Auto-Parallelizing Deep Learning for Multi-machine, Multi-GPU Environments

Soojeong Kim Eunji Jeong Joo Seong Jeong Gyeong-In Yu Hojin Park Byung-Gon Chun *
Seoul National University

1. Introduction

Being able to use a cluster of GPU resources is favorable especially for machine learning researchers, as training neural nets typically requires at least tens of GPUs in order to finish within feasible time. Neural networks can be made to be trainable on multiple devices, which are CPUs or GPUs in multi-machines, to speed up training and improve convergence. This is further facilitated by the introduction of deep learning systems such as TensorFlow [2], MXNet [4] and Caffe2 [1]; such frameworks allow users to easily utilize multi-machine, multi-GPU environments to train networks, to a certain degree.

Unfortunately, extending single-machine, single-GPU neural net models to work in distributed environments is not a trivial job for common machine learning researchers. In most deep learning frameworks [1, 2, 4], a deep learning job is represented as a computation graph, and the computation graph needs to be changed for distributed environments, which requires partitioning model parameters across machines to balance out communication overheads, as well as replicating and assigning graph operators to devices so that hardware resources can cooperate to train a model.

To this end, we introduce Parallax, an auto-parallelization module that helps machine learning researchers extend their single-model code to operate in data parallelism with multi-GPU and multi-machine. Parallax receives a single-device graph, analyzes the graph, then transforms it into a multi-machine, multi-GPU version of the computed settings. The automatically transformed graph can finally be run in distributed environments.

Preliminary experiments show that with the help of Parallax, the ResNet-50 [9] model can be trained on a total of 12 GPUs across 3 machines with sublinear scale-out improvements in computation throughput. We also discuss several extensions on Parallax, including the application of model parallelism strategies to boost performance for models with relatively large parameters, as well as hybrid parallelism strategies that utilize both data parallelism and model parallelism.

* Corresponding author.

2. Parallax Overview

Parallax provides transparent data parallelism, automatically executing single-GPU computation graphs on distributed environments. Parallax is implemented on top of one of the state-of-the-art deep learning frameworks, TensorFlow [2]. We assume the parameter server (PS) architecture [5, 6, 11], in which *server* processes store network parameters and *worker* processes perform the main computation according to the given computation graph with one or more GPUs. Note that Parallax is not necessarily bound to the PS architecture; Parallax can be implemented on other deep learning frameworks that do not provide explicit PS abstractions such as Caffe2 [1].

Execution Model The following steps outline the overall execution model of Parallax. First, the user gives a computation graph and cluster information to a machine in the cluster, which becomes in charge of initiating the auto-parallelization mechanism. Next, the computation graph is analyzed, and sent to other machines in the cluster to be transformed into a distributed version. Note that the transformation process is not necessarily all done in one machine; rather, the master machine just determines how the input graph should be transformed and sends that information, together with the input graph itself, to the other machines. It is only after the transform information arrives at other machines that the input graph is converted to its distributed version. Such a parallelization mechanism is required because the graph transformation process can take a significant amount of time if the target graph is large, potentially making the master machine a bottleneck. Finally, each machine executes its graph, resulting in distributed training.

Graph Analysis Parallax analyzes the input graph to group consisting operators into several categories, in order to decide what action to apply to which operator. Parallax automatically classifies operators according to the structure of typical deep learning jobs, which contain feedforward and backpropagation computation operators and optimizers for updating computed gradients. This reduces user efforts for manually parsing the computation graph.

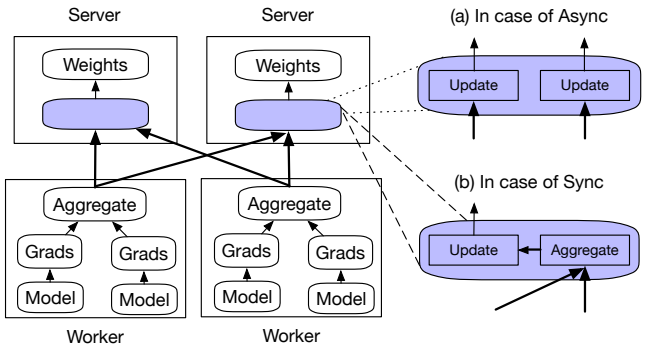

Figure 1: Graph conversion for various operators.

Graph Conversion Parallax parallelizes the execution of deep learning jobs by transforming their corresponding computation graphs, replicating graph operators and partitioning model parameters across machines.

The replication and device assignment policies of operators differ per operator type: main computation operators, weights, and weight update operators (Figure 1).

Main computation operators that are used to generate gradients are replicated as much as the number of worker and given to each worker (*Model* and *Grads* in Figure 1). This allows workers to perform computation in parallel on different batches of data, i.e. data parallelism.

Network parameters, or weights, are evenly partitioned across servers, based on their sizes (*Weights* in Figure 1). Unlike computation operators, weights must be shared across workers and are not replication targets. Rather, in order to balance out communication overheads across machines, Parallax assigns weights to each device while keeping the deviation of communication overhead as low as possible, partitioning large-sized weights into smaller portions if necessary.

Weight update operators are treated slightly differently from the previous categories. In fact, the replication policy depends on the communication scheme: asynchronous training or synchronous training. For asynchronous training, gradients produced by workers are applied to network parameters without any coordination, thus weight update operators are simply replicated, one for each worker (*Update*s in Figure 1(a)). On the other hand, for synchronous training, gradients are aggregated before being applied to the parameters, therefore weight update operators are not replicated. Instead, Parallax inserts additional aggregation operators that collect gradients from each worker. The aggregators wait until all workers send their gradients, and then pass the gradient means to the weight update operators (*Update* and *Aggregate* in Figure 1(b)).

3. Preliminary Evaluation

We experiment on a homogeneous GPU cluster of three servers, each of which is shipped with four NVIDIA Titan Xp GPUs. Each GPU processes a batch of 64 data instances, therefore using more GPUs increases the total batch size. The servers are connected via Mellanox ConnectX-4 cards with IB (100Gb/s).

# GPU	Throughput	Convergence speed	
		Top-1	Top-5
1	88	0.04	0.10
4	324	0.10	0.18
12	838	0.12	0.20

Table 1: Evaluation of ResNet-50 on 1, 4, and 12 GPUs. Throughput and convergence speed are measured as the number of processed instances per second and the improved accuracy per hour, respectively.

We trained ten models using Parallax, including CNNs [7, 9, 14], RNNs [17], encoder-decoder models [3, 10], and mixture models [8, 13, 15, 16]. In this paper, we present the training results of the ResNet-50 [9] model with synchronous SGD on 100K images that were randomly sampled from the ImageNet [12] dataset. We compare a 4-GPU setting of a single machine and a 12-GPU setting of three machines with a single-GPU baseline.

Table 1 shows the evaluation results. The computation throughput increases sublinearly as more GPUs are used, and the convergence speed improves as well.

4. On-going Work

Automatic Model Parallelism In model parallelism, devices cooperate to process a single mini-batch data by distributing the operators of a graph across themselves, exchanging neuron activation values instead of model parameters. We are extending Parallax to determine the most efficient partitioning scheme for model parallelism during graph analysis, by putting partition boundaries between operators that communicate small activation values to minimize the total communication overhead, while balacing out the computation overhead for devices at the same time. This is especially effective for large models that do not fit in a single GPU device due to GPU memory constraints and thus must be trained on more than one GPU.

Automatic Hybrid Parallelism Hybrid parallelism is a mixed form of parallelism; hardware resources are sorted into several groups, and the devices in each group hold a model replica. This achieves data parallelism by training multiple replicas for a single model while also achieving model parallelism as each replica is processed by more than one device, though determining the number of replicas and the number of devices per replica is not straightforward. We are expanding Parallax to predict the optimal hybrid parallelism strategy for a model under a distributed environment, through systematic calculation of communication overheads and computation throughput per replica setting.

Acknowledgement

We thank anonymous reviewers for their comments. This research was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TC1603- 01.

References

[1] Caffe2. `https://caffe2.ai`.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[3] K. Bousmalis, G. Trigeorgis, N. Silberman, D. Krishnan, and D. Erhan. Domain separation networks. In *NIPS*, 2016.

[4] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[5] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[7] L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

[8] C. Finn, I. Goodfellow, and S. Levine. Unsupervised learning for physical interaction through video prediction. In *NIPS*, 2016.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[10] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler. Skip-thought vectors. In *NIPS*, 2015.

[11] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[12] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[13] R. Smith, C. Gu, D.-S. Lee, H. Hu, R. Unnikrishnan, J. Ibarz, S. Arnoud, and S. Lin. End-to-end interpretation of the french street name signs dataset. In *ECCV*, 2016.

[14] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[15] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. *IEEE transactions on pattern analysis and machine intelligence*, 39(4):652–663, 2017.

[16] Z. Wojna, A. Gorban, D.-S. Lee, K. Murphy, Q. Yu, Y. Li, and J. Ibarz. Attention-based extraction of structured information from street view imagery. *arXiv preprint arXiv:1704.03549*, 2017.

[17] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.