Week 7 Reading Summaries: GPipe + Alpa

Amogh Patankar
University of California, San Diego
3869 Miramar Street Box #3037, La Jolla, CA- 92092
apatankar@ucsd.edu

Abstract

With these readings, we dive deep into GPipe and Alpa, two methods to use parallelism in order to scale the sizes of workloads executed. GPipe uses a strategy called micro-batch parallelism, resulting in an almost linear speedup. Alpa uses inter-parallelism and intra-parallelism for distributed deep learning, outperforming hand-tuned systems, and generalizing to models as well.

1 GPipe

1.1 Introduction

Deep learning has improved due to the scale of neural networks, and is especially true for computer vision tasks. However, scaling these networks brings up challenges such as memory constraints, and bandwidth issues. The solutions cannot generalize well; GPipe allows to efficiently train a neural network, by creating microbatches. Complementing GPipe with data parallelism is the best way of scaling training.

1.2 GPipe Library

The GPipe library has many design features.

1.2.1 Interface

GPipe's interface allows user specification for a cost estimation function for a neural network. That is, the user only specifies model partitions, # of micro-batches, and sequences/definitions of their layers.

1.2.2 Algorithm

GPipe's algorithm partitions the network using the number of partitions specified. It then divides mini-batches into micro-batches during forward pass, and computes gradients for each during the backward pass. Finally, gradients are all accumulated and update the model parameters.

1.2.3 Performance Optimization

GPipe allows only output activations to be stored in forward passes, and recomputations of the forward function in backward passes. GPipe has a low communication overhead, and scaling performance can be boosted on any kind of accelerators de to passing activation tensors at boundaries.

1.3 Performance Analyses

With respect to performacne analysis of GPipe, they evaluate based on AmoebaNet CNN and Transformer Seq2Seq model. Notably, they show that with a higher number of micro-batches, the overhead is negligible, and that with a small M value, the overhead is no longer negligible.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

1.4 Image Classification

The authors scale AmoebaNet using GPipe, and show that scaling through GPipe results in higher accuracy in six of the eight tests they run on vvarious image classification datasets.

1.5 Massive Massively Multilingual Machine Translation

The authors use GPipe to scale neural machine translation models for NLP tasks, uing a multilingual dataset of 25 billion training examples across 102 languages and English. They show the ability to handle both low-resource and high-resource languages, demonstrating that a single NMT model can learn mappings and outperform bilingual models. Comparing a 1.3B wide model and a 1.3B deep model, the deeper model significantly outperforms the wide model on low-resource languages, meaning that increasing depth improves generalization. Training deep models has its challenges including sharp activations and dataset noise, causing training instability.

1.6 Design Features and Tradeoffs

Single Program Multiple Data (SPMD) methods like Mesh-Tensorflow can enable linear scaling of model parameters but have high communication overhead and limited applicability. In contrast, pipeline parallelism approaches like PipeDream struggle with weight staleness and memory inefficiency. GPipe minimizes communication overhead and bubble inefficiency by pipelining micro-batches and applying synchronous gradient updates. The only assumption is single-layer memory constraints.

2 Alpa

2.1 Introduction

Deep learning advances recently, have been simply to increase the model size. The issue here is that it requires manual tuning and scaling models efficiently is very complex. The authors automate the parallelization of large-scale models, to speed up ML research. Moreover, parallel processing can occur hierarchically, through intra- and inter-operator parallelism.

2.2 Background: Distributed Deep Learning

2.2.1 Conventional View of ML Parallelism

There are four approaches: data parallelism, operator parallelism, pipeline parallelism, and a combination of parallelisms. In essence, all these parallelisms are used to compute workloads in some parallel schema, and a combination of parallelisms means you have a configuration to follow.

2.2.2 Intra- and Inter-Operator Parallelisms

Intra-operator parallelism refers to partitioning tensors along a dimension, assigning the computations to multiple devices, and execute those portions simultaneously. Inter-operator parallelism means that different operators execute on different distributed devices. Both have differing granularities, but both fit within compute cluster structure.

2.3 Overview

Alpa generates model-parallel optimization plans at both parallelism levels. Alpa minimizes interop parallelization latency, and minimizes the execution cost for intraop parallelizaton.

2.4 Intra-Operator Parallelism

Alpa optimizes parallelism within a device mesh, namely, the SPMD intraop parallelism.

2.4.1 Space of Intra-Operator Parallelism

The authors note that a 2D device mesh with different parallelization strategies, and HW tradeofffs occur. IN addition, sharding defines partitioning of axes.

2.4.2 ILP Formulation

Integer Linear Programming minimizes the total execution cost of a computational graph including compute, communication, and resharding. Communication and compute are both estimated based on bytes and bandwidth with lightweight operators merged to simplify the graph. After ILP, post-optimization techniques like reduce-scatter and all-gather as opposed to all-reduce are used to reduce communication and computation overhead.

2.5 Inter-Operator Parallelism

2.5.1 Space for Inter-Operator Parallelism

Using a computational graph, we define the latency as the minimized ILP reported by the intra-op pass. We try to solve this by adding a constraint for the forward pass, and backward latency.

2.6 Parallelism Orchestration

Parallel orchestration is used to address cross-mesh communication; cross-mesh sharding and pipeline execution instruction generation are used. Cross-mesh sharding is responsible for sending/receiving, and gathering instructions in cross-mesh, while pipeline execution instructions are generated by Alpa.

2.7 Limitations and Discussion

The author talk about how Alpa's hierarchical view of inter- and intra-operatior parallelisms gives three flexibilities, namely, random # of layers, various shape maps, and non-uniform parallelism configuration.

2.8 Evaluation

2.8.1 End-To-End Performance

The authors evaluate Alpa by using 16K LoC and 6K LoC in Python and C++, with Jax and XLA. Namely, they show that with an exceeding # of GPUs, Alpa outperforms Megatron-LM, interop only, and intraop only- for GPT, MoE, and ResNet architectures, using metrics like PFLOPS.

2.8.2 Intra-Op Parallelism Ablation Study

In this study, the authors compare the ILP solution with alternatives like ZeRO; they note that vanilla data parallelism runs out of memory, ZeRO-methods don't optimize, while ILP is the highest performing.

2.8.3 Inter-Op Parallelism Ablation Study

In this study, the authors compare the DP solution with equal layer and equal operator; they note that the DP method always outperforms equal operator, while DP vs equal layer depends on model architecture. This is dependent on model and compute, both simultaneously.

2.8.4 Compilation Time

Alpa scales very well to large models and/or clusters, as compilation time grows linearly with model size and # of GPUs.

2.9 Cross-Mesh Resharding

They enable all-gather optimizations to increase speedup.

2.10 Wide-ResNet

In the case of Wide-ResNet, Alpa uses intra-operator parallelism, partitioning along the batch axis for the few layers. For higher quantities of GPUs, Alpa slices the model into more stage, with more GPUs assigned to later stages.

3 Related Work

The authors discuss systems for DP training like Horovod, systems for MP training like PipeDream, and more advanced techniques to optimize DL execution through compilers.