Cornstarch: Distributed Multimodal Training Must Be Multimodality-Aware

Insu Jang Runyu Lu Nikhil Bansal Ang Chen Mosharaf Chowdhury
University of Michigan

Abstract

Multimodal large language models (MLLMs) extend the capabilities of large language models (LLMs) by combining heterogeneous model architectures to handle diverse modalities like images and audio. However, this inherent heterogeneity in MLLM model structure and data types makes makeshift extensions to existing LLM training frameworks unsuitable for efficient MLLM training.

In this paper, we present Cornstarch, the first general-purpose distributed MLLM training framework. Cornstarch facilitates modular MLLM construction, enables composable parallelization of constituent models, and introduces MLLM-specific optimizations to pipeline and context parallelism for efficient distributed MLLM training. Our evaluation shows that Cornstarch outperforms state-of-the-art solutions by up to 1.57× in terms of training throughput.

Cornstarch is an open-source project available at https://github.com/cornstarch-org/Cornstarch.

1 Introduction

Large language models (LLMs) have seen unprecedented adoption over the past two years. Building on this momentum, multimodal large language models (MLLMs) aim to extend LLMs' reasoning capabilities to perform complex tasks across various data modalities, such as images and audio [2, 6, 7, 9, 28, 29, 33, 34, 45, 52, 54, 56, 59, 63]. For instance, MLLMs are increasingly used in healthcare to analyze medical images and patient records, aiding in accurate diagnoses [3, 30, 35]. In robotics, they are used to process visual and auditory inputs, enabling robots to interact with their environment [48, 64]. As the volume of multimodal data continues to grow, the importance of MLLMs will only increase.

MLLM training fundamentally differs from LLM training in both data and model structure. While LLMs process only text data, MLLMs accept multiple modalities as input, each requiring varying amounts of computation. These inputs are typically processed by separate encoder models before being fed into the LLM. Consequently, MLLMs, which combine multiple modality encoders and an LLM, are more memory and computation-intensive and more heterogeneous than LLMs. This increased computational burden has changed MLLM training methods. While the entire model can still be trained from scratch like LLMs, MLLM training often starts with pretrained encoders and LLMs that may be frozen, focusing on training newly added projector models to align the pretrained models.

The larger size of MLLMs and the need for more data processing power make distributed training essential. However, there is no existing general-purpose distributed training framework optimized for MLLMs. A few LLM training frameworks, such as Megatron-LM [38], have been extended in attempts to support distributed MLLM training with 3D parallelism (pipeline, tensor, and data parallelism). Meta also used 3D parallelism designed for training LLMs to train their multimodal Llama [2]. However, these frameworks do not consider the unique characteristics of MLLMs, leading to several inefficiencies in distributed training.

1. *False encoder dependency:* Modality encoders in an MLLM process different types of data independently, yet they are executed sequentially with false dependency in existing training frameworks to maintain a chain-like execution flow similar to LLM training.
2. *Neglecting frozen status:* MLLMs include pretrained models that may remain frozen during training. Considering this in the MLLM parallelization process is crucial for high throughput but has so far been overlooked in existing frameworks.
3. *Lack of context parallelism support:* Existing context parallelism mechanisms that focus on text token distribution in full or causal relationships found in LLMs are inadequate for MLLMs. The intricate interactions between multimodal tokens in MLLMs require memory-efficient multimodal attention representation and compute-efficient token distribution mechanisms.

In this paper, we introduce Cornstarch, which, to the best of our knowledge, is the first multimodality-aware distributed training framework. Cornstarch is designed to be modular and flexible, allowing users to easily construct an MLLM by *gluing together individual unimodal models*. It also provides a modular interface for distributed MLLM training, designed to be aware of the unique characteristics of MLLMs.

We address the false dependency between encoders by introducing a novel parallelism dimension unique to MLLMs: *modality parallelism*. It enables the parallel execution of independent modality encoders. We augment existing parallelism dimensions too – namely, pipeline parallelism and context parallelism – to accommodate MLLM-specific heterogeneity, thereby optimizing workload distribution and increasing

training throughput. Specifically, for pipeline parallelism, we examine the impact of the frozen status of constituent models within an MLLM and their placement within the overall structure, leading to optimized pipeline stage partitioning. For context parallelism, we propose a compact multimodal attention representation and a novel token distribution algorithm that together improve the throughput of multimodality-aware context parallelism.

We have implemented Cornstarch on top of PyTorch [42], HuggingFace Transformers [55], and Colossal-AI [25]. Cornstarch can be used to construct and train more than 10,000 different MLLMs using various types and size of encoders and LLMs individually available on HuggingFace. Our extensive evaluation on MLLMs of varying sizes and modality combinations shows that Cornstarch outperforms existing MLLM training approaches (Megatron-LM and Meta-Llama) by up to 1.57× in terms of training throughput.

Overall, we make the following contributions in this paper:

- We present Cornstarch, which, to the best of our knowledge, is the first general-purpose distributed training framework tailored for MLLMs.
- Cornstarch introduces a novel parallelization dimension (modality parallelism) and enhances existing dimensions (pipeline parallelism and context parallelism) for efficient MLLM training.
- We conduct extensive evaluations on a variety of MLLMs, demonstrating that Cornstarch surpasses current MLLM training frameworks in both training time and scalability.

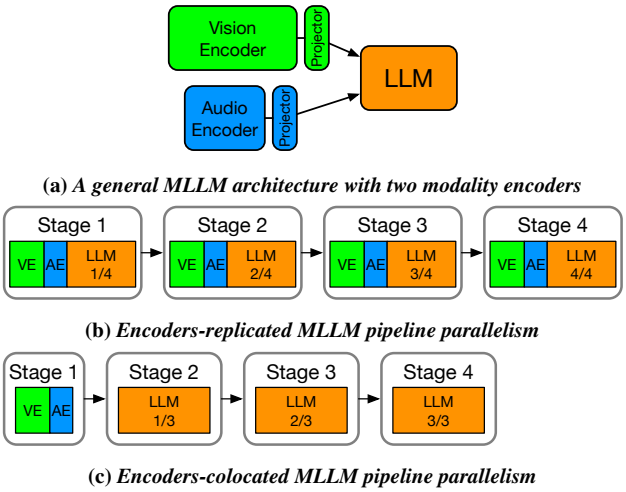
2 Background and Motivation

We first introduce the background of MLLMs (§2.1), followed by existing approaches for distributed MLLM training (§2.2). Next, we discuss their shortcomings, which motivate the need for a multimodality-aware distributed training framework (§2.3).

2.1 Multimodal Large Language Models (MLLMs)

MLLMs exploit the reasoning capabilities of LLMs to understand various types of inputs and generate outputs [59, 63]. There are many MLLMs supporting diverse modalities; e.g. vision language models (VLMs) [2, 7, 28, 29, 56], audio language models (ALMs) [10, 45], 3D point clouds language models [6], and others. To reduce training costs, it is common to use readily available pretrained modality encoders and an LLM – all frozen – and only train projectors between them to align their embedding spaces [59]. Thereafter, the whole model is fine-tuned for a downstream task.

MLLMs are distinct from LLMs because different modalities within the same MLLM have different characteristics. The sizes of unimodal models, their arithmetic intensities, and their data types are heterogeneous [15, 26]. Consequently, the interactions between modalities are more complex than causal interactions found in LLMs, which simply attend to all previous tokens.



(a) ***A general MLLM architecture with two modality encoders***

(b) ***Encoders-replicated MLLM pipeline parallelism***

(c) ***Encoders-colocated MLLM pipeline parallelism***

Figure 1: ***Existing distributed MLLM pipeline parallelism (PP) executing an MLLM represented as a directed-acyclic graph (DAG) in (a). Encoders-replicated PP has redundant computation problem, while Encoders-colocated PP has an imbalance problem.***

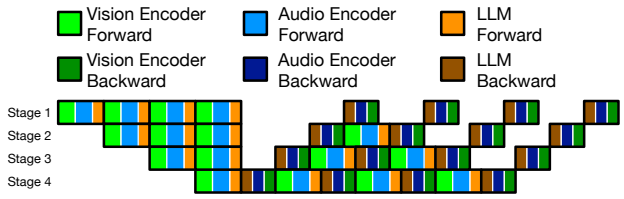
2.2 Distributed Training of MLLMs

Large-scale LLM training is a well-studied topic and leverages four parallelism dimensions – data, tensor, pipeline, and context parallelism – to achieve high training throughput [2,22,27,38,47]. Tensor and pipeline parallelism partitions the model within each layer and across layers, respectively. Data and context parallelism partitions data; the former partitions a large batch of sequences into smaller minibatches, while the latter partitions each input sequence into segments.

In contrast, distributed MLLM training is challenging due to the model size and the heterogeneity in MLLM model structures and data types. Existing efforts differ in how they parallelize heterogeneous MLLM layers in the pipeline parallelism dimension and can be divided into two broad categories: *encoders-replicated* and *encoders-colocated*.

Encoders-replicated pipeline parallelism. The Llama training team at Meta adapted and used LLM pipeline parallelism for multimodal Llama training [2]. They first partition the LLM using pipeline parallelism for LLMs, and then replicate modality encoders in every pipeline stage, as shown in Figure 1b. This approach achieves computational balance across pipeline stages as long as the LLM pipeline stages are balanced by the existing LLM pipeline parallelism. However, redundant modality processing eventually costs in overall iteration time and memory usage. Figure 2a illustrates that repetitively executing modality encoders in every pipeline stags can drastically slow down the training process.

Encoders-colocated pipeline parallelism. Megatron-LM recently released its alpha support for pipeline-parallel VLM training [39]. However, its implementation is also an extension to the existing *chain-like* pipeline parallel execution of LLMs. Figure 2b represents the execution flow of encoders-colocated pipeline parallelism. To balance execution between



(a) ***Encoders-replicated pipeline parallelism. Only 4 microbatches are shown; executing 8 microbatches as Figure 2b and 2c takes 1.57x longer.***

(b) ***Encoders-colocated pipeline parallelism. Encoders are partitioned into the same number of stages and colocated.***

(c) ***Ideal pipeline parallelism without encoder dependency and partitioning considering MLLM characteristics.***

Figure 2: ***1F1B pipeline execution of the two existing multimodality-unaware pipeline parallelism approaches compared with multimodality-aware pipeline parallelism.***

pipeline stages, encoders can be partitioned into the same number of stages and colocated. Within each colocated stage, the encoders are executed sequentially; however, it provides an illusion of having a single module by fusing them and allows the chain-like execution schedule with multiple encoders, as illustrated in Figure 1c. Having heterogeneous modality encoders in different pipeline stages may introduce imbalanced pipeline stage partitioning. Therefore, users need to carefully partition the model to avoid low utilization due to imbalance.

2.3 Challenges and Opportunities

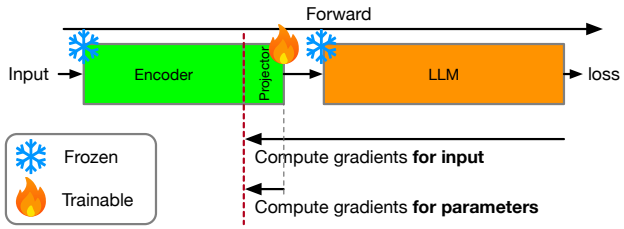
Among the four parallelism dimensions, two – pipeline and context parallelism – require adjustment for distributed MLLM training; the former due to model heterogeneity and the latter for data heterogeneity.

2.3.1 Challenges in MLLM Pipeline Parallelism

Current efforts for distributed MLLM training focus primarily on distributing heterogeneous MLLM layers in pipeline parallelism to address model heterogeneity. However, two key challenges still remain.

C1: False encoder dependency. In MLLM training, modality encoders process different types of data and thus have no dependency on each other. However, existing LLM pipeline parallelism forces a chain-like execution schedule, where each pipeline stage is executed sequentially. It creates a *false dependency* between modality encoders, which is not necessary.

C2: Overlooked frozen status. Unlike LLMs, where all parameters are trainable, MLLM training typically starts with pretrained modality encoders and an LLM, which may be



(a) ***Visualization of computation flow that causes backward time imbalance. The frozen encoder skips the entire backward pass, while others do not.***

Encoder & LLM Frozen Status	Time (ms)	Encoder	Projector	LLM
Frozen	Fwd	67.89	3.74	397.11
	Bwd	0.01	9.01	530.67
Not Frozen	Fwd	67.94	3.75	400.87
	Bwd	205.09	9.47	1184.65

(b) ***An execution time breakdown example of a VLM – CLIP (encoder) + Mistral-7b (LLM) – with different frozen status of the encoder and the LLM. The projector is unfrozen and trainable in both cases. Run on a single NVIDIA A40 with batch size 2. Activation checkpointing is used [23].***

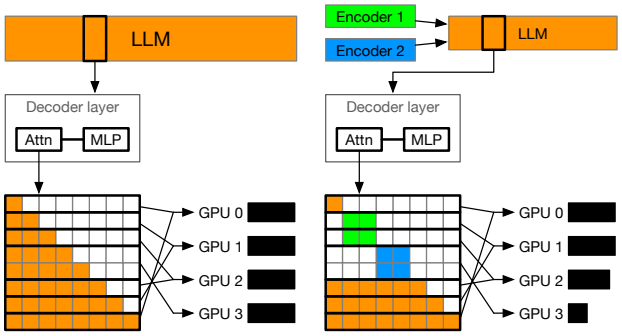
Figure 3: ***An example of various backward time due to frozen satus and module location. Even the encoder and LLM are both frozen, the LLM still needs to compute gradients for the input to backpropagate them to the projector.***

frozen to train projectors between them to align their embedding spaces. The amount of backward computation varies depending on the frozen status and the location of frozen components. However, existing LLM pipeline parallelism approaches do not consider the frozen status of components in partitioning. Figure 3 illustrates the difference in backward computation between frozen and trainable components. The location of frozen components also affects backward computation: although both the encoder and the LLM are frozen, the LLM still needs to compute gradients for the input – but not for the parameters – to backpropagate them, while the encoder (non-projector part) does nothing during the backward pass. This invalidates the long-held rule of thumb that *backward passes take roughly twice as long as forward passes* [37] and causes execution time imbalance between pipeline stages.

2.3.2 Challenges in MLLM Context Parallelism

Existing context parallelism solutions are limited to causal dependencies observed in LLMs [13, 32], and they cannot address MLLM-induced data heterogeneity. We observe two challenges in context parallelism unique to MLLMs.

C3: Lack of efficient multimodal attention representation. The causal relationship between language tokens is easy to represent as a 1D binary vector, where each element indicates whether the corresponding token needs to be attended to or not. Having such an efficient attention representation is crucial, as the full attention mask requires $O(T^2)$ memory, where T is the number of tokens. A full causal attention mask is temporarily constructed and freed after the attention computation



(a) ***LLM context parallelism with zigzag token distribution.*** **(b)** ***MLLM context parallelism with the same token distribution as LLM.***

Figure 4: ***Context parallelism load balancing for LLM text tokens with causal relationship. If the same distribution is applied to MLLM, it is likely to cause imbalanced workloads across GPUs.***

is done. However, the attention relationship between multimodal tokens is more complicated, as shown in Figure 4b, and cannot be represented as a simple binary vector.

C4: Lack of balanced multimodal token distribution. Some works have proposed distributing tokens in a balanced way for LLM context parallelism [2, 4, 13]. Specifically, the input sequence is partitioned into $2 \times \text{cp\_size}$, where cp_size is the number of ranks in context parallelism dimension, and i-th rank gets i-th and $(2 \times \text{cp\_size} - 1 - i)$-th chunks as in Figure 4a, which is also called *zigzag* distribution. This guarantees perfect load balancing in causal attention, but it is not suitable for MLLM context parallelism due to the intricate interactions between multimodal tokens. If the zigzag distribution is applied to MLLM attention representation, it is likely to cause imbalanced workloads across GPUs, as shown in Figure 4b.

3 Cornstarch

Cornstarch is a distributed MLLM training framework. In this section, we first introduce its design principles (§3.1), followed by its programming model (§3.2).

3.1 Design Principles

Cornstarch is designed to exploit the modularity of MLLMs and parallelize them in a multimodality-aware fashion. Figure 5 shows the design overview of Cornstarch.

Multimodality-aware model construction. Instead of thinking an MLLM as *monolithic*, where its internal structure is opaque to the system, Cornstarch maintains the modularity information for easy MLLM construction, management, and parallelization. The modularity is not limited to modality modules (e.g., encoders, projectors, and an LLM), but it also includes multimodal-specific functionalities that are not present in any of the constituent unimodal models. For example, various ways of interactions between multimodal data happening between projectors and an LLM can be modularized. Some models adopt cross-attention, some use simple concatenation,

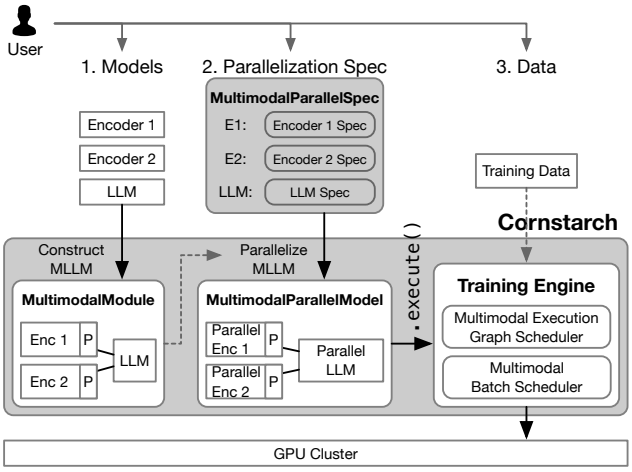


Figure 5: ***Cornstarch architectural overview***

and others embed modality tokens into the middle of text tokens. Instead of copying the same code for each multimodal model, Cornstarch provides a general interface to modularize them.

Hierarchical parallelization. Cornstarch adopts a *hierarchical parallelization approach* for MLLM parallelization, following the classic divide and conquer principle. Modality modules can be parallelized independently with its own parallelization specification. Applying a parallelization specification to each modality allows reusing existing unimodal parallelization techniques, and it reduces burden of developing new parallelization specifications for MLLMs. Cornstarch ensures the output correctness by internally maintaining an execution graph of the MLLM and updates it properly when parallelizing the model. The graph construction does not add any false dependencies if there is no data flow between modules.

Multimodality-aware parallelization. Cornstarch's parallelization techniques are *aware of MLLM characteristics* to address both model heterogeneity and data heterogeneity. Modality encoders are neither replicated nor executed sequentially, but they are parallelized independently (§4.1). Pipeline stage partitioning is optimized by considering the frozen status of the modules that reduces pipeline bubbles (§4.2). LLM pipeline stages after multimodality tokens are embedded into the text embedding space adopt a more balanced multimodality-aware token distribution for context parallelism (§4.3).

3.2 Cornstarch Programming Model

We show an example of using Cornstarch APIs available to users for MLLM training in Listing 1. Bold classes and methods are from Cornstarch. While creating or parallelizing an MLLM, Cornstarch internally maintains modularity information and generates an execution graph based on this information for training. We describe the key concepts below.

Listing 1 An example code using Cornstarch APIs for distributed MLLM training.

```python
# Cornstarch modules
from cornstarch import (
    ModalityModule, MultimodalModule,
    ParallelSpec, MultimodalParallelSpec,
    MultimodalParallelModule,
)

# Load unimodal models
vis = SiglipVisionModel.from_pretrained("...")
audio = WhisperEncoder.from_pretrained("...")
# ... more encoders
llm = LlamaForCausalLM.from_pretrained("...")

# Create an MLLM from unimodal models
mllm = MultimodalModule(
    encoders = {
        "vision": ModalityModule(vis, proj="mlp"),
        "audio": ModalityModule(audio, ...),
        # ... more encoders
    }
    language_model = ModalityModule(llm),
)

# Set frozen status as needed
mllm.vision_encoder.module.train(mode=False)
mllm.vision_encoder.projector.train(mode=True)
...

# Parallelize the MLLM
torch.distributed.init_process_group(...)
vis_spec = ParallelSpec(
    tp_size=2, cp_size=1, pp_size=2,
)
# ... more specs for each modality module
mm_spec = MultimodalParallelSpec(
    encoder_specs={"vision": vis_spec, ...},
    language_model_spec=llm_spec,
    num_microbatches=...,
    microbatch_size=...,
)
parallel_mllm: MultimodalParallelModule =
    mm_spec.apply(mllm)

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

Modules. A *module* (or a `ModalityModule`) represents a set of executable components (e.g., an LLM or a vision encoder with a projector) that are executed sequentially. Users can create a tree hierarchy of several modules by forming a `MultimodalModule` to represent an MLLM. This tree structure of modules gives users full control over the MLLM formulation.

Parallelism specification. A *parallelism specification* (or a `ParallelSpec`) specifies how a single `ModalityModule` should be parallelized, e.g., tensor parallel degree, number of pipeline stages, etc. A set of `ParallelSpecs` can be combined into a `MultimodalParallelSpec` to parallelize an MLLM. This follows the exact same hierarchical structure with modules under a `MultimodalModule` so that each parallelism specification can be applied to the corresponding module. When `MultimodalParallelSpec.apply(MultimodalModule)` is called, all `ParallelSpecs` are applied to the corresponding `ModalityModules`, recursively changing the execution graph of each module to have multiple nodes, so that they can be distributed. After all specifications are applied, a parallelized MLLM as an instance of `MultimodalParallelModule` is returned. `MultimodalParallelModule.execute()` can be used to execute the parallel execution graph in a distributed cluster.

Execution graph. A `MultimodalModule` maintains the *execution graph* of the modules, specifying the order in which the modules should be executed. Each module is represented as a single node in the graph, with directed edges between nodes representing the data flow between modules. When parallelized, each `ModalityModule` in the graph is split into multiple nodes, and the edges are updated to reflect the data flow between the parallelized nodes.

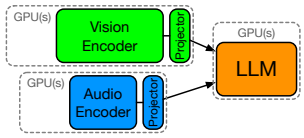
4 Multimodality-Aware Parallelization

In this section, we introduce MLLM-aware parallelization to address the challenges mentioned in Section 2.3. We first introduce a new dimension for parallelism called *modality parallelism* to solve the false dependency issue in encoder execution (§4.1). We then introduce how to enhance existing parallelism dimensions – pipeline parallelism (§4.2) and context parallelism (§4.3) – to address model heterogeneity and data heterogeneity, respectively.

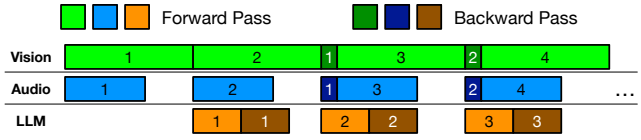
4.1 Modality Parallelism

An MLLM can have multiple different modality encoders for understanding various types of information, forming a directed acyclic graph (DAG), as in Figure 6a. When the modality encoders are assigned to different (sets of) GPUs, they can be executed in parallel without any dependency between each other. We introduce *modality parallelism* to parallelize the execution of modality encoders.

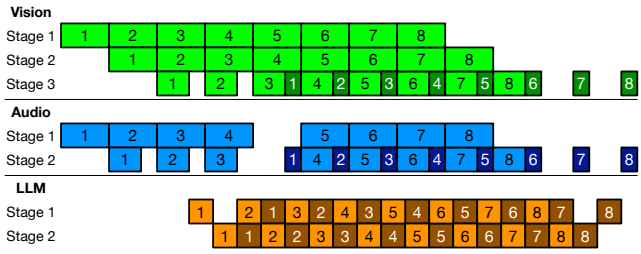
Modality parallelism analyzes the execution DAG and assigns modules that do not have any dependency between each other (e.g., sets of encoder + projector in Figure 6a) to different GPUs so that they can be executed in parallel. The node that has incoming edges from multiple nodes (e.g., the LLM node in Figure 6a) is also assigned to a dedicated GPU to remove dependency as a form of an incoming edge in the middle of the execution in a single GPU. Therefore, the MLLM in Figure 6a can be parallelized into 3 GPUs, and its execution timeline is shown in Figure 6b. The forward pass of the LLM has dependency on both vision and audio encoders, thus it starts only after both encoders finish their forward pass computation. Likewise, backward passes of the modality en-



(a) ***A parallelized execution DAG of an MLLM.***



(b) ***A modality parallel execution of the MLLM in (a). The vision and audio encoders can be executed in parallel, and encoders and the LLM can be pipelined.***



(c) ***Modalities can be parallelized with their own parallelization spec and pipelined across modalities.***

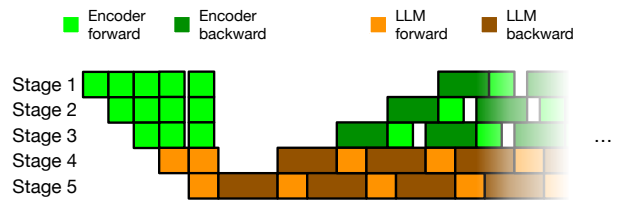
Figure 6: ***An example of modality parallelism, with two modality encoders (vision and audio) and an LLM. Vision and audio encoders are executed in parallel without dependency to each other, and their outputs are embedded into the text embedding space and fed into the LLM. Numbers in the boxes are microbatch indices.***

coders can start only after the LLM finishes its backward pass computation, but they can be executed in parallel.

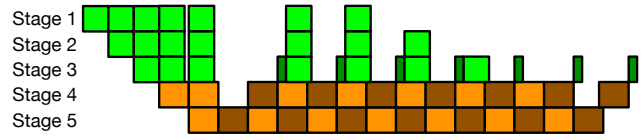
Modality modules can be computationally heterogeneous, which could lead to bubbles in parallel execution. Modality parallelism allows applying different parallelization specs to modality modules to balance cross-modality pipeline execution. Figure 6c shows that different number of pipeline parallelism degree is applied to the modality encoders and the LLM. Even though the number of pipeline stages per modality is different, cross-modality 1F1B pipeline execution still works well.

4.2 Frozen Status-Aware Pipeline Parallelism

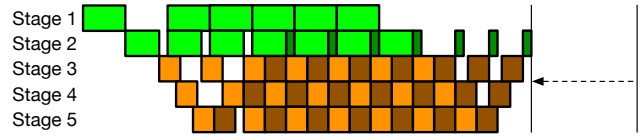
Traditionally, in pipeline parallelism, it is assumed that all parameters are trainable, and backward pass takes twice as long as forward pass [37]. Neither of these assumptions hold for MLLM training, where the amount of backward pass computation can vary depending on the frozen status of constituent models and their locations in the overall MLLM architecture. Figure 7b illustrates the pipeline parallel execution for an MLLM partitioned using existing solutions as Figure 7a, where the encoder and the LLM are both frozen and the projector between them is trainable. Although the time taken by the forward pass across pipeline stages is well balanced, it becomes imbalanced when the backward pass starts, leading to more pipeline bubbles in the middle.



(a) ***1F1B pipeline schedule assuming all parameters are trainable.***



(b) ***Actual pipeline execution of Figure 7a with a frozen encoder and a frozen LLM.***



(c) ***Optimized 1F1B pipeline scheduling with frozen status-aware partitioning.***

Figure 7: ***1F1B pipeline parallelism in an MLLM with an encoder and an LLM. The assumption of all parameters to be trainable leads to imbalanced execution time when it comes to play with frozen models.***

Note that simply considering the frozen status and completely skipping backward pass computation is not correct either. Even though they are frozen, some modules might still have to compute gradients and backpropagate them depending on their location. For example, in Figure 7b, the LLM still has backward pass computation as it needs to backpropagate gradients to the projector. The encoder, however, only has very small backward pass computation for the projector in the last stage of the encoder (i.e., stage 3 in Figure 7b); the rest of the backward pass computation can be skipped since there is no trainable parameter before it. The time of backward pass for the LLM stages is also smaller than expected in Figure 7a. This is because the LLM computes gradients only for inputs – not for its parameters – as it is frozen.

We observe that pipeline stage partitioning needs to be based on the actual execution time of *one forward pass + one backward pass* in consideration of the frozen status and the module location. Given that the encoder pipeline stages have very few trainable parameters, one forward + one backward time is more balanced when they have more computation in forward passes. Figure 7c shows such pipeline stage partitioning. It has one fewer pipeline stage for the encoder thus more computation in stages 1 and 2, while the LLM has one more stage, having less computation in each stage. Although forward passes across pipeline stages are not balanced, the sum of one forward and one backward across pipeline stages is

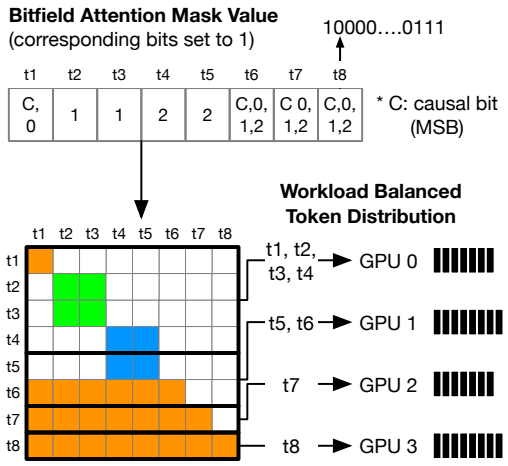


Figure 8: ***Cornstarch multimodality-aware context parallelism overview. It consists of a memory efficient attention mask representation (Bitfield Attention Mask) and a balanced token distribution algorithm.***

more balanced, resulting in a faster iteration time with fewer pipeline bubbles.

We provide a simple equation to estimate the backward pass time of a module, which can be used in manual partitioning or automatic partitioning algorithms:

$$T_{\text{backward}} = \begin{cases} 0 & \text{if frozen and no trainable module prior to this module} \\ 1 \times T_{\text{forward}} & \text{if frozen and trainable module prior to this module} \\ 2 \times T_{\text{forward}} & \text{if not frozen} \end{cases}$$

where T_{forward} and T_{backward} means the execution time of a single forward and backward pass, respectively.

Note that MLLM pipeline stage partitioning should also consider whether gradient checkpointing (i.e. activation recomputation [23]) is enabled. When it is enabled, the forward pass needs to be executed again to compute activation, only if the module has some gradients to compute.

4.3 Multimodality-Aware Context Parallelism

Context parallelism distributes a single sequence by partitioning it into multiple sets of tokens. All existing studies that balance the amount of computation per token in context parallelism assume the attention is *causal*. However, in multimodal context parallelism, many non-causal attention masks can be generated [4, 24]. We first introduce *Bitfield Attention Mask (BAM)* as an efficient way of representing arbitrary attention mask (§4.3.1). Then, we introduce a new token distribution algorithm for multimodal context parallelism for balanced parallel execution (§4.3.2).

4.3.1 Bitfield Attention Mask

A full attention mask for one batch and one head is a matrix of shape $[T, T]$, where T is the number of tokens. If T is 1 million, one full $[T, T]$ attention mask already needs 1TB of memory; thus it is typically compressed to and stored as a 1D boolean vector [11]. However, with a 1D boolean vector, complex attention relationships between multiple modality inputs cannot be represented. To address this, we introduce *Bitfield Attention Mask (BAM)*, a memory-efficient attention mask representation. BAM can represent a full attention mask in a 1D vector of 64-bit integers, where each bit represents whether the token needs to attend *specific modality output*. BAM supports up to ∼60 different modality encoders with a few bits reserved for control bits, which is enough for most multimodal models.

We assign bits from the least significant bit (LSB) to the most significant bit (MSB) to the modality encoders and the LLM. For example, if there are encoders A and B and an LLM as in Figure 8. 0th LSB is assigned to text modality, 1st LSB is assigned to encoder A modality, and 2nd LSB is assigned to encoder B modality. `t2 t3` are tokens from encoder A, thus they only have 1st LSB set to 1. Likewise, `t4 t5` are tokens from encoder B with 2nd LSB set to 1. `t6 t8` are text tokens that need to attend all of its previous tokens including modality tokens, thus they have all corresponding LSBs set to 1.

When attention mask is needed, a 4D matrix of $[B, H, T, T]$ (or $[B, H, T/\text{sp\_size}, T]$ if context parallelism with sp_size number of ranks is enabled) is temporarily constructed and used to compute attention as the original attention processing; where B is batch size, H is the number of heads, and T is the number of tokens. Blockwise attention mask instantiation can also be used instead of instantiating a full mask to reduce peak memory usage [11, 14, 31].

BAM is especially useful when used with pipeline parallelism. The interaction between multimodal tokens should be maintained across all LLM pipeline stages to compute attention correctly. With BAM, we can easily transfer the attention mask representation across pipeline stages with minimal networking overheads.

4.3.2 Workload-Balanced Token Distribution

Instead of simply distributing the same number of tokens to each GPU, we need to consider *the amount of computation* per token for distribution. Balancing distributed the amount of computation is clearly equivalent to makespan minimization scheduling or job-shop scheduling problem which has proven to be NP-hard [50]. We first introduce the problem formulation as an integer linear programming and then propose a heuristic algorithm to solve it efficiently.

Integer linear programming (ILP). We formulate the problem as an integer linear programming (ILP) problem as follows:

$$
\begin{aligned}
\underset{x,C}{\text{minimize}} \quad & C \\
\text{subject to} \quad & \sum_{g=1}^{G} x_{i,g} = 1, \qquad i = 1,\ldots,T, \\
& \sum_{i=1}^{T} W_i \cdot x_{i,g} \le C, \quad g = 1,\ldots,G, \\
& x_{i,g} \in \{0,1\}
\end{aligned} \tag{1}
$$

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 the attention mask. The linear programming balances workload by minimizing the completion time C, which is the maximum workload assigned to any GPU.

Greedy makespan minimization. For a long sequence, the ILP problem is intractable in real-time during training, thus we adopt the greedy Longest-Processing-Time-First (LPT) algorithm (see Appendix A) to assign tokens to GPUs in a context parallelism group for fast and efficient distribution [12]. 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 token T, and the number of GPU G. Since $t_{\max}$ is negligible with large T, the time is getting closer to the perfectly balanced distribution as T increases.

It requires $O(TG\log T)$ time complexity, where $T\log T$ is consumed by sorting the tokens in descending order of their workloads. As token assignment is done in block granularity – a set of contiguous tokens is assigned to the same block – for efficient parallelization in accelerators, the overhead of distributing tokens is negligible; distributing 1 million tokens with 128 block size can be done within 1 ms for example.

5 Implementation

Cornstarch is implemented in around 24,500 new SLOC on top of PyTorch 2.6.0 nightly (dev-20241121+cu124) [42] and Colossal-AI 0.4.4 [25]. Cornstarch's model partitioning, scheduling, execution, communication, and checkpointing are implemented upon Colossal-AI interface.

5.1 Callback Interface

Cornstarch provides a general interface to add user-defined callbacks before and after each module is executed to process interactions between modules. The callbacks are useful when modules are not designed for multimodality and may need additional processing.

For example, LLaVA-Next [33] proposes AnyRes technique that splits an image into multiple image blocks for high resolution image processing, which its underlying CLIP vision encoder [44] does not support. `cb_before_encoder` in Listing 2 is an example that shows how such image preprocessing (not supported by the vision encoder) can be added using a callback. Users can implement or use the one provided by Cornstarch to split images into blocks of Bs x Bs resolution (224, 384 or any resolution that the vision encoder supports) while leaving the vision encoder unmodified.

Listing 2 Callback interface for customized interaction between modules.

```python
# Users implement custom callbacks
def cb_before_encoder(
    inputs: dict      # input to encoder
) -> dict[str, Any]:  # preprocessed input for encoder
    # split images to blocks and flatten them as batch
    # shape: [B, 3, W, H] -> [B, N, 3, Bs, Bs]
    # where N is the number of blocks
    inputs["pixel_values"] = ...
    return inputs

def cb_after_encoder(
    inputs: dict,   # input to encoder
    output: tuple,  # output from encoder
) -> tuple:         # preprocessed input for projector
    return output # by default return output as it is

def cb_after_proj(
    inputs: dict,   # input to encoder
    output: tuple,  # output from projector
) -> tuple:
    return output # by default return output as it is

vision_encoder = ModalityModule(
    model=vision,
    projector="mlp",
    preprocess_callback=cb_before_encoder,
    postprocess_module_callback=cb_after_encoder,
    postprocess_projector_callback=cb_after_proj,
)

def cb_before_llm(
    encoder_outputs: dict, # outputs from projectors
    inputs: dict,          # input to llm
) -> dict:
    return ... # input to llm with modality tokens

mllm = MultimodalModule(
    encoders={"vision": vision_encoder},
    language_model=llm,
    preprocess_callback=cb_before_llm,
)

# Call order
# cb_before_enc -> vision_encoder -> cb_after_enc ->
# projector -> cb_after_proj -> cb_before_llm -> llm
```

`preprocess_callback` in `MultimodalModule` is the one where projected encoder tokens are merged and embedded into the LLM embedding space. While a naive approach is to simply prepend the visual tokens prior to text tokens, modern MLLMs use a special token (e.g., `<img>`) in text inputs, which will be replaced by modality tokens after the encoders are executed to provide richer contextual information. As the underlying LLM does not know about the special token and how to handle it, the callback can be used to implement such modality token merging policy without LLM modification.

5.2 Auto Parallelization

Algorithm 1 Loosely-coupled multimodal parallelization

```
Input: E encoder models, L language model
Output: E' parallelized encoders, L' parallelized language model
    ▷ Run any unimodal auto parallelization algorithm
for each number of feasible LLM stages i do
    L_i, t_i = get_parallel_model(L, num_stages=i)   ▷ t_i is time of a single stage
for each encoder e in E do
    e_i = get_parallel_model(e, target_stage_time=t_i)
E_i = set of e_i for each e in E
    ▷ Find optimal combination with minimum iteration time
for i do
    T_i = get_iteration_time(L_i, E_i)
return E_i, L_i with minimum T_i
```

Cornstarch focuses on providing modular multimodal training and does not provide its own auto-parallelization scheme. Instead, users can use an existing auto parallelization solution [19, 53, 62] that is optimized for partitioning unimodal models. Cornstarch glues the parallelized unimodal models into a parallel MLLM and executes them following the execution graph.

Simply gluing parallelized models, however, does not provide the optimal throughput. Each module may have different optimal parallelization configurations with different per pipeline-stage execution time, which may lead to imbalanced execution when glued together. Cornstarch introduces *loosely-coupled constraints* which guide its parallelization algorithm to find the optimal parallelization configuration with the specific target for per pipeline-stage execution time. Algorithm 1 shows how to parallelize an MLLM using unimodal parallelization with loosely-coupled constraints. We first generate a set of parallelization options of an LLM with different number of pipeline stages (t_i represents the time of a single stage forward pass + backward pass of i-th parallelization option). This information is used in finding optimal parallel configurations of encoders while cross-modality per-state execution time is balanced. When such constraints are satisfied, a multimodal parallel model is still balanced even if each modality is partitioned individually.

5.3 Implementation of Context Parallelism

There are various ways of implementing context parallelism: all-to-all [18], P2P ring attention [32], and all-gather ring attention [2], etc. All-to-all has no imbalance problem as it converts parallelization dimension from the token space to the head space and each rank computes exactly the same shape of attention. However, it has high communication overhead that cannot be overlapped with computation, and it has limited scalability as its maximum parallelization degree is limited to the number of heads that conflicts with tensor parallelism [13].

Cornstarch implements the state-of-the-art context parallelism implementation, which is used in Llama3 training [2], using PyTorch FlexAttention [14]. This implementation gathers all keys and values of all tokens and compute row-wise attention for local tokens. Overlapping communication and computation is done in the head space; while GPUs compute attention for one or a few heads, it transfers keys and values for the next head(s). This simplifies Algorithm 1 while remaining efficient and scalable. For P2P ring attention, however, computation per GPU and per round must be tracked, and the amount of workloads must be updated by traversing all tokens for every round, which makes algorithm's time complexity $O((TG)^2 \log T)$, where T is the number of tokens and G is the number of GPUs, which may not be practical with long sequences.

Random distribution. In case users have to use another non all-gather based context parallelism mechanism, Cornstarch provides a random token distribution algorithm that randomly assigns tokens to GPUs. We observe that when T is large enough ($T >> G^2$), then token distribution variance from simple random distribution becomes close to that from the greedy algorithm (Algorithm 2 in the appendix) but much faster and can be done instantly. The distribution variance bound has theoretically proven based on the Chernoff bound in probability theory [8].

6 Evaluation

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

- Cornstarch provides up to 1.57× higher end-to-end training throughput of MLLM training and shows that its scalability is even better with larger encoders, which opens the door to training larger MLLMs (§6.2).
- Modality parallelism provides flexibility in MLLM parallelization without sacrificing throughput, allowing users to choose and train a model from more than 10,000 different combinations of MLLMs (§6.3).
- Multimodality-aware pipeline parallelism provides up to 1.53× faster iteration time in MLLMs (§6.4).
- Multimodality-aware context parallelism provides up to 1.22× faster attention execution with more balanced token distribution (§6.5).

6.1 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 baselines as follows:

1. *Encoders-colocated 4D parallelism*: it parallelizes an MLLM based on forward pass time assuming backward

Table 1: ***Modality (LLM, vision, and audio) configurations.***

Model Arch.	Model Size	# Layers	Hidden Size	# Params
Llama 3.1 [2] (LLM)	Small	16	2048	1.2b
	Medium	32	4096	8b
	Large	64	5120	32b
EVA-CLIP [49] (Vision)	Small	40	1408	1b
	Medium	32	4096	8b
	Large	48	5120	18b
Whisper [45] (Audio)	Small	32	1920	1.4b
	Medium	40	3840	7b
	Large	48	5120	15b

time takes as twice as forward, and pipelined stages of modality encoders are colocated to make the execution flow chain-like.

2. *Encoders-replicated 4D parallelism*: it first parallelizes an LLM to pipeline stages, and replicates all modality encoders into every LLM pipeline stages.

Both use zigzag token distribution in context parallelism. Both are implemented on Cornstarch as well.

Training dataset. Since there is no open-source dataset with long text, image, and audio data altogether, we use a synthetic dataset for evaluation. We use 1k text tokens, an 1280x720 image, and a 30-second audio clip for each sample. Image tokens and audio tokens are injected into the middle of text tokens when they are projected to the LLM embedding space, having 1.5k∼4k tokens in total. For every experiment, we use 24 microbatches, each of which has a single sample.

Model configurations. We pick one model architecture per modality and create various sizes of models as listed in Table 1. Then we create MLLMs by combining them: vision-language models (VLMs), audio-language models (ALMs), and vision-audio-language models (VALMs). We denote the model name with suffix letters to represent the sizes of its modality encoders (e.g., VALM-SL means a VALM with a small vision encoder and a large audio encoder). We assume encoders and an LLM are frozen and only projectors are trainable. We use a single linear layer as a projector [34].

We profile each module and manually craft parallelization policies for Cornstarch and the baselines. Each modality module is parallelized into a maximum of 6 pipeline stages, utilizing up to 24 GPUs in total. Because each configuration may have different number of GPUs, we normalize throughput by the number of GPUs used.

6.2 End-to-End Performance

6.2.1 MLLMs with a Single Encoder

Currently many MLLMs have a single encoder and an LLM that process X+text data and return text data. We evaluate the end-to-end performance of Cornstarch with two examples

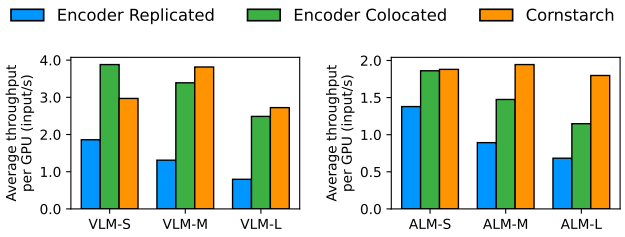


Figure 9: ***End-to-end performance comparison of vision-language models (VLMs) and audio-language models (ALMs) with a medium-size LLM between Cornstarch and baselines.***

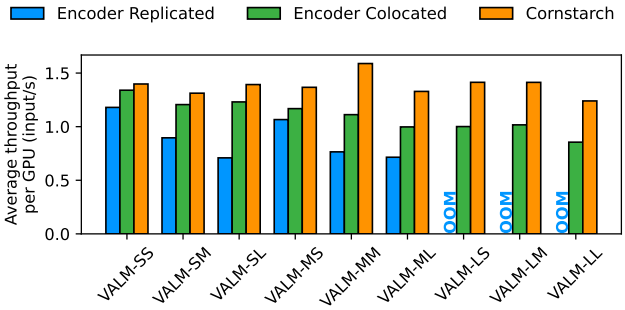


Figure 10: ***End-to-end performance comparison of vision-audio-language models (VALMs) with a medium-size LLM between Cornstarch and baselines.***

of such MLLMs: VLMs or ALMs. Figure 9 shows the end-to-end performance comparison of VLMs and ALMs with a medium-size LLM. See Appendix B for results with other model sizes.

In general, Cornstarch outperforms the baselines by up to 1.57×, with one exception: `VLM-S`. According to our data, partitioning is more balanced in Cornstarch. Encoders-colocated parallelism shows 50ms ∼ 131ms range of per-stage forward + backward time, while Cornstarch has 51ms ∼ 88ms. Cornstarch assigns more pipeline stages to the LLM to distribute the LLM workload, and it leads to more pipeline bubbles due to higher number of pipeline stages. As a result, per-GPU throughput is lower, though iteration time is ∼ 13% faster. However, this problem is mitigated for larger encoders.

6.2.2 MLLMs with Multiple Encoders

We now evaluate Cornstarch with MLLMs with two encoders – vision and audio – as a representative of N-modality MLLMs. Figure 10 shows the end-to-end performance comparison of VALMs with a medium-size LLM. Results with other LLM model sizes are in Appendix B. As larger encoders are added to the model, when they are partitioned based on the assumption of all parameters being trainable, the imbalance of pipeline stages increases. For `VALM-MM` as an example, the encoders-colocated parallelism assigns 3 and 4 stages to the LLM and the set of encoders, respectively, to balance their forward execution time in range of 79ms ∼ 99ms. Cornstarch, in contrast, assigns 4, 1, 1 stages to the LLM, the vision en-

Table 2: ***Comparison of throughput between encoders-colocated and modality parallelism with a medium-size LLM. Modality parallelism provides flexibility in diverse MLLM construction without sacrificing throughput.***

∗ C: a module with colocated encoders, V: vision encoder, and A: audio encoder.

Model	Encoder Colocated		Tput /GPU (input/s)	Modality Parallel			Tput /GPU (input/s)
	LLM	C		LLM	V	A	
VALM-SS	6	1	3.09	6	1	1	2.68
VALM-SM	6	2	2.64	6	1	1	2.73
VALM-SL	6	2	2.57	6	1	2	2.25
VALM-MS	6	2	2.57	6	2	1	2.29
VALM-MM	6	3	2.21	6	1	1	2.22
VALM-ML	6	4	1.91	6	2	2	2.05
VALM-LS	6	4	1.95	6	3	1	1.92
VALM-LM	6	4	1.88	6	3	1	1.97
VALM-LL	6	5	1.74	6	3	2	1.82

coder, and the audio encoder, respectively, awaring the frozen status of the modules. As a result, the range of forward + backward time is reduced from 79ms $\sim$ 230ms to 124 $\sim$ 174ms, reducing imbalance and increasing throughput by 1.44×.

6.3 Efficiency of Modality Parallelism

Modality parallelism parallelizes a general execution DAG of MLLMs. It is not limited to the specific model structure; Cornstarch can create and parallelize any MLLM with any combination of encoders and an LLM. Cornstarch supports various model families and model sizes so that users **can train more than 10,000 different combinations of MLLMs**, unimodal models of which come from 10 LLM families (Gemma [51], Gemma2 [51], GPT [40], InternLM2 [5], Llama [2], Mistral [20], Mixtral [21], OPT [60], Phi-3 [36], and Qwen2LM [57]), 7 vision encoders (CLIP [44], Dinov2 [41], EvaCLIP [49], InternVIT [7], Pixtral [1], Qwen2Vision [54], and Siglip [58]), and 2 audio encoders (Whisper [45] and Qwen2Audio [9]). All unimodal models in the families above that are available in the HuggingFace hub can be used in creating an MLLM [17].

To demonstrate that modality parallelism can provide flexibility without sacrificing throughput, we compare it with encoders-colocated parallelism by fixing the number of LLM pipeline stages and changing the size of encoders and their partitioning in Table 2. Appendix C includes results with other LLM model sizes. `VALM-LS` is an example of the flexibility of modality parallelism. The encoders-colocated parallelism assigns 4 pipeline stages for the colocated encoders; all encoders in the colocated module must be partitioned with the same number of stages. However, modality parallelism assigns 3 and 1 stages to the vision and audio encoders, respectively, providing more flexibility in partitioning modules. Additionally, when a new modality encoder is added, the entire pipeline has to be reconfigured in encoders-colocated parallelism, while Cornstarch can simply attach new independent pipeline stages for the new encoder.

Table 3: ***Pipeline parallel execution throughput with and without frozen status-awareness. Frozen-unaware partitioning balances forward time between encoders and an LLM, while frozen-aware partitioning balances forward + backward time.***

Model	Frozen Aware	Per-Stage Fwd (ms)		Per-Stage Bwd (ms)		Tput/GPU (input/s)
		Encoder	LLM	Encoder	LLM	
VLM-S	✓	49.06	27.46	0.12	43.03	6.47
	✗	26.49	28.13	0.16	51.32	5.30
VLM-M	✓	75.04	28.15	0.27	51.30	5.15
	✗	45.90	45.26	0.16	84.89	3.73
ALM-S	✓	57.25	81.73	0.38	157.12	1.94
	✗	57.79	81.94	0.38	157.14	1.94
ALM-M	✓	176.80	82.19	0.64	157.51	1.96
	✗	93.39	98.25	0.34	189.11	1.70
VLM-L	✓	101.02	34.67	0.20	63.77	3.17
	✗	70.42	67.03	0.15	127.34	2.06
ALM-L	✓	308.16	100.20	0.73	188.50	1.98
	✗	145.97	121.21	0.33	235.23	1.69

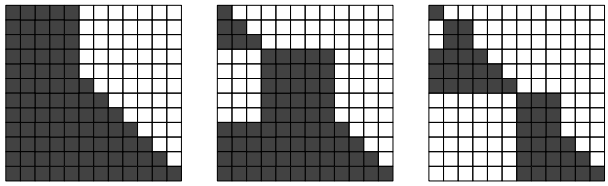
6.4 Impact of Frozen Status-Aware Pipeline Parallelism

We partition VLMs and ALMs with a medium-size LLM using frozen status-aware and -unaware partitioning, and compare their throughput. The number of pipeline stages are the same in both cases. Table 3 shows the result. See Appendix D for other LLM sizes. Without frozen status-awareness, partitioning is done based on the assumption of all parameters are trainable, which tries to minimize variance of forward time across stages, expecting backward time is proportional to forward time. For example, in `VLM-L`, the frozen status-unaware partitioning partitions the encoder and the LLM to have similar forward time (70.42ms and 67.03ms, respectively).

In reality, however, the encoders and the LLM are frozen, and gradient computation for their parameters is skipped. The LLM still needs to backpropagate gradients to the trainable projectors, thus its backward time is similar to its forward time, while the encoders do not have to compute gradients at all and their backward time is negligible except the projectors. Having frozen status in mind, `VLM-L` rather has higher forward time in encoder stages (101.02ms) than in LLM stages (34.67ms); however, its forward + backward time is more balanced across the stages (101.02 + 0.20 = 101.22ms, 34.67 + 63.77 = 98.44ms) than the frozen status-unaware partitioning (70.42 + 67.03 = 137.45ms, 67.03 + 127.34 = 194.37ms), thus the overall throughput is increased by 1.53×.

6.5 Impact of Multimodality-Aware Context Parallelism

This section evaluates how Cornstarch token distribution based on LPT algorithm (§4.3.2) and random distribution (§5.3) distributes non-causal attention execution well, compared to the baselines: ring attention [32] and zigzag [13]. We run a single attention layer of a Llama 3.1 70B LLM with 8 ranks, and compare the execution time of attention with various attention masks and 16k, 32k, and 64k context length.



(a) *Encoder outputs prepended (EP).* **(b)** *Encoder outputs embedded (EE).* **(c)** *Multimodal packing (MP).*

Figure 11: ***Various attention masks used in MLLM training.***

Table 4: ***Execution time of a single LLaMA 3.1 70b LLM attention layer in 8 ranks using context parallelism with various attention masks.***

∗ Mask types are illustrated in Figure 11. EP: encoder outputs prepended. EE: encoder outputs embedded. MP: multimodal packing.

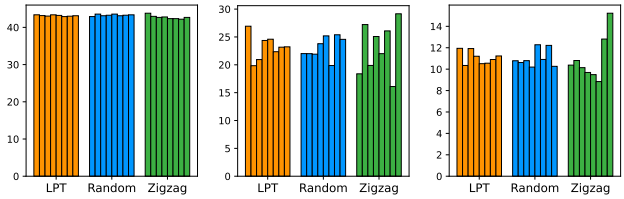
Sequence Length	Mask Type	Attention Time (ms)			
		LPT	Random	Naive Ring	Zigzag
16384	EP	3.92	3.93	4.24	3.96
	EE	4.31	4.32	4.86	4.37
	MP	3.00	2.99	3.12	3.08
32768	EP	10.01	10.07	11.76	10.21
	EE	11.12	11.12	13.43	11.60
	MP	6.09	6.18	6.76	6.57
65536	EP	25.43	25.48	32.24	28.17
	EE	36.99	37.12	46.67	37.36
	MP	14.18	14.15	16.56	15.33

We run 50 times for each mask type and an attention mask is randomly generated for every run. Table 4 reports the average attention execution time.

LPT and random provide faster attention computation time than naive ring and zigzag in most cases. To understand how each algorithm distributes tokens, we sample one measurement with 64k tokens in Figure 12. Prepending encoder outputs (EP) generates simpler attention mask as in Figure 11a thus all algorithms provide balanced token distribution. However, embedding encoder outputs (EE) and multimodal packing (MP), which are expected to be used more for precise interaction [7, 33, 54], generate more complicated attention mask as in Figure 11b and Figure 11c, respectively. Simple zigzag struggles to distribute tokens evenly, while LPT and random provide more balanced distribution.

7 Related Work

Distributed training frameworks. There are numerous frameworks for efficient and scalable distributed LLM training [2, 25, 38, 47]. These frameworks focus on optimizing memory usage, parallelism, and communication to meet the demands of large-scale LLM training. DistMM [15] presents a method that adaptively applies different parallelism strategies to modality modules in terms of model heterogeneity. However, it is limited to contrastive learning which is not directly applicable to MLLM training. For distributed MLLM training, there is no existing general-purpose framework that considers the unique characteristics of MLLMs.



(a) *EP* **(b)** *EE* **(c)** *MP*

Figure 12: ***One sampled result of a Llama 3.1 70B LLM attention layer with 64k tokens. 8 bars with the same color represent the execution time of 8 ranks in the context parallelism. Y-axis is the execution time in ms.***

Fully Sharded Data Parallel (FSDP). FSDP [46, 61] is a commonly used parallelism technique for distributed MLLM training due to its ease of use [5, 34, 54]. Unlike traditional data parallelism, where all GPUs have a full copy of the model, FSDP shards model weights across GPUs and temporarily unshards them during forward and backward passes. As only a portion of the model is unsharded at a time, FSDP reduces memory consumption while mitigating communication overheads my overlapping unsharding the next set of weights with computing on the currently unsharded weights. However, FSDP has limited scalability and communication efficiency challenges, resulting in restrictions on the maximum model size [27]. Therefore, solely relying on FSDP is not sufficient for large-scale MLLM training.

4D parallelism. No single parallelism technique can fully address the challenges of large model training, so a combination of parallelism techniques is typically used. 4D parallelism – data, tensor, pipeline, and context parallelism – has been used to train large LLMs [2, 25, 38]. Each parallelism dimension has been extensively explored to optimize training efficiency and scalability [2, 4, 13, 16, 18, 24, 32, 37, 38, 43]. Combining them together, several works have shown significant improvements in training scalability [2, 22] to more than 10k GPUs. However, as shown throughout this paper, none of the existing efforts consider MLLM-specific challenges.

8 Conclusion

In this paper, we presented Cornstarch, the first multimodality-aware distributed MLLM framework. We observe that simply retrofitting existing distributed training frameworks for MLLMs is not sufficient. Instead, Cornstarch addresses challenges arising from model and data heterogeneity in MLLM training from first principles. We introduce a new parallelism dimension called modality parallelism and improve the efficiency of pipeline parallelism by adding frozen status awareness into consideration. We also propose a new efficient token representation and distribution mechanism for MLLM context parallelism. Cornstarch shows up to 1.57× speedup over the state-of-the-art distributed training frameworks.

References

[1] 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.

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

[3] Rawan AlSaad, Alaa Abd-alrazaq, Sabri Boughorbel, Arfan Ahmed, Max-Antoine Renault, Rafat Damseh, and Javaid Sheikh. Multimodal large language models in health care: Applications, challenges, and future outlook. *J Med Internet Res*, 26:e59505, 2024.

[4] 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.

[5] 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*, abs/2403.17297, 2024.

[6] 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*, 2024.

[7] 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*, 2024.

[8] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

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

[10] Yunfei Chu, Jin Xu, Xiaohuan Zhou, Qian Yang, Shiliang Zhang, Zhijie Yan, Chang Zhou, and Jingren Zhou. Qwen-audio: Advancing universal audio understanding via unified large-scale audio-language models, 2023.

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

[12] Ronald Lewis Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[13] 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. *CoRR*, abs/2406.18485, 2024.

[14] Horace He, Driss Guessous, Yanbo Liang, and Joy Dong. Flexattention: The flexibility of pytorch with the performance of flashattention, 2024.

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

[16] 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*, 2019.

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

[18] 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.

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

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

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

[22] 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*, 2024.

[23] 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*, 2023.

[24] 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*, 2024.

[25] 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*, 2023.

[26] Paul Pu Liang, Yiwei Lyu, Xiang Fan, Jeffrey Tsaw, Yudong Liu, Shentong Mo, Dani Yogatama, Louis-Philippe Morency, and Russ Salakhutdinov. High-modality multimodal transformer: Quantifying modality & interaction heterogeneity for high-modality representation learning. *TMLR*, 2023.

[27] 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 pre-training, 2024.

[28] 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.

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

[30] Fenglin Liu, Tingting Zhu, Xian Wu, Bang Yang, Chenyu You, et al. A medical multimodal large language model for future pandemics. *NPJ Digital Medicine*, 6(1):226, 2023.

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

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

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

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

[35] Bertalan Meskó. The impact of multimodal large language models on health care's future. *J Med Internet Res*, 25:e52865, 2023.

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

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

[38] 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*, 2021.

[39] NVIDIA. Megatron-lm, 2024.

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

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

[42] 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*, 2019.

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

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

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

[46] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *SC*, 2020.

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

[48] Dhruv Shah, Błażej Osiński, brian ichter, and Sergey Levine. Lm-nav: Robotic navigation with large pre-trained models of language, vision, and action. In *ICRA*, 2023.

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

[50] V. Tanaev, W. Gordon, and Y.M. Shafransky. *Scheduling Theory. Single-Stage Systems*. Springer Netherlands, 2012.

[51] Gemma Team. Gemma: Open models based on gemini research and technology, 2024.

[52] 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*, abs/2406.16860, 2024.

[53] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, et al. Unity: Accelerating dnn training through joint optimization of algebraic transformations and parallelization. In *OSDI*, 2022.

[54] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, et al. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution, 2024.

[55] 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*, 2020.

[56] 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*, 2021.

[57] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, et al. Qwen2 technical report. *CoRR*, abs/2407.10671, 2024.

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

[59] 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*, 2024.

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

[61] 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, 2023.

[62] 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*, 2022.

[63] 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*, 2024.

[64] Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *CRL*, 2023.

A Longest Processing Time (LPT) Algorithm

Algorithm 2 Longest Processing Time Algorithm

Input: W: List of token workloads, G: Number of GPUs
Output: Assignment of tokens to GPUs
$L[G] \leftarrow$ minheap() ▷ Workloads per GPU
$A \leftarrow []$ ▷ Assignment of tokens to GPUs
$B \leftarrow$ sort(W) ▷ Sort in descending order of workloads
for each token index t and workload $w \in W$ **do**
    $g \leftarrow$ heap_pop(L) ▷ Get GPU with min workload
    $A[t] \leftarrow g$ ▷ Assign block to GPU
    heap_push(L, g + workload(w)) ▷ Update workload
return A

We adopt the Longest Processing Time (LPT) algorithm to minimize the makespan of the context parallel attention execution. Listing 2 shows the LPT algorithm.

B End-to-End Performance

B.1 Evaluation Setup Details

Table 5: ***Parallelism configurations for end-to-end performance comparison of VLMs and ALMs.***

		PP (LLM, encoder)		TP	CP
		Colocated	Cornstarch		
LLM-S	VLM-S	5, 2	4, 2	2	2
	VLM-M	2, 3	3, 3	2	2
	VLM-L	1, 4	2, 4	2	2
	ALM-S	3, 2	3, 1	2	2
	ALM-M	3, 5	2, 3	2	2
	ALM-L	2, 6	3, 5	2	2
LLM-M	VLM-S	3, 1	5, 1	2	2
	VLM-M	3, 2	3, 1	2	2
	VLM-L	2, 3	3, 2	2	2
	ALM-S	4, 2	5, 1	2	2
	ALM-M	3, 3	4, 2	2	2
	ALM-L	2, 4	4, 2	2	2
LLM-L	VLM-S	5, 1	5, 1	2	2
	VLM-M	4, 1	5, 1	2	2
	VLM-L	3, 2	4, 1	2	2
	ALM-S	5, 1	5, 1	2	2
	ALM-M	5, 1	5, 1	2	2
	ALM-L	5, 2	5, 1	2	2

We manually profile and optimize the 4D paralellism configuration for each model. Table 5 and Table 6 show the evaluation setup details for Section 6.2.1 and Section 6.2.2, respectively. For encoders-replicated parallelism, we always use 6 pipeline stages.

B.2 MLLMs with a Single Encoder

We show the end-to-end performance comparison of VLMs and ALMs with a small-size LLM in Figure 13 and with a large-size LLM in Figure 14.

Table 6: ***Parallelism configurations for end-to-end performance comparison of VALMs.***

∗ L: LLM, C: colocated encoders, V: vision encoder, and A: audio encoder.

		PP		TP	CP
		Colocated (L, C)	Cornstarch (L, V, A)		
LLM-S	VALM-SS	3, 4	3, 1, 1	2	2
	VALM-SM	1, 3	3, 1, 4	2	2
	VALM-SL	1, 4	3, 1, 5	2	2
	VALM-MS	2, 4	3, 3, 1	2	2
	VALM-MM	1, 4	3, 2, 3	2	2
	VALM-ML	1, 5	3, 2, 4	2	2
	VALM-LS	1, 4	3, 5, 1	2	2
	VALM-LM	1, 6	2, 4, 3	2	2
	VALM-LL	5, 2	2, 3, 3	2	2
LLM-M	VALM-SS	5, 2	5, 1, 1	2	2
	VALM-SM	4, 3	5, 1, 1	2	2
	VALM-SL	3, 4	4, 1, 2	2	2
	VALM-MS	4, 4	4, 2, 1	2	2
	VALM-MM	3, 4	4, 1, 1	2	2
	VALM-ML	2, 4	3, 1, 1	2	2
	VALM-LS	2, 4	4, 2, 1	2	2
	VALM-LM	2, 4	4, 2, 2	2	2
	VALM-LL	2, 5	5, 1, 1	2	2
LLM-L	VALM-SS	5, 1	5, 1, 1	2	2
	VALM-SM	5, 2	5, 1, 1	2	2
	VALM-SL	5, 2	5, 1, 1	2	2
	VALM-MS	4, 1	5, 1, 1	2	2
	VALM-MM	4, 2	5, 1, 1	2	2
	VALM-ML	4, 3	5, 1, 1	2	2
	VALM-LS	4, 2	5, 1, 1	2	2
	VALM-LM	4, 3	5, 1, 1	2	2
	VALM-LL	4, 3	5, 1, 1	2	2

B.3 MLLMs with a Multiple Encoder

We show the end-to-end performance comparison of VALMs with a small-size LLM and a large-size LLM in Figure 15.

C Modality Parallelism

D Frozen Status-Aware Pipeline Parallelism

D.1 Evaluation Setup Details

Table 9 shows pipeline parallel configurations for the evaluation in Section 6.4. Encoders-colocated parallelism is frozen statud-unaware, while Cornstarch is. Without ontext parallelism, LLM-L runs out of memory. Thus we increase tensor parallel size to 4 for LLM-L model-based evaluation results.

D.2 Results with Various LLM Size

Table 10 and Table 11 are the evaluation results of frozen status-aware pipeline parallelism with a small-size LLM and a large-size LLM, respectively.

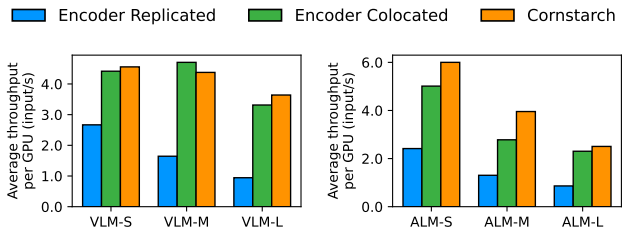


Figure 13: ***End-to-end performance comaparison of VLMs and ALMs with a small-size LLM.***

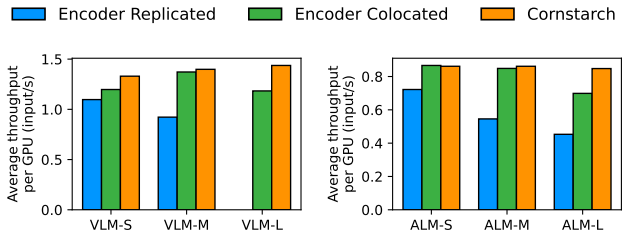
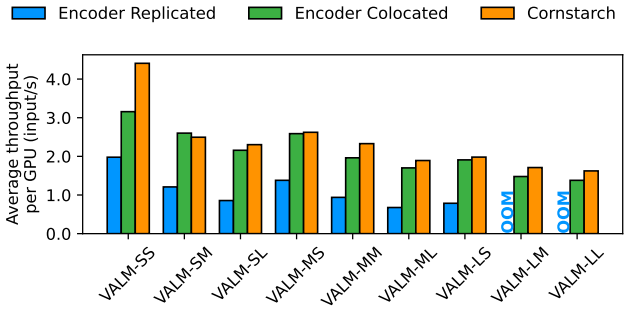
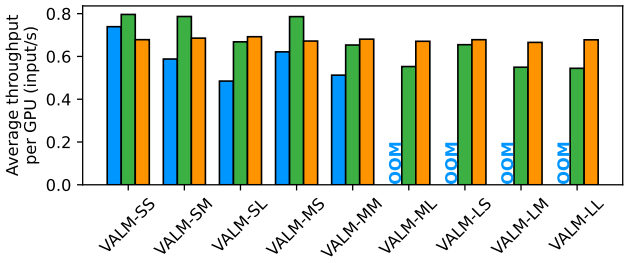


Figure 14: ***End-to-end performance comaparison of VLMs and ALMs with a large-size LLM.***



(a) ***End-to-end performance comparison of VALMs with a small-size LLM.***



(b) ***End-to-end performance comparison of VALMs with a large-size LLM.***

Figure 15: ***End-to-end performance comparison of VALMs with a small-size LLM and a large-size LLM.***

Table 7: ***Comparison of throughput between encoders-colocated and modality parallelism with a small-size LLM.***

∗ C: a module with colocated encoders, V: vision encoder, and A: audio encoder.

Model	Encoder Colocated		Tput /GPU	Modality Parallel			Tput /GPU
	LLM	C	(input/s)	LLM	V	A	(input/s)
VALM-SS	3	4	6.31	3	1	1	8.82
VALM-SM	1	3	5.20	3	1	4	4.99
VALM-SL	1	4	4.31	3	1	5	4.61
VALM-MS	2	4	5.18	3	3	1	5.24
VALM-MM	1	4	3.93	3	2	3	4.66
VALM-ML	1	5	3.40	3	2	4	3.78
VALM-LS	1	4	3.82	3	5	1	3.96
VALM-LM	1	6	2.96	2	4	3	3.42
VALM-LL	1	6	2.76	2	3	3	3.25

Table 8: ***Comparison of throughput between encoders-colocated and modality parallelism with a large-size LLM.***

∗ C: a module with colocated encoders, V: vision encoder, and A: audio encoder.

Model	Encoder Colocated		Tput /GPU	Modality Parallel			Tput /GPU
	LLM	C	(input/s)	LLM	V	A	(input/s)
VALM-SS	5	1	1.57	5	1	1	1.33
VALM-SM	5	2	1.37	5	1	1	1.36
VALM-SL	5	2	1.33	5	1	1	1.37
VALM-MS	4	1	1.57	5	1	1	1.37
VALM-MM	4	2	1.32	5	1	1	1.37
VALM-ML	6	1	1.04	5	1	1	0.93
VALM-LS	4	2	1.31	5	1	1	1.36
VALM-LM	4	3	1.10	5	1	1	1.33
VALM-LL	4	3	1.09	5	1	1	1.36

Table 9: ***Parallelism configuration for pipeline parallelism evaluation***

		PP (LLM, encoder)		TP	CP
		Colocated	Cornstarch		
LLM-S	VLM-S	4, 4	4, 2	2	1
	VLM-M	1, 4	2, 4	2	1
	VLM-L	1, 5	1, 4	2	1
	ALM-S	3, 2	5, 1	2	1
	ALM-M	2, 3	4, 2	2	1
	ALM-L	2, 4	4, 3	2	1
LLM-M	VLM-S	3, 1	6, 1	2	1
	VLM-M	4, 3	5, 2	2	1
	VLM-L	3, 5	5, 4	2	1
	ALM-S	5, 1	6, 1	2	1
	ALM-M	4, 4	6, 1	2	1
	ALM-L	5, 5	4, 2	2	1
LLM-L	VLM-S	3, 5	5, 1	4	1
	VLM-M	5, 1	5, 1	4	1
	VLM-L	4, 2	4, 1	4	1
	ALM-S	5, 1	5, 1	4	1
	ALM-M	3, 1	5, 1	4	1
	ALM-L	4, 2	5, 1	4	1

Table 10: ***Comparison of through with and without frozen status-awareness, with a small-size LLM.***

Model	Frozen Aware	Per-Stage Fwd (ms)		Per-Stage Bwd (ms)		Tput/GPU (input/s)
		Encoder	LLM	Encoder	LLM	
VLM-S	✓	25.57	15.51	0.05	15.09	12.64
	✗	32.80	18.80	0.06	20.09	13.79
VLM-M	✓	75.67	17.31	0.15	20.08	7.85
	✗	45.25	17.61	0.09	20.33	7.63
VLM-L	✓	98.27	26.22	0.11	30.50	4.81
	✗	122.50	47.39	0.13	60.31	6.05
ALM-S	✓	57.52	42.73	0.20	70.21	7.13
	✗	30.14	42.30	0.10	70.22	5.34
ALM-M	✓	92.19	32.54	0.17	52.63	4.85
	✗	59.18	72.90	0.12	105.22	4.09
ALM-L	✓	97.77	33.27	0.12	52.88	4.29
	✗	76.81	62.01	0.09	105.22	3.59

Table 11: ***Comparison of through with and without frozen status-awareness, with a large-size LLM.***

Model	Frozen Aware	Per-Stage Fwd (ms)		Per-Stage Bwd (ms)		Tput/GPU (input/s)
		Encoder	LLM	Encoder	LLM	
VLM-S	✓	52.54	88.58	0.10	171.72	2.39
	✗	85.83	106.26	0.10	171.38	2.10
VLM-M	✓	249.38	88.95	0.49	171.68	2.29
	✗	132.16	113.64	0.26	213.58	1.85
VLM-L	✓	481.44	106.53	0.74	215.06	1.76
	✗	246.22	146.62	0.38	287.14	1.81
ALM-S	✓	76.84	286.16	0.24	596.40	0.72
	✗	78.80	285.07	0.24	594.31	0.73
ALM-M	✓	197.21	287.37	0.45	597.47	0.72
	✗	101.97	359.27	0.24	743.91	0.62
ALM-L	✓	319.31	286.31	0.50	596.50	0.71
	✗	177.29	356.06	0.27	742.80	0.61