Dogwild! — Distributed Hogwild for CPU & GPU

Cyprien Noel
Flickr Vision & Machine Learning Group
Yahoo! Inc
cypof@yahoo-inc.com

Simon Osindero
Flickr Vision & Machine Learning Group
Yahoo! Inc
osindero@yahoo-inc.com

Abstract

Deep learning has enjoyed tremendous success in recent years. Unfortunately, training large models can be very time consuming, even on GPU hardware. We describe a set of extensions to the state of the art Caffe library [3], allowing training on multiple threads and GPUs, and across multiple machines. Our focus is on architecture, implementing asynchronous SGD without increasing Caffe's complexity. We isolate parallelization from Caffe's existing SGD code, train unmodified models, and run on commodity hardware. Isolation is achieved by extending the Hogwild model, i.e. running parallel SGD solvers without synchronization, by also removing synchronization between solvers and components in charge of streaming gradients between nodes. In this modular design, components interact exclusively through unsynchronized reads and writes to the weight buffer. Each component is free to loop over the weights at a different pace, keeping both compute and network resources fully utilized. SGD's resiliency against gradient loss allows further performance improvements by avoiding reliable network protocols. It enables the use of multicast messages, and of low level packets streaming through raw sockets or InfiniBand verbs. We show linear performance scaling for small clusters on MNIST, and early results on ImageNet.

1 Introduction & Previous Work

Training a neural network for a task such as ImageNet typically takes about two weeks on a modern GPU. Parallelizing SGD to accelerate this process is challenging. Downpour [2] and a GPUs + InfiniBand implementation [1] successfully scaled to 16000 CPU cores and 64 GPUS respectively, but only when training locally connected models. Downpours performance on 1600 cores for a globally connected network is not significantly better than a single GPU [2] [8]. [8] identified a low upper bound on the expected benefits of parallelizing SGD, even in the asynchronous case.

Nevertheless, satisfactory performance can be achieved at small scale [9] [5] [7] — for instance, cuda-convnet2 [4] is a relatively task and hardware specific implementation, but scales efficiently to 8 GPUs.

2 Hogwild

In a Hogwild setting, multiple SGD processes run on the same weights using different shards of training data. Each thread computes gradients using private data and layers state, but reads and writes to a shared memory location for weights. The cache hierarchy is responsible for propagating updates between cores. Coordination happens at two points, when the weights are read during gradient computation (memory load), and when gradients are applied (unsynchronized addition: load, add, store).

Races are expected, where loads and stores from different threads interleave and additions are lost. In practice, our measurements show surprisingly low loss rates. Our experiment consisted of adding a constant value during gradient update and looking at the end value. For single CPU socket machines, no loss was measured. For two-sockets machines, where the latency between cores can be higher, the loss stayed below 0.5% for most network sizes.

3 Distributed Hogwild

Most of the distributed SGD implementations mentioned above isolate computation and exchange of gradients. Separation is achieved either through time, by stopping SGD during weight and gradient exchange, or through space, by maintaining separate memory locations and switching between them between phases. Asynchronous SGD goes further and allows weights and gradients transfers to occur concurrently with computation. Weak coordination still occurs at the beginning and end of a batch to launch transfers asynchronously.

This work takes inspiration from the Hogwild [6] approach to go further still in two directions. We increase the amount of decoupling between computation and data streaming, and also tolerate occasional gradient loss in order to use unreliable and multicast network transports. This approach has several advantages:

- A clean and modular code base, with complete separation of the training and synchronization code. We modified Caffe[3] to allocate all weights inline in a single buffer that can easily be shared between components. Building a training architecture amounts to starting a collection of SGD solvers, and a set of synchronization segments, e.g. CPU-GPU, CPU-Ethernet, GPU-InfiniBand. A component performs a single task, and can optionally run in its own process if the weights are mapped to `/dev/shm`.

- Better resource utilization, as both SGD and synchronization run continuously and at the maximum rate allowed by the computing power and network bandwidth available. SGD seems relatively resilient to unbalance between the two. Removing all synchronization, and relying instead on the highly tuned modern memory hierarchies allows individual tasks to perform faster, and might be the way to the highest possible performance.

- Very high speed networking. Unreliable transports like raw socket/packet_mmap and InfiniBand UD have very low compute and memory overhead. They can use more of the available bandwidth, and display low and regular latency characteristics. Packets can be sent through multicast, allowing higher performance and potentially scalability, within the constraints of the SGD algorithm.

4 Architecture

The current model only supports data-parallelism. Model weights are replicated on all nodes, i.e. hosts or GPUs in the cluster. The mechanism is very similar for both types of nodes. For each segment, one of the ends is designated master, either for the whole buffer, or for a given range. The master loops over the weight buffer, sending a set of weights at a time to all slaves. Each slave compares its own weights to a copy. The copy did not change since the last message from the master, so the difference is the gradient since the last synchronization. The slave updates its weights using the master position plus the gradient, and sends the gradient to the master.

This mechanism allows nodes to apply each others gradients and return their own in one pass over the memory. Memory is scanned in order on all nodes, which helps locality. Multicast messages can be used on the master to keep all slave copies synchronized in one message.

According to our experiments on Hogwild, the amount of work lost due to races during concurrent additions is negligible. In almost all cases, each sides gradient will be added to the other without loss. The master is continuously sending the reference position, which keeps slaves within a bounded distance.

5 Implementation

One of this projects goals was to run on commodity hardware. Most data center machines are equipped with Ethernet, often a single adapter. Streaming data at the highest speed that hardware can deliver gives rise to a surprising number of practical difficulties, both on the networking side and for internal communication with GPUs.

6 Networking

Our first satisfying results were achieved using raw sockets, by exchanging Ethernet packets with an Ethertype different from IP. This traffic avoids filtering and other processing that regular traffic like UDP is subject to. In our experiments, UDP streaming at maximum speed congests the network stack, and renders machines unreachable. TCP solutions oscillate at bandwidths well below the hardware limits, and are subject to latency spikes.

Reliable transports and APIs like TCP, RDMA, or MPI are subject to latency spikes on Ethernet caused by retry mechanisms. Packet ordering causes head of line blocking, and is not necessary for distributed SGD. Our experiments showed up to 10% packet loss, but this did not seem to incur any measurable effect on SGD speed or stability. Attempts to correct the situation by throttling, or switching to reliable transports only lowered the convergence rate.

Another valuable optimization enabled by unreliable messaging is the use of multicast or broadcast messages. It allows the master to iterate faster over its weights by sending each chunk only once for all slaves, reducing the distance between weights copies, them and improving SGD efficiency.

We have not been able to benchmark reliable messaging on InfiniBand and Converged Ethernet. They offer reliable communication, removing the need for retries on the software side. It is our understanding that reliable multicast is not provided, and that packet loss is avoided by pessimistically allocating safe bandwidth limits. It is therefore unlikely to help our use case.

Our Ethernet implementation uses a Linux feature called packet_mmap, and on InfiniBand, ibverbs. Both APIs enable user-space networking, i.e., allow the network adapter to directly send and receive blocks of memory mapped to the application address space, avoiding system calls and memory copies.

7 Performance

Results on MNIST are encouraging, and show linear scaling with the number of solvers. We used two clusters, 6 machines with 2 Xeons and 2 K-20 each over 1G Ethernet, and another with two machines with 2 Xeons, 8 GPUs and 10G Ethernet. Learning curves as a function of wall-time are shown in Figure 1

Validating the approach on large models like ImageNet will take more work. We are running into various bottlenecks, including simply feeding the training data at a rate that can keep the GPUs running. Software stability prevented us from running long jobs in time for this submission, but we are confident that those issues can be resolved. Early results seem to show the same scalability pattern at MNIST, as can be seen in Figure 2.

8 Future Directions

This work is still in its preliminary stages, and there are a number of ongoing directions of expansion in terms of our capabilities as well as additional and more rigorous evaluations that we plan to make. We outline some of these in the following subsections (and anticipate being able to present extended results at the workshop). Furthermore, once we have reached a sufficient level of maturity, our Dogwild code will be made available as part of the Caffe open-source package.

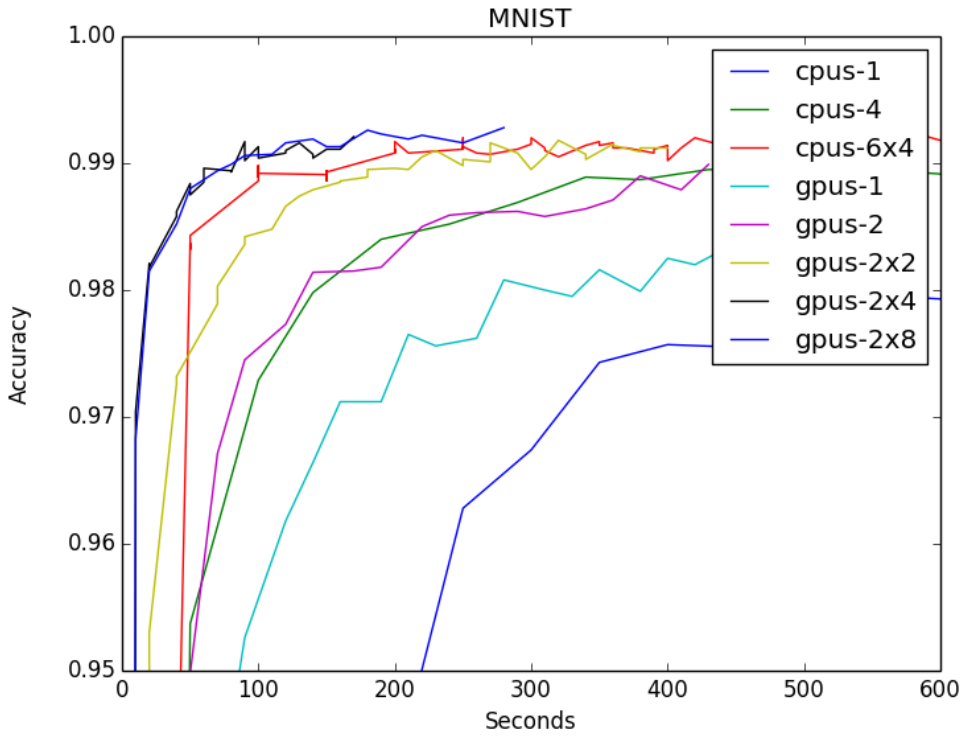

Figure 1: MNIST results. We show validation set performance as a function of wall-time.

8.1 Data Feeding

Caffe's current data feeding implementation does not deliver sufficient bandwidth to keep 4 GPUs fully active on ImageNet, at least on our hardware. CUDA features like asynchronous streams, and additional pre-fetching and caching need to be experimented.

8.2 Benchmarking

This approach needs to be evaluated against synchronous techniques. We need to understand the impact of unbalance between computation and networking that is allowed by their decoupling.

8.3 Momentum

Each solver currently allocates its own momentum buffer. We suspect momentum should be approached differently, for both performance and SGD stability reasons. On a large number of CPU cores, a single momentum buffer could be shared by all solvers. In a distributed setting, the master could be in charge of adding the momentum before broadcasting each new reference value.

8.3.1 Learning rate

Our approach for rate scheduling has been to sum iterations over all solvers, and considering this as the current training progress. Synchronization is not equivalent to more iterations from a single solver, so a better estimate needs to be investigated.

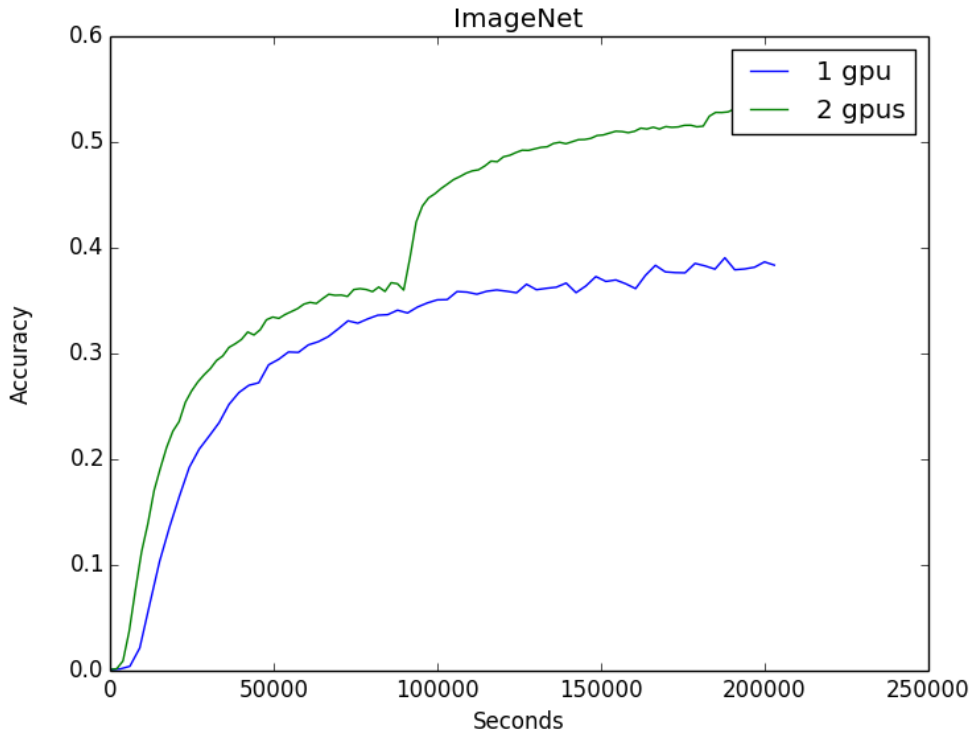

Figure 2: ImageNet results. We show validation set performance as a function of wall-time. Note the "notch" in the 2-GPUs curve at approximately 100,000 seconds is due to the learning rate scheduler, which decreases the learning rate by a factor of 10 at this point. It would occur on 1-GPU curve if we trained longer.

8.3.2 Specialization

In a distributed setting, it might make sense for each node to perform a different computation. One node could for example be running a second-order training algorithm to evaluate a learning rate, that could in turn be used by other nodes to run regular SGD.

Acknowledgments

The authors gratefully acknowledge the support from the Vision & Machine Learning, and Production Engineering teams at Flickr (in alphabetical order: Andrew Stadlen, Arel Cordero, Clayton Mellina, Frank Liu, Gerry Pesavento, Huy Nguyen, Jack Culpepper, John Ko, Mehdi Mirza, Pierre Garrigues, Rob Hess, Stacey Svetlichnaya, Tobi Baumgartner, and Ye Lu). We are also grateful for the support of our collaborators at NVIDIA (Douglas Holt, Jonathan Cohen, and Mike Houston).

References

[1] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1337–1345, 2013.

[2] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[3] Y. Jia. Caffe: An open source convolutional architecture for fast feature embedding. `http://caffe.berkeleyvision.org/`, 2013.

[4] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[5] Y. Miao, H. Zhang, and F. Metze. Distributed learning of multilingual dnn feature extractors using gpus. *Proc. Interspeech*, 2014.

[6] F. Niu, B. Recht, C. R, and S. J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *In NIPS*, 2011.

[7] T. Paine, H. Jin, J. Yang, Z. Lin, and T. S. Huang. GPU asynchronous stochastic gradient descent to speed up neural network training. *CoRR*, abs/1312.6186, 2013.

[8] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech dnns. In *ICASSP*. IEEE SPS, May 2014.

[9] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *ICASSP'13*, pages 6660–6663, 2013.