Artem Abzaliev
abzaliev@umich.edu

Saisamrit Surbehera
saisam@umich.edu

1 Summary of ”PipeDream: Generalized Pipeline Parallelism for DNN Training”

1.1 Problem and Motivation

Training of Deep Neural Networks (DNN) can be time-consuming, but the time can be reduced by utilizing efficient parallelization schemes. However existing parallelization approaches such as data parallelism and model parallelism exhibit high communication overhead, which can be up to 90% of total training time.

1.2 Hypothesis

The authors proposed PipeDream, a system that uses a new approach *pipeline parallelism*. It substantially cuts communications costs and gives a significant speedup compared to existing parallelism techniques. To understand how it works we have to consider two existing parallelization approaches in the literature.

One of the most popular ways to parallel DNN training is intra-batch parallelization, where one training iteration is split across a set of workers. One example of intra-batch parallelization is model parallelism, where each worker updates a subset of the models' parameters for all inputs. Figure 1 shows model parallel training with 4 workers, where numbers represent the batch id and back-propagation takes twice as long as a forward pass. Such a setting would allow for the training of huge models. However, it can be seen that the workers are under-utilized. Another drawback is that partitioning has to be done programmatically.

Inter-batch parallelization is another approach, that uses the idea of *pipelining*, where computations are run consecutively on one worker, while other workers start computations as soon as the output of the previous worker is available. Figure 2 illustrates inter-batch parallelism. While the re-

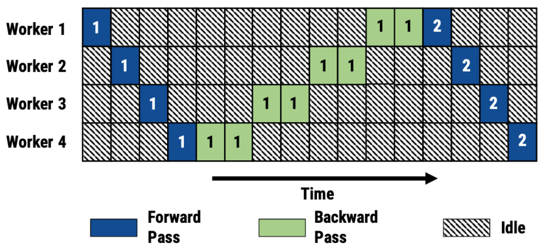

Figure 1: An example of intra-batch parallelsm

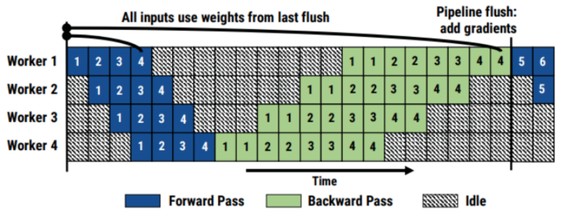
All inputs use weights from last flush
Pipeline flush: add gradients
Worker 1
Worker 2
Worker 3
Worker 4
Time
Forward Pass
Backward Pass
Idle

Figure 2: An example of inter-batch parallelism

sources are utilized better, there are still some idle workers.

Authors build upon the idea of model parallelization, combining it with inter-batch parallelization. PipeDream divides the model among available workers, assigning a group of layers to each of them, and then overlaps the computation and communication of different inputs in a pipelined fashion while minimizing the amount of data communicated.

1.3 Solution Overview

Pipeline-parallel computation partitions a DNN model into multiple stages, where each stage consists of a consecutive set of layers in the model. Each stage is mapped to a separate worker that performs the forward and backward pass for all layers in that stage. To keep all workers utilized, the authors propose to have multiple minibatches at the same time in the system. With one single minibatch, proposed approach would be equivalent to model parallelization.

On completing its forward pass for a particular minibatch, each stage asynchronously sends the output activations to the next stage, while simultaneously starting to process another minibatch. The last stage starts the backward pass on a minibatch immediately after the forward pass completes. On completing its backward pass, each stage asynchronously sends the gradient to the previous stage while starting computation for the next minibatch. Figure 3 illustrates the process.

Pipeline parallelization is faster than other approaches for two main reasons. First, there is much less communication time, because each worker communicates with only a single worker at a time. Pipeline parallelization communication is peer-to-peer, as opposed to all-to-all in the existing approaches. Second, Pipeline parallelization makes computation and communication to be independent of each other and hence leads to easier parallelization.

The authors describe three challenges of PipeDream and describe their proposed solutions:

- Work partitioning. To make a balanced pipeline, PipeDream partitions DNN layers into stages such that each stage takes approximately the same time to compute. To measure the time, the authors use a profiling run of 1000 minibatces on a single GPU.

- Work scheduling. At any point in time, a single worker needs to determine whether to perform a forward or backward pass. The authors proposed a simple algorithm - one-forward-one-backward (1F1B). In 1F1B's steady state worker just alternates between performing its forward pass and backward pass. Despite the simplicity, it is very efficient.

- Effective learning. To compute backward pass, the weights should be the same that were used during the forward pass. However, PipeDream assigns several forward passes for a worker until the 'original' backward pass comes to utilize the resources fully. Those forward passes update the weights. For instance, Figure 3 shows that for worker 1 and batch 1, there are forward passes for batch 2, 3, 4 until there is a backward pass for batch 1. To address this problem, PipeDream proposes a *weight stashing* algorithm. Weight stashing represents a versioning system for model weights, one version for each active minibatch in a system. For each forward pass of a batch, a worker keeps the weights in memory and uses them when the corresponding backward pass comes.

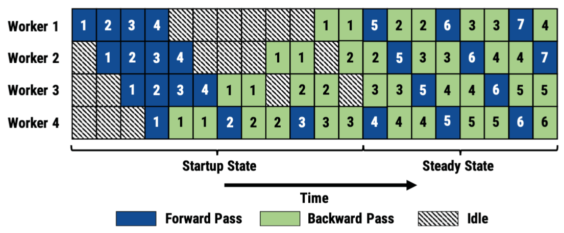

Figure 3: An example of pipeline parallelism

Authors make extensive experiments across a wide range of learning tasks and hardware setups. They show that PipeDream provides significant speedups while maintaining high accuracy.

1.4 Limitations and Possible Improvements

The consequence of weight stashing is an increase in memory usage, although peak per-worker memory footprint is similar to the one by data parallelism. The authors don't describe possible improvements.

1.5 Summary of Class Discussion

Comment: PipeDream looks very similar to software pipelining used in compilers, or instruction pipelines in CPU.

Question: When are the parameters at each layer updated in that figure? After each backward pass? Answer: Yes, after the backward pass. However, to implement backward pass properly, the authors also proposed weights stashing mechanism.

Question: So the workers are repeating computation? Or when a mini batch is passed, it continues where the last worker finished? Answer: The workers are not repeating the computation. After a mini batch is passed, the computation flows from one workers to another, where each worker is only doing forward or backward pass for this mini batch.

Question: Are the weights stored in GPU memory or in main memory? Answer: In GPU memory. It does not significantly increase the per-worker memory, because in worst-case it is just equal to data parallelism.

Question: am curious how stragglers will effect the performance of the pipeline, like a worker running very slow. I think in data parallelism, people can train model asynchronously, but in the pipeline structure, one worker needs to wait for the outcome of the previous worker. Answer: The authors did not consider this aspect.

2 A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters

2.1 Problem and Motivation

Data centers which run Deep Neural Networks (DNNs) are heterogeneous. Most of the GPU clusters simultaneously run numerous jobs, many of which do not heavily use CPUs or network bandwidth. Current, existing distributed DNN's training architectures, all-reduce and Parameter Server (PS), cannot fully utilize such heterogeneous resources. This leaves a lot of computational resources on the table and if utilized properly can speed up training of Models.

2.2 Hypothesis

DNN Models optionally run distributed training. The most popular distributed DNN training approach is data parallelism. It partitions the dataset to multiple distributed computing devices mostly each GPU holds the complete DNN model. For distributed training, there are two families of data parallelism approaches: all-reduce and Parameter Server (PS).

All-reduce utilizes a ring based structure to enable parallel communication. Where each node serves as a starter of a ring and end of another ring. The whole process can be broken down into two steps: scatter and all gather. If n is the number of GPU machines. k is the number of additional CPU machines. M is the model size. B is the network bandwidth. In Reduce-scatter, each nodes sends $\frac{n-1}{m}$ bytes and in all Gather each nodes sends $\frac{(n-1)M}{m}$ bytes. This means that the total communication time is $\frac{2(n-1)M}{nB}$. In this cases the optimal case is when k = 0.

Parameter servers on the other hand consists of two types of nodes:servers and workers. Workers usually run on GPU machines and the Parameter servers usually run on CPU's. There are two placement strategies for PS: non-colocated mode and collected mode. In the Non- colocated modes, server is on CPU machines and the model is partitioned to k parts on k different CPUs machines. Each GPU worker must send M bytes gradients and receives M bytes parameters back. Each CPU machine must receive in total $\frac{nM}{k}$ gradients from the GPU workers and send back $\frac{nM}{k}$ parameters. Each GPU worker sends and receives M bytes. In this case the communication time is $\max(\frac{M}{B}, \frac{nM}{kB})$. In the co-located Parameter server model, the PS process is on every GPU worker and reuses its spare CPU resources. The communication time is the same as All-reduce.

Table 1: The theoretical communication time required by each training iteration. n is the number of GPU machines. k is the number of additional CPU machines. M is the model size. B is the network bandwidth. We will revisit the *Optimal?* row in §4.1.

	All-reduce	Non-Colocated PS	Colocated PS
Time	$\frac{2(n-1)M}{nB}$	$\max(\frac{M}{B}, \frac{nM}{kB})$	$\frac{2(n-1)M}{nB}$
Optimal?	Only if $k = 0$	Only if $k = n$	Only if $k = 0$

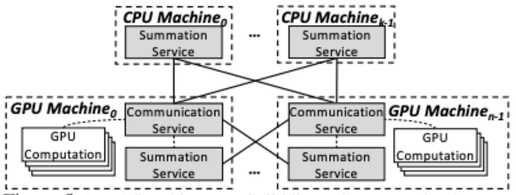

Figure 5: BytePS architecture. Solid lines: the connection between CPU machines and GPU machines. Dashed lines: the data flow inside GPU machines.

From Table 1, we can see that spare CPUs and bandwidth are not fully utlized in most cases on production GPU clusters. This leads us the authors to create a more dynamic methods to allocation bandwidth/compute to all the devices. Making such changes can speed up training of Models and reduce costs.

2.3 Solution Overview

The authors in this paper present a new distributed DNN training architecture called BytePS. BytePS architecture, as explained in figure 5, consists of two modules: Communication services(CS) and Summation Services (SS). CS are responsible for deciding the traffic volume to each SS (both internal and external). It is also responsible for internally synchronizing the tensors among multiple local GPUs and externally communicating with SS. The load assignment strategy is based on our analysis of the optimal communication strategy. SS is only responsible for receiving tensors that are sent by CS, summing up the tensors and sending them back to CS. CPU machines have the summa-

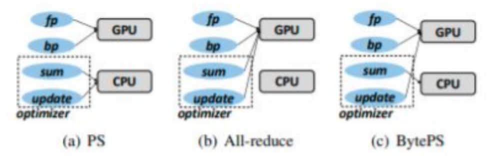

Figure 10: Component placement comparison between all-reduce, PS and BytePS.

tion service only and the GPU machines have the communication service and summation service.

This architecture enables BytePS to flexibly utilize any number of additional CPU resources and network bandwidth. When the number of CPU machines is 0, i.e., k = 0, the communication will fallback to only using SSs on GPU machines. When the number of CPU machines is the same as GPU machines, BytePS is as communication optimal as non-colocated. In other cases, BytePS can leverage SSs on all machines together. These improvements make BytePS always communication optimal with any additional CPU and bandwidth resources. For BytePS to achieve the optimal communication, it requires wise SS workload assignment among CPU machines and GPU machines. Usually the optimal workload assignment is Model M for SS on CPU ($M_{SS_{CPU}}$) is $\frac{2(n-1)}{n^2+kn-2k}M$ and SS on GPU $M_{SS_{GPU}}$ is $\frac{n-k}{n^2+kn-2k}M$.

BytePS also gets the optimal inter-machine communication time by reducing the CPU bottleneck as it cannot match increasing network bandwidth. This means that the CPU were only performing summation and the GPU were performing updates.

2.4 Limitations and Possible Improvements

- BytePS is that optimal intra-machine communication strategy is tightly coupled with the internal topology. If BytePS, could detect the topology, probe the bandwidth, and generate the best strategy, it would further reduce training time.

- BytePS is the number of CPU machines has to be less than the number of GPU machines. This means that in cluster with a larger share of CPU machines, BytePS will not be as efficient as possible

- Modern clusters have dynamic CPU resources. Since BytePS can adapt to any number of CPU machines, it enables elasticity of service.

2.5 Summary of Class Discussion

We had a very small discussion in the class for the paper. It was only limited two questions

Q : Do you have unlimited number of CPU machines?

A : The number of the CPU machines is mostly less than GPU machines. In a case where the CPU machines is more than GPU machines, this frame work might not work.

Q: How does BytePS solve the CPU bottleneck problem ?

One solution in this paper is that they keep summation only on CPU while update is moved to GPU. They want to leverage the use use of spares CPU