CSE-234: Data Systems for Machine Learning, Winter 2025

Parellization: Part 4

Lecturer: Hao Zhang

Scribes: Yuan Gao, Aaron Ang, Jinyi Wan, Zhecheng Li, Shuting Zhao, Zixuan Song, Lifan Sun, Mengyang Liu, Yuchen Feng, Jenish Thanki, Heng Zhu

1 Recap of Synchronous Pipeline Schedules

1.1 Pipeline Bubble Percentage Calculation

In synchronous pipeline schedules, we often have devices running, while some devices idling (i.e. not running). The idling devices in the pipeline schedules are called bubbles, which can be seen as a waste. Therefore, we need a metric to characterize the amount of bubbles and develop schedules minimizing it. The metric often used is called **pipeline bubble percentage**, which is calculated by the following formula:

$$\text{pipeline bubble percentage} = \frac{\text{bubble area}}{\text{total area}} \tag{1}$$

In Figure 1, the blue areas are stages that have a device running and the gray areas are bubbles. Suppose we have D devices and we schedule the pipeline in the way shown in Figure 1. The total area is D^2 and the bubble area is $D(D-1)$. Then, our **pipeline bubble percentage is** $(D-1)/D$ after simplification.

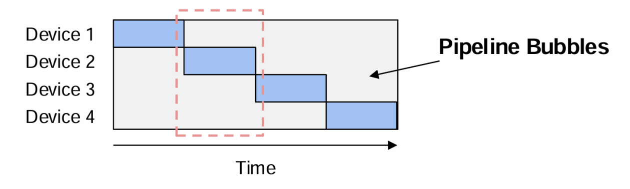


Figure 1: A figure demonstration for pipeline bubbles.

1.2 GPipe

GPipe [4] split each mini-batch into N micro-batches in the batch dimension. Then, each micro-batch is processed sequentially on the devices. After the forward computation of all micro-batches are finished, we run the backward passes for each micro-batch. When the backward passes of all micro-batches are finished, we update the weights of the model. Figure 2 gives an illustration of GPipe scheduling.

The **pipeline bubble percentage of GPipe is:** $(D-1)/(D-1+N)$, where N is the number of micro-batches and D is the number of devices. In Figure 2, we have $N = 6$ and $D = 4$. Despite large N gives less bubbles, it increases the peak memory usage. The **peak memory usage of GPipe is Parameters + Activation** $\times$ N.

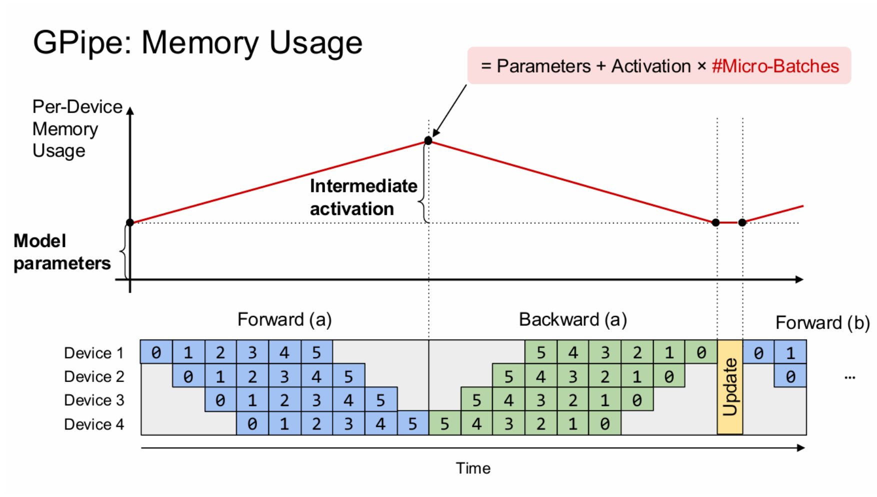


Figure 2: GPipe pipeline schedule and peak memory usage.

1.3 1F1B

1F1B (one forward one backward) [1] is a modified version of GPipe that reduces the peak memory usage. Whenever one micro-batch has completed forward pass, the backward pass of this micro-batch is started. Figure 3 gives an illustration of 1F1B scheduling.

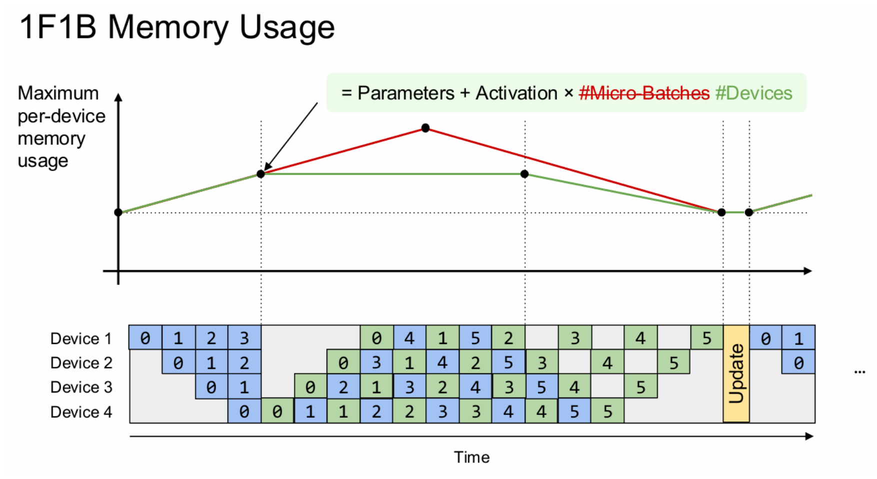

1F1B Memory Usage
= Parameters + Activation × ~~#Micro-Batches~~ #Devices
Maximum per-device memory usage
Device 1
Device 2
Device 3
Device 4
Update
Time

Figure 3: 1F1B pipeline schedule and peak memory usage.

The **pipeline bubble percentage of 1F1B is:** $(D-1)/(D-1+N)$, which is the same as GPipe. The **peak memory usage of 1F1B is Parameters + Activation** $\times D$. This is because we do not need to keep the activations of all micro-batches in the memory. The backward passes of the starting micro-batches are finished earlier and then moved out of the memory.

1.4 Chimera

Chimera [5] reduces the bubbles in 1F1B by storing bidirectional stages and use bidirectional pipelines. The idea of using bidirectional pipelines is adopted to train DeepSeek-V3 [6]. Figure 4 gives an illustration for the Chimera pipeline schedule.

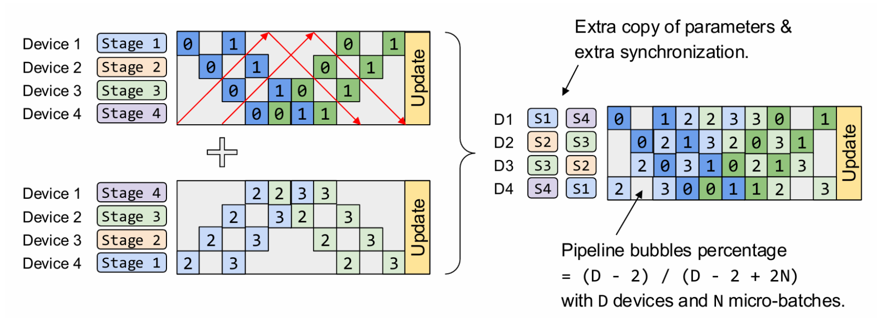

Device 1
Device 2
Device 3
Device 4
Stage 1
Stage 2
Stage 3
Stage 4
Update
Extra copy of parameters & extra synchronization.
D1
D2
D3
D4
S1
S2
S3
S4
Pipeline bubbles percentage
= (D - 2) / (D - 2 + 2N)
with D devices and N micro-batches.

Figure 4: Chimera pipeline schedule.

The **pipeline bubble percentage of Chimera is:** $(D-2)/(D-2+2N)$, which is less than GPipe and 1F1B. However, Chimera requires the devices to **store two copies of the original model, resulting in a higher peak memory usage than 1F1B**. This is because according to the schedule, the i-th device not only stores the i-th partition of the model counted from forward, but also the i-th partition counted backward. In addition, Chimera only works with even number of devices.

1.5 Synchronous Pipeline Schedule Summary

For all the schedules we introduced in last lecture, they are call synchronous pipeline schedule.

It is something very similar to what we discussed in data parallelism that we always make sure that all the workers are in the same pace.

- **Pros:** Keep the convergence semantics. The training process is exactly the same as training the neural network on a single device.
- **Cons:** Pipeline bubbles. Reducing pipeline bubbles typically requires splitting inputs into smaller components, but too small input to the neural network will reduce the hardware efficiency.

2 Asynchronous Pipeline Schedules

Remove the synchronization point in the pipeline schedule, and now we can start the next round of forward pass before backward pass of the previous round finishes.

- **Pros:** No Pipeline bubbles.
- **Cons:** Break the synchronous training semantics. Now the training will involve stalled gradient. Algorithms may store multiple versions of model weights for consistency.

2.1 AMPNet

If we let all the devices to go free and excute at their own pace, we basically get AMPNet [2] schedule.

Idea: Fully asynchronous. Each device performs forward pass whenever free and updates the weights after every backward pass.

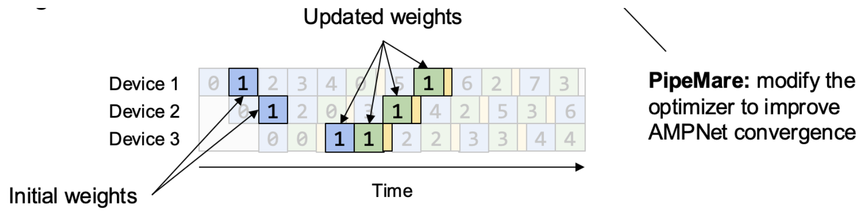

Updated weights
Device 1
Device 2
Device 3
Initial weights
Time
PipeMare: modify the optimizer to improve AMPNet convergence

Figure 5: AMPNet

With this schedule, the different stages in the forward and backward pass can be on different versions of the weights because each device update on their own pace.

For example, in Figure 5, for data 1, the forward pass in stages 1 and 2 still uses the initial weights (blue 1), but during the backward pass, its gradient updates are based on the weights modified after training with data 0 (green 1). Since the forward and backward passes use inconsistent weights, this **weight staleness** introduces noise, affecting training convergence. PipeMare alleviates this issue by optimizing the optimizer strategy.

Convergence: Achieve similar accuracy on small datasets (MNIST 97%), hard to generalize to larger datasets.

PipeMare improve the convergence of AMPNet: It modifies the gradient optimizer to improve the convergence a little bit. But it cannot foundamentally changed the drawback of this way.

2.2 Pipedream

Achive better convergence by reducing the asynchrony. The timeline of the Pipedream [3] looks very similar to 1F1B, but the main difference is we update the model's weights once the backward has finished.

Idea: Enforce the same version of weight for a single input batch by storing multiple weight versions.

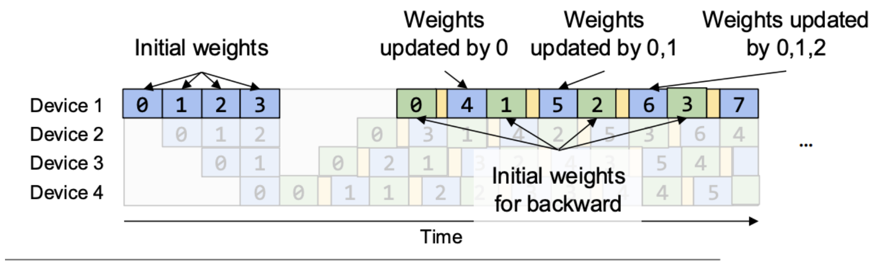


Figure 6: Pipedream

In Figure 6, let us focus on Device 1. During the forward and backward passes for input batches 0 to 3, the same initial weights are used. However, for input batch 4, the forward pass utilizes weights that have been updated by input batch 0.

Since the backward pass for input batches 1 to 3 still requires the initial weights, at this point, two copies of weights exist:

1. Initial weights (for the backward pass of input batches 1 to 3).

2. Weights updated after processing input batch 0 (for the forward pass of input batch 4).

Similarly, for input batch 5, weights updated by input batches 0 and 1 are needed. For input batch 6, weights updated by 0, 1, and 2 are required.

As a result, we need to store four copies of weights in total. However, since the neural network is divided into only four stages, no memory is saved, defeating the purpose of pipeline parallelism in terms of memory efficiency.

Convergence: Similar accuracy on ImageNet with a 5x speedup compared to data parallel.

Con: No memory saving compared to single device case.

2.3 Pipedream-2BW

To reduce the memory usage, the authors of Pipedream proposed a modification to the original pipeline by updating the weights less frequently, which is called Pipedream-2BW [8].

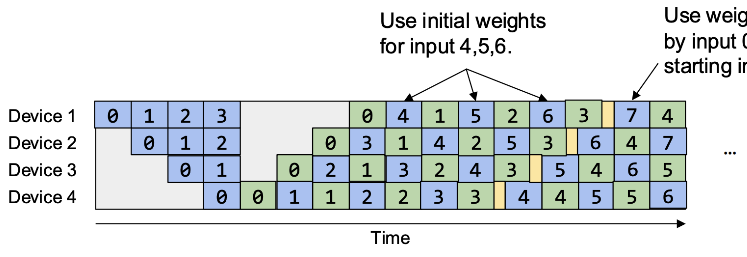

Use initial weights for input 4,5,6.
Use weights updated by input 0,1,2,3 starting input 7.
Device 1
Device 2
Device 3
Device 4
Time
...

Figure 7: Pipedream-2BW

In Figure 7, for input batches 4, 5, and 6, the initial weights are still used. Starting from input batch 7, the model begins using weights updated by input batches 0 to 3.

This means that the green weights (updated weights) are only stored for one update cycle before being replaced, significantly reducing the memory storage requirement compared to standard Pipedream.

Idea: Reduce Pipedream's memory usage (only store 2 copies) by updating weights less frequently. Weights always stalled by 1 update.

Convergence: Similar training accuracy on language models (BERT/GPT)

3 Imbalanced vs. Balanced Pipeline Stages

For all the pipeline scheduling algorithms shown before, we assume the running times of different pipeline stages are exactly the same. But there are some cases that the latency of these stages are not the same, which is the unbalanced case. For different inputs, the execution time may be different, which is a worse case. So an important factor when doing pipeline parallelism is to make sure each device has the same workload.

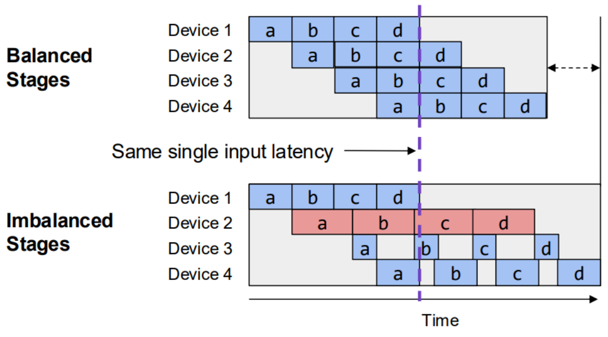


Figure 8: Comparison between balanced and imbalanced pipeline stages

According to Figure 8, balanced stages lead to optimal pipeline throughput, while imbalanced stages cause increased bubbles and latency.

4 Frontier: Automatic Stage Partitioning

4.1 Goal

Minimize maximum stage latency: Reduce the runtime of the slowest (bottleneck) stage.

Maximize parallelization: Keep as many devices busy as possible, leveraging concurrency to increase throughput.

4.2 Two Main Approaches

1. **Reinforcement Learning (RL)-Based** (primarily for device placement): Use an RL agent to explore and learn optimal or near-optimal ways of assigning model layers/operators to devices.

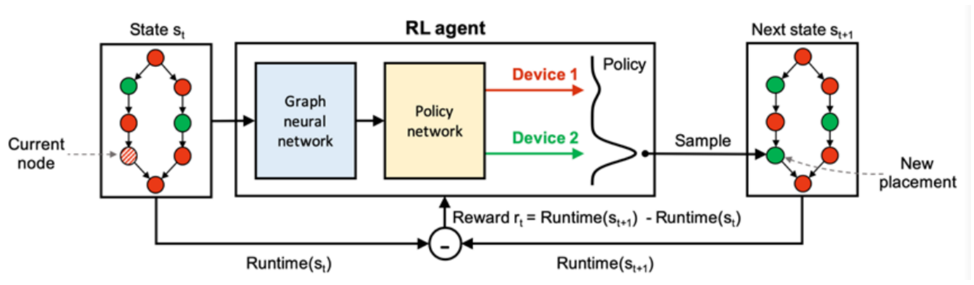


Figure 9: RL-Based Partitioning Algorithm

- **State**: Device assignment plan for a computational graph.

- **Action**: Modify the device assignment of a node.
- **Reward**: Latency difference between the new and old placements.

The RL-based methods can be trained with **policy gradient** algorithm.

2. **Optimization-Based** (Dynamic Programming / Linear Programming): Formulate stage/device placement as a mathematical optimization problem with constraints (e.g., memory limits, communication overhead, etc.). Solve via dynamic or linear programming techniques.

5 Inter-Operator Parallelism Summary

Idea: Assign different operators of the computational graph to different devices and execute them in a pipelined fashion.

Method	General Computational Graph	No Pipeline Bubbles	Same Convergence as Single Device
Device Placement	×	×	×
Synchronous Schedule	✓	×	✓
Asynchronous Schedule	✓	✓	×

Table 1: Comparison of inter-operator parallelism methods.

Note:

- *Stage Partitioning:* If stages are imbalanced, more pipeline bubbles occur.
- *RL-Based / Optimization-Based Automatic Stage Partitioning:* Approaches that systematically assign operators (or stages) to devices to reduce imbalances and improve overall efficiency.

6 Intra-operator Parallelism

6.1 Intra-op and Inter-op

This part is a basic recap for two parallelization strategies: Inter-operator Parallelism and Intra-operator Parallelism.

1. **Inter-operator Parallelism:** Different operators in a computation graph (e.g., matrix multiplication, ReLU, subtraction) are executed in parallel. This strategy focuses on distributing entire operators across different devices or processes.

2.**Intra-operator Parallelism:** A single operator is split into smaller tasks that run in parallel. This approach is beneficial when a single operator is computationally intensive.

6.2 Parallelize One Operator

6.2.1 Element-wise Operators

Figure 10 explains how to parallelize element-wise operations of adding two matrices A and B. Since there are no dependencies between iterations of the loops, the computation can be split arbitrarily across devices.

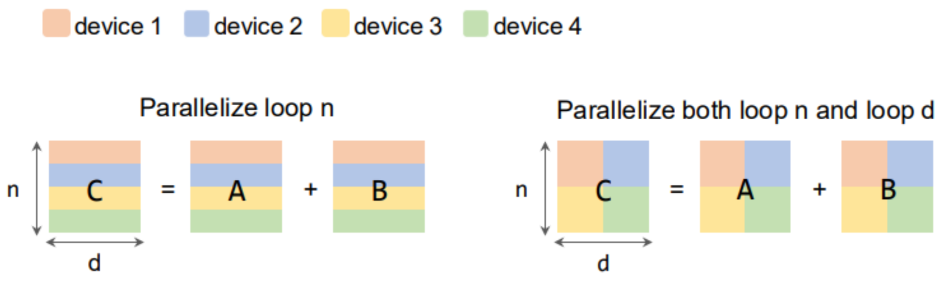


Figure 10: Element-wise operators

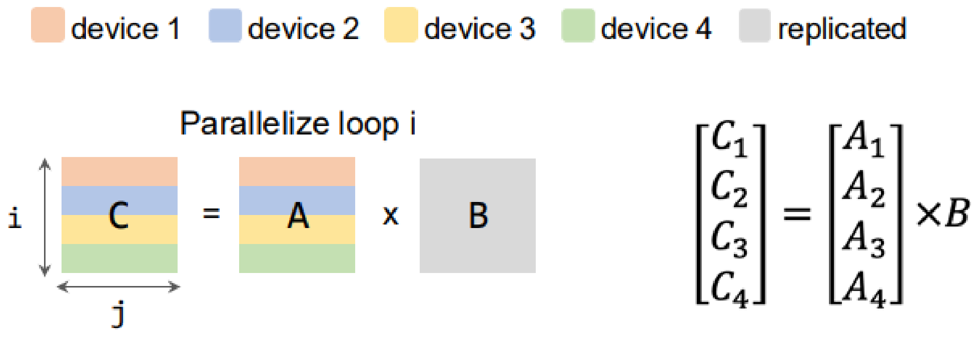
There are multiply ways to parallelize this operator. One example is to divide the rows among multiple devices. For example, device 1 processes the first set of rows, device 2 processes the next set of rows, and so on. Another one is to parallelize both rows and columns, where each device handles a block of rows and columns. However, there are no communication needed for this operator.

6.2.2 Matrix Multiplication

Matrix multiplication involves more complex dependencies compared to element-wise operations because it includes a reduction loop.

The outer loops (i, j) are independent and can be split across devices, but the reduction loop (k) introduces dependencies because partial results must be accumulated to compute $C[i, j]$.

Therefore, the first parallelization strategy would be to split the rows of C and corresponding rows of A across devices, and replicate the full matrix B to each devices, so that each one could compute a subset of rows independently.



$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \times B$$

Figure 11: Matrix Multiplication-Strategy 1

Another strategy would be to split matrix A among its columns, and split matrix B along its rows. Each device computes partial results for its assigned slice of A and B. After local computations, and all-reduce operation aggregates these partial results to produce the final matrix C.

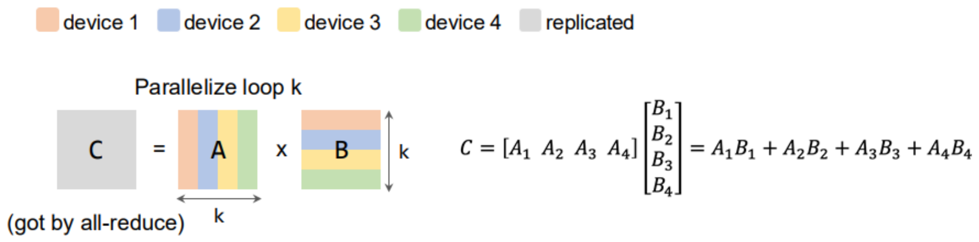

device 1
device 2
device 3
device 4
replicated
Parallelize loop k
C
=
A
x
B
k
k
(got by all-reduce)
$C = [A_1\ A_2\ A_3\ A_4]\begin{bmatrix}B_1\\B_2\\B_3\\B_4\end{bmatrix} = A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4$

Figure 12: Matrix Multiplication-Strategy 2

The third strategy is to parallelize two loops together. The output matrix C is both column-partitioned and row-partitioned and is divided into 4 regions. Each device would compute a portion of C independently. The input matrices A and B are partially tiled to ensure all devices have the necessary data for their assigned computations.

Similarly, we can parallelize both loop i and k, which results in a partially tiled matrix C. In this method we an all-reduce is needed to get matrix C.

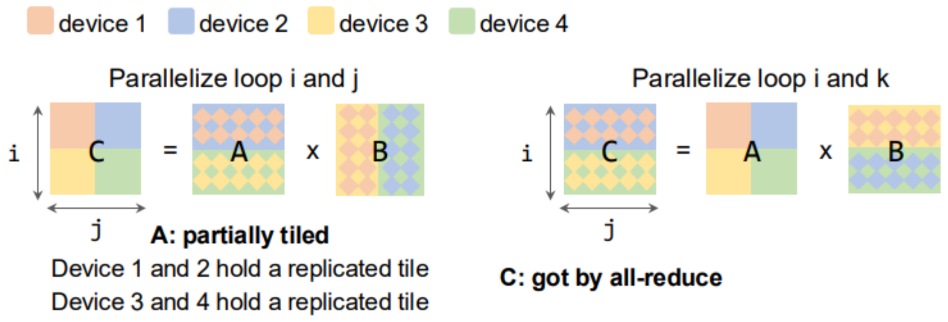


Figure 13: Matrix Multiplication-Strategy 3

6.2.3 2D Convolution

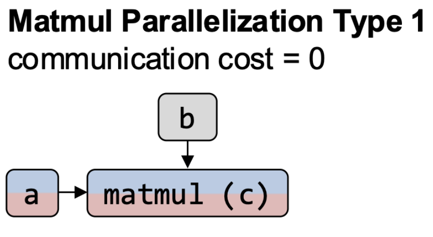
In terms of 2D convolution operations, one simple way is to parallelize the batch dimension, output channels as well as input channels, and then the parallelization strategies are almost the same as matmul's. However, there are also many more complicated cases we can enumerate.

6.3 Two Intra-Op Parallelism strategies for matmul

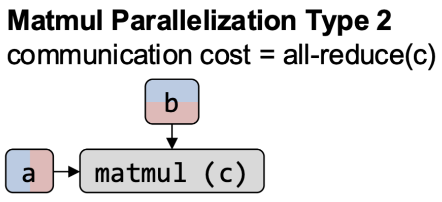
Data parallelism is a special case of intra-op parallelism. We use legends to represent whether a tensor is replicated, row-partitioned, or column-partitioned.

Two types of intra-op parallelism strategies for matrix multiplication (matmul) are considered here:

Type 1: Partitioning Matmul Along Rows. As shown in Figure 14(a). In this case, no communication is needed. Input **A** is partitioned, and input **B** is replicated. The result matrix is composed of sub-regions.



(a) Type 1


Matmul Parallelization Type 2
communication cost = all-reduce(c)
b
a
matmul (c)

(b) Type 2

Figure 14: Two types of partition.

Type 2: Partitioning the Reduction Loop (K-Dimension). In this type, both inputs are partitioned along the reduction dimension. An **all-reduce** operation is required to complete the computation.

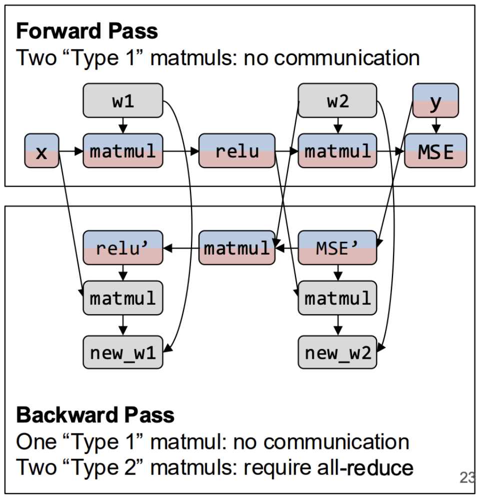

Forward Pass
Two "Type 1" matmuls: no communication
w1
w2
y
x
matmul
relu
matmul
MSE
relu'
matmul
MSE'
matmul
matmul
new_w1
new_w2
Backward Pass
One "Type 1" matmul: no communication
Two "Type 2" matmuls: require all-reduce
23

Figure 15: Forward and Backward pass computation graph for a 2-layer MLP

6.4 Computation Graph for a Two-Layer MLP

A Forward and Backward pass computation graph for a 2-layer MLP is shown in Figure 15.

6.4.1 Forward Pass

- Weight tensors **W1** and **W2** are replicated (data parallelism).
- Matmul and ReLU operations are partitioned along the batch dimension, requiring no communication.

6.4.2 Backward Pass

- Contains one Type-1 matmul and two Type-2 matmuls.

- Some all-reduce operations on the gradients are required.
- Data parallelism necessitates communication due to specific partitioning choices.

6.5 Intra-Operator Parallelism and Repartitioning communication Cost

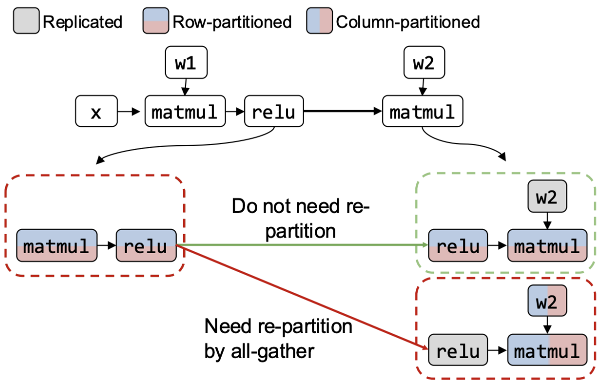


Figure 16: Repartitioning cost in intra-op parallelism.

As shown in Figure 16. When applying intra-operator parallelism to an entire computational graph, there is a repartitioning communication cost on the edges. This arises because different operator parallelism strategies may require different partitioning formats for the same tensor, necessitating repartitioning.

To illustrate this, let us consider a two-layer MLP, as shown in Figure 16. Specifically, we examine the edge between the ReLU activation and the second matrix multiplication (matmul), which is a longer edge in the computational graph. The partitioning format of the ReLU follows that of the preceding matmul, as it is inherited. Assuming that the ReLU operation is row-partitioned and the previous computations were also row-partitioned, the sharding persists through ReLU. Since ReLU is an element-wise operation, no additional partitioning is required.

However, for the second matmul, its partitioning strategy—whether type I, type II, or type III—dictates the required input partitioning format. If the second matmul also adopts a type I partitioning strategy, where the input is partitioned along the rows, then the ReLU operation must also remain row-partitioned. Recall that in type I partitioning, the first input must be row-partitioned while the second input is replicated.

Now, if we enforce the second matmul to follow the type I partitioning strategy, we find that the ReLU operation naturally aligns with its predecessor, both being row-partitioned. Consequently, computation proceeds seamlessly since each device retains the necessary ReLU results without additional communication overhead.

Conversely, if the second matmul employs a different parallel strategy—one requiring the ReLU output to be replicated while the weight matrix W_2 is column-partitioned—then the ReLU results must be duplicated across all devices. This necessitates an additional communication operation, specifically an `all-gather`, to synchronize the replicated ReLU results across devices. As a result, choosing distinct partitioning strategies for different matmul operations introduces repartitioning costs, manifesting as additional collective communication primitives such as `all-gather`.

To optimize the execution cost of the entire computational graph, it is crucial to strategically select partitioning strategies for each operator to minimize these repartitioning overheads when designing the partitioning scheme for the entire network.

When applying intra-operator parallelism to a whole graph, there is a repartitioning communication cost on the edges. Different operator parallelism strategies may require different tensor partitioning formats, necessitating repartitioning.

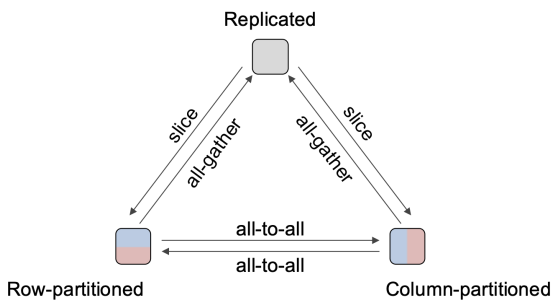


Figure 17: Re-partition Communication Cost

6.6 Optimization Problem: Minimizing Execution Cost

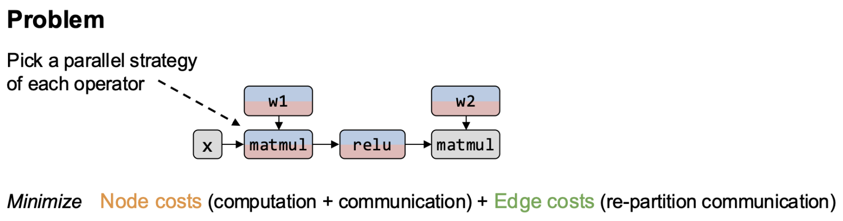

Problem
Pick a parallel strategy of each operator
w1
w2
x
matmul
relu
matmul
Minimize Node costs (computation + communication) + Edge costs (re-partition communication)

Figure 18: Optimization problem formulation for parallelization.

As shown in Figure 18, to optimize parallel execution of a graph, we must:

- Pick one parallel strategy per operator to minimize execution cost.
- Formulate the problem as an optimization: 1. Select a coloring scheme for each node. 2. Enumerate all possible communication costs. 3. Minimize total cost to achieve efficient execution.

Solutions for Efficient Parallelization

Possible approaches to solving this optimization problem include: Manual design, Randomized search, Dynamic programming, Integer linear programming.

7 Model-specific Intra-op Parallel Strategies

7.1 Introduction

Intra-op parallelism is a critical technique in scaling modern neural network architectures, enabling efficient computation across increasingly large models and diverse hardware accelerators. Unlike inter-op parallelism, which distributes separate operations across devices, intra-op parallelism focuses on parallelizing the execution within a single operation, such as matrix multiplications or expert computations in a layer. This approach is particularly vital for models like AlexNet, GShard, and Megatron-LM. After the popularization of these strategies, newer models have adapted similar practices.

7.2 AlexNet

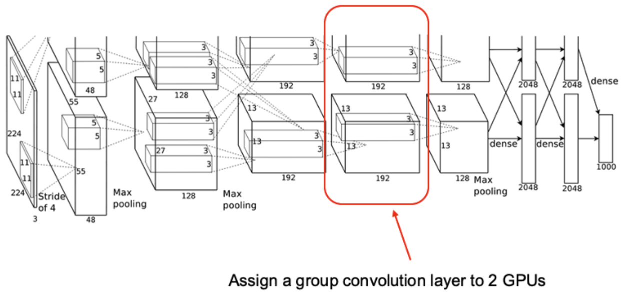


Figure 19: The image illustration shows AlexNet intra-op CNN.

In AlexNet, intra-op parallelization was implicitly employed through the use of two NVIDIA GTX 580 GPUs, which were tasked with handling the intensive computations of its eight-layer architecture, five convolutional layers followed by three fully connected layers. For instance, the convolution operations, which dominate the computational cost in CNNs, were parallelized at the kernel level. Each GPU's CUDA cores could simultaneously process different portions of the input feature maps or filters, effectively speeding up the convolution process. This intra-op parallelism was facilitated by the CuDNN library and CUDA framework, which optimized low-level operations to exploit the GPU's massively parallel architecture.

Additionally, AlexNet's design split the model across two GPUs, with certain layers divided such that half the neurons were processed on one GPU and half on the other. Within this model parallelism, intra-op parallelization further enhanced efficiency by allowing each GPU to independently parallelize its assigned operations. During the forward and backward passes, operations like ReLU activation, max-pooling, and local response normalization were executed in parallel across the GPU cores for the subset of data each GPU handled. By combining intra-op parallelization with inter-GPU communication for synchronization, AlexNet set a precedent for future DL models.

7.3 Megatron-LM

Megatron-LM is one of the first DL model that utilized intra-layer model tensor parallelism. By partitioning the workload of large operations like General Matrix-Matrix Multiplications within the self-attention and MLP blocks of a transformer, Megatron-LM achieves efficient scaling. As seen in Figure 20, Megatron-LM implements intra-op parallelization by splitting key operations along specific dimensions to optimize both computation and communication. For the MLP block, the first GEMM is partitioned in a column-parallel fashion, dividing the weight matrix and allowing each GPU to compute its portion independently, followed by a GeLU nonlinearity applied locally without immediate synchronization. The subsequent GEMM is then split row-wise, taking the output directly and requiring only a single all-reduce operation across GPUs to combine results, thus reducing synchronization points. Similarly, in the self-attention block, the key, query, and value (K, Q, V) GEMMs are split column-wise across GPUs.

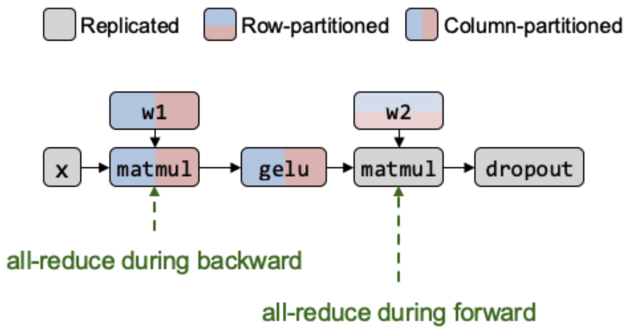


Figure 20: The image illustration shows intra-op operations in Megatron-LM.

7.4 GShard MoE

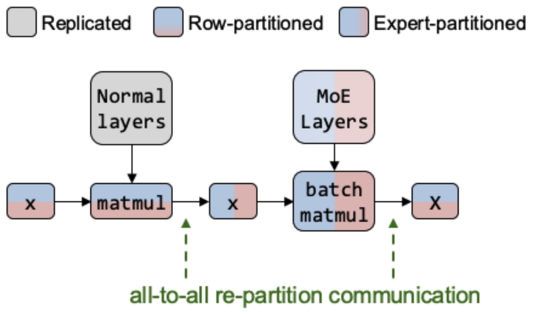


Figure 21: The image illustration shows intra-op operations in Megatron-LM.

GShard's intra-op parallel strategy operates by replacing every other feedforward neural layer in a Transformer model with an MoE layer, typically employing a top-2 gating mechanism. This gating function routes each input token to the top two most relevant experts based on a learned probability distribution, while the XLA compiler automatically shards the computation across devices. The intra-op parallelism is enhanced by splitting the tokens and dispatching them to their assigned experts in parallel, followed by an all-to-all instruction to recombine results. GShard also introduces innovations like random routing for the second expert and expert capacity limits to prevent overload, ensuring balanced and efficient workloads. This approach enabled GShard to scale a Transformer model beyond 600 billion parameters, achieving superior translation quality across 100 languages.

8 ZeRO Optimizer

8.1 Introduction

The **ZeRO (Zero Redundancy Optimizer)** is an optimization technique designed to improve memory efficiency in large-scale distributed training. It addresses the inefficiencies of traditional data parallelism, which involves replicating optimizer states, gradients, and model weights across all devices, leading to excessive memory consumption.

8.2 Problem with Data Parallelism

In traditional data parallelism, all devices maintain a copy of: Optimizer states (which take significant memory, especially in large-scale models), Gradients, Model weights. This redundancy results in a large memory footprint and limits the scalability of deep learning models.

8.3 ZeRO Optimizer's Idea

The ZeRO optimizer aims to reduce memory usage by partitioning (instead of replicating) optimizer states, gradients, and model weights across devices. This allows for more efficient utilization of memory and enables training larger models.

8.4 Breakdown of ZeRO Stages

ZeRO optimization is implemented in multiple stages, with each stage progressively reducing redundancy:

Approach	Optimzer States (12M)	Gradients (2M)	Model Weights (2M)	Memory Cost	Communication Cost
Data Parallelism	Replicated	Replicated	Replicated	$16M$	all-reduce(2M)
ZeRO Stage 1	Partitioned	Replicated	Replicated	$4M + \frac{12M}{N}$	all-reduce(2M)
ZeRO Stage 2	Partitioned	Partitioned	Replicated	$2M + \frac{14M}{N}$	all-reduce(2M)
ZeRO Stage 3	Partitioned	Partitioned	Partitioned	$\frac{16M}{N}$	1.5 all-reduce(2M)

Table 2: ZeRO Optimization Stages and Memory Efficiency

8.4.1 Key Takeaways from Each Stage

- **Data Parallelism**: High memory cost ($16M$) due to full replication.
- **ZeRO Stage 1**: Partitions optimizer states but keeps gradients and model weights replicated, reducing memory usage to $4M + \frac{12M}{N}$.
- **ZeRO Stage 2**: Further partitions gradients while keeping only model weights replicated, leading to $2M + \frac{14M}{N}$ memory cost.
- **ZeRO Stage 3**: Fully partitions optimizer states, gradients, and model weights across devices, achieving the lowest memory cost ($\frac{16M}{N}$) and reducing communication cost.

8.4.2 Benefits of ZeRO

- **Lower Memory Usage**: By partitioning optimizer states, gradients, and model weights across devices, ZeRO reduces the overall memory footprint.

- **Larger Model Training**: Enables the training of much larger models that wouldn't fit in memory with traditional data parallelism.
- **Efficient Communication**: While ZeRO introduces additional communication overhead, it is optimized to keep it manageable (only increasing marginally at Stage 3).

8.5 ZeRO Stage 2 Overview

ZeRO Stage 2 builds upon ZeRO Stage 1 by **partitioning both optimizer states and gradients**, reducing memory consumption even further.

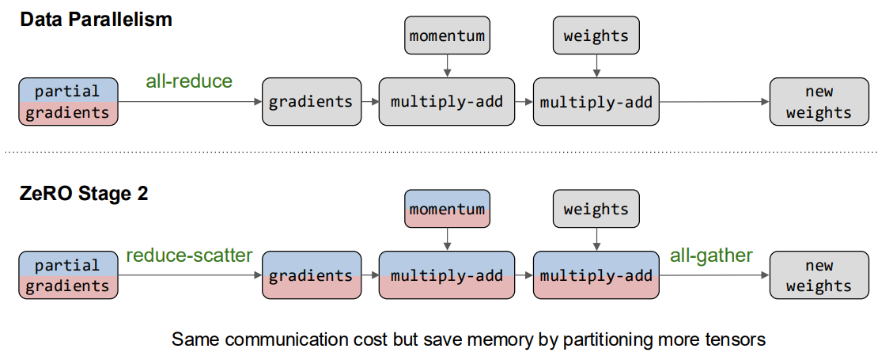


Figure 22: The image illustration about ZeRO Stage 2 of the ZeRO framework.

8.5.1 Key Idea

The primary optimization in ZeRO Stage 2 is replacing the traditional all-reduce operation with a combination of:

- **Reduce-scatter**: Distributes different portions of the gradients to different devices, avoiding full gradient replication.
- **All-gather**: Collects updated values from different devices after computation.

This modification retains the same communication cost as all-reduce but significantly reduces memory consumption by partitioning more tensors.

8.5.2 Comparison: Data Parallelism vs. ZeRO Stage 2

The following highlights the differences in computation and memory efficiency:

1. Data Parallelism Workflow In standard data parallelism, the training process follows these steps:

1. **Partial Gradients Computation** (on each device).
2. **All-reduce Operation**: Each device receives the full set of gradients.

3. **Momentum Update**: The gradients are adjusted based on momentum.
4. **Weight Update**: Weights are updated using gradients via multiply-add operations.
5. **New Weights Generation**: The updated weights are used for the next iteration.

Issue: All gradients are fully replicated on every device, leading to high memory consumption.

2. ZeRO Stage 2 Workflow ZeRO Stage 2 modifies the workflow as follows:

1. **Partial Gradients Computation** (on each device).
2. **Reduce-Scatter Operation**: Gradients are partitioned across devices instead of being fully replicated.
3. **Momentum Update**: Only the necessary portion of the gradients is updated per device.
4. **Weight Update**: Each device computes weight updates on its partitioned gradients.
5. **All-Gather Operation**: Updated gradients are assembled to produce new weights.

Advantage: Memory savings due to partitioning gradients instead of replicating them. The communication overhead remains similar to all-reduce.

8.5.3 Advantages of ZeRO Stage

- **Memory Efficiency**: ZeRO Stage 2 significantly reduces memory usage by partitioning gradients alongside optimizer states.
- **Scalability**: Enables training larger models across multiple GPUs or nodes.
- **Same Communication Cost**: Despite gradient partitioning, the total communication cost remains unchanged compared to standard data parallelism.

8.5.4 Conclusion

ZeRO Stage 2 extends the optimization of ZeRO Stage 1 by partitioning gradients, leading to further reductions in memory consumption while maintaining efficiency. This makes it an essential technique for scaling up deep learning models, especially when working with limited GPU memory resources.

8.6 ZeRO Stage 3 Overview

ZeRO Stage 3 builds upon ZeRO Stage 2 by **fully partitioning optimizer states, gradients, and model weights**, allowing deep learning models to scale further with minimal memory overhead.

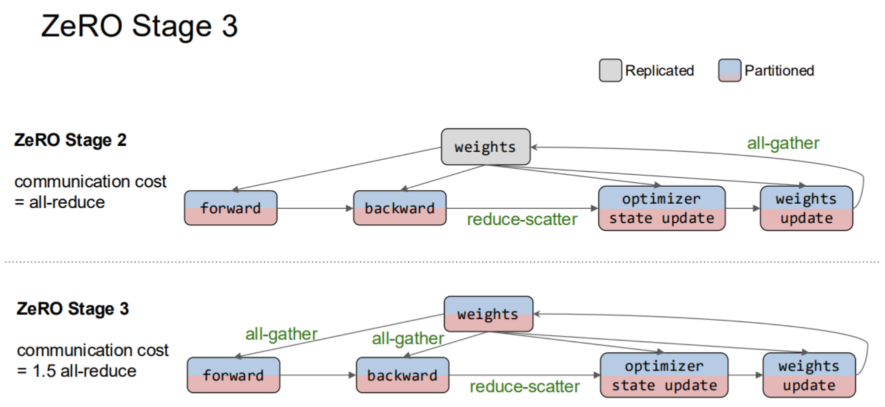


Figure 23: The image illustration of the comparison between ZeRO Stage 2 and ZeRO Stage 3.

8.6.1 Key Idea

ZeRO Stage 3 **extends memory optimizations** from ZeRO Stage 2 by partitioning model weights across devices in addition to gradients and optimizer states. This results in maximum memory savings while maintaining computational efficiency.

The key difference in ZeRO Stage 3 is:

- **ZeRO Stage 2**: Uses all-reduce for communication.
- **ZeRO Stage 3**: Uses reduce-scatter and all-gather, leading to **1.5x all-reduce communication cost** instead of 1x.

This slight increase in communication overhead is outweighed by the massive memory savings, enabling the training of much larger models on limited GPU resources.

8.6.2 Comparison: ZeRO Stage 2 vs. ZeRO Stage 3

The following outlines the differences in workflow:

1. ZeRO Stage 2 Workflow

1. **Forward Pass**: Model processes input.
2. **Backward Pass**: Computes gradients.
3. **Reduce-Scatter**: Partitions and distributes gradient updates across devices.
4. **Optimizer State Update**: Updates optimizer states (partitioned).
5. **Weights Update**: Updates model weights (still replicated).
6. **All-Gather**: Collects partitioned weights.

Issue: Model weights are still **fully replicated**, limiting memory savings.

2. ZeRO Stage 3 Workflow

1. **All-Gather Before Forward Pass**: Since model weights are partitioned, an all-gather operation reconstructs weights before computation.
2. **Forward Pass**: Model processes input.
3. **All-Gather Before Backward Pass**: Partitioned weights are gathered again for gradient computation.
4. **Backward Pass**: Computes gradients.
5. **Reduce-Scatter**: Partitions gradient updates across devices.
6. **Optimizer State Update**: Optimizer states are updated (partitioned).
7. **Weights Update**: Partitioned weights are updated.

Advantage: Model weights are now **partitioned**, **dramatically reducing memory usage**.

Downside: Additional all-gather operations introduce **1.5x all-reduce communication cost**.

8.6.3 Advantages of ZeRO Stage 3

- **Maximal Memory Efficiency**: Fully partitions optimizer states, gradients, and model weights across devices.
- **Enables Training of Billion-Scale Models**: Makes it possible to train enormous models that wouldn't fit into memory with traditional approaches.
- **Trade-off Between Memory and Communication**: While communication overhead increases (1.5x all-reduce), the memory footprint reduction allows training much larger models on fewer GPUs.
- **Better GPU Utilization**: Since memory usage is minimized, GPUs can handle **larger batch sizes**, improving throughput.

8.6.4 Conclusion

ZeRO Stage 3 is the **most memory-efficient stage** of the ZeRO framework. By fully **partitioning optimizer states, gradients, and model weights**, it allows training models several times larger than what is possible with standard data parallelism. While it introduces a slightly higher communication cost (1.5x all-reduce), the significant reduction in memory consumption far outweighs this cost, making ZeRO Stage 3 ideal for large-scale deep learning workloads.

9 Mesh-TensorFlow: Mapping Tensor Dimensions to Mesh Dimensions

9.1 Tensor Dimensions

Mesh-TensorFlow defines tensor dimensions using `mtf.Dimension(name, size)`. Each tensor dimension represents a logical data axis in the computation. For example:

```
batch = mtf.Dimension("batch", b)
io = mtf.Dimension("io", d_io)
hidden = mtf.Dimension("hidden", d_h)
```

9.2 Mesh Dimensions

Mesh dimensions represent the distributed computational grid. They are defined using a list of tuples specifying the mesh dimension names and their sizes:

```
mesh_shape = [("rows", r), ("cols", c)]
```

9.3 Mapping Tensor Dimensions to Mesh Dimensions

Tensor dimensions are mapped to mesh dimensions using a computation layout. This mapping determines how tensors are distributed across the computation grid for efficient parallelism:

```
computation_layout = [("batch", "rows"), ("hidden", "cols")]
```

This means:

- The `"batch"` dimension is distributed along `"rows"`.
- The `"hidden"` dimension is distributed along `"cols"`.

9.4 Summary

- Mesh-TensorFlow enables **explicit model parallelism** by partitioning tensors across multiple devices.
- The **tensor dimensions** define the logical shape of data.
- The **mesh dimensions** represent the hardware topology.
- The **mapping (computation layout)** determines how tensor dimensions are split across available compute resources.

9.5 GSPMD

GSPMD is the successor of Mesh-TensorFlow. It uses a similar compiler-based approach. Users insert annotations to specify the parallelization strategy for important tensors. Later, the compiler propagates the strategy to the entire graph and generates SPMD (Single Program Multiple Data) parallel executables.

10 Combining Intra-op and Inter-op Parallelism

As shown in Fig. 24, we can divide the above computational graph into multiple stages and assign each stage to a different device mesh with different numbers of GPUs. This helps to keep the computational

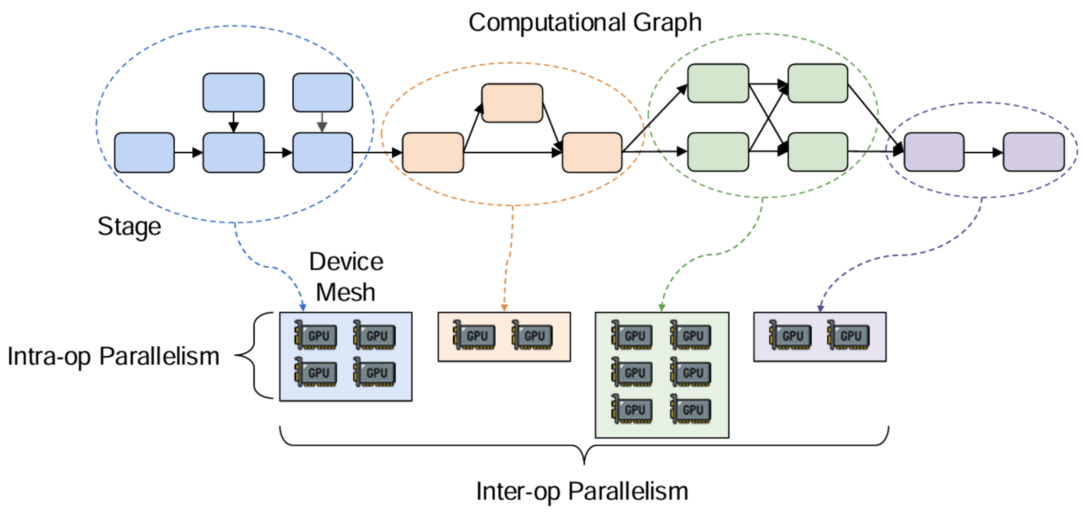


Figure 24: Combining Intra-op and Inter-op Parallelism

time approximately the same for each stage and benefits pipeline parallelism by reducing bubble size. Inside each stage, intra-op parallelism is performed. Combining both parallelisms helps achieve the best scalability, especially with large numbers (> 1000) of GPUs.

We can also combine parallelism with other optimizations mentioned in previous lectures, including gradient checkpointing and swapping for system-level memory optimization, as well as quantization, sparsification, and low-rank approximation for ML-level optimization.

11 Intra-operator Parallelism Summary

We can summarize the intra-operator parallelism part.

- We can parallelize a single operator by exploiting its internal parallelism.
- To use intra-operator parallelism for a whole computational graph, we need to choose strategies for all nodes in the graph to minimize the communication cost.
- Intra-op and inter-op can be combined to achieve the best performance.

11.1 Other techniques for training large models

In this part we focus on the parallelization. But in reality, we also want to combine the parallelization techniques with other techniques shown in previous lectures. We can do parallelization while also using other techniques, which includes:

- **System-level Memory Optimizations:** Rematerialization / Gradient Checkpointing, Swapping
- **ML-level Optimizations:** Quantization, Sparsification, Low-rank approximation

12 Auto-parallelization: Motivation and Problem Definition

12.1 Motivation

Machine learning developers face challenges in selecting the appropriate parallelization strategy for their models and computing clusters. Various parallelization techniques exist, including: **Data Parallelism**, Operator Partitioning, Pipeline Parallelism, ZeRO (Zero Redundancy Optimizer).

Different frameworks support these methods, such as:

- `DeepSpeed`, `FairScale FSDP`, `Megatron-LM`, `Mesh-TF`, `GSPMD`, `GPipe`, `1F1B`

As models evolve from CNNs to large-scale architectures like BERT, GPT-3, and Mixture of Experts (MoE), the selection of an appropriate parallelization strategy becomes more complex. The key challenge for ML developers is: *"Which parallelization strategy best fits my model and cluster?"*

12.2 Problem Definition

The core problem of auto-parallelization can be formulated as an optimization problem:

$$\max_{\text{strategy}} \text{Performance}(\text{Model}, \text{Cluster}) \tag{2}$$

subject to:

$$\text{strategy} \in \text{Inter-op} \cup \text{Intra-op} \tag{3}$$

Here, the goal is to maximize the training performance by selecting an optimal parallelization strategy from inter-operator (Inter-op) and intra-operator (Intra-op) parallelization methods.

12.3 Problem Breakdown

12.3.1 Model Representation

A deep learning model can be represented as a computational graph consisting of various operations, such as: `matmul` (matrix multiplication), `ReLU` (activation function), `MSE` (loss computation), `sub` (weights update).

These operations must be mapped efficiently to hardware to optimize performance.

12.3.2 Cluster and GPU Interconnect

The cluster consists of multiple GPU nodes, where execution efficiency is heavily influenced by the **interconnect topology and link characteristics between GPUs**. Key aspects include:

- **NVLink:** High bandwidth, low latency inter-GPU communication.
- **PCIe:** Lower bandwidth, higher latency, typically used for CPU-GPU communication.
- **InfiniBand:** Scales across multiple nodes but introduces communication overhead.

The selection of an optimal parallelization strategy must take into account:

- The communication overhead introduced by inter-GPU links.
- The placement of operations across GPUs to minimize data transfer bottlenecks.

12.3.3 Strategy Space

The number of possible parallelization strategies is large. Different ways to distribute model computations across available GPUs and nodes must be considered.

12.4 Search Space Complexity

The search space for auto-parallelization is vast, making it computationally challenging. The major contributing factors include:

- **# of operations in a real model (nodes to color):** $100 - 10K$
- **# of operation types (types of nodes):** $80 - 200+$
- **# of devices in a cluster (available colors):** $10s - 1000s$

13 Automatic Parallelization Methods

13.1 Overview of Methods

Automatic parallelization methods can be broadly classified into three categories:

- **Search-based methods:** These methods explore the parallelization space using techniques such as Markov Chain Monte Carlo (MCMC) and heuristics.
- **Learning-based methods:** These methods leverage machine learning techniques, including reinforcement learning, ML-based cost models, and Bayesian optimization, to guide the parallelization strategy.
- **Optimization-based methods:** These approaches formulate parallelization as an optimization problem and apply techniques such as dynamic programming, integer linear programming, and hierarchical optimization to find efficient solutions.

13.2 General Recipe for Automatic Parallelization

The process of automatic parallelization follows a structured workflow, as in Figure 25:

- **Search Space:** The initial set of all possible parallelization strategies.
- **Space Reduction:** Techniques are applied to prune infeasible or suboptimal strategies, reducing the search space.

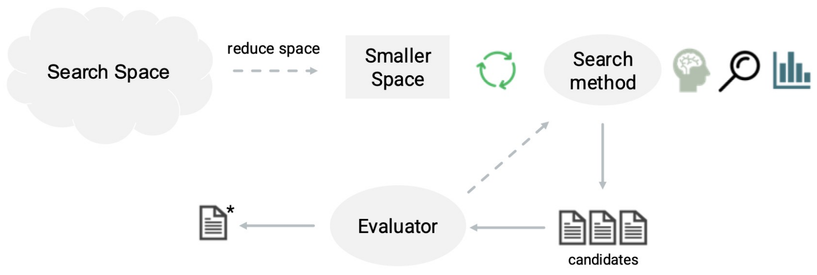


Figure 25: General Recipe for Auto-parallelization Methods

- **Search Method:** A strategy is selected to explore the reduced space, which can involve heuristics, learning-based approaches, or optimization techniques.
- **Evaluator:** The performance of different strategies is assessed, and the best candidates are iteratively refined.

This structured approach ensures that the automatic parallelization framework can efficiently find an optimal or near-optimal strategy tailored to a given model and cluster configuration.

14 Learning-based Method: ColocRL

14.1 Methodology

The ColocRL method [7] leverages an ML model to explore the space of inter-operator placement strategies using reinforcement learning. The key components of the approach include:

- **Computational Graph Representation:** The input is a directed acyclic graph (DAG), where nodes represent operations (e.g., matrix multiplications, activations) and edges represent data dependencies.
- **RL-based Policy Learning:** A sequence-to-sequence (seq2seq) model with an attention mechanism is trained to predict the optimal device placement.
- **Policy Gradients:** The model updates its policy based on real runtime measurements as rewards, improving placement decisions iteratively.
- **Execution and Evaluation:** Candidate placements are executed on real hardware, and execution time serves as a feedback signal to guide further policy optimization.

14.2 ColocRL Model

The RL agent is structured as an attentional sequence-to-sequence model, as in Figure 26:

- **Encoder:** An LSTM that processes the computational graph and encodes operation dependencies.
- **Decoder:** An LSTM that generates device assignments for each operation.

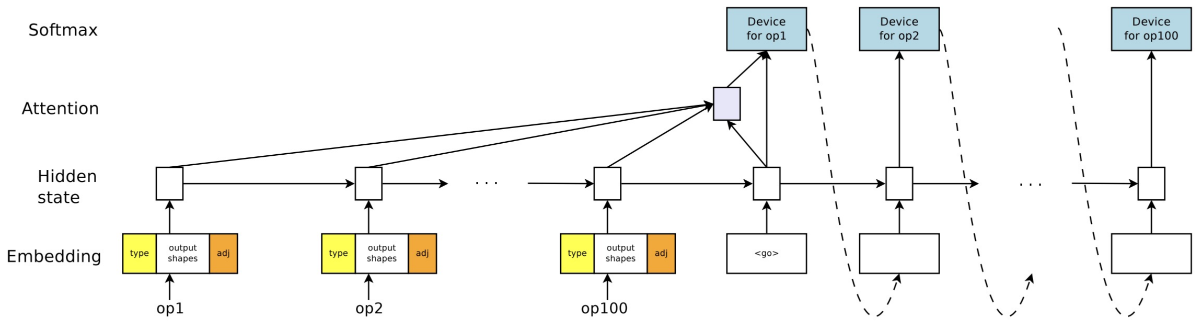


Figure 26: ColocRL Model Architecture

- **Attention Mechanism:** Helps the decoder focus on relevant parts of the graph when making decisions.

The placement decision for each operation is sequentially predicted, and the model refines its strategy through reinforcement learning.

14.3 Training Objective

The RL-based placement model optimizes the following objective:

$$J(\theta) = \mathbb{E}_{\mathcal{P}\sim\pi(\mathcal{P}|\mathcal{G};\theta)}[R(\mathcal{P})|\mathcal{G}] \tag{4}$$

where $\mathcal{G}$ represents the computational graph, $\mathcal{P}$ is a candidate device placement, $R(\mathcal{P})$ is the real execution runtime of placement $\mathcal{P}$, $\pi(\mathcal{P}|\mathcal{G};\theta)$ is the learned policy for device placement.

Policy gradients are computed via REINFORCE, adjusting model parameters θ to minimize execution time.

14.4 Results and Discussion

Experiments show that ColocRL achieves strong performance in device placement optimization. In addition to improving execution efficiency, the model also discovers non-trivial placement strategies that are unlikely to be found by human experts. These findings demonstrate the ability of reinforcement learning to uncover complex and effective placements that may not be intuitive to manual tuning.

15 Optimization-based Method: Alpa

Alpa[9] takes on an optimization-based approach to automated parallelization. The system employs **hierarchical optimization** that separates inter-operator parallelism (across devices) from intra-operator parallelism (within devices), coordinating them through a **cost feedback loop**, as shown in Figure 27.

The Alpa compiler begins by partitioning the computational graph (e.g., sequential operations like A$\to$B$\to$C$\to$D) into pipeline stages while preserving operator dependencies. Concurrently, it represents the GPU cluster as a 2D device mesh and generates candidate submesh configurations for each stage.

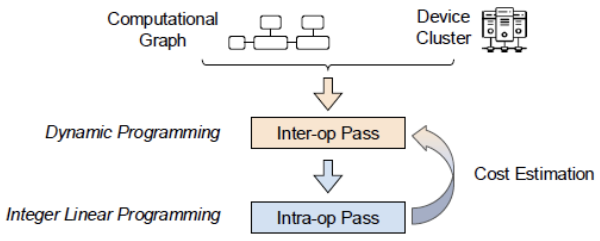


Figure 27: Alpa Compiler Hierarchical Optimization

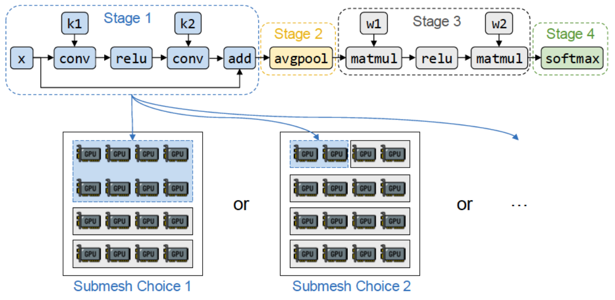


Figure 28: Inter-op Pass

Next, Alpa uses a dynamic programming approach that models pipeline latency through two components:

1. **Total stage latency** $\left(\sum_i^S t_i\right)$, where t_i is the optimal latency of executing stage i on mesh i. This is equivalent to the latency of the first microbatch going through the pipeline.
2. **Remaining execution time** $\left((B-1)\cdot \max_{1\le j\le S} t_j\right)$: pipelined execution time for the rest of $B-1$ microbatches, bounded by the slowest stage.

The optimization objective is to find the **optimal (stage, mesh) pairs that minimize** $T = \sum_i^S t_i + (B-1)\cdot \max_{1\le j\le S} t_j$. The DP algorithm first enumerates the second term $t_{max} = \max_{1\le j\le S} t_j$ and minimizes the first term for each different t_{max}.

References

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

[2] Alexander L. Gaunt, Matthew A. Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. Ampnet: Asynchronous model-parallel training for dynamic neural networks, 2017.

[3] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

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

[5] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[6] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[7] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning, 2017.

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

[9] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *CoRR*, abs/2201.12023, 2022.

16 Contributions

1. Mengyang Liu: Section 1.1 to 1.4.
2. Shuting Zhao: Section 1.5 to 2.3.
3. Yuchen Feng: Section 3 to 5.
4. Zixuan Song: Section 10 to 6.2
5. Jinyi Wan: Section 6.3 to 6.6.
6. Jenish Thanki: Section 7
7. Zhecheng Li: Section 8 to 9
8. Yuan Gao: Section 9.5 to 10
9. Heng Zhu: Section 11, proof read
10. Lifan Sun: Section 12 to 14.
11. Aaron Ang: Section 15.