SpeeDO: Parallelizing Stochastic Gradient Descent for Deep Convolutional Neural Network

Zhongyang Zheng
HTC Research
Beijing, China
zhongayang_zheng@htc.com

Wenrui Jiang
HTC Research
Beijing, China
roy_jiang@htc.com

Gang Wu
HTC Research
Beijing, China
simon.g_wu@htc.com

Edward Y. Chang
HTC Healthcare & Research
edward_chang@htc.com

Abstract

Convolutional Neural Networks (CNNs) have achieved breakthrough results on many machine learning tasks. However, training CNNs is computationally intensive. When the size of training data is large and the depth of CNNs is high, as typically required for attaining high classification accuracy, training a model can take days and even weeks. In this work, we propose *SpeeDO* (for Open DEEP learning System in backward order). *SpeeDO* uses off-the-shelf hardwares to speed up CNN training, aiming to achieve two goals. First, parallelizing stochastic gradient descent (SGD) on a GPU cluster with off-the-shelf hardwares improves deployability and cost effectiveness. Second, such a widely deployable hardware configuration can serve as a benchmark on which software algorithmic approaches can be evaluated and improved. Our experiments compared representative SGD parallel schemes and identified bottlenecks where overhead can be further reduced.

1 Introduction

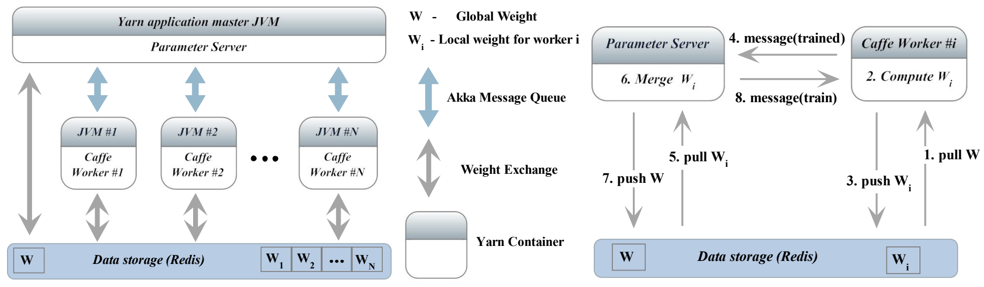
Deep learning has attracted a lot of attention in recent years, thanks to its high-accuracy achievements in classifying multimedia data using convolutional neural network (CNN). To train a model to attain high accuracy, CNN requires a large number of training data. Therefore, the training process can take weeks of time to complete. Several efforts have been devoted to speeding up CNN training, e.g., COTS systems [1, 2] of Google, Facebook [3], project Adam [4] of Microsoft, Mariana [5] of Tencent, and Deep Image [6] of Baidu. To achieve speedup via parallelization, most of these algorithms employ GPUs, while making sure IO overhead is in check with specialized hardwares such as FDR InfiniBand and GPUDirect RDMA. These training algorithms run on specialized hardwares and are limited in deployability and portability.

In this paper we introduce *SpeeDO* (Open DEEP learning System in the backward order), a deep learning system designed for off-the-shelf hardwares. *SpeeDO* can be easily deployed, scaled and maintained in a cloud environment, such as AWS EC2 cloud, Google GCE, and Microsoft Azure, since no specialized hardware is required. In addition, it can be used as a benchmark for comparing algorithm performance side-by-side. In the experiment section, we compare both the accuracy and speed of five representative parallel stochastic gradient descent (SGD) implementations. We identify their IO overheads and will address these and other bottlenecks using both algorithmic and system approaches in *SpeeDO*'s future iterations. We plan to make *SpeeDO* publicly available via Apache open source license.

2 Architecture

The main goal of *SpeeDO* is building a scalable cluster on off-the-shelf hardwares to speed up different implementations of distributed stochastic gradient descent, not limited to those described in this paper. We have stress-tested with two configurations, one with twelve machines on a Google GCE cluster, and the other with three machines each equipped with a 4-GPU cluster on AWS EC2 cloud. *SpeeDO* is expected to scale to large clusters of CPUs and GPUs.

The parameter server architecture proposed in [7] has been used for building distributed machine learning systems [8]. *SpeeDO* adopts this design as well. Figure 1 shows the architecture and data flow of *SpeeDO*, which is written in Java and Scala and runs on Java Virtual Machine (JVM).



(a) Architecture of *SpeeDO*. (b) The data flow of the ith Caffe worker.

Figure 1: Architecture and data flow of *SpeeDO*.

SpeeDO takes advantage of many existing solutions in the open-source community, including Caffe [9], Redis [10], Akka [11] and Yarn [12]. Caffe is an open source deep learning project that is actively evolving. *SpeeDO* relies on Caffe (written by C++) to perform SGD computation. In order to reduce calculation overhead, most calculation is performed in Caffe using JNI wrapper. Akka is used to handle message concurrency between the parameter server and workers. However, model weights, which are up to hundreds or thousands megabytes, are not exchanged directly using the message queue in Akka, but through Redis, a fast memory-based distributed database. In this way, scalability and concurrency of the parameter server is mostly handled by Akka and Redis. Yarn is used for resource management including creating the parameter server and worker JVMs. With the help of existing products (e.g. Cloudera CDH), it is easy to deploy an entire cluster.

The parameter server is running in a JVM on a master node randomly chosen by Yarn. Each Caffe worker runs on a separate JVM on its own. On a CPU cluster, typically only one worker is created on each machine since Caffe already supports multi-CPU parallelism. However, since Caffe does not yet support multiple GPUs, *SpeeDO* provides the flexibility to spawn multiple workers on the same machine so that we can take advantage of a multi-GPU cluster.

Figure 1(b) illustrates the data flow of *SpeeDO*. Each worker first pulls the latest weight W from Redis (step #1 in the figure), performs required computation using Caffe (step #2), pushes generated W_i back to Redis (step #3), and informs the parameter server that it has finished training (step #4). The parameter server checks whether it needs to immediately merge the delta, wait for other workers then merge together, or discard the delta depending on the employed parallel scheme of SGD described in Section 3. If immediate merging delta is required, the parameter server pulls W_i from Redis, performs merging, and then pushes updated W back to Redis (steps #5 to #7). Finally, the parameter server notifies the worker to start training the next iteration (step #8). When the stopping criteria is met, the parameter server may inform all workers to terminate.

3 Parallelizing Stochastic Gradient Descent (SGD)

Stochastic gradient descent (SGD) is a popular algorithm for training a wide range of models in machine learning. SGD is a common solution for solving deep learning model parameters. SGD is scalable to very large datasets with reasonable convergence capability [13]. In this section, we discuss five distributed implementations of SGD to speed up training, with pros and cons in terms of convergence capability and overhead cost.

3.1 Synchronous SGD

A randomly sampled mini-batch (very small compared to the entire training set) is used as training input for each iteration of SGD. An obvious strategy is to evenly divide each mini-batch into P parts [14]. Each of the P workers is responsible for training one part ($^1/_P$ mini-batches) in every SGD iteration. The parameter server then merges all gradients and updates weights.

Synchronous SGD enjoys exactly the same convergence capability as the single-machine version of SGD. However, all workers are blocked until they have received the message from the parameter server and then pull the updated weights for next iteration of SGD. Synchronous SGD has to wait for the slowest worker to complete, and this synchronization overhead degrades speedup performance.

3.2 Asynchronous SGD

In asynchronous SGD, each worker is assigned one entire mini-batch and trains completely on its own. Whenever a worker completes, the parameter server updates weights and the worker starts next iteration. Obviously, eliminating the synchronization overhead improves speedup. However, since each worker doesn't consider updates of the other workers, asynchronous SGD converges more slowly than synchronous SGD.

To further speed up the training speed, each worker can merge with the master for every S iterations instead of one. The selection of S depends on the speedup and convergence rate tradeoff.

3.3 Partially Synchronous SGD

Workers may complete training in different speeds depending on factors such as machine speed, machine load, and network capability. However, asynchronous SGD doesn't penalize a slow worker, which may be training with very old weights. This may lead to high variance of gradient updates and deteriorate convergence capability [15].

A strategy proposed by Petuum [16, 17] forces synchronization all workers if the gap among workers exceeds a limit. The workers normally follow asynchronous SGD, but a forced synchronization is triggered if the fastest worker is s iterations ahead of the slowest worker. This forced synchronization can be expensive if the number of workers is very large.

3.4 Weed-Out SGD

We propose a strategy similar to how MapReduce handles draggers called Weed-Out SGD. This scheme restarts slow workers on different machines and accepts updates from other fast workers (weeding out draggers). Weed-Out SGD can help if the problem of draggers is severe (typically when the number of workers is large the delay caused by draggers can be significant), and when the additional resource cost is acceptable.

3.5 Elastic Averaging SGD

Zhang et al. [18] proposed a strategy that converges faster than asynchronous SGD. The parameter server and the workers maintain their own copies of network weights so workers can perform more exploration while the parameter server keeps stable central weights.

Each worker employs asynchronous SGD and merges with the parameter server every S iterations, except that delta weights are merged in a different way. Instead of adding delta values to central weights directly, the parameter server calculates a panelized weight update, which is applied to both the central weights and the local worker's weights.

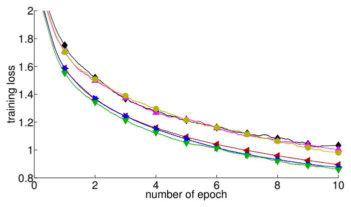
4 Experiments

Three experimental results are presented in this section: 1) validation of all SGD parallel schemes on a CPU cluster, 2) comparison of all schemes on a GPU cluster, and 3) comparison of different number of workers on a GPU cluster. The CPU experiment uses Cifar10 [19] dataset ($50,000$ 32×32

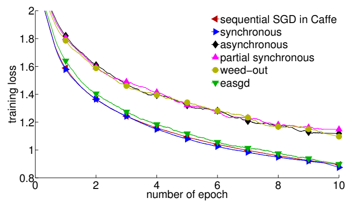
images). The GPU experiments use GoogleNet [20] model with dataset of 60, 000 224×224 images. In all experiments, the sequential SGD implementation in Caffe is used as the baseline.

4.1 Comparing Different SGD schemes on a CPU Cluster

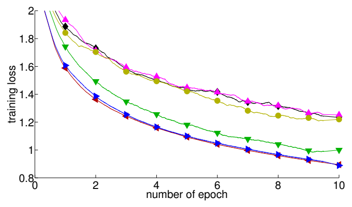
The Cifar10 dataset is used to validate all parallel implementations on a CPU cluster with four 8-core instances. The result is shown in Figure 2, where asynchronous, partially synchronous and weed-out implementations failed to scale even when the number of workers is very small.



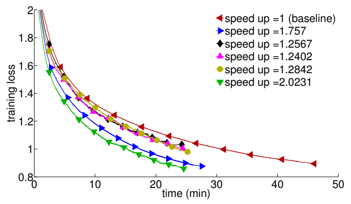
(a) Loss vs. epoch with 2 workers



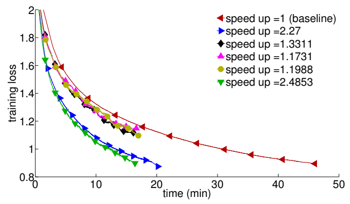
(b) Loss vs. epoch with 3 workers



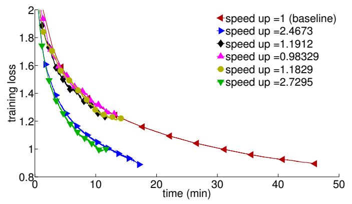
(c) Loss vs. epoch with 4 workers



(d) Loss vs. time with 2 workers



(e) Loss vs. time with 3 workers

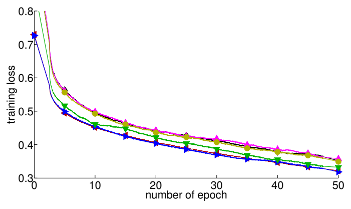


(f) Loss vs. time with 4 workers

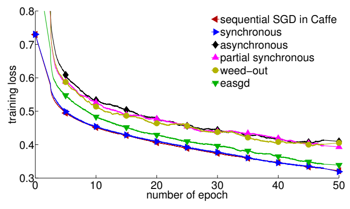
Figure 2: Validation of SGD parallel schemes on a CPU cluster.

Both synchronous SGD and EASGD share a similar convergence rate with the sequential SGD in Caffe (shown in Figure 2(a)-(c)). (Again, the sequential SGD is supposed to enjoy the best convergence rate.) This result indicates that EASGD may reduce the adversary effect of diverging gradients among workers, while still enjoying advantages in training time over synchronous SGD. EASGD enjoys the best speedup with a slight degradation in training loss. When the number of workers is 4, EASGD achieves a speedup of 2.7925 times, thanks to both parallelization and asynchronous parameter update.

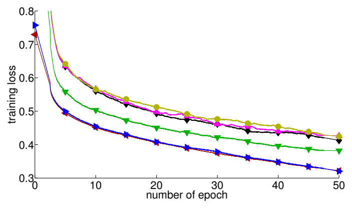
4.2 Comparing Different SGD schemes on a GPU Cluster



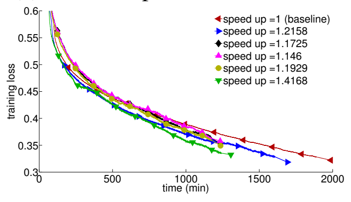
(a) Loss vs. epoch with 2 workers



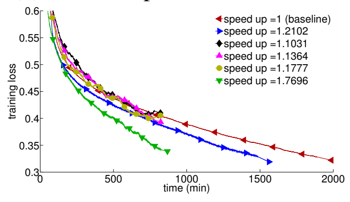
(b) Loss vs. epoch with 3 workers



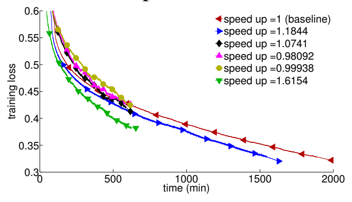
(c) Loss vs. epoch with 4 workers



(d) Loss vs. time with 2 workers



(e) Loss vs. time with 3 workers



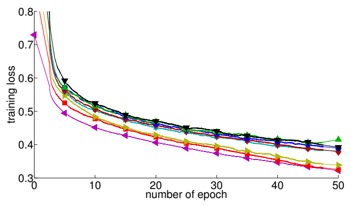
(f) Loss vs. time with 4 workers

Figure 3: Validation of SGD parallel schemes on a GPU cluster.

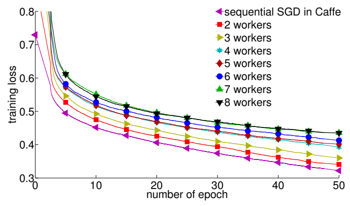
Although the speedup on the CPU cluster looks good, a single GPU can easily achieve better performance. However, since GoogleNet is far larger than Cifar10, it may take days to train even on a single GPU. Therefore, we hope that the employment of multiple-GPUs could help reduce training time. Figure 3 compares all schemes for training GoogleNet on a GPU cluster.

It can be observed that the relative speedup performance of the five schemes maintains to be similar to their relative order on the CPU cluster — EASGD achieves the best speedup. However, the best speedup is only 1.7696, when the number of worker is 3 instead of 4. This result indicates that the IO overhead dominates the total training time. In addition, the convergence rate of EASGD is worse than the sequential SGD in Caffe as we speculate that the GoogleNet model is harder to train.

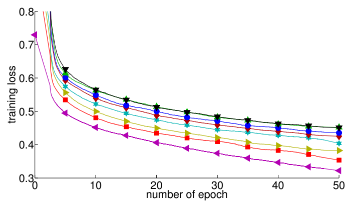
4.3 Parameter Analysis of EASGD on GPU Cluster



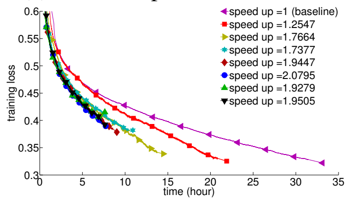
(a) Loss vs. epoch when $S = 1$.



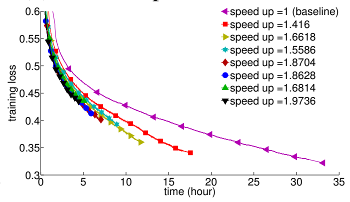
(b) Loss vs. epoch when $S = 5$.



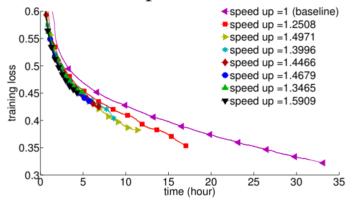
(c) Loss vs. epoch when $S = 10$.



(d) Loss vs. time when $S = 1$.



(e) Loss vs. time when $S = 5$.



(f) Loss vs. time when $S = 10$.

Figure 4: EASGD on GPU Cluster. Workers merge with parameter server every S iterations.

Figure 4 uses EASGD to show the impact of two parameters: the number of workers and the setting of parameter S. As expected, a larger S reduces the IO overhead by slightly degrading training accuracy. The speedup reported in Figure 4 is slightly different from that reported in Figure 2(d)-(f), since we calculated speedup with a different training loss (the worst loss of all schemes at the end of training was used).

The speedup performance is disappointing when the number of workers increases. The best speedup of 2.0795 is achieved by using 6 workers and $S = 1$. Since GPU is much faster than CPU, the IO overhead in both memory access and communication exacerbates, especially when the number of workers is large. With different S values, the computation time is the same while the communication overhead is reduced to ${}^{1}/_{S}$. The percentage of the communication overhead is 29% with the best configuration. According to Amdahl's law, the overhead portion of SGD caps the maximum expected improvement to the overall system.

5 Conclusion

We presented *SpeeDO*, a distributed SGD system built upon off-the-shelf hardware. Several parallel SGD schemes were evaluated and compared on both CPU and GPU clusters. In a multi-CPU environment, all schemes can achieve reasonable speedup. In a multi-GPU environment, Amdahl's law caps the room that can be achieved by employing only parallel computing.

This paper evaluated all prior schemes ([2, 5, 6]) side-by-side and demonstrates that the best achievable speedup may be highly dependent on the system configuration and size of training data. Though the sizes of training datasets used in our experiments are relatively small, we believe that a larger training set may increase both computation and IO time in similar proportions, and the conclusion of this work holds. Clearly, improving IO performance [21] via hardware and software schemes is the top priority of future research work.

References

[1] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[2] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 2013.

[3] Omry Yadan, Keith Adams, Yaniv Taigman, and MarcAurelio Ranzato. Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853*, page 17, 2013.

[4] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014.

[5] Yongqiang Zou, Xing Jin, Yi Li, Zhimao Guo, Eryu Wang, and Bin Xiao. Mariana: Tencent deep learning platform and its applications. *Proceedings of the VLDB Endowment*, 7(13):1772–1777, 2014.

[6] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 2015.

[7] Zhiyuan Liu, Yuzhou Zhang, Edward Y Chang, and Maosong Sun. Plda+: Parallel latent dirichlet allocation with data placement and pipeline processing. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):26, 2011.

[8] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598, 2014.

[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[10] Redis. `http://redis.io/`.

[11] Akka. `http://akka.io/`.

[12] Yarn: Yet another resource negotiator. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[13] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20, pages 161–168. NIPS Foundation (http://books.nips.cc), 2008.

[14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[15] Ji Liu, Stephen J Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *arXiv preprint arXiv:1311.1873*, 2013.

[16] Wei Dai, Jinliang Wei, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[17] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[18] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. *arXiv preprint arXiv:1412.6651*, 2014.

[19] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.

[20] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[21] Mohammed Sourouri, Tor Gillberg, Scott B. Baden, and Xing Cai. Effective multi-gpu communication using multiple cuda streams and threads. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 981–986, 2014.