Summary of "PipeDream: Generalized Pipeline Parallelism for DNN Training"

Adam Stautberg (astaut) and Mihir Patel (patilmi)

Problem and Motivation

The main motivation behind this paper was finding a way to improve the speed of DNN(Deep Neural Networks) training. DNNs have facilitated tremendous progress across a applications such as image classification, translation, language modeling, and video captioning. DNN training is extremely time-consuming, and PipeDream is able to dramatically increase speed up to 5.3x while maintaining high accuracy.
To understand how PipeDream works we must first understand how intra-batch and inter-batch parallelism work. Intra-batch parallelism is where a single iteration of training is split across available workers. The weights are periodically synchronized and the amount of data communicated is proportional to the number of model weights and the number of workers. Inter-batch parallelism is where multiple stages are run consecutively on one worker while the earlier stages are finishing up on other workers. This is best illustrated in the figure below:

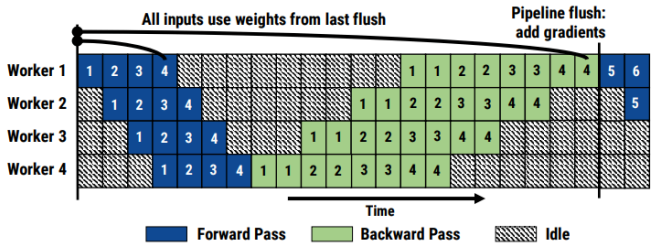

Figure 3: GPipe's inter-batch parallelism approach. Frequent pipeline flushes lead to increased idle time.

Another downside of using simple data parallelism is that programmers often have to determine for themselves how to partition data, often resulting in under-utilization of resources even when using parallelism techniques. PipeDream partitions and schedules the work for the user automatically.

Hypothesis

PipeDream combines inter-batch pipelining with intra-batch parallelism in order to increase utilization of available hardware to increase speed. PipeDream also partitions and schedules work for the user to ensure proper utilization of hardware and increase speed.

Solution Overview

Combining inter-batch pipelining with intra-batch parallelism is best illustrated in the following diagram:

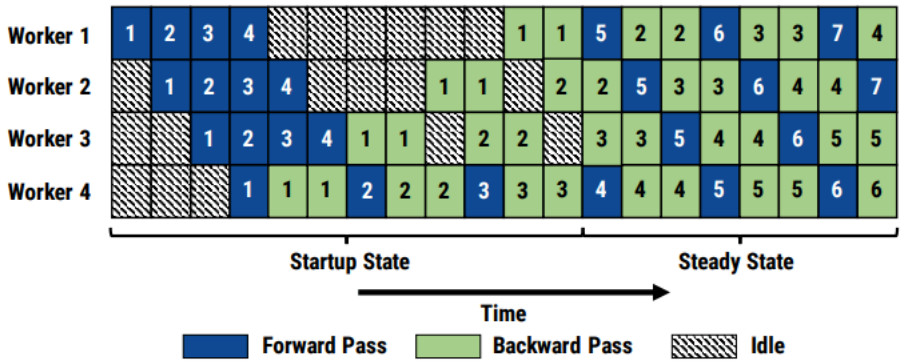

Figure 4: An example PipeDream pipeline with 4 workers, showing startup and steady states. In this example, the backward pass takes twice as long as the forward pass.

As we can see, the first worker immediately begins working on the second minibatch after processing the forward pass of the first minibatch. This pattern continues until it reaches the steady state where all workers are utilized working on either a forward or backward pass of a minibatch. Intuitively we can see how this would increase resource utilization and speed. We now must address the problems of maintaining accuracy and scheduling work.

In order to execute this pipeline while maintaining accuracy, PipeDream stores intermediate weights using a technique called weight stashing. As demonstrated in the figure below, PipeDream stores the latest weights when executing a forward pass and then uses those weights when executing a backwards pass. This ensures minimum staleness and allows the model to execute with a high degree of accuracy.

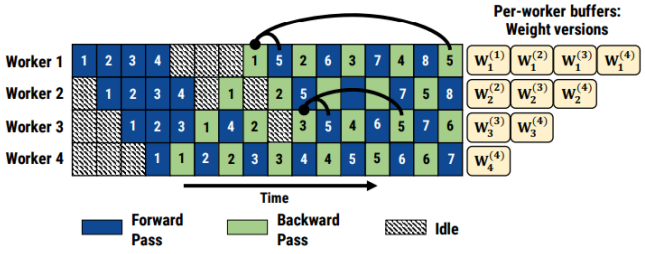

Figure 9: Weight stashing as minibatch 5 flows across stages. Arrows point to weight versions used for forward and backward passes for minibatch 5 at the first and third stages.

In order to ensure high utilization of resources, PipeDream uses dynamic programming to figure out how to best partition DNN layers into stages such that each stage completes at roughly the same rate. This allows the pipeline to achieve maximum throughput.

Overall, PipeDream provides a faster DNN training method that effectively combines inter-batch pipelining with intra-batch parallelism and efficiently partitions and schedules work for the user. The results can be seen below:

Task	Model	Dataset	Accuracy Threshold	# Servers × # GPUs per server (Cluster)	PipeDream Config	Speedup over DP	
						Epoch time	TTA
Image Classification	VGG-16 [48]	ImageNet [44]	68% top-1	4x4 (A)	15-1	5.28×	5.28×
				2x8 (B)	15-1	2.98×	2.46×
	ResNet-50 [26]	ImageNet [44]	75.9% top-1	4x4 (A)	16	1×	1×
				2x8 (B)	16	1×	1×
	AlexNet [37]	Synthetic Data	N/A	4x4 (A)	15-1	4.92×	N/A
				2x8 (B)	15-1	2.04×	N/A
Translation	GNMT-16 [55]	WMT16 EN-De	21.8 BLEU	1x4 (A)	Straight	1.46×	2.2×
				4x4 (A)	Straight	2.34×	2.92×
				2x8 (B)	Straight	3.14×	3.14×
	GNMT-8 [55]	WMT16 EN-De	21.8 BLEU	1x4 (A)	Straight	1.5×	1.5×
				3x4 (A)	Straight	2.95×	2.95×
				2x8 (B)	16	1×	1×
Language Model	AWD LM [40]	Penn Treebank [41]	98 perplexity	1x4 (A)	Straight	4.25×	4.25×
Video Captioning	S2VT [54]	MSVD [11]	0.294 METEOR	4x1 (C)	2-1-1	3.01×	3.01×

Table 1: Summary of results comparing PipeDream with data parallelism (DP) when training models to advertised final accuracy. A PipeDream config of "2-1-1" means the model is split into three stages with the first stage replicated across 2 workers, and a "straight" configuration is a pipeline with no replicated stages—e.g., "1-1-1-1" on 4 workers. Batch sizes used to train these models are reported in § 5.1.

Limitations and Possible Improvements

Pipedream in the worst case scenario simply runs data parallelism, so there really aren't any clear downsides. It is limited in that in such cases PipeDream does not offer any speedup when compared to data parallelism. The paper does not really touch on possible improvements, though in class we did talk briefly about how they could look into more efficient ways of handling failures.

Summary of Class Discussion

Q: How does it handle failures?
A: They use checkpointing, it is not clear that they have looked into other methods. There was some debate about whether or not this was a good method, but no agreement was reached.

Q: Do they talk about why some things don't get faster whereas some got 5.3x faster?
A: In class we said no. I'm including a short paragraph about one of the image classification tasks that saw no speed up as an answer:

> “For training ResNet-50 on Cluster-A, PipeDream’s partitioning algorithm recommends data parallelism as the optimal configuration (no pipelining or model parallelism). Later, in § 5.5, we show the reason for this recommendation: non data-parallel configurations incur higher communication overheads than DP for ResNet-50, since ResNet-50 is composed of convolutional layers which have compact weight representations but large output activations. “

Summary of “SUPPORTING VERY LARGE MODELS USING AUTOMATIC DATAFLOW GRAPH PARTIONING”

Problem and Motivation

Size of a DNN is limited by hardware memory space. DNN sizes are doubling every two or three years while GPU/TPU memory isn’t. Training such large networks isn’t possible without a model parallelization scheme. Existing schemes for model parallelization can be too slow, unwieldy or not general enough.

Hypothesis

Rather than partitioning tensors according to the layer they correspond to, the authors suggest that partitioning according to operators (from a dataflow framework) is more powerful. Once operators are described in TDL (Halide inspired language) then a search algorithm can be run on the required operators to build a data flow graph capable of handling large DNNs

Solution Overview

Basic structure of Tofu:
Group forward-backwards and elementwise operators together to shrink the search space which the authors refer to as “coarsening”. Recursively apply a Dynamic Programming search algorithm by partitioning each tensor with a single split every recursion. The optimization goal here is to minimize worker communication costs in order to minimize memory footprint.
Optimizations:
-User describes operators in TDL and then Tofu performs “TDL analysis” using symbolic interval analysis
-Grouping/coalescing operators before partitioning

-Recursive partitioning
-Using/maintaining original operator dependencies so that the original memory planner can re-use the buffer of a dependent operator.
Key Claims/Insights:
-Recursion dramatically cuts down search time
-Partitions are commutative

Limitations and Possible Improvements

-Requires users to describe operators
-Operator descriptions are not verified by Tofu
-Only supports parallelization via partition-n-reduce
-Some operators cannot be described in TDL
-For smaller models, Tofu's approach may be no better or even worse than data parallelism approaches.
-Only optimizes for worker communication

Summary of Class Discussion

There was not a ton of class discussion on this article since most was on the previous topic. There was some discussion during the presentation about how it was NP hard to partition dataflow graphs, but we didn't revisit the topic. The conclusion was partly that this paper was less principled and didn't have as clear results