BREAKING THE PARETO FRONTIER OF COMPUTE AND MEMORY IN MULTIMODAL LLM TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Training multimodal large language models (MLLMs) is challenged by both **model** and **data heterogeneity**. Existing systems redesign the training pipeline to address these challenges, but remain bound by a Pareto frontier between compute and memory efficiency, improving one only at the expense of the other. We present BIGMAC, a new training pipeline for multimodal LLMs. The core idea of BIGMAC is to elegantly nest the encoder and generator computation into the original LLM pipeline, forming a **dependency-safe nested pipeline** structure. With this design, BIGMAC reduces the activation memory complexity of the encoder and generator to $O(1)$ while keeping the activation memory complexity of the LLM unchanged. At the same time, it achieves the same computational efficiency as the idealized setting with unlimited memory. As a result, BIGMAC breaks the Pareto frontier between computational efficiency and memory usage, enabling simultaneous optimization of both computation and memory in MLLM training. We evaluate BIGMAC on multiple MLLMs and training workloads. Experimental results show that BIGMAC achieves a 1.08×–1.9× training speedup over baseline systems while maintaining stable memory usage as batch size increases.

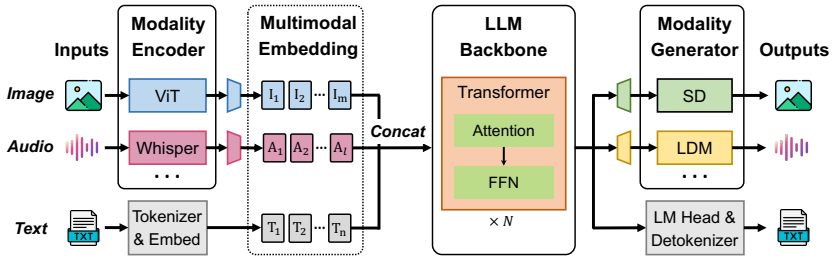


Figure 1: The architecture of multimodal LLMs.

1 INTRODUCTION

Multimodal large language models (MLLMs) combine the reasoning, instruction-following, and generative capabilities of LLMs with modality-specific components, providing a unified framework for processing multimodal inputs and generating multimodal outputs. They underpin a broad range of applications, including visual understanding (Liu et al., 2023b; Li et al., 2023; Chen et al., 2023) and generation (Chameleon Team, 2024), audio understanding (Chu et al., 2024), embodied AI (Driess et al., 2023; Brohan et al., 2023), and interactive multimodal assistants (OpenAI, 2023; Alayrac et al., 2022).

Figure 1 illustrates a representative multimodal LLM architecture (OpenAI, 2023; Wu et al., 2025), consisting of a modality encoder, an LLM backbone, and a modality generator connected by projectors such as MLPs or cross-attention modules. The encoder maps multimodal inputs into modality tokens, which are organized into a token sequence and processed by the LLM; the generator maps the LLM outputs back to the target modality. Training such architectures is challenging due to both model and data heterogeneity, as identified by DistTrain (Zhang et al., 2025). Model heterogeneity arises from the structural and computational differences among the encoder, LLM backbone, and generator, which exhibit distinct execution patterns. Data heterogeneity further complicates training:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

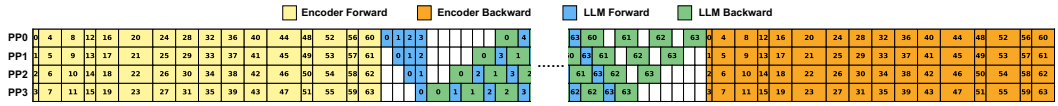


Figure 2: The pipeline structure of compute-efficient MLLM training (pp size = 4, batch size = 64).

even with a fixed sequence length, multimodal samples may contain different numbers of modality tokens, causing workload imbalance and inefficient resource utilization.

Existing systems address these challenges by redesigning the training pipeline, but remain constrained by a Pareto frontier between compute and memory efficiency. Compute-efficient systems, such as LoongCat (Team et al., 2025) and Optimus (Feng et al., 2025), improve throughput by decoupling encoder computation from the LLM pipeline and executing encoder outside the LLM pipeline. While this eliminates computational interference between the encoder and LLM, it requires retaining all encoder activations until the LLM pipeline finishes, resulting in $O(M/P)$ encoder activation memory, where M is the number of microbatches and P is the pipeline parallelism degree. With a generator, the problem is further amplified because LLM activations must also be preserved until generator computation completes. In contrast, memory-efficient systems, such as DistTrain (Zhang et al., 2025) and Megatron (Shoeybi et al., 2019; Narayanan et al., 2021), integrate the encoder and generator into the LLM pipeline as its initial and final layers. This reduces the activation memory complexity of the encoder and generator to $O(P)$ and $O(1)$, respectively. However, sequentially executing all components in one pipeline introduces computational interference, making the iteration time dictated by the slowest stage. Consequently, these systems achieve substantially lower compute efficiency than compute-efficient alternatives.

To this end, we present BIGMAC, a new training framework for multimodal LLMs. Its key idea is a dependency-safe nested pipeline that interleaves the encoder and generator with the original LLM pipeline without altering its dependencies. This design matches the compute efficiency of an idealized compute-efficient setting with unlimited memory, while reducing encoder and generator activation memory to $O(1)$. As a result, BIGMAC breaks the Pareto frontier between compute efficiency and memory usage, enabling both high compute efficiency and low memory usage in MLLM training. In our evaluation, BIGMAC reduces training iteration time by up to 1.9× over baseline systems, while keeping peak memory stable as batch size increases.

2 BACKGROUND AND RELATED WORK

2.1 BACKGROUND

Multimodal LLMs. As shown in Figure 1, a representative MLLM consists of modality encoders, an LLM backbone, and modality generators (OpenAI, 2023; Wu et al., 2025). The encoders convert raw modality inputs into intermediate features, e.g., using ViT (Dosovitskiy et al., 2021; Radford et al., 2021) for images or Whisper (Radford et al., 2023) for audio, which are then projected into the LLM embedding space through MLP or cross-attention projectors. The LLM backbone reasons over the resulting embeddings, and output projectors map its representations to modality generators, such as diffusion (Rombach et al., 2022) for images or AudioLDM (Liu et al., 2023a) for audio.

Model heterogeneity. MLLMs exhibit substantial model heterogeneity because their constituent modules differ in architecture and computational characteristics. Modality encoders such as ViT typically use relatively narrow transformer (Vaswani et al., 2017) layers, the LLM backbone uses much wider transformer layers, and modality generators such as diffusion models combine convolution and attention operators. As a result, the encoder, backbone, and generator have distinct execution times, creating imbalance in pipeline-parallel (PP) training. If an LLM PP stage is the bottleneck, the encoder and generator stages wait, reducing resource utilization. Conversely, if the encoder or generator is slower than the LLM stages, it stalls the LLM pipeline and introduces bubbles. This latter case is particularly harmful at scale, where most GPUs are allocated to the LLM backbone, leading to substantial resource underutilization.

Data heterogeneity. MLLM training data exhibit substantial data heterogeneity because modality subsequences within each sample are highly skewed in both length and count, causing large variation in the number of modality tokens (Deng et al., 2025). For example, one sample may contain only

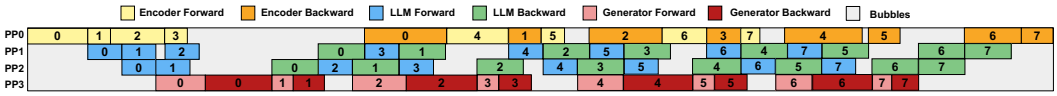


Figure 3: The pipeline structure of memory-efficient MLLM training (pp size = 4, batch size = 8).

a few tokens from a single image, while another may contain hundreds or thousands tokens from multiple high-resolution images. This heterogeneity leads to significant execution-time variation in the modality encoder and generator, even when the LLM backbone has stable execution time under a fixed sequence length. As a result, pipeline-parallel training suffers from microbatch stragglers: slower microbatches delay faster ones and introduce pipeline bubbles that reduce GPU utilization and prolong end-to-end MLLM training (Zhang et al., 2025).

2.2 Related Work

Compute-efficient systems. To address model and data heterogeneity, existing compute-efficient systems (Team et al., 2025; Feng et al., 2025; Xue et al., 2026) redesign the MLLM training pipeline to eliminate computational interference among the encoder, generator, and LLM backbone. They typically apply different parallelization strategies to different modules: data parallelism for the encoder and generator, and pipeline parallelism for the LLM backbone. As shown in Figure 2, these systems first execute the encoder forward pass for all microbatches and retain the resulting activations for backward computation. After the encoder forward finishes, the LLM pipeline processes the modality embeddings using a 1F1B schedule (Huang et al., 2019; Narayanan et al., 2019). By separating encoder and LLM computation along the time dimension, this design avoids cross-module interference. Thus, differences in encoder, generator, and LLM execution times do not introduce pipeline bubbles. However, this compute efficiency comes at the cost of high memory usage. All encoder activations must be retained until the LLM pipeline completes, and the problem becomes more severe when a generator is included. In that case, LLM activations must also be preserved until generation finishes. Since LLM activations are substantially larger than encoder activations, this approach becomes impractical for production-scale MLLM training with modality generators.

Memory-efficient systems. Memory-efficient systems (Zhang et al., 2025; Shoeybi et al., 2019; Narayanan et al., 2021) take the opposite approach by integrating the modality encoder and generator into the LLM pipeline as the initial and final layers, respectively. As shown in Figure 3, they execute the encoder, LLM backbone, and generator sequentially in a single 1F1B pipeline. This design reduces the activation memory complexity of the encoder and generator to $O(P)$ and $O(1)$, respectively, where P is the pipeline-parallelism degree. However, this memory efficiency comes at the cost of compute efficiency. While LLM microbatches have relatively stable execution time under a fixed sequence length, encoder and generator microbatches can vary substantially due to model and data heterogeneity. Because all components are coupled by sequential pipeline dependencies, this variation introduces bubbles across the pipeline, as shown in Figure 3. Although automatic parallelism tuning (Zheng et al., 2022) and dynamic microbatch scheduling (Zhang et al., 2025) can partially mitigate the issue, they still fall short of the compute efficiency achieved by compute-efficient systems.

Performance analysis. The two baselines above expose a fundamental memory–compute tradeoff. Let M be the number of microbatches and P the LLM pipeline-parallelism degree, and assume that one encoder/generator microbatch has activation footprint A_m while one full LLM microbatch has footprint A_l. Compute-efficient systems decouple encoder and generator computation from the LLM pipeline, so the LLM only incurs the standard 1F1B fill/drain overhead, with bubble rate approximately $(P-1)/(M+P-1)$, which becomes small when $M \gg P$. However, this low bubble rate comes from extending activation lifetime: on the first pipeline stage, peak activation memory is approximately $\frac{M}{P}A_m + A_l$, because modality activations for all assigned microbatches must be retained while the LLM pipeline executes. Memory-efficient systems instead integrate the encoder and generator into the pipeline, bounding peak modality activation memory by PA_m on the first PP stage or $\frac{P-1}{P}A_l$ on the second PP stage, thereby removing the memory term that grows with M. However, this design introduces additional steady-state bubbles beyond 1F1B fill/drain overhead, because encoder and generator microbatches may have different execution times. Thus, when M is large, compute-efficient systems achieve low bubble rates at high activation-memory cost, while memory-efficient systems reduce memory by sacrificing compute efficiency.

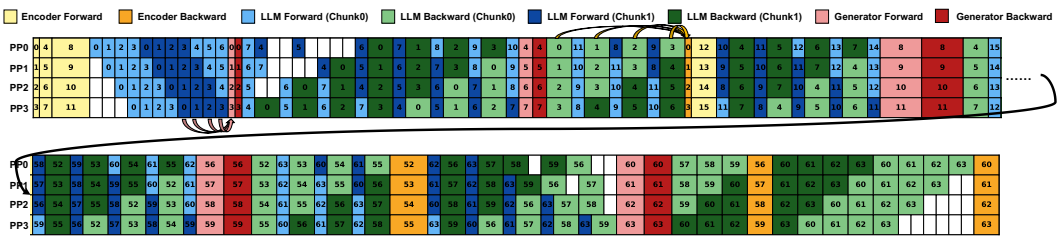


Figure 4: The pipeline scheme of BIGMAC (pp size = 4, vpp size = 2, batch size = 64).

3 BIGMAC DESIGN

BIGMAC has two components: the pipeline scheduler, which determines how microbatches from different modules are scheduled, and the pipeline executor, which runs the resulting schedule on the underlying training framework.

3.1 BIGMAC PIPELINE SCHEDULER

The scheduler orchestrates compute and communication operators by **dependency-safe nesting**.

Compute operator. Figure 4 shows the overall pipeline scheme. The scheduler targets two goals: preserving the compute efficiency of the LLM pipeline and bounding encoder/generator activation memory. To avoid computational interference, BIGMAC treats the encoder, LLM, and generator as separate parallelism domains rather than a single stage-by-stage pipeline. This matches production MLLM training, where the LLM backbone usually requires pipeline parallelism, while smaller encoders and generators can run with DDP (Li et al., 2020) or FSDP (Zhao et al., 2023). Thus, within one pipeline group, an LLM micro-batch traverses all pipeline ranks, whereas encoder and generator micro-batches are processed locally.

Compute efficiency. BIGMAC groups encoder and generator operators into units of pp_size micro-batches, where pp_size is the LLM pipeline parallelism size. Each unit corresponds to one LLM pipeline column in Figure 4. Production systems adopt a wide range of LLM schedules, including GPipe (Huang et al., 2019), 1F1B (Narayanan et al., 2019), interleaved 1F1B (Narayanan et al., 2021), bidirectional pipeline (Fan et al., 2021; Li & Hoefler, 2021), and zero-bubble (Qi et al., 2023) variants. BIGMAC therefore starts from the user-specified LLM schedule and nests encoder and generator units within it, rather than prescribing a schedule of its own. We analyze the widely used interleaved 1F1B case below. Since encoder and generator units are not part of the LLM pipeline stages, BIGMAC can nest encoder/generator units into the LLM schedule without introducing bubbles. Furthermore, this nested structure preserves the original LLM dependencies while eliminating cross-module execution interference.

Memory efficiency. For memory efficiency, BIGMAC executes encoder and generator backward operators as soon as their dependencies are satisfied. This requires respecting cross-module edges: LLM forward consumes encoder outputs, and LLM backward depends on gradients produced through generator backward. BIGMAC therefore places each encoder and generator unit at the earliest dependency-safe point without changing the LLM order. It first warms up a constant number (W) of encoder forward units, then pairs each ready encoder backward unit with the next encoder forward unit. For the generator, it runs forward as soon as LLM outputs are ready and backward immediately afterward. This policy preserves all cross-module dependencies, keeps the LLM schedule unchanged, and requires only $W = 3$ encoder activations and one generator activation. Figure 5 shows the pseudocode of the scheduling algorithm.

Analysis. For interleaved 1F1B schedules with $vpp_size \geq 2$, BIGMAC sets $W = 3$, because such schedules require only a three-unit encoder lookahead. Let U_i denote the i-th encoder unit, which contains pp_size consecutive micro-batches. Let F_i denote the first entry-side LLM forward operator that consumes the encoder output of U_i, and let G_i denote the point at which the LLM produces all the input gradient required by U_i's backward. At the unit level, interleaved 1F1B with $vpp_size \geq 2$ satisfies the following order property:

$$F_i,\ F_{i+1},\ F_{i+2} \prec G_i \prec F_{i+3}.$$

```python
# M: micro-batches; P: pipeline size
# W: warmup number of encoder units
def build_schedule(llm_schedule, M, P, W):
    units = partition_microbatches(M, unit_size=
      P)
    schedule, next_unit = [], 0

    # Warm up encoder units.
    while next_unit < min(W, len(units)):
        schedule += [enc_fwd(units[next_unit])]
        next_unit += 1

    # Preserve the original LLM pipeline order.
    for column in columns(llm_schedule):
        schedule += column

        for op in column:
            u = unit(op)
            if llm_output_ready(op, u):
                schedule += [gen_fwd(u), gen_bwd
      (u)]

            if encoder_grad_ready(op, u):
                schedule += [enc_bwd(u)]
                if next_unit < len(units):
                    schedule += [enc_fwd(units[
      next_unit])]
                    next_unit += 1

    return schedule
```

Figure 5: BUILDSCHEDULE nests encoder/generator units into the LLM pipeline schedule.

```python
def encoder_forward(data_iter):
    input_ids, encoder_inputs = next(data_iter)
    encoder_feature = encoder(encoder_inputs)
    return input_ids, encoder_feature

def llm_forward_step(llm_iter):
    input_ids, encoder_feature = next(llm_iter)
    embedding = embed_preprocess(input_ids,
                                 encoder_feature
      )
    output = llm(embedding)
    return output

def generator_forward(output):
    loss = generator(output)
    return loss

def train_step(data_iter, pp_size,
               microbatch_num, warmup_bound):
    # E.g., interleaved 1F1B schedule
    llm_schedule = get_llm_schedule(pp_size,

      microbatch_num)
    # BigMac pipeline schedule
    schedule = build_schedule(llm_schedule,
                              microbatch_num,
                              pp_size,
                              warmup_bound)
    schedule = insert_comm_ops(schedule)
    schedule = deadlock_check(schedule)
    executor = create_executor(data_iter,
                               llm_forward_step,
                               encoder_forward,
                               generator_forward
      )
    executor.load_schedule(schedule)
    executor.execute()
```

Figure 6: BIGMAC pipeline executor interface.

After the LLM finishes consuming the output of U_i, it consumes the outputs of U_{i+1} and U_{i+2} before the gradient for U_i returns, and this gradient becomes ready before the LLM needs the output of U_{i+3}. This property follows from the steady phase of 1F1B: each new forward wave is matched by the backward wave of an older unit, so the distance between forward consumption and backward return is bounded by three encoder units. Therefore, before F_{i+3} is reached, BIGMAC can execute the backward of U_i to release the U_i's activations, and then execute the forward of U_{i+3} to prepare the next unit. As a result, the ready window slides from U_i, U_{i+1}, U_{i+2} to $U_{i+1}, U_{i+2}, U_{i+3}$ before the LLM consumes U_{i+3}. This invariant ensures that only three encoder-unit activations need to be kept in memory to keep the LLM pipeline fully fed, yielding $O(1)$ encoder activation memory. By induction on i, encoder outputs are always ready when needed, so encoder units neither stall nor reorder the LLM pipeline, allowing BIGMAC to match the compute-efficient pipeline time.

Communication operator. BIGMAC makes the compute schedule executable by materializing its implicit data dependencies as communication operators. For each compute operator, BIGMAC derives a pre-communication operator that fetches required inputs and a post-communication operator that forwards produced tensors to consumers. This preserves the pipeline compute order in Figure 5 while exposing all cross-device activation and gradient transfers to the executor.

Pipeline communication. For the LLM backbone, BIGMAC uses standard point-to-point pipeline communication. Stage i receives forward activations from stage $i-1$ and sends outputs to stage $i+1$, with symmetric communication in the backward pass. Under interleaved 1F1B, each communication operator carries the micro-batch and virtual chunk metadata of its compute operator, allowing the executor to distinguish tensors across chunks and micro-batches.

Cross-module communication. Cross-module communication connects separately scheduled encoder/generator units with the LLM pipeline. Encoder forward gathers modality embeddings to the LLM entry stage, while LLM backward scatters input-embedding gradients back to the owning encoder ranks. The generator handoff is analogous on the LLM output side: the last LLM stage scatters generator inputs and gathers generator-input gradients back.

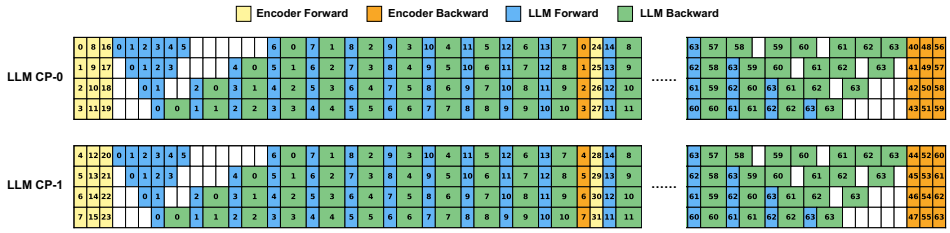


Figure 7: The pipeline scheme of BIGMAC with context parallel (LLM CP = 2, Encoder CP = 1).

3.2 BIGMAC PIPELINE EXECUTOR

BIGMAC's pipeline executor turns the operator schedule produced by the scheduler into concrete training execution. Each pipeline rank receives its local schedule, which contains both compute and communication operators. The executor interprets this schedule as an opcode stream: LLM forward and backward operators are dispatched to user-provided LLM runtime, point-to-point communication operators are dispatched to the runtime communication backend, and encoder/generator operators are dispatched to user-provided module functions. This design keeps the scheduler independent of framework-specific execution details, while allowing the executor to reuse optimized runtime components such as fused send-receive primitives and DP gradient overlap.

Figure 6 shows the executor's programming interface: the user supplies only the forward implementations of the encoder, LLM, and generator, and the executor hides all parallelism and communication details.

Runtime buffers. The executor maintains runtime buffers keyed by microbatch id, with an additional virtual chunk id under interleaved 1F1B. Forward operators write output activations to these buffers, backward operators consume the corresponding activations and produce gradients. Communication operators transfer tensors in the buffer across adjacent pipeline stages. The executor issues the communication operations and records the returned asynchronous communication handles. The executor waits on receive handles before consuming received tensors. Thus, the executor preserves the scheduler-determined operator order while overlapping communication with computation.

Module handoff. The executor bridges encoder/generator modules and the LLM through dedicated handoff adapters. For encoder-to-LLM handoff, it validates encoder outputs using explicit metadata, and then gathers them into the LLM entry-stage. The communication is launched asynchronously with readiness events so encoder or LLM computation can overlap with gather communication. The backward path reverses this route: the first LLM stage retains input tensors, collects their gradients after LLM backward, scatters them to the owning encoder ranks, immediately runs encoder backward once a unit's gradient is ready, and releases saved encoder activations. For distributed data-parallel encoder/generator modules, the executor suppresses non-final gradient synchronization and finalizes gradients at schedule end to preserve standard accumulation semantics. Generator handoff follows the same pattern on the LLM output side: the last LLM stage scatters output activations to generator ranks, runs generator forward and backward for the unit, and gathers the resulting input gradients to trigger LLM backward.

3.3 BIGMAC COMPATIBILITY

Context parallelism. Context parallelism (CP) (Liu et al., 2024) is a standard technique for training LLMs with long input contexts. CP partitions the input sequence across multiple ranks and parallelizes attention computation. In MLLM training, CP may be applied simultaneously to the LLM backbone, encoder, and generator, with different CP degrees for different modules due to their differing sequence lengths and attention costs. BIGMAC supports this multi-module CP setting by allowing the encoder and LLM to use distinct CP groups and by extending the module handoff and pipeline scheduler accordingly.

When the encoder and LLM use different CP groups, the encoder-to-LLM handoff converts modality embeddings from the encoder CP layout to the LLM CP layout through an additional all-to-all communication. BIGMAC represents this conversion as an asynchronous CP-conversion operator between encoder operators and LLM operators, using readiness events to overlap communication

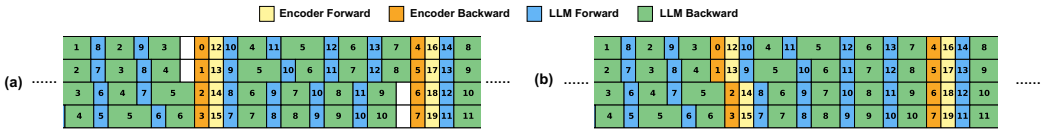


Figure 8: A pipeline slice of BIGMAC with an encoder trained using FSDP.

with computation. Different CP degrees also constrain pipeline scheduling: pipeline groups within the same LLM CP group consume the same encoder outputs in lockstep because context parallel synchronizes LLM ranks after each micro-batch. Thus, BIGMAC enlarges the encoder scheduling unit from pp_size to $pp_size \times llm_cp_size/encoder_cp_size$ micro-batches. In Figure 7, for example, LLM CP size two and encoder CP size one yield an eight micro-batch encoder unit while preserving the original LLM pipeline order.

Fully sharded data parallelism. FSDP shards model parameters, gradients, and optimizer states across GPUs (Rajbhandari et al., 2020), so each rank holds only a parameter shard during training. Before executing an FSDP-wrapped module in either the forward or backward pass, all ranks in the FSDP group must first `all-gather` the full parameters. Each FSDP module therefore introduces a group-wide synchronization point, which may interact poorly with BIGMAC's nested pipeline scheme.

As shown in Figure 8(a), LLM operator time varies across micro-batches. For example, the backward operator of micro-batch 5 takes longer than the others. This variation comes from sequence packing, which creates different attention workloads across micro-batches (Ge et al., 2025), and MoE expert parallelism (Lepikhin et al., 2021; Fedus et al., 2022), where expert load imbalance creates uneven GPU workloads (DeepSeek-AI, 2025). Under such non-uniform execution, FSDP's strict synchronization introduces additional pipeline bubbles in the steady phase.

One-Sided pull. To solve this issue, BIGMAC replaces the collective communication, i.e., `all-gather` of FSDP-wrapped encoder and generator with a request-driven one-sided pull mechanism. Instead of requiring all ranks in an FSDP group to enter the same `all-gather` at the same time, each rank publishes its local parameter shard in NVSHMEM-accessible memory. When a rank reaches a FSDP-wrapped encoder or generator operator, it materializes the required full parameter bucket by fetching the missing shards from their owner ranks using one-sided NVSHMEM `get` operations. Thus, BIGMAC removes the FSDP synchronization barrier, allowing ranks to proceed independently according to the pipeline schedule. As shown in Figure 8(b), this design keeps FSDP's sharded-memory benefit while removing the all-rank `all-gather` barrier that creates the pipeline bubbles.

4 EVALUATION

4.1 EXPERIMENTAL SETUP

Testbed. We evaluate BIGMAC on a 16-server cluster with 128 NVIDIA Hopper GPUs. Each server contains eight GPUs, 160 Intel Xeon CPU cores, and 1.8 TB of host memory. Intra-server GPU communication uses 900 GB/s NVLink, while inter-server communication uses 400 Gb/s NVIDIA Mellanox InfiniBand with GPUDirect RDMA. The software stack uses PyTorch 2.8.0 and CUDA 12.2 with driver 535.161.08.

Models & Datasets. We evaluate BIGMAC on two representative MLLM training workloads: MLLM-Understanding and MLLM-Generation. For MLLM-Understanding, we use Qwen3-30B-A3B (Qwen Team, 2025) as the backbone LLM and a 1.3B-parameter ViT (Dosovitskiy et al., 2021) as the vision encoder. The model is trained on image-text pairs from a large-scale web-crawled dataset to learn visual-textual understanding and reasoning. For MLLM-Generation, we use the same backbone LLM and vision encoder, and attach a 20B-parameter MMDiT (Esser et al., 2024) as the multimodal generator. This follows the model architecture in Qwen-Image (Wu et al., 2025). The model is trained on internal text-to-image (T2I), text-image-to-image (TI2I), and image-to-image (I2I) pairs to generate images from textual and visual inputs. Both workloads use an 8K input sequence length for the LLM backbone. All modules remain trainable.

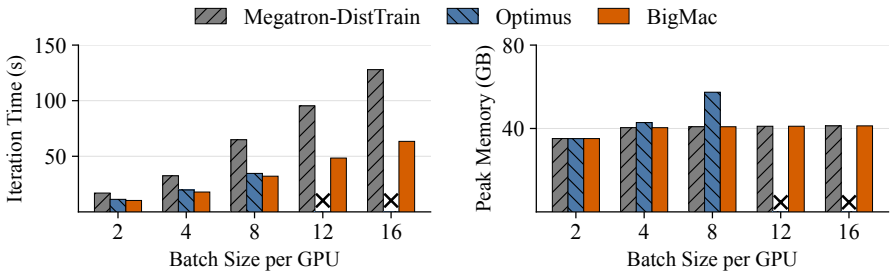


Figure 9: Overall performance of BIGMAC under training MLLM-Understanding.

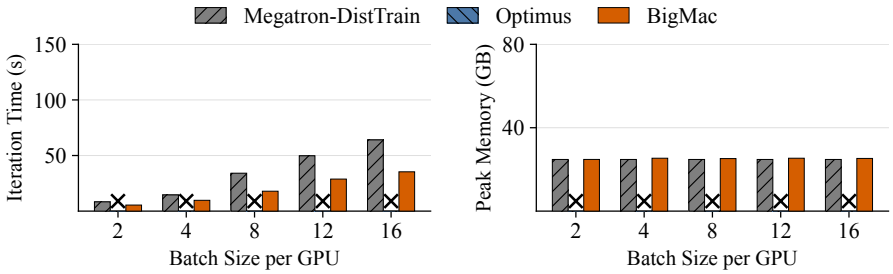


Figure 10: Overall performance of BIGMAC under training MLLM-Generation.

Baselines & Metrics. We compare BIGMAC with two representative MLLM training baselines. The first is Optimus (Feng et al., 2025), a compute-efficient design that runs encoder forward, LLM forward and backward, and generator backward sequentially. The second is Megatron-DistTrain (Zhang et al., 2025), a memory-efficient design that integrates the encoder and generator as the first and last stages of the pipeline. We call it Megatron-DistTrain since we use Megatron's open-source implementation of DistTrain. These two baseline systems are widely adopted in production MLLM training (Team et al., 2025; Huang et al., 2025; Jiang et al., 2024). We report training iteration time and peak GPU memory usage, the two primary metrics for MLLM training efficiency and scalability.

4.2 OVERALL PERFORMANCE

MLLM-Understanding. We first evaluate BIGMAC on the MLLM-Understanding workload. Figure 9 reports iteration time and peak GPU memory usage under different per-GPU batch sizes. BIGMAC consistently achieves the lowest iteration time, improving over Optimus by 1.08×–1.1× and over Megatron-DistTrain by 1.6×–1.9×. Compared with Megatron-DistTrain, BIGMAC removes the execution interference between the encoder and LLM, thereby reducing pipeline bubbles. Compared with Optimus, BIGMAC keeps activation memory low, reducing memory pressure and avoiding the expensive memory operations. For peak memory, BIGMAC and Megatron-DistTrain maintain a stable memory usage when batch size increases, whereas Optimus's memory usage grows rapidly because it retains encoder activations across the LLM pipeline. When the per-GPU batch size exceeds 8, Optimus runs out of memory. Cross markers denote these out-of-memory points.

MLLM-Generation. We then evaluate BIGMAC on the MLLM-Generation workload as shown in Figure 10. Optimus runs out of memory at all batch sizes since it needs to retain all LLM activations until the generator finishes. Compare with Megatron-DistTrain, BIGMAC reduces iteration time by 1.5×–1.9× by eliminating the execution interference between the encoder, LLM, and generator. For peak memory, BIGMAC and Megatron-DistTrain maintain a stable memory usage when batch size increases.

4.3 CASE STUDY

Context parallelism. We evaluate whether BIGMAC can benefit from decoupling context parallelism (CP) across MLLM components. The homogeneous CP baseline applies the same CP degree to both the encoder and the LLM backbone, which is suboptimal because the two modules have different model architectures and process different tokens. BIGMAC instead allows the encoder

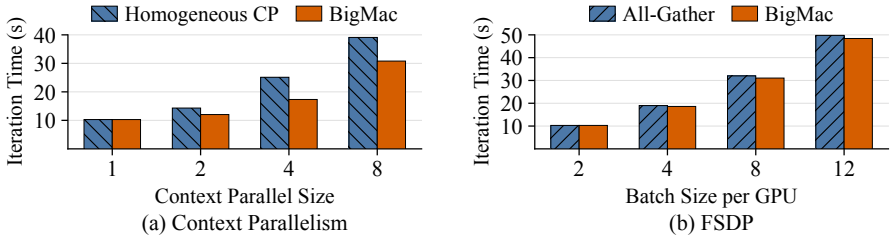


Figure 11: Case study of BIGMAC's compatibility.

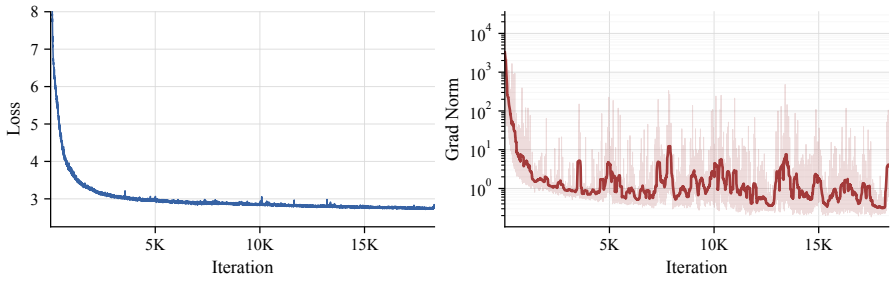


Figure 12: Production deployment metrics of BIGMAC.

and LLM to use separate CP groups. Figure 11(a) compares BIGMAC with homogeneous CP as the LLM CP degree increases. BIGMAC reduces iteration time by up to 1.45×, demonstrating the effectiveness of decoupled CP for MLLM training.

FSDP. We next evaluate BIGMAC's compatibility with FSDP-trained modality modules. The baseline uses standard FSDP `all-gather` communication to materialize full parameters before each FSDP-wrapped encoder or generator operator. BIGMAC replaces this collective synchronization with one-sided pull, allowing each rank to proceed according to its local pipeline schedule. This reduces training iteration time by up to 1.03× while preserving FSDP's sharded-parameter memory benefit.

Bitwise alignment. BIGMAC changes only the order in which module gradients are accumulated, preserving the synchronous training semantics of the original pipeline. To isolate pipeline effects from kernel-level nondeterminism, we disable FlashAttention (Dao et al., 2022) and use deterministic operators, then run BIGMAC and Megatron-LM on the same data stream under different per-GPU batch sizes. Across the first 128 iterations, the average absolute loss difference stays below 10^{-3}, confirming that BIGMAC preserves training semantics.

4.4 PRODUCTION DEPLOYMENT

BIGMAC has been deployed in production MLLM training for several months. We present a representative deployment result from an internal 345B-parameter MLLM training run on 1,536 Hopper GPUs. Figure 12 reports the loss and gradient norm over more than 18K training iterations. The loss decreases rapidly in the early phase and then stabilizes at a low value, showing that BIGMAC preserves the expected optimization behavior of the training job. The gradient norm shows transient spikes during the initial high-loss phase, but quickly settles and remains bounded throughout the rest of training. These results indicate that BIGMAC maintains stable training dynamics in production, while enabling the compute-memory efficient execution described in § 3.

5 CONCLUSION

BIGMAC addresses the compute–memory tradeoff in MLLM training with a dependency-safe nested pipeline. It preserves the user-specified LLM schedule and nests encoder/generator units only when their dependencies are satisfied, eliminating cross-module interference while keeping modality encoder and generator's activation memory $O(1)$. Across MLLM-understanding and generation workloads, BIGMAC achieves 1.08×–1.9× speedups while maintaining stable memory usage as batch size increases.

AI USE STATEMENT

In this work, we used generative AI tools to polish the wording and grammar of text that the authors had already drafted, and to assist with routine coding tasks such as plotting scripts for the figures in this paper. We did not use generative AI tools for research ideation, for deriving or verifying the scheduling algorithm and its correctness argument, for designing or running the experiments, or for producing any of the reported measurements. Literature search, related-work analysis, and all technical claims were carried out by the authors. All AI-assisted text and code were reviewed by at least two authors, and every experimental number in the paper was produced by our own runs and checked against the raw logs. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob L. Menick, Sebastian Borgeaud, Andy Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karén Simonyan. Flamingo: a visual language model for few-shot learning. In *Advances in Neural Information Processing Systems*, 2022.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, Pete Florence, Chuyuan Fu, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Kehang Han, Karol Hausman, Alexander Herzog, Jasmine Hsu, Brian Ichter, Alex Irpan, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Kuang-Huei Lee, Sergey Levine, Yao Lu, Henryk Michalewski, Igor Mordatch, Karl Pertsch, Kanishka Rao, Krista Reymann, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Pierre Sermanet, Jaspiar Singh, Anikait Singh, Radu Soricut, Huong Tran, Vincent Vanhoucke, Quan Vuong, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Jialin Wu, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. RT-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

Chameleon Team. Chameleon: Mixed-modal early-fusion foundation models. *arXiv preprint arXiv:2405.09818*, 2024.

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. *arXiv preprint arXiv:2312.14238*, 2023.

Yunfei Chu, Jin Xu, Qian Yang, Haojie Wei, Xipin Wei, Zhifang Guo, Yichong Leng, Yuanjun Lv, Jinzheng He, Junyang Lin, Chang Zhou, and Jingren Zhou. Qwen2-audio technical report. *arXiv preprint arXiv:2407.10759*, 2024.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

DeepSeek-AI. EPLB: Expert parallelism load balancer. `https://github.com/deepseek-ai/eplb`, 2025.

Chaorui Deng, Deyao Zhu, Kunchang Li, Chenhui Gou, Feng Li, Zeyu Wang, Shu Zhong, Weihao Yu, Xiaonan Nie, Ziang Song, et al. Emerging properties in unified multimodal pretraining. *arXiv preprint arXiv:2505.14683*, 2025.

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

Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. Palm-e: An embodied multimodal language model. In *International Conference on Machine Learning (ICML)*, 2023.

Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, Kyle Lacey, Alex Goodwin, Yannik Marek, and Robin Rombach. Scaling rectified flow transformers for high-resolution image synthesis. In *International Conference on Machine Learning (ICML)*, 2024.

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

William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23, 2022.

Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. Optimus: Accelerating large-scale multi-modal llm training by bubble exploitation. In *USENIX ATC*, 2025.

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

Ailin Huang, Boyong Wu, Bruce Wang, Chao Yan, Chen Hu, Chengli Feng, Fei Tian, Feiyu Shen, Jingbei Li, Mingrui Chen, et al. Step-audio: Unified understanding and generation in intelligent speech interaction. *arXiv preprint arXiv:2502.11946*, 2025.

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. In *Advances in Neural Information Processing Systems*, 2019.

Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *USENIX NSDI*, 2024.

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. In *International Conference on Learning Representations (ICLR)*, 2021.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven C. H. Hoi. BLIP-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International Conference on Machine Learning (ICML)*, 2023.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 2020.

Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *ACM/IEEE Supercomputing (SC)*, 2021.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *International Conference on Learning Representations*, 2024.

Haohe Liu, Zehua Chen, Yi Yuan, Xinhao Mei, Xubo Liu, Danilo Mandic, Wenwu Wang, and Mark D. Plumbley. AudioLDM: Text-to-audio generation with latent diffusion models. In *International Conference on Machine Learning (ICML)*, 2023a.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In *Advances in Neural Information Processing Systems*, 2023b.

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 *ACM SOSP*, 2019.

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 *ACM/IEEE Supercomputing (SC)*, 2021.

OpenAI. GPT-4V(ision) system card, 2023. `https://openai.com/index/gpt-4v-system-card/`.

Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241*, 2023.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning (ICML)*, 2021.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning (ICML)*, 2023.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *ACM/IEEE Supercomputing (SC)*, 2020.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2022.

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.

Meituan LongCat Team, Bairui Wang, Bin Xiao, Bo Zhang, Bolin Rong, Borun Chen, Chang Wan, Chao Zhang, Chen Huang, Chen Chen, et al. Longcat-flash-omni technical report. *arXiv preprint arXiv:2511.00279*, 2025.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Chenfei Wu, Jiahao Li, Jingren Zhou, Junyang Lin, Kaiyuan Gao, Kun Yan, Sheng-ming Yin, Shuai Bai, Xiao Xu, Yilei Chen, et al. Qwen-image technical report. *arXiv preprint arXiv:2508.02324*, 2025.

Chunyu Xue, Yangrui Chen, Jianyu Jiang, Ningxin Zheng, Junda Feng, Jingji Chen, Shixiong Zhao, Shen Yan, Yi Lin, Lei Shi, et al. Megascale-omni: A hyper-scale, workload-resilient system for multimodal llm training in production. In *EuroSys*, 2026.

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

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. *Proceedings of the VLDB Endowment*, 2023.

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 *USENIX OSDI*, 2022.