. Existing MLLM approaches do not consider the frozen status in model partitioning [49, 173, 34]. Even with profiler-based automated approaches that actually measure the backward pass time [101, 93], they cannot account for different computational costs of modules due to the frozen status. We precisely compute the backward pass time based on the frozen status and the placement of modules to enable accurate pipeline stage balancing.

Workload balanced context parallelism for MLLMs (§3.3.2). Cornstarch introduces novel workload distribution algorithms that balance the computational cost of non-causal attention patterns at both inter-GPU and intra-GPU granularity. At the inter-GPU granularity, we compute the computational cost of each token and distribute them so that GPUs have similar amount of workload as much as possible. However, we notice that even if the GPUs have the same total amount of workload, the workload may not be evenly distributed to compute units within a GPU that leads to imbalance and performance degradation. At the intra-GPU granularity, we shard the workload within each GPU to balance the computational cost across compute units.

We have implemented Cornstarch and conducted extensive evaluations on MLLMs of varying structures, modalities, and sizes. Our evaluation results show that Cornstarch outperforms existing approaches by 2.26× on average (1.61×—3.59× across various model sizes) in training throughput.

To summarize, we make the following contributions:

- We identify higher-order heterogeneity-borne challenges in MLLMs that affect the performance of distributed MLLM training.
- We design Cornstarch, a general-purpose distributed MLLM training framework that addresses those challenges.

- Our evaluation shows that Cornstarch surpasses existing approaches by 2.26× on average in training throughput.

3.2 Background and Motivation

We first introduce 4D-parallel distributed LLM training (§3.2.1). We then enumerate the unique characteristics of MLLMs that challenge existing LLM oriented training paradigms which motivates us to design Cornstarch (§3.2.2).

3.2.1 4D Parallelism in LLM Training

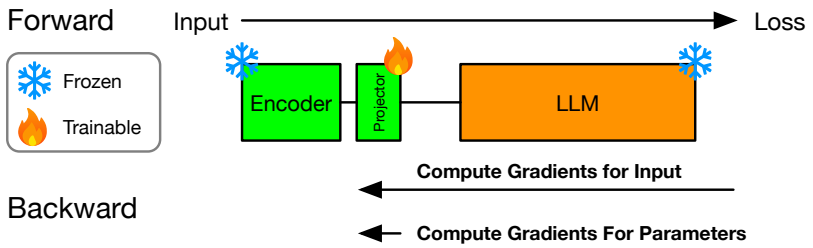
Large-scale LLM training is a well-studied topic and leverages four parallelism dimensions – tensor, pipeline, data, and context parallelism – to achieve high training throughput [102, 81, 4, 125, 64]. Tensor and pipeline parallelism (TP and PP) partitions the model within each layer and across layers, respectively. Data and context parallelism (DP and CP) partitions data; the former partitions a large batch of sequences into smaller minibatches, while the latter partitions each input sequence into segments. In all parallelization dimensions, balancing the workload across GPUs is important to achieve high throughput. In model partitioning, pipeline stages may have different amount of computation thus balancing them has been extensively studied [136, 101, 93, 177, 144]. In data partitioning, the amount of workload can be imbalance across data parallel replicas and within each replica due to LLM's causal attention pattern [36, 85, 151, 164].

3.2.2 Unique Characteristics of MLLMs

MLLMs have unique characteristics that introduce new challenges to the existing 4D parallelism.

Model and data heterogeneity. Unimodal LLMs usually contain repeated transformer layers with homogeneous structure, and unimodal text inputs go through the entire model. Unlike unimodal LLMs, MLLMs have multiple modality encoders prior to the LLM with different structures. The way of processing the input data is also different. Modality encoders first process the modality input data that they are responsible for, and then project the output to the LLM. The LLM then embeds the output of all modality encoders and text embedding, and computes the final output.

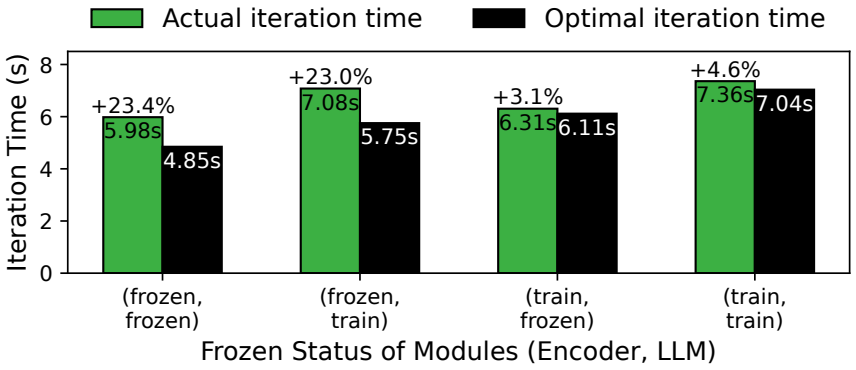
Model and data heterogeneity of MLLMs introduce imbalance in distributed training, which has been addressed by recent works. For instance, DistMM [49], DistTrain [173],



(a) The frozen status and the location of layers alters the amount of gradient computation during the backward pass.

(b) Forward and backward pass execution time of the VLM with different combination of frozen status. Activation checkpointing is enabled [66].

Trainable			Encoder Time (ms)		LLM Time (ms)	
Enc	Proj	LLM	Fwd	Bwd	Fwd	Bwd
√	√	√	44.43	120.57	140.39	375.93
×	√	×	44.26	**0.50**	138.33	**284.50**

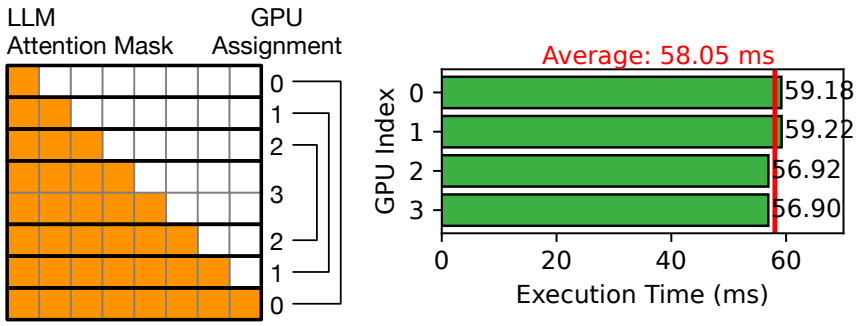


(c) Execution time of the VLM with different combination of frozen status using pipeline parallelism on 4 NVIDIA A40 GPUs. The number of microbatch is 64.

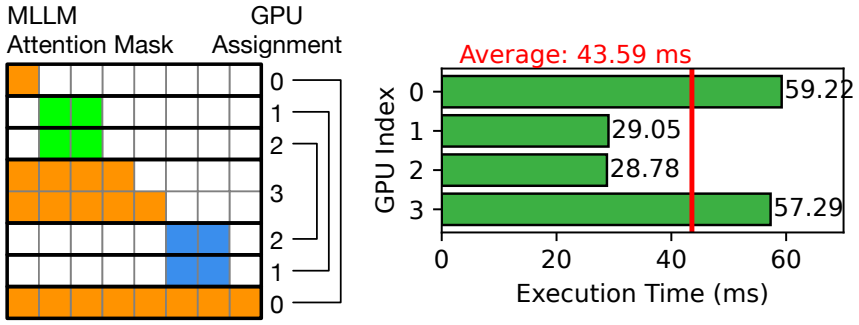
Figure 3.1: The impact of frozen status to the backward pass and the balance of pipeline stages. A VLM with Siglip (Encoder) and Llama-3.2 1b (LLM) is used.

Optimus [34], and DIP [163] disaggregate parallelism by applying different parallelization strategies to modalities to balance heterogeneous workload.

Using pretrained models with different frozen status. While unimodal LLMs are usually trained from scratch with randomly initialized parameters, MLLM training typically starts with a pretrained LLM and multiple unimodal encoders to exploit the representative capabilities of the pretrained models [86, 149, 15, 4]. Projectors between the modality



(a) Causal attention in LLM can easily be distributed evenly.



(b) Distributing MLLM attention patterns in the same way as in causal attention distribution is imbalanced.

Figure 3.2: Balanced context parallelism optimized for LLMs. It is not applicable to MLLMs.

encoders and the LLM are newly added and trained to align the embedding spaces of the modality encoders and the LLM. Usually, the modality encoders and the LLM are frozen during MLLM training and only the projectors are trained [69, 79].

The frozen status of MLLM components significantly alters the amount of computation during the backward pass, as depicted in Figure 3.1a and Table 3.1b, invalidating the long-held rule of thumb that *backward passes take roughly twice as long as forward passes* [101]. When pipeline parallelism is used, lack of considering the frozen status leads to execution time imbalance across pipeline stages. Figure 3.1a illustrates the impact of frozen status to the balance of pipeline stages. The model is partitioned to 4 pipeline stages to be balanced when all parameters are trainable, however, when the encoder is frozen, the balance between pipeline stages breaks, leading to slower execution.

Non-causal attention patterns break CP balance. In LLMs, a token attends to all preceding tokens and itself, forming a lower triangular matrix in the attention matrix, namely causal attention, as illustrated in Figure 3.2a. Most existing context parallelism works exploit the causal attention pattern [151, 164]. They partition the sequence into $2 \times \text{cp\_size}$, where

cp_size is the number of ranks in context parallelism dimension, and the i-th rank gets i-th and $(2 \times \text{cp\_size} - 1 - i)$-th chunks. This distribution guarantees balanced workload in causal attention.

However, MLLMs have non-causal attention patterns due to cross-modality interactions, as shown in Figure 3.2b, where the same distribution is imbalanced. Moreover, the attention pattern is variable depending on the input data, different from LLMs that have fixed causal attention pattern. Statically distributing the workload assuming fixed attention pattern is not applicable to MLLMs.

DCP [63] is the first work that considers dynamic non-causal context parallelism. However, its design is based on ring context parallelism [85], which has proven to be inefficient in large scale training and replaced by All-Gather based context parallelism [43, 17].

3.3 Multimodality-Aware Parallelization

Based on the observation in Section 3.2, we introduce Cornstarch's MLLM-specific parallelization strategies. We first introduce frozen status-aware pipeline parallelism (§3.3.1) and then workload-balanced context parallelism (§3.3.2).

3.3.1 Frozen Status-Aware Pipeline Parallelism

Cornstarch's frozen status-aware pipeline parallelism partitions the model into pipeline stages where the sum of one forward execution time and one backward execution time (1F+1B) is balanced across stages considering the frozen status of the layers.

However, simply considering the frozen status and adopting all-or-nothing approach – add backward pass computation time if trainable, otherwise skip – is not correct either. Even if a layer is frozen, it may still need to backpropagate gradients to the trainable parameters ahead of it. Backward pass computation of a layer L_l consists of two parts: gradient computation for parameters B_{wl} and gradient computation for data B_{dl} [115]:

$$B_l = B_{wl} + B_{dl} \tag{3.1}$$

If trainable parameters are located ahead of a frozen layer, the frozen layer, while it can skip the gradient computation for its parameters (i.e., $B_w = 0$), needs to *backpropagate input gradients* to the trainable parameters (i.e., $B_d \neq 0$), so that the trainable parameters can

update themselves. Then, the backward pass execution time B_l can be computed as:

$$\begin{aligned} B_l =& (B_{wl} \text{ if } f(L_l) \text{ is False else } 0) \\ &+ (B_{dl} \text{ if } p(L_l) \text{ is True else } 0) \end{aligned} \tag{3.2}$$

where $f(L_l)$ returns the frozen status of the l-th layer L_l and $p(L_l)$ returns whether the l-th layer L_l has trainable parameters ahead of it. $p(L_l)$ exhibits forward propagation; once it is set to True at some layer that is trainable, all the layers after it need to have $p(L)$ True to backpropagate gradients. Thus, $p(L_l)$ can be computed as:

$$\begin{aligned} p(L_l) &= \begin{cases} \text{True} & \text{if } p(L_{l-1}) \text{ is True or } f(L_l) \text{ is False} \\ \text{False} & \text{otherwise} \end{cases} \\ p(L_0) &= \text{not } f(L_0) \end{aligned} \tag{3.3}$$

Checking trainable parameters ahead of the layer should also be done across modalities. If some parameters in the modality encoders are trainable, all layers after it in the same modality as well as the LLM need to backpropagate gradients, since the LLM sits after the modality encoders.

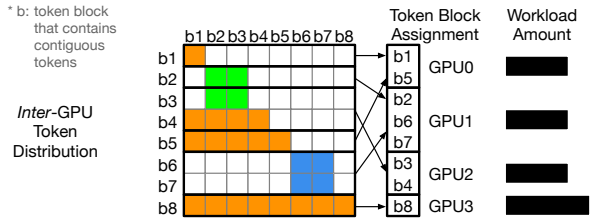
The per-layer cost $T_l = F_l + B_l$ derived from these rules is then used to partition the model into K balanced pipeline stages, minimizing the bottleneck stage cost. Any off-the-shelf partitioning algorithms – such as dynamic programming [177] or divide and conquer [57] – can be used to partition the model into pipeline stages, as long as it accepts the per-layer cost T_l as input.

3.3.2 Token Workload-Balanced Context Parallelism

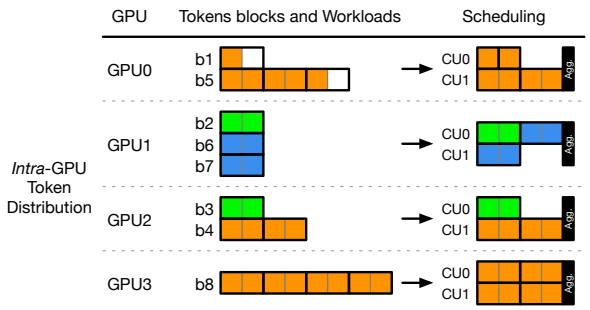
In multimodal context parallelism, many non-causal attention masks can be generated [74, 25, 148, 28], which the existing token distribution for LLMs fails to balance. We observe that to achieve genuine workload-balanced context parallelism, workload distribution across and within GPUs should be considered simultaneously. We call them *inter-GPU* and *intra-GPU* workload balancing and discuss in Section 3.3.2.1 and Section 3.3.2.2, respectively.

3.3.2.1 Inter-GPU Workload Balancing

Inter-GPU workload imbalance indicates that the amount of workloads distributed to each GPU is not balanced. This is because modern attention implementations introduce variations in the amount of computation per token [21, 20, 28]. They partition tokens into blocks



(a) Inter-GPU workload balancing. Tokens are distributed across GPUs.



(b) Intra-GPU workload balancing. Each token has at most 8 blocks to compute. These blocks are partitioned into subblocks of up to 2 blocks each, creating at most 4 subblocks per token that are scheduled to CUs.

Figure 3.3: Two-step (inter-GPU and intra-GPU) workload balanced context parallelism.

and skip block computations for efficiency if the corresponding block is completely masked-out. The amount of workloads to compute attention output per query token block can be computed by counting the number of colored blocks *rowwise*. In Figure 3.3a, for example, the workloads of 8 blocks are 1, 2, 2, 4, 5, 2, 2, 8, respectively, which are varied and irregular. We therefore propose a new method of distributing the tokens across GPUs based on the amount of computations, which we call *inter-GPU workload-balanced distribution*.

Problem formulation. We first formulate the problem as an integer linear programming (ILP) problem as follows:

$$\begin{aligned} \min_{x,C} \quad & C \\ \text{s.t.} \quad & \sum_{g=1}^{G} x_{i,g} = 1, \qquad i = 1, \ldots, T \\ & \sum_{i=1}^{T} W_i \cdot x_{i,g} \leq C, \quad g = 1, \ldots, G \\ & x_{i,g} \in \{0, 1\} \end{aligned} \tag{3.4}$$

Here, $x_{i,g}$ is a binary decision variable that indicates whether token i is assigned to g-th GPU over G GPUs. W_i represents the workload of i-th token x_i, which can be computed by row-wise sum of unmasked part of the attention mask that needs computation. The linear programming balances workload by minimizing the completion time C, which is the maximum workload assigned to any GPU.

Weighted makespan minimization. For a long sequence, the ILP problem is intractable

Algorithm 1 Token workload-balanced context parallelism algorithm.

Input: Tokens T, block size N_B, # GPUs G, and attention mask A
Output: Token assignment to GPUs $X_0, X_1, \ldots, X_{G-1}$
$B \leftarrow$ partition T into blocks of size N_B
for $b \in B$ **do**
    $W_b \leftarrow$ # blocks to compute in attention A for b
$L \leftarrow$ `minheap()`, $X \leftarrow$ `dict()`
for $g \in 0, \ldots, G-1$ **do**
    `L.heappush(`$g, 0$`)`
    $X[g]$ `= list()`
for $b, W_b \in B, W$ **do**
    $g, W[g] \leftarrow$ `L.heappop()`
    $X[g]$`.append(`b`)`
    `L.heappush(`$g, W[g] + W_b$`)`
return X

in real-time during training, thus we adopt the greedy Longest-Processing-Time-First (LPT) algorithm to assign tokens to GPUs in a context parallelism group for fast and efficient distribution [42]. Algorithm 1 shows an adapted LPT algorithm that considers the characteristics of parallel accelerators that compute with a large amount of data. We first partition the tokens into blocks of size N_B (e.g., 128). For each token block, we count the number of blocks to compute to measure the workload of the token. If the corresponding attention mask block is full of zeros, the block is skipped. We then use the LPT algorithm to assign the token block to the GPU with the least amount of workload assigned so far.

The longest processing time in the worst case has proven to be $\sum_{i=0}^{T-1} \frac{t_i}{G} + t_{\max}$, where i-th token's amount of attention computation is t_i, total number of tokens T, and the number of GPUs G [42]. As T increases, $\sum \frac{t_i}{G}$ dominates the processing time, and it is getting closer to the perfectly balanced distribution. It requires $O(GTlogT)$ time complexity, where $TlogT$ is consumed by sorting the tokens in descending order of their workloads.

3.3.2.2 Intra-GPU Workload Balancing

Even with inter-GPU workload balanced distribution, which evenly distributes the *total amount of computation* across GPUs, architectural characteristics of GPUs and implementation of attention can still lead to imbalanced execution when the jobs are dispatched to compute units (CUs).

Revisiting modern attention implementations [96, 117, 21, 20, 28], they are designed to avoid unnecessary memory accesses as much as possible. CUs use online softmax algorithm and compute the final attention output of a single query token block by keeping the inter-

mediate output in the cache and iterating over the entire key and value blocks in a single kernel. This minimizes the number of memory accesses by not writing intermediate variables to global memory.

However, assigning attention computation of a block *as a whole* to a CU introduces imbalanced amount of workload across CUs. In Figure 3.3b, for example, b1 and b5 assigned to GPU0 are executed in parallel on CU0 and CU1, respectively. While the amount of computation of b1 (1 block) and b5 (5 blocks) are extremely different, computing b5 cannot be parallelized across CUs; thus, CU0 has to wait for CU1 to finish before proceeding to the next kernel execution.

We observe that the idea of blockwise parallel attention, which was originally designed to parallelize attention across multiple accelerators, can also be used to balance the workload across CUs in a single GPU [84]. We adopt it for intra-GPU workload balancing, where the attention computation of a single set of query tokens is split into multiple subblocks and scheduled in parallel. Figure 3.3b, for example, partitions attention computations to subblocks of size 2. Each kernel computes partial attention output for a single subblock and writes it to local memory. Then, we launch an additional aggregation kernel that gathers the local attention outputs and computes the final attention output for all query blocks. Unlike the original attention computation, which writes the final attention output to memory, our output is local attention output per subblock that needs to be aggregated. The size of blocks (e.g., 2 subblocks per block in Figure 3.3b) affects the performance; with smaller subblocks, workloads are more balanced, but more local outputs should be written to the global memory, and then read again for aggregation. Large subblocks, on the other hand, have less overhead of aggregation but more imbalance. We empirically find that using 16—32 subblocks per block achieves good performance.

3.4 Implementation

Cornstarch is implemented in around 26k new Python SLOC on top of PyTorch 2.6.0 [111], HuggingFace Transformers 4.51.0 [155], and Colossal-AI 0.4.6 [76]. Cornstarch's model partitioning, scheduling, execution, communication, and checkpointing are implemented upon Colossal-AI interface. Cornstarch supports various model families and model sizes so that users can train mode than 10,000 different combinations of MLLMs. All unimodal models in the supported model families available in the HuggingFace hub can be used in creating an MLLM [51]. See Appendix B.2 for the list of supported models.

3.4.1 Implementation of Pipeline Stage Partitioning

Cornstarch assigns each layer a per-layer cost from Section 3.3.1 and partitions the model into S contiguous pipeline stages. The partitioner minimizes the bottleneck stage cost – the 1F+1B time of the slowest stage – which is the standard objective for pipeline stage balancing [177, 57]. We use dynamic programming to solve the contiguous pipeline stage balancing problem. We index layers from 0 to $N-1$. Let T_ℓ be the (frozen-aware) 1F+1B execution time of layer ℓ, where each stage cost is modeled as the sum of per-layer costs assigned to that stage. Let $P_i = \sum_{\ell=0}^{i-1} T_\ell$ be the prefix workload of the first i layers (so $P_0 = 0$ and $P_N = \sum_{\ell=0}^{N-1} T_\ell$). We define $D(i, s)$ as the minimum possible bottleneck workload when the first i layers are partitioned into s contiguous stages:

$$D(i,s) = \begin{cases} P_i, & s = 1, \\ \min\limits_{s-1 \le j < i} \max\left(D(j, s-1),\, P_i - P_j\right), & s > 1. \end{cases} \tag{3.5}$$

In the recurrence, j denotes the boundary before the last stage (i.e., the first $s-1$ stages cover layers 0 through $j-1$). For a fixed boundary j, $D(j, s-1)$ is the best (minimum) possible bottleneck workload among the first $s-1$ stages, while $P_i - P_j$ is the workload of the last stage (layers j through $i-1$). The maximum of these two terms is the bottleneck workload of the resulting s-stage partition, and the recurrence chooses the boundary that minimizes this bottleneck.

Cornstarch stores the minimizing boundary for each subproblem $D(i, s)$, so the optimal bottleneck stage workload for the full model is $D(N, S)$. To obtain the partition, Cornstarch backtracks from $D(N, S)$: if the optimal split for (i, s) is $j^\star$, then the last stage contains layers $j^\star$ through $i-1$, and the algorithm recurses on the prefix subproblem $D(j^\star, s-1)$ until $s = 1$.

Cornstarch's frozen-aware pipeline parallelism is orthogonal to the choice of this partitioning algorithm. The dynamic programming recurrence does not need to know which layers are frozen; it only consumes the per-layer costs T_ℓ. The key difference is how Cornstarch computes T_ℓ: for each layer, it accounts for whether the layer performs backward computation in addition to the forward computation. Therefore, existing partitioning algorithms (dynamic programming and other contiguous balancing heuristics) can be used directly as long as they are provided these frozen-aware costs, and the resulting stage boundaries may differ from conventional estimates that assume full forward+backward work for every layer.

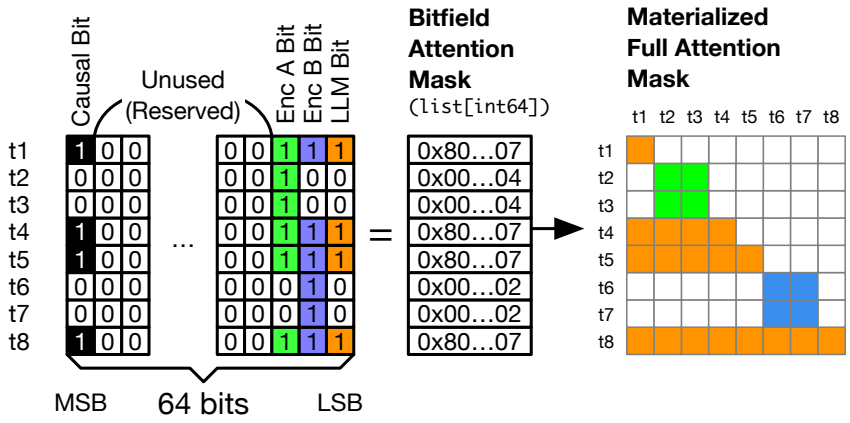


Figure 3.4: Bitfield attention mask representation.

3.4.2 Implementation of MLLM Attention

Cornstarch implements *bitfield attention* in Triton [142, 105] for high performance non-causal attention execution. Naive attention implementation computes attention scores for all tokens, and then applies the attention mask as a whole to the attention scores. This is inefficient as it requires high memory bandwidth and is not parallelizable. Bitfield attention mask is a sparse representation of the attention mask to represent multimodal interactions into attention patterns efficiently. Full attention mask is a very large 4D tensor (batch $\times$ # heads $\times$ sequence length $\times$ sequence length), which needs too much memory for long sequences. Bitfield attention mask is a 2D 64-bit integer tensor (batch $\times$ sequence length), where each bit represents which modalities the token at that position needs to attend to. Figure 3.4 shows an example of a bitfield attention mask. We assign bits from the least significant bit (LSB) to the most significant bit (MSB) to the modality encoders and the LLM. The LSB (1st index) is assigned to the LLM, and 2nd and 3rd bits are assigned to the modality encoders A and B, respectively, for example. The most significant bit (64th index) is reserved for causal bit; when this bit is set to 1, the token attends to all of its previous tokens. For example, tokens `t2, t3` are tokens from the encoder A, thus have 2nd LSB set to 1. As it does not have causal bit, it attends to the other tokens only from the encoder A. `t4, t5`, however, are text tokens with causal bit and all modality bits set to 1. Thus it can attend to all modality tokens, but only previous tokens. Cornstarch bitfield attention implementation is compatible with context parallelism (§3.4.3), while standard FlashAttention does not natively support.

3.4.3 Implementation of Context Parallelism

There are various ways of implementing context parallelism: all-to-all [54], ring attention [85, 63], and All-Gather based [4, 17], etc. Cornstarch implements the SOTA All-Gather based context parallelism implementation. This implementation gathers all keys and values of all tokens and compute row-wise attention for local queries. Overlapping communication and computation is done in the head dimension; while GPUs compute attention for one or a few heads, it transfers keys and values for the next head(s). This simplifies Algorithm 1 in computing per-token workload. If we adopt P2P ring attention, it would have been more complicated to compute per-token workload as it requires to recompute the amount of workloads every round.

3.5 Evaluation

In this section, we evaluate Cornstarch and show its effectiveness in training MLLMs. Our key results are:

- Cornstarch achieves 2.26× higher end-to-end training throughput on average for MLLM training (§3.5.2).
- Frozen status-aware pipeline parallelism partitions MLLMs more effectively by considering the frozen status and provides up to 2.46× faster iteration time in MLLMs (§3.5.3).
- Workload-balanced context parallelism distributes tokens more evenly across GPUs and within a single GPU, which improves the performance of attention execution by up to 1.18× (§3.5.4).

3.5.1 Experimental Setup

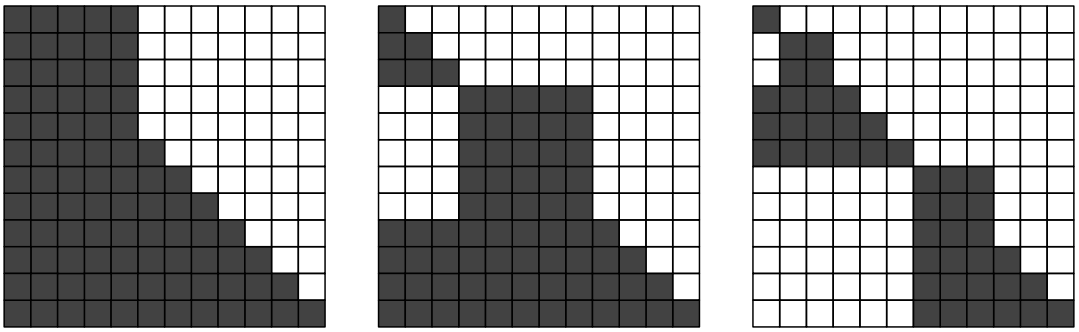
Testbed. We run our evaluation workloads in a GPU cluster with 6 nodes, each with four NVIDIA A40-48GB GPUs and a NVIDIA Mellanox ConnectX-6 200Gbps Infiniband adaptor (total 24 GPUs). The four GPUs in a node are connected in pairs using NVLink and connected to the node via PCIe 4.0.

Baselines. We set the baselines as follows:

1. *FSDP*: FSDP is widely used in distributed MLLM training thanks to its ease of use [15, 47, 87, 86]. It shards parameters and distributes them across all GPUs to reduce memory footprint. Parameters are temporarily gathered for computation and then sharded again. We use FSDP2, which offers higher performance [81].
2. *Megatron**: Megatron-LM extends LLM pipeline parallelism to MLLMs by adding a

Table 3.1: Modality (LLM, vision, and audio) configurations.

Model Arch.	Model Size	# Layers	Hidden Size	# Params
Llama-3 (LLM)	Small	16	2048	1b
	Medium	32	4096	8b
	Large	64	5120	32b
Qwen2 Vision	Small	32	1280	0.6b
	Medium	48	2560	3.9b
	Large	64	3840	11.6b
Phi4 Audio	Small	24	1024	0.5b
	Medium	32	3072	3.4b
	Large	48	5120	12.4b



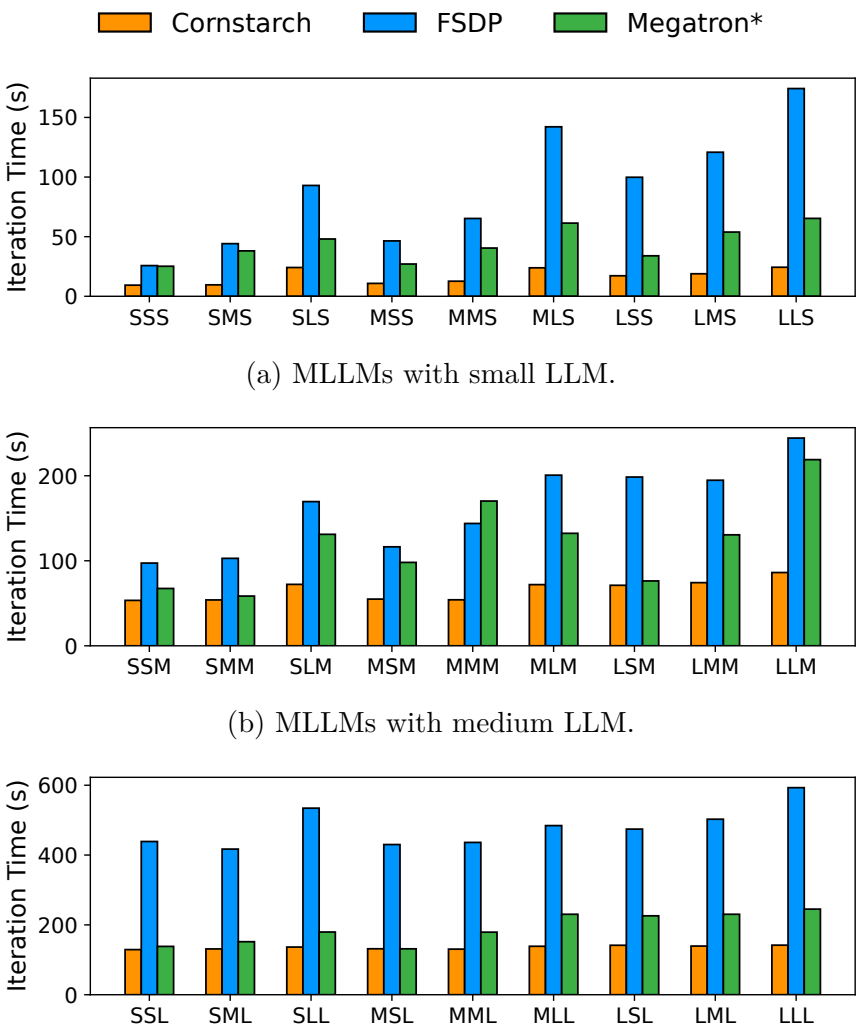
(a) Causal prefix. (b) Encoder outputs embedded. (c) Multimodal packing.

Figure 3.5: Various attention masks used in MLLM training.

vision encoder as the first pipeline stage [104]. We chose Megatron-LM as a representative of the existing LLM-optimized 4D parallelization.

Training data. We use a synthetic dataset for evaluation. Each sample consists of 1k text tokens, a 1280x720 image, and a 2-minute audio clip. Image tokens and audio tokens are projected into the text embedding space after being processed by the corresponding modality encoder. We use a global batch size of 48. FSDP uses a mini-batch size of 2, while microbatch size 4 is used in Megatron* and Cornstarch with pipeline parallelism.

We use three different types of attention patterns in our evaluation, as illustrated in Figure 3.5. Modality tokens can either be prepended or embedded, and several multimodal sequences can be packed into a single long sequence. While the number of tokens for each modality is fixed, the placement of the modality data is randomly assigned.



(a) MLLMs with small LLM.

(b) MLLMs with medium LLM.

(c) MLLMs with large LLM.

Figure 3.6: End-to-end performance comparison of Cornstarch and baselines with various model configurations.

Model configurations. We evaluate various MLLM configurations created by combining two modality encoders (vision and audio) and an LLM, each selected from the sizes listed in Table 3.1. The modality encoders are merged into a single module which processes both modalities. We freeze the merged modality encoder and the LLM and only train the projector modules. An MLLM configuration is denoted by suffixes representing the sizes (S, M, L) of its vision encoder, audio encoder, and LLM, respectively (e.g., MLLM-SML combines a small vision encoder, a medium audio encoder, and a large LLM).

3.5.2 End-to-End Performance

We first evaluate Cornstarch against the baselines in terms of end-to-end training iteration time and show the results in Figure 3.6. When models are small enough (e.g., MLLM-SSS in Figure 3.6a), FSDP shows reasonably good performance. However, as model size increases, FSDP's performance drops significantly due to intensive communication overhead. For Megatron*, its current limitations cause a large imbalance in pipeline stages due to the lack of frozen status awareness. This is observed especially well when the modality encoders are relatively larger than LLMs (e.g., MLLM-LLS in Figure 3.6a or MLLM-LLM in Figure 3.6b).

Cornstarch shows the best performance across all model configurations. It chooses better modality parallelism, allocates pipeline stages to the encoders and the LLM based on their frozen status and their placement, and balances the attention computation on the fly. We discuss the performance of Cornstarch in detail in the subsequent sections. Overall, Cornstarch outperforms the baselines 2.26× on average (3.36× vs FSDP and 1.62× vs Megatron*).

3.5.3 Impact of Frozen Status-Aware Pipeline Parallelism

We parallelize the models with frozen status-aware pipeline parallelism and compare the performance with the same models but parallelized without frozen status awareness. Table 3.2 presents the results. For brevity, we only show a few model configurations with encoders being colocated.

Without frozen status-awareness, partitioning is done based on the assumption of all parameters being trainable, which tries to minimize variance of forward time across pipeline stages. For example, MLLM-LLL, the frozen status-unaware partitioning partitions the modality encoders and the LLM to have similar forward execution time ($\sim$ 1600ms). However, gradient computations for the frozen encoders and the LLM are skipped, their backward execution time is significantly different (3.06ms and 15878.58ms), breaking the balance between pipeline stages.

With frozen status-awareness, the partitioning is balanced based on the forward execution time plus the backward execution time (5320.08ms and 7051.51ms, respectively), decreasing pipeline bubbles. We also observe a few exceptions: MLLM-LLS and MLLM-SSL. These are the most extreme cases in model size distinction, thus even assigning maximum number of pipeline stages to the large module is not enough to balance the pipeline stages. In other cases, frozen status-aware pipeline parallelism assigns workloads more evenly across pipeline stages, which improves the overall performance by up to 2.46×.

Table 3.2: Model forward and backward execution time breakdown parallelized with and without frozen status awareness.

Model	Frozen Aware	Per-Stage Fwd (ms)		Per-Stage Bwd (ms)		Iter. Time (s)	Impr. (×)
		Enc	LLM	Enc	LLM		
SSS	√	301.61	149.40	1.04	518.43	21.81	1.18x
	×	207.13	296.25	0.86	1032.63	25.67	-
MMS	√	903.86	102.61	2.72	346.54	40.34	1.02x
	×	635.56	297.62	1.19	1032.12	41.21	-
LLS	√	1240.50	298.37	1.41	1029.98	66.14	1.00x
	×	1259.13	297.92	1.15	1030.14	66.20	-
SSM	√	464.98	331.43	3.53	3017.01	66.56	1.11x
	×	388.33	388.73	2.30	4030.74	73.94	-
MMM	√	2330.90	273.89	2.09	2418.27	70.37	**2.46x**
	×	712.72	1159.76	1.60	12113.67	173.01	-
LLM	√	2199.91	376.11	4.15	4023.22	87.43	2.03x
	×	1403.97	1161.76	1.56	12109.73	177.39	-
SSL	√	773.10	741.17	3.55	6309.11	138.45	1.00x
	×	774.19	708.62	3.84	6306.18	137.62	-
MML	√	2280.58	705.73	3.11	6311.23	138.97	1.30x
	×	1015.60	1154.85	2.78	10546.34	180.28	-
LLL	√	5316.08	736.05	4.00	6315.46	143.76	1.72x
	×	1597.36	1686.28	3.06	15878.58	247.79	-

3.5.4 Impact of Workload-Balanced Context Parallelism

This section evaluates how effectively Cornstarch's workload-balanced context parallelism (§3.3.2) distributes non-causal attention execution. We run LLMs with a 64k sequence length and simulate attention masks that represent mixtures of multiple modalities. Figure 3.5 illustrates the attention masks. Early vision language models [87] use a causal prefix (Figure 3.5a), which simply prepends the encoder outputs to the input tokens. Subsequent models embed the encoder outputs in the middle of the input tokens (Figure 3.5b), with their locations specified by special tokens, e.g., `[IMG]` and `[AUD]` [86, 83, 149]. Multimodal packing (Figure 3.5c) is a more complex mask that packs multiple sequences with different masks into a single sequence. See Appendix B.3 for results with different sequence lengths.

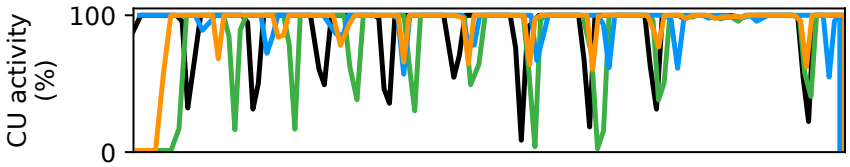
Table 3.3 shows the results of a single attention layer and the entire LLM with various

Table 3.3: Execution time of a single attention layer and entire LLM with 64k sequence length using various context parallelization policies.

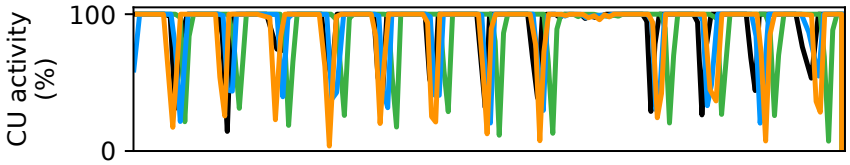
Time (ms) (Impr. (×))		Causal CP	Inter-GPU Balance Only	Intra-GPU Balance Only	Cornstarch
LLM-S	Attn	243.44	255.59	225.73	204.95
		(-)	(0.95x)	(1.08x)	**(1.19x)**
	Model	5541.25	5665.77	5250.40	4856.60
		(-)	(0.98x)	(1.06x)	(1.14x)
LLM-M	Attn	460.13	487.44	440.86	417.31
		(-)	(0.94x)	(1.04x)	(1.10x)
	Model	24534.50	25389.74	23712.89	22815.79
		(-)	(0.97x)	(1.03x)	(1.08x)
LLM-L	Attn	568.18	610.67	558.56	551.60
		(-)	(0.93x)	(1.02x)	(1.03x)
	Model	77378.44	79671.34	75055.69	74864.71
		(-)	(0.97x)	(1.03x)	(1.03x)

context parallelization policies. The numbers in the table are the average of 50 times of the attention execution time with the attention masks in Figure 3.5. We only show the results of LLM-L, as the same patterns are observed in other model sizes. Cornstarch shows the best performance, outperforming the existing causal context parallelism optimized for LLMs by up to 1.18×. Intra-GPU workload balancing also shows improvement. Even with additional overheads from aggregation, parallelizing attention subblocks within a single GPU effectively removes tail latency caused by stragglers.

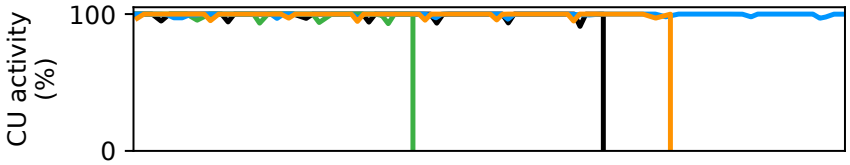
Surprisingly, however, balancing workload distribution only at a token level (inter-GPU balancing only) does not provide performance improvement. To understand this, we further perform CU activity analysis of a single attention layer, depicted in Figure 3.7. Severe downward spikes are observed in both causal context parallelism (Figure 3.7a) in inter-GPU only balance context parallelism (Figure 3.7b). The spikes happen at the end of every attention head computation. This is because attention kernel for the next head cannot be launched until the in-flight attention kernel for the current head is entirely finished, leaving CUs inactive. Intra-GPU balancing fundamentally solves this problem by distributing workloads of attention computation of each single block in finer granularity across CUs, showing higher CU activity (Figure 3.7c). Still, only balancing Intra-GPU workloads does not balance the total amount of workloads across GPUs; some GPUs become idle much earlier while others are busy, reducing overall utilization. Combining inter- and intra-GPU balancing, Cornstarch



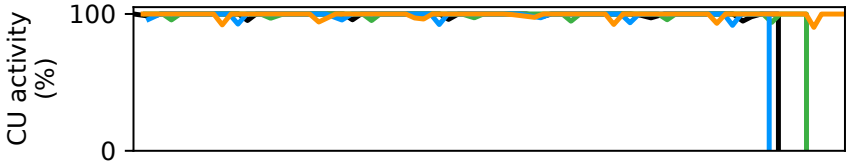
(a) Causal context parallelism.



(b) Inter-GPU balanced only context parallelism.



(c) Intra-GPU balanced only context parallelism.



(d) Cornstarch context parallelism.

Figure 3.7: CU activity analysis with various context parallelization policies running a single attention layer of LLM-L. Each line represents one GPU.

achieves the best performance (Figure 3.7d).

3.5.5 Comparison with DCP

While DCP [63] is not designed for MLLMs, its capabilities can be adapted to MLLMs, representing MLLM-specific attention patterns. We compare Cornstarch with DCP in terms of single attention layer performance and end-to-end training time on 16 GPUs. We use 4 attention patterns demonstrated in the DCP paper for comparison [63].

Table 3.4 shows the result of a single attention layer with various attention patterns. With shorter sequence length, DCP shows better performance than Cornstarch. This is because

Table 3.4: Performance comparison between Cornstarch and DCP for a single attention layer with various attention masks. Time is in ms.

(a) Lambda.

Seq length	32k	64k	128k	256k
Cornstarch	177.41	143.56	375.49	1193.32
DCP	46.36	124.72	417.07	1545.83

(b) Causal.

Seq length	32k	64k	128k	256k
Cornstarch	211.51	158.15	380.88	1189.57
DCP	54.63	149.18	503.56	1861.39

(c) Causal blockwise.

Seq length	32k	64k	128k	256k
Cornstarch	228.36	124.72	333.91	1002.70
DCP	41.13	104.87	316.93	1083.57

(d) Shared questions.

Seq length	32k	64k	128k	256k
Cornstarch	199.99	179.59	351.41	1190.41
DCP	39.79	120.74	374.99	1358.43

Table 3.5: End-to-end training time on Cornstarch and DCP.

Seq Length	Time (s)	Causal	Lambda	Causal Blockwise	Shared Questions
32k	Cornstarch	5.7	5.8	10.5	5.7
	DCP	26.5	49.0	42.0	46.5
128k	Cornstarch	37.2	37.4	29.0	33.9
	DCP	46.5	107.1	100.8	102.4

Cornstarch requires enough sequence length to fully hide the communication overhead. With longer sequence length, however, DCP rather struggles to hide the communication overhead, since it relies on ring context parallelism, which is inefficient in training with long sequence lengths or large number of context parallel degrees, while Cornstarch all-gather based context parallelism gets more benefits from longer sequences.

Table 3.5 shows a full iteration time of Llama3-1b model (planning + 1 forward + 1 backward). DCP suffers from significant planning overhead to find the optimal computation schedule. Cornstarch's heuristic token assignment is efficient, finishing planning in less than 1 second in all cases, outperforming DCP.

3.6 Related Work

4D parallelism. Large-scale LLM training combines four parallelism dimensions—tensor, pipeline, data, and context parallelism (TP, PP, DP, and CP)—to scale both model size and sequence length [102, 4, 64, 81]. TP and PP partition the model within and across

layers, respectively, while DP and CP partition training data at the batch and sequence granularity. Balancing the workload across GPUs is critical in every dimension. For model partitioning, pipeline stage balancing has been extensively studied to minimize the bottleneck stage cost [101, 93, 136, 177]. For data partitioning, context parallelism has been developed to distribute long input sequences across GPUs, with several works specifically exploiting the causal attention pattern of LLMs to guarantee balanced computation [85, 54, 43, 164, 151, 72, 10]. However, all of these techniques assume homogeneous transformer stacks and fixed causal attention, and do not address the unique characteristics of MLLMs: heterogeneous modality encoders, variable frozen status across components, and non-causal cross-modality attention patterns (§3.2).

Distributed multimodal training. Most MLLM practitioners today rely on FSDP [**?** 122] for its simplicity—sharding parameters across GPUs and gathering them on demand—but FSDP's all-reduce-heavy communication pattern limits scalability as model size grows beyond a single node [81]. To overcome this, recent works address first-order model and data heterogeneity in MLLM training. DistMM [49] applies independent parallelism strategies to heterogeneous modality modules, but is limited to contrastive learning where modalities interact only through the loss function, not mid-iteration as in LLM-backbone MLLMs. DistTrain [173] and Optimus [34] disaggregate encoder and LLM parallelism and exploit pipeline bubbles to overlap their execution. DIP [163] interleaves encoder and LLM pipeline stages to further improve utilization. GraphPipe [58] generalizes pipeline parallelism to DAG-shaped execution flows, accommodating the graph-like structure of MLLMs with multiple encoders. Despite these advances, none of these works model how frozen versus trainable components alter backward-pass computation cost, which invalidates standard pipeline stage balancing heuristics [101]. Nor do they address context parallelism under the dynamic non-causal attention patterns introduced by cross-modality interactions [28, 148]. DCP [63] is the first to handle dynamic non-causal context parallelism, but its ring-based communication is inefficient at large scale and is superseded by All-Gather-based approaches [43, 4]. Cornstarch builds on multimodality-aware 4D parallelism to address frozen status-aware pipeline partitioning and workload-balanced context parallelism for non-causal MLLM attention (§3.3).

3.7 Conclusion

In this paper, we presented Cornstarch, a multimodality-aware distributed MLLM training framework. Cornstarch addresses higher-order challenges arising from model and data heterogeneity in MLLM training. We introduce frozen status-aware pipeline parallelism that balances the computational cost of MLLM pipeline stages. We also introduce workload bal-

anced context parallelism which computes the amount of workloads both in intra-GPU and inter-GPU. Cornstarch provides 2.26× speedup over the state-of-the-art distributed MLLM training frameworks on average.

CHAPTER 4

Entrain: Addressing Variable Heterogeneity in Multimodal Training Workloads

The previous chapter addressed workload variance *within* a training iteration: the heterogeneity arising from the diverse architectures and frozen statuses of multimodal components. Yet an equally significant source of inefficiency operates *across* iterations; the ratio of computation between modalities shifts from batch to batch as the composition of training samples varies, rendering any parallel configuration derived from a single batch or small microbatch potentially suboptimal for the rest of training. Counter-intuitively, a single static parallel configuration does suffice, but only when it is anchored to the macroscopic workload distribution of the full global batch, where variability converges by the Law of Large Numbers. Pipeline parallelism, however, partitions each global batch into microbatches that are too small for this convergence to hold, and variability re-emerges at that finer granularity. This chapter introduces Entrain, which addresses cross-batch workload variability through macroscopic profiling that locks in the optimal configuration, combined with a deferral-based microbatch assignment strategy that resolves the residual intra-iteration imbalance.

4.1 Introduction

The rapid evolution of Large Language Models (LLMs) into Multimodal LLMs (MLLMs) [107, 108, 39, 149, 162, 86, 71, 89, 157] has made efficient distributed training critical, yet balanced workload distribution remains particularly challenging due to the inherent *heterogeneity* and *variability* of multimodal datasets. Heterogeneity stems from the distinct computational requirements and arithmetic intensities of different modalities. Variability arises because modality workload proportions fluctuate drastically across samples, with each modality following an independent distribution. In text-only LLM training, recent works address sequence length variability by dynamically adapting the model-parallel

configuration on the fly [37, 35]. A natural intuition would be to extend this dynamic reconfiguration to MLLMs.

Unfortunately, because each modality exhibits independent variability yet remains tightly entangled within a single sample, per-sample dynamic reconfiguration is prohibitively expensive. Our analysis reveals that dynamic adaptation is fundamentally unnecessary. While multimodal workloads are highly chaotic at the microscopic (sample or microbatch) level, the aggregate workload ratio between modalities reliably converges to a stable constant at the macroscopic scale of a global batch. This convergence implies that a single, static model-parallel configuration suffices for optimal load balancing, provided it is anchored to the global distribution. This exposes a critical shortcoming in existing MLLM training systems [49, 173, 34, 163], which derive their static model-parallel configurations by profiling a non-representative slice (e.g., a single sample or a small microbatch). Profiling at a microscopic scale dominated by extreme sample-level variance captures a distorted snapshot of the dataset. Consequently, these configurations inevitably suffer from severe load imbalances and pipeline bubbles when processing the full, highly variable dataset.

To exploit this macro-level stability while managing the unavoidable micro-level pipeline execution imbalance, we propose Entrain, a distributed MLLM training system that addresses multimodal variability at both macroscopic and microscopic scales. At the macro level, Entrain shifts the profiling paradigm from isolated single samples or microbatches to the entire global batch. Such macro-level profiling accurately captures the converged computational ratio between the various modalities and the core LLM. Unlike prior works that blindly extrapolate brittle configurations from arbitrary, high-variance snapshots, Entrain mathematically anchors its parallel configuration to the globally converged workload distribution. Using the Law of Large Numbers, we prove that a single, static configuration derived from this macroscopic ratio suffices to guarantee optimal load balance throughout training. This sidesteps both the brittle configurations of micro-level profiling and the prohibitive overheads of dynamic reconfiguration.

While the macro level profiling establishes a stable global baseline, partitioning the global batch into discrete microbatches for execution inevitably re-exposes localized variability. Because each microbatch is much smaller than the global batch, the Law of Large Numbers no longer holds at this granularity, and workload variability across microbatches persists. Entrain addresses this through a decoupled microbatch assignment strategy: it treats the MLLM data flow as a producer-consumer pipeline and decouples the scheduling constraints of each modality to resolve imbalance. With encoders as producers and the LLM as the consumer, Entrain ensures constant production and consumption rates independently, balancing the execution time of modality pipeline stages while avoiding buffer stalls. Our hierarchical

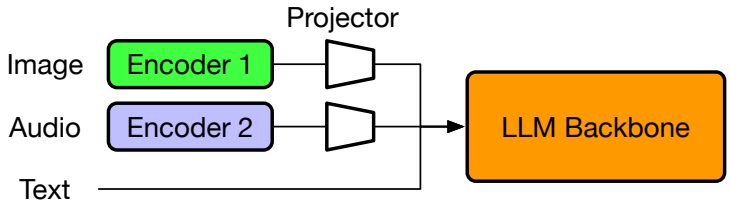


Figure 4.1: Multimodal LLM architecture.

microbatch assignment algorithm first achieves a constant production rate by distributing encoder workloads evenly across microbatches, then balances the LLM workload by deferring excess workload from highly variable samples.

We have implemented Entrain on top of PyTorch and Cornstarch [56]. We evaluate Entrain on vision-language models (based on Qwen2.5Vision with Llama3-1b and 3b models) across four multimodal datasets with distinct distributions. Compared to DistTrain [173] and DIP [163], Entrain reduces workload variability across microbatches by up to 10.6×, improving end-to-end training throughput by up to 1.40×.

In summary, we make the following contributions:

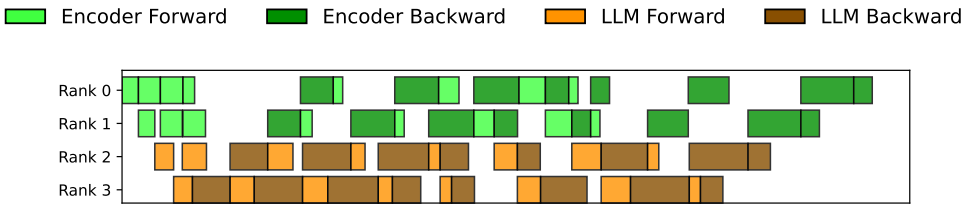
- We propose Entrain, the first distributed training framework to address both heterogeneity and variability in multimodal training workloads via batch-level profiling and decoupled microbatch-level workload balancing.
- We introduce a profiling approach that targets the global batch instead of micro-level samples to derive the optimal model-parallel configuration.
- We design a hierarchical microbatch assignment algorithm that defers excess workload within each iteration to stabilize workload variability across microbatches.

4.2 Background and Motivation

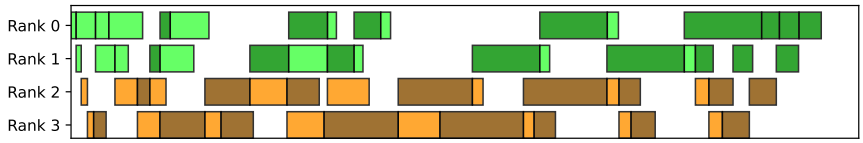
We first introduce MLLM architecture and parallelism (§4.2.1), then describe the dataset characteristics that make balanced workload distribution challenging (§4.2.2), and finally discuss the limitations of existing approaches (§4.2.3).

4.2.1 MLLM Architecture and Parallelism

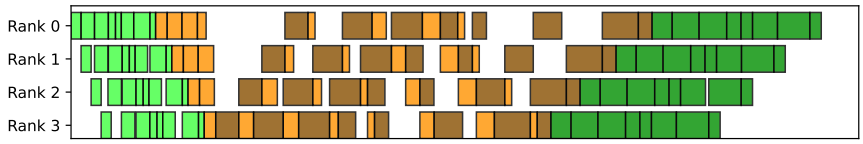
To efficiently train massive MLLMs, modern distributed systems employ 4D parallelism, combining data (DP), tensor (TP), context (CP), and pipeline parallelism (PP) [77, 17]. While these dimensions effectively scale the model across devices, the inherently hetero-



(a) 1F1B pipeline parallel schedule.



(b) DistTrain [173] pipeline parallel schedule. Same as 1F1B but with different microbatch order.



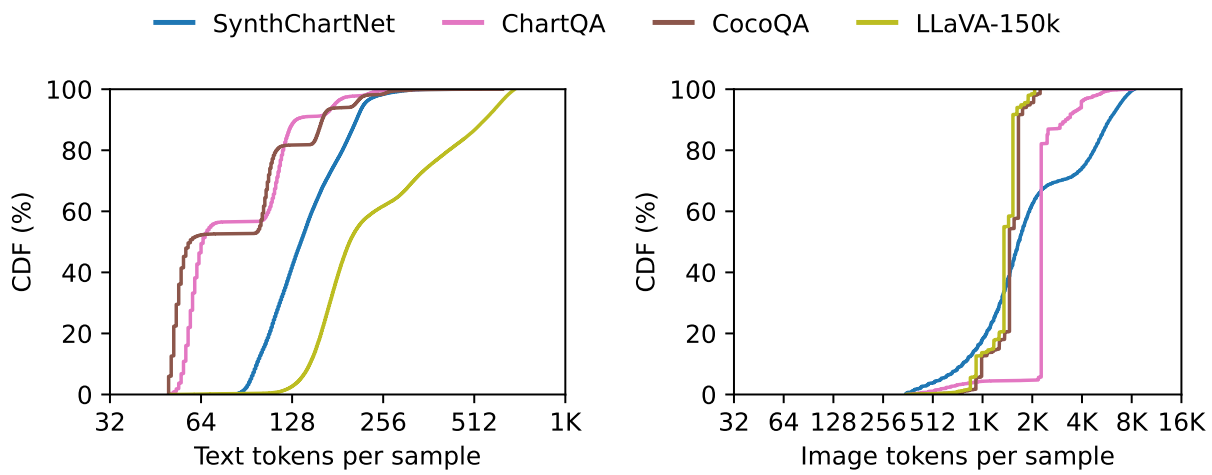
(c) DIP [163] pipeline parallel schedule.

Figure 4.2: Visualization of different pipeline parallel schedules using 8 microbatches for a vision language model (VLM).

geneous architecture of MLLMs requires careful parallelization to balance computational workloads across the cluster.

MLLMs enforce a strict structural dependency: raw multimodal inputs must first be processed by modality-specific encoders (e.g., Vision Transformers for images, Whisper for audio), whose outputs are then projected into a unified embedding space for the core LLM backbone, as shown in Figure 4.1. Under PP, this architectural separation places encoders on earlier pipeline stages and the LLM on subsequent stages. To maintain high throughput, the global batch is partitioned into microbatches that are injected sequentially.

This pipeline design requires strict temporal and spatial balance: execution times must remain consistent across consecutive microbatches, and all stages must have roughly equal execution times for a given microbatch. Violations in either dimension immediately produce pipeline bubbles and stragglers. Figure 4.2a shows the standard 1F1B schedule for a vision-language model (VLM), where vision encoder stages precede the LLM stages. DistTrain [173] (Figure 4.2b) uses the same schedule but reorders microbatches to reduce pipeline bubbles. DIP [163] (Figure 4.2c) parallelizes modalities independently and colocates their pipeline stages.



(a) Number of text tokens

(b) Number of vision tokens

Figure 4.3: Distributions of number of vision and text tokens in various datasets. They are independently varying.

4.2.2 MLLM Dataset Characteristics

While a carefully tuned parallel configuration can theoretically balance the heterogeneous architecture of an MLLM, maintaining this balance at runtime requires perfectly balanced data partitions. However, two inherent dataset characteristics severely disrupt this: workload heterogeneity and variability.

Workload heterogeneity refers to the systematic difference in computational characteristics across modalities. Modality-specific encoders and the LLM have fundamentally different structures; processing a high-resolution image through a vision encoder exhibits drastically different arithmetic intensity and memory access behavior from processing text tokens through the LLM. *Workload variability* refers to the sample-to-sample fluctuation in how much of each modality appears. Unlike unimodal text datasets, where sequence length is the dominant source of variation, each modality in an MLLM dataset follows its own workload distribution. Figure 4.3 plots the CDFs of text and vision tokens across several vision-language datasets, confirming that each modality varies independently.

The challenge is further compounded by an intrinsic coupling constraint: modalities are bound within each sample and must be processed together. The inter-modality workload ratio is therefore determined jointly per sample rather than controlled independently. Figure 4.4 shows that this per-sample ratio between the vision encoder and the LLM fluctuates drastically, spanning a wide range with no stable central tendency, making any fixed GPU partition inherently mismatched for the vast majority of samples it encounters.

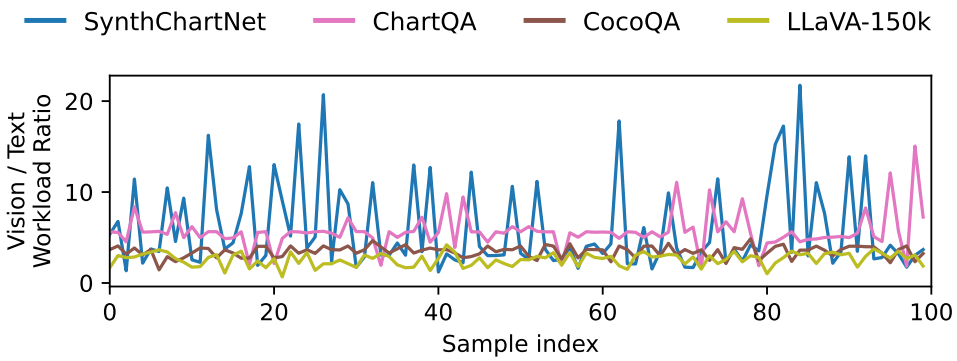


Figure 4.4: Workload ratio of vision encoder (Qwen2Vision) and LLM (Llama3-1B) across 100 samples in datasets.

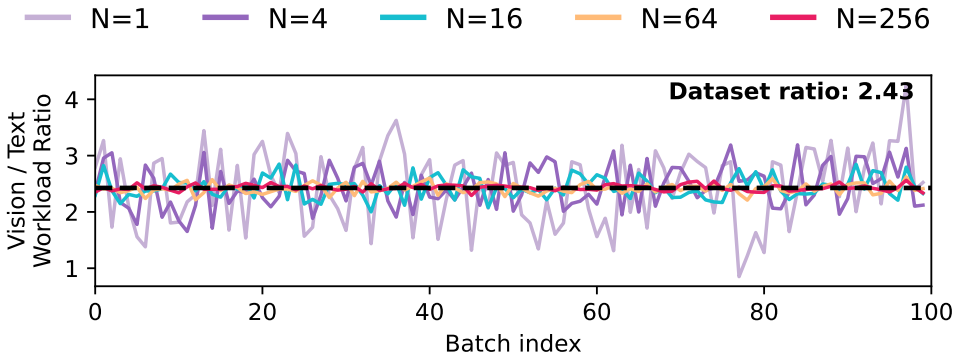


Figure 4.5: Workload ratio variability with different sample sizes in LLaVA-150K dataset. The mean ratio of vision-to-text workload of the entire dataset is 2.43. With larger sample size N, the ratio between batches becomes more stable and converges to the dataset mean.

4.2.3 Limitations of Existing Works

Existing MLLM training systems [34, 173, 163] recognize that workload imbalances arise at two granularities – profiling and execution – but no single system addresses both.

Profiling granularity. DistTrain [173] and DIP [163] derive parallel configurations by profiling a single sample or a small microbatch, which may not represent the entire dataset. The high per-sample variance suggests that dynamic reconfiguration – as applied to sequence-length variability in LLM training [37, 35] – is the right approach. Counterintuitively, however, Figure 4.5 shows that the workload ratio converges to a stable dataset mean as batch size increases, justifying a static configuration – but only when derived from this macroscopic ratio rather than an unrepresentative micro-sample.

Execution granularity. Even with a correct parallel configuration, the workload ratio con-

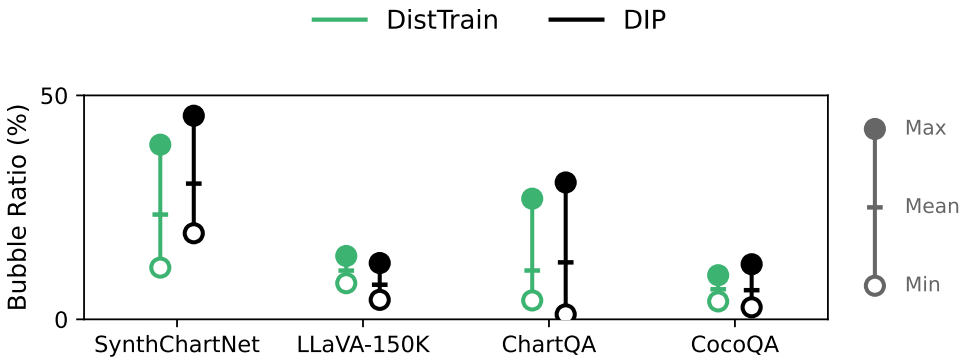


Figure 4.6: Pipeline bubbles of existing works vs ideal pipeline schedule with perfect workload balance.

tinues to fluctuate across microbatches. Optimus [34] and DistTrain strictly couple modalities within the microbatch boundary, forcing the encoder and LLM to process an identical set of samples per microbatch. Optimus schedules encoder stages into LLM pipeline bubbles, but with more microbatches or techniques such as zero-bubble pipeline parallelism (ZBPP) [115], few bubbles remain to exploit, and fragmented bubbles from imbalance are not exploitable. DistTrain relies on data reordering, which mitigates but cannot resolve workload variability across microbatches. DIP [163] partitions microbatches into sub-microbatches but still confines these partitions within the rigid boundaries of the parent microbatch. Figure 4.6 shows pipeline bubbles across four datasets (100 iterations, 4-stage PP, 16 microbatches); imbalance-driven bubbles compound the irreducible data-dependency bubbles.

4.3 Entrain Overview

Entrain is a distributed MLLM training framework that addresses workload heterogeneity and variability in multimodal datasets. Entrain maximizes training throughput by co-designing static hardware resource allocation and dynamic data scheduling for distributed multimodal training:

- **Macroscopic profiling-based parallelization (§4.4)**:
  Entrain profiles large batch to derive a parallel configuration that provides optimal load balance across modalities throughout training.
- **Microscopic workload balancing via deferral (§4.5)**: Inspired by the producer-consumer model, Entrain decouples microbatch partitioning across modalities and balances workload by deferring excess LLM computation across microbatch boundaries.

Figure 4.7 shows the architecture of Entrain, which consists of two components: the

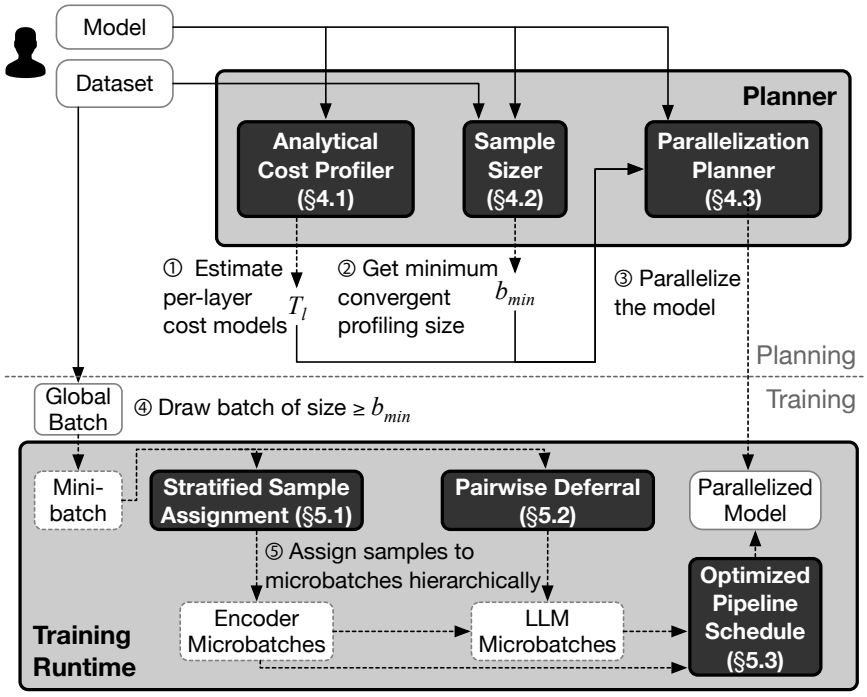


Figure 4.7: Entrain design overview.

planner and the training runtime.

Planner. The planner runs before training begins, analyzing the model and dataset to derive the parallel configuration to allocate hardware resources to each modality. Instead of profiling with a single sample or a small microbatch, the planner profiles the large batch, capturing the converged workload ratio between modalities rather than a noisy, unrepresentative snapshot. The derived parallel configuration assigns hardware resources to each modality in proportion to its computational demand.

Since profiling the full large batch before evaluating parallelization is infeasible, the planner first builds an analytical cost model by profiling with synthetic samples and fitting a polynomial equation via linear regression (§4.4.1). Using this cost model, the planner then derives the minimum profiling batch size (b_{min}) that guarantees statistical convergence of the workload ratio (§4.4.2). The user must set the global batch size to b_{min} or larger; otherwise, Entrain does not guarantee optimal load balance across iterations. Finally, the cost model and b_{min} together enable Entrain to find the optimal parallel configuration that maximizes training throughput (§4.4.3).

Training runtime. At each iteration, the runtime fetches a global batch and distributes it evenly across data-parallel replicas, each receiving a *minibatch* of size B_{global}/DP. To distribute samples, the runtime sorts them by encoder workload in descending order and

greedily assigns each to the replica with the minimum current LLM workload. Sorting by encoder workload descending spreads the heaviest encoder samples across replicas before smaller ones fill the gaps, balancing encoder workload. For LLM workload, each sample is assigned to the replica that has accumulated the least LLM workload so far, preventing any single replica from accumulating disproportionately more LLM workload than the others. Together, these heuristics balance both encoder and LLM workloads across replicas.

Each replica then partitions its minibatch into microbatches for pipeline execution. While the planner's configuration is macroscopically optimal, this partitioning re-exposes sample-level workload variance. Because encoder and LLM workloads vary independently across samples, balancing both simultaneously within each microbatch is intractable. Entrain addresses this by decoupling microbatch boundaries across modalities: the encoder and LLM need not process the same samples in a given microbatch. Entrain first strictly balances encoder microbatches (§4.5.1), then *defers* excess LLM workload from overloaded to underloaded microbatches, equalizing execution time across the pipeline (§4.5.2). The overhead of deferral due to the reversed data dependency in backward pass is handled by optimizing pipeline schedule (§4.5.3).

4.4 Macroscopic Analysis-Based Model Parallelization

The primary objective of macroscopic analysis is to derive a single, static parallel configuration that optimally balances the heterogeneous MLLM architecture. As established in Section 4.2.3, deriving this configuration via microscopic profiling fails due to the extreme sample-level variance inherent in multimodal datasets. Entrain instead anchors its resource allocation to a macroscopic view of the dataset.

This section proceeds in three stages. We first build a hardware-calibrated analytical cost model $T_{l,i}$ that estimates per-layer workloads without exhaustive profiling (§4.4.1). We then use it to determine a profiling batch size b_{min} large enough to yield a stable macro-level allocation (§4.4.2). Finally, we derive the static parallel configuration through hierarchical pipeline balancing that first optimizes each modality component locally and then aligns them under a shared pipeline schedule (§4.4.3).

4.4.1 Hardware-Calibrated Analytical Cost Model

Determining b_{min} and searching over configurations both require repeated workload estimates across many batches and candidate partitions. Exhaustive profiling is prohibitively expensive

in both cases, and pure FLOP counting is insufficient because it misses hardware-specific effects such as memory-bandwidth saturation and kernel-launch overhead.

Entrain therefore pre-builds a hardware-calibrated analytical cost model before any profiling or parallel configuration search begins. It profiles every layer separately over the valid (TP, CP) configurations using synthetic inputs at representative sizes (e.g., number of tokens $x \in \{64, 256, 1k, 4k, 16k\}$), then fits the measurements to a configuration-aware quadratic model $T_{l,i}(x, \text{TP}, \text{CP}) = ax^2 + bx + c$ [102, 145] via linear regression. We fit a model per layer because different layer types scale differently with sequence length – e.g., attention is $O(x^2)$ while MLPs and embeddings are $O(x)$ – and it enables estimating any pipeline stage cost by summing the costs of the individual layers it contains.

4.4.2 Deriving Minimum Stable Profiling Batch Size

Before beginning parallel configuration search, Entrain must determine a minimum profiling batch size b_{min} large enough that random batches of b_{min} samples consistently yield the same discrete GPU allocation under the chosen data-parallel degree DP. Entrain establishes this guarantee in two steps. First, it repeatedly samples batches of size b_{min} and uses Bernoulli trials to test whether the resulting allocation is stable. Second, it uses the Law of Large Numbers to show that stability at batch size b_{min} implies stability for any larger global batch size $B_{global} \geq b_{min}$.

Algorithm 2 Probabilistic Macroscopic Profiling Strategy

Input: Confidence level $1 - \alpha$, Target error limit p_{error}, Initial sample size n_0, GPUs N_{total}, Data Parallel degree DP, Cost Models $T_{l,i}$
Output: Stable profiling batch size b_{min}

$k \leftarrow \lceil \ln(\alpha) / \ln(1 - p_{error}) \rceil$ # Compute required validation trials via Binomial bounds
$n \leftarrow n_0$
loop
  $\vec{P}_{ref} \leftarrow \text{EstimateMacroscopicProportions}(n, T_{l,i})$
  $\vec{M}_{ref} \leftarrow \text{ProportionalAllocation}(N_{total}, DP, \vec{P}_{ref})$
  $IsStable \leftarrow \text{True}$
  for $t = 1$ to k **do**
    $\vec{P}_{test} \leftarrow \text{EstimateMacroscopicProportions}(n, T_{l,i})$
    $\vec{M}_{test} \leftarrow \text{ProportionalAllocation}(N_{total}, DP, \vec{P}_{test})$
    if $\vec{M}_{test} \neq \vec{M}_{ref}$ **then**
      $IsStable \leftarrow \text{False}$
      break
  if $IsStable$ **then**
    return n # Minimum stable profiling batch size b_{min}
  $n \leftarrow n \times 2$ # Increase batch size if variance is too high

Algorithm 2 describes the probabilistic procedure for determining the minimum stable profiling batch size b_{min}. For a candidate n, Entrain first draws a reference batch and estimates the workload proportions across modality components $\vec{P}_{ref}$ (Line 4). These proportions are continuous floating-point values, but GPU assignment must be discrete; thus, Entrain converts them into $\vec{M}_{ref}$, a vector of per-modality per-replica GPU counts, by distributing the per-replica budget of N_{total}/DP GPUs proportionally across modality components and rounding to the nearest feasible integers (Line 5). It then draws k additional independent validation batches (Line 7), and for each one, treats the event "the resulting discrete GPU allocation differs from the reference allocation" as a Bernoulli failure (Line 12). Because the continuous ratio is rounded to integer counts, small fluctuations between batches often fall into the same allocation (e.g., 1:0.98 and 1:1.12 both round to 1:1 when only 2 GPUs are available).

If all k trials return the same allocation, standard binomial bounds imply that, with confidence $1 - \alpha$, a fresh batch of size b_{min} will yield a different allocation with probability at most p_{error} (Line 14). At batch size b_{min}, the Bernoulli test thus establishes that sampling noise is small enough that repeated random draws induce the same discrete allocation with high probability. If the test fails, Entrain doubles the batch size and repeats. The loop is guaranteed to terminate: by the Strong Law of Large Numbers, the sample mean converges almost surely to the population mean, so the test must eventually pass (Appendix C.1). In our experiments, convergence occurs by batch size 256 across all evaluated datasets and GPU configurations (Appendix C.5).

The Bernoulli test certifies stability only at the profiled size b_{min}; the Law of Large Numbers extends the guarantee to all larger batches. $\vec{P}_{ref}$ and $\vec{P}_{test}$ are sample means over per-sample workloads. By the Law of Large Numbers, they converge to the population mean as the batch size grows; so the estimator at any $B_{global} \geq b_{min}$ is at least as concentrated as the one at size b_{min}. Hence, if random size-b_{min} batches already induce the same allocation with high probability, any global batch of size $B_{global} \geq b_{min}$ does so with at least as high probability. Appendix C.2 provides the full derivation. Overall, this probabilistic procedure allows Entrain to use a small profiling batch while still recovering the correct macro-level hardware allocation.

4.4.3 Heterogeneous Pipeline Balancing

Given a stable profiling size b_{min} and calibrated layer costs $T_{l,i}$, the remaining task is to derive a static parallel configuration. An MLLM forms a heterogeneous pipeline: modality-specific encoders feed the LLM, and end-to-end throughput is bounded by β_{max}, the slowest stage

across all modality components. Rather than searching over all parallel dimensions of the full model, Entrain exploits the modularity of MLLMs – each modality encoder and the LLM are individual components – and decomposes the search into two tiers: it first optimizes each modality component locally, then evaluates the resulting per-component bottlenecks under a shared pipeline schedule. Algorithm 3 summarizes this search.

Algorithm 3 Heterogeneous Model Parallel Configuration Search

Input: Stable profiling batch size b_{min}, Global Batch B_{global}, Microbatch size μ, GPUs N_{total}, Modality components $\mathcal{C}$, Cost Models $T_{l,i}$
Output: Optimal static parallel configuration C^*

$\vec{P}_{pop} \leftarrow$ EstimateMacroscopicProportions$(b_{min}, T_{l,i})$
$MaxThroughput \leftarrow 0$
for each valid hardware topology $C_{hw} = \{DP, \vec{TP}, \vec{CP}, \vec{PP}\}$ **do**
  $\vec{M} \leftarrow$ ProportionalAllocation$(N_{total}, DP, \vec{P}_{pop})$
  if C_{hw} violates VRAM limits or $B_{global} \not\equiv 0 \pmod{DP \cdot \mu}$ **then**
    continue
  $K \leftarrow B_{global}/(DP \cdot \mu)$
  # Tier 1: Intra-module balancing via Dynamic Programming
  for each modality component $i \in \mathcal{C}$ **do**
    $\{\tau_{i,p}\}_{1 \le p \le PP_i} \leftarrow$ IntraModuleBalance$(i, PP_i, TP_i, CP_i, T_{l,i})$
    $\beta_i \leftarrow \max_{1 \le p \le PP_i} \tau_{i,p}$
  # Tier 2: Inter-module balancing via Slowest-Stage Evaluation
  $\beta_{max} \leftarrow \max_{i \in \mathcal{C}}(\beta_i)$
  $\mathcal{P}_{reshard} \leftarrow$ ComputeReshardCost$(\vec{TP}, \vec{CP}, K, B_{global}, DP)$
  $T_{iter} \leftarrow \mathcal{T}_{\mathcal{S}}\Big(K, \{\tau_{i,p}\}, \beta_{max}\Big) + \mathcal{P}_{reshard}$
  $Throughput \leftarrow (DP \cdot K)/T_{iter}$
  if $Throughput > MaxThroughput$ **then**
    $MaxThroughput \leftarrow Throughput$
    $C^* \leftarrow C_{hw}$
return C^*

Intra-Module Balancing. Intra-module balancing balances the workload of pipeline stages within a modality component. Each modality encoder and the LLM backbone are internally homogeneous, consisting of repeated layers with similar structure. Entrain therefore treats each modality component $i \in \mathcal{C}$ as an independent unimodal partitioning problem, which is well-studied [177, 57, 144]. A 1D dynamic program uses the layer-wise costs evaluated under the chosen TP_i and CP_i to partition the modality component into PP_i stages, minimizing the maximum stage latency [177].

Let $F_i(\ell, p)$ denote the minimum bottleneck latency when the first ℓ layers of modality

component i are partitioned into p stages. The recurrence is:

$$F_i(\ell, p) = \min_{0 \le \ell' < \ell} \max \left(F_i(\ell', p-1), \sum_{j=\ell'+1}^{\ell} T_{j,i} \right), \tag{4.1}$$

with base case $F_i(\ell, 1) = \sum_{j=1}^{\ell} T_{j,i}$. Let $\tau_{i,p}$ denote the latency of stage p in the optimal partition of modality component i; the per-component bottleneck is $\beta_i = \max_{1 \le p \le PP_i} \tau_{i,p} = F_i(L_i, PP_i)$. Backtracking recovers both the stage latencies $\tau_{i,p}$ and the layer-to-stage mapping for execution.

Inter-Module Balancing. These per-component optima are only local. When independently optimized modality components are combined into a heterogeneous pipeline, discrepancies in latency of pipeline stages across components directly translate to pipeline bubbles. Given the shared microbatch count $K = B_{global}/(DP \cdot \mu)$, where μ is the microbatch size, Entrain evaluates each valid factorization using the stage-latency set $\{\tau_{i,p}\}$ and the the slowest stage latency across all components $\beta_{max} = \max_{i \in \mathcal{C}} \beta_i$. Here, $\mathcal{T}_{\mathcal{S}}$ denotes the analytical iteration time of pipeline schedule $\mathcal{S}$ [177]:

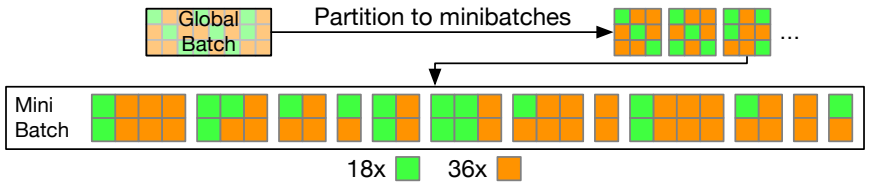
$$\mathcal{T}_{\mathcal{S}}\Big(K, \{\tau_{i,p}\}, \beta_{max}\Big) = \sum_{i \in \mathcal{C}} \sum_{p=1}^{PP_i} \tau_{i,p} + (K-1)\beta_{max}. \tag{4.2}$$

When adjacent modality components use different tensor- or context-parallel degrees, Entrain models the resulting communication overhead at their boundaries as a resharding cost $\mathcal{P}_{reshard}$. The total iteration time thus includes $\mathcal{P}_{reshard}$. The selected configuration maximizes the resulting end-to-end training throughput.

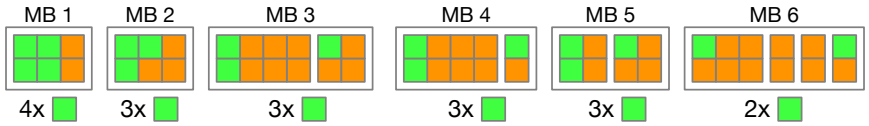
This decomposition keeps the search tractable. For each component i, the search enumerates only valid spatial factorizations satisfying $TP_i \times CP_i \times PP_i = M_i$, where M_i is the per-replica GPU budget for component i, rather than brute-forcing arbitrary parallel dimensions.

4.5 Hierarchical Microbatch Assignment

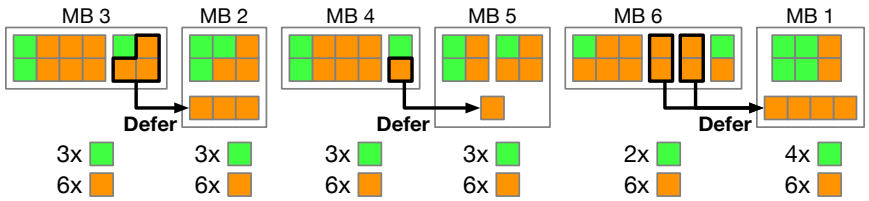
Section 4.3 described how Entrain distributes the global batch across replicas. However, the harder challenge arises when each replica's minibatch is partitioned into microbatches for pipeline parallelism. The small number of samples per microbatch amplifies individual workload variance, and encoder and LLM stages exhibit independent workload distributions that cannot be balanced simultaneously. Figure 4.8 shows that encoder and LLM workloads are highly variable between microbatches and how hierarchical microbatch assignment



(a) An example of a minibatch after the global batch is partitioned.



(b) Stratified sample assignment to microbatches focusing on balancing encoder workload (§4.5.1).

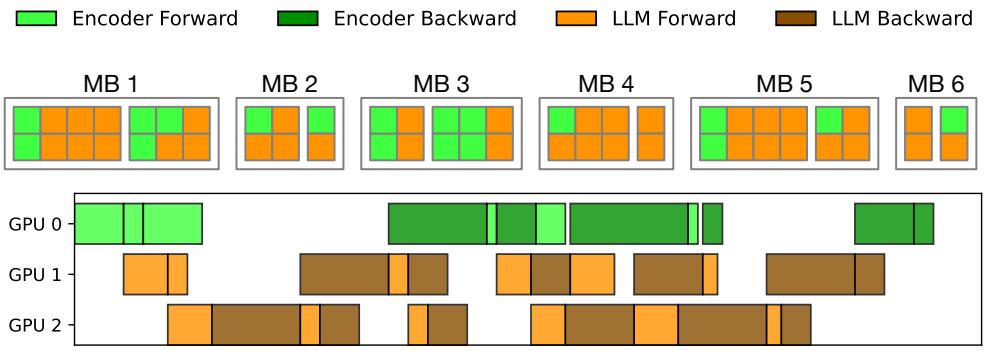


(c) Pairwise deferral optimization to balance LLM workload (§4.5.2). Execution is done in order from left to right, not in the number order.

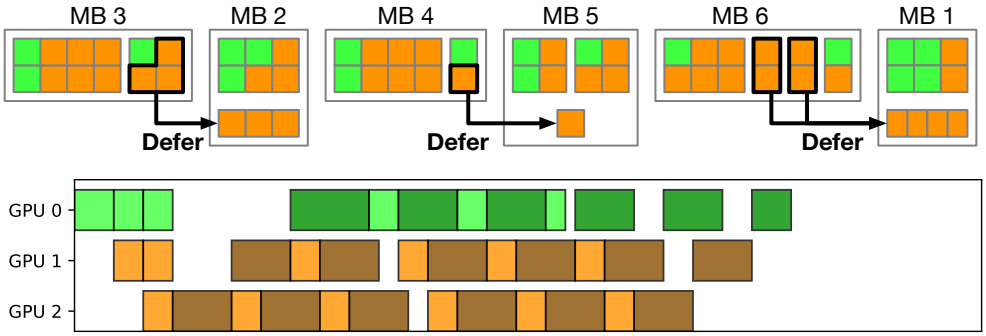
Figure 4.8: The hierarchical microbatch assignment algorithm. The number of boxes represents the amount of workload of each sample, where green boxes are for encoder workload and orange boxes are for LLM workload.

balances the workload. Considering the total workload ratio of vision and LLM is 1:2, a 3-stage pipeline schedule with one stage for vision and two stages for LLM is supposed to be balanced, however, such variability introduces a lot of internal fragmentation as depicted in Figure 4.9a. With pairwise deferral optimization, all microbatches across all pipeline stages are well balanced, as in Figure 4.9b.

Entrain resolves this by adopting a *producer-consumer strategy*: it exploits the pipeline buffer between the encoders (producers) and the LLM (consumer) as a workload variability absorber. Instead of always enforcing equal sample counts per microbatch across both stages, Entrain decouples their schedules through a two-step hierarchical assignment, as depicted in Figure 4.8. In the first step, stratified sample assignment balances encoder execution time while laying the groundwork for the subsequent deferral (§4.5.1). In the second step, pairwise deferral optimization shifts residual LLM workload across microbatches to balance the LLM execution time without disturbing the encoder schedule (§4.5.2). Algorithm 4 summarizes the hierarchical microbatch assignment algorithm.



(a) Pipeline schedule before pairwise deferral optimization.



(b) Pipeline schedule after pairwise deferral optimization.

Figure 4.9: Comparison of 3-stage pipeline parallel schedule before and after pairwise deferral optimization.

The backward pass, however, inherently reverses the data dependency; the encoder backward pass requires gradients from the LLM, thus deferral widens this latency gap for deferred samples. Entrain handles this through split-backward processing and eager forward scheduling (§4.5.3).

If the model has multiple modality encoders (e.g., Omni-modal models [162, 3, 88]), merging the modality encoders into a single unified encoder module allows hierarchical microbatch assignment to be applied.

4.5.1 Stratified Sample Assignment to Microbatches

Entrain assigns samples to microbatches using a stratified strategy that balances encoder execution time while ensuring each microbatch retains sufficient fine-grained samples for the subsequent deferral phase.

Before assignment, Entrain determines the effective number of microbatches K_{eff} to use (line 3). As each sample's computation is indivisible across microbatches, a single sample

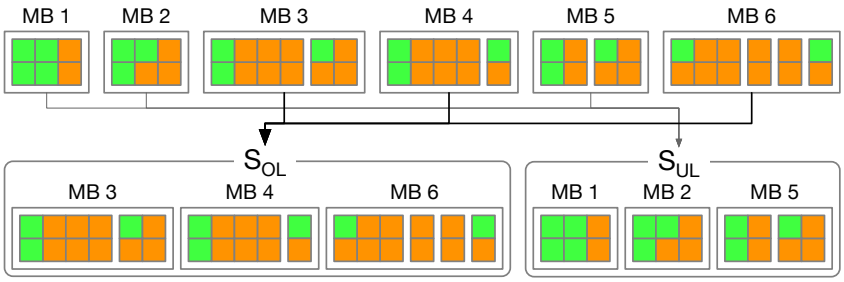
Algorithm 4 Hierarchical microbatch assignment.

Input: Global batch B_{global}, Data Parallel degree DP, number of microbatches K
Output: Encoder microbatches $\mathcal{M}^{\text{enc}}$, LLM microbatches $\mathcal{M}^{\text{LLM}}$

Section 4.3: DP-level sample assignment to replicas
1: Sort B_{global} by $w_{\text{encoder},i}$ in descending order
2: Assign samples to each replica; let S denote samples assigned to this replica
Section 4.5.1: stratified sample assignment to microbatches
3: $K_{\text{eff}} \leftarrow \min(K, \lfloor \sum_i w_{\text{encoder},i} / w_{\text{encoder,max}} \rfloor)$
4: Partition S into S_{c} and S_{f} by LLM workload
5: Sort S_{c} and S_{f} by $w_{\text{encoder},i}$ in descending order
6: Assign w_{encoder} of S_{c} then S_{f} to $\mathcal{M}^{\text{enc}}$ via Min-Max greedy
Section 4.5.2: pairwise deferral optimization
7: Sort microbatches in $\mathcal{M}^{\text{enc}}$ by w_{LLM} in descending order
8: Partition into S_{ol} (top $K/2$) and S_{ul} (bottom $K/2$)
9: **for** each pair (i, j) where $i \in S_{\text{ol}}$ and $j \in S_{\text{ul}}$ **do**
10: $S_{\text{deferred}} \leftarrow \textsc{SubsetSumDynamicProgramming}(i, j)$
11: $V_{i,j} \leftarrow \textsc{CalcBottleneck}(S_{\text{deferred}}, i, j)$ # Equation 4.3
12: $L \leftarrow [w_{\text{LLM},i} \text{ for } i \in S_{\text{ol}}]$
13: $T^*, \mathcal{P} \leftarrow \textsc{BottleneckMatch}(V, L, S_{\text{ol}}, S_{\text{ul}})$
14: Reorder microbatches in $\mathcal{M}^{\text{enc}}$ into interleaved pairs $(ol_0, ul_0, ol_1, ul_1, \ldots)$ per $\mathcal{P}$
15: Assign LLM workload of corresponding samples in $\mathcal{M}^{\text{enc}}$ to $\mathcal{M}^{\text{LLM}}$, deferring samples specified by $\mathcal{P}$
16: **return** $\mathcal{M}^{\text{enc}}, \mathcal{M}^{\text{LLM}}$

with the maximum encoder workload $w_{\text{encoder,max}}$ sets a lower bound on any microbatch's encoder workload. If K is too large, this microbatch is dominated by $w_{\text{encoder,max}}$ while the remaining $K-1$ microbatches would be significantly lighter. In this case, even though K is given by the user, Entrain reduces K to K_{eff} so that each microbatch has workload close to $w_{\text{encoder,max}}$. If other samples' workload sum is large enough to create $K-1$ microbatches with similar or larger workload, Entrain uses the user's K as K_{eff}.

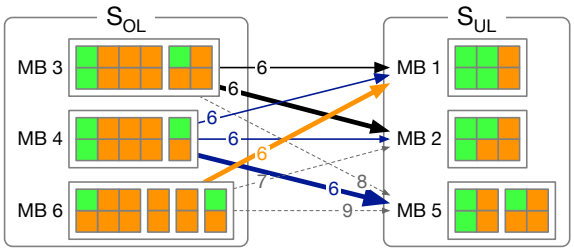
After determining the effective number of microbatches, Entrain assigns samples to K_{eff} microbatches. Samples are first partitioned into a coarse-grained set S_c with high LLM workload and a fine-grained set S_f with low LLM workload (line 4), then assigned to microbatches sequentially. Within each set, samples are sorted by $w_{\text{encoder},i}$ in descending order (line 5) and assigned to microbatches via a Min-Max greedy heuristic on encoder workload (line 6). Figure 4.8b illustrates the result of assigning samples in Figure 4.8a to microbatches by applying the stratified sample assignment. The total encoder workload is 18 units, almost evenly distributed across 6 microbatches with 3 units each on average. We do not consider the LLM workload balancing in this phase, so the microbatches are balanced only on the encoder workload. Unlike traditional pipeline parallelism, Entrain allows microbatch sample



(a) Dividing a set of microbatches to an overloaded set (S_{ol}) and an underloaded set (S_{ul}).

	MB1	MB2	MB5
MB3	MB3: 6 MB1: 5	MB3: 6 MB2: 6	MB3: 6 MB5: 8
MB4	MB4: 6 MB1: 3	MB4: 6 MB2: 4	MB4: 6 MB5: 6
MB6	MB6: 6 MB1: 6	MB6: 7 MB2: 6	MB6: 9 MB5: 6

(b) Computed deferral set S_{deferred} for microbatch i and j and LLM workload $w_{\text{LLM},i}$ and $w_{\text{LLM},j}$ after deferral.



(c) A bipartite graph $\mathcal{G}(S_{\text{ol}}, S_{\text{ul}}, V)$. Edges with $V_{i,j} > T$ are represented as grey dotted lines. Bold lines represent the matching $\mathcal{P}$.

Figure 4.10: A visualization of pairwise deferral optimization with microbatches in Figure 4.8b.

counts to vary, prioritizing encoder balance over count uniformity.

The partition into S_c and S_f is to facilitate the deferral phase. Without this partition, the greedy – which is blind to LLM workload – may leave some microbatches with only high-LLM-workload samples, starving them of the low-LLM-workload units needed for deferral and preventing deferral from balancing LLM workload. By separating samples into S_c and S_f upfront, every microbatch is guaranteed to receive S_f samples, ensuring pairwise deferral always has units available to shift and balance LLM workload delicately. This sequential two-subset assignment does not break the encoder balancing guarantee. The greedy always assigns to the least-loaded microbatch and each subset is independently sorted in decreasing order of encoder workload, thus the combined assignment is a valid longest processing time (LPT) list scheduling run, which by Graham's theorem yields a makespan within $(2 - 1/K)$ times optimal, where K is the number of microbatches $B_{global}/(DP \cdot \mu)$ [42].

4.5.2 Pairwise Deferral Optimization

After stratified sample assignment, Entrain balances LLM execution time across microbatches by *deferring* selected sample's LLM computation from overloaded microbatches to underloaded ones, leaving the encoder schedule intact. After deferral, microbatches are paired and reordered so that each overloaded microbatch is immediately followed by its underloaded partner, and selected samples' LLM workload is shifted between each pair. This *pairwise reordering* minimizes peak activation buffer memory. Deferred samples must retain their encoder activations until the LLM processes them, and placing the underloaded partner immediately after its overloaded counterpart bounds the buffering duration to a single microbatch interval. Figure 4.8c demonstrates the result of applying the pairwise deferral optimization to the microbatches in Figure 4.8b. The encoder workload is not affected by the deferral, while the LLM workload is evenly distributed across microbatches with 6 units each.

The core challenge is that the two decisions required for deferral are tightly coupled: which microbatches to pair, and which specific samples to transfer within each pair. A naive approach that optimizes them independently – e.g., greedily pairing the most overloaded microbatch with the most underloaded one – is suboptimal, because the achievable transfer amount depends on the discrete sample composition of the pair and is therefore pair-specific. Entrain therefore formulates deferral as a *bottleneck assignment problem* that co-optimizes both decisions to minimize the maximum LLM execution time across all microbatches. The solution proceeds in three steps: finding the optimal subset of samples to defer for every candidate pair (Optimal deferral set calculation), aggregating these into a bottleneck cost matrix (Bottleneck cost formulation), and determining the globally optimal pairing (Bottleneck matching optimization). Figure 4.10 visualizes how the pairwise deferral optimization works.

Optimal deferral set calculation. Entrain sorts the K microbatches by LLM workload and partitions them into an overloaded set S_{ol} (top $K/2$) and an underloaded set S_{ul} (bottom $K/2$), as shown in Figure 4.10a (line 8). For each candidate pair (i, j) where $i \in S_{\text{ol}}$ and $j \in S_{\text{ul}}$, Entrain identifies the subset S_{deferred} of samples in microbatch i as an optimal deferral set between the two microbatches (line 10). The total LLM workload of S_{deferred} is closest to the target transfer amount $\delta = (w_{\text{LLM},i} - w_{\text{LLM},j})/2$. Figure 4.10b tabulates the resulting S_{deferred}s and post-deferral LLM workloads. Finding this subset is an instance of the subset sum problem, which Entrain solves with a discretized dynamic programming: a table D of size w' (the total rounded LLM workload of the overloaded microbatch) is queried for the sum k^* that minimizes the residual $|\delta - k^*|$, identifying the optimal S_{deferred}. This runs in

pseudo-polynomial time $O(N_{\text{ol}} \times w')$, where N_{ol} is the number of samples in the overloaded microbatch.

Bottleneck cost formulation. With S_{deferred} determined for every candidate pair, Entrain builds a bottleneck cost matrix V and a standalone cost vector L. $V_{i,j}$ is the minimum bottleneck LLM execution time achievable when microbatch $i \in S_{\text{ol}}$ defers S_{deferred} to microbatch $j \in S_{\text{ul}}$ (line 11):

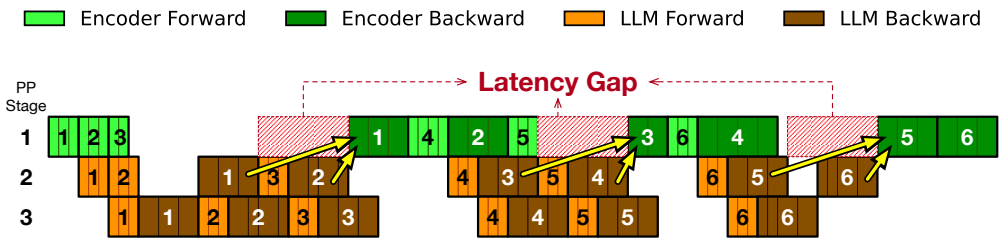
$$V_{i,j} = \max\left(w_{\text{LLM},i} - w_{\text{deferred},i}, w_{\text{LLM},j} + w_{\text{deferred},i}\right) \tag{4.3}$$

L_i is the cost of microbatch $i \in S_{\text{ol}}$ if it remains unpaired, i.e., $L_i = w_{\text{LLM},i}$ (line 12). L is used in the bottleneck matching to determine whether a microbatch's LLM workload is high enough to require deferral, or it if can execute within budget on its own.

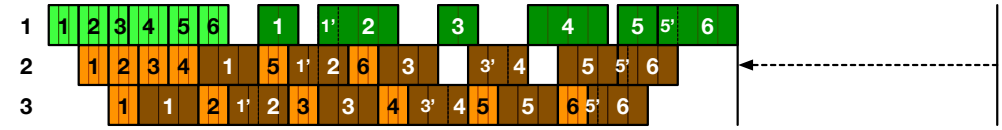
Bottleneck matching optimization. Given V and L, Entrain finds the minimum feasible threshold T^* via BottleneckMatch (line 13). T^* is the smallest value in $V \cup L$ such that every microbatch can complete LLM execution within T^* either by deferring LLM workload to an underloaded partner, or by executing alone without deferral. Since feasibility is monotone (if T is feasible, any larger T is also feasible), binary search over the at most $O(K^2)$ candidates in $V \cup L$ efficiently finds T^*. For each candidate T, Entrain constructs a restricted bipartite graph $\mathcal{G}(S_{\text{ol}}, S_{\text{ul}}, V)$ with edges only where $V_{i,j} \leq T$, and verifies feasibility via DFS-based bipartite matching on critical microbatches. Non-critical microbatches ($L_i \leq T^*$) are arbitrarily assigned to remaining S_{ul} members with no deferral. Each feasibility check costs $O(E\sqrt{K})$ where E is the edge count of $\mathcal{G}$; given the small K, the total overhead is negligible. The algorithm returns T^* and the complete matching $\mathcal{P}$. Microbatches are then arranged into an interleaved execution sequence $(ol_0, ul_0, ol_1, ul_1, \ldots)$ per $\mathcal{P}$ (line 15), with matched pairs transferring selected samples' LLM workload from the overloaded to the immediately following underloaded microbatch.

4.5.3 Optimizing Backward Pass Dependency

Our scheduling strategy focuses on optimizing the forward pass. However, in pipeline parallel training, the backward pass introduces strict reverse dependencies: the gradients for the encoder cannot be computed until the corresponding gradients from the LLM are available. Deferring the LLM workload inherently delays the completion of their forward pass, and as a result, the generation of gradients for these deferred samples is also delayed. This creates a *latency gap* for the encoder backward pass, as shown in Figure 4.11a. After completing its forward pass for microbatch i, the encoder stage expects gradients for a whole microbatch i to return. If some LLM workload in the microbatch is deferred to microbatch $i + 1$, the



(a) Backward passes of the encoder stage for microbatches deferring LLM workload are delayed until the gradients from the LLM backward pass are available.



(b) Split-backward processing allows the encoder backward pass to be executed immediately after getting the gradients from the LLM backward pass. n' refers to deferred samples in backward pass.

Figure 4.11: Visualization of pipeline parallelism schedule of microbatches in Figure 4.8c with and without backward dependency optimization.

encoder faces a potential pipeline stall waiting for the delayed gradients to arrive.

The split-backward processing. To prevent this delay from stalling the encoder backward pass, we modify the execution logic to support split-backward processing. Since the scheduler is deterministic, the encoder is aware of which specific samples within the microbatch i will be deferred during the LLM forward pass. When the backward pass for microbatch i is scheduled, the encoder does not wait for the complete set of gradients. Instead, it immediately executes the backward pass for the non-deferred samples. The backward pass for the deferred samples is also deferred until their gradients arrive from the LLM backward pass. Figure 4.11b shows the pipeline parallelism schedule with split-backward processing.

Eager forward pass execution. Even with split-backward processing, the backward pass workload increases later in the iteration because of the deferred samples. To accommodate this back-loaded computation, we must reserve compute capacity in the later microbatches. We achieve this by scheduling forward passes as eagerly as possible. While the standard 1F1B schedule executes only $s - i + 1$ forward passes in the initial warm-up phase (where s is the number of stages and i is the stage index starting from 1) as in Figure 4.11a before entering the steady phase of interleaving one forward and one backward pass, we execute as many forward passes as memory constraints allow before switching to the steady phase. This eager execution reduces the number of forward passes required in the middle of the iteration, effectively yielding time slots that compute nodes can use to process the backward

pass of deferred samples.

This makes Entrain more friendly to more advanced pipeline parallelism techniques, such as zero-bubble pipeline parallelism (ZBPP) [115], which recommends executing the forward passes as eagerly as possible.

Numerical correctness. With the split-backward processing, the number of backward passes – and naturally the number of gradient accumulations – increases. In low-precision training (e.g., BF16, FP8, or NVFP4 [135, 16, 113]), such an increased number of gradient accumulations theoretically heightens the risk of numerical instability and swamping. However, this amplified accumulation depth is structurally unavoidable in large-scale distributed training. Many research papers have proposed algorithmic safeguards to solve this issue and thus it is no longer a destabilizing factor. The proposed algorithms, such as stochastic rounding (SR) [19], Kahan Summation [170], or mixed precision with higher precision for gradients [113], effectively neutralize the negative impact of increased number of gradient accumulations, ensuring convergence.

4.6 Implementation

Entrain is implemented on top of PyTorch, Cornstarch [56], and HuggingFace Transformers [155] in Python. Entrain adds batch sizer and macroscopic profiler to derive the static model-parallel configuration and uses Cornstarch's multimodal 4D parallelization capability to partition the model and distribute it to GPUs accordingly. Entrain exploits Cornstarch's multimodal 4D parallel execution capability to run distributed training, with the following modifications.

Microbatch scheduler. We replace DistributedSampler with Entrain sampler that sorts samples and assigns them to the microbatches. Deferral optimization is also executed at this point with estimated workload cost. Executing deferral optimization at the scheduler level allows Entrain to be compatible with sequence packing. Sequence packing is effective to reduce padding overhead, thus widely used [67, 175, 7, 151, 167]. By running deferral optimization before packing sequences and sending deferral information together with the packed microbatches, the pipeline execution engine can track the deferred samples within the packed microbatches.

Pipeline execution engine. We modified the existing 1F1B [100] and Zero-Bubble Pipeline Parallelism [115] pipeline schedule to support deferral optimization. The execution engine receives deferral information from the scheduler along with the microbatches before executing each iteration, and executes the pipeline schedule. During the execution, the engine temporarily stores deferred activations and tracks the deferred sub-microbatches to ensure

Table 4.1: Parallel configuration of Entrain and the baselines and execution setup. For all frameworks, TP=2, CP=1, and DP=4. E.PP and L.PP represent encoder pipeline parallel degree and LLM pipeline parallel degree, respectively. * indicates DIP colocates vision and LLM stages to the same pipeline stage.

SynthChartNet		DistTrain	DIP	Entrain
Execution Setup	Profiling Size	1	4	256
	Global Batch	512	512	512
Qwen2Vision Llama3-1b	E.PP	4	8*	5
	L.PP	4	8*	3
Qwen2Vision Llama3-3b	E.PP	5	8*	4
	L.PP	3	8*	4

all deferred samples are properly processed in the backward pass.

4.7 Evaluation

We evaluate Entrain on a variety of datasets and models and compare it against DistTrain and DIP. We summarize the results as follows:

- Entrain outperforms the baselines in end-to-end training throughput by up to 1.40× (§4.7.2).
- Entrain's proposed macroscopic profiling paradigm effectively estimates the workload ratio between modalities and provides a stable parallel configuration (§4.7.3).
- Entrain's hierarchical microbatch assignment and deferral optimization stabilize the workload variability across microbatches by up to 10.6× (§4.7.4).

4.7.1 Experimental Setup

Cluster setup. We evaluate Entrain using 16 NVIDIA A40-48GB GPUs on 4 nodes. Each node has 4 NVIDIA A40 GPUs and a NVIDIA Mellanox ConnectX-6 200Gbps Infiniband adaptor. The four GPUs in a node are connected in pairs using NVLink and connected to the node via PCIe 4.0.

To emulate a larger cluster, we exploit the fact that data-parallel replicas synchronize only at the end of each iteration. We execute one DP replica at a time using TP, PP, and CP on real hardware (16 GPUs) and execute 4 replicas sequentially. The iteration time is taken

as the maximum across the 4 executions, mirroring the all-reduce barrier to synchronize gradients. All experiments use this emulated 64-GPU setup.

Datasets and models. We evaluate Entrain using HuggingFace FineVision dataset [154]. It is a multimodal dataset that contains more than 20 million samples of image and text pairs, where samples are collected from over 200 datasets. We pick 4 subdatasets from the FineVision dataset – SynthChartNet [103], ChartQA [91], CocoQA [128], and LLaVA-150k [87] – that have distinct data distributions. We show SynthChartNet data in the main text because it is the most variable, and other datasets are shown in Appendices.

For models, we run various sizes of vision-language models (VLMs) using Qwen2.5Vision vision transformer [149] to process various resolutions of images natively and Llama3 [4] (1b and 3b parameters) for text processing.

Baselines. We compare Entrain to the following baselines:

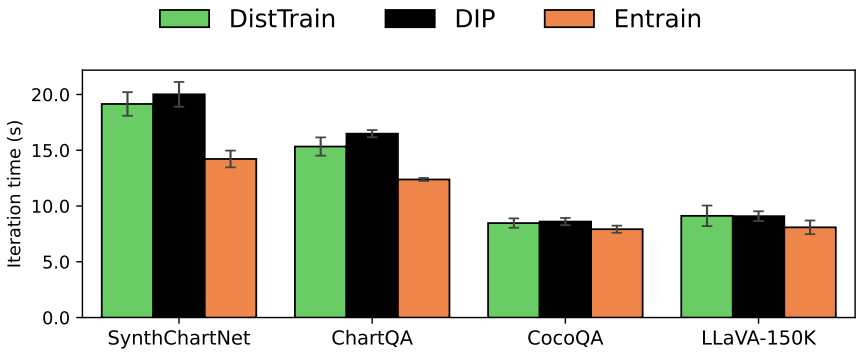
1. *DistTrain [173]*: DistTrain reorders samples to mitigate the data heterogeneity problem.
2. *DIP [163]*: DIP introduces decoupled modality scheduling to address heterogeneity and variability in multimodal training workloads.

We run 4D parallelism with different parallel configurations between Entrain and the baselines. Table 4.1 shows the parallel configurations (See Appendix C.3 for parallel configurations on other datasets). DistTrain profiles 1 sample to derive a parallel configuration, while DIP profiles 1 microbatch (4 samples) to derive its parallel configuration. We use microbatch size 4 and global batch size 512. Although Entrain supports both 1F1B and ZBPP schedules, we use 1F1B for Entrain in evaluation.

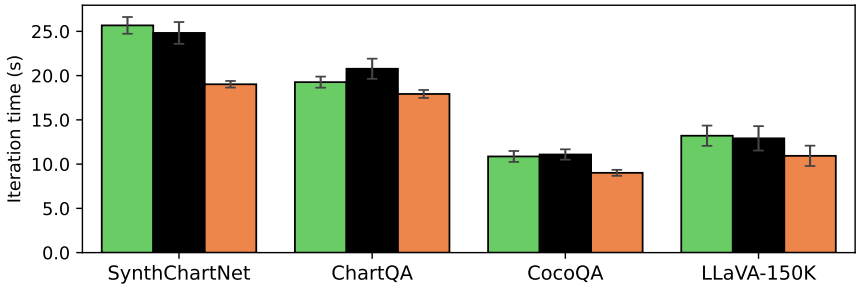
4.7.2 End-to-End Training Performance

Training throughput. Figure 4.12 shows end-to-end training performance of Entrain and the baselines using various datasets and models. The main performance improvement of Entrain over the baselines comes from two factors. First, Entrain's macroscopic profiling based parallel configuration is close to the optimal parallel, removing fragmented pipeline bubbles across modality pipeline stages. Second, Entrain's hierarchical microbatch assignment balances the workload between microbatches, which is not possible with static microbatch partitioning. Combined, Entrain achieves up to 1.40× faster end-to-end training throughput. We also visualize one iteration of the pipeline schedule of Entrain and the baselines in Figure 4.13. Entrain shows balanced workload distribution between modalities and microbatches.

Memory consumption. Figure 4.14 shows the memory consumption of the baselines and Entrain for SynthChartNet dataset on Qwen2.5Vision+Llama3-3b VLM. Entrain's deferral



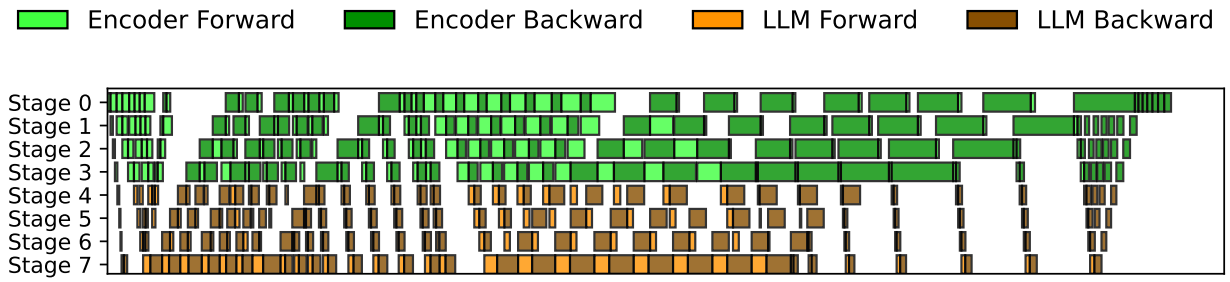
(a) Qwen2.5Vision+Llama3-1b



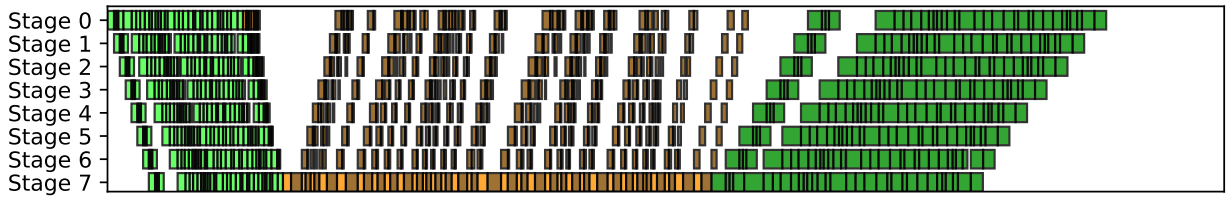
(b) Qwen2.5Vision+Llama3-3b

Figure 4.12: End-to-end training performance of Entrain and the baselines.

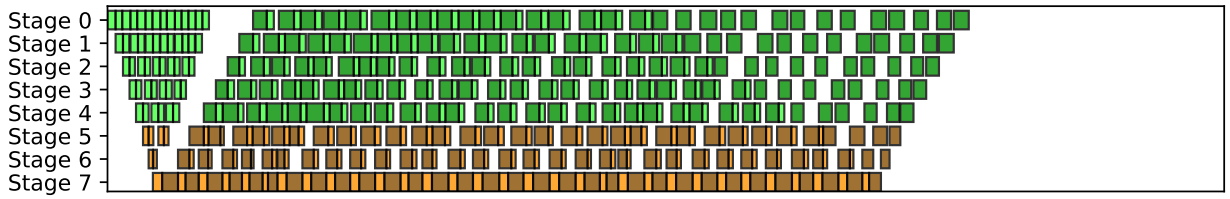
optimization temporarily stores encoder activations in the pipeline buffer and thus requires more memory than non-deferral pipeline schedules. However, increased memory consumption is minuscule because pairwise schedule only holds encoder activations for up to a single microbatch interval, and activation memory is very small compared to the total memory consumption. Rather, more balanced workload distribution between modalities and microbatches makes Entrain's memory consumption more stable than the baselines, as shown in Figure 4.14c. DistTrain (Figure 4.14a), even though it follows 1F1B as Entrain, shows much more memory variability than Entrain due to imbalanced workload distribution, ending up with much higher peak memory consumption. Meanwhile, DIP's scheduling processes all modality encoder forward passes before executing the LLM, and starts modality encoder backward pass after all LLM execution is done, as shown in Figure 4.2c. This requires much more memory (∼12GB in rank 7) than DistTrain and Entrain to maintain encoder activations until they are freed during encoder backward pass, as shown in Figure 4.14b. See Appendix C.4 for memory consumption of other configurations.



(a) DistTrain pipeline schedule visualization.



(b) DIP pipeline schedule visualization.

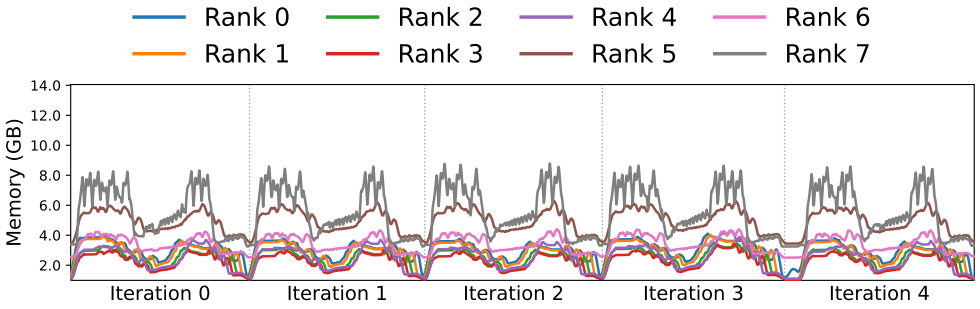


(c) Entrain pipeline schedule visualization.

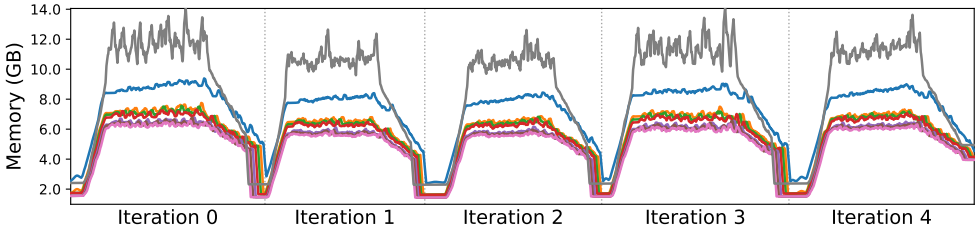
Figure 4.13: Pipeline schedule visualization of Entrain and the baselines on SynthChartNet dataset on Qwen2.5Vision+Llama3-3b VLM.

4.7.3 Analysis of Macroscopic Profiling

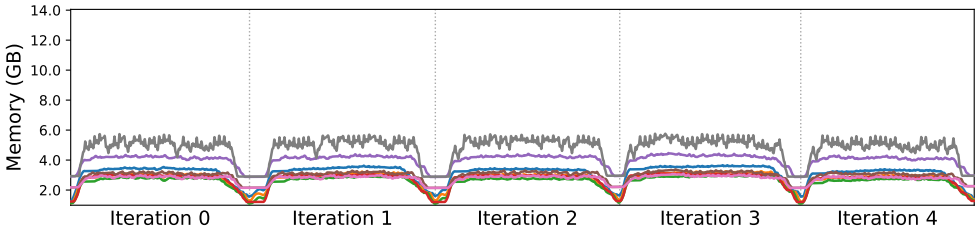
We analyze how the macroscopic profiling affects derivation of the workload ratio and parallel configuration. We draw $k = 59$ batches – providing 95% confidence level with $p_{error} = 5\%$ – with different batch sizes from the datasets and compute the workload ratio between modalities for each batch using the cost model and the metadata of the samples. Table 4.2 shows the workload ratios shown in Bernoulli trials for SynthChartNet dataset on Qwen2.5Vision+Llama3-3b VLM. Across all datasets, smaller batch sizes show more variability in the workload ratio, which leads to more variability in GPU allocation to the modalities. Depending on which samples are drawn, smaller batch sizes may allocate GPUs to the encoder and the LLM in different proportions (e.g., 10:6 or 6:10), while the true dataset workload ratio mean is 1.27:1. Different trial data are shown in Appendix C.5.



(a) Memory consumption of DistTrain pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.



(c) Memory consumption of Entrain pipeline schedule.

Figure 4.14: Memory consumption of different schedules for SynthChartNet dataset on Qwen2.5Vision+Llama3-3b VLM.

Sensitivity analysis to macroscopic profiling quality. To demonstrate the impact of parallel configuration quality on performance, we profile Entrain under varying parallel configurations, with results shown in Figure 4.15 (Appendix C.6 for other datasets). The throughput of Entrain drops significantly by 85% if a bad parallel configuration is used, while the parallel configuration derived from the macroscopic profiling with profiling size 256 – the black bars – is the best.

4.7.4 Effect of Hierarchical Microbatch Assignment to Variability

We evaluate how modality workload fluctuates across microbatches due to data variability, and how much the deferral optimization mitigates the imbalance. Figure 4.16 shows

Table 4.2: Workload ratios in Bernoulli trials of SynthChartNet dataset in Qwen2.5Vision+Llama3-3b VLM using 16 GPUs.

Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	10:6, 9:7, 8:8, 7:9, 6:10
4	✗	9:7, 8:8, 7:9
16	✗	9:7, 8:8, 7:9
64	✓	8:8
256	✓	8:8

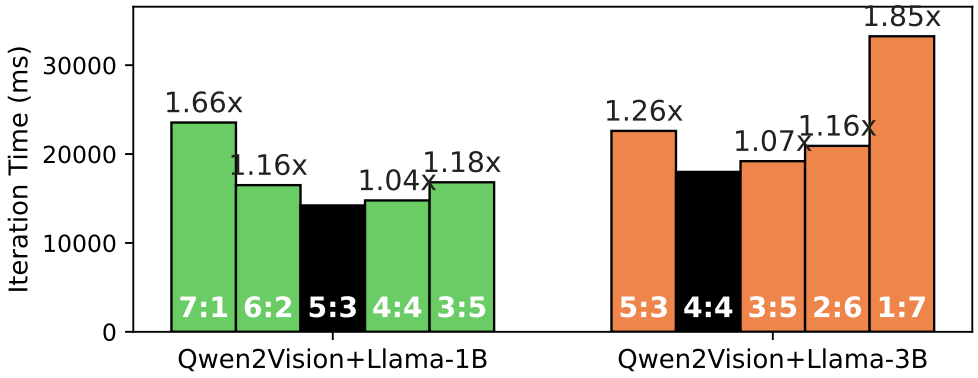


Figure 4.15: Sensitivity analysis on SynthChartNet dataset. Ratios are encoder-to-LLM workload ratios.

the variability of forward time of each modality in VLMs (Qwen2.5Vision, LLama3-1b, and LLama3-3b) across microbatches in SynthChartNet dataset (Appendix C.7 for other datasets). Each microbatch's forward time is computed as the sum of forward time of all corresponding pipeline stages. DistTrain's heuristic microbatch reordering algorithm focuses on mitigating pipeline bubbles from the holistic view, hence it fails to address the data variability between microbatches and shows high variability in both vision and LLM forward time. Entrain shows both vision and LLM forward time much more stable than the baselines. More details in numerical values are shown in Table 4.3. Low variability in vision forward time is due to the stratified sample assignment in the first phase of deferral optimization. We do not statically assign samples to have homogeneous number of samples per microbatch, but assign different number of samples to each microbatch to balance the encoder workload. Encoder workload-focused assignment breaks the balance of LLM workload, but the deferral optimization mitigates the imbalance. Because Entrain defers LLM workload as atomically – not partitioning a sample into multiple chunks and deferring only a part of the sample – the deferral does not fully address the imbalance. Yet, the deferral optimization still mitigates

Table 4.3: Standard deviation (std) of forward time of modalities in different pipeline schedules and datasets.

		Synth ChartNet	ChartQA	CocoQA	Llava-150k
Vision	DistTrain	208.07	61.34	22.82	17.98
	DIP	102.13	73.42	28.79	14.82
	Entrain	**19.60**	**15.23**	**22.11**	**8.11**
Llama 1b	DistTrain	77.92	61.34	22.82	17.98
	DIP	37.45	73.42	28.79	14.82
	Entrain	**18.79**	**15.23**	**22.11**	**8.11**
LLama 3b	DistTrain	164.55	35.44	**5.37**	32.92
	DIP	84.36	40.56	7.77	33.27
	Entrain	**40.24**	**25.72**	5.52	**31.66**

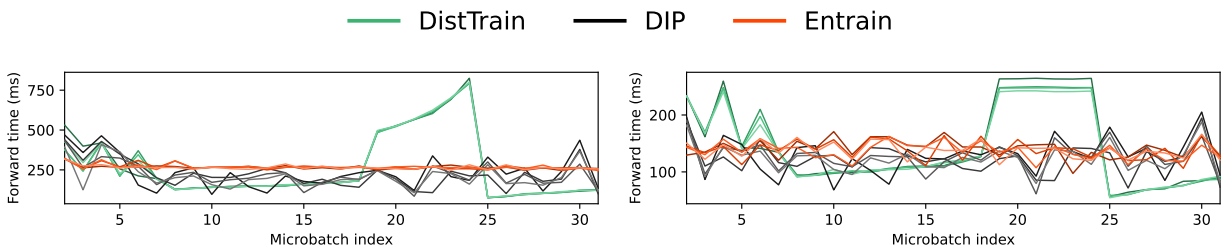
the imbalance significantly.

4.8 Related Work

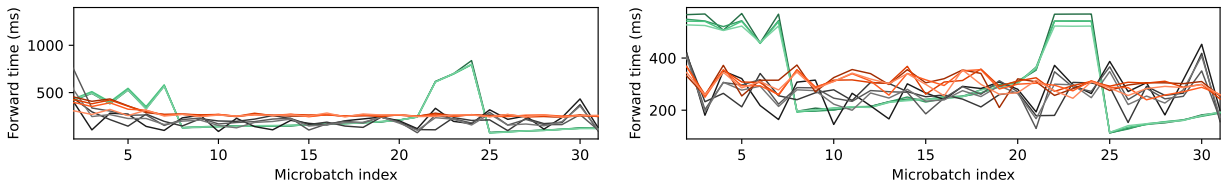
Large-scale distributed training. Driven by the scaling law [65], distributed training frameworks such as Megatron-LM [102], DeepSpeed [125], Colossal-AI [76], and TorchTitan [81] have enabled 1T+-parameter model training [165, 5]. As the focus shifts toward larger datasets and longer contexts [46], context parallelism has been proposed to distribute long sequences across GPUs [77, 43, 150, 4, 151].

Distributed multimodal LLM training. DistMM [49] is among the earliest works on distributed multimodal training, targeting CLIP-like models without LLM backbones. DistTrain [173], Optimus [34], Cornstarch [56], and DIP [163] tackle the heterogeneity between modality encoders and the LLM backbone; Cornstarch additionally handles structural characteristics such as frozen parameters and non-causal attention. None of these systems, however, fully addresses dataset variability at both the profiling and execution granularities.

Dataset heterogeneity and variability. HotSPa [37] and ByteScale [35] address sequence length variability in unimodal LLM training via dynamic parallelism reconfiguration, but assume a single axis of variability. In MLLM training, each modality follows its own independent distribution yet is coupled within each sample, making the inter-modality workload ratio chaotic at fine granularities. DistTrain [173] reorders samples to reduce pipeline bubbles but remains oblivious to intra-sample modality coupling; DIP [163] interleaves encoder and LLM pipeline stages but incurs high memory overhead from retained activations. These works overlook that the modality workload ratio converges to a stable mean at global batch scale, a property that neither demands dynamic reconfiguration nor can be captured



(a) Qwen2.5Vision (left) and Llama3-1b (right) forward time variability across microbatches.



(b) Qwen2.5Vision (left) and Llama3-3b (right) forward time variability across microbatches.

Figure 4.16: Variability of modality forward time across microbatches in SynthChartNet dataset on VLMs. Each line represents a DP replica.

by micro-level profiling.

4.9 Conclusion

In this paper, we presented Entrain, a distributed MLLM training framework that addresses both heterogeneity and variability in multimodal training workloads. Entrain counters the intuition that dynamic model-parallel configuration is necessary for MLLMs with data variability by demonstrating that at the macroscopic scale, the aggregate workload ratio between modalities reliably converges to a stable constant. Entrain also addresses the fundamental limitation of variability at the microscopic scale by proposing a hierarchical microbatch assignment algorithm that defers excess workload within each iteration, stabilizing execution time across microbatches. Entrain outperforms the baselines in end-to-end training throughput by up to 1.40×, reducing workload variability across microbatches by up to 10.6×.

CHAPTER 5

Conclusion

The explosive growth of large-scale multimodal foundation models has made distributed training across thousands of GPUs not merely advantageous but strictly necessary. Yet the very scale that makes such training possible also makes it fragile. The aggregate probability of hardware failure grows with cluster size, and the heterogeneous computational footprints of diverse modalities create persistent workload imbalance. Because all devices in a hybrid-parallel training job are bound by collective synchronization barriers, both forms of variance directly translate into systemic idle time: resource variability – the constantly shifting pool of available GPUs as hardware fails and recovers – can bring the entire cluster to a halt, while workload heterogeneity and variability – the heterogeneous and fluctuating computational footprints imposed by diverse modalities – creates persistent straggler effects that degrade end-to-end throughput. This dissertation presents three complementary systems that together neutralize this intertwined heterogeneity and variability to achieve efficient and robust distributed multimodal training.

Oobleck (Chapter 2) addresses resource variability by composing heterogeneous pipeline templates that always map all surviving GPUs onto a valid hybrid-parallel layout; upon failure, missing model states are copied from surviving replicas and the batch is rebalanced proportionally, providing continuous fault-tolerance without checkpointing or idle backup GPUs. Cornstarch (Chapter 3) introduces higher-order heterogeneity and variability in multimodal training by considering the frozen status of model components to balance the pipeline stages in pipeline parallelism, and by proposing a workload-aware context parallelism algorithm that evenly distributes dynamic non-causal attention patterns across GPUs. Entrain (Chapter 4) observes multimodal workload heterogeneity is variable across iterations, and addresses it in two ways. Across iterations, counterintuitively, variability of workload ratio between modalities converges to a stable constant at the macroscopic scale, which allows a single static parallel configuration derived from macroscopic batch-level profiling to suffice for optimal load balancing. Within an iteration, variability is re-exposed at the microscopic

scale, which is exposed when a batch is split into microbatches in pipeline parallelism. Entrain addresses this with a hierarchical microbatch assignment and deferral optimization to stabilize variability across microbatches.

The remainder of this chapter reflects on lessons learned from building these three systems (§5.1) and discusses promising directions for future work (§5.2).

5.1 Lessons Learned from Building Adaptive Multimodal Training Systems

5.1.1 Neither Offline Nor Online Planning Alone Is Enough

A first lesson is that large-scale multimodal training cannot rely exclusively on either offline planning or online adaptation. We initially expected that profiling a model and its workload before training would yield a parallel configuration that remains effective throughout execution. In practice, every system in this dissertation encountered a form of variability that invalidated that expectation. Node failures change the GPU pool in Oobleck, input-dependent attention patterns shift the per-token workload in Cornstarch, and the modality ratio within microbatches fluctuates in Entrain. The natural alternative – recomputing the parallel configuration online whenever conditions change – proved equally impractical. Solving the partitioning, scheduling, or assignment problem from scratch at every batch is too expensive to keep off the critical path of training.

What we learned is that the key design decision is not whether to be static or adaptive, but where to draw the boundary between the two. In Oobleck, offline-generated pipeline templates reduce the runtime search to a composition problem over pre-validated layouts; online adaptation then only selects among those compositions when the resource pool changes. In Cornstarch, an offline frozen-status-aware pipeline partition eliminates the most expensive balancing decisions; online adaptation handles only the residual, input-dependent attention imbalance through a lightweight heuristic that can be prefetched. In Entrain, macroscopic profiling shows that the workload ratio converges at the batch level, making a single static parallel configuration sufficient across iterations; online adaptation is needed only within an iteration, where microbatch-level variability is resolved through a deferral plan.

Looking back, the shared insight is that the offline step should absorb as much of the problem as the stable structure of the workload allows, leaving the online step with a small, well-bounded residual. When the online part is small enough, it can be made infrequent, overlapped with computation, or prefetched, thus keeping it off the critical path entirely.

5.1.2 Multimodal Workloads Expose Assumptions Worth Revisiting

A second lesson is that the largest gains in each system came not from optimizing within the boundaries of existing distributed training abstractions, but from discovering that multimodal workloads violate assumptions those abstractions silently encode. In each case, we began with a standard abstraction, found that it produced unexpected inefficiency under multimodal training, and traced the inefficiency back to an assumption that had been invisible under unimodal workloads.

Three assumptions turned out to be wrong. First, we assumed that fault tolerance requires explicitly added redundancy, whether through periodic checkpointing or redundant computations. Building Oobleck revealed that data-parallel training already maintains redundant model states across replicas during normal execution; once we recognized this, recovery could reconstruct missing states from surviving replicas without any added redundancy. Second, we assumed that pipeline balance depends primarily on the FLOP count of each stage, and that distributing tokens uniformly across GPUs is sufficient for context parallelism. Building Cornstarch showed that frozen components eliminate backward computation for certain stages and that non-causal attention makes per-token work highly irregular – two properties that are invisible when all components are trainable and attention is causal, but that create severe imbalance in multimodal model training. Third, we assumed that a microbatch is an indivisible unit: every modality processes the same set of data elements together. Building Entrain showed that decoupling microbatch boundaries across modalities and deferring elements between microbatches can substantially improve load balance, turning the microbatch from a fixed constraint into a degree of freedom.

The broader lesson for future multimodal systems is to audit the assumptions inherited from any distributed training abstraction. Multimodal workloads introduce structure – frozen components, non-causal attention, or variable modality ratios – that standard abstractions were not designed to exploit, and redesigning around that structure is often where the largest efficiency gains are found.

5.2 Future Directions

5.2.1 Extending Adaptation to Inference and Serving

This dissertation focuses on distributed "training", where heterogeneity and variability arise from changing hardware availability and fluctuating multimodal workloads. However, as

foundation models are increasingly used through test-time scaling, tool use, and agentic workflows, inference and serving are becoming equally important targets for systems optimization and ML performance improvement. In these settings, variability is not limited to the input batch. The system often does not know in advance how many output tokens a request will generate, how many reasoning steps will be taken, which tools will be invoked, or how many agents will participate in solving a task. As a result, the amount of work created by each request can evolve dynamically during execution.

This form of unpredictability makes inference scheduling fundamentally different from training scheduling. Training repeatedly executes a mostly fixed computation graph, allowing systems to exploit repetition, amortize profiling costs, and plan around stable aggregate behavior. Inference and serving workloads, by contrast, are often interactive, latency-sensitive, and shaped by decisions made during generation. Optimizing such systems therefore requires scheduling mechanisms that can react to newly revealed work while still maintaining high accelerator utilization and meeting service-level objectives. The principles explored in this dissertation, especially the separation between what can be planned offline and what must be adapted online, suggest a promising starting point. Future systems can extend these ideas to design serving schedulers that reason about uncertainty in token generation, agent coordination, and tool execution, opening a new direction for adaptive systems beyond distributed training.

5.2.2 Adaptive Compilation for Variable Workloads

Another important direction is to reconcile adaptive execution with machine learning compilation. The systems in this dissertation rely on adaptation to address resource and workload variability, but this style of adaptation is not naturally compatible with current compilation techniques. Modern ML compilers and megakernel approaches can significantly improve performance by fusing operations, reducing memory traffic, and avoiding hardware bottlenecks. To generate the most efficient code, however, compilation often benefits from knowing shapes, schedules, communication patterns, and execution structure in advance. This assumption conflicts with the adaptive strategies in this dissertation, where the best execution plan may depend on conditions revealed only at runtime, such as failures, input-dependent attention patterns, or microbatch-level workload variation.

Both sides of this tension are essential for future high-throughput systems. Adaptation is needed because large-scale ML workloads are increasingly variable and unpredictable, while compilation is needed because accelerator performance increasingly depends on carefully optimized kernels and data movement. A promising research direction is therefore adaptive

compilation, where compilation and runtime scheduling are designed together. Such systems could compile families of execution plans, generate kernels parameterized by workload structure, or trigger lightweight recompilation when runtime behavior moves outside the planned regime. By combining the robustness of adaptation with the efficiency of compilation, adaptive compilation can help future distributed ML systems sustain high throughput under real-world variability.

APPENDIX A

Oobleck Appendix

A.1 Proof of Nodes Specification Covering All Nodes

We prove the following theorem which shows a finite set of p number of pipeline templates, where the number of nodes is $(n_0, n_1, \ldots, n_{p-1})$ $(n_i < n_{i+1})$, can fully cover the node cluster with feasible number of nodes N' any time irrespective of how many failures happen on the cluster as long as its feasibility holds.

Theorem 1. *N' nodes ($(f+1)n_0 \leq N' \leq N$) can always be represented as a linear combination of the p pipeline templates with $(n_0, n_1, \ldots, n_{p-1})$ number of nodes, respectively, if the following two conditions are satisfied:*

1. *$p > n_0 - 1$.*
2. *n_i are consecutive integers ($n_i + 1 = n_{i+1}$) .*

Proof. We first formulate an integer linear combination to represent N':

$$N' = x_0 n_0 + x_1 n_1 + \ldots, x_{p-1} n_{p-1} \tag{A.1}$$

where x_i is the number of pipelines to be instantiated from the pipeline template with n_i number of nodes.

The Frobenius number $g(n_0, n_1, \ldots, n_{p-1})$, the largest number that cannot be represented as a linear combination of Equation A.1, has proven to be:

$$g = \left(\left\lfloor \frac{n_0 - 2}{p - 1} \right\rfloor \right) + d(n_0 - 1) \tag{A.2}$$

if the integer set $(n_0, n_1, \ldots, n_{p-1})$ is an arithmetic sequence, i.e., $n_i = n_0 + d(i-1)$ [124].

When we apply both Requirements 1 and 2, $g = n_0 - 1$. Fault tolerance threshold f is a non-negative integer; the minimum feasible number of nodes $N' = (f+1)n_0$ is n_0.

Therefore, any feasible N' that is larger than g and can be represented as a linear combination of Equation A.1. □

A.2 Proof of Guarantee for Pipeline Template Availability When Merging Pipelines

We first show this when failures happen in a single pipeline. When we lose k nodes ($k > 0$) from a pipeline, where all pipelines have n_0 nodes and are not able to yield any node, Oobleck instantiates a new pipeline with $2n_0 - k$ nodes by merging it with another n_0-node pipeline. We prove that a pipeline template with $2n_0 - k$ nodes is always available.

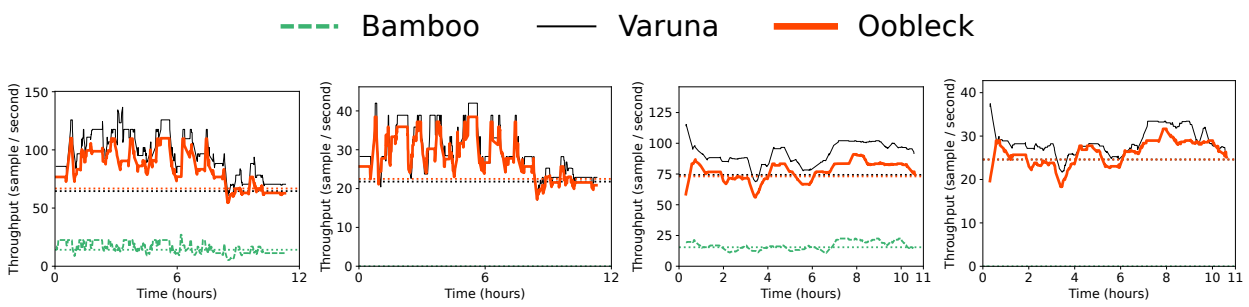
Theorem 2. *A set of pipeline templates always includes a pipeline template with $2n_0 - k$ nodes ($2n_0 - k \geq n_0$).*

Proof. A set of pipeline templates has pipeline templates with up to $N - fn_0$ nodes (§2.4.1.1). Assume that we do not have a pipeline template with $2n_0 - k$ number of nodes specification, then $N - fn_0 < 2n_0 - k$ and $N < (f+2)n_0 - k$ are assumed to be true. To not break the fault tolerance threshold that we maintain at least $f + 1$ model replicas after merging two pipelines, we must have at least $f + 2$ replicas, i.e., we should have had at least $(f + 2)n_0$ nodes before failures. Since the initial number of nodes N is always larger than the number of currently remaining nodes, $N > (f + 2)n_0$ inequality holds and it contradicts our initial assumption. Therefore, we have a pipeline template with $2n_0 - k$ number of nodes. □

When failures happen across several pipelines, multiple pipelines can have less than n_0 nodes. Oobleck repeatedly merges two pipelines until a new pipeline has enough number of nodes. Assume we merged m pipelines to get enough number of nodes, i.e., $\sum_{i=0}^{m} n_{p_i} \geq n_0$. It means merging $m-1$ pipelines was not enough to get n_0 nodes, i.e., $\sum_{i=0}^{m-1} n_{p_i} < n_0$. With $n_{p_m} \leq n_0$, we have an inequality $n_0 \leq \sum_{i=0}^{m-1} n_{p_i} + n_{p_m} < 2n_0$. It has already been proved by Theorem 2 that we have a pipeline template for all numbers in the range.

A.3 Throughput of All Models in Spot Instances

Figure A.1 shows throughput of unpresented models in the paper due to lack of space, running on Amazon EC2 `P3` spot instances and Google `a2-highgpu-1g` spot instances. Varuna could avoid fallback overhead by successfully checkpointing ahead of preemption in small models, (e.g., BERT-large, GPT-2, and GPT-3 medium), thus Varuna throughput matches Oobleck on average. However, in large models, Oobleck outperforms it. Note that lines are smoothed



(a) GPT-2 in P3. (b) GPT-3 Medium in P3. (c) GPT-2 in a2-highgpu-1g. (d) GPT-3 Medium in a2-highgpu-1g.

Figure A.1: GPT-2 and GPT-3 medium throughput changes in Amazon EC2 P3 and Google a2-highgpu-1g instances.

for visibility and do not precisely represent throughput. Varuna has more spikes down to 0 throughput in reality thus has less throughput.

APPENDIX B

Cornstarch Appendix

B.1 Cornstarch Programming Interface

Listing B.1 shows the programming interface of Cornstarch. MultimodalModule is a wrapper class that contains the modality encoders and the LLM that can be executed standalone without parallelization specifications (line 7). Cornstarch accepts parallelization specifications for each modality encoder and the LLM (line 18 to line 20). The parallelization specifications are passed to MultimodalParallelModule, which is a wrapper class that contains the specification and more hyperparameters required for distributed training (line 24). After creating a distributed MLLM, users can call `execute` method to run the training (line 39).

B.2 Supported Models

Table B.1 lists the supported models in Cornstarch at the time of writing. However, Cornstarch is not limited to these models; those in the table are just the ones we have tested.

B.3 Workload-Balanced Context Parallelism

Figure B.1 shows context parallelism results with smaller sequence lengths than 64k. Similar patterns as in Table 3.3 are observed. Intra-GPU balance balances workloads of long sequences across SMs within each GPU. While inter-GPU balance, if applied alone, is worse than context parallelism optimized for causal attention, it provides further optimized performance when combined with intra-GPU balance.

B.4 Context Parallelism Using Multiple Streams

Using multiple streams in CUDA can improve performance by allowing concurrent execution of multiple attention head computations. It does improve performance by overlapping

Listing B.1: Cornstarch APIs for distributed MLLM training.

```
# Load unimodal models
vis = SiglipVisionModel.from_pretrained("...")
aud = WhisperEncoder.from_pretrained("...")
llm = LlamaForCausalLM.from_pretrained("...")

# Create an MLLM with modularity information
mllm = MultimodalModule(
  encoders = {
    "vision": EncoderModule(vis, proj="mlp"),
    "audio": EncoderModule(aud, proj="linear"),
    # ... more encoders
  },
  language_model = llm,
)

# Define parallel spec per modality
# either by manually or by automatically
vis_spec = ParallelSpec(...)
aud_spec = ParallelSpec(...)
llm_spec = ParallelSpec(...)

# Parallelize the MLLM
torch.distributed.init_process_group(...)
dist_mllm = MultimodalParallelModule(
  mllm,
  modality_parallelism="parallel",
  encoder_specs={
    "vision": vis_spec,
    "audio": aud_spec,
    # ... more encoders
  },
  language_model_spec=llm_spec,
  num_microbatches=...,
  microbatch_size=...,
)

# Run distributed training of MLLM
for batch in dataloader:
  output = dist_mllm.execute(batch)
  optimizer.step()
  optimizer.zero_grad()
```

Table B.1: Supported models in Cornstarch.

Modality	Model Names
LLM	Llama (3, 4) [4], Mistral [61], Mixtral [62], Gemma (1, 2) [24, 23, 25], Qwen (2, 2.5, 3) [166], Phi (3, 4) [95], InternLM2 [13]
Vision Encoder	CLIP [118], Dinov2 [109], Siglip [168], EvaCLIP [134], Pixtral [2], Qwen2Vision [149]
Audio Encoder	Whisper [119], Qwen2Audio [18], Phi4Audio [1]

Table B.2: Model execution time with inter-GPU balancing + using multiple CUDA streams.

Time (ms)	Causal CP	Inter-GPU Balance + Multistreams	Cornstarch
LLM-S	5541.25	5294.12	4856.60
LLM-M	24534.50	23794.29	22815.79
LLM-L	77378.44	76457.72	74864.71

attention head computations in the middle of each attention layer as presented in Table B.2. However, in Figure B.2, spikes are still observed at the end of every attention iteration, as the last attention head computation cannot be overlapped with the next head.

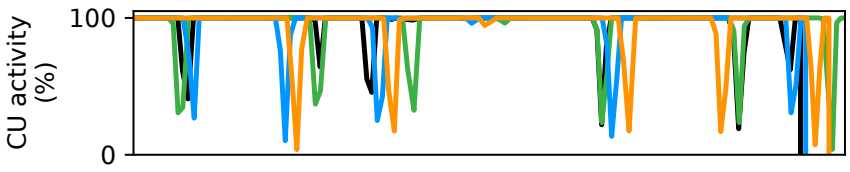
(a) Workload-balanced context parallelism with 32k sequence

Time (ms)		Causal CP	Inter-GPU Balance Only	Intra-GPU Balance Only	Cornstarch
LLM-S	Attn	66.00	74.44	57.65	57.32
	Model	1858.73	1976.68	1731.12	1705.38
LLM-M	Attn	113.37	127.46	111.17	105.16
	Model	8345.84	8789.94	8213.21	8029.79
LLM-L	Attn	148.75	159.39	141.85	143.28
	Model	29372.62	30234.54	28767.26	28518.80

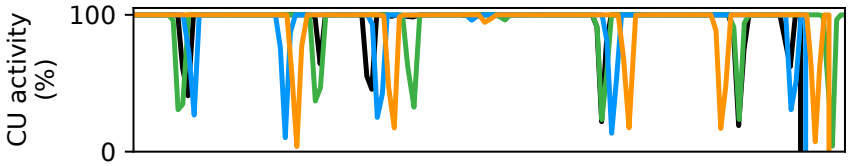
(b) Workload-balanced context parallelism with 16k sequence

Time (ms)		Causal CP	Inter-GPU Balance Only	Intra-GPU Balance Only	Cornstarch
LLM-S	Attn	23.54	24.26	17.61	18.23
	Model	800.29	802.07	691.22	705.14
LLM-M	Attn	40.01	40.72	35.35	34.47
	Model	3682.71	3700.22	3467.52	3491.53
LLM-L	Attn	48.02	50.77	44.74	41.41
	Model	13089.76	13184.58	12677.96	12628.59

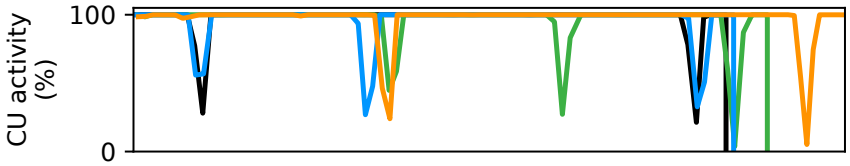
Figure B.1: Workload-balanced context parallelism with different sequence lengths.



(a) Small LLM



(b) Medium LLM



(c) Large LLM

Figure B.2: CU activity analysis with inter-GPU balancing + multiple CUDA streams.

APPENDIX C

Entrain Appendix

C.1 Termination Proof for the Loop

This appendix proves that the loop in Algorithm 2 terminates in finitely many iterations.
Setup. Let each training sample i contribute a workload vector $\mathbf{w}_i = (w_{\text{encoder},i},\, w_{\text{LLM},i})$, where $w_{\text{encoder},i}$ and $w_{\text{LLM},i}$ are the workloads of the vision encoder and LLM for that sample (following the notation of Section 4.5). Project $\mathbf{w}_i$ onto the scalar workload ratio:

$$r_i \;=\; \frac{w_{\text{encoder},i}}{w_{\text{encoder},i} + w_{\text{LLM},i}} \;\in\; [0,1] \tag{C.1}$$

which governs the proportional GPU allocation. Let $\nu_W = \mathbb{E}[r_i]$ denote the population mean. Each iteration of the algorithm draws n independent and identically distributed samples and computes the sample mean $\bar{r}_n$.
Termination proof. Since $r_i \in [0,1]$ are i.i.d. bounded random variables, the Strong Law of Large Numbers guarantees that the sample mean converges to the true mean with probability 1:

$$\bar{r}_n \;\longrightarrow\; \nu_W \quad \text{as } n \to \infty. \tag{C.2}$$

The proportional GPU allocation function $Q(\cdot)$ is piecewise constant: it rounds the continuous ratio to the nearest integer split, producing a finite number of breakpoints in $[0,1]$. Let $d > 0$ denote the distance from ν_W to the nearest breakpoint. This is positive as long as ν_W does not fall exactly on a breakpoint, which holds for any continuously-valued workload distribution.

By Equation C.2, there exists a finite threshold n^* such that for all $n \geq n^*$, every independently drawn batch of size n satisfies $|\bar{r}_n - \nu_W| < d$, and hence produces the same allocation $Q(\bar{r}_n) = Q(\nu_W)$. Therefore, once n reaches n^*, all k Bernoulli validation draws agree with the reference batch, *IsStable* = True, and the loop returns. The loop terminates in finitely many doublings.

Convergence rate. The above argument does not specify the value of n^*. Under the Central Limit Theorem approximation, one can show $n^* \leq (6\,\sigma_{pop}/d)^2$, at which point the probability of any single batch producing a different allocation falls to $\approx 0.0000002\%$. In practice, the Bernoulli test passes at batch sizes far smaller than this bound, as shown empirically in Appendix C.5.

C.2 Probabilistic Proof of Configuration Stability

This appendix formalizes the guarantee used in Section 4.4.2. The argument has two steps. Section C.2.1 bounds the probability that a random profiling batch of size n yields a configuration different from the observed reference allocation. Section C.2.2 then shows that the same decision remains valid for any larger batch size $b \geq n$, in particular the runtime global batch size B_{global}, because the workload-ratio estimator concentrates as batch size grows.

C.2.1 Bounding the Error Rate at Batch Size n

Let C_{ref} denote the discrete allocation returned by the first reference batch of size n under the chosen data-parallel degree DP. For each subsequent independent validation batch of size n, let C_n denote the allocation returned by the same procedure. Define the failure event as $C_n \neq C_{ref}$, and let

$$p_{error} = \Pr(C_n \neq C_{ref} \mid C_{ref}) \tag{C.3}$$

be the probability that a fresh size-n batch yields a different allocation than the observed reference allocation.

After fixing the reference batch, running the procedure on k additional independent validation batches gives k Bernoulli trials. The probability of observing zero failures is:

$$\Pr(0 \text{ errors in } k \text{ trials}) = (1 - p_{error})^k \tag{C.4}$$

For observing zero failures to constitute significant evidence against an error rate of p_{error}, we require $(1 - p_{error})^k \leq \alpha$. Solving the boundary case for the minimum number of trials:

$$k = \left\lceil \frac{\ln(\alpha)}{\ln(1 - p_{error})} \right\rceil \tag{C.5}$$

Therefore, observing identical configurations across k independent validation trials allows the system to state with confidence $1 - \alpha$ that a random batch of size n will differ from the observed reference allocation C_{ref} with probability at most p_{error}. For example, with $\alpha = 0.05$ and $p_{error} = 0.05$, the required number of trials is $k \approx 59$.

C.2.2 Lifting the Guarantee from n to Larger Batch Sizes

Section C.2.1 only certifies stability for random profiling batches of size n relative to the observed reference allocation C_{ref}. However, the configuration is used at training time for larger global batches, so we must show that the same decision remains valid at those larger scales.

Let each sample i contribute a workload vector $\mathbf{w}_i = (w_{\text{encoder},i},\, w_{\text{LLM},i})$. Let $\mu = \mathbb{E}[\mathbf{w}_i]$ and $\Sigma = \text{Cov}(\mathbf{w}_i)$. For any batch size $b \geq n$, the estimated macroscopic workload ratio is derived from the sample mean

$$\bar{\mathbf{w}}_b = \frac{1}{b}\sum_{i=1}^{b} \mathbf{w}_i. \tag{C.6}$$

Assuming independent and identically distributed sampling, its covariance scales as

$$\text{Cov}(\bar{\mathbf{w}}_b) = \frac{1}{b}\Sigma. \tag{C.7}$$

Thus, the estimator becomes more concentrated as b grows.

Now consider the mapping from the continuous workload ratio to a discrete configuration. Because GPUs are indivisible, this mapping partitions the workload space into piecewise-constant decision regions. Let $V(C_{ref})$ denote the region that maps to the observed reference allocation C_{ref}.

From the binomial argument in Section C.2.1, the estimator at batch size n already lands in this region with high probability:

$$\Pr(\bar{\mathbf{w}}_n \in V(C_{ref}) \mid C_{ref}) \geq 1 - p_{error}. \tag{C.8}$$

Because proportional allocation maps continuous workload ratios to integer GPU counts via rounding, each decision region $V(C_{ref})$ is a convex polytope. Moreover, the high success rate established above implies that the population mean μ lies in the interior of $V(C_{ref})$: if μ were outside or near the boundary of $V(C_{ref})$, the size-n estimator would frequently land outside the region, contradicting the Bernoulli test.

For any $b \geq n$, the estimator $\bar{\mathbf{w}}_b$ shares the same mean μ but has strictly smaller covariance $\Sigma/b \preceq \Sigma/n$. Because μ lies in the interior of the convex region $V(C_{ref})$, concentrating more tightly around μ can only increase the probability of remaining in $V(C_{ref})$. The allocation selected from profiling batches of size n therefore remains valid for any larger batch size, including the runtime global batch size B_{global} and, as a limiting case, the full dataset size N.

Once a modest profiling batch size n is large enough that repeated random draws agree

Table C.1: Parallel configurations of Entrain and the baselines on various datasets.

		DistTrain/DIP/Entrain		
		LLaVA-150k	ChartQA	CocoQA
Qwen2Vision Llama3-1b	E.PP L.PP	5/8/4 3/8/4	5/8/5 3/8/3	4/8/5 4/8/3
Qwen2Vision Llama3-3b	E.PP L.PP	4/8/3 4/8/5	4/8/4 4/8/4	4/8/3 4/8/5

on the same discrete allocation, averaging over any larger batch only reinforces that decision.

C.3 Parallel Configurations

Table 4.1 shows the parallel configuration of Entrain and the baselines on Qwen2.5Vision + Llama3-1b VLM and Qwen2.5Vision + Llama3-3b VLM. Note that for DIP, vision stages and LLM stages are colocated.

C.4 Pipeline Schedule Memory Consumption

C.4.1 Qwen2.5Vision + Llama3-1b

Figure C.1, Figure C.2, Figure C.3, and Figure C.4 show the pipeline schedule memory consumption of SynthChartNet, LLaVA-150k, ChartQA, and CocoQA datasets on Qwen2.5Vision + Llama3-1b VLM.

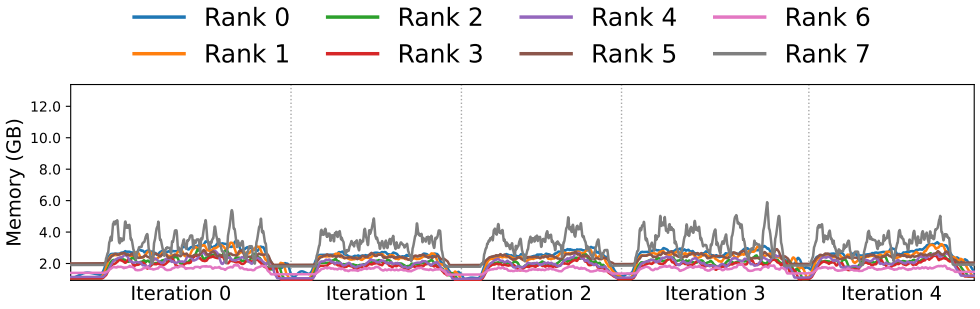
C.4.2 Qwen2.5Vision + Llama3-3b

Figure C.5, Figure C.6, Figure C.7 show the pipeline schedule memory consumption of LLaVA-150k, ChartQA, and CocoQA datasets on Qwen2.5Vision + Llama3-3b VLM.

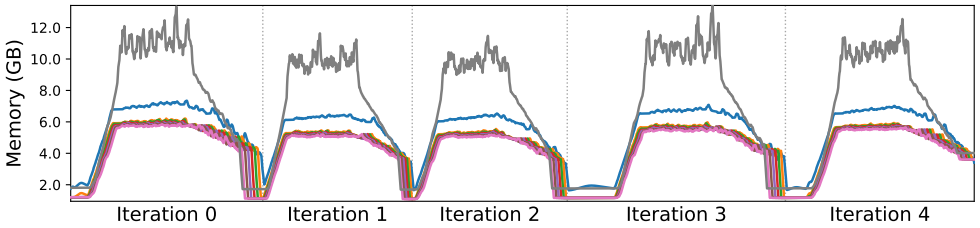
C.5 Workload Ratios in Bernoulli Trials

C.5.1 Qwen2.5Vision + Llama3-1b

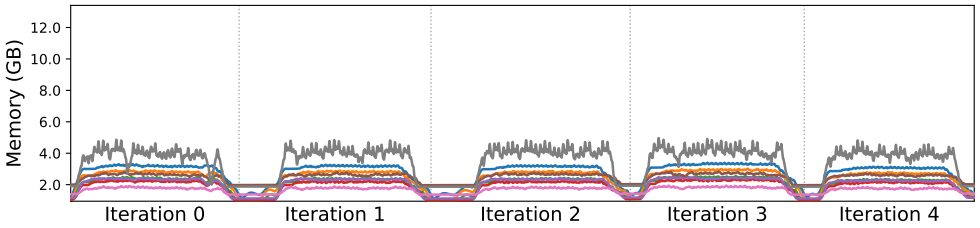
Table C.2, Table C.3, Table C.4, and Table C.5 show the workload ratios in Bernoulli trials of SynthChartNet, LLaVA-150k, ChartQA, and CocoQA datasets using Qwen2.5Vision + Llama3-1b.



(a) Memory consumption of 1F1B pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.

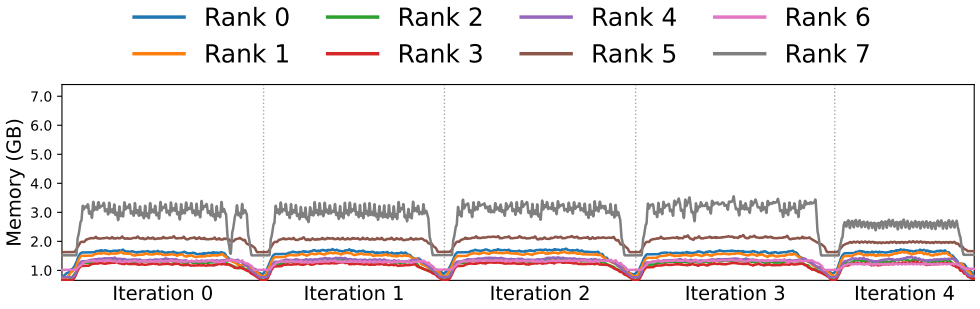


(c) Memory consumption of Entrain-1F1B pipeline schedule.

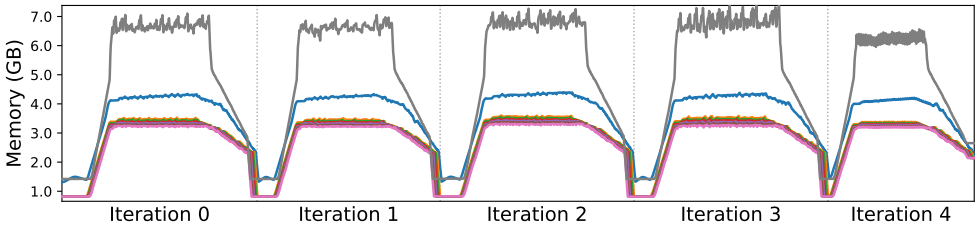
Figure C.1: SynthChartNet on Qwen2.5Vision+Llama3-1b VLM.

Table C.2: Workload ratios in Bernoulli trials of SynthChartNet dataset using Qwen2.5Vision + Llama3-1b.

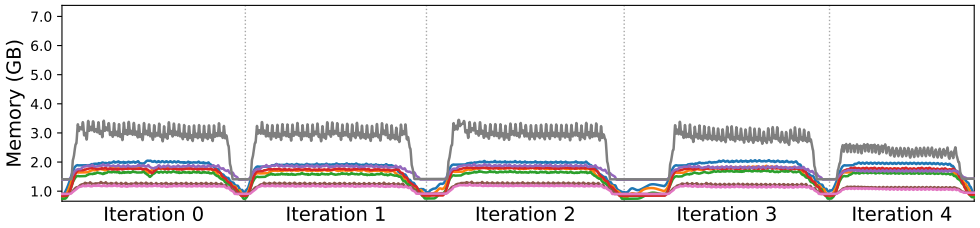
Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	11:5, 10:6, 9:7
4	✗	11:5, 10:6, 9:7
16	✗	11:5, 10:6
64	✓	10:6
256	✓	10:6



(a) Memory consumption of 1F1B pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.

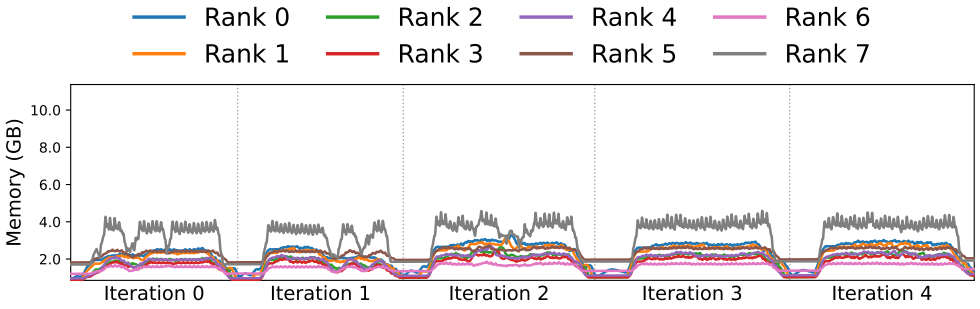


(c) Memory consumption of Entrain pipeline schedule.

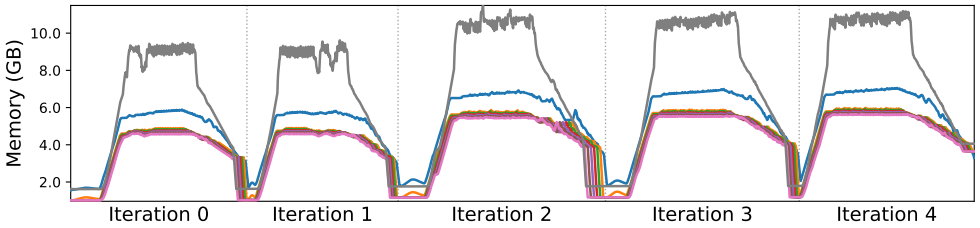
Figure C.2: LLaVA-150k on Qwen2.5Vision+Llama3-1b VLM.

Table C.3: Workload ratios in Bernoulli trials of LLaVA-150k dataset using Qwen2.5Vision + Llama3-1b.

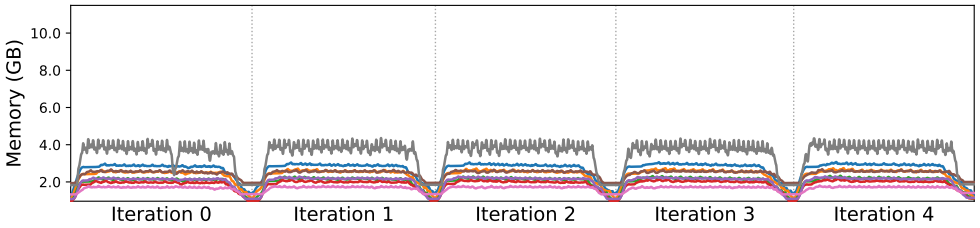
Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	10:6, 9:7, 8:8
4	✗	10:6, 9:7, 8:8
16	✗	9:7, 8:8
64	✓	9:7
256	✓	9:7



(a) Memory consumption of 1F1B pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.

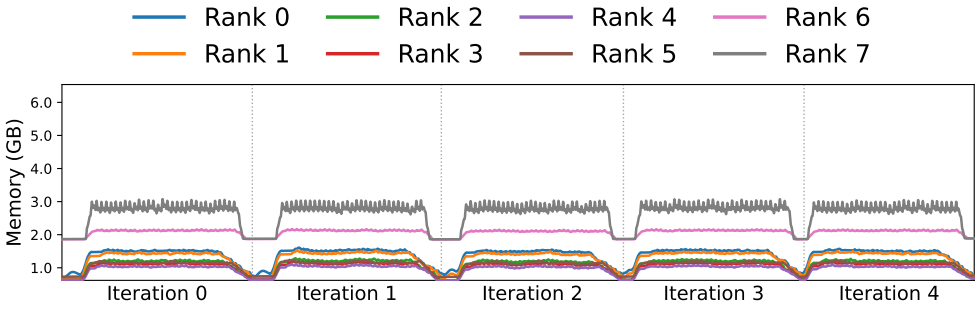


(c) Memory consumption of Entrain pipeline schedule.

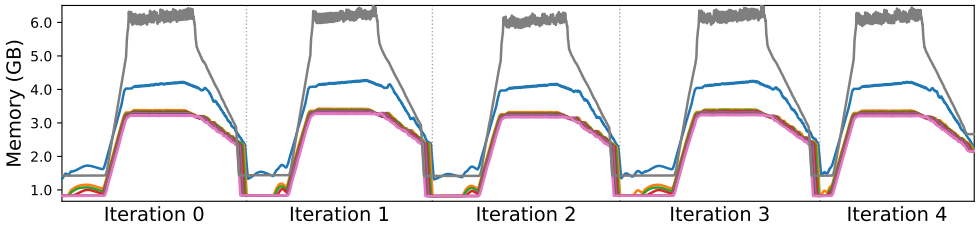
Figure C.3: ChartQA on Qwen2.5Vision+Llama3-1b VLM.

Table C.4: Workload ratios in Bernoulli trials of ChartQA dataset using Qwen2.5Vision + Llama3-1b.

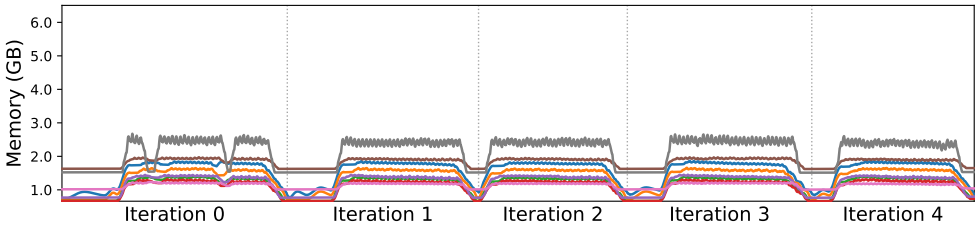
Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	10:6, 9:7
4	✗	10:6, 9:7
16	✓	10:6
64	✓	10:6
256	✓	10:6



(a) Memory consumption of 1F1B pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.

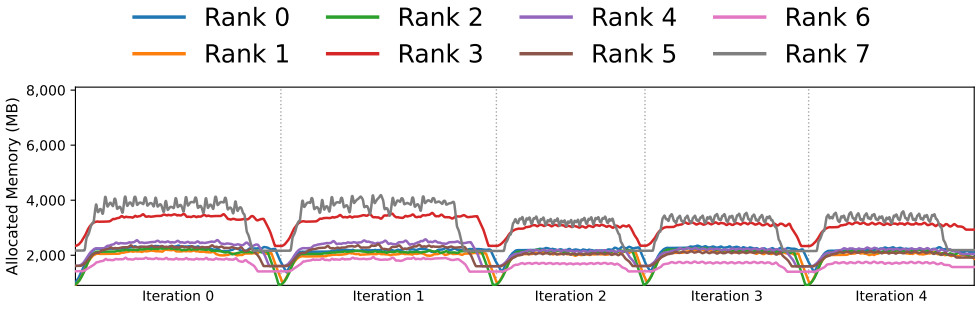


(c) Memory consumption of Entrain pipeline schedule.

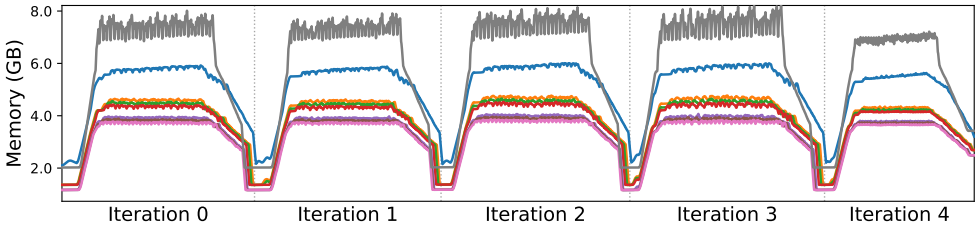
Figure C.4: CocoQA on Qwen2.5Vision+Llama3-1b VLM.

Table C.5: Workload ratios in Bernoulli trials of CocoQA dataset using Qwen2.5Vision + Llama3-1b.

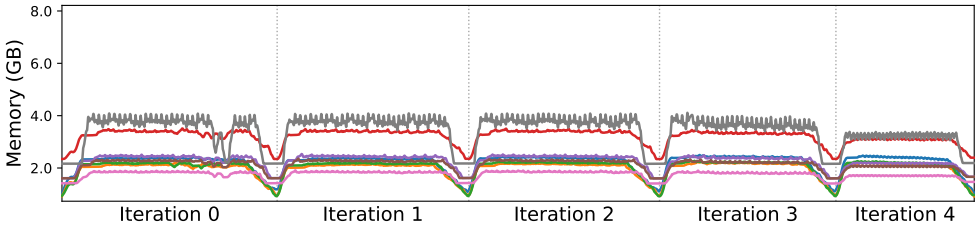
Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	10:6, 9:7
4	✗	10:6, 9:7
16	✓	10:6
64	✓	10:6
256	✓	10:6



(a) Memory consumption of 1F1B pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.

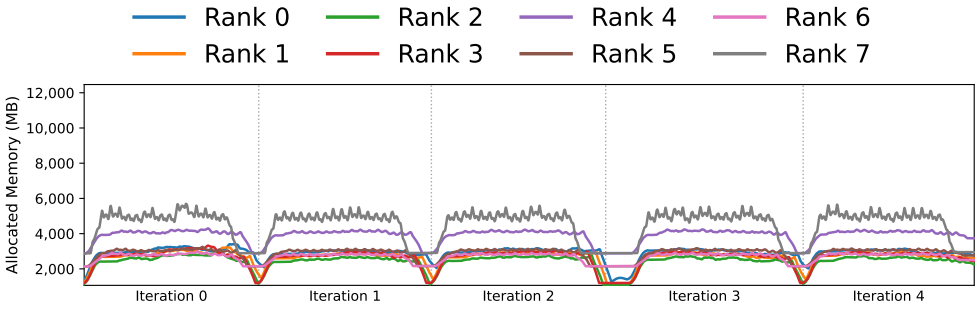


(c) Memory consumption of Entrain pipeline schedule.

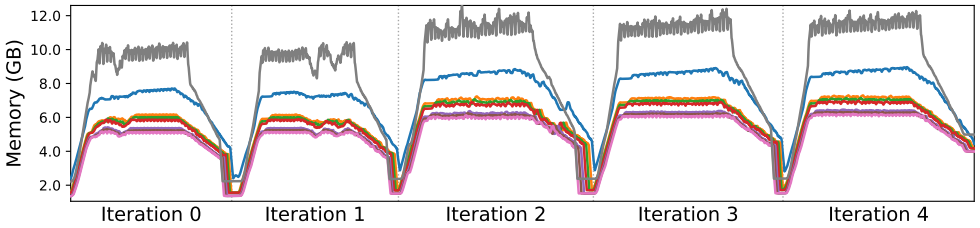
Figure C.5: LLaVA-150k on Qwen2.5Vision+Llama3-1b VLM.

C.5.2 Qwen2.5Vision + Llama3-3b

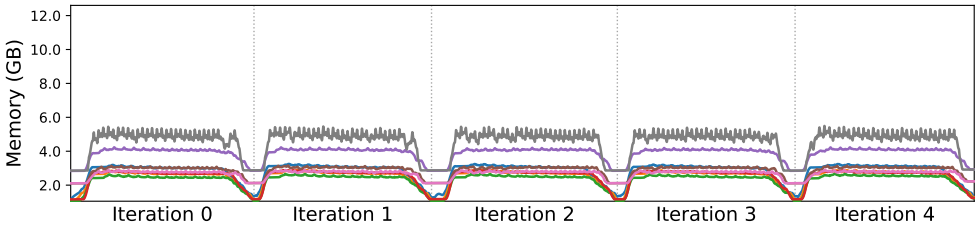
Table C.6, Table C.7, and Table C.8 show the workload ratios in Bernoulli trials of LLaVA-150k, ChartQA, and CocoQA datasets using Qwen2.5Vision + Llama3-3b.



(a) Memory consumption of 1F1B pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.

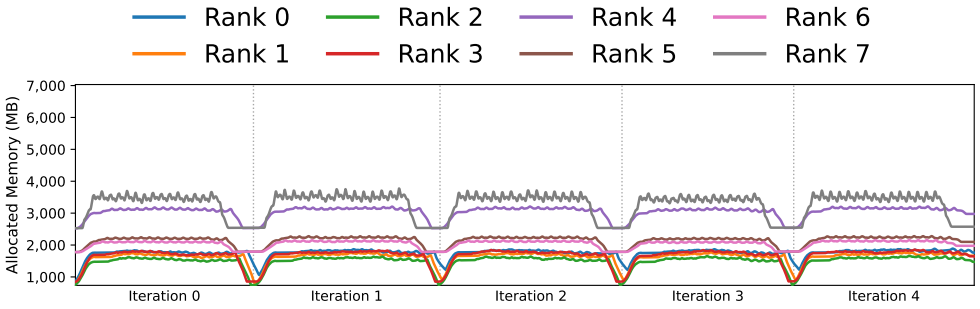


(c) Memory consumption of Entrain pipeline schedule.

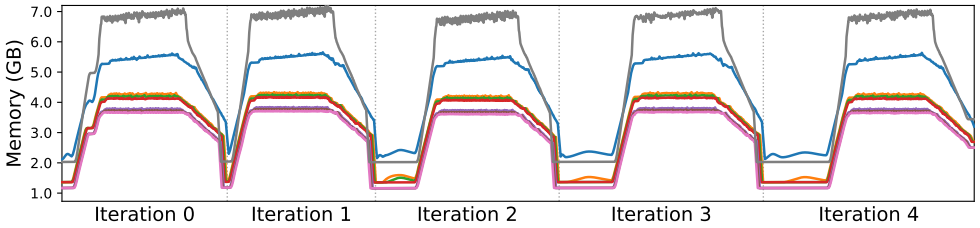
Figure C.6: ChartQA on Qwen2.5Vision+Llama3-1b VLM.

Table C.6: Workload ratios in Bernoulli trials of LLaVA-150k dataset using Qwen2.5Vision + Llama3-3b.

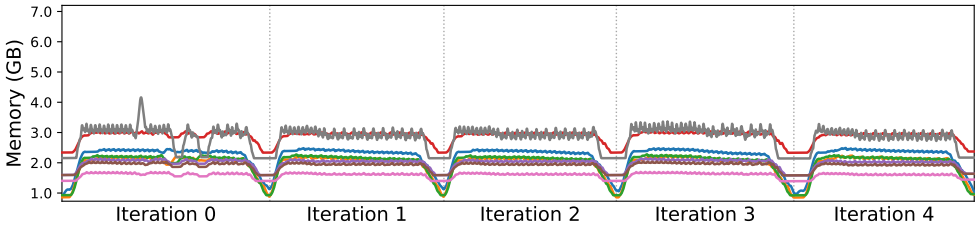
Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	7:9, 6:10, 5:11
4	✗	7:9, 6:10
16	✗	7:9, 6:10
64	✗	7:9, 6:10
256	✓	7:9



(a) Memory consumption of 1F1B pipeline schedule.



(b) Memory consumption of DIP pipeline schedule.



(c) Memory consumption of Entrain pipeline schedule.

Figure C.7: CocoQA on Qwen2.5Vision+Llama3-1b VLM.

Table C.7: Workload ratios in Bernoulli trials of ChartQA dataset using Qwen2.5Vision + Llama3-3b.

Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	8:8, 7:9
4	✓	8:8
16	✓	8:8
64	✓	8:8
256	✓	8:8

Table C.8: Workload ratios in Bernoulli trials of CocoQA dataset using Qwen2.5Vision + Llama3-3b.

Batch Size	Trial Pass	Ratios Shown (Vision:LLM)
1	✗	8:8, 7:9
4	✓	8:8
16	✓	8:8
64	✓	8:8
256	✓	8:8

C.6 Sensitivity Analysis

Figure C.8, Figure C.9, and Figure C.10 show the sensitivity analysis of the profiling batch size on LLaVA-150k, ChartQA, and CocoQA datasets.

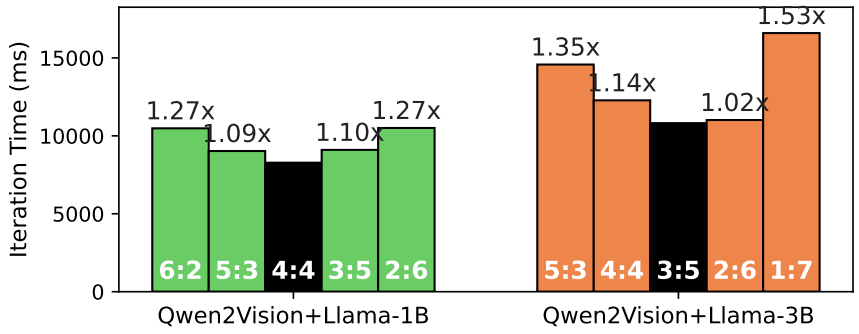


Figure C.8: Sensitivity analysis of the profiling batch size on LLaVA-150k dataset.

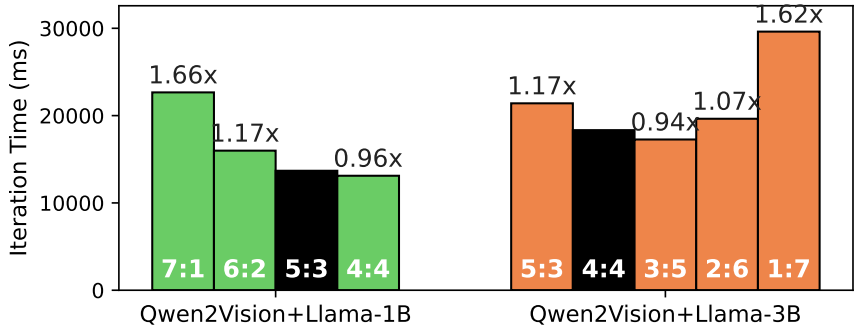


Figure C.9: Sensitivity analysis of the profiling batch size on ChartQA dataset.

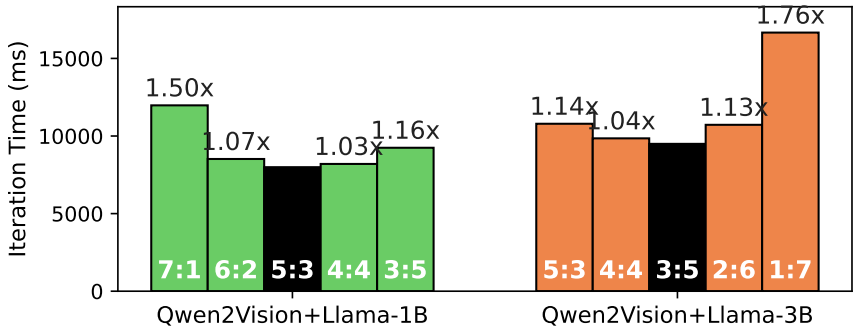
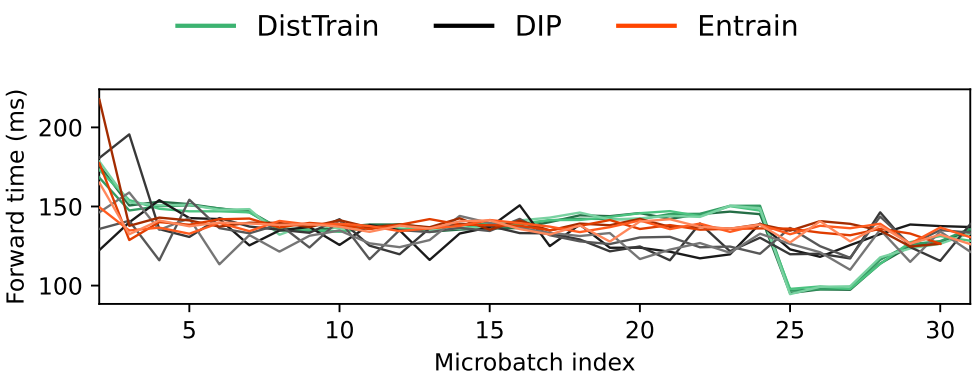


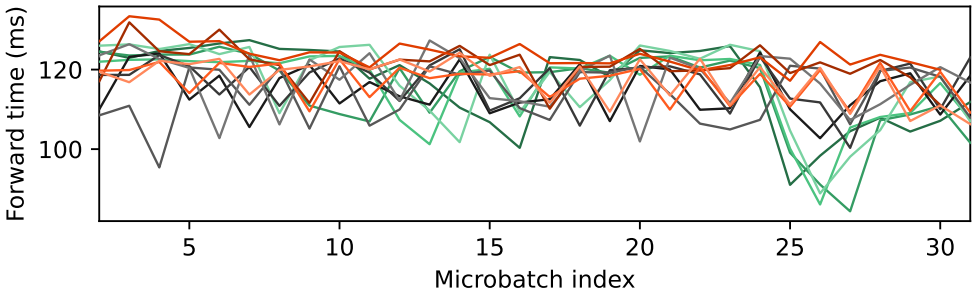
Figure C.10: Sensitivity analysis of the profiling batch size on CocoQA dataset.

C.7 Variability of Modality Forward Time Across Microbatches

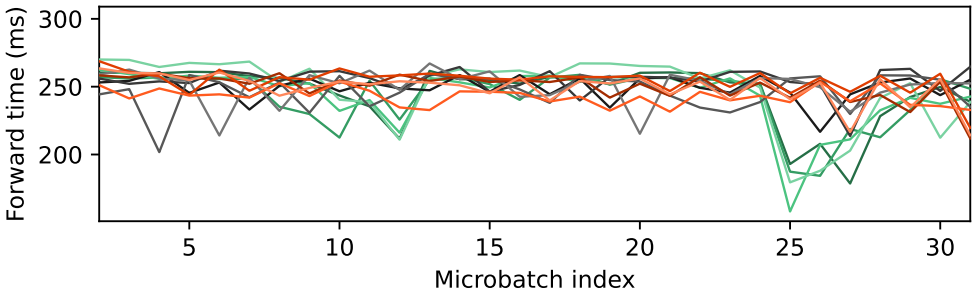
Figure C.11, Figure C.12, and Figure C.13 show the variability of modality forward time across microbatches in LLaVA-150k, ChartQA, and CocoQA datasets.



(a) Qwen2.5Vision on LLaVA-150k dataset.

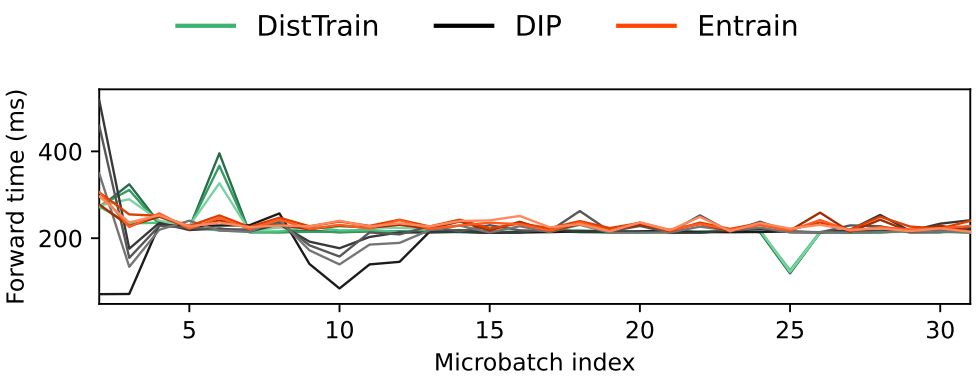


(b) Llama3-1b on LLaVA-150k dataset.

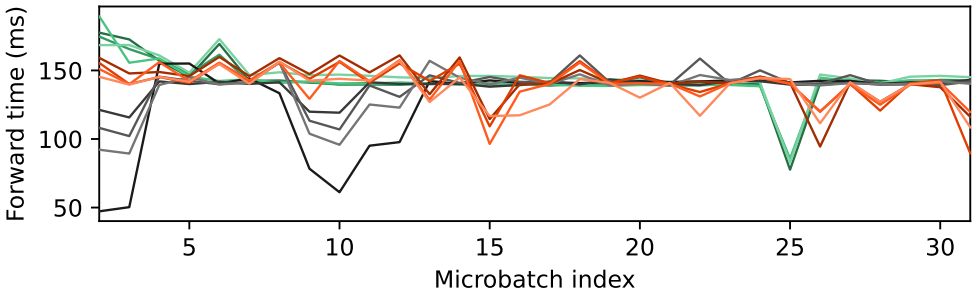


(c) Llama3-3b on LLaVA-150k dataset.

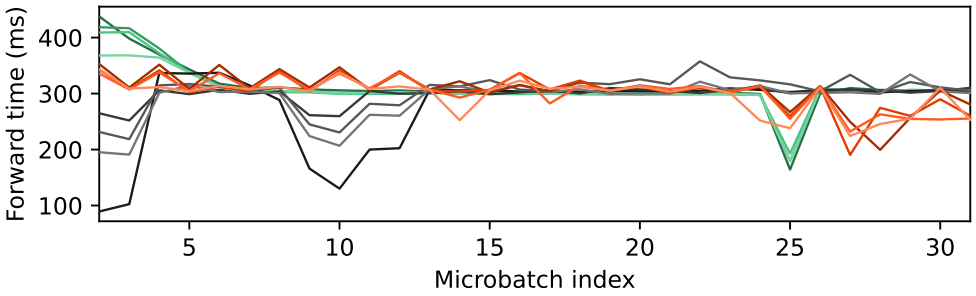
Figure C.11: Variability of modality forward time across microbatches in LLaVA-150k dataset.



(a) Qwen2.5Vision on ChartQA dataset.

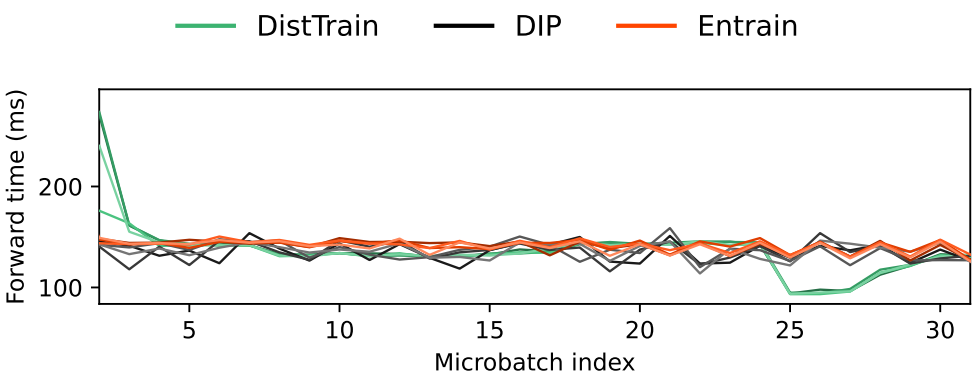


(b) Llama3-1b on ChartQA dataset.

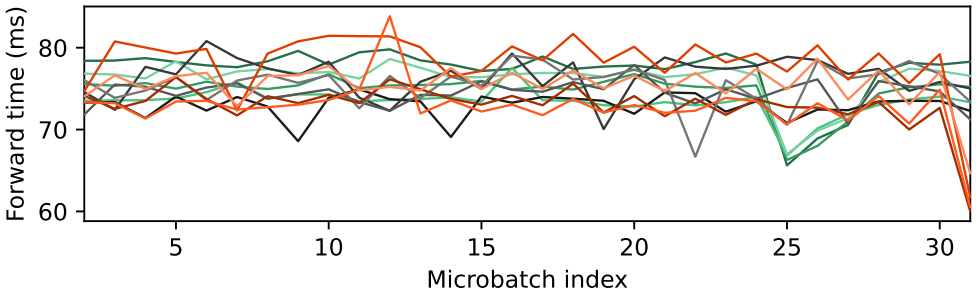


(c) Llama3-3b on ChartQA dataset.

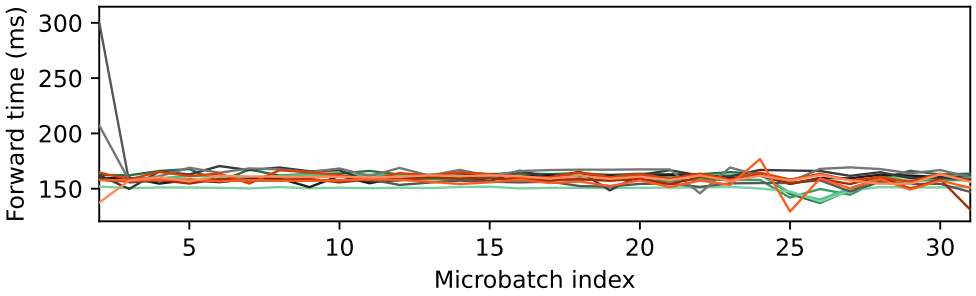
Figure C.12: Variability of modality forward time across microbatches in ChartQA dataset.



(a) Qwen2.5Vision on CocoQA dataset.



(b) Llama3-1b on CocoQA dataset.



(c) Llama3-3b on CocoQA dataset.

Figure C.13: Variability of modality forward time across microbatches in CocoQA dataset.

BIBLIOGRAPHY

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 technical report, 2024.

[2] Pravesh Agrawal, Szymon Antoniak, Emma Bou Hanna, Baptiste Bout, Devendra Chaplot, et al. Pixtral: A large-scale vision-language model with parameter-efficient fine-tuning, 2024.

[3] Inclusion AI and Ant Group. Ming-omni: A unified multimodal model for perception and generation, 2025.

[4] Meta AI. The llama 3 herd of models, 2024.

[5] Meta AI. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation, 2025.

[6] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, low-cost training of massive deep learning models. In *EuroSys 22*, 2022.

[7] Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, Jie Tang, Yuxiao Dong, and Juanzi Li. LongAlign: A recipe for long context alignment of large language models. In *EMNLP 24*, 2024.

[8] Romain Beaumont. Large scale openclip: L/14, h/14 and g/14 trained on laion-2b, 2022.

[9] Richard Bellman. On the theory of dynamic programming. *Proc. Natl. Acad. Sci.*, 1952.

[10] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers, 2023.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS 20*, 2020.

[12] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–Optimization Modeling in Python*. Springer Science & Business Media, 2021.

[13] Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, et al. Internlm2 technical report. *CoRR*, 2024.

[14] Sijin Chen, Xin Chen, Chi Zhang, Mingsheng Li, Gang Yu, Hao Fei, Hongyuan Zhu, Jiayuan Fan, and Tao Chen. Ll3da: Visual interactive instruction tuning for omni-3d understanding reasoning and planning. In *CVPR 24*, 2024.

[15] Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, Bin Li, Ping Luo, Tong Lu, Yu Qiao, and Jifeng Dai. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In *CVPR 24*, 2024.

[16] Brian Chmiel, Maxim Fishman, Ron Banner, and Daniel Soudry. FP4 all the way: Fully quantized training of large language models. In *NeurIPS 25*, 2025.

[17] Weiwei Chu, Xinfeng Xie, Jiecao Yu, Jie Wang, Amar Phanishayee, Chunqiang Tang, Yuchen Hao, Jianyu Huang, Mustafa Ozdal, Jun Wang, Vedanuj Goswami, Naman Goyal, Abhishek Kadian, Andrew Gu, Chris Cai, Feng Tian, Xiaodong Wang, Min Si, Pavan Balaji, Ching-Hsiang Chu, and Jongsoo Park. Scaling Llama 3 training with efficient parallelism strategies. In *ISCA 25*, 2025.

[18] Yunfei Chu, Jin Xu, Qian Yang, Haojie Wei, Xipin Wei, Zhifang Guo, et al. Qwen2-audio technical report, 2024.

[19] Matteo Croci, Massimiliano Fasi, Nicholas J. Higham, Theo Mary, and Mantas Mikaitis. Stochastic rounding: implementation, error analysis and applications. *Royal Soc. Open Sci.*, 2022.

[20] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *ICLR 24*, 2024.

[21] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS 22*, 2022.

[22] Anwesha Das, Frank Mueller, and Barry Rountree. Systemic assessment of node failures in hpc production platforms. In *IPDPS 21*, 2021.

[23] Google Deepmind. Gemma 2: Improving open language models at a practical size, 2024.

[24] Google Deepmind. Gemma: Open models based on gemini research and technology, 2024.

[25] Google Deepmind. Gemma 3 technical report, 2025.

[26] DeepSeek-AI. Deepseek-v3 technical report, 2025.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL 19*, 2019.

[28] Juechu Dong, Boyuan Feng, Driss Guessous, and Horace He. Flexattention: A programming model for generating fused attention variants. In *MLSys 25*, 2025.

[29] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In *NSDI 22*, 2022.

[30] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models. In *PPoPP 21*, 2021.

[31] Jiarui Fang, Jinzhe Pan, Aoyu Li, Xibo Sun, and Jiannan Wang. Pipefusion: Patch-level pipeline parallelism for diffusion transformers inference. In *NeurIPS 25*, 2025.

[32] Jiarui Fang, Jinzhe Pan, Xibo Sun, Aoyu Li, and Jiannan Wang. xdit: an inference engine for diffusion transformers (dits) with massive parallelism, 2024.

[33] Jiarui Fang and Shangchun Zhao. Usp: A unified sequence parallelism approach for long context generative ai, 2024.

[34] Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. Optimus: Accelerating Large-Scale Multi-Modal LLM training by bubble exploitation. In *ATC 25*, 2025.

[35] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. Bytescale: Communication-efficient scaling of llm training with a 2048k context length on 16384 gpus. In *SIGCOMM 25*, 2025.

[36] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. Bytescale: Efficient scaling of llm training with a 2048k context length on more than 12,000 gpus, 2025.

[37] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *SOSP 24*, 2024.

[38] GLM-5-Team. Glm-5: from vibe coding to agentic engineering, 2026.

[39] Google. Gemini: A family of highly capable multimodal models, 2023.

[40] Google. Palm 2 technical report, 2023.

[41] Google. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

[42] Ronald Lewis Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 1969.

[43] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, Yonggang Wen, Tianwei Zhang, Xin Jin, and Xuanzhe Liu. Loongtrain: Efficient training of long-sequence llms with head-context parallelism, 2024.

[44] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: Long-term measurement, analysis, and implications. In *SC 17*, 2017.

[45] William E Hart, Jean-Paul Watson, and David L Woodruff. Pyomo: Modeling and solving mathematical programs in python. *MPC*, 2011.

[46] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[47] Wenyi Hong, Weihan Wang, Ming Ding, Wenmeng Yu, Qingsong Lv, Yan Wang, Yean Cheng, Shiyu Huang, Junhui Ji, Zhao Xue, Lei Zhao, Zhuoyi Yang, Xiaotao Gu, Xiaohan Zhang, Guanyu Feng, Da Yin, Zihan Wang, Ji Qi, Xixuan Song, Peng Zhang, Debing Liu, Bin Xu, Juanzi Li, Yuxiao Dong, and Jie Tang. Cogvlm2: Visual language models for image and video understanding, 2024.

[48] Horovod. Elastic horovod, 2019.

[49] Jun Huang, Zhen Zhang, Shuai Zheng, Feng Qin, and Yida Wang. DISTMM: Accelerating distributed multimodal model training. In *NSDI 24*, 2024.

[50] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS 19*, 2019.

[51] HuggingFace. Hugging face: The ai community building the future, 2024.

[52] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale. In *MLSys 23*, 2023.

[53] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *NSDI 21*, 2021.

[54] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.

[55] Insu Jang and Mosharaf Chowdhury. Addressing variable heterogeneity in distributed multimodal training with entrain, 2026.

[56] Insu Jang, Runyu Lu, Nikhil Bansal, Ang Chen, and Mosharaf Chowdhury. Efficient distributed mllm training with cornstarch. In *ICML 26*, 2025.

[57] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *SOSP 23*, 2023.

[58] Byungsoo Jeon, Mengdi Wu, Shiyi Cao, Sunghyun Kim, Sunghyun Park, Neeraj Aggarwal, Colin Unger, Daiyaan Arfeen, Peiyuan Liao, Xupeng Miao, Mohammad Alizadeh, Gregory R. Ganger, Tianqi Chen, and Zhihao Jia. Graphpipe: Improving performance and scalability of dnn training with graph pipeline parallelism. In *ASPLOS 25*, 2025.

[59] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *ATC 19*, 2019.

[60] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous gpus. In *ATC 22*, 2022.

[61] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, et al. Mistral 7b, 2023.

[62] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, et al. Mixtral of experts, 2024.

[63] Chenyu Jiang, Zhenkun Cai, Ye Tian, Zhen Jia, Yida Wang, and Chuan Wu. DCP: Addressing input dynamism in long-context training via dynamic context parallelism. In *SOSP 25*, 2025.

[64] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, et al. Megascale: Scaling large language model training to more than 10,000 gpus. In *NSDI 24*, 2024.

[65] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[66] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *MLSys 23*, 2023.

[67] Matej Kosec, Mario Michael Krell, Sergio P. Perez, and Andrew Fitzgibbon. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance, 2022.

[68] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models. *TPDS*, 2023.

[69] Hugo Laurençon, Andrés Marafioti, Victor Sanh, and Leo Tronchon. Building and better understanding vision-language models: insights and future directions. In *NeurIPS Workshop RBFM 24*, 2024.

[70] Sungjae Lee, Jaeil Hwang, and Kyungyong Lee. Spotlake: Diverse spot instance dataset archive service. In *IISWC 22*, 2022.

[71] Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Peiyuan Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. LLaVA-onevision: Easy visual task transfer. *TMLR*, 2025.

[72] Dacheng Li, Rulin Shao, Anze Xie, Eric P. Xing, Xuezhe Ma, Ion Stoica, Joseph E. Gonzalez, and Hao Zhang. Distflashattn: Distributed memory-efficient attention for long-context llms training. In *CoLM 24*, 2024.

[73] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Aryl: An elastic cluster scheduler for deep learning, 2022.

[74] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. BLIP-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *ICML 23*, 2023.

[75] Muyang Li, Tianle Cai, Jiaxin Cao, Qinsheng Zhang, Han Cai, Junjie Bai, Yangqing Jia, Kai Li, and Song Han. Distrifusion: Distributed parallel inference for high-resolution diffusion models. In *CVPR 24*, 2024.

[76] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *ICPP 23*, 2023.

[77] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *ACL 23*, 2023.

[78] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *SC 21*, 2021.

[79] Zongxia Li, Xiyang Wu, Hongyang Du, Fuxiao Liu, Huy Nghiem, and Guangyao Shi. A survey of state of the art large vision language models: Benchmark evaluations and challenges. In *CVPR 25*, 2025.

[80] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *KDD 22*, 2022.

[81] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. Torchtitan: One-stop pytorch native solution for production ready LLM pretraining. In *ICLR 25*, 2025.

[82] Bin Lin, Yang Ye, Bin Zhu, Jiaxi Cui, Munan Ning, Peng Jin, and Li Yuan. Video-llava: Learning united visual representation by alignment before projection, 2024.

[83] Ji Lin, Hongxu Yin, Wei Ping, Pavlo Molchanov, Mohammad Shoeybi, and Song Han. Vila: On pre-training for visual language models. In *CVPR 24*, 2024.

[84] Hao Liu and Pieter Abbeel. Blockwise parallel transformers for large context models. In *NeurIPS 23*, 2023.

[85] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *ICLR 24*, 2024.

[86] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. In *CVPR 24*, 2024.

[87] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In *NeurIPS 23*, 2023.

[88] Zuyan Liu, Yuhao Dong, Jiahui Wang, Ziwei Liu, Winston Hu, Jiwen Lu, and Yongming Rao. Ola: Pushing the frontiers of omni-modal language model, 2025.

[89] Haoyu Lu, Wen Liu, Bo Zhang, Bingxuan Wang, Kai Dong, Bo Liu, Jingxiang Sun, Tongzheng Ren, Zhuoshu Li, Hao Yang, Yaofeng Sun, Chengqi Deng, Hanwei Xu, Zhenda Xie, and Chong Ruan. DeepSeek-VL: Towards real-world vision-language understanding, 2024.

[90] Runyu Lu, Shiqi He, Wenxuan Tan, Shenggui Li, Ruofan Wu, Jeff J. Ma, Ang Chen, and Mosharaf Chowdhury. Tetriserve: Efficiently serving mixed dit workloads. In *ASPLOS 26*, 2026.

[91] Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. ChartQA: A benchmark for question answering about charts with visual and logical reasoning. In *ACL 22*, 2022.

[92] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *ICLR 17*, 2017.

[93] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *VLDB*, 2022.

[94] Microsoft. Varuna, 2022.

[95] Microsoft. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[96] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax, 2018.

[97] MinIO. Minio: High performance object storage for ai, 2023.

[98] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained dnn checkpointing. In *FAST 21*, 2021.

[99] Hyoungwook Nam, Gerasimos Gerogiannis, and Josep Torrellas. Meshslice: Efficient 2d tensor parallelism for distributed dnn training. In *ISCA 25*, 2025.

[100] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *SOSP 19*, 2019.

[101] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *ICML 21*, 2021.

[102] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *SC 21*, 2021.

[103] Ahmed Nassar, Matteo Omenetti, Maksym Lysak, Nikolaos Livathinos, Christoph Auer, Lucas Morin, Rafael Teixeira de Lima, Yusik Kim, A. Said Gurbuz, Michele Dolfi, and Peter W. J. Staar. Smoldocling: An ultra-compact vision-language model for end-to-end multi-modal document conversion. In *ICCV 25*, 2025.

[104] NVIDIA. Megatron-lm, 2024.

[105] OpenAI. Triton: Open-source gpu programming for neural networks, 2021.

[106] OpenAI. Gpt-4 technical report, 2024.

[107] OpenAI. Gpt-4o system card, 2024.

[108] OpenAI. Openai gpt-5 system card, 2026.

[109] Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy V. Vo, Marc Szafraniec, et al. Dinov2: Learning robust visual features without supervision. *TMLR*, 2024.

[110] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism. In *ATC 20*, 2020.

[111] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS 19*, 2019.

[112] Guilherme Penedo, Hynek Kydlíček, Vinko Sabolčec, Bettina Messmer, Negar Foroutan, Amir Hossein Kargaran, Colin Raffel, Martin Jaggi, Leandro Von Werra, and Thomas Wolf. Fineweb2: One pipeline to scale them all - adapting pre-training data processing to every language. In *COLM 25*, 2025.

[113] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, Ruihang Li, Miaosen Zhang, Chen Li, Jia Ning, Ruizhe Wang, Zheng Zhang, Shuguang Liu, Joe Chau, Han Hu, and Peng Cheng. Fp8-lm: Training fp8 large language models, 2023.

[114] PyTorch. Torch elastic, 2020.

[115] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *ICLR 24*, 2024.

[116] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI 21*, 2021.

[117] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

[118] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, et al. Learning transferable visual models from natural language supervision. In *ICML 21*, 2021.

[119] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *ICML 23*, 2023.

[120] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskeve. Language models are unsupervised multitask learners, 2019.

[121] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *ICML 22*, 2022.

[122] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC 20*, 2020.

[123] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *SC 21*, 2021.

[124] Jorge L. Ramírez Alfonsín. *The Diophantine Frobenius Problem*. Oxford University Press, 2005.

[125] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD 20*, 2020.

[126] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. torch.fx: Practical program capture and transformation for deep learning in python. In *MLSys 22*, 2022.

[127] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *ATC 21*, 2021.

[128] Mengye Ren, Ryan Kiros, and Richard Zemel. Exploring models and data for image question answering. In *NIPS 15*, 2015.

[129] Joseph B. Roberts. Note on linear forms. *Proc. Amer. Math. Soc.*, 1956.

[130] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. *TDSC*, 2010.

[131] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In *ICS 23*, 2023.

[132] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model, 2022.

[133] Dan Su, Kezhi Kong, Ying Lin, Joseph Jennings, Brandon Norick, Markus Kliegl, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. Nemotron-cc: Transforming common crawl into a refined long-horizon pretraining dataset. In *ACL 25*, 2025.

[134] Quan Sun, Yuxin Fang, Ledell Wu, Xinlong Wang, and Yue Cao. Eva-clip: Improved training techniques for clip at scale, 2023.

[135] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In *NeurIPS 19*, 2019.

[136] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *ASPLOS 24*, 2024.

[137] UCLA System. Bamboo, 2023.

[138] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In *NeurIPS 21*, 2021.

[139] Kimi Team. Kimi k2: Open agentic intelligence, 2025.

[140] Ling Team. Every step evolves: Scaling reinforcement learning for trillion-scale thinking model, 2025.

[141] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. In *NSDI 23*, 2023.

[142] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *MAPL 19*, 2019.

[143] Shengbang Tong, Ellis Brown, Penghao Wu, Sanghyun Woo, Manoj Middepogu, Sai Charitha Akula, Jihan Yang, Shusheng Yang, Adithya Iyer, Xichen Pan, Austin Wang, Rob Fergus, Yann LeCun, and Saining Xie. Cambrian-1: A fully open, vision-centric exploration of multimodal llms. *CoRR*, 2024.

[144] 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 *OSDI 22*, 2022.

[145] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS 17*, 2017.

[146] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap, 2022.

[147] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. Tesseract: Parallelize the tensor parallelism efficiently. In *ICPP 22*, 2022.

[148] Guoxia Wang, Jinle Zeng, Xiyuan Xiao, Siming Wu, Jiabin Yang, Lujing Zheng, Zeyu Chen, Jiang Bian, Dianhai Yu, and Haifeng Wang. Flashmask: Efficient and rich mask extension of flashattention. In *ICLR 25*, 2025.

[149] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution, 2024.

[150] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jiashi Li, Faming Wu, and Bin Cui. FlexSP: Accelerating Large Language Model training via flexible sequence parallelism. In *ASPLOS 25*, 2025.

[151] 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. In *OSDI 25*, 2025.

[152] Maurice Weber, Daniel Y. Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Ré, Irina Rish, and Ce Zhang. Redpajama: an open dataset for training large language models. In *NeurIPS 24*, 2024.

[153] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *NSDI 22*, 2022.

[154] Luis Wiedmann, Orr Zohar, Amir Mahla, Xiaohan Wang, Rui Li, Thibaud Frere, Leandro von Werra, Aritra Roy Gosthipaty, and Andrés Marafioti. Finevision: Open data is all you need, 2025.

[155] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *EMNLP 20*, 2020.

[156] BigScience Workshop. Bloom: A 176b-parameter open-access multilingual language model, 2023.

[157] Zhiyu Wu, Xiaokang Chen, Zizheng Pan, Xingchao Liu, Wen Liu, Damai Dai, Huazuo Gao, Yiyang Ma, Chengyue Wu, Bingxuan Wang, Zhenda Xie, Yu Wu, Kai Hu, Jiawei Wang, Yaofeng Sun, Yukun Li, Yishi Piao, Kang Guan, Aixin Liu, Xin Xie, Yuxiang You, Kai Dong, Xingkai Yu, Haowei Zhang, Liang Zhao, Yisong Wang, and Chong Ruan. DeepSeek-VL2: Mixture-of-experts vision-language models for advanced multimodal understanding, 2024.

[158] Zhiyuan Wu, Shuai Wang, Li Chen, Kaihui Gao, Dan Li, Yanyu Ren, Qiming Zhang, and Yong Wang. Communication-efficient serving for video diffusion models with latent parallelism, 2025.

[159] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. Elan: Towards generic and efficient elastic training for deep learning. In *ICDCS 20*, 2020.

[160] Hu Xu, Gargi Ghosh, Po-Yao Huang, Prahal Arora, Masoumeh Aminzadeh, Christoph Feichtenhofer, Florian Metze, and Luke Zettlemoyer. Vlm: Task-agnostic video-language model pre-training for video understanding. In *ACL Findings 21*, 2021.

[161] Jin Xu, Zhifang Guo, Jinzheng He, Hangrui Hu, Ting He, Shuai Bai, Keqin Chen, Jialin Wang, Yang Fan, Kai Dang, Bin Zhang, Xiong Wang, Yunfei Chu, and Junyang Lin. Qwen2.5-omni technical report, 2025.

[162] Jin Xu, Zhifang Guo, Hangrui Hu, Yunfei Chu, Xiong Wang, Jinzheng He, Yuxuan Wang, Xian Shi, Ting He, Xinfa Zhu, Yuanjun Lv, Yongqi Wang, Dake Guo, He Wang, Linhan Ma, Pei Zhang, Xinyu Zhang, Hongkun Hao, Zishan Guo, Baosong Yang, Bin Zhang, Ziyang Ma, Xipin Wei, Shuai Bai, Keqin Chen, Xuejing Liu, Peng Wang, Mingkun Yang, Dayiheng Liu, Xingzhang Ren, Bo Zheng, Rui Men, Fan Zhou, Bowen Yu, Jianxin Yang, Le Yu, Jingren Zhou, and Junyang Lin. Qwen3-omni technical report, 2025.

[163] Zhenliang Xue, Hanpeng Hu, Xing Chen, Yimin Jiang, Yixin Song, Zeyu Mi, Yibo Zhu, Daxin Jiang, Yubin Xia, and Haibo Chen. DIP: Efficient large multimodal model training with dynamic interleaved pipeline. In *ASPLOS 26*, 2026.

[164] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jongsoo Park, and Jianyu Huang. Context parallelism for scalable million-token inference. In *MLSys 25*, 2025.

[165] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[166] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, et al. Qwen2 technical report. *CoRR*, 2024.

[167] Yongqiang Yao, Jingru Tan, Kaihuan Liang, Feizhao Zhang, Jiahao Hu, Shuo Wu, Yazhe Niu, Ruihao Gong, Dahua Lin, and Ningyi Xu. Hierachical balance packing: Towards efficient supervised fine-tuning for long-context LLM. In *NeurIPS 25*, 2025.

[168] Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. Sigmoid loss for language image pre-training. In *ICCV 23*, 2023.

[169] Duzhen Zhang, Yahan Yu, Jiahua Dong, Chenxing Li, Dan Su, Chenhui Chu, and Dong Yu. Mm-llms: Recent advances in multimodal large language models. In *ACL Findings 24*, 2024.

[170] Jinbin Zhang, Nasib Ullah, Erik Schultheis, and Rohit Babbar. ELMO : Efficiency via low-precision and peak memory optimization in large output spaces. In *ICML 25*, 2025.

[171] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, et al. Opt: Open pre-trained transformer language models, 2022.

[172] Tianyu Zhang, Kaige Liu, Jack Kosaian, Juncheng Yang, and Rashmi Vinayak. Efficient fault tolerance for recommendation model training via erasure coding. *VLDB*, 2023.

[173] Zili Zhang, Yinmin Zhong, Yimin Jiang, Hanpeng Hu, Jianjian Sun, Zheng Ge, Yibo Zhu, Daxin Jiang, and Xin Jin. Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models. In *SIGCOMM 25*, 2025.

[174] Fei Zhao, Chengcui Zhang, and Baocheng Geng. Deep multimodal data fusion. *ACM Comput. Surv.*, 2024.

[175] Siyan Zhao, Daniel Mingyi Israel, Guy Van den Broeck, and Aditya Grover. Prepacking: A simple method for fast prefilling and increased throughput in large language models. In *AISTATS 25*, 2025.

[176] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *VLDB*, 2023.

[177] 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 *OSDI 22*, 2022.

[178] Bin Zhu, Bin Lin, Munan Ning, Yang Yan, Jiaxi Cui, HongFa Wang, Yatian Pang, Wenhao Jiang, Junwu Zhang, Zongwei Li, Cai Wan Zhang, Zhifeng Li, Wei Liu, and Li Yuan. Languagebind: Extending video-language pretraining to n-modality by language-based semantic alignment. In *ICLR 24*, 2024.