Scaling Llama 3 Training with Efficient Parallelism Strategies

Weiwei Chu†, Xinfeng Xie†, Jiecao Yu†, Jie Wang†, Amar Phanishayee, Chunqiang Tang, Yuchen Hao, Jianyu Huang, Mustafa Ozdal, Jun Wang, Vedanuj Goswami, Naman Goyal, Abhishek Kadian, Andrew Gu, Chris Cai, Feng Tian, Xiaodong Wang, Min Si, Pavan Balaji, Ching-Hsiang Chu, and Jongsoo Park

Meta Platforms, Inc.

Abstract

Llama is a widely used open-source large language model. This paper presents the design and implementation of the parallelism techniques used in Llama 3 pre-training. To achieve efficient training on tens of thousands of GPUs, Llama 3 employs a combination of four-dimensional parallelism: fully sharded data parallelism, tensor parallelism, pipeline parallelism, and context parallelism. Beyond achieving efficiency through parallelism and model co-design, we also address other equally critical aspects. First, we enhance *flexibility*—for example, through novel pipeline parallelism that supports evolving batch sizes and heterogeneous model architectures, and innovative context parallelism that enables model innovations such as document-mask attention. Second, we prioritize *practicality*—for example, by enabling the diagnosis of performance and numerical issues at scale. Finally, drawing on our experience with large-scale training, we provide recommendations for future hardware design.

CCS Concepts

• **Computing methodologies** → **Distributed computing methodologies**; • **Computer systems organization** → *Distributed architectures.*

Keywords

Large Language Model, Training, Parallelism, Distributed System

ACM Reference Format:

Weiwei Chu†, Xinfeng Xie†, Jiecao Yu†, Jie Wang†, Amar Phanishayee, Chunqiang Tang, Yuchen Hao, Jianyu Huang, Mustafa Ozdal, Jun Wang, Vedanuj Goswami, Naman Goyal, Abhishek Kadian, Andrew Gu, Chris Cai, Feng Tian, Xiaodong Wang, Min Si, Pavan Balaji, Ching-Hsiang Chu, and Jongsoo Park. 2025. Scaling Llama 3 Training with Efficient Parallelism Strategies. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731410

1 Introduction

Large language models (LLMs) have revolutionized natural language processing (NLP) by exhibiting remarkable capabilities across a wide range of tasks, including conversational agents, language translation, and code generation [3, 18, 28]. Expanding beyond text-based applications, multimodal models further extend these capabilities by enabling the understanding and generation of content across diverse modalities such as audio, images, and video [2, 4, 19, 36, 40].

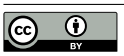
Llama [10, 37, 38] is a widely adopted open-source LLM, significantly influencing both industry and research. Llama 3, released on April 18, 2024 [23], features its largest model with 405 billion parameters, pre-trained on 16,384 H100 GPUs over several months, utilizing a total of 3.8×10^{25} FLOPs [10]. Training at this hyperscale necessitates efficient parallelism strategies that distribute the model and schedule computation and communication across GPUs. Consequently, Llama 3 was trained using a combination of four-dimensional (4D) parallelism techniques: fully sharded data parallelism (FSDP) [30, 31, 44], tensor parallellism (TP) [14, 15, 33], pipeline parallellism (PP) [11, 16, 25], and context parallellism (CP) [20].

Each of these parallelism techniques presents its own performance trade-offs, and their combination creates a complex design space that requires careful exploration to achieve optimal performance. While numerous prior works have tackled this challenge from different angles [9, 12, 25, 29, 35, 36, 45], our production experience in training Llama at hyperscale highlights the importance of *flexibility* to dynamically adjust configurations and *practicality* for debugging performance and numerical issues. We next elaborate on the challenges in efficiency, flexibility, and practicality.

Efficiency: Llama 3 pre-training is a capability computing problem, where the primary objective is to minimize total pre-training time by maximizing the utilization of 16K GPUs. This poses a unique challenge due to the limited achievable parallelism along the data batch dimension. Specifically, designing an efficient PP schedule is challenging, given the small number of micro-batches available to hide pipeline bubbles. Furthermore, reducing the pipeline stages is not a viable alternative, as it would require increasing model parallelism, particularly TP, where communication latency is fully exposed, slowing down training.

Flexibility: Llama 3 pre-training consists of multiple phases, each designed to achieve specific training objectives, such as optimizing for short context, long context, or multimodal understanding. These phases are configured with varying hyperparameters (e.g., global batch size and sequence length), heterogeneous model architectures (e.g., alternating self-attention and cross-attention layers), and differing resource allocations (e.g., the number of GPUs used). Furthermore, the implementation of document masking [10] introduces an input-dependent attention mask, leading to dynamic variations in computation patterns across different training batches. This inherent dynamism in the training workloads demands a highly flexible system capable of adapting to these diverse changes while maintaining high efficiency.

† Co-primary authors



ISCA '25, June 21–25, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731410

Practicality: Implementing and optimizing 4D parallelism across 16K GPUs poses significant practical challenges, especially in debugging. Performance debugging (i.e., identifying the root cause of performance slowdowns) is complex, as issues can propagate across the entire distributed system. The first rank where a problem is observed is often not the true source, complicating root-cause analysis. Moreover, debugging numerical issues adds another layer of complexity. For instance, determining whether a deviation in loss curves stems from an implementation error or from the accumulation of small precision differences across many parallel ranks using low-precision floating-point arithmetic is particularly challenging.

This paper details the Llama 3 training framework, which introduces new features that address efficiency, flexibility, and practicality challenges at scale. Our contributions include:

- We optimize 4D parallelism for 16K GPUs, fitting the large model into memory and overcoming batch size constraints. Moreover, we co-design parallelism with model hyperparameters (number of layers and layer types) to balance computation and memory usage in PP. For the 405B Llama 3 model on 16K GPUs, we achieve 400 TFLOPs per GPU (8K sequence length) and 380 TFLOPs per GPU (131K sequence length).
- We introduce a flexible PP schedule that supports variable global batch sizes and heterogeneous layer sharding, demonstrated in Llama 3 multimodal pre-training with balanced memory usage and high throughput. We also propose a novel all-gather-based CP solution that facilitates model innovations (e.g., document-mask attention [10]), achieving performance comparable to state-of-the-art baselines [21] and strong scalability (3.89× attention latency reduction on four GPUs compared to one GPU).
- We share our methodology and lessons in debugging performance and numerical issues at scale. Our top-down trace analysis identifies the slowest rank across parallelism dimensions and pinpoints the root cause. Our numerical debugging method isolates non-parallel implementations to rule out software bugs and to identify critical gradient buffers that require high-precision floating-point accumulations.

The rest of this paper is organized as follows: Section 2 provides background on parallelism and Llama 3 pre-training. Sections 3 and 4 detail our novel PP and CP solutions. Sections 5 and 6 describe combined parallelism configurations and debugging methodology. Section 7 presents system evaluation results, and Section 8 offers hardware recommendations based on our experience with Llama 3 pre-training. Finally, we conclude in Section 9.

2 Background

This section outlines the parallelism strategies employed in Llama 3 pre-training, followed by an overview of the text and multimodal pre-training phases.

2.1 4D Parallelism

Given the ever-increasing scale of large language models (LLMs), distributed training is essential; accordingly, we utilize a 4D parallelism approach.

Fully sharded data parallelism (FSDP): Conventional data parallelism, i.e., distributed data parallelism (DDP) [17], replicates the full model weights across workers (GPUs) and distributes data batches among them. This necessitates global gradient synchronization at the end of each training iteration. Llama 3 pre-training leverages an in-house implementation based on Pytorch's fully sharded data parallelism (FSDP) [44] which extends data parallelism by sharding model weights, gradients, and optimizer states across workers. For simplicity, we use DP and FSDP interchangeably in the remainder of this paper. Our FSDP implementation supports three sharding strategies aligned with the Zero Redundancy Optimizer (ZeRO) definitions from DeepSpeed [30] (ZeRO-1, ZeRO-2, and ZeRO-3), allowing for optional sharding of model parameters, gradients, and optimizer states.

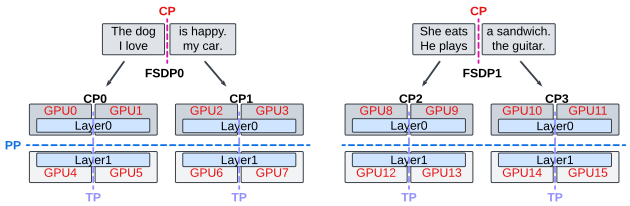


Figure 1: A two-layer LLM is sharded across 16 GPUs using 4D parallelism. FSDP and CP shard input data, with FSDP sharding along the batch size dimension and CP sharding along the sequence dimension. TP and PP shard model parameters, with TP sharding within the same layer and PP sharding across layers.

Tensor parallelism (TP): TP partitions the linear modules of the transformer layer across GPUs. Our implementation follows the approach introduced in Megatron-LM [33], splitting GEMM operators along either input or output dimensions. TP distributes computation and memory costs across GPUs but introduces additional communication overheads. Sequence parallelism (SP) is often used in conjunction with TP to further reduce activation memory costs [14]. SP shards sequence-dependent operations across TP ranks, typically involving all-gather and reduce-scatter communication around the TP-partitioned modules, which reduces memory at the cost of increased communication.

Pipeline parallelism (PP): PP divides model layers into sequential stages and distributes these stages across different PP ranks. Computation across micro-batches then proceeds in a pipelined fashion. Figure 2 illustrates an example using the interleaved 1F1B PP schedule [33]. In this configuration, each rank manages multiple *virtual stages* composed of non-consecutive model layers (e.g., rank 0 handles layers 0 and 3). The example shows 6 micro-batches processed in 2 rounds, where each virtual stage handles $nc = 3$ consecutive micro-batches per round (processing micro-batches 0-2 or 3-5). Activation memory usage on each PP rank depends on the number of warm-up micro-batches. The interleaved 1F1B schedule requires the total batch size to be a multiple of the number of PP ranks.

Context parallelism (CP): CP shards the input sequence data across GPUs. This applies directly to modules invariant to the sequence dimension (e.g., feed-forward networks). However, the

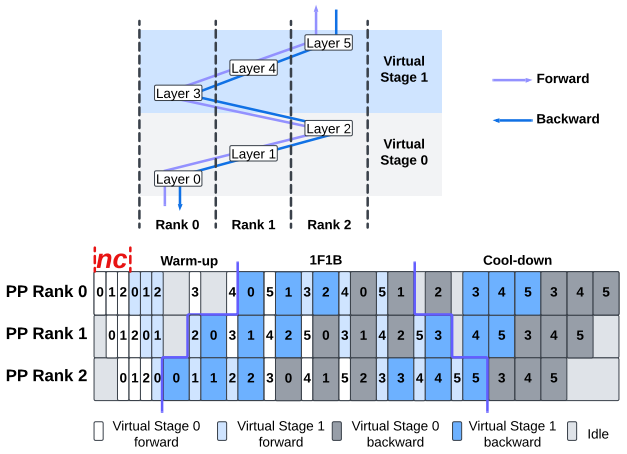


Figure 2: A 6-layer LLM is sharded across 3 PP ranks, which executes 6 micro-batches using the 1F1B PP schedule [33]. Each rank hosts two virtual stages, with each virtual stage containing a single model layer. The model layers are distributed in an interleaved manner, such that layer 0 and layer 3 are on rank 0, layer 1 and layer 4 are on rank 1, and so on. The 6 micro-batches are divided into two rounds, with each virtual stage processes *nc* consecutive micro-batches per round, where $nc = 3$ in this example.

attention mechanism requires the full sequence context, necessitating communication for reconstructing the sequence. While prior work employed ring-style communication to pass key/value tensors between adjacent ranks [21], overlapping communication and computation, we propose a straightforward all-gather-based CP approach. In our method, the communication latency (all-gather) is fully exposed. We detail this simple yet flexible and efficient approach in Section 4.

2.2 Llama 3 Pre-training Overview

Llama 3 pre-training [10] consists of three major phases: short context pre-training, long context text pre-training, and multimodal pre-training. Throughout the pre-training process, we incrementally increased the number of GPUs, global batch sizes, and sequence lengths. For multimodal pre-training, we enhanced the text model by introducing additional cross-attention layers and a trainable image encoder. The cross-attention layers take the outputs from the image encoder and the preceding transformer layer as inputs, effectively capturing the interactions between images and text. Notably, during pre-training, the original text model layers (i.e., non-cross-attention layers) remain frozen, while only the cross-attention layers and image encoder are trained.

To maximize training efficiency, we employ 3D parallelism (FSDP, TP, and PP) for short context pre-training and 4D parallelism (FSDP, TP, PP, and CP) for long context pre-training. For multimodal pre-training, we adopt a hybrid sharding strategy, where the image encoder is sharded using 2D parallelism (FSDP and TP) and the text model is sharded using 3D parallelism.

In the subsequent two sections, we present our novel PP and CP designs, which achieve high efficiency and flexibility to accommodate diverse training workloads. For clarity, we summarize all parameters and their explanations used throughout this paper in Table 1.

Table 1: Parameters and definitions used in this paper.

Parameter	Definition
$ngpu$	Number of GPUs
seq	Sequence length
gbs	Global batch size
bs	Batch size per data parallel group
mbs	Micro-batch size in pipeline stage execution
$dp/tp/cp/pp$	GPU number in one data/tensor/context/pipeline parallel group
ndp	Number of data parallel group
v	Number of virtual stages on one PP rank
ppr	The index of PP ranks
nc	Number of continuous micro-batches for a virtual stage
nmb	Number of micro-batches for each virtual stage
tmb	Sum of nmb for v virtual stages on one PP rank

3 Pipeline Parallelism

In this section, we present the design of PP and its application to multimodal training.

3.1 Design Overview

Our PP design is based on the interleaved 1F1B schedule [25]. We have made multiple optimizations and co-designed with the Llama model during pre-training to enhance efficiency and flexibility. We present the details of our optimizations as below.

3.1.1 Flexible PP schedule that supports arbitrary batch size. The original interleaved 1F1B schedule implementation constrains the batch size to be a multiple of the number of PP ranks [25]. During Llama 3 training, the global batch size is adjusted across multiple phases, necessitating a PP schedule that supports flexible batch size. We implement a flexible PP schedule that removes this constraint on the number of micro-batches.

In the 1F1B schedule, the number of warm-up micro-batches from each pipeline stage is $(v-1)\times nc+2\times(pp-ppr\times v-1)$, where v is the number of virtual stages on each PP rank, nc is the number of consecutive micro-batches per stage, pp is the pipeline size, and ppr is the pipeline rank index. The total number of micro-batches on one PP rank, tmb, is the sum of micro-batches across all virtual stages, $nmb \times v$, where nmb is the number of micro-batches. The original 1F1B schedule requires $nc == pp$ and $nmb\%nc == 0$. Note that in PP, there are certain phases when GPUs are idle, waiting for the new micro-batches or tokens from other PP ranks. We refer to these idle times as PP bubbles. The PP bubble ratio, defined as the PP idle time over the forward and backward compute time, is computed as $(pp-1)/nmb/v$ [33]. To minimize the PP bubble, we prefer a smaller pp, more micro-batches nmb and more PP virtual stages v.

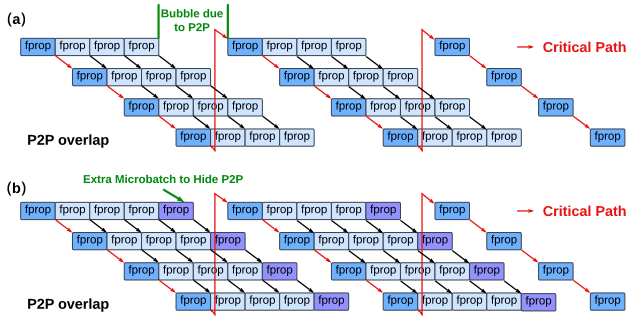


Figure 3: (a) In 1F1B schedule, bubble is introduced by exposed P2Ps (red arrows). (b) Running extra micro-batches (purple) helps reduce the bubble from exposed P2Ps.

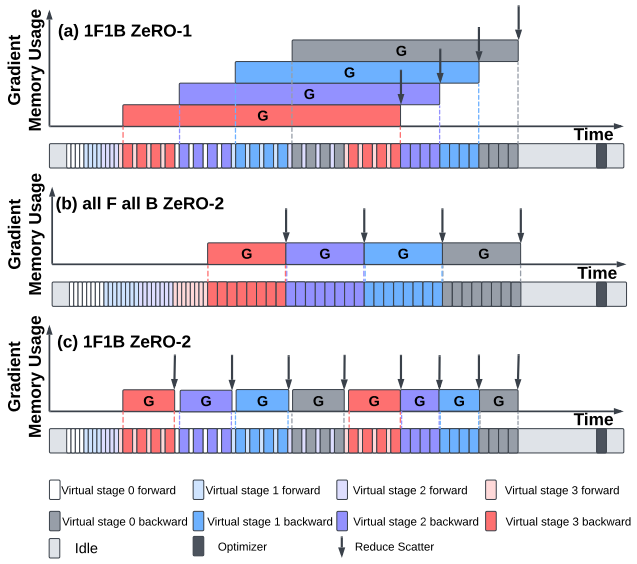


Figure 4: Gradient memory lifetime in different combinations of PP schedules and FSDP ZeRO modes: (a) 1F1B with ZeRO-1, reduce-scatter is launched only on the last micro-batch. (b) All forward all backward (same behavior between ZeRO-1/2. (c) 1F1B with ZeRO-2, reduce-scatter is launched on the last consecutive micro-batch.

Our flexible PP schedule allows nc to be set to any number between 1 and nmb, and nmb can be any required number. When nc exceeds pp, we insert $nc - pp$ more micro-batches per virtual stage into the warm-up phase. These extra micro-batches help to overlap point-to-point communication, as shown in Figure 3. However, this comes at the cost of increased peak memory usage due to $(nc - pp) \times (v - 1)$ more in-flight warm-up micro-batches compared to the original interleaved 1F1B schedule. When nc is less than pp, the schedule degenerates into an all-forward-all-backward schedule [11], where we execute the forward passes of all virtual stages before initiating the backward passes, as illustrated in Figure 4.

3.1.2 Balanced PP with model co-design. Uniformly sharding model layers across PP ranks can lead to memory and computation imbalance. This imbalance is caused by the varying number of warm-up micro-batches on different PP ranks and the presence of specialized model structures, such as input embedding and output head, which are only located on the first and last PP ranks. Consequently, training may encounter out-of-memory (OOM) issues on the first PP rank, while later ranks have substantial available memory, or experience pipeline bubbles due to the heavy computational load on the last PP rank. To mitigate these issues, we co-designed the PP schedule with the model architecture. Specifically, we reduced one layer from the first pipeline rank to decrease the peak memory across PP ranks, and similarly reduced one layer from the last pipeline rank to balance the computational workload. As a result, the Llama 3 405B model is configured with 126 layers (instead of 128 layers at the very beginning).

3.1.3 Co-optimization of PP and FSDP. We investigate the combination of FSDP with ZeRO-1/ZeRO-2 and PP. The 1F1B schedule alternates between the execution of different virtual stages, necessitating the gradient accumulation across executions of the same virtual stage. FSDP ZeRO-2 with PP reshards gradients to save memory, but at the cost of additional gradient reduce-scatters, as illustrated in Figure 4. In contrast, FSDP ZeRO-1 with PP retains unsharded gradients across virtual stages, trading off increased memory usage for reduced communication overhead. For Llama 3 training, we adopt FSDP ZeRO-1 with the 1F1B schedule when $bs >= 2 \times pp$ and ZeRO-2 with all-forward-all-backward when $bs < 2 \times pp$, to achieve better performance. Our experiments reveal that, at larger scales, FSDP reduce-scatter can lead to traffic congestion with other parallelisms, resulting in degraded point-to-point (P2P) performance.

3.2 Case Study: Multimodal Training

In this section, we provide a detailed overview of Llama 3's multimodal training, highlighting how we adapt our PP schedule to enable efficient and flexible training.

The Llama 3 multimodal model combines a pre-trained ViT image encoder [42] and the pre-trained Llama 3 text model. To integrate the two, we insert transformer layers that utilize cross-attention (hereafter referred to as cross-attention layers) into every few original transformer layers from the text model, which employ self-attention (hereafter referred to as self-attention layers). The cross-attention layers take inputs from both the image encoder and self-attention layers. The model architecture is illustrated in Figure 5. During pre-training, the self-attention layers are frozen, while the image encoder and cross-attention layers are trained. For further training details, please refer to the Llama 3 technical report [24].

Due to the differences in model architecture mentioned above, we encounter two key challenges when scaling multimodal pre-training.

- **Challenge 1: Sharding of image encoder**. In addition to the text model, we must also consider the sharding of the image encoder, which has distinct compute and memory characteristics. Furthermore, the sharding strategy must be general and flexible to scale effectively across various image encoder configurations,

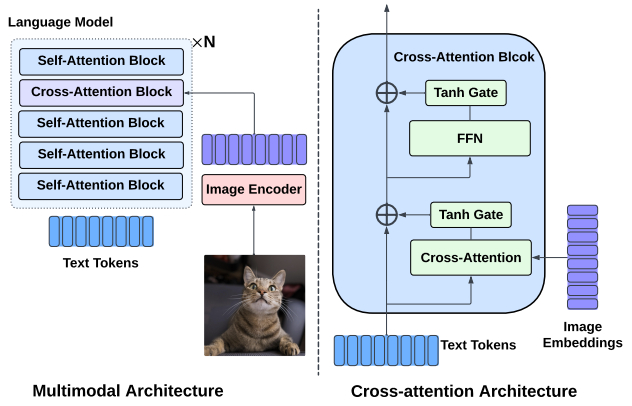


Figure 5: Illustration of Llama 3 multimodal architecture.

as overfitting to a specific configuration can result in suboptimal performance on others.

- **Challenge 2: Workload imbalance of the text model**. The self-attention and cross-attention layers exhibit distinct compute and memory characteristics: (1) the text sequence length is much shorter (less than 200 tokens) compared to the pre-trained text model (8K tokens), and (2) the majority of the model weights (self-attention layers) is frozen. Consequently, directly reusing the PP configuration from text pre-training, which assigns one transformer layer per virtual PP stage, results in a severe workload imbalance across PP ranks.

In the following sections, we describe how we adapt our PP designs to address these two challenges.

3.2.1 Sharding of image encoder. We evaluated three candidate sharding choices before implementation, as depicted in Figure 6.

Option 1: Shard the whole (image + text) model with PP. We place the image encoder on the first PP rank and run it together with the first text model virtual stage on the first PP rank for each micro-batch. Outputs from the image encoder are passed down along with transformer layer outputs to all other PP ranks through P2P communication.

Option 2: Separate the image and text model, and apply PP only to the text model. We separate the image encoder from the text model, run the image encoder as a pre-processing stage of the text model on the first PP rank, and then broadcast image tokens to all pipeline stages and split them into micro-batches to feed the text model, which is trained with PP. After the text model pipeline finishes, the gradients of the image tokens are all-reduced, and we run the backward pass of the image encoder.

Option 3: Shard the image model across PP ranks. We shard or duplicate the image encoder across all PP ranks. Each image encoder replica takes a portion of the input (bs/pp). The outputs are all-gathered across the PP stages and fed to the text model pipeline.

Among the three options, Option 1 requires minimal code changes, as we could reuse the PP design for the text model and only pack more tokens (to include the image tokens) for P2P communication. However, this design worsens the workload balancing issues for PP, as the first PP rank is assigned more compute, including the image encoder. As the configurations of image encoder could change during training, this design is inflexible and struggles to adapt to these changes while maintaining high efficiency.

In contrasat, Option 2 and 3 decouple the image encoder from the text model and offer more flexibility in configuration during training. In our initial implementation, we adopted Option 2 for ease of implementation. By placing the image encoder and reducing the text model transformer layers on the first PP rank, we achieved a good training throughput. However, later during the training stages, the image resolution was significantly increased (from 448×448 to 672×672 pixels), and more transformer layers were added into the image encoder. Combining these factors, the image encoder took a much longer latency (up to 33% of the combined image and text model training latency), leading to a significant drop in overall training throughput.

To address this issue, we switched to Option 3, replicating the image encoder on each PP rank, splitting the data batch into micro-batches, and letting the encoder compute them in parallel. This optimization reduced the encoder compute ratio from 33% to 8% and recover the TFLOPs achieved before the model changes.

3.2.2 Workload imbalance of text model. There are two key differences between cross-attention and self-attention layers.

- Cross-attention takes both image and text sequences as the inputs, whereas self-attention only takes the text sequence as input. The image sequence is much longer than the text sequence during pre-training (e.g., 1.2K tokens for 448×448 resolution and 3K tokens for 672×672 resolution, compared to less than 200 tokens for the text sequence). As a result, the compute FLOPs of cross-attention layers in the forward pass are much larger than those of self-attention layers, depending on the ratio of image and text sequence lengths in the batch.
- Self-attention layers are frozen in the training. As a result, during the backward pass, self-attention layers only compute input gradients, whereas cross-attention layers compute both weight and input gradients, further exacerbating the workload imbalance between the two layers.

When partitioning the text model for PP, we explored two options for placing self-attention and cross-attention layers: (1) wrapping n self-attention layers and one cross-attention layer in one PP virtual stage, or (2) wrapping either n self-attention layers or one cross-attention layers in one virtual stage. Option 1 achieves a more balanced workload distribution across PP ranks but results in fewer PP virtual stages and a larger PP bubble ratio. In comparison, Option 2 generates more PP virtual stages with a smaller PP bubble ratio. However, achieving workload balance is challenging due to the workload differences discussed above. In Llama 3 multimodal pre-training, we adopted Option 1 due to its simplicity. We co-designed the multimodal model to determine the final ratio of cross-attention layers to self attention layers (4:1), which achieved a reasonable training throughput and helped meet the production training deadline given the compute budget.

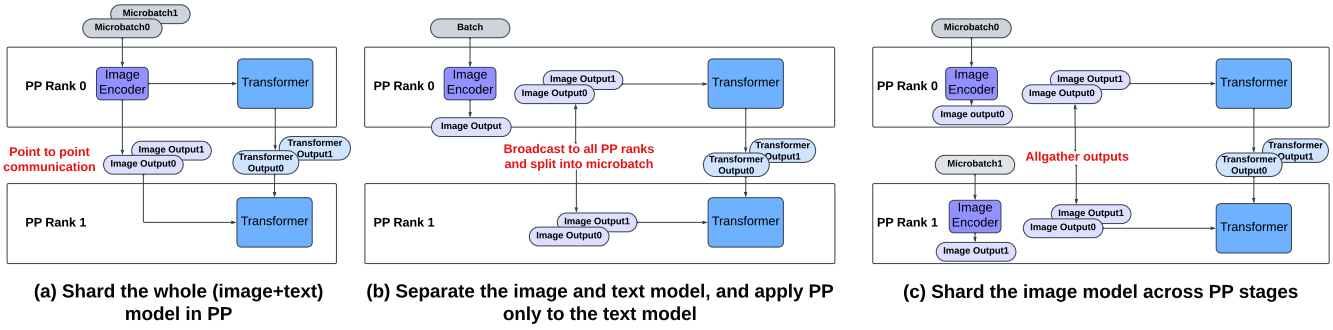


Figure 6: Options to shard the encoder with PP. (a) Place the encoder on the first PP rank, and communicate outputs from both the image encoder and transformer layers across PP ranks. (b) Place the encoder on the first PP rank, pre-process all inputs with the image encoder and broadcast the encoder output; then run the training of the text transformer model. (c) Replicate/shard the encoder on all PP ranks, and shard inputs across PP so that each encoder replica processes bs/pp portion of inputs. All-gather outputs before running the transformer.

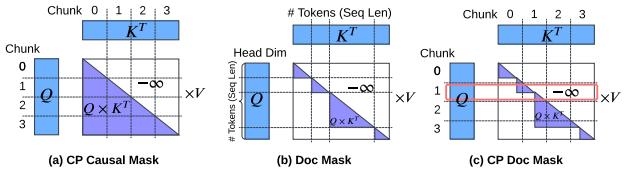


Figure 7: Examples of CP sharding for different attention masks.

4 Context Parallelism

To enable long context training for Llama 3, we introduce context parallelism (CP) by splitting input tokens along the sequence length dimension. Although decreasing the DP size dp to increase CP size cp requires an increase of the batch size per DP group bs to maintain the same global batch size gbs, the use of PP in 4D parallelism makes the peak memory usage independent of bs. Consequently, when CP splits along the sequence length, it reduces the peak memory usage despite the increasing bs. For more information on how each parallelism affects memory usage, please refer to Section 5.

Design: In Llama 3, we propose and develop an all-gather-based CP attention to deliver an efficient and flexible solution. This is achieved by all-gathering key(K) and value(V) tensors before attention computation. Although existing work, such as RingAttention [21], has proposed solutions to overlap P2P communication of blocks of tokens with computation, we adopt the all-gather-based CP attention for two primary reasons:

First, the Llama 3 model architecture requires flexibility to support irregular attention masks, whereas existing work assumes the usage of a full causal mask. Specifically, the attention mask in Llama 3 enforces tokens to attend only to other tokens from the same document (i.e., document mask), and this document boundary depends on the position of the end-of-sequence IDs (eos_ids) in input tokens. Computing the mask on every tile of tokens is error-prone, and irregular token communication makes it challenging to fully utilize network bandwidth.

Second, the all-gather approach does not incur significant performance overhead compared to ring-based approaches. Due to multi-group attention (MGA) or group query attention (GQA), the number of KV heads is smaller than the number of heads, resulting in smaller K and V tensors compared to the Q tensor in attention. Moreover, the communication latency is under a time complexity of $O(seq)$ with increasing of sequence length (denoted as seq), while the time complexity of attention computation is $O(seq^2)$. This makes the exposed all-gather communication latency a smaller portion of CP attention as seq increases. With smaller sequence lengths (e.g., $seq = 8192$ or $seq = 16384$), all-gather-based attention can still achieve a comparable performance to RingAttention because RingAttention requires merging partial attention results with scaling and rescaling according to softmax log-sum-exp results [7, 8]. More concrete results on the performance comparison between all-gather-based and ring-based CP attention can be found in Section 7.2.

Implementation: In our CP attention implementation, we split the input tokens evenly into $2 \times cp$ chunks and assign each rank i to process both the i-th and $(2 \times cp - i - 1)$-th chunks of tokens. This sharding strategy ensures a balanced computation workload among CP ranks. For example, Figure 7(a) illustrates a split of input tokens into 4 chunks for $CP = 2$, where the purple area represents the computation workload among token chunks with the causal mask (token i only attends token $0, 1, ..., i-1$). Figure 7 also shows an example of a document mask in Llama 3, where tokens only attend to other tokens from the same document. Although the document mask reduces the computation workloads, we still adopt the sharding strategy optimal for the full causal mask because the elapsed time of a train step is often bounded by the slowest rank of DP group and PP stages. In these cases, the slowest rank often processes the full long sequence without an eos_id to split tokens.

The document mask poses a challenge to CP attention, as the document boundary is irregular and input-dependent, and it does not align with the static CP sharding of tokens. In our all-gather-based implementation, we all-gather K and V tensors, making computation on each Q token independent. For example, Chunk 1 in

Figure 7(c) only needs Chunk 0 and Chunk 1's K and V tensors to compute output results. However, some tokens in Chunk 1 still need to attend to tokens from Chunk 0 because there belong to the same document across the CP sharding boundary. For instance, considering the example in Figure 7 with 16 tokens and a document length of [3, 3, 8, 2], the first two tokens in Chunk 1 need to attend to all three tokens from the same document. To address this challenge, we pad the Q sequence length with leading zeros for attention computation outside this chunk of tokens. However, we retain the K and V sequence length information, as we have a full K and V tensors after all-gather. This approach enables us to implement an efficient and accurate CP attention mechanism that supports the document mask.

Integration: When integrating CP into the end-to-end training system, we need to consider its impact on several components:

Data parallel (DP) group: Although CP splits input tokens along the sequence length, the tokens processed by the same CP group still share the same set of model parameters. Consequently, CP can be seen as an extension of DP when communicating model parameters, such as during all-gather operations of parameters and reduce-scatter of parameter gradients.

CP ranks: Ranks within the same CP group select their local tokens and compute their own attention masks using the entire sequence. As detailed in our CP attention implementation, each CP rank requires a complete sequence to accurately compute the KV seqlen and adjust the pad Q seqlen. Local token selection follows our sharding method, where rank i takes both i-th and $(2\times cp-i-1)$-th chunks of tokens. Additionally, positional encodings should be selected appropriately [41].

Dataloaders: Dataloaders continue to provide different batches of input training data to the original DP groups. The sequence length split is not visible to the tokenizer because each CP rank requires the full sequence information to compute the attention mask accurately.

5 4D Parallelism for Llama 3

We provide an overview of parallelism configurations for training Llama 3 at scale in Table 2. Training Llama 3 models on 16K GPUs is a capability computing challenge, where the goal is to maximize efficiency to reduce the total training time. At this scale, a fixed batch size of 16M tokens per training step restricts the degree of parallelism achievable across training samples. In this section, we elaborate on our reasoning process for finalizing parallelism configurations (i.e., determining the size of each parallelism dimension) to fully leverage 16K GPUs in the training of Llama 3 models with a global data batch size of 16M tokens.

5.1 The Size of Parallelism Dimensions

We decouple the size of multiple parallelism dimensions into several key steps: First, we determine the minimal TP size considering the limited global batch size. Second, we explore the rational of using 3D parallelism instead of 2D parallelism in Llama 3. Finally, we explain the advantages of incorporating CP in long context training, and the configuration of our 4D parallelism. The definitions of the symbols used in this section are provided in Table 1.

Table 2: The size of each parallelism dimension for Llama 3 pre-training of 405B models with a global batch size of 16M tokens on 16K GPUs (more details in the Table 4 of the Llama 3 technical report [10]).

Context Length	Global Batch Size	TP	CP	PP	DP
8,192	2,048	8	1	16	128
131,072	128	8	16	16	8

TP size: For a global token budget of 16M and sequence length seq 8K, the global batch size gbs is 2048. The per-GPU batch size $bs = gbs/ndp$, where ndp is the number of data parallel groups, calculated as $ndp = ngpu/dp = ngpu/tp/pp/cp$. Using 2D parallelism, bs becomes $gbs/ndp = 2K/(16K/tp/1/1) = tp/8$. To ensure $bs \geq 1$, we require $tp \geq 8$. With 3D parallelism, bs is $tp \times pp/8$. For efficient PP with minimal bubbles, we prefer $bs \geq pp$, thus $tp \geq 8$. Therefore, when training with 16M tokens per step on 16K GPUs, $tp \geq 8$ is necessary for both 2D and 3D. In our training cluster setting, each node has 8 GPUs. Setting $tp \leq 8$ limits TP to use intra-node communication (i.e., NVLink), offering much higher bandwidth than inter-node communication. In summary, $tp = 8$ is optimal TP given the batch size and hierarchical network bandwidth constraints.

2D or 3D parallelism: To fit the model into memory, we consider either 2D parallelism (FSDP ZeRO-3 + TP) or 3D parallelism (FSDP ZeRO-1/2 + TP + PP). With $tp = 8$ for both 2D and 3D, the efficiency mainly depends on the FSDP and PP communication overheads. For 2D with $bs = 1$, the computation latency is not long enough to hide FSDP communications, while 3D has cheaper and more stable P2P communications; thus, we chose 3D parallelism. For example, each model parameter contributes to 2 bytes of communication data in FSDP ZeRO-3 (assuming BF16 data type) and 2 computation FLOPs for every token in the forward pass. When training with 8K tokens ($bs = 1$ and $seq = 8192$), the arithmetic intensity of computation over communication is $(2 \times 8K)/2$ FLOPs per communication byte, which is much lower than the hardware ratio of peak computation FLOPs over network bandwidth, i.e. 989 TFLOPs for BF16 on Nvidia H100 GPU [26] over 50 GB/s RoCE network bandwidth ($989K/50 = 19.78K$) [24].

When configuring 3D parallelism, we chose $pp = 16$ to fit the model into memory. We did not consider FSDP ZeRO-3 with PP for three reasons: First, the model parallelism dimensions ($pp = 16$ and $tp = 8$) are large enough to accommodate 405B models. Second, FSDP ZeRO-3 has extra communication overheads on every PP stage forward and backward. Third, FSDP communications have performance interference when overlapped with PP. In particular, inter-host P2P communications of PP are slowed down due to the resource contention of network hardware bandwidth between PP and FSDP communications.

4D parallelism for long context: For the long context phase of Llama 3 pre-training, the global batch size is reduced from 2K to 128 as the sequence length is increased to 128K, while the number of tokens per global batch remains the same. Without changing the parallelism configuration, this immediately means bs drops to 1 which completely tanks performance due to unbearable bubble in PP. Decreasing DP leads to larger TP or PP in exchange for a larger

bs. However, neither trade-offs is favorable because increasing PP at the same rate does not resolve the pipeline bubble issue, and increasing TP beyond 8 introduces expensive inter-host TP communications on the critical path. Given this, CP comes in as the perfect solution by sharding the sequence dimension within each training sample.

When we introduce CP to existing parallelism, we first need to consider replacing which parallelism dimensions with CP. When the global token budget remains 16M but the sequence length increased to 131K, gbs becomes 128. With the batch size constraint $bs \geq 1$, we cannot replace TP or PP with CP; thus we can only replace DP with CP. With $tp = 8$ and $pp = 16$, as discussed previously, $gbs/(ngpu/tp/pp/cp) \geq pp$ is strongly preferred for PP efficiency, which means $cp \geq 16$. We use $cp = 16$ to minimize CP communication overheads. Overall, CP allows us to easily scale to the long context training phase while keeping the same bs and pp configurations and additionally reduces activation memory usage by sharding the sequence length dimension.

5.2 The Order of Parallelism Dimensions

To maximize the efficiency of our training cluster's network bandwidth, we order the parallelism levels carefully. Our training cluster has a hierarchical network topology, ranging from high-bandwidth NVLink for GPUs within the same host as the innermost layer to lower-bandwidth cross-node networks as the outer layers. As a guiding principle, we place the parallelism dimensions with higher communication demands (i.e. higher communication data volume, higher communication frequency, and/or communication latency more difficult to hide) into the inner levels of parallelism:

TP communication: TP involves all-gathering and reduce-scattering activation or gradient tensors on every linear module. These communications are fully exposed to the critical path and occur four times in every transformer layer, twice for the attention module and twice for the feed-forward network (FFN) module.

CP communication: CP involves all-gathering KV tensors or reduce-scattering the gradients of KV tensors on the inner-attention module. The communication latency is fully exposed, and it occurs once in each transformer layer. Although CP has a similar communication data volume to PP, it involves cp ranks in a collective communication, whereas PP involves P2P between two ranks. Therefore, CP communication latency is longer due to the synchronization among CP ranks.

PP communication: PP communicates on every virtual pipeline stage. There is no synchronization between the two P2P ranks due to decoupled asynchronous P2P send and receive. When $pp = bs$, all P2P communications are fully exposed. Nevertheless, we prioritize CP over PP due to the aforementioned reasons when analyzing CP communication.

DP communication: DP with ZeRO-1/ZeRO-2 communicates only once per training step, i.e. all-gathering model parameters and reduce-scattering gradients. Although its communication volume is comparable to PP, we can potentially hide its communication latency with forward / backward computation (i.e. all-gathering parameters overlapped with model forward and reduce-scattering gradients overlapped with model backward). Thus, we place DP as the outermost level in 4D parallelism.

Figure 8: Identify slow ranks in process groups.

In summary, considering the communications along every parallelism dimension, we have a parallelism order of [TP, CP, PP, DP] from the innermost level to the outermost level.

6 Debugging Parallelism at Scale

When scaling up the training of Llama 3 using efficient multi-dimensional parallelism, debugging performance, memory usage, and numerical issues becomes a crucial task. We share our debugging process and lessons learnt from this process to facilitate future research and development.

6.1 Performance: Identifying the Slow Rank

In multi-dimensional parallelism, debugging performance issues at scale can be challenging, particularly when trying to identify the root cause of the problem. The interactions between various parallelisms complicate the process of tracing a slow communication collective to its root cause. For example, Figure 8 illustrates a configuration with 8 GPUs and ($cp = 2$, $tp = 4$), along with a stacked performance trace of TP communication collectives for 4 GPUs within a TP group. The trace reveals that Rank 2 is the slowest rank in this group, as its communication collectives are the shortest, indicating that other ranks are waiting for Rank 2 to join. However, it is unclear whether Rank 2 is the bottleneck of the entire system, as its slowness could be caused by its CP communication collectives, where its peer rank (Rank 6) in the CP group might be the actual bottleneck.

To address this challenge, our performance trace analysis employs a top-down approach, starting from the outermost parallelism level. As detailed in Section 5.2, our parallelism is ordered as [TP, CP, PP, DP] from inner to outer levels. We begin by analyzing the DP groups to identify the slowest one, and then iteratively repeat this process for PP, CP, and TP groups to narrow down the range of slow ranks. Once the slow rank is identified, we can examine the detailed profiling trace of CPU, GPU compute, and GPU communication to investigate the root cause, whether it be a software or hardware issue (e.g., a faulty GPU). This method is similar to failure localization in distributed systems [39, 43], where the problematic host is not necessarily the first one to crash and report errors. An automatic tool for analyzing performance traces and identifying the root cause of the slowest rank would be a valuable asset for performance debugging in Llama training systems.

6.2 Numerical Issues in 4D Parallelism

The partitioning of training data and model parameters in 4D parallelism inevitably alters numerical behaviors due to the non-commutative and non-associative nature of floating-point additions. The use of low-precision data types, such as BFloat16 (BF16) [13]

further exacerbates the issue. Therefore, identifying and mitigating numerical gaps is crucial to ensure training stability, and it is an important design goal of the training system.

Distinguishing numerical issues from implementation bugs: When developing 4D parallelism, it is essential to distinguish whether the training loss behaviors are different as a result of numerical issues (e.g., different accumulation orders) or an implementation bug. The former can be mitigated by a higher precision accumulation order in certain parallelism implementations, whereas the latter needs further investigations to fix the root cause. As parallelism splits computation into chunks and reduces partial results, it cannot achieve bit-wise matching results as the sequential version. To distinguish these two different reasons, we adopt an approach to split the sequential version into the same accumulation order as the parallel one and check for bit-wise exact matching. For example, we maintain a 2D parallelism (DP and TP) design with micro-batching to emulate the accumulation order of PP micro-batching, serving as a reference baseline to confirm whether numerical gaps are due to PP implementation bugs or accumulation order differences.

Accumulating gradients in FP32: With the aforementioned debugging process to identify numerical issues, we use FP32 accumulation for gradients and optimizer states to bridge numerical gaps, while maintaining BF16 formats for model computation and communication. Specifically, we use FP32 for DP group reduce-scatter of gradients and for accumulating gradients of micro-batches in PP backwards. This accumulation precision aligns with hardware units, where GEMM kernels accumulate partial results in FP32 for two BF16 input matrices. In the backward computation, accumulation occurs along the batch size dimension, where DP splits it into mini-batches and reduce-scatters gradients, and PP further splits mini-batches further into micro-batches and accumulates gradients during PP backwards. In multimodal training, we further cast image tokens sharded by all the cross-attention layers to FP32 so that the gradients are reduced across all cross-attention in FP32 precision during the backward pass.

6.3 Memory Optimizations

When running 4D parallelism in large-scale systems, we find that 4D parallelism itself brings unique opportunities to improve memory efficiency other than splitting model parameters and input training data. For example, PP stage only needs forward output tensor metadata (i.e. tensor shape) to kick-off the backward pass but conventional PyTorch autograd engine is conservative in releasing memory with reference counting. To optimize memory usage in Llama 3 training system, we first profile the memory cost by the memory snapshot tool [1] to get a detailed memory allocation trace. Then we either develop a customized autograd operator to save tensor checkpoints during forward, or utilize existing autograd engine but release underlying tensor data by resizing the tensor storage manually. We note that these optimizations are helpful in our parallelism configurations to eliminate activation recomputation and avoid increasing PP or TP to fit the training into the memory thus helping with the training efficiency.

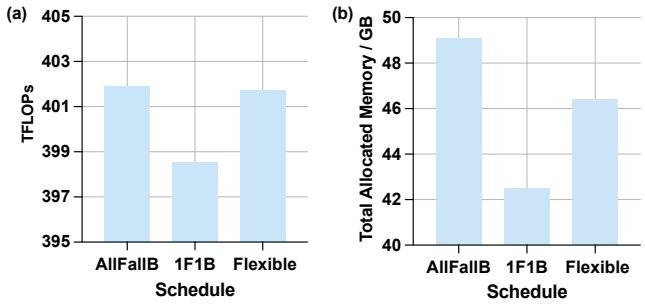


Figure 9: (a) Training TFLOPs for all-forward-all-backward, 1F1B, and flexible PP. (b) Max allocated memory usage.

7 Evaluation

In this section, we evaluate PP and CP individually and also assess the end-to-end performance of 3D and 4D parallelism.

7.1 Pipeline Parallelism

To validate our optimizations, we conduct small-scale experiments using a scaled-down version of the Llama 3 405B model. Specifically, we maintain the same model dimensions but reduce the number of layers, and use a sequence length of 8192 for our experiments.

7.1.1 Training throughput and memory comparison between all-forward-all-backward, 1F1B, and flexible PP. To compare between different schedules, we use a shrunk model with 26 layers and $pp = 4$, $bs = 12$. We reduce two layers from 28 to 26 for more balanced computation, as described in Section 3. The all-forward-all-backward schedule processes 12 micro-batches at once; while 1F1B processes pp micro-batches in one round and 3 rounds per PP virtual stage in total. Flexible PP, on the other hand, processes 6 micro-batches in one round and 2 rounds in total. The results are presented in Figure 9, which shows the memory usage and TFLOPs achieved by each schedule. The 1F1B schedule has the lowest memory usage due to its prioritization of the backward pass, but it achieves the lowest TFLOPs due to exposed P2P communications. In contrast, the all-forward-all-backward schedule achieves the highest TFLOPs by processing more micro-batches to hide exposed P2Ps, but it results in the highest memory usage. Flexible PP strikes a balance between memory usage and training throughput.

7.1.2 Balanced and imbalanced pipeline parallelism. In Llama 3 training, we use a vocabulary size of 128K, which leads to a large embedding module on the first PP rank and a large output module on the last PP rank. This, in turn, causes computation and memory imbalances across PP ranks. To mitigate this issue, we remove one layer from the first and last PP stages, resulting in more balanced memory allocation across PP ranks and improved training throughput due to balanced computation. As is shown in Figure 10, this optimization reduces the maximum allocated memory usage by 5GB and improves TFLOPs by 6.5%. Furthermore, with this optimization, we can turn off activation recomputation [5] in Llama 3 training. For the scaled-down model, balanced PP yields a 17.5% improvement in TFLOPs by avoiding recomputation.

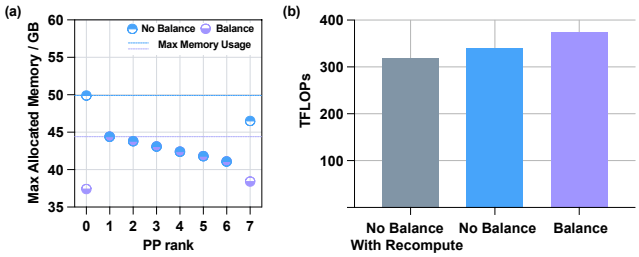


Figure 10: (a) Max allocated memory usage across PP ranks with and without workload balance. (b) Training throughput with balanced PP.

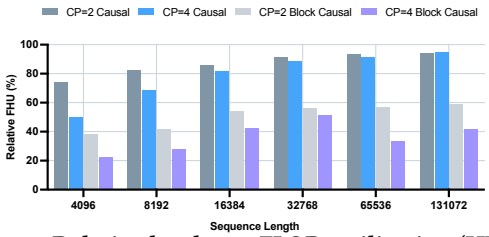


Figure 11: Relative hardware FLOPs utilization (HFU) over attention on single GPU.

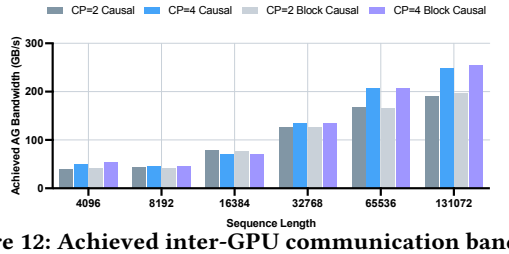


Figure 12: Achieved inter-GPU communication bandwidth of context parallelism all-gather.

7.2 Context Parallelism

As CP only introduces extra inter-GPU communication in the attention layers, we evaluate the efficiency of the attention layers to demonstrate the effectiveness and scalability of our CP solution.

We begin by comparing the efficiency and scalability of our CP solution with state-of-the-art attention kernels on GPU. Specifically, we use Flash-Attention V2 [7] as our single-GPU baseline and measure the hardware FLOPs utilization (HFU) [6]. We then normalize the HFU of CP attention over Flash-Attention on a single GPU. Since CP introduces communication between GPUs, we expect the relative HFU to be less than 100%, with higher values indicating better efficiency of CP attention. We conduct experiments on H100 with HBM2e [26] to assess the scalability of CP attention in a lower memory bandwidth setup. We anticipate better scalability of CP attention in HBM3, as attention kernels are generally compute-bound, while the extra element-wise and communication overheads are typically memory or network bandwidth-bound. We evaluate both $cp = 2$ and $cp = 4$ for different sequence lengths with full causal masks and block causal masks (i.e., document mask), where the average document length is 1K. Our results in Figure 11 show that (1) the relative HFU for longer sequence lengths is higher (up to 95% relative HFUs for 128K sequence length), and (2) CP attention for block causal masks has a lower relative HFU. The first observation is consistent with our time complexity analysis in Section 4, which indicates that the elapsed time of CP communication scales linearly, while computation scales quadratically with respect to sequence length. To further understand the second observation, we measure the achieved network bandwidth of CP all-gather communication and present the results in Figure 12. The achieved bandwidth of all-gather is comparable between causal and block causal masks. This suggests that the lower relative HFU of block causal mask is due to workload imbalance, where our static sharding of tokens across CP does not align with document mask boundaries, as shown in Figure 7.

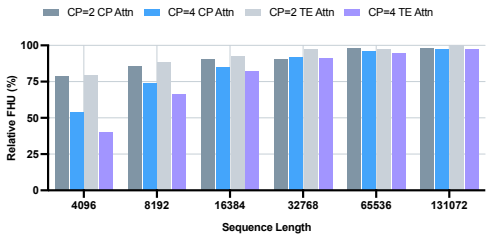


Figure 13: Relative hardware FLOPs utilization (HFU) comparison between context parallel attention (CP Attn) and TransformerEngine [27] attention (TE Attn).

In addition to the scalability study of CP attention, we conduct experiments comparing our CP solution with Transformer Engine [27] (TE) in a branch prior to initiating Llama 3 pre-training. TE attention employs a computation-communication overlapped method similar to RingAttention [21]. Specifically, it splits along the sequence length dimension into $2 \times cp$ chunks, assigns chunks to CP ranks to balance computation, iterates through chunks to compute partial attention results overlapped with P2P communication between adjacent ranks, and finally merges partial attention output results. In our forked branch of TE, prior to Llama 3 training, it does not support variable sequence lengths; therefore we only evaluate the full causal mask to compare with our CP attention solution. We conduct experiments on production hardware for Llama 3 training (i.e., H100 with HBM3), and Figure 13 presents the relative HFU results. We observe from Figure 13 that TE has a slightly higher relative HFU for $cp = 2$, but both our CP and TE attention achieve relative HFU over 95% when the sequence length exceeds 64K. Notably, our CP attention consistently outperforms TE attention for $cp = 4$, especially for sequence lengths of 4K and 8K, where we observe up to 13.53% better relative HFU. The superior performance of our CP attention can be attributed to the differences in the attention mechanism. Unlike our CP attention, TE's ring-style attention involves $O(cp)$ computation kernels, each working on a chunk of $seq/(2 \times cp)$ tokens to compute partial results. When cp is large and sequence length is small, this ring-style attention in TE suffers from both fragmented compute kernels with lower compute efficiency and the compute overheads of merging attention partial results.

7.3 End-to-End Performance

Llama 3 405B is trained on up to 16K H100 GPUs, each operating at 700W TDP with 80GB HBM3, using Meta's Grand Teton AI server

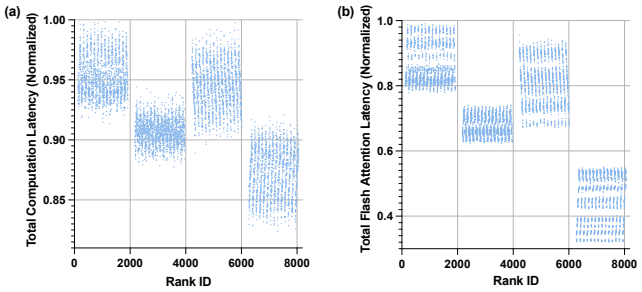


Figure 14: (a) The distribution of total computation time across all GPUs. (b) The distribution of the total time of attention kernels across all GPUs.

platform [22]. The setup for 4D parallelism and input batch size for Llama 3 pre-training with both 8K and 131K sequence lengths on 16K GPUs is detailed in Table 2. With the optimizations discussed in this paper, we achieve 400 TFLOPs/GPU for 8K sequence length and 380 TFLOPs/GPU for 131K sequence length. We further analyze the end-to-end performance and key take-aways for PP and CP below.

7.3.1 Text model with 3D parallelism. We overlap FSDP's all-gather and reduce-scatter with other computation and communications, exposing only the first all-gather and last reduce-scatter. P2P communications are exposed during the warm-up forward and cool-down backward phases. Except for the shorter first and last model chunks due to having fewer transformer layers, forward and backward passes on regular model chunks are balanced across micro-batches and across PP stages, achieving a 5% bubble ratio when per data parallel group $bs = 2 \times pp$ and 12% bubble ratio when $bs = pp$.

7.3.2 Long context text model with 4D parallelism. When training long context text model with 128K sequence length, we enable $cp = 16$ so that each GPU rank still receives 8K sequence length, similar to the base model with 3D parallelism. We conducted extensive profiling of long context jobs with 8K GPUs and found that the exposed latency of CP communication (all-gather in forward and reduce-scatter in backward) accounts for 7.64% of the total elapsed time. However, an in-depth analysis reveals that, 65.75% of CP exposed latency results from waiting for the slowest rank in the CP group to join the collective. The root cause stems from workload imbalances across all GPUs due to the document mask used in Llama 3 training [10], and this workload imbalance issue worsens with longer sequence length and larger cp. Figure 14 (a) shows the total time of computation kernels across all GPUs, where the slowest rank spends 1.44× more time on computation than the fastest rank. Further analysis in Figure 14 (b) reveals that this gap in total computation time is entirely due to the difference in attention kernel time across GPUs, indicating that the imbalance caused by the document mask contributes to the computation imbalance we observed. Consequently, a large portion of the CP exposed communication latency is attributed to this imbalance. Note that all parallel algorithms on CP that overlap CP communication and attention computation must wait for the slowest CP rank to complete, leaving an upper-bound of 2.62% end-to-end performance improvement compared to our all-gather-based CP solution.

8 Recommendations for Future Hardware

Hardware optimizations tailored for large-scale LLM training can be different from the ones for general AI workloads. Based on our experience with Llama training, we offer recommendations for future hardware specifically for LLM training.

8.1 Node level recommendations

For LLMs that require high dimensions of parallelism, achieving high throughput from a combination of accelerators and hosts is crucial.

Optimize compute efficiency for a wide range of shapes: It is not sufficient for a hardware accelerator to provide high compute throughput only for very large shapes. Parallelisms will reduce the dimension of GEMMs. For example, PP reduces batch size and CP shards on sequence length. This can lead to lower arithmetic intensity operations, so it is essential to ensure that sufficient memory bandwidth is provided relative to compute throughput.

Higher HBM capacity can improve performance: Higher HBM capacity can increase the feasible hyper-parameter space for multi-dimensional parallelism, leading to higher overall performance. For example, sharding less in the tensor dimension leads to higher memory usage but also reduces TP communication overheads due to better amortization of communication relative to compute. In Llama 3 small scale experiments on 2K GPUs, we observed approximately 10% end-to-end performance improvement by reducing TP size from 8 to 4. Higher HBM capacity allows exploring all such options. However, the actual benefits depend on the model parameters, hardware platform, and the cluster size.

Ensure sufficient CPU performance: The gen-over-gen performance improvement trend is significantly faster for accelerators than CPUs. Large-scale LLM training for future accelerators can become CPU-bound for multiple reasons. Scaling to large clusters leads to smaller GEMM dimensions assigned to each accelerator, as stated above. Furthermore, model engineers can incorporate complex operations for exploration, leading to a sequence of lightweight kernels with relatively high host CPU overheads. These can cause CPU times spent preparing and launching kernels to become comparable to accelerator runtimes unless addressed through hardware and software improvements.

Minimize performance variations and make DVFS deterministic: Parallelisms create synchronization among accelerators. If there are performance variations across different accelerators, the whole cluster performance will be determined by the slowest one. Furthermore, if different accelerators slow down at different times due to transient issues, the slowdown will accumulate due to fine-grain synchronization across a large number of accelerators involved in the cross product of TP, CP, and PP domains. Therefore, it is essential to ensure that dynamic voltage frequency scaling (DVFS) [34] policies are deterministic across accelerators to avoid transient slowdowns at different times.

8.2 Training cluster level recommendations

To further scale up training, we also need to consider how to connect nodes with efficient networks and make the best use of power for a data center.

Optimize network hierarchy: Scaling LLM training to 100K or more accelerators requires an efficient network with multiple levels of switches. Providing the same network bandwidth at each level of hierarchy will not be the most cost- or power-efficient design. Instead, a hierarchical network can be designed where the upper-level switches have less or oversubscribed bandwidth. It is essential to consider the requirements of different parallelism dimensions while determining these network parameters. We recommend co-designing the network parameters based on the anticipated workload requirements, taking into account model hyper-parameter as well as all dimensions of parallelism.

Ensure robust network performance: A slow down between two ranks (e.g., due to congestion or packet loss) can affect the whole cluster performance due to fine-grain parallelism across a large set of accelerators involved in TP, CP, PP and DP communication. We need to ensure that the entire network operates at a consistent performance without transient slowdowns.

Prioritize power efficiency: It has been reported that future LLM training clusters are trending toward 100K GPUs or potentially more [32]. These large clusters are constrained by the total amount of power available in a data center region rather than the number of AI accelerators that can be procured. Therefore, an accelerator's effective performance per unit of power consumption (Perf/Watt) is as important as, or even more important than, its absolute performance.

9 Conclusion

This paper presents the details of the training system for Llama 3 text and multimodal pre-training. We adopt 4D parallelism in this system to scale out across up to 16K GPUs and apply numerous optimizations to achieve high efficiency under batch size constraints. Our system is designed to be highly flexible, supporting dynamic workloads for different training phases and heterogeneous model architectures. Additionally, we provide a methodology for debugging performance and numerical issues at large scale. Drawing from our experience with Llama 3 training, we offer suggestions for future training node and cluster design. While efficient Llama training requires a holistic co-design of model architectures, learning algorithms, and training infrastructure (e.g. fault tolerance) beyond 4D parallelism, we hope that the details and insights shared in this paper will shed light on the directions of future model development and software-hardware co-design.

References

[1] Zachary DeVito Aaron Shi. 2023. *Understanding GPU Memory 1: Visualizing All Allocations over Time.* https://pytorch.org/blog/understanding-gpu-memory-1/

[2] 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, Mikoł aj Bińkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karén Simonyan. 2022. Flamingo: a Visual Language Model for Few-Shot Learning. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 23716–23736. https://proceedings.neurips.cc/paper_files/paper/2022/file/960a172bc7fbf0177ccccbb411a7d800-Paper-Conference.pdf

[3] Anthropic. 2024. *Introducing the next generation of Claude.* https://www.anthropic.com/news/claude-3-family

[4] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-vl: A frontier large vision-language model with versatile abilities. *arXiv preprint arXiv:2308.12966* (2023).

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2024. PaLM: scaling language modeling with pathways. *J. Mach. Learn. Res.* 24, 1, Article 240 (March 2024), 113 pages.

[7] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR).*

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS).*

[9] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024).

[10] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet,

Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. arXiv:1811.06965 [cs.CV] https://arxiv.org/abs/1811.06965

[12] 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. 2025. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 41, 16 pages.

[13] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A Study of BFLOAT16 for Deep Learning Training. (2019). arXiv:1905.12322 [cs.LG] https://arxiv.org/abs/1905.12322

[14] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. arXiv:2205.05198 [cs.LG] https://arxiv.org/abs/2205.05198

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[16] Joel Lamy-Poirier. 2023. Breadth-First Pipeline Parallelism. arXiv:2211.05953 [cs.DC] https://arxiv.org/abs/2211.05953

[17] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. (2020). arXiv:2006.15704 [cs.DC] https://arxiv.org/abs/2006.15704

[18] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[19] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2024. Visual instruction tuning. *Advances in neural information processing systems* 36 (2024).

[20] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. arXiv:2310.01889 [cs.CL] https://arxiv.org/abs/2310.01889

[21] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. arXiv:2310.01889 [cs.CL] https://arxiv.org/abs/2310.01889

[22] Meta. 2022. *Meta open compute project, grand teton ai platform.* https://engineering.fb.com/2022/10/18/open-source/ocp-summit-2022-grand-teton

[23] Meta. 2024. *Introducing Llama 3.1: Our most capable models to date.* https://ai.meta.com/blog/meta-llama-3-1/

[24] Meta. 2024. *Introducing Meta Llama 3: The most capable openly available LLM to date.* https://ai.meta.com/blog/meta-llama-3/

[25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM. https://arxiv.org/abs/2104.04473

[26] NVIDIA. 2022. *NVIDIA Hopper Architecture In-Depth.* https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[27] NVIDIA. 2024. *NVIDIA TransformerEngine.* https://github.com/NVIDIA/TransformerEngine

[28] OpenAI. 2022. *Introducing ChatGPT.* https://openai.com/index/chatgpt/

[29] Qwen Team. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024).

[30] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs.LG] https://arxiv.org/abs/1910.02054

[31] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. arXiv:2101.06840 [cs.DC] https://arxiv.org/abs/2101.06840

[32] SemiAnalysis. 2025. 100,000 H100 Clusters: Power, Network Topology, Ethernet vs InfiniBand, Reliability, Failures, Checkpointing. https://semianalysis.com/2024/06/17/100000-h100-clusters-power-network/

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[34] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. 2019. The Impact of GPU DVFS on the Energy and Performance of Deep Learning: an Empirical Study. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*. 315–325.

[35] Jakub Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 24829–24840.

[36] Team Gemini. 2023. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805* (2023).

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023). arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[38] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[39] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[40] Zhiyu Wu, Xiaokang Chen, Zizheng Pan, Xingchao Liu, Wen Liu, Damai Dai, Huazuo Gao, Yiyang Ma, Chengyue Wu, Bingxuan Wang, Zhenda Xie, Yu Wu, Kai Hu, Jiawei Wang, Yaofeng Sun, Yukun Li, Yishi Piao, Kang Guan, Aixin Liu, Xin Xie, Yuxiang You, Kai Dong, Xingkai Yu, Haowei Zhang, Liang Zhao, Yisong Wang, and Chong Ruan. 2024. DeepSeek-VL2: Mixture-of-Experts Vision-Language Models for Advanced Multimodal Understanding. (2024). arXiv:2412.10302 [cs.CV] https://arxiv.org/abs/2412.10302

[41] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, Madian Khabsa, Han Fang, Yashar Mehdad, Sharan Narang, Kshitiz Malik, Angela Fan, Shruti Bhosale, Sergey Edunov, Mike Lewis, Sinong Wang, and Hao Ma. 2023. Effective Long-Context Scaling of Foundation Models. arXiv:2309.16039 [cs.CL] https://arxiv.org/abs/2309.16039

[42] Hu Xu, Saining Xie, Xiaoqing Ellen Tan, Po-Yao Huang, Russell Howes, Vasu Sharma, Shang-Wen Li, Gargi Ghosh, Luke Zettlemoyer, and Christoph Feichtenhofer. 2023. Demystifying clip data. *arXiv preprint arXiv:2309.16671* (2023).

[43] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 131–146.

[44] 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. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arXiv:2304.11277 [cs.DC] https://arxiv.org/abs/2304.11277

[45] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 559–578.