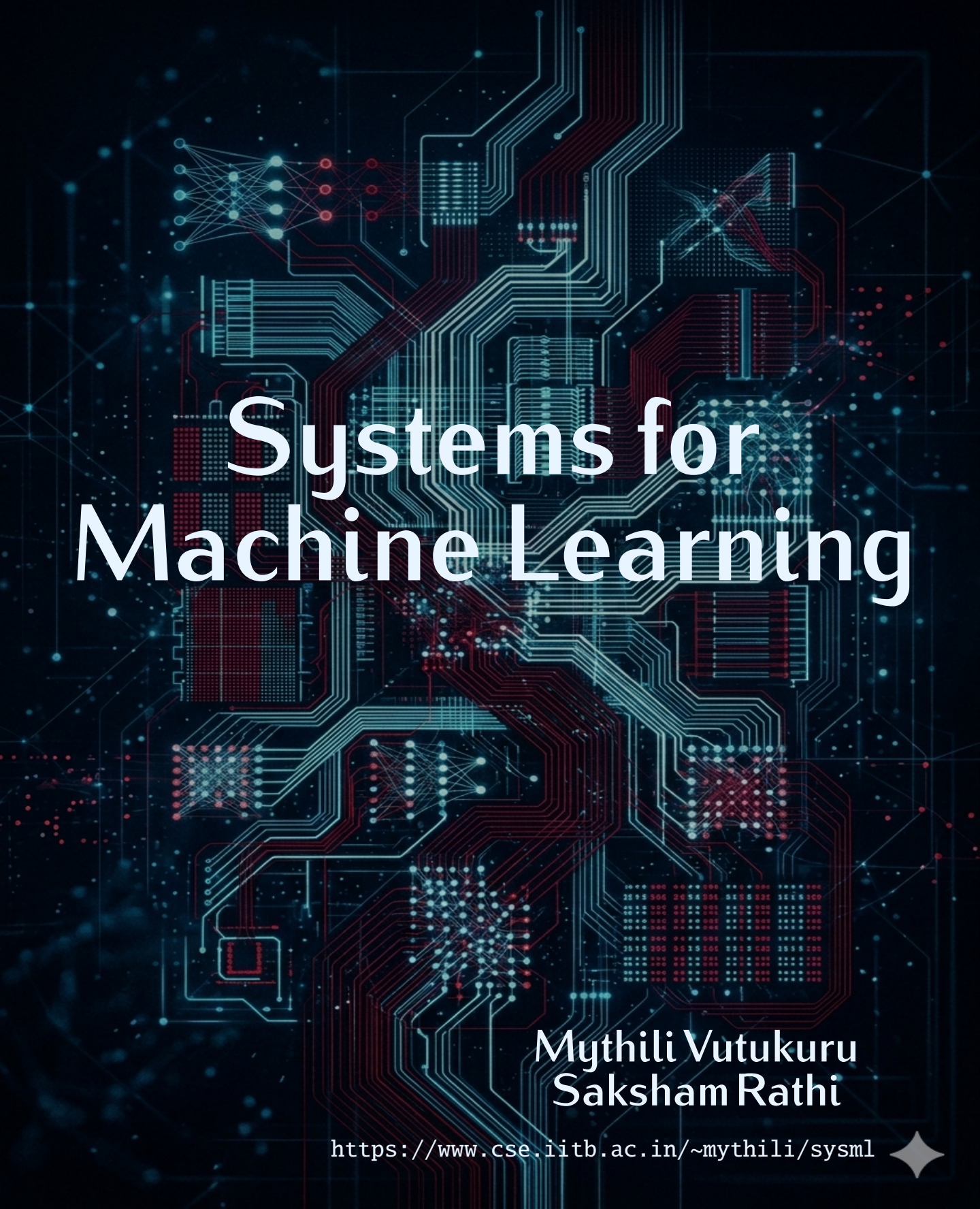

Systems for Machine Learning
Mythili Vutukuru
Saksham Rathi
https://www.cse.iitb.ac.in/~mythili/sysml

DRAFT CHAPTER

August 2026

6

Distributed Training

So far, we have looked at the building blocks that make deep learning systems work, from the core concepts to the hardware and software infrastructure that lets us run the models efficiently. All of the discussion so far, however, shared one quiet assumption: that the model and its training data can fit comfortably within a single GPU. This assumption holds for smaller models, but it breaks down completely once we turn to the large language models (LLMs) and other large networks that comprise much of modern deep learning. These models can have billions or even trillions of parameters, and they are trained on datasets that run into terabytes of text, images, or other data. No single GPU, and often no single node with several GPUs, has enough memory or compute to train such a model in a reasonable amount of time. In this chapter, we study distributed training, the collection of techniques used to distribute the work of training a large model across many GPUs.

Training a model involves a forward pass, where an input is passed through a sequence of layers to produce an output, and the backward pass, where the error in the output is propagated backward through the network to compute gradients, which are then used by an optimizer to update weights (Figure 6.1). The activations produced during the forward pass cannot be discarded immediately, because they are needed again during the backward pass to compute gradients. Keeping these activations in memory until the backward pass is one of the reasons why training consumes more memory than simply running inference on a trained model. Finally, in addition to the model parameters, activations, and gradients, memory is also required for the state maintained by the optimizer, which is usually a few bytes per parameter, depending on the exact optimizer used.

In practice, training is almost never performed using one input at a time. Instead, a batch of inputs is processed together, and their gradients are accumulated across the entire batch before a single weight update is made. The choice of batch size has two competing consequences that we must weigh carefully. On one hand, a larger batch size means that each optimizer step makes use of more data, which typically reduces the total number of steps needed to reach a given level of training progress, and can therefore make training faster in terms of wall-clock time. On the other hand, a larger batch size also means that more activations must be held in memory at once, since every input in the batch contributes its own set of activations that must survive until the backward pass. This creates a direct trade-off between the speed of convergence and the memory footprint of training, and this trade-off will resurface repeatedly as we discuss how large models are trained.

Figure 6.2 shows a real trace of memory usage over the course of several training steps. We see

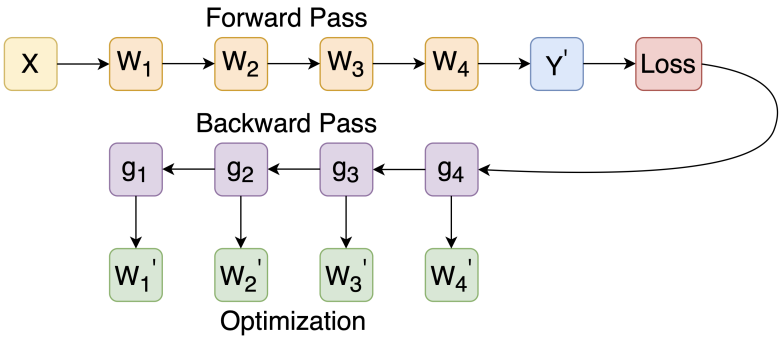


Figure 6.1: Training

that the various components that consume memory rise and fall in a rhythmic pattern: activation memory grows during the forward pass and is released as the backward pass consumes it, gradient memory grows during the backward pass and is released once the optimizer step is complete, while the memory used by the parameters and the optimizer state, once it has taken its first step, remains present throughout training persistently. Understanding this pattern is essential, because it tells us precisely which components of memory grow with batch size, which grow with model size, and which are fixed, and this understanding allows us to reason clearly about why a model might fail to fit on a single device.

Once a model becomes large enough to exceed the capacity of a single GPU — whether because it has a large number of parameters, or because it is trained on a big dataset divided into large batches — training must be distributed across many GPUs, and often across many GPU nodes. The largest modern language models used today are being trained across thousands of GPUs running continuously for weeks at a time, as we will see later in this chapter. Now, given that we have many GPUs available, how should the work of training be divided among them? This chapter describes two foundational strategies, which will be developed in great depth in the sections that follow. The first is data parallelism, in which the training data is split into chunks, with each GPU holding a full copy of the model and training on its own portion of the data, before the resulting gradients are combined across many GPUs. The second is model parallelism, in which the model itself, rather than only the data, is split across GPUs, so that no device needs to hold the entire model in memory at once. As we will see, these two strategies address different bottlenecks: data parallelism primarily addresses the problem of compute and training time, while model parallelism primarily addresses the problem of memory, and in practice, training the largest models combines both strategies, along with several refinements to each, to make training feasible at all. This chapter discusses each of these strategies in turn, examining their mechanism, optimizations, and systems implications.

Note that while we focus on techniques to distribute training across devices, the same techniques

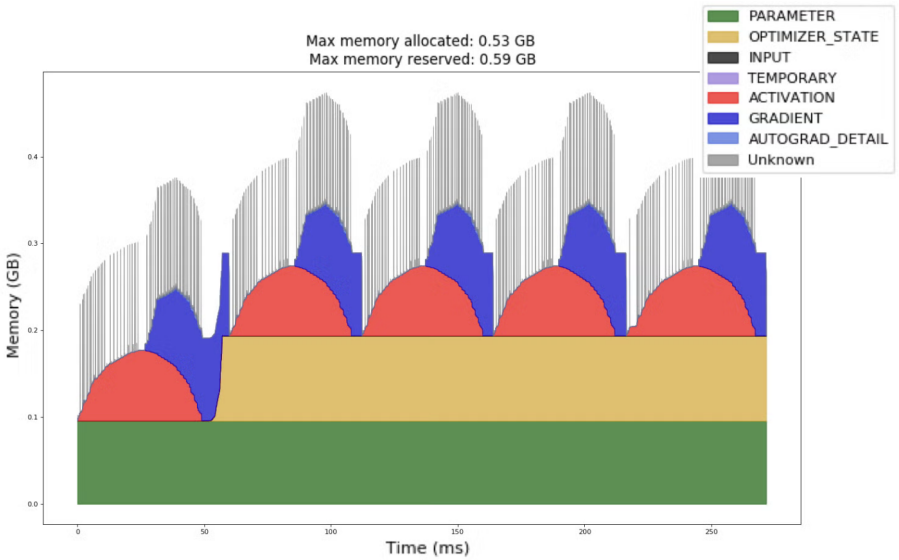


Figure 6.2: Memory usage during training; Image credit: [Shi and DeVito, 2023]

generally apply to distributing inference across multiple GPUs as well. However, because inference is generally simpler than training — involving no backward pass or associated overheads — we do not discuss distributed inference specifically; the techniques discussed for distributed training mostly apply to inference too.

6.1 Data Parallelism

We now turn to the first of the two foundational strategies for distributed training across many GPUs: data parallelism (DP). Data parallelism is, in many ways, the most natural strategy to reach for, since it does not require us to change the model at all. Instead, it changes only how much data each device sees and how often devices talk to one another. Despite this simplicity, data parallelism raises a number of subtle questions about batch size, memory, and communication that we will walk through carefully in this section.

The idea behind data parallelism is straightforward. We take the full training dataset and split it into chunks, one chunk per GPU. Each GPU is then given a complete, identical copy of the model, and each GPU performs the forward pass and the backward pass independently on its own chunk of data. Because every GPU is running its own copy of the same model on different data, the gradients computed on different GPUs will, in general, be different from one another, since they reflect different inputs. Before the model weights can be updated, these gradients must be combined, typically by averaging them across all the GPUs, so that every copy of the model receives the same update and remains identical to every other copy. This averaging step is what keeps the many replicas of the model synchronized with each other throughout training and makes data parallelism work (Figure 6.3).

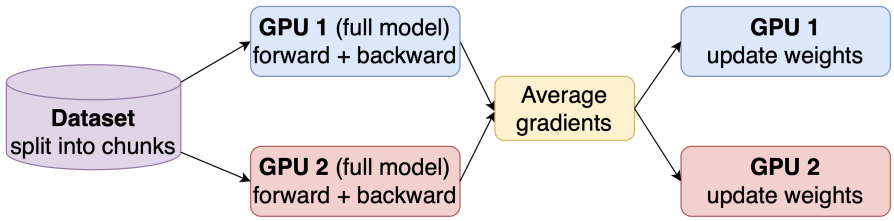


Figure 6.3: Data parallelism

We refer to the number of GPUs across which this replication and splitting takes place as the degree of data parallelism. If we use a data parallel degree of eight, for instance, we mean that eight identical copies of the model are running simultaneously, each on its own subset of the data, with their gradients averaged together at every step. Increasing the degree of data parallelism increases the amount of data processed at every step, and, as we will see shortly, this has direct consequences for both training speed and the amount of communication required.

In ordinary mini-batch stochastic gradient descent, we already compute the gradient by averaging over a batch of examples rather than using a single example. Data parallelism simply distributes the computation of this average gradient over a batch across multiple devices: each device computes a partial sum of gradients over its own mini-batch (Figure 6.4), and the final average gradient used for the update is the average of these partial averages, weighted appropriately. Because the mathematics

of the update remains an average over the same underlying data, data-parallel training preserves the statistical properties of ordinary gradient descent. This is an important point, because it means that, at least in principle, data parallelism does not change what the optimizer is doing, only how the work of computing the gradient is divided among machines.

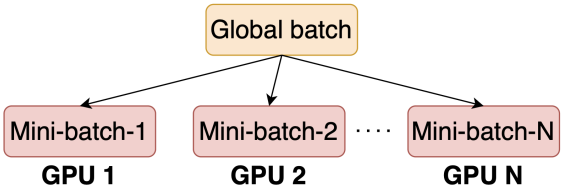


Figure 6.4: Global batch divided into multiple mini-batches across many GPUs

This benefit does not come for free. In order to average gradients across GPUs, the GPUs must communicate with one another at every training step, and the amount of data that must be exchanged grows both with the number of parameters in the model and with the degree of data parallelism. For today's large models, which can have many billions of parameters, this communication is far from negligible, and requires some engineering effort and optimization to work well in practice.

6.1.1 Choice of Batch Size

We first answer a basic question with data parallelism: how large a mini-batch should each GPU process before the gradients are combined? A small mini-batch size means that each GPU finishes its forward and backward passes quickly, but it also means that gradients must be averaged more frequently, and more frequent communication translates into more communication overhead relative to the amount of useful computation being done. Further, a full pass through the global batch now requires more steps, which increases training time. A small mini-batch also means that each gradient estimate is computed from fewer examples, and is therefore noisier, which can make optimization less stable and can sometimes slow down the convergence of learning. On the other hand, a large mini-batch size reduces the noise in the gradient estimate, and reduces the frequency with which communication must occur relative to the amount of computation performed, but it also increases the memory required to store activations across the mini-batch. Because GPU memory is a fixed and often scarce resource, this places a hard ceiling on how large the mini-batch can get before it overflows GPU memory. In practice, the usual approach is to choose the largest mini-batch size that fits comfortably within the memory of a single GPU, and to treat this as the mini-batch size used at every node. This choice keeps communication overhead manageable relative to computation, while avoiding gradient noise that is unnecessarily high.

What if the largest mini-batch that fits in memory is still smaller than what we would like it to be? A technique called gradient accumulations comes to our rescue. Gradient accumulation lets us simulate a larger batch size than what would otherwise fit in memory, without actually holding

all of the corresponding activations in memory at once. The idea is to break the desired mini-batch into several smaller micro-batches, and to process these micro-batches one after another on the same GPU. For each micro-batch, we run the forward pass and the backward pass exactly as before, but rather than immediately using this gradient to update the weights, we add it into a running total. Only after all of the micro-batches that make up the mini-batch have been processed do we use this accumulated, summed gradient to perform a single weight update (Figure 6.5).

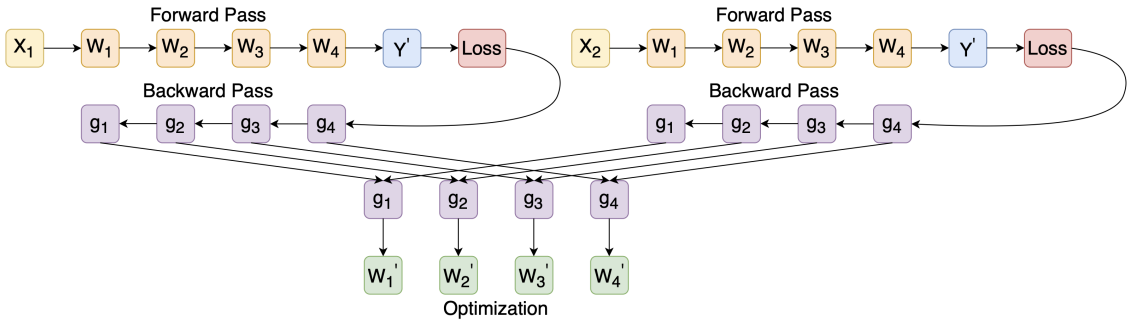


Figure 6.5: Gradient accumulation

The key benefit of this approach lies in reduced memory consumption. Because each micro-batch is processed on its own, only the activations belonging to the current micro-batch need to be held in memory at any one time; the activations of earlier micro-batches have already been consumed by their own backward pass and released before the next micro-batch begins. This means that peak activation memory during gradient accumulation is determined by the size of a single micro-batch, not by the size of the full accumulated mini-batch. As a result, gradient accumulation allows us to reach an effective mini-batch size that is far larger than what the GPU's memory would otherwise permit, simply by choosing to accumulate over more micro-batches.

Since communication of gradients across GPUs only needs to happen once per weight update, and not once per micro-batch, we can use gradient accumulation to reduce the frequency of cross-GPU communication as well. Each GPU accumulates gradients locally over several micro-batches, and only the final accumulated gradient is exchanged and averaged across GPUs. Therefore, much like the degree of data parallelism, the number of gradients accumulated in a mini-batch is also a lever for controlling the trade-off between the size of the effective batch, the memory footprint of training, and the frequency of communication. It is common in practice for large training runs to use both a substantial degree of data parallelism and a number of gradient accumulation steps at each GPU, combining the two to reach very large effective batch sizes that are often used when training large language models.

6.1.2 Synchronizing Gradients with AllReduce

Once each GPU has computed its own gradient, whether from a single mini-batch or from several accumulated micro-batches, these gradients must be combined across all the GPUs participating in data parallelism before the weight update can be applied. The operation that performs this combination is called AllReduce. In an AllReduce, every participating device contributes a tensor, in this case the local gradient, and every device ends the operation holding the same result, which in our case is the sum (Figure 6.6), or equivalently the average, of all the contributed gradients.

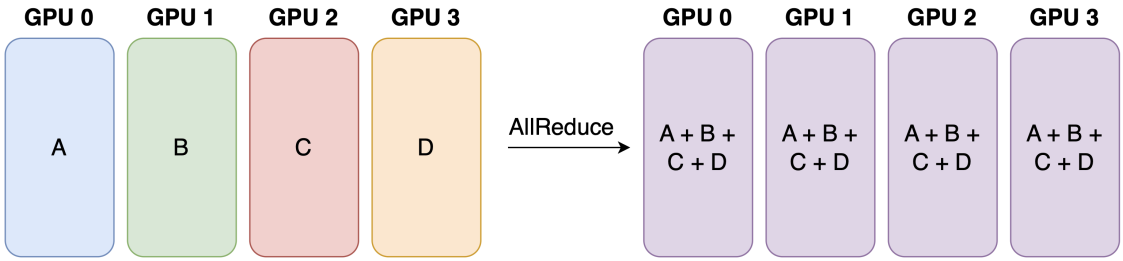


Figure 6.6: AllReduce

AllReduce could, in principle, be implemented in the most naive way imaginable, with every device sending its gradient directly to every other device, so that each device can sum the results it receives locally. This all-to-all approach works, but it is wasteful, since the amount of data sent grows quadratically with the number of devices, and much of this traffic is redundant. In practice, AllReduce is implemented far more efficiently by collective communication libraries, which arrange the communication along a carefully chosen topology so that the all-to-all communication overhead between devices can be avoided.

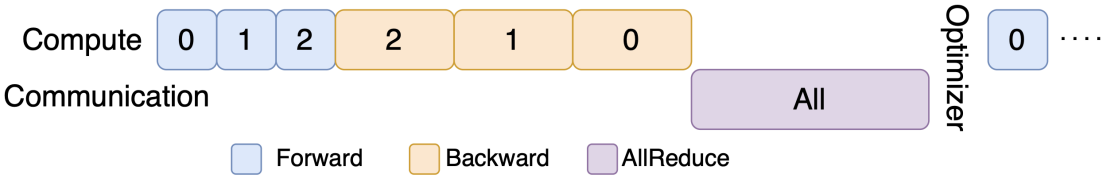
Because the amount of data that must be communicated during an AllReduce depends directly on the number of parameters (i.e., the number of gradient values being summed) and the number of participating GPUs, we can already see why data parallelism carries an inherent communication cost that grows with both model size and degree of data parallelism. This cost cannot be eliminated, but, as we discuss next, it can be substantially hidden behind useful computation, so that its effect on total training time is much smaller than what a naive analysis would suggest.

6.1.3 Overlapping Communication with Computation

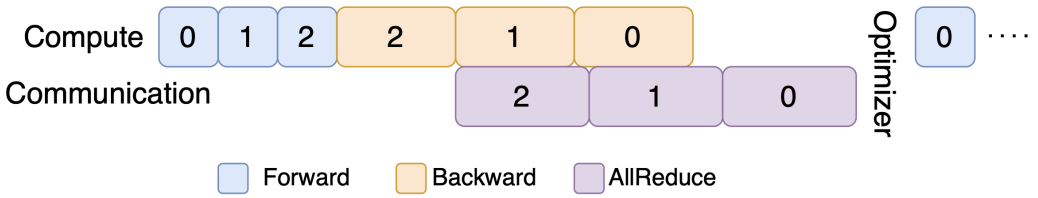
If we implement data-parallel training in the most straightforward way, the sequence of events at every step looks like this: first, the forward pass runs to completion, then the backward pass runs to completion and produces gradients for every layer, then the AllReduce operation runs to completion and produces the synchronized, averaged gradient, and only then does the optimizer update the weights (Figure 6.7a). When implemented this way, the GPU sits idle during the entire AllReduce

operation, since it has no useful computation to perform while it waits for communication to finish. When the model is large, this idle time can be substantial, and directly impacts the overall throughput of training.

A key observation lets us avoid most of this idle time. During the backward pass, gradients are not all produced at the same time, and backpropagation proceeds from the output layer towards the earlier layers, one layer at a time. Now, once the gradient for a given layer has been computed, there is no reason to wait for the rest of the backward pass to finish before beginning to communicate that gradient; the corresponding chunk of AllReduce can begin immediately, while the backward pass continues computing gradients for earlier layers. This technique is often called wait-free backpropagation, or overlapped gradient communication, and it allows the communication for the gradients of later layers to proceed concurrently with the computation of gradients for earlier layers, rather than waiting until the entire backward pass has finished (Figure 6.7b).



(a) Data-parallel training without wait-free backpropagation



(b) Data-parallel training with wait-free backpropagation

Figure 6.7: Wait-free backpropagation

In practice, this overlap is implemented by issuing the AllReduce for each layer, or for a group of layers, as an asynchronous operation as soon as that layer's gradient becomes available, rather than issuing a single AllReduce for the entire model's gradient after the backward pass has fully completed. Because modern GPUs and their network interfaces can perform computation and communication simultaneously, using separate hardware resources for each, this overlap can hide a substantial fraction of the communication cost of AllReduce, and make data-parallel training work efficiently for a reasonable number of data-parallel GPUs.

6.1.4 Limits to Scaling Data Parallelism

Given that data parallelism is conceptually simple and preserves the statistical behaviour of ordinary gradient descent, it is natural to ask whether we can simply keep increasing the degree of data parallelism indefinitely to speed up training. In practice, this does not work, and data parallelism scales well only up to some limit, often somewhere in the range of a few dozen GPUs, depending on the model, the hardware, and the network. As the degree of data parallelism increases, so does the amount of gradient data that must be summed across GPUs at every step, and, even with an efficient AllReduce implementation and overlap between computation and communication, there comes a point at which communication can no longer be fully hidden behind computation. Beyond this point, adding more GPUs to the data-parallel group increases the communication burden faster than it increases useful computation, and the marginal benefit of each additional GPU shrinks, sometimes to the point where adding more GPUs actually slows training down rather than speeding it up.

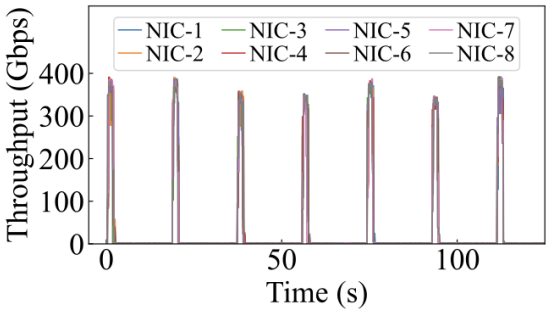


Figure 6.8: NIC egress traffic pattern during production model training; Image credit: [Qian et al., 2024]

This effect is compounded by the bursty nature of the communication pattern in data-parallel training. Because AllReduce operations are concentrated around the backward pass and the weight update, network traffic during data-parallel training tends to arrive in short, intense bursts rather than as a steady stream, as illustrated by measurements taken from production training clusters, where network throughput spikes sharply at regular intervals corresponding to these synchronization points (Figure 6.8). This bursty traffic pattern makes it considerably harder to design data center networks that can absorb the resulting load efficiently, since the network must be provisioned to handle brief periods of very high demand, even though average utilization may be much lower.

For these reasons, data parallelism alone is not sufficient to train the very largest models efficiently at the scale of thousands of GPUs. Instead, we combine a moderate degree of data parallelism, chosen to stay within the regime where the communication overhead can still be managed, with other forms of parallelism that split the model itself across devices in several ways.

Practice Problem 6.1

Consider a training setup with a dataset of 4 million (4M) tokens. You are given the following memory figures for a single GPU with **40** GB of HBM:

Component	**Memory**
Model parameters (fp16)	14 GB
Optimizer states (fp32)	18 GB
Gradients (fp16)	4 GB
Total model states	**36** GB

Activations are stored in fp16 and consume approximately **25** KB (25,000 bytes) per token per layer, and the model has **32** layers. Intermediate buffers (KV cache, etc.) add a fixed overhead of **0.8** GB. Assume 1 GB = 10^9 bytes and 1 KB = 10^3 bytes for memory allocation calculations.

(i) Based on the remaining available HBM on a single GPU, what is the largest mini-batch size (in number of tokens) that can be processed in a single training step at a single GPU without running out of memory (OOM)?

Solution: First, find the remaining memory available for activations:

$$\text{Available Memory} = 40\,\text{GB} - 36\,\text{GB(model states)} - 0.8\,\text{GB(overhead)} = 3.2\,\text{GB} = 3{,}200{,}000\,\text{KB}$$

Next, calculate the activation memory required per token across all 32 layers:

$$\text{Memory per token} = 25\,\text{KB/token/layer} \times 32\,\text{layers} = 800\,\text{KB/token}$$

Divide the available memory by the memory per token to find the maximum batch size:

$$\text{Max Batch Size} = \frac{3{,}200{,}000\,\text{KB}}{800\,\text{KB/token}} = 4{,}000 \text{ tokens (4K tokens).}$$

(ii) How many training steps are required to complete one epoch (i.e., to process the entire 4M-token dataset) on a *single* GPU?

Solution: Each step processes 4K tokens on one GPU. To cover 4M tokens:

$$\text{Steps} = \frac{4{,}000{,}000}{4{,}000} = 1{,}000 \text{ steps.}$$

(iii) Now suppose we employ Data Parallelism (DP) with a DP degree of $D = 32$, meaning 32 GPUs each independently process a different batch of 4K tokens per step, and their gradients are aggregated after every step. How many steps are now required to complete one epoch?

Solution: With 32 GPUs each processing 4K tokens per step, the effective batch size per step is $32 \times 4{,}000 = 128{,}000$ tokens. Therefore:

$$\text{Steps} = \frac{4{,}000{,}000}{128{,}000} = \frac{1{,}000}{32} \approx 31.25 \implies 32 \text{ steps.}$$

Equivalently, the number of steps is reduced by a factor equal to the DP degree.

(iv) To stabilize training, you decide to scale up to a target global batch size of **512,000** tokens using the same 32 GPUs. To avoid OOM errors, each GPU must maintain its hardware limit of tokens per micro-batch computed earlier, while utilizing gradient accumulation — a technique where gradients are calculated over multiple micro-batches and summed together before a single weight update is performed. How many gradient accumulation steps are required per parameter update?
Solution: One cluster-wide micro-step processes $32 \times 4{,}000 = 128{,}000$ tokens. To reach the target global batch size of 512,000 tokens, the number of gradient accumulation steps required is:

$$\text{Gradient Accumulation Steps} = \frac{\text{Global Batch Size}}{\text{Tokens per Micro-batch}} = \frac{512{,}000}{128{,}000} = 4 \text{ steps.}$$

(v) Briefly explain the key trade-offs introduced by increasing the DP degree D.
Solution: A higher DP degree reduces the number of training steps per epoch proportionally (by a factor of D), directly reducing wall-clock training time, assuming compute is not the bottleneck.
The primary cost involved is the communication overhead. After each step, an AllReduce over all D GPUs must complete before the next step begins. As D grows, the volume of gradient data exchanged and the number of participating nodes both increase, which raises synchronization latency and becomes a bottleneck that limits scaling efficiency.

Programming Assignment 6.1: Data Parallelism

In this assignment, you will implement data-parallel training with a simulated AllReduce to synchronize gradients across multiple GPUs rather than splitting the model itself. You're training replicas of a two-layer MLP by having each simulated GPU, launched as its own worker process, sample a private mini-batch and independently run a full forward and backward pass to compute a local gradient using its own copy of the model. Workers send their local gradients to a coordinator through a shared `mp.Queue`, and a central `NetworkThread`'s job is to accumulate all `num_gpus` gradients, average them, and broadcast the averaged gradient back to every worker so that all replicas apply an identical update and remain synchronized. The assignment, along with a detailed `README`, is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/data_parallelism

6.2 ZeRO

As we saw in the previous section, in standard data parallelism, every GPU keeps a full copy of the model parameters, gradients, and optimizer states. If we are training with an optimizer such as Adam, the optimizer state alone, consisting of a running estimate of the first and second moments of the gradient, can be larger than the parameters themselves. Even though each GPU is only responsible for processing its own slice of data with data parallelism, the requirement to hold a full copy of the model at every GPU imposes a significant memory overhead. It also leaves a GPU with very little memory left to hold activations during the forward and backward passes, and limits the effective batch size. To address this issue, modern deep learning models employ an optimization called ZeRO, short for the Zero Redundancy Optimizer.

The central observation behind ZeRO is that not every GPU needs to hold a full copy of every model state at every point in time. For example, parameters are needed in full during the forward and backward pass of a given layer, gradients are needed in full only when the optimizer is about to consume them, and optimizer states are needed only during the optimizer step itself. Between these steps, there is no reason for every GPU to keep its own full copy of model states in memory. ZeRO exploits this observation by partitioning, or sharding, the model states across the GPUs participating in data parallelism, so that each GPU is permanently responsible for storing only a fraction of each model state, roughly $1/D$ of the total, where D is the degree of data parallelism. Whenever the full, unsharded version of a state is actually required for computation, the GPUs temporarily reconstruct it by communicating with one another, use it for as long as it is needed, and then discard it again. In this way, a GPU can keep its memory footprint small, even though it computes the full model.

This is a crucial distinction from the kind of model parallelism we discuss later in this chapter: ZeRO does not change how the computation of the model is divided across devices. Every GPU still runs the forward and backward pass over the entire model, on its own slice of data, exactly as in ordinary data parallelism. What changes is only where the model states physically live between these computations. For this reason, ZeRO is best understood not as a new parallelism strategy in its own right, but as a memory optimization layered on top of data parallelism, one that trades a small amount of additional communication for a large reduction in per-GPU memory.

ZeRO is organized into three cumulative stages, conveniently called ZeRO-1, ZeRO-2, and ZeRO-3, each of which shards a larger set of model states than the one before it. Each stage subsumes the memory savings of the previous one while introducing additional communication of its own, so that moving from ZeRO-1 to ZeRO-3 traces out a trade-off between memory footprint and communication volume (Figure 6.9).

6.2.1 ZeRO-1: Sharding Optimizer States

The first and mildest stage of ZeRO, ZeRO-1, leaves parameters and gradients replicated across all GPUs exactly as in ordinary data parallelism, and shards only the optimizer states. Every GPU still runs the forward pass and the backward pass using a full, unsharded copy of the parameters,

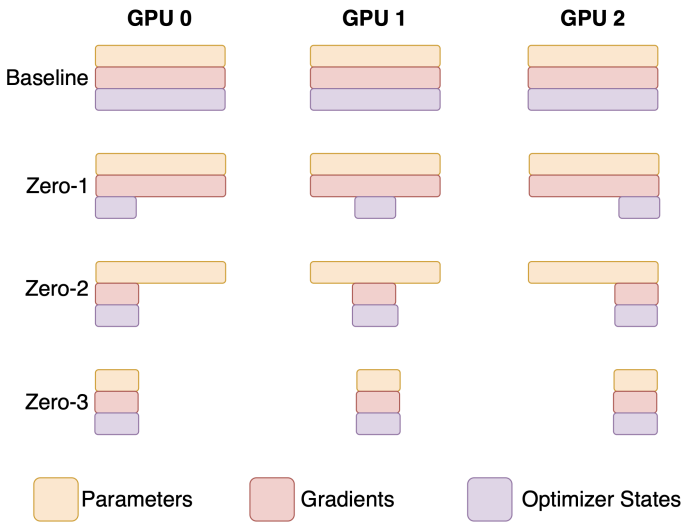


Figure 6.9: GPU memory footprint under baseline data parallelism, ZeRO-1, ZeRO-2, and ZeRO-3

and every GPU still computes a full, unsharded gradient for every parameter in the model during the backward pass. The difference appears only once these full gradients are ready to be consumed. Rather than performing an AllReduce that leaves every GPU holding the complete, averaged gradient, ZeRO-1 (Figure 6.10) performs a ReduceScatter: the gradients are summed across GPUs exactly as before, but the result is scattered, so that each GPU ends up owning only the slice of the summed gradient that corresponds to the shard of the optimizer state it is responsible for (Figure 6.11). Concretely, ReduceScatter can be thought of as an AllReduce followed by a split: the reduction (summation) across GPUs still happens over the entire gradient, but instead of every GPU receiving a full copy of the result, each GPU receives only its own chunk of the gradients, so the gradient output is sharded across the group rather than replicated.

Every GPU then runs the optimizer step using only its own shard of the gradient and its own shard of the optimizer state, producing an updated shard of the parameters. At this point, every GPU holds a different, updated piece of the model, and none of them individually has the full picture. To prepare for the next forward pass, the GPUs perform an AllGather (Figure 6.12) on the updated parameter shards, so that every GPU is once again brought up to date with the complete, unsharded set of parameters. During AllGather, each GPU broadcasts its own shard to every other GPU, and every GPU concatenates the incoming shards to assemble the full model state. Because the optimizer state for an optimizer like Adam is often the single largest of the three model states, being stored in 32-bit precision and typically requiring more memory than the parameters and gradients combined, sharding it alone reduces the memory footprint at each GPU significantly.

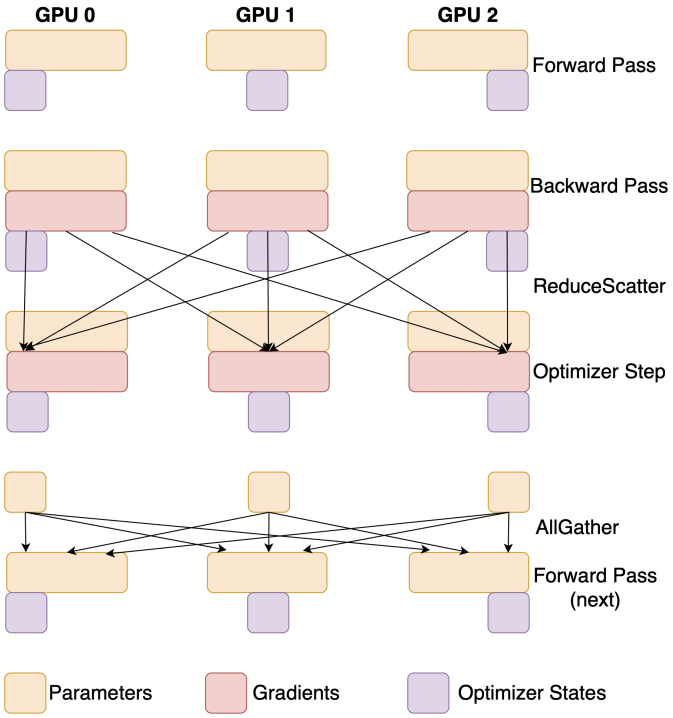


Figure 6.10: ZeRO-1

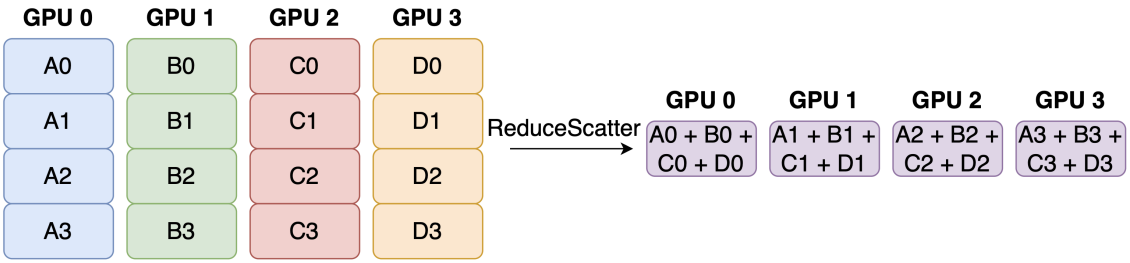


Figure 6.11: ReduceScatter

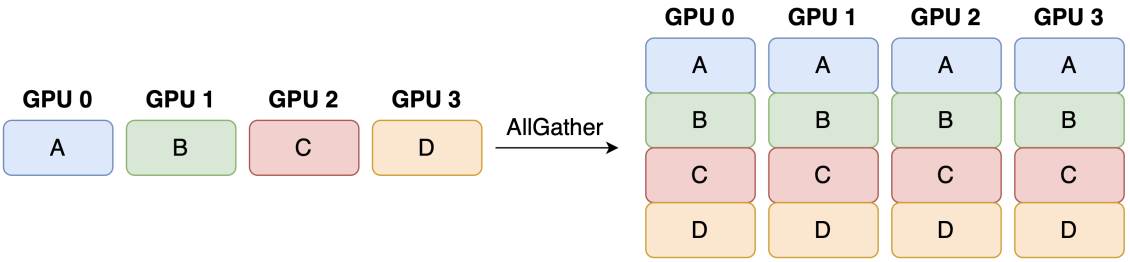


Figure 6.12: AllGather

6.2.2 ZeRO-2: Sharding Optimizer States and Gradients

ZeRO-2 extends the idea behind ZeRO-1 one step further by sharding gradients in addition to optimizer states. As before, every GPU still computes a full gradient during the backward pass, since the backward pass itself is unchanged and still touches every parameter of the model. The difference is that, once the ReduceScatter has produced each GPU's shard of the summed gradient, the full, unsharded gradient tensor is no longer kept around. Each GPU discards the parts of the gradient that are not its own responsibility immediately after the ReduceScatter completes, keeping in memory only the shard it needs to run its own local piece of the optimizer step.

This brings a meaningful change in memory footprint, since gradients, stored in a precision such as fp16, can themselves occupy several gigabytes for a large model, and this memory is now no longer duplicated D times across the data-parallel group. In essence, ZeRO-2's only real change relative to ZeRO-1 is a matter of when memory is freed: the same ReduceScatter that ZeRO-1 already performs is reused, but each GPU now drops the gradient shards it doesn't own immediately afterward instead of holding onto a full copy, so the savings come from discarding data earlier rather than from any new communication pattern. In fact, the pattern of computation and communication after this point is otherwise identical to ZeRO-1: each GPU runs the optimizer using its own gradient shard and optimizer state shard, producing an updated parameter shard, and an AllGather reassembles the full, updated parameters before the next forward pass begins. Because sharding gradients does not require any additional communication between GPUs beyond what ZeRO-1 already performs, ZeRO-2 achieves a strictly larger memory saving than ZeRO-1 at essentially no extra communication cost. For this reason, ZeRO-2 is generally considered the more sensible default of the two.

6.2.3 ZeRO-3: Sharding Optimizer States, Gradients, and Parameters

ZeRO-3 removes the last remaining source of redundancy by sharding the parameters themselves, so that at rest, no GPU holds a complete copy of the model at all. With a data parallelism degree of D, each GPU permanently owns only a $1/D$ slice of every parameter tensor in the model. This is a more invasive change than ZeRO-1 or ZeRO-2, because it means that even the forward pass, which

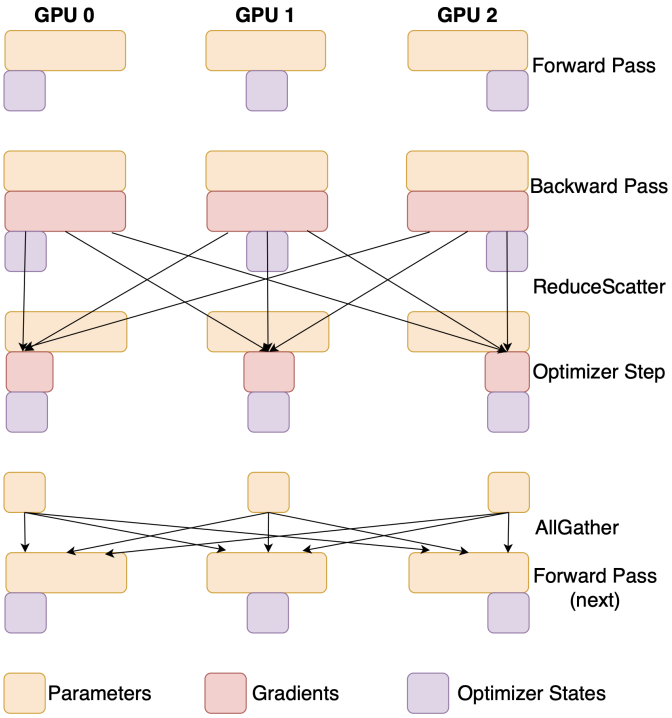


Figure 6.13: ZeRO-2

previously required no special handling, must now be modified. ZeRO-3 performs an AllGather to reconstruct the full parameters of a layer immediately before that layer is used in the forward pass, uses the reconstructed parameters to compute the layer's output, and then immediately discards the unsharded copy, retaining only its own shard once again (Figure 6.14).

The same pattern repeats during the backward pass. Before the gradient of a given layer can be computed, that layer's full parameters must again be assembled through an AllGather, since backpropagation through a layer requires its complete weight matrix just as the forward pass did. Once the local gradient contribution for that layer has been computed, it is reduced across GPUs and scattered, exactly as in ZeRO-2, so that each GPU ends up holding only its own shard of the gradient, and the temporarily reconstructed parameters are released once again (Figure 6.15).

The result is that, at any given instant, a GPU running ZeRO-3 needs to hold in memory only its own permanent shard of parameters, gradients, and optimizer states, plus the temporarily reconstructed, full-sized parameters of whichever single layer is currently being computed. This is a significant reduction in memory usage as compared to the baseline, where a GPU must hold the

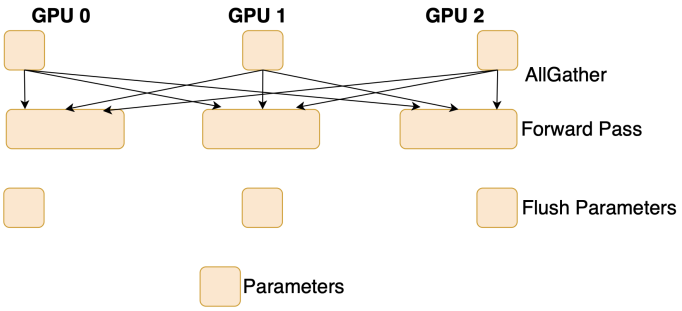


Figure 6.14: ZeRO-3 forward pass

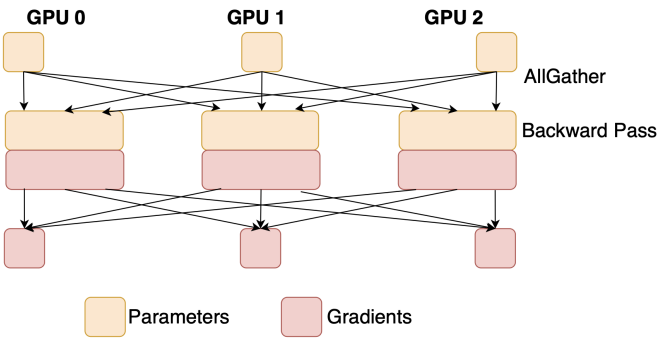


Figure 6.15: ZeRO-3 backward pass

complete model states for every layer simultaneously. This is what allows ZeRO-3 to train models whose total parameter count would not come close to fitting in the memory of any single GPU, since no GPU is ever asked to materialize more than one layer's worth of unsharded parameters at a time.

However, this memory saving does not come for. Whereas ZeRO-1 and ZeRO-2 each perform one round of collective communication per training step, ZeRO-3 must perform an AllGather of parameters before essentially every layer's forward pass and again before every layer's backward pass. This means the number of collective communication operations grows with the number of layers in the model, not just once per training step, and the total volume of data communicated per step is also correspondingly higher than in ZeRO-1 or ZeRO-2. Because of this, ZeRO-3 tends to place a heavier burden on the interconnect between GPUs, and its performance is much more sensitive to network bandwidth, and to how well this additional communication can be overlapped with computation.

6.2.4 Overlapping Communication with Computation in ZeRO

Just as with ordinary data parallelism, the naive execution of ZeRO would leave GPUs idle whenever they are waiting on a collective communication operation to finish. For ZeRO-1 and ZeRO-2, the relevant overlap opportunity is between the ReduceScatter of gradients and the backward pass itself. Because backpropagation computes gradients layer by layer, from the output layer of the network toward the input, the gradient for a later layer becomes available before the gradient for an earlier layer. As soon as a layer's gradient has been computed, its contribution to the ReduceScatter can begin immediately, proceeding concurrently with the computation of gradients for earlier layers, in much the same wait-free fashion described earlier for plain data parallel AllReduce (Figure 6.16).

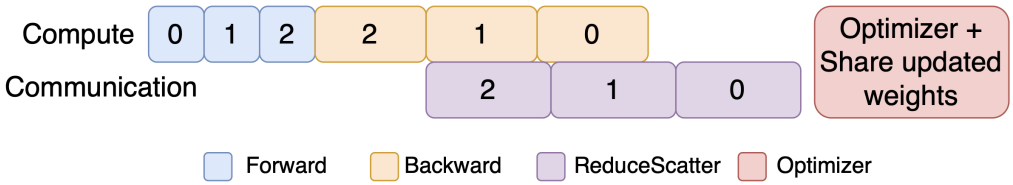


Figure 6.16: Overlapping the ReduceScatter of ZeRO-1 and ZeRO-2 with backward-pass computation

ZeRO-3 offers a richer set of opportunities for overlap, precisely because it communicates so much more frequently. The AllGather needed to reconstruct a layer's parameters before its forward pass can be issued one layer ahead of time, so that the parameters for layer $k+1$ are fetched over the network while layer k is still being computed, hiding the latency of the fetch behind useful work. The same idea applies in reverse during the backward pass. Even so, because ZeRO-3 issues so many more collective operations than ZeRO-1 or ZeRO-2, communication overhead is harder to fully hide, and in practice, some stalling is inevitable, unless the interconnect is fast relative to the compute time of a single layer (Figure 6.17). One further optimization sometimes used with ZeRO-3 is to avoid releasing a layer's reconstructed parameters immediately after the forward pass, keeping them resident in memory if enough capacity is available, so that the AllGather does not need to be repeated when the same layer is visited again during the backward pass.

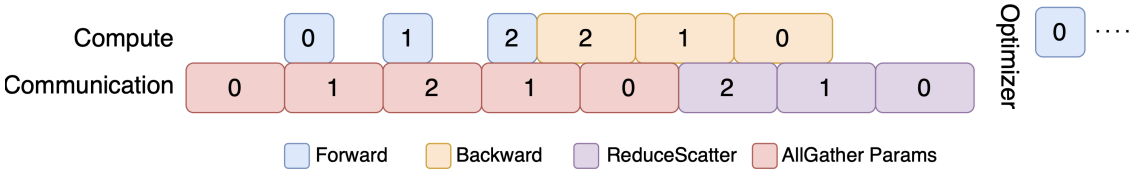


Figure 6.17: Overlapping the AllGather of parameters in ZeRO-3 with forward and backward computation

Taken together, the three stages of ZeRO trace out a clear progression. ZeRO-1 shards only optimizer states, offering a large memory saving relative to its cost, since it is comparable in cost of communication with plain data parallelism. ZeRO-2 sharding gradients in addition to optimizer states costs nothing further in communication and strictly increases the memory saved, which is why, in practice, ZeRO-2 is almost always preferred over ZeRO-1 whenever both are available. ZeRO-3 removes the final piece of redundancy by sharding parameters themselves. This unlocks the ability to train models whose full size would not fit on a single GPU, but this comes at the cost of substantially more frequent communication, since parameters must now be reconstructed layer by layer during both the forward and backward pass. It is worth noting that the memory freed up by the ZeRO optimizations is not only useful for fitting larger models on GPUs, but it can also be spent on increasing the per-GPU batch size and hence reduce overall training time. However, a larger per-GPU batch introduces its own trade-off, since a larger batch size typically means more compute and communication per step as well, and the net gains depend on how the costs and benefits stack up relative to each other. The right variant of ZeRO to use in practice depends on what the limiting factor is. If a model and its gradients comfortably fit in the memory of a GPU but not the optimizer state, then ZeRO-2 is usually sufficient and preferable, since it avoids the additional communication that ZeRO-3 requires. Only when the parameters themselves are too large to hold in full on a GPU does the extra communication cost of ZeRO-3 become worth paying for.

6.2.5 ZeRO in Practice: PyTorch FSDP

PyTorch's Fully Sharded Data Parallel, or FSDP, is a popular realization of the ZeRO idea, and essentially implements the ZeRO-3 strategy discussed above, though the terminology differs slightly. In FSDP, the model is organized into a set of units, typically corresponding to individual layers or small groups of layers, and parameters are sharded across the data-parallel group at the granularity of these units rather than for the model as a whole (Figure 6.18).

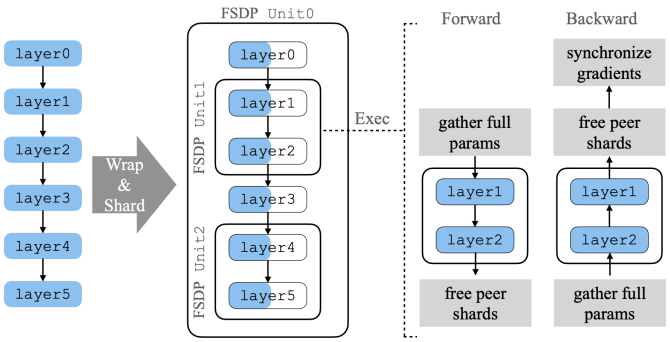


Figure 6.18: PyTorch FSDP; Image credit: [Zhao et al., 2023]

During the forward pass, FSDP gathers the full, unsharded parameters of a unit immediately before that unit executes, exactly as described for ZeRO-3, and frees the peer shards it does not own once the unit's computation is finished. The backward pass mirrors this: full parameters for a unit are gathered again to compute its gradient, and once the local gradient has been produced it is reduced and synchronized across the data-parallel group before the corresponding parameter shards are freed. Because of this design, the peak memory required by a single GPU under FSDP is not proportional to the size of the whole model, but rather to the size of that GPU's own permanent parameter shard plus the size of the single largest unsharded unit it must ever materialize at once. This allows FSDP to train models with far more parameters than would fit on any individual GPU.

Practice Problem 6.2

Answer the following questions based on the ZeRO methods of reducing memory footprint during data parallelism. Recall that ZeRO-1 shards only optimizer states, ZeRO-2 shards optimizer states + gradients, and ZeRO-3 shards all model state (parameters, gradients, optimizer states) across all data-parallel nodes.

(i) Consider a 4-billion model, where the parameters and gradients require 2 bytes per element, and the optimizer states occupy 6 bytes per parameter. The degree of data parallelism is 8. What is the amount of memory required to store model states without any ZeRO optimizations, and with each of the three variants of ZeRO? We are expecting four answers below.

Solution: In the no-ZeRO case, every GPU stores a full replica of all model states, which amounts to $4GB \times (2 + 2 + 6) = 40GB$ (accounting for parameters, gradients, and optimizer states).
ZeRO-1 shards optimizer states, so $4GB(2 + 2 + \frac{6}{8}) = 19GB$.
ZeRO-2 shards optimizer states, and gradients, so $4GB(2 + \frac{2+6}{8}) = 12GB$.
ZeRO-3 shards optimizer states, gradients and parameters, so $4GB(\frac{2+2+6}{8}) = 5GB$.

(ii) Consider the ZeRO-2 implementation of the 4-billion model mentioned above which is arranged as a 3-layer neural network on a GPU. The forward and backward passes of each layer takes 1 ms (millisecond) and 2 ms respectively for a batch of inputs. The ReduceScatter step takes 10 ms per layer each, and the optimizer step (along with propagating updated weights) takes another 10 ms overall. Assume that compute can be overlapped with the network communication wherever possible. What is the minimum time required to process one batch of inputs on this GPU in ZeRO-2 mode?

Solution: The parameters are unsharded, so no communication needed and forward pass takes 3ms (in total). The ReduceScatter of layer l can be overlapped with the backward pass of layer $l-1$. Layer 3 BP runs from $t = 3ms$ to $t = 5ms$, and layer 3's ReduceScatter can start immediately at $t = 5ms$. Total time taken for FP, BP, and RS is therefore, 35ms. The optimizer step takes another 10ms. Therefore, minimum time required in ZeRO-2 mode is 45ms (Figure 6.19)

(iii) Repeat the above question for the ZeRO-3 optimization mode. You may assume that unsharded parameters are not freed up after the forward pass, so that parameters need to be assembled only once at the start of the forward pass. Assume that the time taken to do this AllGather operation is 10 ms per layer.

Solution: Parameters are sharded, therefore, they must be gathered before the forward pass of each layer. So, total FP time is 31 ms (AllGather of the next layer overlapping with FP of the previous layer). BP + RS take 32ms, as before. Therefore, including the optimizer step, the total time required is 73ms (Figure 6.20)

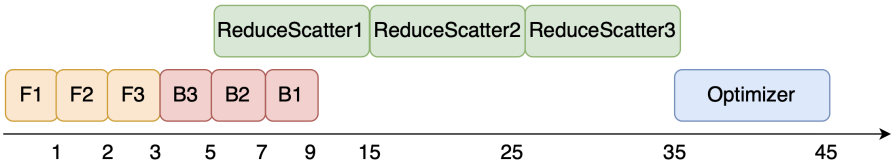


Figure 6.19: ZeRO-2

Figure 6.20: ZeRO-3

(iv) Let's say you are given that processing a single input requires 0.8GB of GPU memory for activations. The GPU has a total of 80GB memory to store all model states and activations. What is the maximum batch size that can be supported for data-parallel training on this GPU, both with and without ZeRO-3 optimizations?
Solution: Without ZeRO-3, the space remaining for activations is $80 - 40 = 40$ GB. Therefore, the max batch size will be 50.
With ZeRO-3, the space remaining for activations is $80 - 5 = 75$ GB. Max batch size is 93.

(v) Now, you must come up with a quantitative justification of whether ZeRO-3 optimization is beneficial for overall throughput or not. You must consider the benefit of increase in supported batch size with ZeRO-3 and weigh it against the cost of increase in batch processing time due to additional communication overheads. You may assume that all timings provided above (for forward and backward passes, for the optimizer step, and for all communication primitives like ReduceScatter and AllGather) stay the same irrespective of batch size. You must state whether ZeRO-3 is beneficial for improving training throughput, and provide a suitable explanation.
Solution: Without ZeRO, no collective communication is required. The forward pass takes $3 \times 1 = 3$ ms, the backward pass takes $3 \times 2 = 6$ ms, and the optimizer step takes 10 ms, giving a total batch processing time of 19ms. With a maximum batch size of 50, the throughput is: $\frac{50}{19} \approx 2.63$ samples/ms. With ZeRO-3, from the previous part, the total batch processing time is 73ms. With a maximum batch size of 93, the throughput is: $\frac{93}{73} \approx 1.27$ samples/ms. Therefore, ZeRO-3 is not beneficial for overall throughput in this setting. Although ZeRO-3 enables a significantly larger batch size, the additional communication overhead from AllGather operations during the forward pass increases the batch processing time by a larger ratio, lowering the overall throughput.

6.3 Pipeline Parallelism

Data parallelism, discussed in the previous section, is an effective strategy as long as the entire model, or at least a shard of it together with its optimizer state, fits comfortably in the memory of a single accelerator. Modern neural networks, however, routinely exceed the memory capacity of a single GPU. When this happens, replicating the model across workers is simply not an option, and a different strategy is required: rather than replicating the model and splitting the data, we instead split the model itself and let each worker hold only a part of it. This approach is called model parallelism (Figure 6.21).

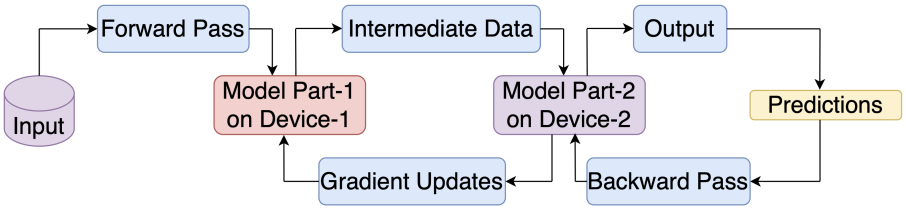


Figure 6.21: Model parallelism

The simplest form of model parallelism partitions the model layer-wise into contiguous groups of layers, called stages, and assigns each stage to a different device. During the forward pass, the input travels through the stages in sequence: the first device computes its portion of the network and sends the resulting intermediate activations to the second device, which continues the computation, and so on until the final device produces the output and the loss is computed. During the backward pass, gradients flow in the reverse order, from the last stage back to the first, with each device computing gradients for its own parameters and propagating the loss gradients back to the previous stage. Each device then updates the weights of its stage using the gradients it has computed locally.

This scheme directly addresses the memory capacity problem: since each device is responsible for only a fraction of the layers, the memory required to store parameters, activations, and optimizer state at any one device is a corresponding fraction of what would be needed to hold the full model. It is also attractive from a communication standpoint for very large models, because instead of exchanging the entire set of weights or gradients across all workers, as data parallelism would require, devices only need to exchange the (much smaller) intermediate activations and gradients that cross stage boundaries.

6.3.1 Limitations of Naive Model Parallelism

Although model parallelism solves the memory problem, a naive implementation of it is inefficient in terms of compute utilization. Because the stages of the model form a strict producer-consumer chain, a device cannot begin its forward computation on a batch until it has received the intermediate activations from the previous device, and it cannot begin its backward computation until it has

received the gradient from the following device. If there is only a single batch being processed at a time, then at any given instant only one device in the entire pipeline is actually doing useful work, while every other device sits idle waiting for its input to arrive (Figure 6.22).

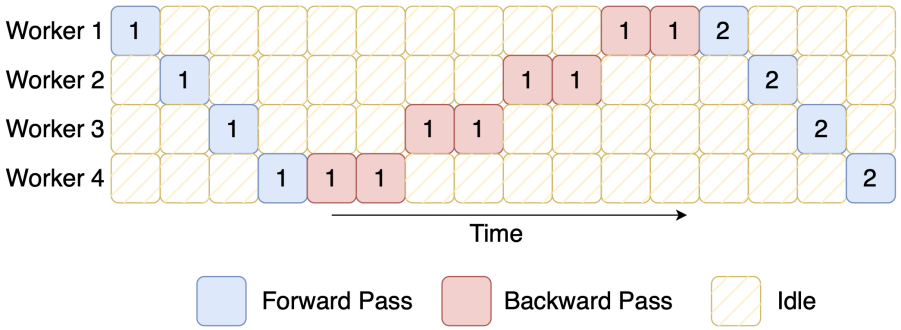


Figure 6.22: Limitations of naive model parallelism

This idle time is commonly referred to as a bubble. If we imagine plotting the activity of each worker over time, the bubble appears as a large block of unused time before a device can start its forward pass, and again before it can start its backward pass. As the number of pipeline stages P grows, so does the size of this bubble, since the depth of the dependency chain that a single batch must traverse before returning to the first device grows linearly with P. When the model is partitioned across many stages, naive model parallelism can leave the majority of the accelerators idle at any given moment, defeating the purpose of distributing the computation in the first place.

6.3.2 Pipeline Parallelism: The GPipe Schedule

Pipeline parallelism (PP) is a technique to make model parallelism more efficient. The key idea is to keep the same layer-wise partitioning of the model across devices, but to reduce the bubble by feeding the pipeline with more than one unit of work at a time. Instead of pushing an entire mini-batch through the pipeline as a single unit, we split the mini-batch into a number of smaller micro-batches and inject them into the pipeline one after another. This way, different devices in the pipeline can be simultaneously processing different micro-batches at different stages, leading to better overall utilization of the compute. This is exactly analogous to an assembly line in manufacturing: once the pipeline is full, every station is busy at every point in time, even though any single item still has to pass through every station sequentially.

This scheduling idea was proposed in the GPipe system [Huang et al., 2019]. In GPipe, all M micro-batches of a mini-batch first flow forward through the pipeline, and only once every micro-batch has completed its forward pass do the backward passes begin, again for each micro-batch in turn. After all M backward passes have completed, the accumulated gradients from every micro-batch are summed and applied in a single synchronous weight update, and the pipeline begins processing

the next mini-batch again. This point, at which all in-flight micro-batches have been drained from the pipeline and the weights are updated, is referred to as a pipeline flush (Figure 6.23). Note that the backward pass is shown as taking twice as long as the forward pass for each micro-batch in the example below, and this is an assumption we will follow throughout this section. While this assumption is not a requirement for the ideas presented in this section to work, the backward pass does involve more compute than the forward pass in general, and is hence assumed to be the case in most examples in related work.

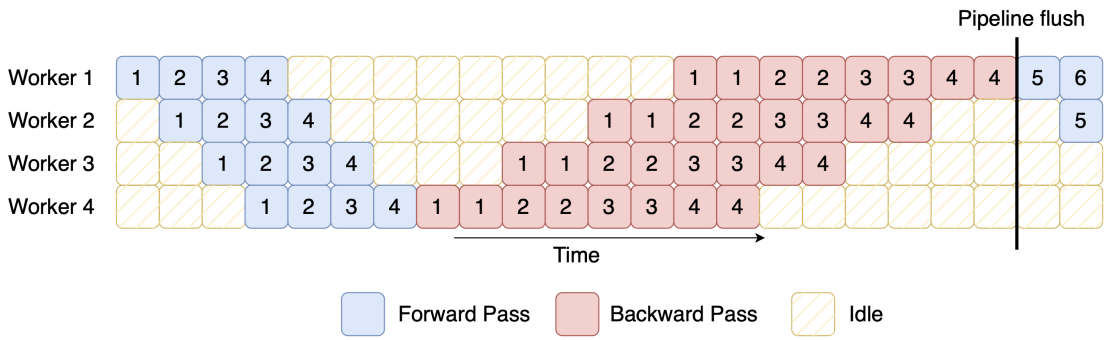


Figure 6.23: GPipe

Splitting a mini-batch into micro-batches substantially reduces the size of the bubble relative to naive model parallelism, because the pipeline stays busy while many micro-batches are in flight, and idle time is confined to the beginning of the pipeline (filling it with the first few micro-batches) and the end of it (draining the last few micro-batches before the flush). However, GPipe still has to wait for a full pipeline flush before it can start on the next mini-batch, and so, the bubble is not fully eliminated. Further, holding on to activations for many in-flight micro-batches consumes significant memory at each device.

Let us quantify the size of the bubble with this type of pipeline parallelism. Assume the pipeline has P stages, and suppose a mini-batch is split into M micro-batches. Let F denote the time taken by a forward pass of one micro-batch through one stage, and let B denote the time taken by the corresponding backward pass. Consider the very first device in the pipeline. Before it can start its backward pass on the first micro-batch, it must wait for that micro-batch to travel all the way to the last stage, and for the resulting gradient to travel all the way back. This wait manifests as an idle period of size

$$\text{Bubble size} = (P-1)(F+B)$$

On the other hand, the amount of useful work that any single device performs when processing all M micro-batches of a mini-batch is

$$\text{Ideal time} = M(F+B)$$

The fraction of the ideal processing time that is wasted in the bubble is therefore approximately

$$\frac{(P-1)(F+B)}{M(F+B)} = \frac{P-1}{M}$$

This simple relationship tells us that in order to keep the bubble small, we need the number of micro-batches M to be much larger than the number of pipeline stages P, that is, $M \gg P$. This is intuitive to understand: the pipeline needs to be kept busy with enough concurrent work to hide the latency introduced by its depth. Unfortunately, increasing M is not free: since GPipe holds on to the activations of every in-flight micro batch until the corresponding backward pass has been performed, a larger M directly translates into a larger memory footprint for stashed activations, and this quickly becomes the limiting factor. This tension between reducing bubble time by increasing M, and controlling the memory footprint by keeping M small, motivates a search for scheduling techniques that reduce the bubble without requiring a large number of micro-batches in flight.

6.3.3 PipeDream: An Asynchronous 1F1B Schedule

The PipeDream system [Narayanan, Harlap, et al., 2019] proposes a solution to this tradeoff between the bubble size and the memory consumed by in-flight micro-batches that is faced by GPipe. Instead of waiting for all micro-batches to complete their forward pass before starting backward passes, PipeDream lets each worker begin the backward pass of an already processed micro-batch as soon as it is possible, interleaving it with the forward passes of subsequent micro-batches. Further, instead of waiting for a full pipeline flush before updating weights, it updates weights asynchronously and locally at each worker as soon as the relevant gradient is ready (Figure 6.24).

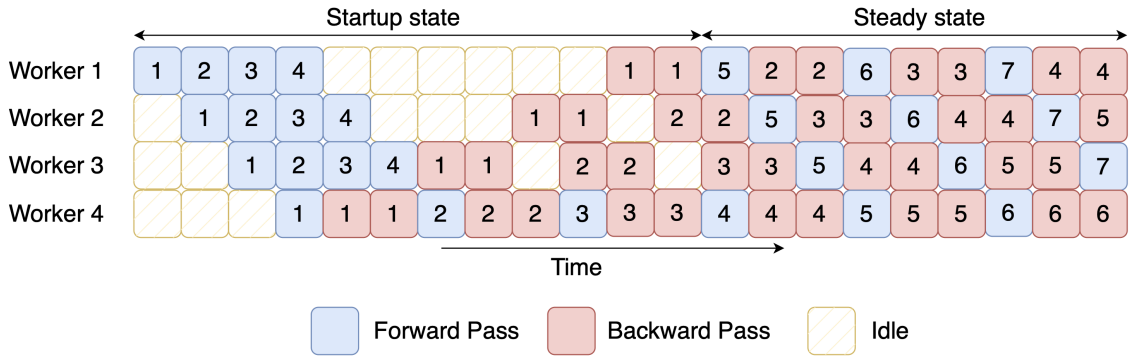


Figure 6.24: PipeDream

After a brief startup period in which the pipeline is only partially filled, each worker settles into a repeating steady-state pattern of performing one forward pass and one backward pass alternately, a schedule that is commonly abbreviated as 1F1B, for one-forward-one-backward. Under 1F1B, once the

pipeline has been warmed up, every worker is continuously busy — there is no waiting for a global flush, and consequently no large bubble of the kind seen in GPipe. The number of micro-batches that need to be simultaneously in flight to sustain this steady state is on the order of the number of workers (equivalently, the depth of the pipeline), which is considerably smaller than what GPipe might require to make its bubble negligible. Therefore, the memory required to hold activations of in-flight micro-batches is also much smaller in this framework than with GPipe.

It is worth noting that the 1F1B schedule achieves close to full utilization only when the pipeline stages are reasonably balanced, meaning that each stage takes roughly the same amount of time to process a mini-batch. Note that the forward and backward computations themselves can take different amounts of time from one another, but the total amount of work assigned to each stage, summed over its forward and backward computation, should be approximately equal across stages. Since the overall throughput of the pipeline is governed by its slowest stage, an imbalanced partitioning of the model across workers reintroduces bubbles and stalls at the faster stages, which must wait for the straggling stages to catch up. Designing a pipeline-parallel system therefore requires partitioning the model across devices in a way that keeps the per-stage workloads balanced.

Weight Stashing

Interleaving forward and backward passes asynchronously, as PipeDream does, introduces a subtle error that is not present in GPipe's fully synchronous schedule. In Figure 6.24, consider a worker (say, worker 1) that performs the forward pass of micro-batch 5 using the weights that were updated after the backward pass of micro-batch 1. Now, by the time this worker performs the backward pass and computes gradients corresponding to micro-batch 5, the local weights may already have been updated further, say, by the gradients of micro-batches 2, 3, and 4. If the backward pass of micro-batch 5 were to be computed using these newer weights, rather than the weights that were actually used during its forward pass, the gradient computed would not correspond to a consistent version of the model, which would corrupt the learning process. In other words, with asynchronous weight updates, the weights that were used in the forward pass, which contributed to the eventual loss, are no longer the same when the loss gradient arrives to update them, which violates the correctness of the backpropagation mechanism itself.

PipeDream avoids this problem through a technique called weight stashing. Each worker keeps multiple versions of its own weights in a small buffer, one for every micro-batch that is currently in flight at that worker (Figure 6.25). When the forward pass of a micro-batch is performed, the version of the weights used is saved; when the corresponding backward pass eventually arrives, the worker retrieves that exact saved version of the weights and uses it to compute the backward pass, guaranteeing that forward and backward computations for any given micro-batch are always consistent with one another at that worker. Because the number of micro-batches simultaneously in flight at a worker is bounded by the depth of the pipeline, the number of weight versions that need to be stashed is similarly bounded, and does not grow with the total number of micro-batches processed.

In Figure 6.25, weight version $W_i^{(j)}$ is created at the moment worker i finishes the backward pass for micro-batch j; this newly updated version becomes the weights that worker i will stash and use for the forward pass of the next micro-batch it receives. A stashed version is only ever discarded once it is certain that no future forward or backward pass will need it, and this happens in one of two ways. The first case is a version becoming obsolete before it is ever used — if worker i completes another backward pass and produces a new version before any forward pass has used the version currently held, the held version is superseded and can be dropped immediately. Worker 3's timeline in Figure 6.25 shows this happening twice in succession at worker 3. Completing the backward pass of micro-batch 1 creates $W_3^{(1)}$, but before any forward pass retrieves it, the backward pass of micro-batch 2 completes and creates $W_3^{(2)}$, so $W_3^{(1)}$ is discarded, unused. The same then happens to $W_3^{(2)}$ once the backward pass of micro-batch 3 produces $W_3^{(3)}$, before $W_3^{(2)}$ is never checked out for any forward pass. This is why worker 3's version list in the figure begins at $W_3^{(3)}$ rather than $W_3^{(1)}$: the two earlier versions were superseded before use, and only $W_3^{(3)}$ survives, being the first version actually used by a forward pass, in this case, by micro-batch 5.

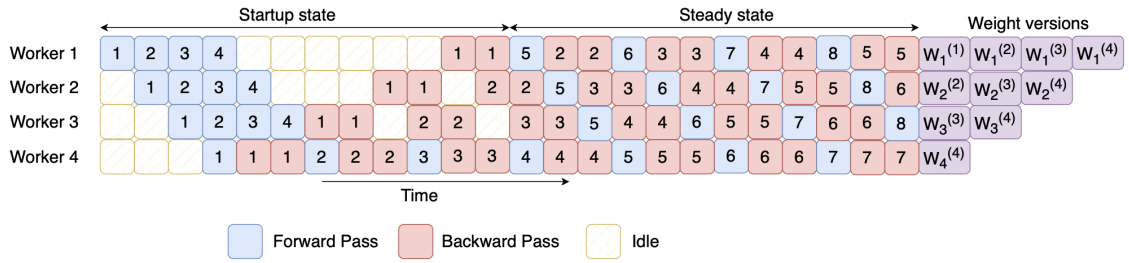


Figure 6.25: Weight Stashing. For each worker, the weight versions listed are those actually held in its stash buffer — not yet consumed or discarded — at the instant just before that worker's own backward pass for micro-batch 5 begins.

The second, more common case in a steady 1F1B schedule is a version being discarded after use. Once a weight version has been used in the forward pass of some micro-batches, and the stashed weights are used in the corresponding backward passes of the same batches, the stashed weights have served their purpose and can be safely freed. Worker 1's timeline illustrates this. After the backward pass of micro-batch 1 completes, worker 1 stashes $W_1^{(1)}$, which is immediately consumed by the forward pass of micro-batch 5. At the moment the diagram ends, micro-batch 5's backward pass has not yet reached worker 1, so $W_1^{(1)}$ is held until that backward pass completes. Note that Figure 6.25 shows the number of weight versions actually held in each worker's stash buffer at the instant just before the backward pass for micro-batch 5 begins at the worker.

Vertical Sync

Weight stashing solves the consistency problem for a single worker, ensuring that the forward and backward pass of one micro-batch at that worker use the same version of weights. It does not, however, guarantee consistency across workers. Because each stage in the pipeline updates its own weights independently and at its own pace, a given micro-batch may end up seeing a different effective version of the model at each stage that it passes through. For example in Figure 6.24, at the first worker, micro-batch 5 might use weights that reflect the update from micro-batch 1, but at the second worker, it might use weights that reflect updates from both micro-batches 1 and 2, since the second worker has had time to complete one more backward pass by the time micro-batch 5 arrives there. The model that micro-batch 5 effectively experiences as it flows through the pipeline is therefore a patchwork of different weight versions at different depths, rather than a single consistent version of the network.

To address this, PipeDream introduces an additional mechanism called vertical sync. The idea is to explicitly tag each micro-batch with the version identifier of the weights it used at the very first stage of the pipeline, and to propagate this tag along with the micro-batch's activations as it moves through the pipeline. Every subsequent stage is then required to use its own stashed weights corresponding to that tagged version, even if newer local versions are available. This guarantees that every stage that processes a given micro-batch does so using weight versions that all originated from the same point in training, restoring a coherent, single view of the model for that micro-batch across the entire pipeline. This mechanism incurs the cost of having to stash and look up the appropriate historical weight version at every stage of the pipeline consistently.

In Figure 6.25, micro-batch 5's forward pass at worker 1 uses $W_1^{(1)}$, so under vertical sync this tag of (1) must follow micro-batch 5 through the rest of the pipeline: workers 2, 3, and 4 are each required to use their own version (1) — $W_2^{(1)}$, $W_3^{(1)}$, and $W_4^{(1)}$ — rather than the newer versions $W_2^{(2)}$, $W_3^{(3)}$, and $W_4^{(4)}$. Notably, versions such as $W_2^{(1)}$ and $W_3^{(1)}$ are already absent from the stash buffers shown in the figure, since plain weight stashing would have discarded them as obsolete before micro-batch 5 ever arrived; vertical sync therefore requires each worker to retain a longer history of weight versions than weight stashing alone would otherwise need.

PipeDream-2BW

Weight stashing and vertical sync, taken together, restore the correctness of learning, but at the cost of increased memory consumption: a worker may need to keep around one saved copy of its weights for every micro-batch that is currently in flight. Since the number of in-flight micro-batches grows with the depth of the pipeline, so does the number of weight versions that must be stored. PipeDream-2BW, short for double-buffered weight updates, is designed to mitigate this cost, while still keeping the gradient computation internally consistent for every micro-batch.

The key idea of PipeDream-2BW is to stop updating the weights after every single micro-batch, and instead to accumulate gradients over a group of P consecutive micro-batches, where P is chosen to match the depth of the pipeline, i.e., the number of workers (Figure 6.26). All P micro-batches

in such a group perform their forward and backward passes using the same frozen version of the weights, just as they would if the pipeline were flushed and restarted between each group of P micro-batches, as in GPipe. However, unlike GPipe, the schedule here never actually drains the pipeline — forward and backward passes continue to be interleaved as usual, with the difference that the weight version used by any given group of micro-batches stays fixed until that group is fully done. Only once all P micro-batches in the group have contributed their gradients does the worker apply a single accumulated update, producing a new weight version.

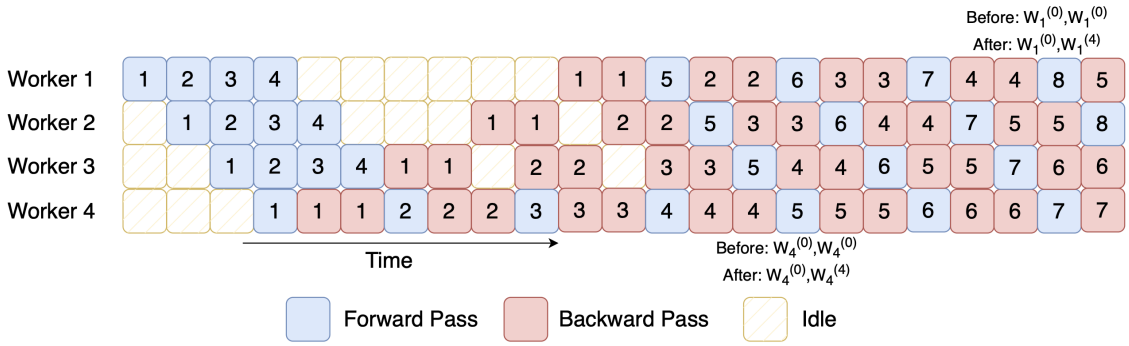


Figure 6.26: PipeDream-2BW. Annotations show the two weight buffers held at worker 1 and worker 4 immediately before and after that worker finishes the backward pass of micro-batch 4, the last micro-batch of the first group.

Because a new version is produced only once every P micro-batches, a worker only ever needs to retain two versions of its weights at a time: a stable version, used for the forward and backward passes of the groups currently executing, and a running version, into which the gradients of that same group's backward passes are being accumulated. As Figure 6.26 shows, the stable version is not replaced the moment the running version finalizes; instead, the newly finalized version waits and becomes the stable version only for the next group of micro-batches, once the group currently in flight has fully passed through the pipeline. For example, in Figure 6.26 where $P = 4$, micro-batches are partitioned into Group 1 (micro-batches 1–4) and Group 2 (micro-batches 5–8). Group 1 executes using the initial weight version $W^{(0)}$. When worker 1 finishes the backward pass of micro-batch 4 (the final micro-batch of Group 1), it applies the accumulated updates to produce version $W_1^{(4)}$. Worker 1 still retains $W_1^{(0)}$ as its stable version (currently serving Group 2), and $W_1^{(4)}$ as its newly finalized version. Version $W_1^{(4)}$ is subsequently adopted as the stable version for Group 3 (micro-batches 9–12) once the ongoing Group 2 finishes executing. This is the origin of the name "double buffered" — no matter how deep the pipeline is or how many micro-batches are in flight, only two weight buffers are ever required per worker, rather than one buffer per in-flight micro-batch.

This grouping also simplifies the consistency problem that vertical sync was designed to solve.

Since every micro-batch inside a group uses the same frozen weight version at the worker where it starts, and since that version does not change until the entire group has finished, a micro-batch is guaranteed to see a single, coherent version of the weights at every stage of the pipeline for the whole time it is in flight, without needing to explicitly tag it with a version identifier and look up matching historical weights at every stage, as vertical sync requires. The price paid for this simplicity and lower memory footprint is that the weights used by any group are slightly stale, since they reflect the accumulated updates only up through the end of the group before last, rather than the very latest gradients, but this staleness is bounded and comparable in spirit to the staleness already introduced by vertical sync.

It is also worth summarizing how these four schemes compare in terms of their memory footprint for storing weight versions. Naive asynchronous pipelining (without weight stashing) needs only a single weight version. Weight stashing and vertical sync both require storing one weight version per micro-batch currently in flight, so their memory footprint grows linearly with the depth of the pipeline P. PipeDream-2BW, by contrast, needs only two weight versions per worker regardless of pipeline depth, making its weight memory constant in P. Activation memory, however, scales with P in all four schemes alike, since the number of micro-batches that must have their forward-pass activations held in memory awaiting a backward pass is always bounded by the number of in-flight micro-batches, which is proportional to P.

Correctness of Gradient Descent

It is useful to compare the different scheduling strategies discussed so far in terms of what update rule they actually implement, since this determines how correctly they reproduce standard synchronous gradient descent. The ordinary, sequential update rule for parameters $w = (w_1, \ldots, w_n)$, where w_i denotes the parameters of the i-th pipeline stage and n is the total number of stages, at step t can be written as

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t)}, w_2^{(t)}, \ldots, w_n^{(t)}\right),$$

where every coordinate of the gradient, i.e., $\nabla f = (\frac{\partial f}{\partial w_1}, \ldots, \frac{\partial f}{\partial w_n})$, with one component per pipeline stage, is computed using the very same, current version of every parameter. If a system waits for all micro-batches in a mini-batch to finish before applying any update, as GPipe does, then this equation is satisfied exactly, since all forward and backward computations for the mini-batch use the same weights throughout, and the resulting gradients are fully consistent with one another.

PipeDream with only weight stashing, and without vertical sync, instead computes an update of the form

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t-n+1)}, w_2^{(t-n+2)}, \ldots, w_n^{(t)}\right),$$

where different coordinates of the gradient are evaluated using weight versions from different points in the recent past at each stage. This means that even though weight stashing guarantees that the forward and backward pass at a single stage are mutually consistent, the gradient computed for

the model as a whole mixes several inconsistent snapshots of the parameters, one per stage. Such gradients are not the gradient of any single, well-defined loss evaluated at one point in the parameter space, which introduces a form of error into the optimization process.

PipeDream with vertical sync instead computes

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t-n+1)}, w_2^{(t-n+1)}, \ldots, w_n^{(t-n+1)}\right),$$

where every coordinate now uses the same past weights version. The gradient computed is therefore internally consistent, corresponding to the gradient of the loss at a single, well-defined point in the parameter space, unlike plain weight stashing. The trade-off is that this point is somewhat stale relative to the very latest weights, since it corresponds to an earlier step of training rather than the current one. In practice, this staleness is bounded by the depth of the pipeline and tends to have a much milder effect on convergence than the inconsistency present in plain weight stashing, since stale-but-consistent gradients still correspond to valid gradient directions of the loss function at some nearby point.

PipeDream-2BW computes an update of the form

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t-s)}, w_2^{(t-s)}, \ldots, w_n^{(t-s)}\right),$$

where s is a small, bounded staleness term (at most one weight-version lag) common to every coordinate. As in vertical sync, every coordinate of the gradient is evaluated using a single, common weight version $w^{(t-s)}$, so the gradient computed is internally consistent and corresponds to a well-defined point in weight space; the difference from vertical sync is purely in how that shared version is produced and maintained: rather than tagging each micro-batch and looking up a matching historical weight version at every stage, PipeDream-2BW freezes the weights for an entire group of P micro-batches at a time and only advances $w^{(t)}$ to $w^{(t+1)}$ once all P micro-batches in the group have contributed their gradients. This achieves the same kind of internal consistency as vertical sync, and with similarly bounded staleness, while requiring only two stored weight versions per worker instead of one version per in-flight micro-batch.

PipeDream-Flush

A middle ground between GPipe and asynchronous PipeDream, adopted by systems such as Megatron-LM [Narayanan, Shoeybi, et al., 2021] is known as PipeDream-Flush. Workers execute the same 1F1B schedule as PipeDream, alternating between one forward pass and one backward pass once the pipeline has been warmed up, so that the device utilization benefits of 1F1B are retained. Unlike asynchronous PipeDream, however, PipeDream-Flush does not update weights continuously; instead, it waits until the end of a mini-batch, flushes the pipeline by finishing all outstanding backward passes, and only then performs a single synchronous weight update, exactly as GPipe does (Figure 6.27). This restores exact correspondence with the standard, synchronous gradient descent update rule, since gradients from every micro-batch of a mini-batch are computed with the same weights and accumulated before any update is applied.

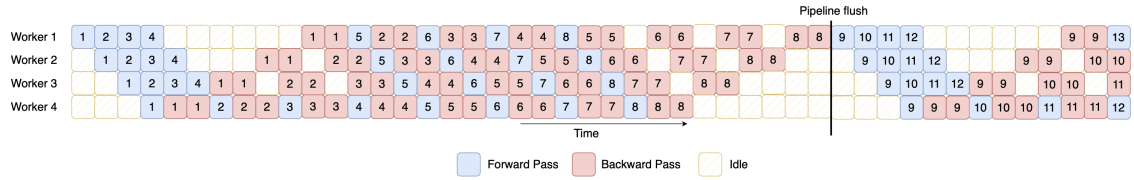


Figure 6.27: PipeDream-Flush

Because PipeDream-Flush uses the 1F1B ordering of micro-batches, rather than GPipe's all-forward-then-all-backward ordering, the number of micro-batches that are in flight, and correspondingly the number of activations that must be stashed, at any one time, is bounded by the depth of the pipeline, rather than by the total number of micro-batches in the mini-batch. This makes PipeDream-Flush considerably more memory-efficient than GPipe while achieving essentially the same accuracy, since both compute synchronous gradients. The size of the bubble in PipeDream-Flush is, however similar to GPipe, since a pipeline flush is still required once per mini-batch.

6.3.4 Optimizations to Pipeline Parallelism

Having examined the key ideas in pipeline parallelism, we now discuss some advanced optimizations being proposed in recent work to further improve efficiency.

Pipeline Parallelism with Interleaved Stages

The schedules discussed so far, including GPipe and PipeDream-Flush, all share one structural feature: each device is assigned a single, contiguous block of layers, and that block does not change for the duration of training. A worker sitting near the beginning of the pipeline is therefore stuck waiting for a micro-batch to traverse the remaining $P-1$ devices before it can begin the corresponding backward pass, and this wait is precisely what produces the pipeline bubble. Interleaved pipeline parallelism, introduced alongside PipeDream-Flush in the Megatron-LM system [Narayanan, Shoeybi, et al., 2021], reduces this bubble not by changing the forward-backward ordering, but by changing how layers are assigned to devices in the first place.

Instead of giving each device one contiguous chunk of the model, we partition the model into a larger number of smaller chunks, and give each device several non-contiguous chunks. Suppose the model has been divided into P pipeline stages, one per device, and that each stage is itself further subdivided into v smaller virtual stages. Rather than assigning device 1 layers 1 through L/P and device 2 layers $L/P+1$ through $2L/P$, and so on, we instead assign device 1 the first virtual stage of every group along with several virtual stages spread further down in the model. Concretely, with $P = 2$ devices in the pipeline, and $v = 2$ virtual stages per device, applied to an eight-layer model, the naive partitioning would give device 1 layers 1–4 and device 2 layers 5–8. Under interleaving, device 1 instead owns layers $1, 3, 5, 7$ and device 2 owns layers $2, 4, 6, 8$, with each device holding two separate, disjoint chunks rather than one contiguous block of layers (Figure 6.28).

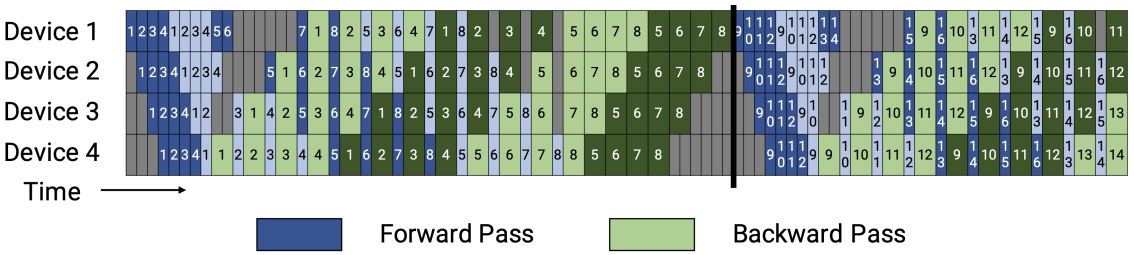


Figure 6.28: Pipeline parallelism with interleaved stages. Within each of the two broad color families — blue for forward passes and green for backward passes — the lighter and darker shades distinguish different non-contiguous chunks of the model layers. Image credit: [Narayanan, Shoeybi, et al., 2021]

A micro-batch flowing through the pipeline now visits a device, moves on to the next device, and eventually comes back to the first device again to process its second chunk of layers, before moving forward once more. Each device therefore performs several shorter forward and backward passes per micro-batch instead of one long forward pass and one long backward pass, and micro-batch activations passes between devices more frequently than in the non-interleaved schedule. This requires model-parallel communication for the activations and gradients between chunks a total of v times as often as before, since a micro-batch now crosses stage boundaries v times more often on its way through the model. What is gained in exchange is a reduction in the size of the bubble. Because each virtual stage represents only $1/v$ of the work that a full device stage used to represent, the forward and backward times per virtual stage shrink to F/v and B/v, and a device no longer needs to wait for an entire block of layers to clear the rest of the pipeline before resuming useful work; it only needs to wait for a much thinner slice of it. As before, all P devices still follow the 1F1B ordering within this finer-grained schedule, alternating between one forward pass and one backward pass at the level of virtual stages, and a full pipeline flush is still performed once per mini-batch, exactly as in PipeDream-Flush. Interleaving therefore trades bubble time for communication volume, and the right choice of v depends on how the communication bandwidth between devices compares to the compute time saved by shrinking the bubble.

Zero Bubble Pipeline Parallelism

All of the schedules discussed so far treat the backward pass as a single, indivisible unit of work. The Zero Bubble pipeline parallelism scheme [Qi et al., 2023] observes that this is not actually necessary, and that the backward pass of a linear layer can be split into two independent parts. For a layer computing $z = Wx$, the backward computation needs to produce both the gradient with respect to the input, $\nabla_x L = W^\top \nabla_z L$ (which is propagated to the earlier layer), and the gradient with respect to the layer's own weights, $\nabla_W L = \nabla_z L \, x^\top$ (which is used for this layer's weight update). Let's refer the computation of the input gradient as stage B, and the computation of the weight gradient, as stage

W (Figure 6.29). The key observation of "zero bubble" pipeline parallelism is that stage B of a given layer is on the critical path of the pipeline — the previous layer's own stage B cannot begin until this layer has produced the input gradient it depends on, so stage B must be scheduled promptly and propagated backward through the pipeline without delay. Stage W, on the other hand, has no such downstream dependency within the backward pass; it only needs to be completed before the corresponding weight update is applied, and it can therefore be deferred and slotted into whatever idle gaps would otherwise appear in the schedule as a bubble.

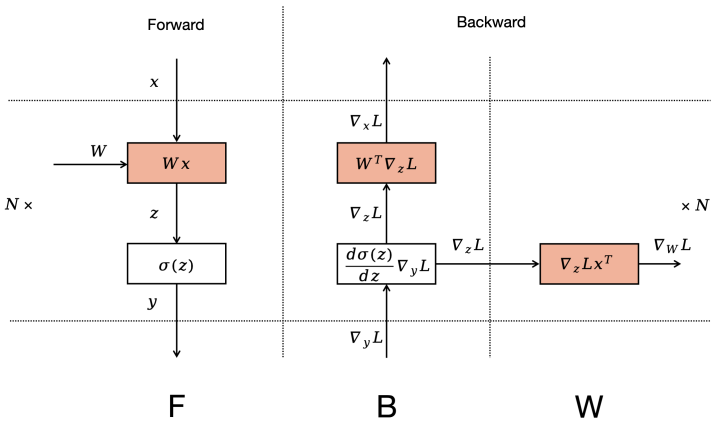


Figure 6.29: MLP Computation; Image credit: [Qi et al., 2023]

By explicitly separating B and W and finding a schedule that interleaves F, B, and W operations across devices subject to their dependency constraints, Zero Bubble pipeline parallelism is able to fill in the gaps that a coarser 1F1B schedule would leave idle. This often reduces the size of the bubble substantially compared to ordinary 1F1B with a synchronous flush, and in favourable configurations the bubble can be reduced to essentially zero, all while preserving the exact, synchronous gradient descent update, since the W computations are simply rearranged in time rather than made asynchronous with respect to the weight update itself (Figure 6.30).

Bidirectional Pipelining in DualPipe

A further refinement, used in the training of large language models such as DeepSeek-V3 [DeepSeek-AI, 2025], is the DualPipe schedule. The central idea is to feed micro-batches into the pipeline from both ends simultaneously, rather than only from one end. One stream of micro-batches flows forward through the devices in the usual order, while a second stream flows in the reverse direction. Because both directions are active at once, each device is, at any given time, likely to have useful work available from one direction even while it might otherwise be waiting on a dependency from the other, which further reduces idle time relative to a purely unidirectional pipeline.

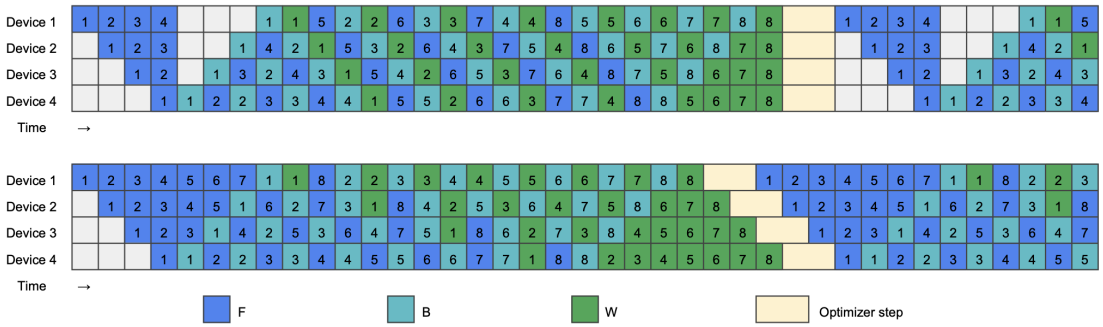


Figure 6.30: Zero bubble pipeline parallelism; Image credit: [Qi et al., 2023]

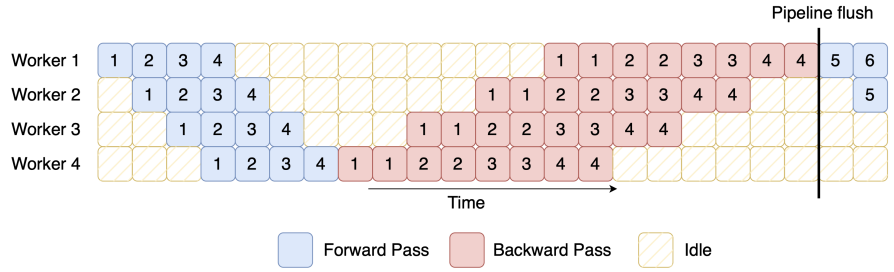
DualPipe combines this bidirectional feeding with the same idea of splitting the backward computation into a backward-for-input part and a backward-for-weights part that Zero Bubble pipeline parallelism introduced, allowing the weight-gradient computation to be scheduled flexibly into whatever gaps remain. In addition, DualPipe explicitly targets overlap between computation and the communication of activations and gradients between devices, so that the transmission of one micro-batch's activations can proceed concurrently with the computation of another micro-batch's forward or backward pass on the same device, rather than these being treated as strictly sequential operations. Taken together, feeding the pipeline symmetrically from both ends, splitting the backward pass, and overlapping compute with communication allow DualPipe to achieve very high device utilization.

Figure 6.31: DualPipe scheduling; Image credit: [DeepSeek-AI, 2025]

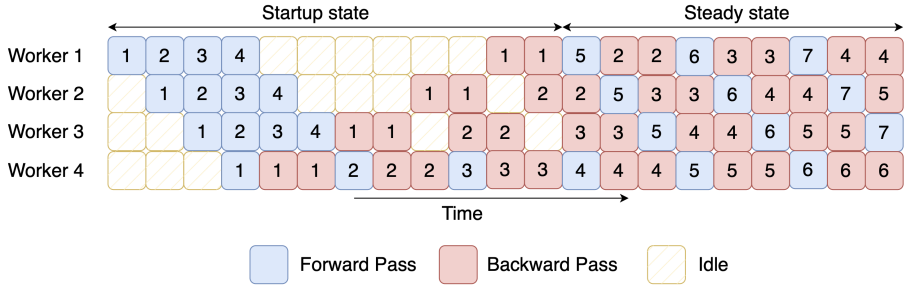
Practice Problem 6.3

Three example PP schedules are shown below (Figure 6.32). You must answer questions over generalized versions of these schedules. The weight update is asynchronous in schedule B, whereas in A and C it is shown by a black solid line.

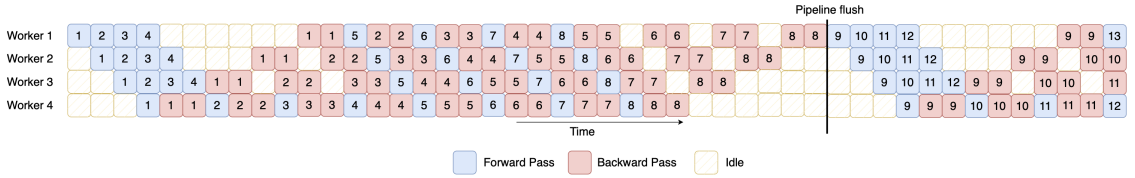
(i) Consider a generalization of schedule A shown above. Suppose we have a batch size of S ($S = 8$ in the figure above), and the pipeline is distributed across p workers in stages ($K = 4$ above). Let F and B be the time taken for the forward and backward passes respectively at each worker node. What is the duration of



(a) Schedule-A



(b) Schedule-B



(c) Schedule-C

Figure 6.32: Schedules

the bubble at the first worker node for schedule A? Write your answer in terms of S, p, F, and B.
Solution: In schedule A, worker W_1 is idle while workers $W_2, \ldots, W_K$ complete their forward passes before W_1 begins its backward pass. The first worker must wait for all $K-1$ other workers to finish their forward and backward passes. So the bubble duration at W_1 is: $(p-1)(F+B)$

(ii) Which of the above schedules consumes the most amount of memory for storing intermediate activations of in-flight batches? Identify all schedules which consume the most memory, and justify your answer.
Solution: Schedule A consumes the most memory for intermediate activations. All S micro-batches complete their forward passes before any backward pass begins. At the peak, W_1 has activations from all S micro-batches simultaneously in flight, so the total activation memory scales as $O(S \cdot K)$ (all micro-batches across all stages).

(iii) Which of the above schedules consumes the least amount of memory for storing intermediate activations of in-flight batches? Identify all schedules which consume the least memory, and justify your answer.
Solution: Schedules B and C. Both schedules employ a 1F1B steady-state. By performing a backward pass sooner (than schedule A) after a corresponding forward pass, they limit the maximum number of in-flight micro-batches at any given time to p.

(iv) Which of the above schedules performs gradient descent accurately, without using stale gradients? Identify all schedules with accurate gradient descent, and justify your answer.
Solution: Schedules A and C perform gradient descent accurately without stale gradients. In both schedules, a weight update occurs only after all micro-batches in a batch have completed both their forward and backward passes. Thus, all gradients used for an update are computed under the same set of weights. Schedule B (PipeDream) uses stale gradients: because weight updates happen asynchronously mid-batch (each worker updates its weights as soon as its own backward pass completes), subsequent micro-batches see different weights at different stages.

Programming Assignment 6.2: Pipeline Parallelism

In this assignment, you will implement pipeline parallelism for a small two-layer MLP, simulated across multiple Python processes to mimic how large models are split across GPUs in real distributed training. The network is split into two stages — Layer 1 (a linear layer + ReLU) owned by a "GPU-0" process, and Layer 2 (a linear layer) owned by a "GPU-1" process — with a separate "Main" process acting as coordinator that computes the MSE loss and kicks off backpropagation. Communication between these three processes happens entirely through `multiprocessing.Queue` objects, standing in for GPU-to-GPU communication on real hardware: activations flow forward from GPU-0 to GPU-1 to Main, and gradients flow backward from Main to GPU-1 to GPU-0, with each process computing and returning its own parameter gradients. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/pipeline_parallelism

6.4 Tensor Parallelism

In the previous section we saw that pipeline parallelism lets us train models whose layers, taken together, do not fit on a single GPU, by placing different layers on different devices and streaming micro-batches through them. Pipeline parallelism, however, does not help us if even a single layer is too large to fit on one GPU. As models grow to have hundreds of billions of parameters, individual weight matrices inside a transformer layer, such as the projection matrices in the attention block or the two linear layers of the feed-forward block, can themselves be larger than the memory of a single accelerator. We need a way to split the computation inside a layer across multiple devices, which is achieved by tensor parallelism (TP), sometimes also called intra-layer model parallelism. While pipeline parallelism partitions a model "vertically", by assigning whole layers to different devices, tensor parallelism partitions a model "horizontally", by splitting the tensors within a single layer, such as its weight matrices and the activations that flow through it, across several devices. With tensor parallelism, every device holds only a slice of every layer, and the devices cooperate through communication to jointly compute the output of that layer. In practice, tensor parallelism is often combined with other types of parallelism like pipeline or data parallelism.

6.4.1 Tensor Parallelism for Matrix Multiplication

We will now use the example of matrix multiplication, which is the dominant operation in most large neural networks, to explain how tensor parallelism works. Consider a linear layer that computes an output activation Y from an input activation X and a weight matrix W, so that $Y = XW$. When one or both tensors, say W, is too large to fit on one device, we would like to use tensor parallelism to partition W to different devices, have each device perform a smaller local matrix multiplication, and then combine the local results to recover the full product Y.

There are two natural ways of slicing a matrix product like this, depending on whether we cut W along its columns or along its rows. We will use the example shown in Figure 6.33 to explore both of these ideas in more detail.

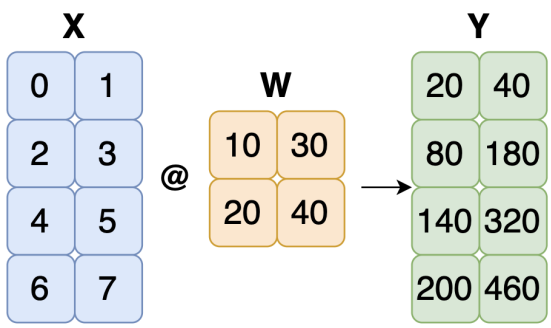


Figure 6.33: Matrix multiplication example

Column-Wise Partitioning

In a column-wise partitioning of the weights tensor, we split the weight matrix W along its columns and place one slice of columns on each device. If we partition W column-wise into blocks $W = [\, W_1 \; W_2 \; \cdots \; W_n \,]$, we can compute Y as:

$$Y = XW = X[\, W_1 \; W_2 \; \cdots \; W_n \,] = [\, XW_1 \; XW_2 \; \cdots \; XW_n \,].$$

Each column block XW_i is a product of the full X with only a slice of W. Consequently, the input activation must first be broadcast to all devices in the tensor-parallel group, using the Broadcast collective communication primitive illustrated in Figure 6.34.

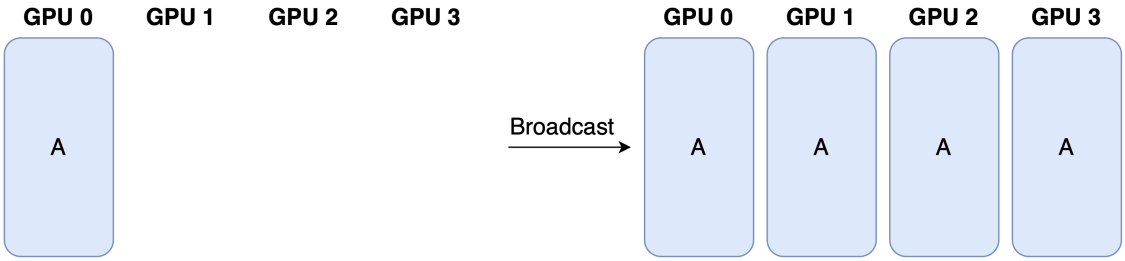


Figure 6.34: Broadcast

Next, each device computes its local product XW_i independently, and now, the final answer is obtained by placing the resulting product blocks side by side, using an AllGather operation (Figure 6.12), after which every device holds every column of Y. If only a single device (e.g., the owner of a subsequent layer) needs the complete result rather than every device in the group, the simpler Gather primitive can be used in place of AllGather, which collects each device's column slice Y_i and concatenates them into the full Y only on one designated root device (Figure 6.35).

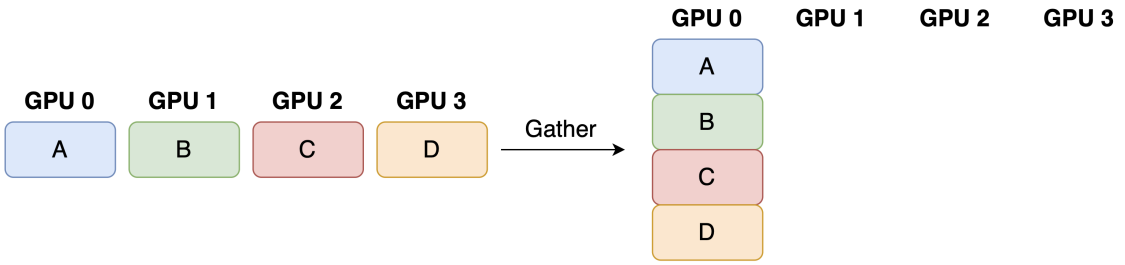


Figure 6.35: Gather

Matrix multiplication using such column-wise partitioning for the simple example shown in Figure 6.33 is illustrated in Figure 6.36. We split W into its two individual columns, W_0 and W_1, and place them in separate devices. Both devices receive a full copy of X through a broadcast operation. The two devices then compute Y_0 and Y_1 independently, after which we gather the full output Y by concatenating the columns of Y_0 and Y_1.

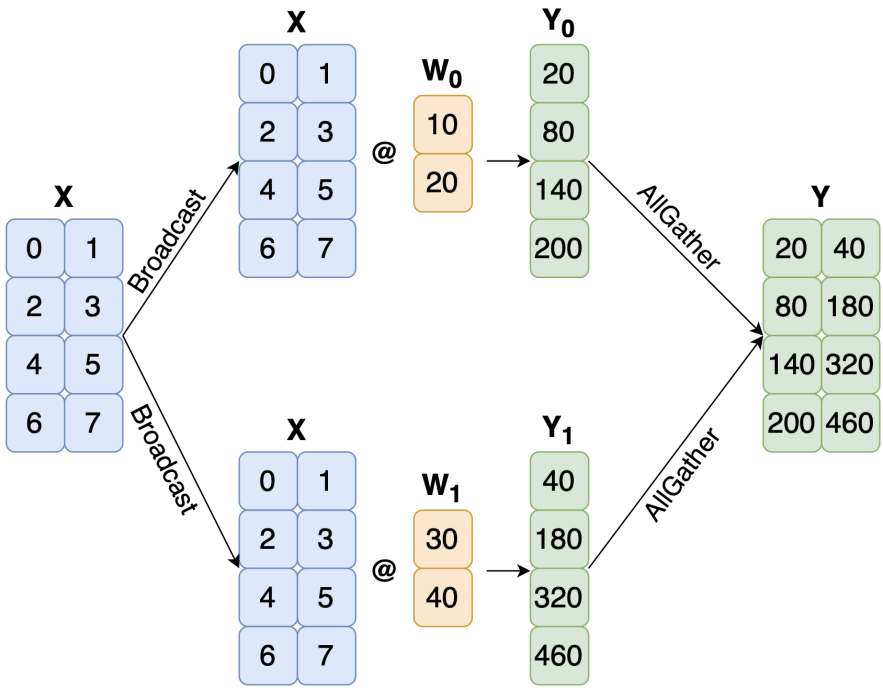


Figure 6.36: Column-wise tensor parallelism

Row-Wise Partitioning

Let us now see another way by which we can apply tensor parallelism to distribute a matrix multiplication operation across multiple GPUs. In the example of computing the product $Y = XW$, we can also consider partitioning W row-wise across the multiple GPUs. We would then have to correspondingly partition X column-wise, so if $X = [\, X_1 \; X_2 \; \cdots \; X_n \,]$ and $W = [\, W_1 \; W_2 \; \cdots \; W_n \,]^\top$, then we compute the product Y as

$$Y = XW = [\, X_1 \; X_2 \; \cdots \; X_n \,] \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_n \end{bmatrix} = \sum_{i=1}^{n} X_i W_i$$

With this row-wise tensor parallelism, we use the Scatter collective communication primitive to distribute the appropriate column slice of X to each device (Figure 6.37). Then each device performs a smaller local matrix multiplication, using its slice of both the input and the weights. Finally, the partial results are added together, rather than concatenated, to produce the final output Y. We can use the AllReduce communication primitive (Figure 6.6) to compute Y, or if the result is needed at only one of the devices, then the simpler Reduce primitive can be used as well, which sums each device's partial result Y_i into the complete $Y = \sum_i Y_i$ on only one designated root device (Figure 6.38).

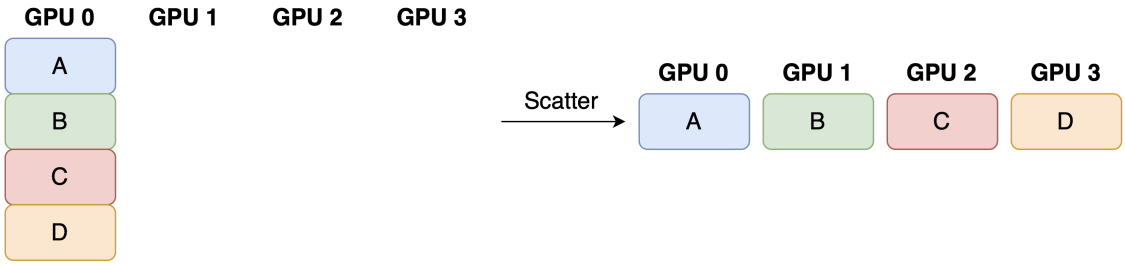


Figure 6.37: Scatter

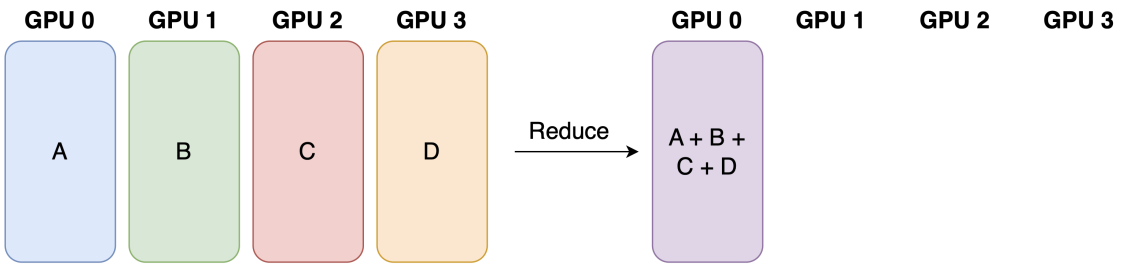


Figure 6.38: Reduce

For the example shown in Figure 6.33, matrix multiplication using row-wise partitioning is illustrated in Figure 6.39. We split W into its two rows: W_0 and W_1, and correspondingly split X into its two columns: X_0 and X_1. One device receives X_0, W_0 and computes $Y_0 = X_0 W_0$, while the

other device holds X_1, W_1 and computes $Y_1 = X_1 W_1$. The two devices then add their local results with an AllReduce operation, after which both devices hold the same, complete product matrix $Y = Y_0 + Y_1$.

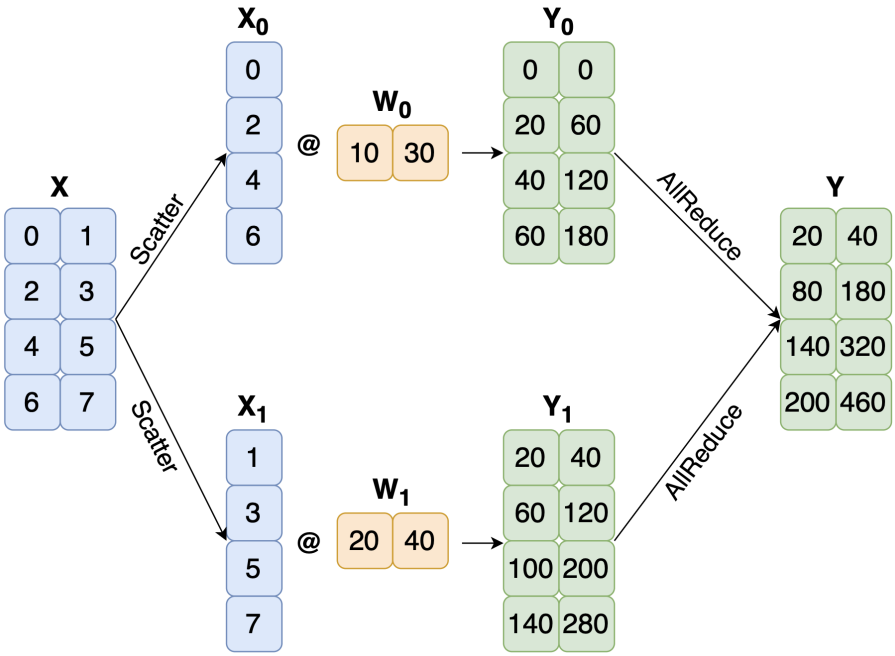


Figure 6.39: Row-wise tensor parallelism

6.4.2 Tensor Parallelism for Larger Models

So far, we have seen how to distribute a simple operations like matrix multiplication across GPUs, in order to realize the idea of tensor parallelism. Modern deep learning networks, however, are composed of more complex layers, e.g., self-attention. We will now discuss a few examples of how operations in such networks can be distributed across devices using tensor parallelism.

Tensor Parallelism with MLP

Let us consider the example of a simple MLP, with two linear layers and a non-linear activation function between them, represented as $Y^2 = f(XW^1)W^2$, where f denotes an element-wise nonlinearity function like ReLU. The linear layers themselves are simple matrix multiplications of the activations and weights, and can be distributed across devices using the techniques studied in the previous section. However, if we naively applied column-wise tensor parallelism to W^1 and then again to W^2, we would need an AllGather after the first layer to reassemble the full activation before the second

layer could even begin, and then another collective communication after the second layer. This is wasteful, and it turns out to be unnecessary.

One clever idea would be to split the two layers in different ways across devices, with the first layer's weights split column-wise, and the second layer's weights split row-wise, as shown in Figure 6.40. The full input X is Broadcast to both devices, exactly as in column-wise tensor parallelism. After the first layer is processed, the GPUs are left with only partial outputs of the first layer, distributed column-wise. Now, if we apply row-wise partitioning of weights for the second layer, then these column-wise activations from the first layer suffice as inputs for the second layer's computations. With this way of splitting the weights of both layers, no network communication whatsoever is needed between the two devices while computing Y_0^1, Y_1^1, Y_0^2, and Y_1^2, and the devices only perform a single AllReduce to obtain Y_2 after both layers are run. Further, any non-linear activation function f that can be applied element-wise (or even column-wise) on the first layer's outputs can be computed locally, using the partial outputs at each device. This example illustrates how tensor parallelism can be extended beyond simple matrix multiplication to networks composed of multiple layers as well.

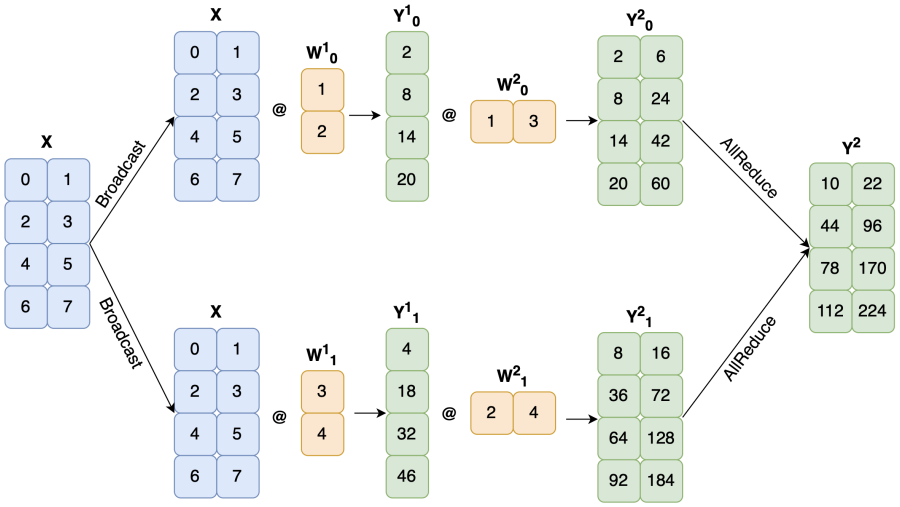


Figure 6.40: Composing row-wise and column-wise splits

Tensor Parallelism with Multi-Head Self-Attention

Next, we discuss how tensor parallelism applies to even more complex layers like self-attention. Recall that self-attention first projects the input into queries, keys, and values using three weight matrices, splits the result into several attention heads, computes attention independently within each head, concatenates the per-head outputs, and finally applies an output projection. In order

to distribute this computation across devices using tensor parallelism, we can split the query, key, and value projection matrices column-wise, with each device receiving the columns corresponding to one or more complete attention heads. Because a head's attention computation depends only on the queries, keys, and values belonging to that same head, every device can carry out the entire self-attention computation for its own heads, including the softmax and the weighted sum over values, without any communication with the other devices. The output projection matrix, which maps the concatenated per-head outputs back to the model's hidden dimension, is then split row-wise, since each device only holds the slice of the attention output corresponding to its own heads. Just as in the case of the MLP example, this combination of column-wise split followed by a row-wise split requires a single AllReduce, performed once at the end of the attention block, to sum the partial contributions from every device and recover the correct output (Figure 6.41).

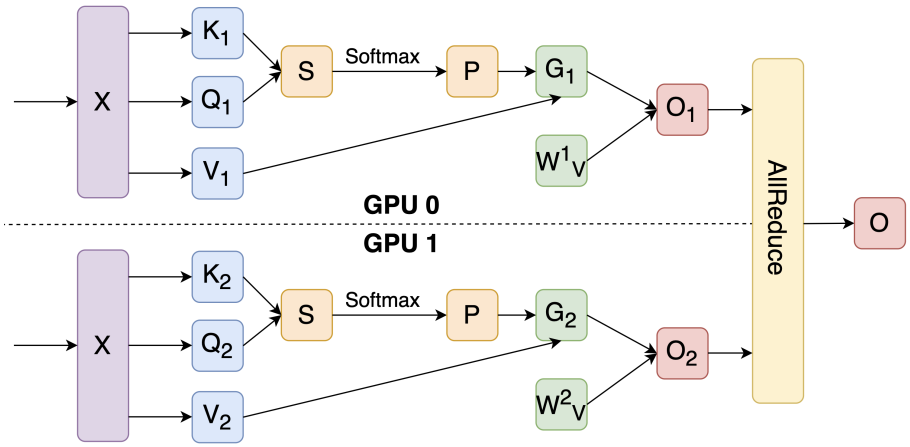


Figure 6.41: Tensor parallelism in multi-head self-attention

6.4.3 Communication Overhead of Tensor Parallelism

Tensor parallelism significantly reduces the per-GPU memory footprint of running a neural network layer, by distributing its tensors across multiple GPUs, while still ensuring correct outputs by carefully aggregating partial outputs across GPUs. However, this reduction in memory footprint is achieved at the cost of communication that is both more frequent and larger in volume than what is required by data parallelism or pipeline parallelism. Where pipeline parallelism only needs to exchange activations at the boundary between pipeline stages, and data parallelism only needs to synchronize gradients once per training step, tensor parallelism requires an AllReduce (or equivalent operation) to compute the aggregate output of every single layer that is partitioned across the tensor-parallel group. The size of the tensors exchanged in these AllReduce steps is also significant — each

AllReduce moves an activation tensor whose size is proportional to the batch size, and the hidden dimension of the model. Further, in pipeline parallelism, a stage can send its output activations or gradients to the neighbouring stage asynchronously, and immediately start working on the next micro-batch in its own pipeline stage, rather than waiting for that send/receive to complete. However, the AllReduce inside a tensor-parallel layer sits directly on the critical path: the next operation in the layer needs the fully reduced result before it can proceed, so this communication is difficult to hide behind useful computation. For these reasons, tensor parallelism is normally applied at a modest degree, commonly between 8 GPUs that are located within a GPU "node" and connected by a high bandwidth, low latency interconnect such as NVLink, rather than being stretched across GPUs in different servers connected only by a slower datacenter network.

Given this communication overhead, it is worth asking why tensor parallelism is used at all, rather than relying only on pipeline parallelism and data parallelism. This is because, while pipeline parallelism reduces the number of layers that must fit on a single device, it does nothing if a single layer by itself is too large for one GPU. Data parallelism, on the other hand, replicates the entire model on every device and only splits the training data, so it does not reduce the memory required to store the model at all, except in case of ZeRO optimizations. Tensor parallelism therefore plays a role that is complementary to the other techniques. It allows training to scale to models whose individual layers exceed the memory of one accelerator, and because activations as well as parameters are sharded across the tensor-parallel group, it substantially reduces the activation memory that each device must hold. This freed up memory can in turn be used to increase the per-device batch size, which improves the efficiency of the overall training process.

Practice Problem 6.4

Consider a simple two layer MLP with input X, intermediate activations Y, output Z, and weight matrices W_1 and W_2. The MLP computes outputs from inputs as follows: $Y = XW_1$, and $Z = YW_2$. For simplicity, ignore biases and non-linear activation functions, and assume all matrices are of size $N \times N$. Now, this MLP does not fit on a single GPU, due to which we use tensor parallelism to distribute the computation across two GPUs. There are two ways of doing this, as described below.

Method A: We split W_1 column-wise into two equal halves W_1^{left} and W_1^{right}, and W_2 row-wise into two equal halves W_2^{top} and W_2^{bottom}. The first GPU stores W_1^{left} and W_2^{top}. The other GPU stores the remaining halves of the two weight matrices.

Method B: We split W_1 row-wise into two equal halves W_1^{top} and W_1^{bottom}, and W_2 column-wise into two equal halves W_2^{left} and W_2^{right}. The first GPU stores W_1^{top} and W_2^{left}. The other GPU stores the remaining halves of the two weight matrices.

Across both methods, the input X is distributed to both GPUs, and the output Z is reassembled, appropriately. The intermediate output is exchanged between GPUs as required. Assume that every parameter occupies 1 byte of memory.

(i) For method A, write down the collective communication primitives that must be used to distribute X at the start, and reassemble Z at the end of the computation.

Solution: The entire input matrix X must be copied and distributed through a Broadcast. The partial

outputs $Z_0 = Y^{left} W_2^{top}$ and $Z_1 = Y^{right} W_2^{bottom}$ are assembled through REDUCE or ALLREDUCE.

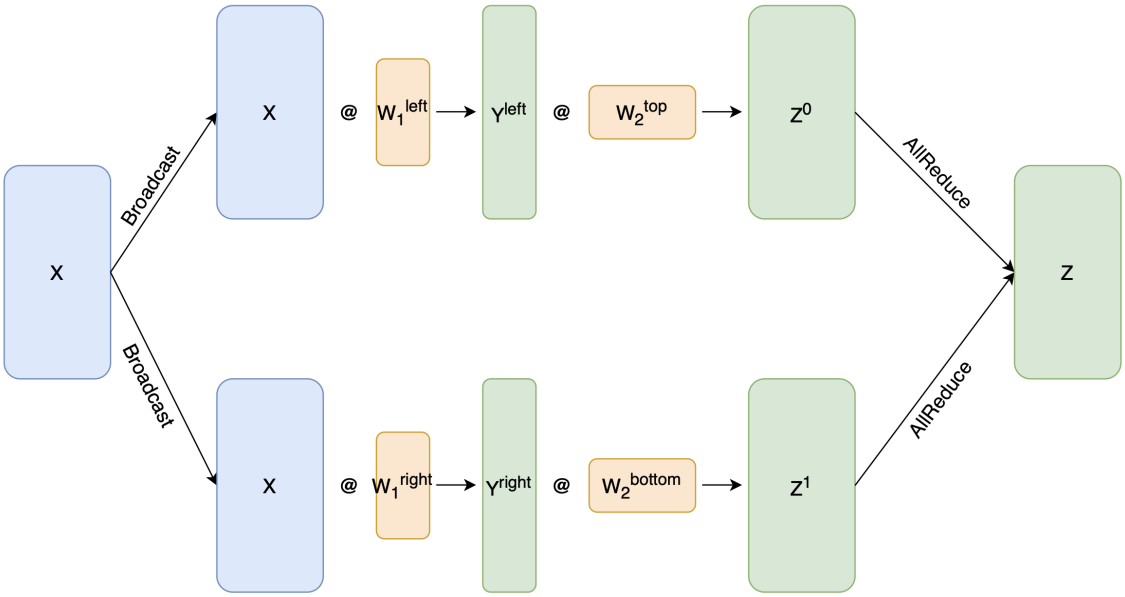


Figure 6.42: Method A

(ii) For method A, compute the amount of intermediate data (related to activation Y) that must be exchanged between both GPUs for correct operation of the MLP. You must compute the total data transmitted and received by both GPUs for the intermediate activations.
Solution: In method A, the first GPU produces Y^{left} and holds W_2^{top} and the second GPU produces Y^{right} and holds W_2^{bottom}. Because the way matrix multiplication works, $Z = Y^{left} W_2^{top} + Y^{right} W_2^{bottom}$. Therefore, no intermediate activations need to be exchanged between the layers.

(iii) For method B, write down the collective communication primitives that must be used to distribute X at the start, and reassemble Z at the end of the computation.
Solution: Because W_1 is split row-wise, the input X must be split column-wise through a SCATTER operation. For the final output Z, a GATHER or an ALLGATHER operation is performed.

(iv) For method B, compute the amount of intermediate data (related to activation Y) that must be exchanged between both GPUs for correct operation of the MLP. You must compute the total data transmitted and received by both GPUs for the intermediate activations.
Solution: In method B, the first GPU computes a partial sum $Y^0 = X^{left} W_1^{top}$ and the second GPU computes a partial sum $Y^1 = X^{right} W_1^{bottom}$, where $Y = Y^0 + Y^1$. Because W_2 is split column-wise in the next layer, both GPUs need the entirety of the full $N \times N$ matrix Y to proceed, requiring an ALLREDUCE of Y^0 and Y^1. Total amount of data transferred for each GPU = $2N^2$ bytes.

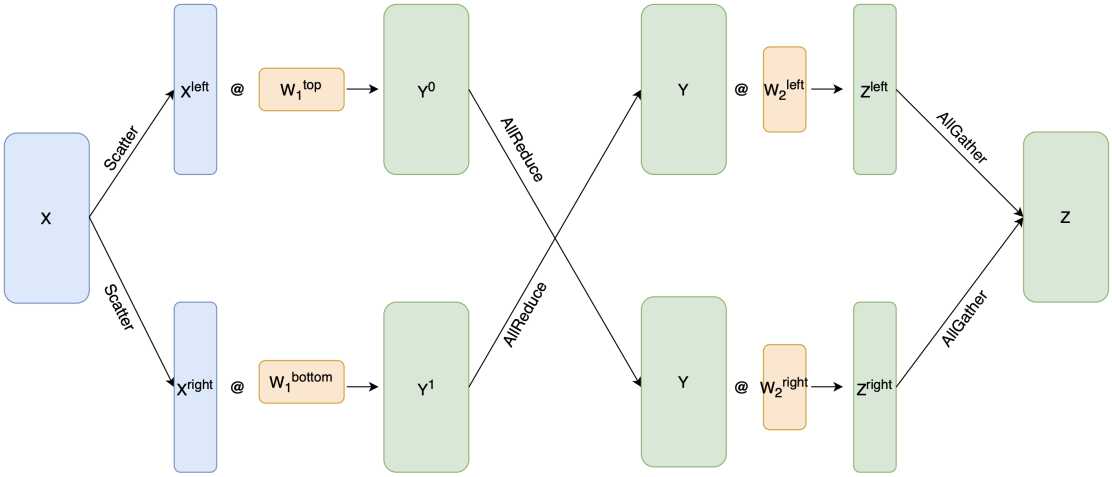


Figure 6.43: Method B

Programming Assignment 6.3: Tensor Parallelism

In this assignment, you will implement tensor parallelism to split individual weight matrices across multiple GPUs rather than splitting layers. You will compute a matrix product $C = AB$ by sharding the weight matrix B column-wise into `num_gpus` equal slices, spawning one worker process per shard, and having each worker independently compute its partial output using the full input A and just its own slice of B. Workers report their results back to a coordinator through a shared `mp.Queue`, and the coordinator's job is to collect all of them and concatenate the partial results in rank order along the column axis to reconstruct the full output C. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/tensor_parallelism

6.5 Other Types of Parallelism

So far, we have studied how we can parallelize training across GPUs by distributing work along the axes of training data chunks (data parallelism), model layers (model / pipeline parallelism), or a single layer's tensors (tensor parallelism). We now examine other, more advanced forms of parallelism that are employed in recent work to distribute training across multiple devices.

6.5.1 Expert Parallelism

As we saw in Chapter-4, in an MoE model, the single large MLP is replaced by a collection of many smaller MLPs, called experts, and a lightweight network decides, for each token, which one or few experts should process it. Only the chosen experts perform computation for that token, so even if the total number of parameters in the model is very large, the amount of computation performed per token stays comparable to a dense model of a much smaller size.

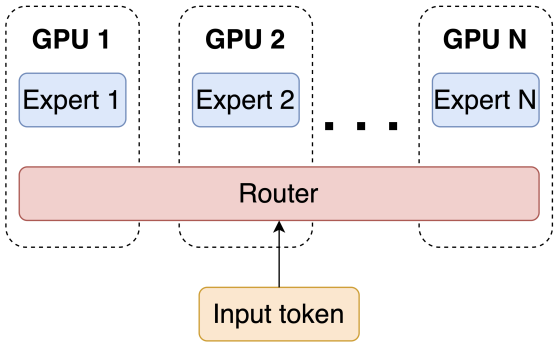


Figure 6.44: Expert parallelism

This architecture maps naturally onto the expert parallelism (EP) strategy. Since each expert is an independent network, different experts can simply be placed on different GPUs, and the router determines which GPU a given token's computation should be sent to (Figure 6.44). Instead of splitting a single dense layer across GPUs, as tensor parallelism does, expert parallelism splits the set of experts across GPUs, and tokens are routed dynamically to wherever their assigned expert lives.

The mechanics of this routing are worth understanding because they rely on a communication pattern that is quite different from those seen so far. After the router computes the expert(s) that should process each token, those tokens must be delivered to the GPUs that host the selected experts. This is accomplished with an AllToAll communication operation. In an AllToAll collective communication, every GPU simultaneously exchanges distinct data with every other GPU, where each GPU sends a different subset of its tokens to each peer and receives different tokens in return (Figure 6.45). The result is that all tokens assigned to a given expert arrive on the GPU that hosts that expert, allowing each expert to process its complete batch locally. Once expert computation finishes,

a second AllToAll performs the reverse exchange, returning the processed token representations to the GPUs that originally owned them, so that the remainder of the transformer layer can execute normally. This pair of dispatch and combine AllToAll operations forms the dominant communication overhead of expert parallelism, and this pattern is fundamentally different from the AllReduce operations used in tensor parallelism or the point-to-point transfers of pipeline parallelism.

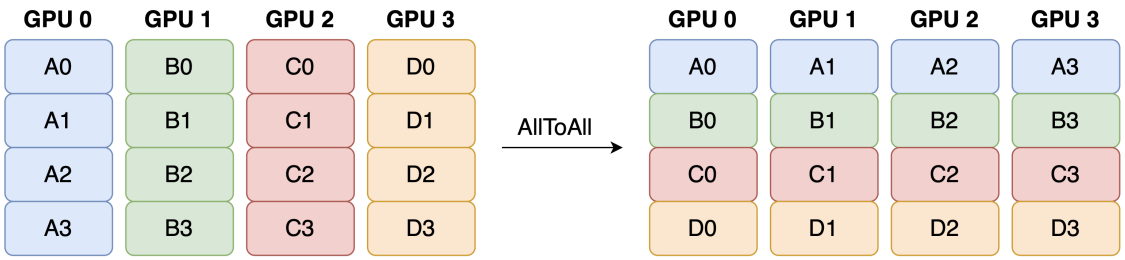


Figure 6.45: AllToAll

A recurring challenge in MoE systems is load balancing. Because the router learns which expert to send each token to, nothing prevents it from favoring a small subset of experts, leaving others starved of tokens, leaving a few GPUs underutilized and others overloaded. This both wastes hardware and degrades model quality, since underused experts receive little training signal. Practical MoE systems address this with auxiliary load-balancing losses added during training, which penalize the router for uneven usage of experts, and with capacity limits, which cap the number of tokens any single expert is allowed to accept in a given batch and drop or reroute the overflow.

Expert parallelism is rarely used by itself. In large MoE models, it is often combined with data, tensor and pipeline parallelism. Because the AllToAll communication in expert parallelism is comparatively heavy, it is, like tensor parallelism, usually restricted to GPUs within a node or a small number of well-connected nodes.

6.5.2 Context Parallelism

Data parallelism distributes different chunks of training data to different GPUs. In contrast, context parallelism (CP) distributes the sequence of tokens that make up a single training example itself to different GPUs. This becomes necessary as models are trained with increasingly long context windows, sometimes spanning hundreds of thousands or even millions of tokens. In such cases, the memory required to store the activations (e.g., KV cache) of a single sequence can exceed the capacity of one GPU, even if the model's parameters fit comfortably in GPU memory.

In context parallelism, a single long sequence is split into contiguous chunks, and each chunk is placed on a different GPU. For most components of a transformer, such as the feed-forward layers and the layer normalization, this split is straightforward to implement, because these operations act on each token independently and require no communication between chunks. The difficulty arises in

the attention mechanism, because computing attention for a given token requires access to the keys and values of every other token in the sequence, not just the tokens in its own chunk. For example, if GPU 2 holds tokens 1001 through 2000 of a sequence, it still needs the key and value vectors for tokens 1 through 1000 held by GPU 1, and for every other chunk held elsewhere, in order to compute attention correctly. How can we correctly compute attention in this case?

Block-Wise Attention

The standard solution to parallelize attention across the context dimension is by using a technique called block-wise attention. We split the K and V matrices into blocks and compute attention block by block. Next, we combine the partial attention results across blocks by rescaling the softmax — much like in Flash Attention, now applied across devices rather than across SRAM tiles.

Formally, let a query block $Q_i \in \mathbb{R}^{B\times d}$ be resident on some device, and let the full key/value sequence be split into blocks $K_1, \ldots, K_M$ and $V_1, \ldots, V_M$, each of size $B \times d$, distributed across M devices. Naively, softmax attention requires the full row of logits before normalizing:

$$\mathrm{Attn}(Q_i, K, V) = \mathrm{softmax}\left(\frac{Q_i K^\top}{\sqrt{d}}\right) V.$$

Block-wise attention instead computes this incrementally. For each incoming block j, define the local (unnormalized) statistics

$$S_{ij} = \frac{Q_i K_j^\top}{\sqrt{d}} \in \mathbb{R}^{B\times B}, \qquad m_{ij} = \mathrm{rowmax}(S_{ij}), \qquad P_{ij} = \exp(S_{ij} - m_{ij}).$$

We maintain running statistics after processing blocks $1, \ldots, j$: a running max $m_i^{(j)}$ and a running normalizer $\ell_i^{(j)}$. Both of these are vectors of size B, one for each row in the local query block. We also maintain a running (unnormalized) output accumulator $O_i^{(j)}$. When block $j{+}1$ arrives, we update these via the standard online-softmax rescaling:

$$m_i^{(j+1)} = \max\left(m_i^{(j)}, m_{i,j+1}\right),$$

$$\ell_i^{(j+1)} = e^{m_i^{(j)} - m_i^{(j+1)}}\, \ell_i^{(j)} \;+\; e^{m_{i,j+1} - m_i^{(j+1)}} \sum_k P_{i,j+1}[:,k],$$

$$O_i^{(j+1)} = e^{m_i^{(j)} - m_i^{(j+1)}}\, O_i^{(j)} \;+\; e^{m_{i,j+1} - m_i^{(j+1)}}\, P_{i,j+1} V_{j+1}.$$

Note that the subscript i always refers to the query block Q_i resident on our device, and this never changes throughout the computation. The subscript j refers to whichever key/value block is currently being processed — the one that has just arrived from another device in the ring. With that in mind, $S_{ij} \in \mathbb{R}^{B\times B}$ is simply the matrix of raw, unnormalized attention logits between the resident queries Q_i and the incoming keys K_j: its entry $S_{ij}[r,k]$ is the (scaled) dot product between query r of block i and key k of block j. From this, we compute locally $m_{ij} = \mathrm{rowmax}(S_{ij}) \in \mathbb{R}^B$, a length-$B$

vector holding, for each query row r, the largest logit that query achieves against this one block j alone; it is a per-block statistic, recomputed from scratch every time a new block arrives. Next, $P_{ij} = \exp(S_{ij} - m_{ij})$ shifts the logits of block j by that block's own row-max before exponentiating. Its entry $P_{ij}[r,k]$ is therefore the unnormalized attention weight that query r places on the k-th key of block j, and summing $P_{i,j+1}[:,k]$ over all $k = 1,\ldots,B$ — as appears in the $\ell_i^{(j+1)}$ update — just gives the row-sum of $P_{i,j+1}$, i.e., the total unnormalized attention mass that the new block $j+1$ contributes to each query row. Next, we compute the local attention output as $P_{i,j+1}V_{j+1}$, and accumulate it with the running output after rescaling.

After the final block M, the correctly normalized output is recovered as $O_i^{(M)}/\ell_i^{(M)}$. Because the rescaling only ever depends on the running max and sum, blocks can be streamed in any order and combined incrementally without ever materializing the full $N \times N$ attention matrix. This is exactly what makes it possible to distribute the attention computation across devices.

Ring Attention

Ring Attention is a specific, communication-efficient way of implementing block-wise attention for context parallelism. The GPUs participating in context parallelism are arranged in a logical ring, and at each step, every GPU sends its local chunk of key and value vectors to its neighbour in the ring while simultaneously receiving a chunk from its other neighbour. As these key and value chunks circulate around the ring, each GPU accumulates the partial attention output against every chunk it has seen so far, and after enough steps to complete a full loop around the ring, every GPU has computed the full attention output for its local queries against the entire sequence (Figure 6.46). This design overlaps communication with computation quite effectively, since a GPU can be computing attention against the chunk it currently holds while simultaneously sending and receiving the next chunk, so the added latency from the ring communication can largely be hidden. After one full loop around the ring, every node has seen every K, V block once, and has therefore computed the exact, fully normalized attention output for its local queries against the entire sequence.

Now, whether the overlap between ring communication and block-wise attention compute is effective depends on the choice of block size B relative to the hidden dimension d and the sequence length N. For a block of size B, the attention matrix for that block is $B \times B$ rather than the full $N \times N$, so the compute cost per block is

$$\underbrace{2B^2d}_{QK^\top} + \underbrace{2B^2d}_{(\text{attn})\cdot V} = 4B^2d \text{ FLOPs}$$

The communication cost of each step sending one K block and one V block to the next device is

$$\underbrace{Bd}_{K} + \underbrace{Bd}_{V} = 2Bd$$

elements transmitted. Since the device simultaneously receives one K block and one V block from

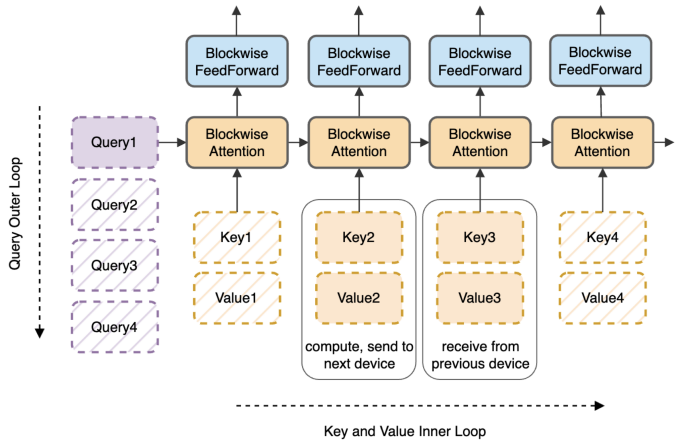


Figure 6.46: Ring attention; Image credit: [Liu et al., 2023]

the previous device, the total bidirectional traffic is

$$2Bd \text{ sent} + 2Bd \text{ received} = 4Bd$$

elements per step.

Now, the compute can fully hide the communication latency if:

$$\frac{4B^2d}{\text{FLOP/s}} \gtrsim \frac{4Bd}{\text{bandwidth}} \iff B \gtrsim \frac{\text{FLOP/s}}{\text{bandwidth}}$$

In other words, larger block sizes B increase compute quadratically (B^2) while communication only grows linearly (B), so beyond some threshold block size (set by the compute-to-bandwidth ratio of the hardware), the K/V transfer for the next block can be completely hidden behind the attention computation for the current block (Figure 6.47).

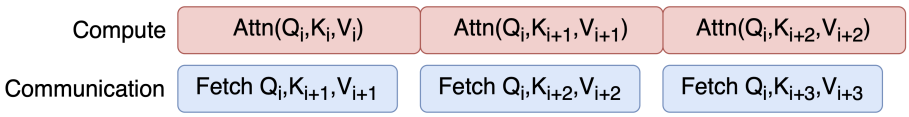
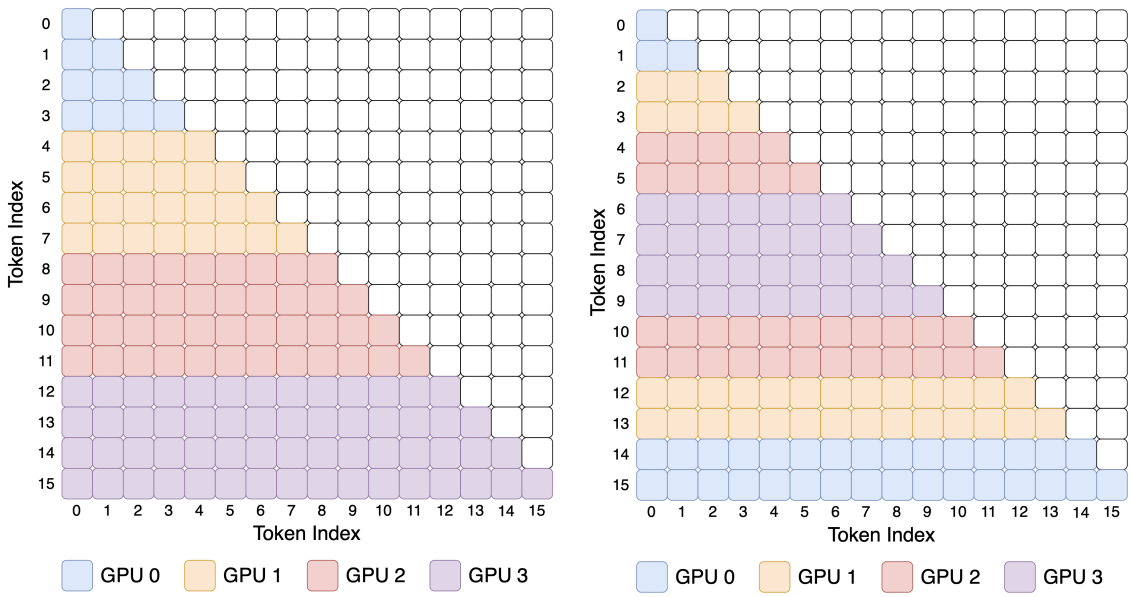


Figure 6.47: Overlapping compute and communication

One complication with ring attention arises when attention is causal, as is the case with autoregressive language modeling. Recall that a token can only attend to earlier tokens, so the attention mask is lower-triangular. If the sequence is divided into contiguous chunks and each chunk assigned to a device in order, the amount of useful work is wildly unbalanced across chunks (Figure 6.48a).



(a) Unbalanced load

(b) Balanced load

Figure 6.48: Load balancing under a causal mask

The device holding the first chunk of tokens only ever computes attention against a small triangular slice, while the device holding the last chunk must compute attention against nearly the entire sequence. So the last device does far more work than the first, and the ring stalls waiting on the slowest node. The fix is to rebalance the assignment of token blocks to devices. Instead of assigning them in strict contiguous order, a common idea is to give each device one block from early in the sequence and one from late in the sequence (e.g., as shown in Figure 6.48b). This equalizes the amount of attention compute each device performs per ring step, instead of leaving early-sequence devices idle while late-sequence devices are overloaded.

As with other forms of parallelism, context parallelism is typically composed with data, tensor, and pipeline parallelism rather than used in isolation, and it is particularly important during the pre-training and fine-tuning of long-context models.

6.5.3 Sequence Parallelism

Tensor parallelism (TP) splits the large weight matrices in the transformer along the hidden dimension h. However, operations that follow the input-weight matrix multiplication, such as LayerNorm, Dropout, and residual additions are not naturally split in this way. For example, LayerNorm computes statistics over the hidden dimension of each token,

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

where μ and σ^2 are computed over the h hidden features of that token. Although these operations require relatively little computation, they cannot be parallelized across the hidden dimension, unlike the input-weight matrix multiplication. Therefore, these operations are replicated across all GPUs in the tensor-parallel group, and this can consume a substantial amount of GPU memory.

Sequence parallelism (SP) removes this redundancy by splitting the activations along the sequence dimension s, and computing operations like the LayerNorm over the entire hidden dimension, but only for a subset of tokens in the sequence. Therefore, a GPU only needs its own sequence slice and the complete hidden vector for each token; it does not need the other GPUs' tokens. The challenge with composing TP and SP is that the large linear layers in transformers are already tensor-parallelized along the hidden dimension. Therefore, we must periodically convert between two different activation layouts, one of which shards across the hidden dimension, and the other across the sequence dimension.

Consider the example of a 2-layer MLP, which we saw in the context of tensor parallelism (Figure 6.40), and let us understand how one can do a LayerNorm on its output. As before, we apply a column-wise split of the weight matrix in the first linear layer, followed by a row-wise split of the weight matrix in the second linear layer. After this, instead of applying an AllReduce, we can simply perform a ReduceScatter on the output of the second layer, let each GPU obtain a row-wise shard along the sequence dimension, and then apply SP to compute LayerNorm in parallel. Further, if the input to the MLP layer must also go through a LayerNorm, we can do a row-wise sharding of the input for LayerNorm using SP, followed by a 2-layer MLP using TP, and a LayerNorm using SP once again, as shown in Figure 6.49.

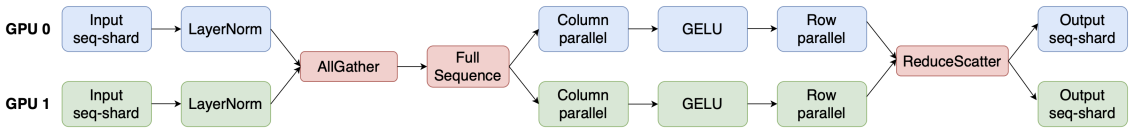


Figure 6.49: Sequence parallelism

6.5.4 Hybrid Parallelism

So far, we have studied different parallelism techniques that distribute the work required to train a large model across many dimensions. Each technique produces certain benefits in terms of reduced resource usage, but also adds costs in terms of communication overheads. Modern distributed training systems often combine these techniques, into various models of hybrid parallelism. One example is shown in Figure 6.50. Here, the model is replicated into multiple data-parallel partitions. Within each data-parallel group, the model is first partitioned across the model dimension using

pipeline parallelism, and within each layer, tensors are sharded using tensor parallelism. Parallelism strategies can be combined in several such ways, keeping in mind the resource capacity limits and communication bandwidths within the GPU cluster.

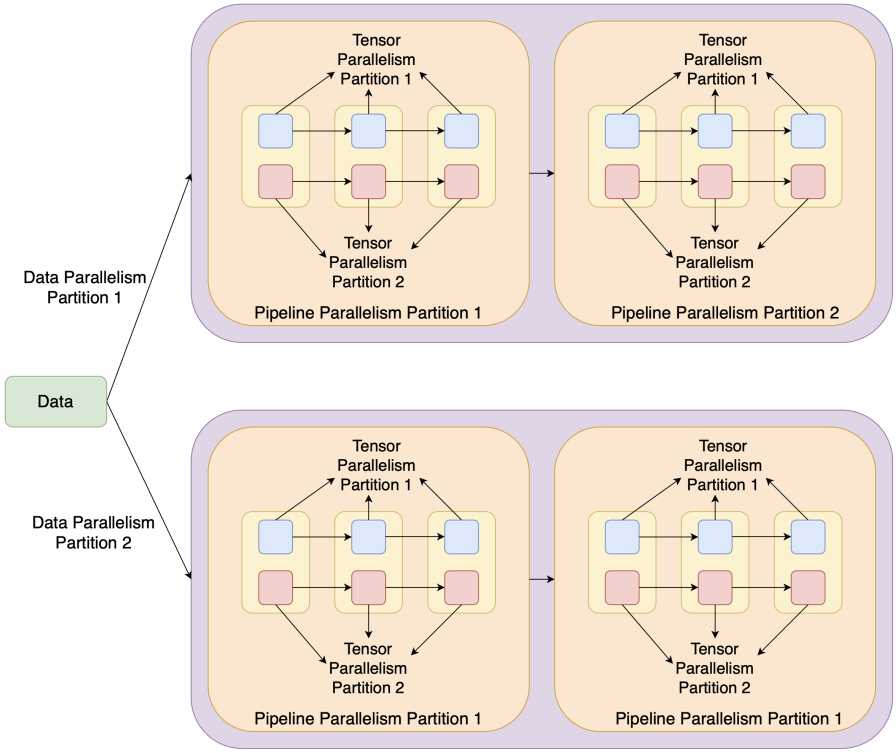


Figure 6.50: Hybrid parallelism

How should the parallelism techniques be combined with each other, and in what order should they be nested? Tensor parallelism requires very frequent communication between GPUs that share a layer, since an AllReduce operation is needed at nearly every layer boundary. This makes it best suited to be used as the innermost parallelism technique, across GPUs that sit inside the same physical server and connected by high-bandwidth links such as NVLink. Expert parallelism, which requires expensive AllToAll communication is also best deployed within a small set of tightly coupled GPUs. Pipeline parallelism, by contrast, only needs to pass activations and gradients between adjacent stages, once per micro-batch, which is a much lighter communication pattern. This makes it a good fit for being deployed between GPUs across different server nodes, where the network is slower than the intra-node interconnect. Data parallelism usually sits at the outermost level: entire groups of GPUs, each running a full tensor-and-pipeline parallel copy of the model, are replicated to

process different chunks of the training batch, and gradient synchronization happens only once per optimizer step. ZeRO is layered on top of this hybrid setup, where each data-parallel replica, which may itself be a tensor-and-pipeline-parallel unit spanning many GPUs, shards its optimizer states and gradients across the other replicas performing the same role. This allows the usage of larger per-replica batch sizes and larger models than would otherwise fit in a single GPU, without having to increase the degree of tensor or pipeline parallelism, both of which have diminishing returns and rising communication overhead as they scale up.

Choosing the degree of parallelism along each axis is one of the central engineering problems in large-scale training. The tensor-parallel degree is usually capped by the number of GPUs in a single node, because of the communication demands described above. The pipeline-parallel degree is chosen based on how many nodes are available and how large the resulting pipeline bubble becomes; too many stages relative to the number of micro-batches leads to GPUs sitting idle for a large fraction of the time. The data parallelism degree is chosen to efficiently use all GPUs in the cluster, after accounting for the degrees of tensor and pipeline parallelism, and also to optimize training time.

Modern large-scale training rarely relies on a single parallelism strategy. A large language model with mixture-of-experts layers and a long context window might simultaneously use tensor parallelism within a node, pipeline parallelism across nodes, expert parallelism for its MoE layers, context parallelism to handle long sequences, and data parallelism, sharded with ZeRO, to scale across the full cluster. Each of these strategies is designed to distribute a different resource, and each has its own characteristic communication pattern and its own sensitivity to network bandwidth and latency. The engineering task of designing a training system is therefore less about choosing a single best strategy, and more about deciding how to nest these strategies against the physical structure of the cluster: which axes of parallelism should be confined to the fast interconnect within a node, which can tolerate the slower connections between nodes, and how much of each is needed to make the model and its activations fit in memory in the first place.

Practice Problem 6.5

Consider a large language model with N bytes of total parameters trained on a distributed GPU cluster using D-way data parallelism, T-way tensor parallelism, P-way pipeline parallelism. The model has L layers in total, evenly distributed across all pipeline stages. Tensor parallelism is applied uniformly applied to every layer.

(i) What is the total memory (in bytes) required per GPU to store the model parameters, and their gradients per GPU?

Solution: The model has N parameters in total. With T-way tensor parallelism and P-way pipeline parallelism, each GPU holds $\frac{N}{T \cdot P}$ bytes.

(ii) During the forward pass, each layer produces activations of size E bytes per token. Let B denote the number of tokens in a single micro-batch processed by one GPU. How much data (in bytes) does a GPU transmit to the next pipeline stage during the forward pass of one micro-batch?

Solution: At the boundary between pipeline stages, the output activations of the last layer in one stage

are sent to the first layer of the next stage. Each layer produces E bytes per token, and the micro-batch contains B tokens, so the data transmitted is $B \cdot E$ bytes.

(iii) Suppose each GPU has G bytes of memory available for storing activations. Retaining activations for the backward pass costs M bytes per token per layer (already accounting for the reduction due to tensor parallelism). Determine the maximum number of tokens B_μ that can fit in a single micro-batch per GPU. Write B_μ in terms of G, P, M, L.
Solution: Each pipeline stage holds L/P layers. The total activation memory required is $B_\mu \cdot M \cdot \frac{L}{P} \leq G$. Solving, $B_\mu = \left\lfloor \frac{GP}{M \cdot L} \right\rfloor$.

(iv) Continuing from the previous part, derive the maximum global batch size B_{global} across the entire cluster.
Solution: Data parallelism replicates the pipeline across D workers, each processing B_μ tokens independently before an AllReduce synchronises gradients. $B_{\text{global}} = D \cdot \left\lfloor \frac{GP}{ML} \right\rfloor$

Practice Problem 6.6

Consider training the **175B-parameter** GPT-3 language model on a dataset of **500 billion (500B) tokens**. Assume the following setup:

- **Model states**: Each parameter requires 2 bytes (BF16 weights) + 2 bytes (BF16 gradients) + 12 bytes (FP32 optimizer state) = 16 bytes per parameter.
- **Cluster**: 128 GPU nodes, each with 8 GPUs, each GPU having 80 GB of memory.
- **Parallelism**: 8-way Tensor Parallelism (TP) within each node, 16 Pipeline Parallelism (PP) stages of 6 layers each, and 8-way Data Parallelism (DP).
- **Activations**: Each transformer layer requires $12 \times E$ bytes per token for forward/backward passes, where $E = 12288$ (the hidden dimension). The TP degree divides activation memory proportionally.
- **Compute**: The cluster achieves 30% utilization of its theoretical peak of 1024×312 TFLOPS/s. Training requires approximately $6 \times N_{\text{params}}$ FLOPs per token.

(i) Compute the total model state memory (in TB) for the 175B model. Then, determine how much memory each GPU must store given the TP and PP degrees, and state how many GB remain free on each GPU for activations.
Solution: Total model state: $175 \times 10^9 \times 16$ bytes = 2,800 GB = 2.8 TB. With 8-way TP and 16-way PP, each GPU stores:

$$\frac{2{,}800 \text{ GB}}{8 \times 16} = \frac{2{,}800}{128} = 21.875 \approx 22 \text{ GB per GPU.}$$

Each GPU has $80 - 22 = 58$ GB free for activations.

(ii) Given 6 layers per PP stage, compute the activation memory per token per GPU (in KB). Then determine the maximum number of tokens that can fit in a single GPU's batch, and derive the global batch size (in tokens) across all 8 DP replicas.
Solution: Activation memory per token (before TP), across 6 layers:

$$12 \times 12{,}288 \times 2 \text{ bytes} \times 2 \text{ (fwd+bwd)} \times 6 \text{ (layers)} = 3{,}538{,}944 \text{ bytes} \approx 3{,}456 \text{ KB.}$$

After dividing by 8 for TP:

$$\frac{3{,}456 \text{ KB}}{8} = 432 \text{ KB per token per GPU.}$$

Tokens per GPU:

$$\frac{58 \times 1024 \text{ MB} \times 1024 \text{ KB/MB}}{432 \text{ KB}} = \frac{59{,}392 \text{ MB} \times 1024}{432 \text{ KB}} \approx 140{,}780 \approx 137\text{K tokens per GPU.}$$

Global batch size with 8-way DP:

$$137{,}000 \times 8 \approx 1{,}096\text{K tokens} \approx 1.1\text{M tokens globally.}$$

(iii) How many optimizer steps are required to train on the full 500B-token dataset?
Solution:

$$\text{Steps} = \frac{500 \times 10^9}{1{,}100{,}000} \approx 454{,}545 \approx 450\text{K steps.}$$

(iv) Compute the time per token (in microseconds), the time per batch (in seconds), and the total estimated training time (in days).
Solution: FLOPs per token: $6 \times 175 \times 10^9 = 1{,}050$ GFLOPs = 1.05 TFLOPs/token. Effective cluster throughput:

$$0.30 \times 1{,}024 \times 312 \text{ TFLOPs/s} = 95{,}846.4 \text{ TFLOPs/s.}$$

Time per token:

$$\frac{1.05 \text{ TFLOPs}}{95{,}846.4 \text{ TFLOPs/s}} \approx 11 \ \mu\text{s per token.}$$

Time per batch:

$$11 \ \mu\text{s} \times 1{,}100{,}000 \approx 12.1 \text{ seconds.}$$

Total training time:

$$12.1 \text{ s} \times 450{,}000 \text{ steps} \approx 5{,}445{,}000 \text{ s} \approx 64 \text{ days.}$$

6.6 Case Studies

Having discussed the individual forms of parallelism—data, tensor, pipeline, expert, context, and sequence parallelism—along with communication and memory trade-offs that each entails, it is useful to see how real production-scale systems combine these techniques. In this section, we examine two large language model training efforts, DeepSeek-V3 [DeepSeek-AI, 2025] and Llama 3 [Grattafiori et al., 2024], that made rather different choices about which forms of parallelism to emphasize, largely as a consequence of differences in model architecture, cluster size, and hardware interconnect. Comparing them side by side illustrates that there is no universally optimal distributed training recipe; the right combination depends heavily on the shape of the model being trained and the network topology available to the training cluster.

6.6.1 DeepSeek-V3

DeepSeek-V3 was trained on a cluster of 2048 H800 GPUs, organized into servers of eight GPUs each connected by NVLink. Because H800 GPUs have deliberately restricted NVLink and interconnect bandwidth relative to their H100 counterparts, the DeepSeek team designed their parallelism strategy to minimize the most communication-intensive operations. In particular, they avoided tensor parallelism entirely. They instead rely on a combination of expert, pipeline, and data parallelism.

Since DeepSeek-V3 is a Mixture-of-Experts (MoE) model, expert parallelism plays a central role in its training strategy. The model uses 64-way expert parallelism, with the experts spread across eight nodes. This means that tokens must be routed across node boundaries during the dispatch and combine phases of MoE computation, and this AllToAll communication pattern becomes one of the dominant costs in the training pipeline. On top of expert parallelism, DeepSeek-V3 uses 16-way pipeline parallelism to partition the model's layers across pipeline stages, and ZeRO-1 style data parallelism to shard optimizer states across data-parallel replicas.

Because MoE AllToAll communication and pipeline communication both compete with computation for GPU resources, DeepSeek-V3 introduces a pipeline scheduling algorithm called DualPipe. As we saw before, DualPipe splits the backward pass into two separate pieces: backward-for-input and backward-for-weight. Along with this, it also feeds micro-batches from both ends of the pipeline simultaneously, to minimize the idle bubble duration.

At the lowest level, the overlap between computation and communication within a single GPU is achieved through warp specialization. Rather than having every warp on a streaming multiprocessor (SM) execute the same instructions in parallel, the workload is partitioned so that different warps are responsible for different roles — some warps are dedicated to computation while others are dedicated to communication, and these warps coordinate using barrier instructions to synchronize as producers and consumers of data. Because MoE communication in DeepSeek-V3 must traverse both NVLink for GPUs within the same node, and InfiniBand for GPUs across nodes, depending on where a given expert happens to reside, separate warps are further specialized to handle NVLink transfers and InfiniBand transfers respectively. The number of warps allocated to each communication task is not

fixed but is dynamically adjusted so that communication can be fully overlapped with, and hidden behind, the ongoing computation on the SM.

Finally, DeepSeek-V3 makes extensive use of mixed precision training with FP8 arithmetic to reduce both compute time and memory footprint (Figure 6.51). Rather than using a single scale factor for an entire tensor, it applies quantization at the tile or block level, which better accommodates the wide dynamic range of activations and gradients, and reduces the quantization error that would otherwise arise from using a single per-tensor scale. The forward pass, the weight-gradient computation, and the input-gradient computation each cast their inputs down to FP8 for the matrix multiplication itself, while accumulation is carried out in FP32 to preserve accuracy. A master copy of the weights and the optimizer states are also maintained at higher precision, BF16 or FP32, and only cast down to lower precision as needed for the matrix multiplications, which prevents the accumulation of quantization error across many training steps.

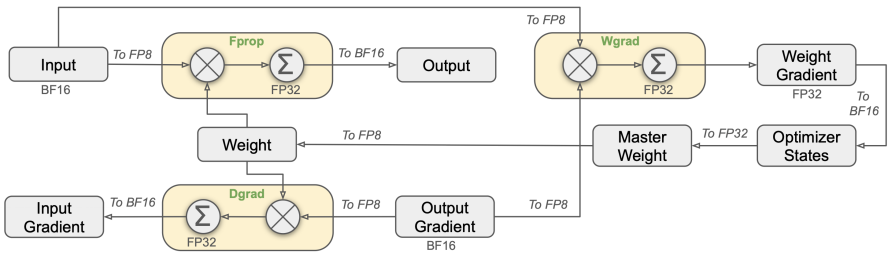


Figure 6.51: Mixed precision framework; Image credit: [DeepSeek-AI, 2025]

6.6.2 Llama 3

Llama 3, in contrast to DeepSeek-V3, is a dense (non-MoE) Transformer, with its largest variant containing 405 billion parameters and supporting context lengths of up to 128K tokens. It was trained on a considerably larger cluster of 16,384 H100 GPUs, again organized into servers of eight GPUs connected via NVLink. The model parameters for the family of models are shown in Table 6.1.

Before committing to a particular model size for the 405B run, the Llama 3 team first ran a large sweep of much smaller training runs to empirically determine how loss depends on model size and training data. For a range of fixed compute budgets, they trained several models of different sizes, each consuming that same budget but trading off model size against number of training tokens, i.e., a larger model trained on fewer tokens, vs. a smaller model trained on more tokens, for the same total FLOPs. Plotting validation loss against the number of training tokens for each compute budget produces a family of U-shaped curves (Figure 6.52), one curve per compute budget. Each curve has a clear minimum, corresponding to the best trade-off between model size and data quantity achievable at that compute budget—spending too few tokens leaves an oversized model undertrained, while

	8B	70B	405B
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function	SwiGLU		
Vocabulary Size	128,000		
Positional Embeddings	RoPE (θ = 500,000)		

Table 6.1: Model parameters for the Llama 3 family of models

spending too many tokens wastes compute on a model too small to make further use of the data.

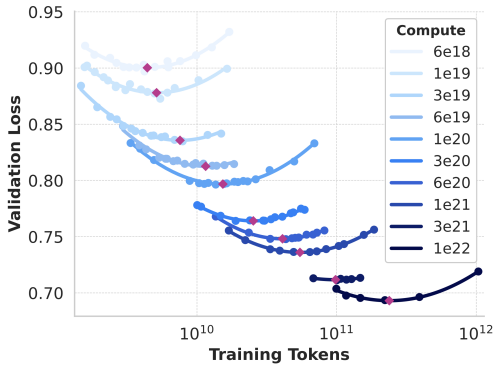


Figure 6.52: IsoFLOPs curves: validation loss vs. training tokens for a range of fixed compute budgets; Image credit: [Grattafiori et al., 2024]

By reading off the training-token count at the minimum of each curve, one obtains the compute-optimal number of training tokens as a function of compute budget. Plotting this relationship on a log-log scale (Figure 6.53) reveals that compute-optimal training tokens scale as a power law in the compute budget. Fitting a line to this trend then allows the relationship to be extrapolated far beyond the scale of the experiments actually run: given the intended compute budget for the full-scale model, the fitted power law directly predicts the compute-optimal number of training tokens, and hence the corresponding model size, without needing to run the full IsoFLOPs sweep at that scale. This extrapolation procedure is what determined the specific sizes and token budgets chosen for the Llama 3 model family.

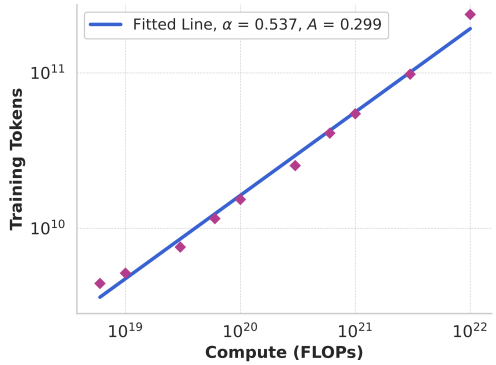


Figure 6.53: Compute-optimal training tokens vs. compute budget; Image credit: [Grattafiori et al., 2024]

Because Llama 3 is dense and trained on GPUs with full-bandwidth NVLink and InfiniBand, and because it must additionally support very long context windows, its parallelism strategy differs substantially from DeepSeek-V3's. Llama 3 adopts what its authors describe as 4D parallelism, combining tensor, context, pipeline, and (fully sharded) data parallelism along four separate axes. Tensor parallelism is applied 8-way within a node, taking advantage of the high-bandwidth NVLink connections among the eight GPUs on a single server to absorb the frequent AllReduce operations that tensor parallelism requires. Pipeline parallelism is applied 16-way across nodes, using an interleaved pipeline schedule with balanced stages to reduce the pipeline bubble. Context parallelism is applied 16-way, and is introduced specifically to handle Llama 3's long-context training. Llama 3 uses grouped-query attention, in which several query heads share a single key/value head. The number of key/value heads is therefore much smaller than the number of query heads, so the key/value tensors that must be communicated are correspondingly smaller than the query tensors that remain local to each device. Finally, data parallelism is realized through Fully Sharded Data Parallelism (FSDP), which shards model parameters, gradients, and optimizer states across the data-parallel replicas.

The effectiveness of this entire parallelism configuration is summarized by Model FLOPs Utilization (MFU), which measures what fraction of a GPU's theoretical peak FLOPs is actually realized once all the overheads introduced by communication and pipeline bubbles are accounted for. Across different stages of training, with different combinations of tensor, context, pipeline, and data parallelism degree and different sequence lengths, Llama 3's training achieved an MFU of roughly 38 to 43 percent in BF16.

6.7 Summary

In this chapter, we looked at distributed training from a systems perspective, focusing on how the work of training a model can be split across many GPUs once it no longer fits comfortably on a single device. Training itself consists of a forward pass, a backward pass, and an optimizer step, and activations must be held in memory until the backward pass consumes them, which is why training is far more memory-hungry than inference. Once a model outgrows the compute and memory limits of a single device, training must be distributed, and we introduced two foundational strategies to do the same: data parallelism, which replicates the full model across GPUs and splits the training data, and model parallelism, which splits the model itself so that no device runs the complete model.

We first examined data parallelism, where every GPU holds an identical model copy and trains on its own chunk of the training data, with gradients synchronized via AllReduce after every step. We saw how mini-batch size trades off convergence speed against activation memory, and how optimizations like gradient accumulation help reach larger effective batch sizes. Data parallelism, however, scales well only up to a point, because the overhead to reduce gradients across all nodes eventually degrades performance.

We then studied ZeRO, which removes the memory redundancy of data parallelism by sharding model states across the data-parallel group and reconstructing them only when needed. ZeRO-1 shards optimizer states using a ReduceScatter and AllGather; ZeRO-2 additionally shards gradients at no extra communication cost; and ZeRO-3 shards parameters themselves, gathering them layer by layer during the forward and backward passes, at the cost of substantially more frequent communication.

Next, we turned to pipeline parallelism, which partitions the model layer-wise into stages across devices. A naive implementation leaves most devices idle due to a bubble that grows with pipeline depth. GPipe shrinks this bubble by streaming micro-batches through the pipeline, but still requires a full flush before each update. PipeDream instead interleaves forward and backward passes asynchronously using weight stashing and vertical sync; PipeDream-Flush combines 1F1B scheduling with a synchronous flush to recover exact correctness. We also studied optimizations like interleaved pipeline parallelism, Zero Bubble parallelism, and DualPipe, which all aim to shrink the size of the pipeline bubble further.

We then studied tensor parallelism, which splits the tensors within a single layer across devices when even one layer is too large to fit on a GPU. We also examined its overhead, and understood why tensor parallelism is confined to a modest degree within a single GPU node. We also reviewed other forms of parallelism in recent research. Expert parallelism places different experts of an MoE model on different GPUs and routes tokens between them via AllToAll operations. Context parallelism distributes a single long sequence's tokens across GPUs, using block-wise attention and Ring Attention to compute attention incrementally without materializing the full attention matrix. Sequence parallelism instead partitions activations along the sequence dimension within a layer to reduce the activation memory that tensor parallelism alone leaves replicated. We also briefly

discussed how the different types of parallelism can be combined in various hybrid parallelism designs, using the more communication-intensive techniques like tensor parallelism within GPUs connected by high-bandwidth intra-node links, while employing low-overhead methods like pipeline or data parallelism across GPUs connected by the slower data center network. Finally, we looked at two case studies of DeepSeek-V3, and Llama 3, to understand how the ideas of this chapter apply to real-world production systems that train large language models.

Having built this understanding of distributed training, we now turn to the networking layer that makes all of this communication possible. The next chapter looks at networking optimizations for machine learning systems, examining the interconnects, topologies, and communication libraries that collective operations such as AllReduce, AllGather, and AllToAll rely on.

References

DeepSeek-AI (2025). *DeepSeek-V3 Technical Report*. URL: https://arxiv.org/abs/2412.19437.

Grattafiori, Aaron et al. (2024). *The Llama 3 Herd of Models*. URL: https://arxiv.org/abs/2407.21783.

Huang, Yanping, 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". In.

Liu, Hao, Matei Zaharia, and Pieter Abbeel (2023). *Ring Attention with Blockwise Transformers for Near-Infinite Context*. URL: https://arxiv.org/abs/2310.01889.

Narayanan, Deepak, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia (2019). "PipeDream: generalized pipeline parallelism for DNN training". In: URL: https://doi.org/10.1145/3341301.3359646.

Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay 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: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. URL: https://doi.org/10.1145/3458817.3476209.

Qi, Penghui, Xinyi Wan, Guangxing Huang, and Min Lin (2023). *Zero Bubble Pipeline Parallelism*. URL: https://arxiv.org/abs/2401.10241.

Qian, Kun, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai (2024). "Alibaba HPN: A Data Center Network for Large Language Model Training". In: URL: https://doi.org/10.1145/3651890.3672265.

Shi, Aaron and Zachary DeVito (2023). *Understanding GPU Memory 1: Visualizing All Allocations over Time*. URL: https://pytorch.org/blog/understanding-gpu-memory-1/.

Zhao, Yanli, 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". In: *Proc. VLDB Endow.* URL: https://doi.org/10.14778/3611540.3611569.