Reading Summary Week 7

Aakash Agrawal
HDSI
University of California San Diego
San Diego, CA, 92092
aaa015@ucsd.edu

1 GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

It is widely recognized that larger models typically deliver better performance and improved generalization. Historically, to push model capacity beyond a single accelerator, practitioners have crafted specialized algorithms and infrastructure tailored to specific architectures and tasks. **GPipe**, however, offers a pipeline parallelism library that efficiently scales arbitrary neural networks to massive sizes, overcoming these limitations with a more flexible approach.

1.1 Batch-Splitting Strategy & Working

In GPipe, models are defined as a sequence of layers, with consecutive layer groups partitioned into cells, each assigned to a separate accelerator. GPipe employs a **batch-splitting strategy**: it divides a mini-batch of training examples into smaller micro-batches and pipelines their execution across the cells. It then uses synchronous mini-batch gradient descent for training, accumulating gradients across all micro-batches within a mini-batch and applying them at the end. As a result, GPipe ensures consistent gradient updates regardless of the number of partitions, enabling researchers to seamlessly scale up model sizes by adding more accelerators. Additionally, GPipe is **orthogonal**, meaning it can be paired with data parallelism to further enhance training scalability.

The GPipe interface is straightforward and user-friendly, requiring only three inputs:

- the number of **model partitions** (K)
- the number of **micro-batches** (M)
- the sequence and definitions of the L layers that compose the model

Next, GPipe divides the network into K cells, assigning the k-th cell to the k-th accelerator. It automatically inserts **communication primitives** at partition boundaries to enable seamless data transfer between adjacent partitions.

Algorithm:
In GPipe, during the forward pass, each mini-batch of size N is split into M equal micro-batches, which are then pipelined across the K accelerators. In the backward pass, gradients for each micro-batch are calculated using the same model parameters as in the forward pass. At the end of the mini-batch, gradients from all M micro-batches are aggregated and applied to update the model parameters across all accelerators.

1.2 Performance Optimization and Complexity Analysis

GPipe achieves a low activation memory footprint by supporting **re-materialization** (aka activation/gradient checkpointing) on every accelerator. During forward computation, each accelerator only stores output activations at the partition boundaries. During the backward pass, the k-th accelerator recomputes the composite forward function.

Space Complexity:
Let the size of the mini-batch be N partitioned into M equal micro-batches. The memory requirement without re-materialization and partitioning would be $O(N \times L)$, whereas, with re-materialization, the memory requirement is reduced to $O(N + \frac{L}{K} \times \frac{N}{M})$, where N/M is the micro-batch size, and L/K is the number of layers per partition.

Micro-batch size and the bubble overhead:
Partitioning introduces some idle time per accelerator, known as the **bubble overhead** given by $O(\frac{K-1}{M+K-1})$ when spread over M micro-steps. Experiments show this idle time becomes very small when the number of micro-batches is at least four times the number of partitions. This happens partly because re-computing values during the backward pass can start earlier, without needing to wait for gradients from previous layers.

GPipe keeps communication overhead low by only passing activation tensors at partition boundaries between accelerators. This enables efficient scaling, even on accelerators without high-speed interconnects.

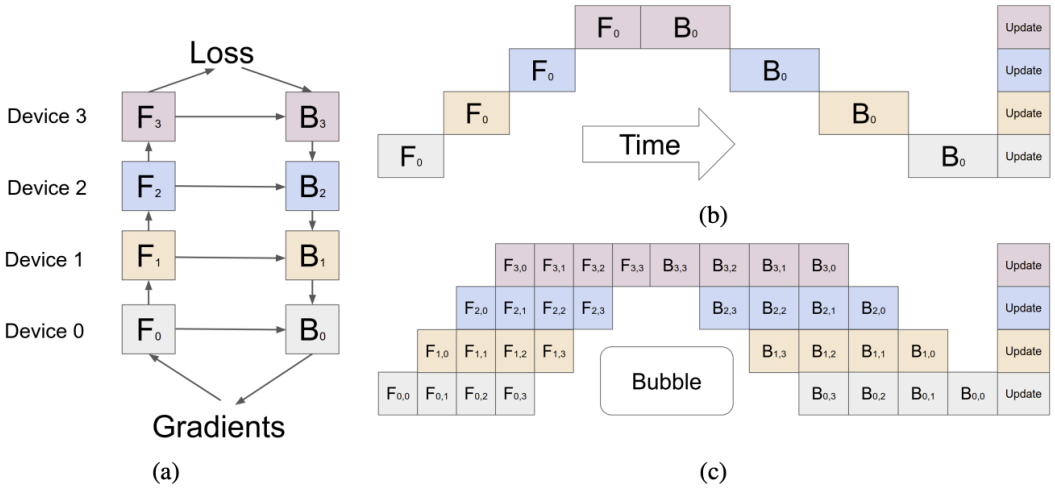

Figure 1: (a) A sample neural network with sequential layers, partitioned across four accelerators. (b) Naive model parallelism approach. (c) Pipeline parallelism implementation.

1.3 Performance Evaluation

The authors evaluate the performance of GPipe by training large-scale neural networks on two different tasks with distinct network architectures:

- Image Classification using an **AmoebaNet** convolutional model
- Multilingual Neural Machine Translation using a sequence to sequence **Transformer** model

1.3.1 Scaling Efficiency

For **AmoebaNet**, experiments ran on Cloud TPUv2s with 8GB memory per accelerator. Without GPipe, a single accelerator could handle an 82M-parameter AmoebaNet, limited by memory. GPipe cuts intermediate activation memory from 6.26GB to **3.46GB** using re-materialization in back-propagation and batch splitting, allowing a 318M-parameter model on one accelerator. With model parallelism across 8 accelerators, AmoebaNet scaled to 1.8 billion parameters—**25x** more than without GPipe. Here, the maximum model size didn't scale perfectly linearly due to uneven parameter distribution across AmoebaNet's layers.

For **Transformer**, experiments ran on Cloud TPUv3s, each with 16GB memory per core. The authors scale the model size by increasing the number of layers. Using re-materialization enabled a **2.7x** larger model on a single accelerator. With 128 partitions, GPipe scaled the Transformer to **83.9B** parameters—a **298x** jump from a single accelerator's capacity. For Transformers, the maximum size grew linearly with accelerators, as each layer maintained consistent parameter and input sizes.

1.3.2 Communication Overhead

The authors talk about the communication overhead with GPipe on GPUs (single node) without NVLinks. They observe a **2.7x** speedup for AmoebaNet-D (18, 128) model when the number of partitions (devices) is increased from 2 to 8. For the 24-layer Transformer, the authors report a speedup of **3.3x** when the number of partitions is increased from 2 to 8. This indicates that the communication bandwidth between devices is no longer a bottleneck for model parallelism since GPipe only transfers activation tensors at the boundaries of partitions.

1.3.3 Image Classification

The authors report their 557-Mn parameter AmoebaNet model, trained with GPipe, achieves a top-1 accuracy of **84.4%** on ImageNet-2012, beating the previous SOTA. The authors also use a transfer learning approach to fine-tune the trained model on a variety of datasets and achieve competitive performance on those as well.

1.3.4 Massive Massively Multilingual Machine Translation

The authors report that a single 6-Bn parameter, 128-layer Transformer model trained on a corpus spanning over 100 languages achieves better quality than all bilingual Transformer models. In order to test the flexibility of GPipe, the authors also perform several scaling experiments along the depth (increasing the number of layers) and width (increasing the hidden dimension) of the transformer model. It is evident that increasing the model size leads to significant quality improvements across all the languages.

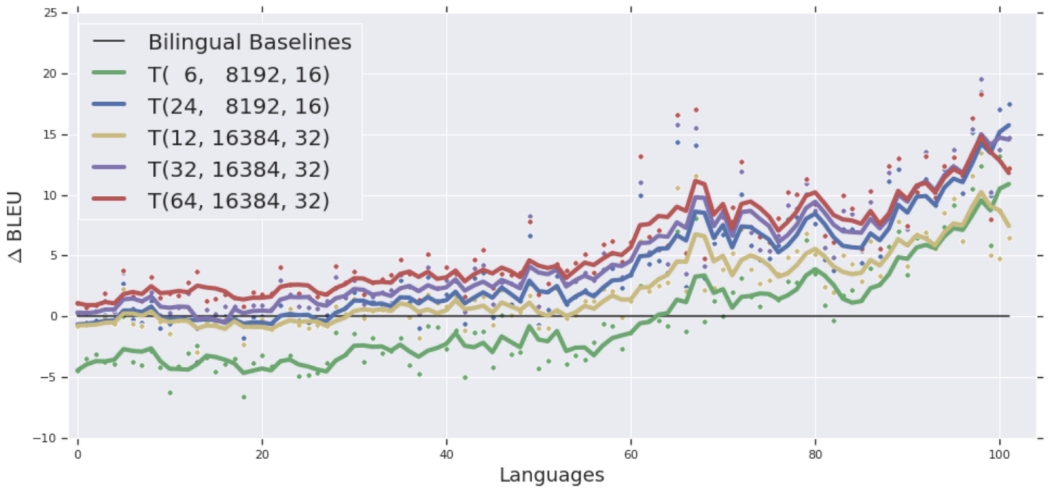

Figure 2: Translation quality across all languages with increasing multilingual model capacity.

Depth-Width tradeoff:
The 1.3B-parameter wide model (12 layers, 16384 width, 32 heads) and the 1.3B-parameter deep model (24 layers, 8192 width, 16 heads) show similar performance on high-resource languages. However, the deeper model significantly outperforms on low-resource languages, suggesting deeper models enhance **generalization**.

Trainability Challenges with Deep Models:
Deeper neural networks boost representational power but complicate optimization. In large-scale tests, the authors observed sharp activations and noisy data caused severe trainability issues, noting that after a few thousand steps, predictions became overly peaky and noise-sensitive, leading to unstable gradients that disrupted learning. To address these issues, authors adopt two strategies: scaling down the initialization of feed-forward layers by the number of layers and clipping the softmax pre-activations.

Table 1: **Summarizing the Pros & Cons of GPipe**

Pros	
1.	Scaling to many micro-batches reduces bubble overhead without requiring asynchronous gradient updates.
2.	Model size scales linearly with the number of accelerators.
3.	Minimal additional communication overhead when scaling the model.
4.	Inter-device communication occurs only at partition boundaries per micro-batch, with marginal overhead, making GPipe effective even without high-speed interconnects.
Cons	
1.	Assumes a single layer fits within one accelerator's memory.
2.	As micro-batch size increases, GPU memory usage eventually spikes dramatically. At its peak, both model parameters and activations for all micro-batches must be stored simultaneously, pushing memory limits.

2 Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

Existing model-parallel techniques either require users to manually create a parallelization plan or generate one automatically from a restricted set of model configurations. In contrast, Alpa is a **compiler** that automates the process of generating execution plans, covering data, operator, and pipeline parallelism. It organizes parallelism into a **hierarchical** structure by combining inter-op and intra-op techniques and uses a series of compilation steps to determine efficient execution plans at each level.

2.1 Background

1. **Data Parallelism**: In this approach, the training data is distributed across multiple workers, while the model is replicated on each worker. Each worker computes parameter updates based on its own data subset and then synchronizes the updates with other workers to ensure that all workers have consistent model parameters throughout the training process.

2. **Operator Parallelism**: This model parallelism technique partitions the computation of a specific operator along the non-batch dimension, and each part of the operator is computed in parallel across devices. Communication between devices is required to fetch data for computation.

3. **Pipeline Parallelism**: This model parallelism technique does not partition operators. Instead, it assigns different groups of operators to different devices and splits a minibatch into smaller micro-batches, processing them sequentially across the devices.

4. **Inter-operator Parallelism**: This involves parallelizing different operations or stages of a model across multiple devices or processors. Each device handles a different operation, enabling parallel computation of the model. Examples include **GPipe**, **PipeDream**, **1F1B**, etc.

    - Here, devices communicate using **P2P** (point-to-point) **communication**. Due to the data dependency between stages or devices, inter-op parallelism results in bubble overhead as some devices become idle during forward and backward computation.

5. **Intra-operator Parallelism**: This refers to parallelizing a single operation, such as matrix multiplication, within a device. The operation is divided into smaller tasks, with each task executed in parallel on different parts of the device. Data parallelism is considered a form of intra-operator parallelism. Examples include **ZeRO** (an optimization of data parallelism), **Megatron-LM**, etc.

    - Due to the partitioning, **collective communication** is required at the split and merge of the operator, which results in substantial communication overhead among distributed devices. However, intra-op parallelism has better device utilization.

2.2 Optimal Execution Plans

It's evident that inter-op and intra-op parallelism takes place at different levels or granularities of the DL computation and have distinct communication patterns. Alpa exploits this idea to design hierarchical parallelization algorithms, and compilation passes to auto-generate execution plans. In this hierarchical setup, intra-op parallelism can be assigned to devices with high-bandwidth connections, while inter-op parallelism is coordinated across distant devices with comparatively lower bandwidth.

At the **intra-op level**, Alpa optimizes the execution cost of a stage (a dataflow subgraph) by determining the best intra-op parallelism strategy for a given device mesh. A device mesh consists of multiple devices, typically with high-bandwidth connections.

At the **inter-op level**, Alpa minimizes the latency of inter-op parallelization by deciding how to partition the model and the device cluster into stages and device meshes, then mapping them as stage-mesh pairs. This optimization relies on execution cost estimates for each stage-mesh pair, which are provided by the intra-op optimizer.

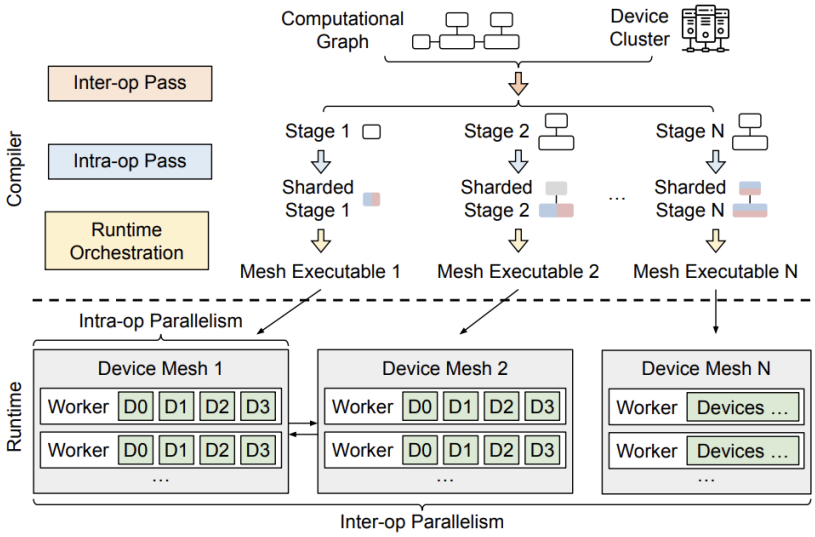

Figure 3: Alpa compilation passes and runtime architecture

To achieve optimal execution plans, Alpa implements three key compilation passes:

1. **Inter-op compilation pass**: Given a model description and cluster configuration, this pass partitions the computation into stages and the device cluster into multiple device meshes. It uses a **Dynamic Programming** (DP) algorithm to assign stages to meshes, invoking the intra-op compilation pass to estimate execution costs for each assignment. By iterating over different allocations, it minimizes inter-op parallelization **latency** and determines the best stage-mesh slicing strategy.
2. **Intra-op compilation pass**: Once assigned to a mesh, each stage undergoes intra-op optimization to determine the best intra-operator parallel execution strategy. This pass minimizes execution cost using an **Integer Linear Programming** (ILP) formulation and reports the cost back to the inter-op pass. This execution cost depends on the **compute**, **communication**, and **resharding** costs.
    - **Resharding**: If one stage col-partitions a tensor and the subsequent stage row-partitions a tensor, a resharding operation (aka layout conversion) is needed, which incurs a **cross-device** communication cost.
    - There can be several parallel algorithms for an operator, each with its own communication costs. The goal of the ILP is to find the algorithm that minimizes the total execution cost of the computational graph and output an intra-op parallelism plan.
    - The compilation automatically inserts collective communication primitives to handle the within-mesh communication required by intra-op parallelism.

3. **Runtime orchestration pass**: Based on the final hierarchical execution plan and pipeline-parallel schedule, this pass ensures communication between adjacent stages assigned to different meshes. It generates static execution instructions for each mesh and orchestrates execution across all meshes according to the pipeline-parallel schedule.
    - **Cross-mesh resharding**: In Alpa, the device meshes hosting two adjacent stages may have different mesh shapes, and the tensor exchanged between these stages may have different sharding specifications. This communication pattern is referred to as cross-mesh resharding.
    - Alpa generates a cross-mesh communication plan in two iterations. In the first, it calculates tensor partition correspondences between source and destination meshes, generating P2P send/recv primitives for communication. In the second iteration, Alpa identifies replication opportunities in the destination mesh's sharding spec, replacing send/recv with all-gather to avoid redundant transfers, utilizing the higher bandwidth of the destination mesh.

Finally, Alpa uses an **MPMD**-style (Multiple Program, Multiple Data) runtime to orchestrate inter-op parallel execution, creating distinct instructions for each device mesh.

Table 2: **Summarizing the Pros & Cons of Alpa**

Pros	
1.	Unlike Megatron-LM, Alpa allows for pipeline stages having an uneven number of operators or layers.
2.	Pipeline stages in Alpa can map to device meshes with different shapes.
3.	Within each stage, the data and operator parallelism configuration is tailored individually for each operator.
4.	As a generic compiler system, Alpa can compose a wide range of parallelism approaches.
Cons	
1.	Alpa does not model cross-stage communication costs, as they are typically small. Doing so would significantly increase the complexity by requiring many more intra-op passes and DP states.
2.	The desired number of micro-batches to use is not taken into account by Alpa's framework.
3.	The inter-op pass does not model dynamic schedules. It only models pipeline parallelism with **static linear schedules** (stages of the pipeline executed one after the other).
4.	Alpa can only handle **static computational graphs** where all tensor shapes are known at compile time.

2.3 Performance Evaluation

The authors evaluate Alpa on large-scale models like **GPT-3**, **GShard** Mixture-of-Experts (**MoE**), and **Wide-ResNet**. For GPT-3, they compare against **Megatron-LM** baseline, while for GShard MoE, they use **DeepSpeed**, which features handcrafted operator parallelism and ZeRO-based data parallelism. Since no existing baseline exists for Wide-ResNet, they construct a "**PP-DP**" baseline using only data and pipeline parallelism. The authors evaluate performance using **PFLOPS** for the entire cluster, measured by running a few batches with dummy data after a proper warm-up.

- **GPT-3 results**: **Intra-op only** parallelism struggles on more than 16 GPUs due to heavy cross-node communication, which creates a **bottleneck**. In contrast, **Inter-op only** parallelism performs well and scales linearly up to 64 GPUs. While Megatron-LM achieves super-linear weak scaling on GPT-3, Alpa automatically generates execution plans and achieves slightly better scaling in several scenarios. Further, Alpa improves performance slightly over Megatron-LM by partitioning weight update operations when data parallelism is used, a feature currently missing in Megatron-LM.
- **MoE results**: DeepSpeed's expert parallelism lacks inter-op parallelism, limiting scalability beyond ≤ 8 GPUs due to low inter-node bandwidth. Intra-op only parallelism faces the same issue. Inter-op only parallelism runs out of memory on 32 and 64 GPUs because the model cannot be evenly sliced when GPUs outnumber layers, causing memory-intensive stages to exceed available memory. Alpa maintains linear scaling on 16 GPUs and scales well to 64 GPUs. Compared to DeepSpeed, Alpa achieves **3.5x speedup** on 2 nodes and a **9.7x speedup** on 4 nodes.

- **Wide-ResNet results**: Wide-ResNet has a heterogeneous architecture, with shrinking activation tensors and growing weight tensors, leading to imbalanced memory and compute distribution across layers. This makes manual plan design challenging. However, Alpa achieves scalable performance on 32 GPUs with **80%** scaling. In contrast, the PP-DP and Inter-op only baselines run out of memory on large models due to the inability to partition weights and balance memory usage. Intra-only struggles with slow communication, preventing it from scaling across multiple nodes.