DISTMM: Accelerating Distributed Multimodal Model Training

Jun Huang[*]
The Ohio State University

Zhen Zhang
Amazon Web Services

Shuai Zheng
Boson AI, Inc

Feng Qin
The Ohio State University

Yida Wang
Amazon Web Services

Abstract

Multimodal model training takes multiple types of inputs to process with differently structured submodules, and aggregates outcomes from the submodules to learn the relationship among various types of inputs, e.g., correlating text to image for text-to-image generation. The differences of submodule architectures as well as their inputs lead to heterogeneity in terms of computation efficiency. Failing to account for such heterogeneity, existing distributed training systems treat all submodules as a monolithic entity and thus have sub-optimal performance. Moreover, the outcome aggregation phase introduces cross-sample dependencies with contrasting positive and negative sample pairs (i.e., contrastive loss). Such dependencies make the existing pipeline parallelism scheduling algorithms not applicable for multimodal training with contrastive loss.

To address the limitations of existing solutions, we propose DISTMM. For a given multimodal model, DISTMM exploits the heterogeneity among submodules, applying different distributed parallelism strategies for each submodule, e.g., using Tensor Parallelism for a computation-intensive submodule, and Data Parallelism for a submodule with a small number of parameters. DISTMM balances the computation of parallelized submodules to reduce the computing resource idle time of waiting for the slowest submodule. DISTMM further optimizes the locality of submodules by leveraging the heterogeneous bandwidth of interconnections among accelerators. To address the limitation of existing pipeline execution schedules, we propose a new pipeline execution primitive, called batch-sync instruction, and a corresponding schedule, called DISTMM-Pipe. We build a prototype of DISTMM and evaluate it with existing solutions on models with various sizes ranging from 1.1 billion to 26 billion parameters and observe 1.32-3.27$\times$ speedup over Megatron-LM.

[*]Work was done when interned at AWS.

1 Introduction

Deep learning has recently witnessed the rise of foundation unimodal models focused on processing unimodal data such as GPT [4], LLaMA [36] and ViT [9], where models are pre-trained on web-scale text or image data. However, real-world data typically do not exist in isolation so human experience itself is multimodal. For example, a movie is a blend of visuals, dialogue, and music. To bridge the gap between isolated data modalities, multimodal models have emerged as a transformative paradigm, which offers a holistic representation to enhance performance in real-world scenarios, capturing intricate relationships that remain elusive to unimodal models.

A multimodal model has unique characteristics in terms of model structure, which consists of multiple submodules. Each submodule has a specific functionality, e.g., transforming image inputs to feature vectors, or combining multiple feature vectors, etc. As the functionalities are different, the architecture and the scale of the submodules vary accordingly. For example, Contrastive Language–Image Pre-training (CLIP) [29] adopts a deeper and wider structure for processing image inputs, while using a relatively shallow design of a submodule for transforming text inputs. Furthermore, the data input sizes also vary across submodules, e.g., the images in Common Objects in Context (COCO) dataset [20] have an input size of 512×512 pixels, while the input length of corresponding text captions is from 5 to 20 words. The heterogeneity among submodules leads to different demands of computing powers. However, existing distributed training systems are mainly designed for training unimodal models with homogeneous computing power requirements, leading to suboptimal performance in multimodal model training.

Moreover, multimodal models require a large number of data samples to compute the contrastive loss function due to model quality requirements [5]. In the distributed training setting, scaling out multimodal model training with data parallelism and tensor parallelism linearly increases the maximum batch size for contrastive loss. However, scaling out multimodal model training with pipeline parallelism results in a bounded maximum batch size for contrastive loss per forward and backward. The absence of pipeline parallelism support hinders the scalability of the multimodal model both in training speed and model size.

To address the aforementioned challenges (more details in Sec 3) for efficient multimodal model training, this paper proposes DISTMM, which is, to the best of our knowledge, the first distributed training system specifically designed for multimodal models. DISTMM introduces four components that work collaboratively to optimize multimodal model training. For a given multimodal model training task and a cluster configuration, *the modality-aware partitioner* of DISTMM adaptively applies parallelism strategies for different submodules to minimize the overhead from parallelization. With efficient parallelism for each submodule, *the load balancer* redistributes the training data to ensure balanced computations among the submodules, which minimizes the computing resource idle time of waiting for the slowest submodule. *The placement manager* then assigns devices for each submodule based on the parallelism solution and the interconnect topology of devices, so that DISTMM can utilize heterogeneous bandwidth. Finally, to achieve a targeted sample size required by the training task, *the pipeline executor* generates the execution schedule (i.e., when and what to do for computations, communications, or memory movements) for combining outputs across multiple micro-batches of input data.

In summary, we make the following contributions:

- We propose an idea to partition and parallelize the submodules of a multimodal model based on their modalities and redistribute the training data, resulting in balanced computation among submodules with high computation efficiency.
- We design a submodule placement mechanism to reduce the communication volume by aligning the model's heterogeneity with the heterogeneous bandwidth in the cluster.
- We propose a new pipeline parallelism instruction to increase the supported batch size by fully utilizing the memory capacity, and a corresponding pipeline parallelism schedule to avoid dependency overheads introduced by the new instructions.
- Based on the above ideas, we build DISTMM and evaluate it on eight Amazon EC2 p3.16xlarge instances with 64 GPUs. Our results show that DISTMM achieves 1.32-3.27× speedup over Megatron-LM on three structurally different multimodal models, i.e., CLIP, CoCa, and LiT.

2 Background

2.1 Unimodal and Multimodal Models

Unimodal models. The unimodal models are designed to interpret and learn from one specific type of data modality, e.g., text, images, or audio. The model architecture is thus homogeneous with sequential execution order for consistently processing a single type of data. A typical unimodal model architecture consists of multiple transformer-based layers [4, 7, 36, 37]. Within each layer, the structure is the same, i.e., attention mechanisms and feed-forward networks.

Multimodal models. On the other hand, multimodal models aim to combine and align multiple data modalities, e.g., processing image and text data simultaneously. It relies on specialized architectures or configurations for processing each modality and aligning the processed feature vectors [9, 29, 38]. Typically, a multimodal model consists of multiple submodules each processing one modality independently. As an example, CLIP [29] employs two specialized modal submodules, namely, text and image. The image submodule is a Vision Transformer (ViT) [9] while the text submodule is a traditional Transformer submodule [37]. On top of multiple modal submodules, the modality interactive submodule takes the outputs from each modal submodule to compute the correlation among them. An exception is the Multimodal Large Language Model (MLLM) [1, 21, 41], whose LLM submodule integrates both the functionality of modality interactive submodule and text modal submodule.

2.2 Multimodal Model Training

To perform various multimodal tasks, multimodal models require different learning approaches. Fusion models integrate multimodal data to perform tasks including classification, detection, or prediction. This integration is either conducted through contrastive learning or simply achieved by fusing the features through operations such as cross-attention [38]. Co-learning models leverage information from one modality to improve or supplement the learning in another modality. It is achieved by contrastive learning to identify a unified representation containing multimodal information. Generative models generate content based on contextual understanding across multiple data modalities. For generative models such as MLLM, modal submodules are trained separately with contrastive learning first, and then the whole model will be trained with multimodal instruction tuning [21, 41].

Contrastive learning. Out of all multimodal learning methods used in various multimodal models, contrastive learning is the most fundamental learning method. Since aligning the representation across features of different modalities is crucial for the downstream task, contrastive learning generates positive and negative samples to contrast them to learn a robust unified representation across different modalities. Compared to cross-modal learning or reconstruction-based representation learning, contrastive learning is more effective by bridging the processing abilities between different modal submodules according to previous study [38]. Most multimodal models adopt contrastive learning [5] as the only training method or combine it with other unimodal learning methods.

Unlike only comparing each sample's output feature vector with the corresponding label in unimodal learning, contrastive learning systematically compares the feature vector of each data point with all feature vectors across diverse modalities. Specifically, positive pairs are constructed by comparing the feature vectors of a single sample across different modalities, affirming their shared identity. Conversely, negative pairs are formed by comparing feature vectors from distinct samples and modalities, establishing dissimilarity. Multimodal models learn effective representations and understand relationships between modalities by distinguishing positive and negative pairs. To achieve this, the core of contrastive learning is to compute a similarity matrix, which measures pairwise similarities between feature vectors. During multimodal model training, the loss of contrastive learning maximizes the similarity scores of positive pairs and minimizes the similarity scores of negative pairs. For example, CLIP model performs contrastive learning by using dot products of their text and vision submodules' feature vectors to compute a similarity matrix. With similarity matrix, the objective of CLIP training is to minimize a cross-entropy loss over similarity scores.

The effectiveness of contrastive learning relies on a large number of positive and negative samples from feature vectors generated by the same model update [2]. For instance, SimCLR [6] used a batch size of 4096 to generate 4096 positive samples and 33,554,432 negative samples. CLIP [29] used a batch size of 32,768 to generate 32,768 positive samples and 2,147,483,648 negative samples. Some low-cost alternatives utilize feature vectors from the previous model updates to approximate comparison in a large batch size. One problem of such approximation is the downgraded model quality since feature vectors are no longer consistent. PIRL [24] used a memory bank to store the previous feature vectors. Meanwhile, MoCo [12] replaced the memory bank with a queue of feature vectors generated by its momentum encoders. According to theory in [17], a larger number of consistent positive and negative samples enhance the model's discriminative power, generalization ability, and understanding of complex cross-modal relationships.

3 Motivation

This section discusses the problems in existing solutions and corresponding objectives in designing a multimodal model training system. The existing solutions did not take the unique model architecture and special learning paradigm of multimodal models into account.

3.1 Multimodal Model Characteristics.

Heterogeneous submodules. As discussed in Section 2.1, multimodal models have multiple modal submodules and one or more modality interactive submodules. Influenced by the design choices and architectural complexity, the sizes

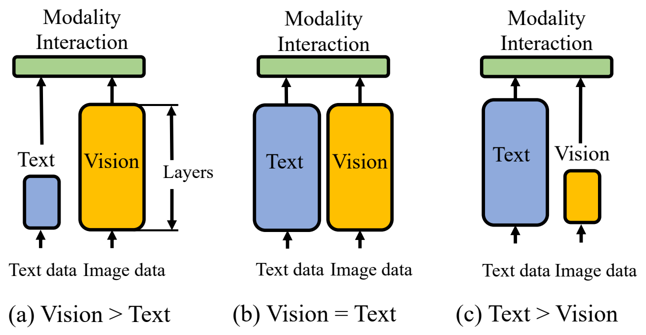

Figure 1: Three categories of vision-language models. The height of each rectangle denotes the number of layers and the width denotes hidden dimension of each layer. Text and Vision denote text submodule and vision submodule.

of submodules in multimodal models can vary significantly, which introduces heterogeneity.

Each modality (e.g., text, image, and audio) has unique characteristics and features that may require specialized processing to effectively capture modality-specific patterns. The focus of the multimodal task may also influence the processing strategy, which requires more understanding of fine-grained details on one modality than on others. The inherent complexity of each modality and the task requirements result in different computational demands among the modal submodules. Figure 1 illustrates three different archetypes of vision-language models. CLIP model belongs to Figure 1(a). The heterogeneity of CLIP model is driven by its vision-focused requirements, leading to a larger vision submodule. The LiT [10] model belongs to the category of Figure 1(c), whose text submodule has more parameters. The Contrastive captioner (CoCa) [38] model falls in the category of Figure 1(b), which uses equal-sized vision and text submodules.

Heterogenous submodules have varied GPU utilization. The parameter size differences result in varied hidden dimensions between heterogenous submodules, which lead to differently scaled operations to execute. The scale of operations affects the choices of kernels for underlying libraries. Large-scale operations are likely to be represented with more efficient kernels than small-scale operations. Additionally, fixed overheads such as kernel launches and memory transfers are amortized by large-scale operations. The training efficiency is limited by the submodules with fewer parameters, as they may not utilize the hardware as efficiently as the larger submodules.

Imbalanced input sizes. Depending on the modality type, the input sizes can vary significantly as well, e.g., the text input length is 77 words while the image input size is 512$\times$512 pixels for CLIP training. Larger input sizes lead to more efficient computation by leveraging the parallelism and vectorization capabilities of the hardware. This results in optimized memory utilization and computational throughput. Within the multimodal model consists of submodules with imbalanced inputs, the training efficiency is limited by the submodule with smaller input sizes.

Large batch size requirement. According to the observations in [5, 29, 39] and the theory in [17], a larger number of negative and positive samples are beneficial for learning robust and generalizable representations in contrastive learning, which directly results in a trained model with higher quality (i.e., more accurate, more robust, and more reliable). Furthermore, prior study [6] shows that training with a larger number of negative and positive samples not only converges faster but converges to a model that performs better, no matter how long it is trained. Increasing the batch size for the similarity matrix is the only way to generate the sufficient number of negative and positive samples without bringing inconsistency. For example, given a batch size of N, there are N positive pairs and $N^2 - N$ negative pairs in a similarity matrix. As a result, multimodal model training in practice requires a large batch size to ensure expected model quality.

3.2 System Challenges.

Although 3-dimension (3D) parallelism strategies (data, tensor, and pipeline parallelism) are effective for most unimodal models, it is inefficient to apply them to train multimodal models. We first lay out the reasons why applying the 3D parallelism to multimodal model training is inefficient, then conclude with the objectives on how to optimize 3D parallelism for multimodal models.

Memory overhead and imbalanced computation of Data Parallelism (DP). For models that can easily fit into one device, data parallelism is the go-to strategy to parallelize the training. Data parallelism only partitions batch dimensions and evenly distributes the data batch to all devices. Applying data parallelism to a multimodal model training results in all submodules being colocated on each device. Due to the colocation, the heterogeneity and imbalanced input sizes among different submodules lead to uneven computation scales and hardware utilization. Moreover, the colocation of different submodules also reduces the available memory per submodule, which limits the batch size for computation and eventually reduces the computation efficiency. In conclusion, the heterogeneous nature of the multimodal model cannot be fully utilized under the colocation solution of data parallelism. Therefore, it is needed to have a non-colocation solution to support more efficient multimodal model training.

Unnecessary partition overheads of Tensor Parallelism (TP). For models that cannot fit into a single device, tensor parallelism is used to evenly split the model and parallelize the model partition's execution. For multimodal model training, tensor parallelism homogeneously partitions submodules with different sizes according to the entire model's memory consumption. Since the smaller submodule only contributes little to the entire model's memory consumption, tensor parallelism will overly partition the smaller submodule, leading to unnecessary overhead. To address this issue, it is needed to have an adaptive partitioning method that considers the model sizes and structures of different submodules.

Unexpected training semantics with Pipeline Parallelism (PP). As discussed in 3.1, multimodal model's model quality is associated with the training batch size. However, all existing pipeline parallelism schedules, including 1F1B [11], GPipe [13], and PipeDream [25] split a global batch into multiple micro-batches and conduct pipelined forward and backward step on every micro-batch. Applying such pipeline parallelism to multimodal model training results in model quality degradation compared to non-pipeline parallelism approaches. Therefore, a new pipeline parallelism scheme for multimodal model training is needed.

4 DistMM Overview

DistMM is a distributed training system optimized for multimodal model training workloads by tackling the challenges discussed in Section 3.2. The design of DistMM improves the computation efficiency of each submodule in a multimodal model, reduces communication overheads in multi-node distributed training, and ensures the model quality when pipeline parallelism is used for training large models. By design, DistMM treats submodules within a model separately with independent parallelism strategies to maximize efficiency. In addition, it aligns the computation duration of each submodule to minimize the cost of interactions among submodules. As shown in Figure 2, DistMM has four components, which are Modality-aware partitioner, Data load balancer, Heterogeneity-aware placement manager, and Pipeline executor. We overview the components in the following text and leave the detailed design and analysis in Section 5.

Modality-aware partitioner. Modality-aware partitioner splits the whole multimodal model into submodules based on their input modalities. The neural network architecture and configuration of each submodule are typically different from each other, due to the input modality differences, e.g., using ViT [9] for vision inputs and BERT [7] for text inputs. Based on the submodule sizes, Modality-aware partitioner applies independent parallelism strategies to avoid overprovisioning parallelism degrees for small submodules. Modality-aware partitioner only handles model-level partitioning by transforming model description input into submodule partitions, leaving batch dimension partitioning to Data load balancer. In principle, this allows DistMM to maintain high computation efficiency for each submodule.

Data load balancer. Without realizing the structural differences, a naive way is to assign the same amount of computing

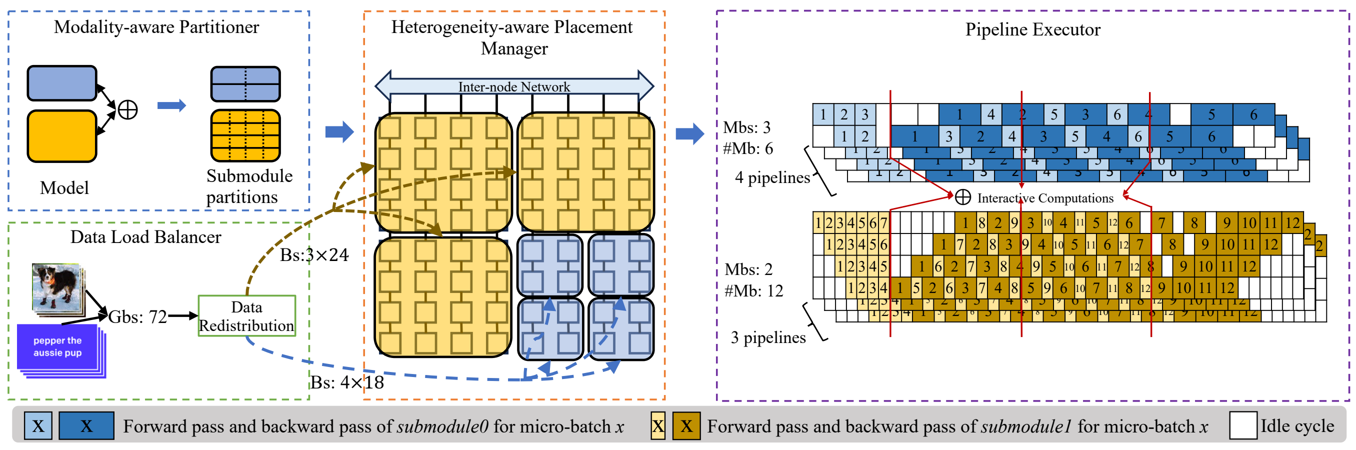

Figure 2: DISTMM overview.

devices and utilize the same batch size across all the submodule partitions of different modalities. This can easily lead to computing resources idling at the interactive computation phase because some submodules need fewer computation cycles than others. To achieve overall high computation efficiency, submodule partitions require a careful design for the resource allocation and adjustment of the batch size for each modality input. Data load balancer takes submodule partitions and cluster configuration as inputs, assigning different numbers of devices to different submodule partitions based on their estimated memory consumption and measured execution time. To minimize device idle time introduced by the heterogeneous resource placement, Data load balancer redistributes batch sizes of different modalities to balance the computation duration.

Heterogeneity-aware placement manager. Computing clusters typically have much higher bandwidths for intra-node communication (e.g., NVLink) than that for inter-node communication (e.g., Ethernet). Heterogeneity-aware placement manager groups submodule partitions within the same modality and places them close to each other to exploit the high-bandwidth links for frequent communication. Furthermore, Heterogeneity-aware placement manager places submodule partitions from different modalities on separate nodes so that the infrequent communication volume (e.g., gradient AllReduce) on low-bandwidth links will be reduced.

Pipeline executor. Pipeline executor generates a customized pipeline parallelism execution schedule for each submodule. The customized schedule maintains the semantics of the training procedure and minimizes hardware idle time. Pipeline executor inserts *mid-point synchronization* to maintain the semantics of the modality interaction among different modal submodules, in which each submodule gathers activations of prior micro-batches (from last mid-point synchronization till now) from submodules with different modalities and itself to complete the modality interaction.

An illustrated example. Figure 2 provides an example of training a vision-language model with DISTMM in a cluster with 64 devices across eight nodes. Assuming the multimodal model has more parameters for encoding image data (yellow) than that of processing text inputs (blue).

Modality-aware partitioner assigns different parallelism strategies for each modal submodule, shown in the top-left part. Different submodules are replicated differently (four versus three) with different numbers of devices by Data load balancer, so that submodules can complete the computation of different batch sizes (18 versus 24) for a given global batch at the same time.

Heterogeneity-aware placement manager deploys vision submodules and text submodules on different devices and groups the devices with the same modality, which reduces both intra-node and inter-node communication volume.

Pipeline executor generates customized pipeline execution plans for each device which integrates a synchronization instruction to preserve the same training semantics with non-pipeline parallelism solution. The synchronization instruction gathers feature vectors to conduct interactive computation. In this case, a gathered global image feature vector consists of 3 pipeline parallelism groups' 4 previous image feature vectors, where each image feature vector's batch size is 2. Similarly, a gathered global text feature vector consists of 4 pipeline parallelism groups' 2 previous text feature vectors, where each text feature vector's batch size is 3. The workloads on submodules for different modalities are balanced by Data load balancer which ensures that they arrive at the synchronized interactive computation at the same time.

5 DistMM Design

5.1 Modality-aware Partitioner

The heterogeneity of the model structure and the difference of the inputs across modalities are the root cause of the low overall computation efficiency in multimodal training (detailed in Section 3.1). Modality-aware partitioner therefore applies adaptive partition strategy to improve the computation efficiency, which accounts for heterogeneity among submodules.

Adaptive Partition Allocation. We design a new partition strategy that fully utilizes the heterogeneous structure with the multimodal model. Modality-aware partitioner takes model definition as an input, which contains the configuration of each modal submodule and modality interactive submodule.

Modality-aware partitioner first parallelizes submodules with independent parallelism strategies to satisfy memory constraints while maintaining low overheads. As discussed in 3.1, modality interactive submodule focuses on aligning high-level information, which does not require a complex computation. It therefore has a highly parallelizable nature. So Modality-aware partitioner equally distributes the computation load of modality interactive submodule to every device.

Modality-aware partitioner then individually applies adaptive partitioning to each modal submodule to maximize efficiency. This component follows the common practice [15,34] of efficiently training an unimodal model, which involves determining the minimum parallelism degree required and selecting parallelism strategies accordingly, with careful consideration of the model size. For modal submodules that can fit into a single device, our approach advocates the use of data parallelism to maximize efficiency. Alternatively, for modal submodules that cannot fit within a single device, employing tensor parallelism is recommended. Notably, for modal submodules that surpass the capacity of a single node, our strategy favors pipeline parallelism, as extant research [26] indicates that extending tensor parallelism across nodes introduces significant overheads. The resulting model partition on each device after Modality-aware partitioner is an adaptively partitioned modal submodule and evenly parallelized modality interactive submodules.

An illustrative example. We construct examples with hypothetical performance numbers to illustrate how Modality-aware partitioner works. Figure 3 (a) shows an example of applying the tensor parallelism in a homogeneous way for the entire model. This homogeneous way of applying tensor parallelism leads to the sequential execution of the first and smaller submodule (blue colored) with 10% GPU utilization, and the second and larger submodule (orange colored) with 30%. As a result, its average GPU utilization for each GPU is $2 \cdot 10\% + 3 \cdot 30\% = 22\%$, where the smaller submodule's 10% utilization is the performance bottleneck. Figure 3 (b) shows the partition result by Modality-aware partitioner, where the

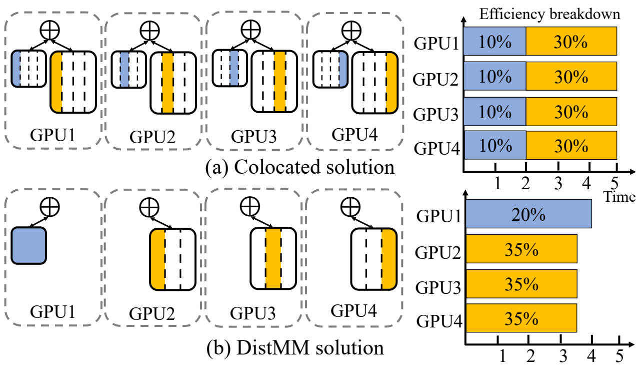

Figure 3: Modality-aware partitioner example. Placements (on left) describe how submodules are partitioned. ⊕ denotes distributed modality interactive submodules. Efficiency breakdown (on right) includes each submodule's duration with its GPU utilization. The x-axis stands for each submodule's duration and y-axis represents the GPU id. Percentages in the duration stand for each submodule's GPU utilization.

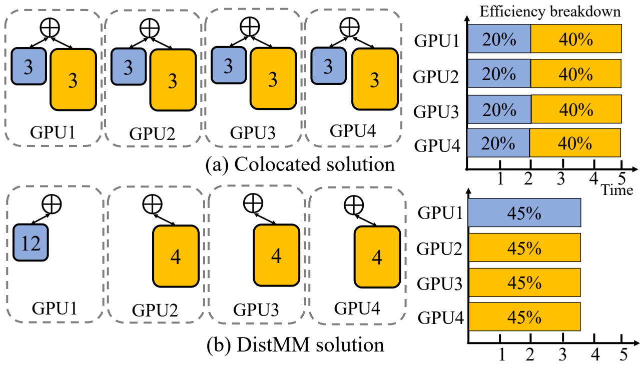

Figure 4: Data load balancer example. Placements (on left) describe the data sample assignment for each submodule. Numbers inside the submodule are the number of assigned data samples per batch.

smaller submodule is not partitioned and the larger submodule is partitioned into three parts via the tensor parallelism. The separation of submodules and adaptive tensor parallelism partitioning allow the smaller submodule to achieve a much better GPU utilization. The GPU utilization of the smaller submodule can be improved from 10% to 20% due to the increase in computational workload. The improved average GPU utilization helps reduce the duration of the entire model. However, it also introduces additional overhead of imbalanced duration among GPUs (i.e., GPU1 versus other GPUs).

5.2 Data Load Balancer

While the adaptive partition strategy reduces partition overheads, it does not address the heterogeneity of inputs. Modality-aware partitioner does not assign the batch size of input data to submodule partitions. Different modal submodule partitions have different computation patterns and inputs which results in different execution durations and memory consumptions under the same batch size. Without considering the differences, some modal submodules may finish first and have to wait for others to gather the outputs.

We view Data load balancer as a resource manager, and the goal is to balance the computing duration among different modal submodules under memory constraints. Data load balancer takes a cluster setup, i.e., the number of nodes and the number of devices per node, and training configuration, i.e., the model partitions generated by Modality-aware partitioner and the global batch size, as inputs. Note that each model partition consists of an adaptively partitioned modal submodule for a specific modality and an evenly partitioned modality interactive submodule. Data load balancer produces a resource assignment plan, which assigns the number of devices and the data batch size for each model partition. This resource assignment problem is equivalent to minimizing the time taken by the slowest model partition, and has the optimal substructure property: a resource assignment plan that maximizes throughput is composed of resource assignments for partial modalities that maximize throughput for smaller clusters. DISTMM uses dynamic programming to find the optimal solution.

An illustrative example. We construct examples with hypothetical performance numbers to illustrate how Data load balancer works. Figure 4 provides an example resource assignment plan from Data load balancer and compares it with a colocated plan. The colocated plan colocates multiple modal submodules and replicates each submodule onto all devices (Figure 4 (a)). In contrast, Data load balancer together with Modality-aware partitioner separates modal submodules with different modalities and replicates them differently, i.e., the smaller submodule has one replica, and the larger submodule has three replicas (Figure 4 (b)). With a smaller number of replicas for each submodule, the number of samples per device is increased, especially for the smaller submodule. Thus, the computation efficiency improves for both submodules.

5.3 Heterogeneity-aware placement manager

The generated resource assignment plan from Data load balancer does not include the model placement to specific devices. Heterogeneity-aware placement manager deploys the resource assignment plan by placing the submodules in a communication-efficient way. It takes bandwidth heterogeneity into consideration to optimize the communication overhead in distributed multimodal model training.

5.3.1 Intra-submodule placements

Heterogeneity-aware placement manager prioritizes the communication patterns within a single modal submodule for placement assignments. The priority is determined by the communication frequency and the data volume transmitted in each communication pattern. Heterogeneity-aware placement manager generates placement assignments where communications with higher priority (TP) are conducted in the network with higher bandwidth (NVLinks within node) and communications with lower priority (PP and DP) are conducted in the lower bandwidth network (Ethernet).

5.3.2 Inter-submodule placements

As a non-colocated solution, Heterogeneity-aware placement manager only places one single-modality partition together with evenly parallelized modality interactive submodules on a single device, which reduces the communication volume involved in modality interaction and gradient synchronization.

Overhead reduction in modality interaction. As described in section 2.2, multimodal model's modality interaction consists of similarity matrix computation. For the multimodal models with vision modality and text modality, the similarity computation on each device consists of two dot products in the distributed setting, the first dot product is between the local image feature vectors and gathered entire text feature vectors and the second dot product is between the local text feature vectors and gathered image feature vectors. With non-colocated placement, since there is only one modal submodule on each device, the similarity computation is reduced to one dot product between the feature vectors of local modality and the gathered feature vectors of opposite modality. So Heterogeneity-aware placement manager halves the communication volume of modality interaction compared with the colocated placement since image and text feature vectors share the same tensor shape.

Table 1: The list of symbols frequently used in the paper.

Symbol	Description
Mbs	Micro-batch size
Rbs	Required micro-batch size for modality interaction
K	Required number of micro-batches ($K = Rbs/Mbs$)
N	Total number of GPUs
P	The number of pipeline parallelism degree
M	Memory capacity per GPU
M_s	Static (Weight, gradient, state) memory consumption
M_g	Gradient memory consumption
M_a	Activation memory consumption

Note: Above symbols are of the whole multimodal model. We add index 1 and 2 to the above symbols to stand for the first and second submodules.

Overhead reduction in gradient synchronization. For communication in gradient all-reduce, Heterogeneity-aware placement manager reduces the communication volume from the entire model's gradients in colocated solution to the gradients of the submodules hosted in this specific device. In the whole

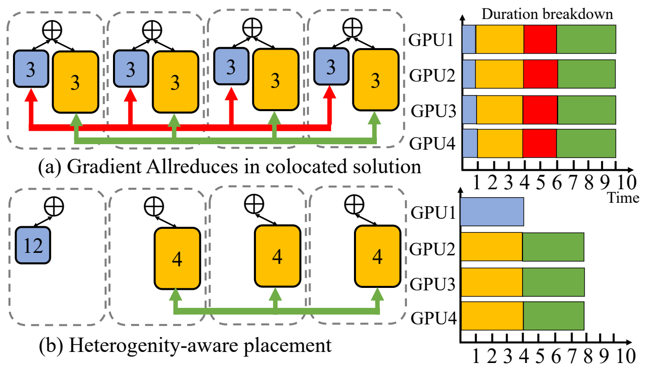
Duration breakdown
GPU1
GPU2
GPU3
GPU4
1 2 3 4 5 6 7 8 9 10
Time
(a) Gradient Allreduces in colocated solution
(b) Heterogenity-aware placement
3
12
4

Figure 5: Communication reduction example. Placements (on left) describe the patterns of gradient AllReduce (green and red) under different placements. Duration breakdown (on right) includes the computation and communication durations within each device. The x-axis stands for each submodule's computation or communication duration.

system's perspective, the volume is reduced from $M_{g1} + M_{g2}$ to $\max(M_{g1}, M_{g2})$. As shown in examples with hypothetical duration numbers described by Figure 5, gradient communication volume is reduced from the whole model's gradients (red and green) to only the gradient of the larger submodule (green). The gradient all-reduce volume does not change with the mini-batch size. The computing duration increases when the mini-batch size increases, which further reduces the communication duration ratio in total training time.

5.4 Pipeline Executor

When applying existing pipeline parallelism to multimodal model training, the available memory after static memory consumption (weight, gradient, and state) is divided into P blocks to store P micro-batches' activation to ensure efficient pipelining. Since static memory consumption and activation are divided into P partitions, the maximum micro-batch size is $((M - M_s/P)/(M_a/P))/P = (M - M_s/P)/M_a)$, which is bounded by M/M_a. Such bounded micro-batch size is much smaller than what is needed for training a quality multimodal model in practice. Note that Table 1 shows the meaning of the symbols used in this paper. As a comparison, data parallelism's maximum micro-batch size is $(M \cdot N - M_s \cdot N)/M_a) = N \cdot (M - M_s \cdot)/M_a)$ and the tensor parallelism's maximum micro-batch size is $(M \cdot N - M_s)/M_a)$, which means scaling out the cluster size N can linearly increase the maximum micro-batch size.

To address this issue, we propose a new instruction in pipeline parallelism called batch-sync instruction used by the Pipeline executor. This instruction can preserve the required semantics by ensuring the modality interactive submodule is executed with the needed large batch size. We also propose a new pipeline parallelism schedule called DistMM-Pipe which adopts batch-sync instruction without introducing any extra idle cycles.

Batch-sync instruction. Given a model quality requirement of batch size Rbs and $K = Rbs/Mbs$, Pipeline executor's batch-sync instruction consists of the following four steps: (1) the memory movement step that concatenates K feature vectors computed by previous K forward instructions into a continuous feature vector, (2) the forward pass of modal interactive submodule is executed with the continuous feature vector, (3) the backward pass of modality interactive submodule is executed, which produces gradients corresponding to the continuous feature vector, and (4) the memory dispatching step that dispatches the continuous gradients to K gradients corresponding to the feature vectors of each micro-batch.

However, integrating batch-sync instruction to pipeline parallelism schedule introduces overhead due to dependency issues. Since the batch-sync instruction depends on the previous forward instructions and vice versa, the backward instructions depend on the batch-sync instruction. The batch-sync instruction works as a barrier between the involved forward and backward instructions. In 1 forward and 1 backward (1F1B) schedule [25], batch-sync instruction can only be inserted between the 1 forward and 1 backward of a single micro-batch due to dependency, which limits the maximum batch size for modality interaction to a single micro-batch. In GPipe schedule [13], batch-sync instructions can be inserted between the required number of forward and backward instructions. As shown in Figure 6, applying the batch-sync instruction without a customized schedule results in dependency issues, where forward and backward instructions are separated by the corresponding batch-sync instruction and cause extra idle cycles.

DistMM-Pipe schedule. To bypass the dependency barrier so as to mitigate the overhead introduced by the batch-sync instruction, we propose DistMM-Pipe schedule. DistMM-Pipe launches $2 \cdot K$ micro-batches with $Mbs/2$ as the micro-batch size for each gradient accumulation cycle. The doubled micro-batches bypass the dependency barrier since each batch-sync instruction only introduces dependency issues between K forward and backward instructions. As shown in Figure 6, after each batch-sync instruction, backward instruction and forward instruction belonging to different K micro-batches are executed in an interleaved way under DistMM-Pipe schedule, which ensures there are no extra idle cycles caused by dependency issues.

In terms of model quality, Pipeline executor's maximum batch size to the modality interaction can be formulated as $((M - M_s/P)/(M_a/(2 \cdot P))) = (M \cdot P - M_s)/(2 \cdot M_a)$, which grows linearly with the number of pipeline stages P. Compared to existing pipeline parallelism's maximum batch size $(M - M_s/P)/M_a$ which is bounded by M/M_a, Pipeline executor can preserve any model quality requirements by scaling out the cluster size while existing pipeline parallelism cannot.

Schedule alignment. With the pipeline parallelism degree

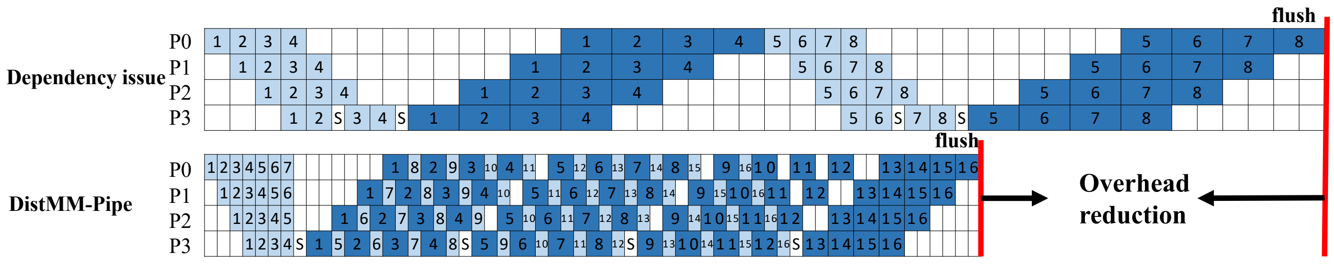

Figure 6: Comparison between pipeline parallelism schedule with dependency issue and DISTMM-Pipe schedule. S stands for batch-sync instruction. The modal quality requirement is $2 \cdot Mbs$.

assigned by Modality-aware partitioner and resource assignment plan by Data load balancer, Pipeline executor generates the DISTMM-Pipe schedule for each modal submodule. The schedules for different modal submodules are aligned through batch-sync instructions, which have a gathering communication for the global similarity matrix. Since the computation durations of each submodule are already balanced by Data load balancer, this alignment will not introduce any overhead caused by imbalanced durations between different schedules.

6 Implementation

We have built DISTMM with ∼2600 lines of code in Python. DISTMM is built on top of PyTorch [28]. It provides simple APIs to transform model scripts written for training on a single device to distributed multi-node multi-device setups. The goal of DISTMM is to minimize the user effort while fully utilizing the hardware resources.

APIs. There are four types of APIs: *partition annotation*, *training loop*, *data loading*, and *initialization*. Users annotate each submodule with the *partition annotation* API to allow DISTMM to partition them accordingly based on the pattern described by annotation. The *training loop* API provides an integrated function that includes both forward and backward computation. Users can use this integrated function to train the model, instead of constructing the training loop by themselves. For logging purposes, we provide hooks to log the intermediate results, e.g., loss values. For loading data samples correctly for load balancing purposes (Section 5.2), users need to wrap their data loading procedure using the *data loading* API. The *data loading* API provides the same user interface as the one from PyTorch framework, allowing a drop-in replacement for the training script with PyTorch. Before the computation starts on the cluster, training scripts are required to call the *initialization* API with configurations of the cluster and the model.

Configuration. DISTMM includes the multimodal model implementations on OpenCLIP [14] with DISTMM's partition annotations. The implemented models can be launched by specifying the model type in the training script. For models that have not been implemented, users need to provide the model description and add partition annotations to the corresponding operations and submodules, which helps Modality-aware partitioner to correctly recognize and partition the whole model. Users need to replace forward and backward calls in the training loop with DISTMM's integrated functions, which return the equivalent result as a single forward and backward computation. For submodules with pipeline parallelism partitioning, the actual computation is a sequence of forward and backward computations under DISTMM-Pipe schedule. Besides DISTMM's forward and backward replacement, users need to call DISTMM's initialization and specify the data source with DISTMM's data loading API in the training loop. Users also need to provide the environment configuration including cluster size, node size, and GPU memory limits for Heterogeneity-aware placement manager. To control the partitioning process, users can provide a partition strategy configuration including the partition strategy candidates and corresponding partition degree range to limit the search or simply specify a particular partitioning solution.

7 Evaluation

In this section, we seek to answer the following questions:

- How well does DISTMM perform on models with different structures?
- How well does DISTMM perform on models ranging from small scale to large scale?
- How effective is each component? Specifically, What is the impact of Modality-aware partitioner on the efficiency improvement of submodules with different structures? What is the impact of Data load balancer on the efficiency improvement with fixed submodule partitions? What is the impact of Heterogeneity-aware placement manager on reducing inter-node communication volume with fixed submodule partitions and fixed resource allocation plans?

Our experiments are conducted on a cluster of 8 Amazon EC2 p3.16xlarge nodes where each node contains dual 24-core Intel Xeon CPUs and 8 NVIDIA Tesla V100 (16GB) GPUs. The GPUs within a node are connected with NVLinks. The nodes are connected to 25 Gbps networks. All nodes run Ubuntu 20.04, CUDA 11.6 and PyTorch 2.0.1.

We measure the iteration time of training three different models, namely, CLIP [29], CoCa [38], LiT [39]. Each model has unique characteristics in terms of the model structure. Specifically, CLIP model, which is designed for image-text retrieval (ITR), has a larger submodule for image modality input; CoCa model, which is used in natural language for visual reasoning (NLVR), has a relatively balanced (i.e., sizes of the submodules are similar) submodules for image and text inputs; LiT model, which is designed for visual question answering (VQA), has a larger submodule for text inputs. These selected models are representative, as they comprehensively cover the design space for multimodal models with two input data modalities, e.g., vision and language in our evaluation.

We further scale up the model sizes to study the system performance characteristics according to scaling law [16]. In our evaluation, we measure the system performance of model sizes in three categories: 1) *single-device scale*: the model is trainable on a single device; 2) *single-node scale*: the model requires memory more than the capacity of a single device, but the aggregated memory of devices in a single node is sufficient; 3) *multi-node scale*: the model is only trainable on multiple nodes. We summarize the configuration of models in Table 2.

We use Megatron-LM [34] as the baseline. Megatron-LM system provides state-of-the-art performance for training transformer-based unimodal models. As elaborated in Section 5.4, existing pipeline parallelism schedules cannot preserve the semantics of the user-defined training configuration. Therefore, we only use Tensor and Data Parallelisms from Megatron-LM. According to the convention of multimodal training for vision and language tasks, we use sequence length of 77 words and images sized 336×336 pixels [9].

7.1 End-to-End Performance

As shown in Figure 7, for models sized at *single-device scale*, DistMM is 1.32–1.39× faster than Megatron-LM. These models include CLIP(760M, 350M), CoCa(760M,760M), and LiT(350M, 760M) [1]. At this model scale, Megatron-LM colocates two submodules in every GPU. In comparison, DistMM uses Modality-aware partitioner to separate submodules, and balances the computation workloads among submodules via Data load balancer. DistMM also optimizes the placement with Heterogeneity-aware placement manager to lower the communication cost.

For models that are not trainable on a single GPU, but can be trained on eight GPUs on a single node, i.e., single-node scale, DistMM is 1.48–1.67× faster than Megatron-LM. The evaluated models include CLIP(6.7B, 2.7B), CoCa(6.7B, 6.7B), and LiT(2.7B, 6.7B). Megatron-LM partitions two submodules using the same tensor parallelism degrees. On the other hand, DistMM applies modality-aware partitioning and assigns the most computation-efficient tensor parallelism degree to each submodule depending on their parameter sizes.

[1](X, Y) denotes the sizes of vision and text submodule, respectively.

When scaling the model size to *multi-node scale*, DistMM outperforms the baseline by up to 3.27×. The evaluation includes CLIP(13B, 6.7B), CoCa(13B, 13B), and LiT(6.7B, 13B). For these models, we enable pipeline parallelism in DistMM using Pipeline executor. Megatron-LM cannot maintain the batch size requirement of the training (detailed in Section 5.4), thus only the tensor parallelism is enabled. To support model sizes at the multi-node scale, Megatron-LM uses tensor parallelism across multiple nodes, e.g., CoCa (13B, 13B) is partitioned into 16 tensor parallelism. Its 16 partitions are placed on two nodes. In comparison, DistMM uses tensor parallelism within each node, and leverages pipeline parallelism crossing nodes. Thus, we have much lower communication overhead [27, 34] as compared to Megatron-LM.

7.2 Effectiveness of Individual Components

To understand the sources of improvement, we designed a sequence of experiments, each dedicated to activating a single component of DistMM at a time. We tailored the model setup and cluster configuration for each targeted component to mitigate potential side effects and dependencies on other components. Modality-aware partitioner and Data load balancer contribute to the training duration reduction by improving GPU utilization through adaptive partitioning and load balancing. In this paper, the GPU utilization is calculated by:

$$\frac{\text{Model FLOPs}}{\text{FLOPS} \cdot T}$$

, where Model FLOPs is the number of floating-point operations required to perform a single forward and backward pass, FLOPS represents the number of floating-point operations per second supported by a GPU, and T denotes duration.

To evaluate the Modality-aware partitioner, we choose the single-node scaled models that need tensor parallelism to demonstrate the GPU utilization differences between DistMM's adaptive partitioning and Megatron-LM's homogenous partitioning. In the evaluation of Data load balancer, we need to mitigate the side effects of Modality-aware partitioner. To do this, we choose the single-device scaled models that do not need tensor parallelism to demonstrate the GPU utilization differences after Data load balancer's load balancing. Heterogeneity-aware placement manager contributes to the training duration reduction by reducing inter-node communication volume through communication-efficient placement. To evaluate it excluding DistMM's computation optimization side effects, we choose the single-device scaled models to demonstrate the communication percentage differences between Megatron-LM and Heterogeneity-aware placement manager's placement without load balancing. Since DistMM-Pipe is the only pipeline parallelism schedule that can be applied to multimodal model training without changing the training semantics as described in Section 5.4, DistMM-Pipe is effective by ensuring the required modal quality.

Model	**Vision submodule**			**Text submodule**			**Total # of Parameters**
	Layers	Hidden Size	# of Parameters	Layers	Hidden Size	# of Parameters	
CLIP	24	1536	760M	24	1024	350M	1.1B
	32	4096	6.7B	32	2560	2.7B	9.3B
	40	5140	13B	32	4096	6.7B	19.7B
CoCa	24	1536	760M	24	1536	760M	1.52B
	32	4096	6.7B	32	4096	6.7B	13.4B
	40	5140	13B	40	5140	13B	26B
LiT	24	1024	350M	24	1536	760M	1.1B
	32	2560	2.7B	32	4096	6.7B	9.4B
	32	4096	6.7B	40	5140	13B	19.7B

Table 2: Configuration details of the models: CLIP, CoCa, and LiT.

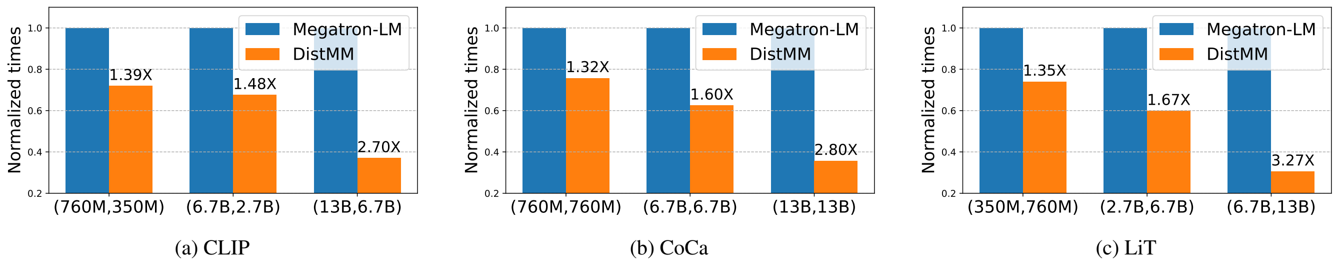

Figure 7: End-to-end iteration times of DISTMM normalized to Megatron-LM on three models with varied parameter configurations. The label of x-axis (X, Y): (size of vision submodule, size of text submodule)

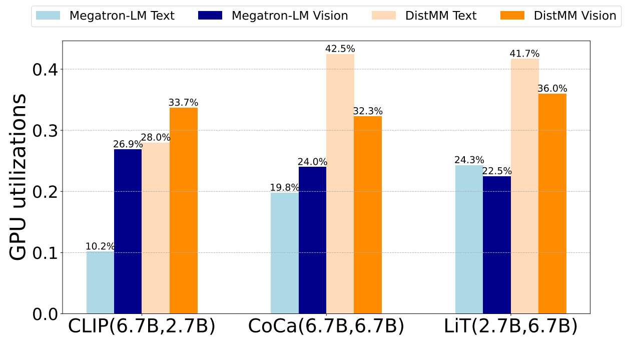

Figure 8: GPU utilization comparison on text and vision submodules between DISTMM and Megatron-LM's data parallelism and tensor parallelism combined solution.

7.2.1 Effectiveness of Modality-aware Partitioner

Experimental setup. We use performance timers to measure the computing duration when running CLIP(6.7B, 2.7B), CoCa(6.7B, 6.7B), and LiT(2.7B, 6.7B) in the 8 Amazon EC2 p3.16xlarge instances setting. Megatron-LM applies tensor parallelism degree 8 to both text and vision submodules and applies data parallelism to replicate the two submodules as a whole. In DISTMM's solution, the Modality-aware partitioner applies adaptive partitioning individually to two submodules.

Results. Figure 8 shows the GPU utilization differences of both text and vision submodules between DISTMM and Megatron-LM. For CLIP(6.7B, 2.7B), the text submodule's GPU utilization is largely increased, since the tensor parallelism is reduced from 8 to 4. For CoCa(6.7B, 6.7B), Megatron-LM's colocated solution limits the GPU utilization of text submodule by sharing the same batch size as vision submodule. In DISTMM's solution, text submodule and vision submodule share the same tensor parallelism partition degree but have larger batch sizes for increasing available memory under non-colocation. For LiT(2.7B, 6.7B), its lower activation memory consumption leads to a larger global batch size, resulting in higher GPU utilization than CLIP(6.7B, 2.7B).

7.2.2 Effectiveness of Data Load Balancer

Experimental setup. We use performance timers to measure the computation duration when running CLIP(760M, 350M), CoCa(760M, 760M), and LiT(350M, 760M) in the 8 Amazon EC2 p3.16xlarge instances setting. Megatron-LM applies data parallelism to the whole model. In DISTMM's solution, the Data load balancer redistributes the data to different submodules, resulting in more balanced workloads.

Results. As shown in Figure 9, the GPU utilization of both submodules have improved. The improvement amount is associated with the size of the submodule. DISTMM's Data load balancer puts more workload on the smaller submodule to balance the duration and efficiency. In CLIP(760M, 350M), the text submodule's batch size is increased from 8 to 150 while the vision submodule's batch size is increased from 8 to 10. The resulting GPU utilization improvement comes from increased batch sizes. For CoCa(760M, 760M), even though the parameter sizes are the same for text and vision submodules, the sequence length differences lead to the workload

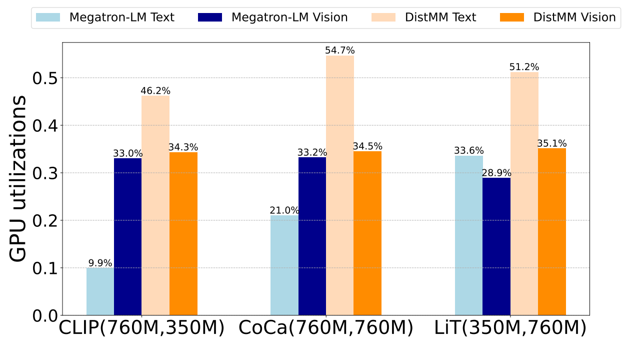

Figure 9: GPU utilization comparison on text and vision submodules between DISTMM and Megatron-LM.

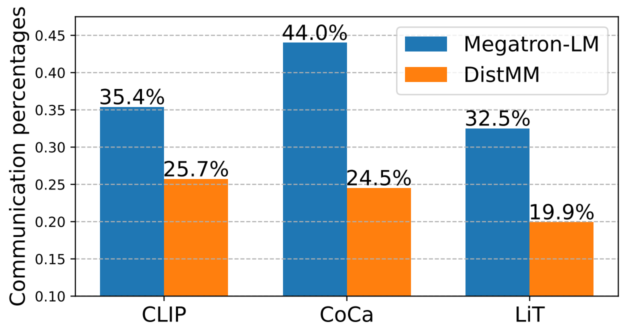

Figure 10: Communication duration's percentage in end-to-end training comparison between DISTMM and Megatron-LM's data parallelism solution.

imbalance. After the data load balancing, the text submodule gains a large GPU utilization improvement, where text submodule replicates 16 times and vision submodule replicates 48 times. For LiT(350M, 760M), since the submodules are balanced before Data load balancer, both submodules gain large GPU utilization improvements for increased batch sizes.

7.2.3 Effectiveness of Heterogeneity-aware Placement Manager

Experimental setup. We use performance timers to measure communication duration when running CLIP(760M, 350M), CoCa(760M, 760M), and LiT(350M, 760M) in the 8 Amazon EC2 p3.16xlarge instances setting. The communication duration involves all-gather communication for distributed modality interaction and all-reduce communication for gradient synchronization. Megatron-LM applies data parallelism to the whole model. In DISTMM's solution, Heterogeneity-aware placement manager places submodules of different modalities onto different nodes without load balancing.

Results. As shown in Figure 10, the communication percentages of each model have been reduced due to the communication volume reduction. DISTMM's solution achieves largest communication reduction in CoCa(760M, 760M), since its text and vision submodules share a similar parameter size.

8 Related Work

Model reordering. IOS [8] explores the parallel opportunities within the modal structure similar to DISTMM through reordering the operators on the same device to improve efficiency, while DISTMM reorders operators on different devices. Rammer [23] orders operators by fusing parallelizable operations without dependencies into a fused kernel to improve resource utilization, which focuses on inference with an extremely small batch size instead of DISTMM's training workload. DeepSpeed-MoE [30] proposes expert parallelism to reorder and parallelize homogeneous MoE experts, while DISTMM focuses on heterogeneous modal submodules.

Model partitioning. Several systems implement one or many strategies out of the 3D parallelism partitioning. Pytorch's DDP [19] and Horovod [32] replicate a model on every device to use data parallelism, while ZeRO [31] splits both weights and model states across all devices to accommodate larger models for data parallelism. FlexFlow [22], Mesh-Tensorflow [33], and Gshard [18] split operations in a way to represent data parallelism and model parallelism. Megatron [34] and DeepSpeed-Megatron [35] support 3D parallelism to parallelize the models in a manually optimized way. Alpa [40] and Pathway [3] combine single program multiple data (SPMD) and multiple programs multiple data (MPMD) abstraction to enable automatic 3D parallelism parallelization. However, unlike DISTMM, these systems do not consider the heterogeneous nature of multimodal models.

9 Conclusion

This paper introduced DISTMM, a distributed multimodal model training system. It consists of four heterogeneity-aware system components designed specifically for multimodal model training. DISTMM utilizes the parallelization opportunities within the multimodal model structure, and successfully solves the performance bottlenecks through adaptively partitioning each submodule and load balancing the partitions. To preserve the model quality, DISTMM also orchestrates a parallel execution by introducing new pipeline parallelism instruction and corresponding schedule. Our experiments show that DISTMM can outperform the state-of-the-art training system for models with varied structures and varied sizes.

10 Acknowledgments

The authors would like to thank our shepherd Dr. Deepak Narayanan and the anonymous reviewers for their valuable feedback and thoughtful suggestions.

References

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.

[2] Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. A theoretical analysis of contrastive unsupervised representation learning. *arXiv preprint arXiv:1902.09229*, 2019.

[3] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[6] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems*, 3:167–180, 2021.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[10] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[12] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[14] Gabriel Ilharco, Mitchell Wortsman, Ross Wightman, Cade Gordon, Nicholas Carlini, Rohan Taori, Achal Dave, Vaishaal Shankar, Hongseok Namkoong, John Miller, Hannaneh Hajishirzi, Ali Farhadi, and Ludwig Schmidt. Openclip. *https://doi.org/10.5281/zenodo.5143773*, 2021.

[15] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

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

[17] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673, 2020.

[18] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[19] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[20] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[21] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*, 2023.

[22] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[23] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 881–897, 2020.

[24] Ishan Misra and Laurens van der Maaten. Self-supervised learning of pretext-invariant representations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6707–6717, 2020.

[25] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[29] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[30] 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 *International Conference on Machine Learning*, pages 18332–18346. PMLR, 2022.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[32] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[33] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

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

[35] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[36] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[38] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917*, 2022.

[39] Xiaohua Zhai, Xiao Wang, Basil Mustafa, Andreas Steiner, Daniel Keysers, Alexander Kolesnikov, and Lucas Beyer. Lit: Zero-shot transfer with locked-image text tuning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18123–18133, 2022.

[40] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[41] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023.