Advanced Topics in Distributed Systems

Paper Review by Tianze Wang

April 8, 2020

The instruction of the review is:
"Task 4: write a critical review of the papers that covers in particular the summary of contributions, solutions, significance, and technical/experimental quality."

1 Introduction

Now at the start of 2020, it is hard to deny that Machine Learning (ML) is everywhere. This is perhaps even more true when it comes to Deep Neural Networks (DNNs) and, thanks to the availability of large datasets, DNNs have driven advances in many practical problems, e.g., image classification, speech recognition, machine translation, image generation, and autonomous driving.

However, together with the boost in performance of Deep Learning models, recent years have also witnessed dramatic growth in the computational requirements for training DNNs. The computational requirements have grown to the point where distributed and paralleled training is now standard practice. Yet, the benefits of employing distributed training can easily be hindered by different factors: e.g., the naive parallelization strategy, the communication-heavy parameter synchronization process, etc. Popular methods to cope with the problems includes but are not limited to:

- Data parallelism
  - efficient for compute-intensive operators with few parameters (e.g., convolution);
  - sub-optimal for operators with a large number of parameters (e.g., embedding).
- Model parallelism
  - eliminates parameter synchronization between devices;
  - but requires data transfer between operators.
- System optimizations: (based on data parallelism)
  - allreduce operation to optimize communication;
  - overlaps gradient synchronization with backpropagation.
- Network parameter reduction
  - Weight pruning method that removes weak connections.
- Gradient compression
  - Lossy compressions like gradient quantization and sparse parameter synchronization.

In the rest of the review, we will be looking into three methods to improve the performance of distributed training. The paper that is going to be covered can be found below.

Papers

1. Jia, Zhihao et al. "Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks." ArXiv abs/1802.04924 (2018): n. pag;
2. Jia, Zhihao et al. "Beyond Data and Model Parallelism for Deep Neural Networks." ArXiv abs/1807.05358 (2018): n. pag;
3. Jayarajan, Anand et al. "Priority-based Parameter Propagation for Distributed

DNN Training." ArXiv abs/1905.03960 (2019): n. pag.

Hereafter, we are going to refer to these three papers by the names of the frameworks they propose which are **OptCNN** for the first paper, **FlexFlow** for the second paper, and **P3** for the third paper.

2 OptCNN

2.1 Motivation

The major motivation behind the paper is from the observation that applying a single parallelization strategy to all layers in a network could result in sub-optimal runtime performance in large scale distributed training because different layers in a network may prefer different parallelization strategies. For example, densely-connected layers with millions of parameters may prefer model parallelism to reduce communication cost, while convolutional layers typically prefer data parallelism to minimize data transfers between two consecutive layers.

2.2 Main Contribution

Intending to find the parallelization strategies for individual layers to jointly achieve the best possible runtime performance while maintaining the original network performance, the paper proposes layer-wise parallelism which:

- Enables each layer in a network to use an individual parallelization strategy;
- Performs the same computation for each layer as the original network;
- Provides a more comprehensive search space which includes data and model parallelism.

2.3 Solution

In standard CNNs for 2D images, data is usually organized as 4-dimensional tensors (i.e., sample, height, width, and channel):

- Sample dimension: partition the training dataset;
- Height and width dimensions: partition the image/feature map;
- Channel dimensions: different neurons for different output channels.

Thus, to parallelize a convolutional layer we should: select from all dimensions and consider the degree of parallelism in each dimension. Each parallelization problem is defined with two graphs: a device graph $\mathcal{D}$ which models all available hardware devices and the connections between them and a computation graph $\mathcal{G}$ which defines the neural network to be mapped onto the device graph. The solution is represented as a parallelization strategy $\mathcal{S}$ which includes a configuration c_i for each layer $l_i \in \mathcal{G}$ and a parallelization configuration c_i of a layer l_i defines how l_i is parallelized across different devices. To find a solution for the problem, a cost function is defined as an estimated per-step execution time for a given parallelization strategy which takes into account the timea to process each layer, the time to synchronize the parameters of each layer, and the time to transfer input tensors to target devices. Then a graph search method is used to find the best parallelization strategy based on the cost function.

2.4 Advantages

- The paper has demonstrated through empirical evaluation that using OptCNN can increase training throughput and reduce communication cost when compared with data parallelism, model parallelism, and some expert strategies;
- Through the analysis of the optimal parallelizations strategies, the authors find some common patterns in the solution which may be seen as a heuristic when parallelizing CNNs;
- The framework is easy to understand,

and the authors also introduce search to help to parallelize CNNs.

2.5 Disadvantages

- Some of the dimensions are left unexplored by OptCNN which is later improved by another paper from the same group;
- While the searching strategy does open a larger search space to find parallelization strategies. The searching schema the is used in the paper can take time to run as it requires running on the actual hardware for each proposed parallelization strategies which could be time-consuming when facing a large search space;
- While the author did provide their own implementation using Legion, cuDNN, and cuBLAS, it might be hard to implement OptCNN in Tensorflow or PyTorch as some of the device placement operations might not be fully supported which could be a barrier for using OptCNN in a production environment.

3 FlexFlow

3.1 Motivation

The motivation of the paper is based on the observation that Tensorflow, PyTorch, Caffe2 are mainly based on data and model parallelism. Popular deep learning frameworks don't exploit is operation level parallelism. For example, the convolution operation can be distributed along the channel or spatial dimensions. While manually-designed strategies that combine data and model parallelism do manage to accelerate specific DNNs, it usually requires a lot of time for the trial and error that comes along the way. Exploring dimensions beyond data and model parallelism can further accelerate DNN training (by up to 3.3x).

3.2 Main Contribution

To automatically exploring the dimensions beyond data and model parallelism, the paper has the following contribution:

- Introduce the SOAP search space for parallelizing DNN applications:
  - **S**amples: partitioning training samples (Data Parallelism);
  - **O**perators: partitioning DNN operators (Model Parallelism);
  - **A**ttributes: partitioning attributes in a sample (e.g., different pixels);
  - **P**arameters: partitioning parameters in an operator.
- Demonstrate that under reasonable assumptions it is possible to reliably predict the execution time of parallelized DNNs using a simulator that is three orders of magnitude faster than directly running the DNNs directly on the target device (OptCNN);
- Proposed a deep learning framework, FlexFlow, that can automatically search optimal parallelization strategies, which can increase training throughput by up to 3.8x compared to state-of-the-art parallelization approaches while improving scalability, in the SOAP search space.

3.3 Solution

The solution proposed by FlexFlow is quite similar to the one proposed by OptCNN. FlexFlow aims to find the optimized parallelization strategies with a search space of possible strategies (which includes information of the computation graph and device topology), a cost model and a search algorithm. Apart from a more comprehensive SOAP search space, the major delta from OptCNN is the execution optimizer which, instead of using dynamic programming, makes use of an MCMC search algorithm and an execution simulator to efficiently explore the

SOAP search space and find the best parallelization strategy.

3.4 Advantages

- SOAP is a larger search space that covers more aspects that we can parallelize DNNs;
- The execution simulator is way much faster than actually running each candidate strategy on the hardware which leads to more search space to be explored given the same amount of time;
- Evaluations on how to parallelize RNN models are also given which is missing from OptCNN;
- FlexFlow guarantees that the same computation as training on a single GPU. The evaluation results are very positive. For the task of training Inception-v3 on the ImageNet dataset until the model reaches the single-crop top-1 accuracy of 72% on the validation set on 16 P100 GPUs (4 nodes), FlexFlow reduce the training time by 38% compared to TensorFlow.

3.5 Disadvantages

- Again, FlexFlow uses a runtime that can facilitate device placement which one might not have access to when using popular Deep Learning frameworks such as TensorFlow;
- FlexFlow has ignored the strategy that uses output reduction which is later taken into consideration by a paper that proposed the TOFO framework.

4 P3

4.1 Motivation

The main motivation of the paper is based on the fact that many Deep Learning frameworks and runtime use the technique of overlapping communication with computation which has been shown to improve the training throughput especially when it comes to optimizing the path and pipelining of data movement across the machine. To be more specific, P3 aims to overlap parameter synchronization with computation to improve the training performance based on the following observations:

- The optimal data representation granularity for the communication may differ from that used by the underlying DNN model implementation;
- Different parameters can afford different synchronization delay;
- The traffic generated by the training processes are generally bursty.

4.2 Main Contribution

The paper has the following contributions:

- Showed that parameter synchronization at layer-wise granularity can cause sub-optimal hardware utilization in certain models. Showed that parameter synchronization can be scheduled better to efficiently use the network bandwidth;
- Propose a new synchronization mechanism called Priority-based Parameter Propagation (P3);
- Implemented and open source P3 on MXNet and evaluated the performance against standard MXNet implementation.

4.3 Solution

To better schedule parameter synchronization, P3 takes into account when the gradients are generated and when the gradients are consumed. For example, during training, the gradients of the layers are generated from final to initial layers and subsequently consumed in the reverse order in the next iteration. Also, P3 takes into consideration the fact that layer-wise granularity may not always be optimal for parameter synchronization.

P3 has two main components: a Parameter Slicing component, which splits layers into smaller slices and synchronizes independently, and a Priority-based Update component, which synchronizes parameter slices based on their priority. The priorities of layers are based on the order that the layers are processed in the forward propagation. And during backpropagation, P3 always allocates network cycles to the highest priority slices in the queue, preempting synchronization of the slices from a previous lower priority layer if necessary.

4.4 Advantages

- Taken into consideration of bandwidth limit. P3 performs really well when there are tight limitations on the bandwidth;

- P3 is orthogonal to device placement, which leads to the possibility of combing P3 with FlexFlow to further improve the distributed training throughput.

4.5 Disadvantages

- While P3 provides an open source implementation using MXNet, other popular deep learning frameworks, e.g., Tensorflow and PyTorch, might not expose the low-level control on how parameters are synchronized;
- Is it possible to further improve the performance by doing parameter synchronization directly through GPU memory? Currently, when doing parameter synchronization, a single parameter might need to be copied from a GPU's VRAM to the RAM and then to another GPU's VRAM. Is it possible to further improve P3 by letting GPUs that sit on a single node to directly talk to each other?

5 Conclusion

To conclude, many techniques can be applied to increase the throughput of distributed taring. OptCNN and FlexFlow focus on searching through multiple dimensions while P3 aims to better schedule the parameter synchronization.