PipeDream: Generalized Pipeline Parallelism for DNN Training

Deepak Narayanan[‡*], Aaron Harlap[†*], Amar Phanishayee[⋆],
Vivek Seshadri[⋆], Nikhil R. Devanur[⋆], Gregory R. Ganger[†], Phillip B. Gibbons[†], Matei Zaharia[‡]
[⋆]*Microsoft Research* [†]*Carnegie Mellon University* [‡]*Stanford University*

ABSTRACT

DNN training is extremely time-consuming, necessitating efficient multi-accelerator parallelization. Current approaches to parallelizing training primarily use intra-batch parallelization, where a single iteration of training is split over the available workers, but suffer from diminishing returns at higher worker counts. We present PipeDream, a system that adds *inter-batch pipelining* to intra-batch parallelism to further improve parallel training throughput, helping to better overlap computation with communication and reduce the amount of communication when possible. Unlike traditional pipelining, DNN training is bi-directional, where a forward pass through the computation graph is followed by a backward pass that uses state and intermediate data computed during the forward pass. Naïve pipelining can thus result in mismatches in state versions used in the forward and backward passes, or excessive pipeline flushes and lower hardware efficiency. To address these challenges, PipeDream versions model parameters for numerically correct gradient computations, and schedules forward and backward passes of different minibatches concurrently on different workers with minimal pipeline stalls. PipeDream also automatically partitions DNN layers among workers to balance work and minimize communication. Extensive experimentation with a range of DNN tasks, models, and hardware configurations shows that PipeDream trains models to high accuracy up to 5.3× faster than commonly used intra-batch parallelism techniques.

1 INTRODUCTION

Deep Neural Networks (DNNs) have facilitated tremendous progress across a range of applications, including image classification [26, 37, 48], translation [55], language modeling [40], and video captioning [54]. As DNNs have become more widely deployed, they have also become more computationally expensive to train, thus requiring parallel execution across multiple accelerators (e.g., GPUs).

DNN training proceeds in iterations of forward and backward pass computations. In each iteration, the training loop processes a minibatch of input data and performs an update to the model parameters. Current approaches focus on parallelizing each iteration of the optimization algorithm across a set of workers. For example, data parallelism partitions the input data across workers [37], model parallelism partitions operators across workers [16, 21], and hybrid schemes partition both [33, 34, 36]. Unfortunately, *intra-batch* parallelization can suffer from high communication costs at large scale. For example, Figure 1 shows the communication overhead for data parallelism across five different DNN models on three different types of multi-GPU servers. Over 32 GPUs, the communication overhead for some models, computed as the percentage of total time spent on communication stalls, is as high as 90% due to expensive cross-server `all_reduce` communication. Communication overheads are high even on servers where GPUs within the server are connected by dedicated interconnects like `NVLink` [4]. Moreover, rapid increases in GPU compute capacity over time will further shift the bottleneck of training towards communication for all models.

In this paper, we propose PipeDream, a system that uses *pipeline parallelism* to enable faster DNN training by combining intra-batch parallelism with *inter-batch* parallelization. PipeDream divides the model among available workers, assigning a group of consecutive operators (called layers in DNN terminology) in the operator graph to each of them, and then overlaps the computation and communication of *different inputs* in a pipelined fashion. This process can greatly reduce inter-worker communication because it limits the communication to layer inputs and outputs (activations in the forward pass and gradients in the backward pass) solely across consecutive layers assigned to different workers, which for many models are much smaller than the size of the entire model. Moreover, this communication is peer-to-peer, as opposed to all-to-all.

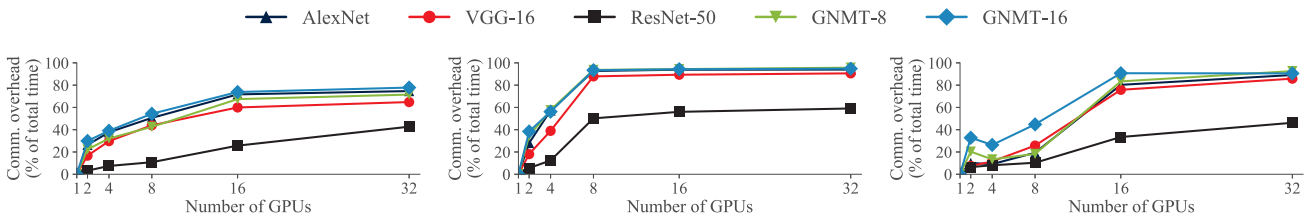
While pipelining is a simple idea, DNN training poses an important challenge not present in traditional pipelining: DNN training is *bi-directional*—the forward pass is followed by a backward pass through the same layers in reverse order, using state and intermediate results from the forward pass. To keep the pipeline full and thus achieve high hardware efficiency, a naïve scheduling mechanism might inject all minibatches in an epoch into the pipeline, first completing forward passes for all input minibatches followed by backward passes. However, this approach suffers from low statistical efficiency [18], increasing the number of passes through the dataset needed to produce a high-quality model. Furthermore, this strategy could prevent the model from reaching the desired target accuracy, since gradients are averaged over all training samples [10, 39]. To improve statistical efficiency, one could inject only a subset of m minibatches into the pipeline, and apply weight updates every m minibatches, as recently proposed by GPipe [28]. However, this reduces hardware efficiency due to more frequent pipeline flushes. Traditional model parallel training corresponds to an extreme case of this ($m = 1$).

PipeDream takes a more nuanced approach to pipelining that outperforms other solutions – it achieves high hardware efficiency with no pipeline stalls in steady state, and high statistical efficiency

[*]Work started as part of MSR internship. Equal contribution.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359646



(a) Instances with 8 1080Tis (private cluster). (b) Instances with 4 V100s (Azure). (c) Instances with 8 V100s and `NVLink` (EC2).

Figure 1: Communication overhead of data-parallel training using different multi-GPU server instances using PyTorch 1.1, NCCL [3], and `fp32` precision. We use the largest per-GPU minibatch size that fits in GPU memory, and keep the per-GPU minibatch size constant as the number of GPUs are scaled up (weak scaling).

comparable to data parallelism using the same number of workers. Given a pipeline of groups of consecutive layers executed on different workers (called a stage), PipeDream uses a scheduling algorithm called 1F1B to keep hardware well utilized while achieving semantics similar to data parallelism. In 1F1B's steady state, each worker strictly alternates between forward and backward passes for its stage, ensuring high resource utilization (negligible pipeline stalls, no pipeline flushes) even in the common case where the backward pass takes longer than the forward pass. 1F1B also uses *different versions* of model weights to maintain statistical efficiency comparable to data parallelism. Each backward pass in a stage results in weight updates; the next forward pass uses the latest version of weights available, and "stashes" a copy of these weights to use during the corresponding backward pass. Although the forward pass will not see updates from incomplete in-flight mini-batches, learning is still effective because model weights change relatively slowly and bounded staleness has been found effective in improving training speeds [19, 43]. However, for the backward pass to compute *numerically correct* gradients, the same weight version used during the forward pass must be used. PipeDream limits the number of "in-pipeline" minibatches to the minimum needed to keep the pipeline full, reducing memory overhead.

Operating the pipeline at peak throughput also requires that all stages in the pipeline take roughly the same amount of time, since the throughput of a pipeline is bottlenecked by the slowest stage. PipeDream *automatically* determines how to partition the operators of the DNN based on a short profiling run performed on a single GPU, balancing computational load among the different stages while minimizing communication for the target platform. PipeDream effectively load balances even in the presence of model diversity (computation and communication) and platform diversity (interconnect topologies and hierarchical bandwidths). As DNNs do not always divide evenly among available workers, PipeDream may decide to use data parallelism for some stages—multiple workers can be assigned to a given stage, processing different minibatches in parallel. Note that vanilla data parallelism corresponds to the pipeline having a single replicated stage. PipeDream extends 1F1B to incorporate round-robin scheduling across data-parallel stages, while making sure that gradients in a backward pass are routed to the corresponding worker from the forward pass since the same weight version and intermediate outputs need to be used for a correct gradient computation. The combined scheduling algorithm, 1F1B-RR, produces a static schedule of operators that each worker runs repeatedly, keeping utilization high across all workers. Thus, pipeline-parallel training can be thought of as a principled combination of inter-batch pipelining with intra-batch parallelism.

Our evaluation, encompassing many combinations of DNN models, datasets, and hardware configurations, confirms the training time benefits of PipeDream's pipeline parallelism. Compared to data-parallel training, PipeDream reaches a high target accuracy on multi-GPU machines up to 5.3× faster for image classification tasks, up to 3.1× faster for machine translation tasks, 4.3× faster for language modeling tasks, and 3× faster for video captioning models. PipeDream is also 2.6× – 15× faster than model parallelism, up to 1.9× faster than hybrid parallelism, and 1.7× faster than simpler approaches to pipelining such as GPipe's approach.

2 BACKGROUND AND RELATED WORK

A DNN model is composed of many operators organized into *layers*. When parallelizing DNN training, these layers may be partitioned over the available workers in different ways. In this section, we cover two broad classes of parallel DNN training: intra- and inter-batch. We also highlight the challenges posed by DNN model and hardware diversity for effective parallelization.

2.1 Intra-batch Parallelism

The most common way to train DNN models today is intra-batch parallelization, where a single iteration of training is split across available workers.

Data Parallelism. In data parallelism, inputs are partitioned across workers. Each worker maintains a local copy of the model weights and trains on its own partition of inputs while periodically synchronizing weights with other workers, using either collective communication primitives like `all_reduce` [24] or parameter servers [38]. The amount of data communicated is proportional to the number of model weights and the number of workers participating in training.

The most commonly used form of data parallelism, referred to as *bulk synchronous parallel* or BSP [52][1], requires each worker to *wait* for gradients from other workers. Despite optimizations such as Wait-free Backpropagation [57], where weight gradients are sent as soon as they are available (common in modern frameworks), communication stalls are sometimes inevitable for large models

[1] In this paper, we use DP to refer to data-parallelism with BSP.

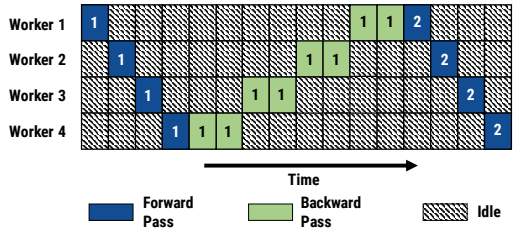


Figure 2: Model parallel training with 4 workers. Numbers indicate batch ID, and backward passes takes twice as long as forward passes. For simplicity, we assume that communicating activations/gradients across workers has no overhead.

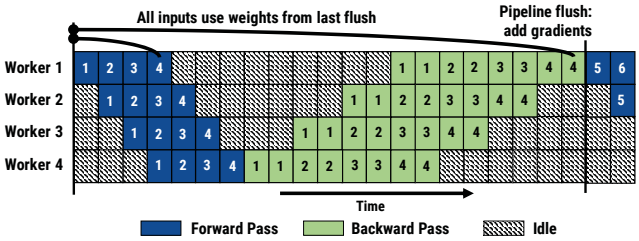


Figure 3: GPipe's inter-batch parallelism approach. Frequent pipeline flushes lead to increased idle time.

where the time needed to synchronize gradients across workers can dominate computation time.

Figure 1 quantitatively shows the fraction of training time spent in communication stalls with data parallelism for different classes of DNNs using three types of servers: 8-1080Ti GPU instances linked over PCIe within servers and 25Gbps interconnects across servers, 4-V100 GPU instances without `NVLink` and 10Gbps interconnects across servers, and 8-V100 GPU instances with `NVLink` interconnects within servers and 25Gbps interconnects across servers.

We focus on four key takeaways. First, the communication overhead for many of these models is high despite using multi-GPU servers and state-of-the-art communication libraries like NCCL. Data parallelism scales well for models like ResNet-50, which have a large number of convolutional layers with compact weight representations, but scales less well for other models with LSTM or fully-connected layers, which have more dense weight representations. Second, applications distributed across multi-GPU servers are bottlenecked by slower inter-server links, as evidenced by communication overheads spiking and then plateauing when training scales out to multiple servers. Data parallelism for such hierarchical networks can be a poor fit, since the same number of bytes are sent over both high- and low- bandwidth channels. Third, as the number of data-parallel workers increases, communication overheads increase for all models, even if training is performed on a multi-GPU instance with `NVLink`. Coleman et al. [17] showed similar results. Fourth, as GPU compute speeds increase (1080Tis to V100s), communication overheads also increase for all models.

Other DP Optimizations. Asynchronous parallel training (ASP) allows each worker to proceed with the next input minibatch before receiving the gradients from the previous minibatch. This approach improves hardware efficiency (time needed per iteration) over BSP by overlapping computation with communication, but also introduces staleness and reduces statistical efficiency (number of iterations needed to reach a particular target accuracy) [12, 20].

Seide et al. [45, 46] looked at quantizing gradients to decrease the amount of data needed to be communicated over the network. This approximation strategy is effective for limited scenarios but lacks generality; it does not hurt convergence for some speech models [47], but has not been shown to be effective for other types of models. Others have explored techniques from the HPC literature to reduce the overhead of communication [9, 24, 50, 51], often using highly specialized networking hardware. Our work is complementary to these techniques and focuses mainly on improving the performance of parallel DNN training when using commodity accelerators and interconnects available in public clouds.

Recent work has demonstrated that using large minibatches is effective for training ResNet-50, especially when combined with Layer-wise Adaptive Rate Scaling (LARS) [24, 31, 56]. Large minibatches reduce the communication overhead by exchanging parameters less frequently; however, our experiments show that such techniques lack generality beyond ResNet-50 and pipeline parallelism can outperform the fastest LARS data-parallel option.

Model Parallelism. Model parallelism is an intra-batch parallelism approach where the operators in a DNN model are partitioned across the available workers, with each worker evaluating and performing updates for only a subset of the model's parameters for all inputs. The amount of data communicated is the size of intermediate outputs (and corresponding gradients) that need to be sent across workers.

Although model parallelism enables training of very large models, vanilla model parallelism is *rarely* used to accelerate DNN training because it suffers from two major limitations. First, model-parallel training results in under-utilization of compute resources, as illustrated in Figure 2. Each worker is responsible for a group of consecutive layers; in this regime, the intermediate outputs (activations and gradients) between these groups are the only data that need to be communicated across workers.[2]

The second limitation for model-parallel training is that the burden of partitioning a model across multiple GPUs is left to the programmer [36], resulting in point solutions. Recent work explores the use of Reinforcement Learning to automatically determine device placement for model parallelism [42]. However, these techniques are time- and resource- intensive, and do not leverage the fact that DNN training can be thought of as a computational pipeline consisting of groups of consecutive layers – these assumptions make the optimization problem more tractable, allowing for exact solutions in polynomial time as we show in § 3.1.

Hybrid Intra-batch Parallelism. Recent work has proposed splitting a single iteration of the optimization algorithm among *multiple* dimensions. OWT [36] split the then-popular AlexNet model by hand, using data parallelism for convolutional layers that have a small number of weight parameters and large outputs, while

[2]While other partitioning schemes are possible, this is the most common, and the one we will use in this paper.

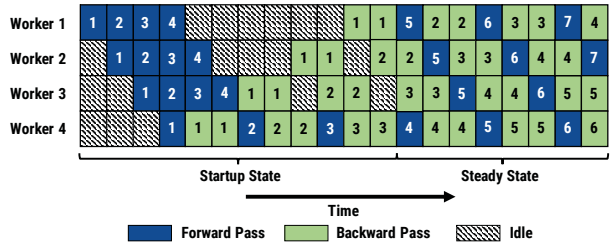


Figure 4: An example PipeDream pipeline with 4 workers, showing startup and steady states. In this example, the backward pass takes twice as long as the forward pass.

choosing to not replicate fully connected layers that have a large number of weight parameters and small outputs. OWT does not use pipelining. FlexFlow [33] proposed splitting a single iteration along samples, operators, attributes, and parameters, and describes an algorithm to determine how to perform this splitting in an automated way. However, FlexFlow does not perform pipelining, and we show in our experiments (§ 5.3) that this leaves as much as 90% of performance on the table.

2.2 Inter-batch Parallelism

Chen et al. [15] briefly explored the potential benefits of pipelining minibatches in model-parallel training, but do not address the conditions for good statistical efficiency, scale, and generality as applicable to large real-world models. Huo et al. [29] explored parallelizing the backward pass during training. Our proposed solution parallelizes both the forward and backward pass.

GPipe (concurrent work with an earlier PipeDream preprint [25]) uses pipelining in the context of model-parallel training for very large models [28]. GPipe does not specify an algorithm for partitioning a model, but assumes a partitioned model as input. GPipe further splits a minibatch into m microbatches, and performs forward passes followed by backward passes for these m microbatches (see Figure 3, $m = 4$). With a focus on training a large model like AmoebaNet, GPipe optimizes for memory efficiency; it uses existing techniques such as weight gradient aggregation and trades computation for memory by discarding activation stashes between the forward and the backward pass, instead opting to re-compute them when needed in the backward pass [14]. As a result, it can suffer from reduced hardware efficiency due to re-computation overheads and frequent pipeline flushes if m is small (§ 5.4).

In comparison, PipeDream addresses key issues ignored in prior work, offering a general solution that keeps workers well utilized, combining pipelining with intra-batch parallelism in a principled way, while also automating the partitioning of the model across the available workers.

2.3 DNN Model and Hardware Diversity

DNN models are diverse, with convolutional layers, LSTMs [55], attention layers [53], and fully-connected layers commonly used. These different types of models exhibit vastly different performance characteristics with different parallelization strategies, making the optimal parallelization strategy highly model-dependent.

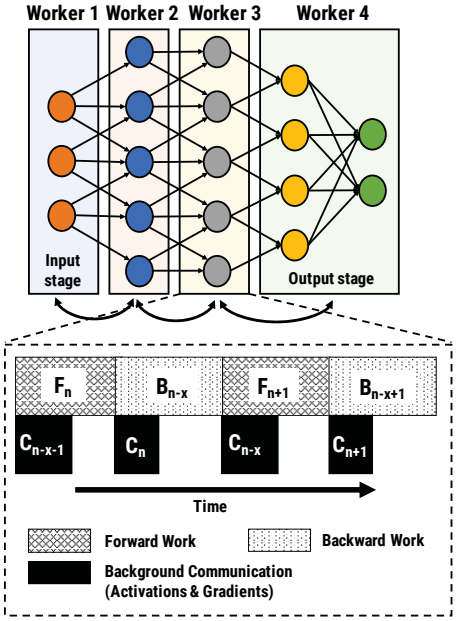


Figure 5: An example pipeline-parallel assignment with four GPUs and an example timeline at one of the GPUs (worker 3), highlighting the temporal overlap of computation and activation / gradient communication.

Picking an optimal parallelization scheme is challenging because the efficacy of such a scheme depends on the characteristics of the target deployment hardware as well; GPUs, ASICs, and FPGAs have very different compute capabilities. Moreover, interconnects linking these accelerators have different topologies and capacities; cloud servers are linked by tens to 100Gbps networks, accelerators within servers might be connected over shared PCIe trees (10 to 15GBps), and specialized expensive servers, such as the DGX-1 [23], use `NVLink` with point-to-point 30GBps bandwidth capabilities. This diversity in models and deployments makes it extremely hard to *manually* come up with an optimal parallelization strategy. PipeDream automates this process, as we discuss in § 3.1.

3 PIPELINE PARALLELISM

PipeDream uses *pipeline parallelism* (PP), a new parallelization strategy that combines intra-batch parallelism with inter-batch parallelism. Pipeline-parallel computation involves partitioning the layers of 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 GPU that performs the forward pass (and backward pass) for all layers in that stage.[3]

In the simplest case, only one minibatch is active in the system, as in traditional model-parallel training (Figure 2); in this setup, at most one GPU is active at a time. Ideally, we would like all GPUs to be active. With this in mind, we inject multiple minibatches into the pipeline one after the other. On completing its forward pass for a minibatch, each stage asynchronously sends the output activations

[3]We use GPUs as a concrete instance of accelerators and use the terms "GPU" and "worker" interchangeably.

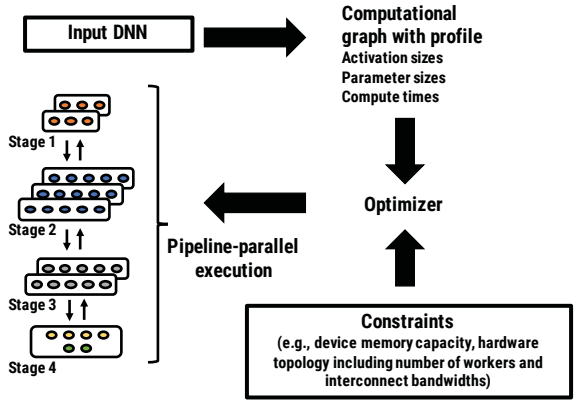


Figure 6: PipeDream's automated mechanism to partition DNN layers into stages. PipeDream first profiles the input DNN, to get estimates for each layer's compute time and output size. Using these estimates, PipeDream's optimizer partitions layers across available machines, which is then executed by PipeDream's runtime.

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 4).

Pipeline parallelism can outperform intra-batch parallelism methods for two reasons:

Pipelining communicates less. PP often can communicate far less than DP. Instead of having to aggregate gradients for all parameters and send the result to all workers, as is done in data-parallel approaches (using either collective communication or a parameter server), each worker in a PP execution has to communicate only subsets of the gradients and output activations, to only a single other worker. This can result in large reductions in communication for *some* models (e.g., >85% reduction for VGG-16, AWD LM).

Pipelining overlaps computation and communication. Asynchronous communication of forward activations and backward gradients across stages results in significant overlap of communication with the computation of a subsequent minibatch, as shown in Figure 5. This computation and communication are completely independent with *no* dependency edges, since they operate on different inputs, leading to easier parallelization.

However, to realize the opportunity of PP, PipeDream must overcome three challenges. In discussing PipeDream's solutions to these challenges, we will refer to Figure 6, which shows PipeDream's high-level workflow.

3.1 Challenge 1: Work Partitioning

PipeDream treats model training as a computation pipeline, with each worker executing a subset of the model as a stage. Like with any pipeline, the steady state throughput of the resulting pipeline is the throughput of the slowest stage. Having each stage process minibatches at vastly different throughputs can lead to bubbles in the pipeline, starving faster stages of minibatches to work on and resulting in resource under-utilization. Excessive communication between workers can also lower the throughput of the training pipeline. Moreover, the allocation of stages to workers needs to be model- and hardware-aware to be effective, and there may be cases where no simple partitioning across the GPUs achieves both limited communication and perfect load balance.

Solution: PipeDream's optimizer outputs a balanced pipeline. Its algorithm partitions DNN layers into stages such that each stage completes at roughly the same rate, while trying to minimize communication across workers in a topology-aware way (for example, large outputs should be sent over higher bandwidth links if possible). To further improve load balancing, PipeDream goes beyond straight pipelines, allowing a stage to be replicated (i.e., data parallelism is used on the stage). This partitioning problem is equivalent to minimizing the time taken by the slowest stage of the pipeline, and has the *optimal sub-problem property*: a pipeline that maximizes throughput given a worker count is composed of sub-pipelines that maximize throughput for smaller worker counts. Consequently, we use dynamic programming to find the optimal solution.

PipeDream exploits the fact that DNN training shows little variance in computation time across inputs. PipeDream records the computation time taken by the forward and backward pass, the size of the layer outputs, and the size of the associated parameters for each layer as part of an initial profiling step; this profile is used as the input to the optimizer's partitioning algorithm (Figure 6). The partitioning algorithm also takes into account other constraints such as hardware topology and bandwidth, number of workers, and memory capacity of the compute devices.

Profiler. PipeDream records three quantities for each layer l, using a short (few minutes) profiling run of 1000 minibatches on a single GPU: 1) T_l, the total computation time across the forward and backward passes for layer l on the target GPU, 2) a_l, the size of the output activations of layer l (and the size of input gradients in the backward pass) in bytes, and 3) w_l, the size of weight parameters for layer l in bytes.

PipeDream estimates the communication time by dividing the amount of data that needs to be transferred by the network bandwidth of the communication link. Assuming efficient `all_reduce` collective communication, in data-parallel configurations with m workers, each worker sends $(\frac{m-1}{m} \cdot |w_l|)$ bytes to other workers, and receives the same amount; this is used to estimate the time for weight synchronization for layer l when using data parallelism with m workers.

Partitioning Algorithm. Our partitioning algorithm takes the output of the profiling step, and computes: 1) a partitioning of layers into stages, 2) the replication factor (number of workers) for each stage, and 3) optimal number of in-flight minibatches to keep the training pipeline busy.

PipeDream's optimizer assumes that the machine topology is hierarchical and can be organized into levels, as shown in Figure 7. Bandwidths within a level are the same, while bandwidths across levels are different. We assume that level k is comprised of m_k components of level $(k-1)$, connected by links of bandwidth B_k. In Figure 7, $m_2 = 2$ and $m_1 = 4$. In addition, we define m_0 to be 1;

Figure 7: An example 2-level hardware topology. Solid green boxes represent GPUs. Each server (dashed yellow boxes) has 4 GPUs connected internally by links of bandwidth B_1; each server is connected by links of bandwidth B_2. In real systems, $B_1 > B_2$. Figure best seen in color.

m_0 represents the number of compute devices within the first level (solid green boxes in Figure 7).

PipeDream's optimizer solves dynamic programming problems progressively from the lowest to the highest level. Intuitively, this process finds the optimal partitioning within a server and then uses these partitions to split a model optimally across servers.

Notation. Let $A^k(i \rightarrow j, m)$ denote the time taken by the slowest stage in the optimal pipeline between layers i and j using m workers at level k. The goal of our algorithm is to find $A^L(0 \rightarrow N, m_L)$, and the corresponding partitioning, where L is the highest level and N is the total number of layers in the model.

Let $T^k(i \rightarrow j, m)$ denote the total time taken by a single stage spanning layers i through j for both forward and backward passes, replicated over m workers using bandwidth B_k.

Formulation. For all k from 1 to L,

$$T^k(i \rightarrow j, m) = \frac{1}{m} \max \begin{cases} A^{k-1}(i \rightarrow j, m_{k-1}) \\ \dfrac{2(m-1)\sum_{l=i}^{j} |w_l|}{B_k} \end{cases}$$

where the first term inside the max is the total computation time for all the layers in the stage using level $k-1$ as the computation substrate, and the second term is the time for data-parallel communication among all layers in the stage. The result of the max expression above gives the effective time spent processing m inputs while performing compute and communication concurrently; thus, the effective time spent processing a single input is this term divided by m.

The optimal pipeline can now be broken into an optimal sub-pipeline consisting of layers from 1 through s with $m - m'$ workers followed by a single stage with layers $s+1$ through j replicated over m' workers. Then, using the optimal sub-problem property, we have:

$$A^k(i \rightarrow j, m) = \min_{i \le s < j} \min_{1 \le m' < m} \max \begin{cases} A^k(i \rightarrow s, m - m') \\ 2a_s / B_k \\ T^k(s+1 \rightarrow j, m') \end{cases}$$

where the first term inside the max is the time taken by the slowest stage of the optimal sub-pipeline between layers i and s with $m - m'$ workers, the second term is the time taken to communicate the activations and gradients of size a_s between layers s and $s+1$, and the third term is the time taken by the single stage containing layers $s+1$ to j in a data-parallel configuration of m' workers.

When solving for level k, we use $A^{k-1}(i \rightarrow j, m_{k-1})$, which is the optimal total computation time for layers i through j using all workers available in a single component at level $(k-1)$ (in the expression $T^k(i \rightarrow j, m)$). In Figure 7, this would represent determining how best to partition intermediate layers of the model using all workers in a yellow server.

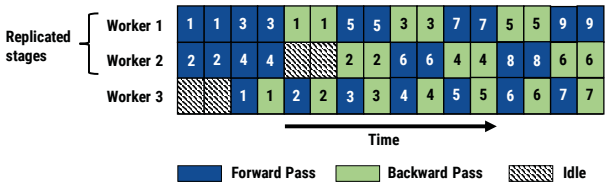


Figure 8: An example PipeDream pipeline with 3 workers and 2 stages. We assume that forward and backward passes in the first stage take two time units, while forward and backward passes in the second stage take only a single time unit. The first stage in this pipeline is replicated twice so that each stage sustains roughly the same throughput. Here, we assume that forward and backward passes take equal time, but this is not a requirement of our approach.

Initialization. Level 0 uses the profiled computation times: $A^0(i \rightarrow j, m_0) = \sum_{l=i}^{j} T_l$. For $k > 0$, optimal compute times with all compute devices in the previous level are used: $A^k(i \rightarrow j, 1) = A^{k-1}(i \rightarrow j, m_{k-1})$.

Runtime Analysis. For a given level k, the total number of sub-problems is $O(N^2 m_k)$. Time complexity per sub-problem is $O(N m_k)$, leading to a total time complexity of $O(N^3 m_k^2)$ for level k. Total time complexity is $\sum_{k=1}^{L} O(N^3 m_k^2)$. In our experiments, the running time is under 8 seconds.

3.2 Challenge 2: Work Scheduling

Unlike traditional uni-directional pipelines, training in PipeDream involves a bi-directional pipeline, where an input minibatch proceeds through the computation pipeline first forward and then backward. Each active minibatch in the pipeline may be in a different stage, either in the forward pass or backward pass. As a result, each worker in the system needs to determine whether it should i) perform its stage's forward pass for a minibatch, pushing the minibatch to downstream workers, or ii) perform its stage's backward pass for a different minibatch, pushing the minibatch to upstream workers. In addition, how should minibatches be routed with replicated stages?

Solution: In the startup phase, the input stage admits enough minibatches to keep the pipeline full in steady state. Based on the partitioning generated by our algorithm, the optimal number of minibatches admitted *per input stage replica* to keep the pipeline full in steady state is given by:

NUM_OPT_ACTIVE_MINIBATCHES ($NOAM$) =

⌈ (# workers) / (# of replicas in the input stage) ⌉.

Once in steady state, each stage *alternates* between performing its forward pass for a minibatch and its backward pass for an earlier minibatch. We call this the *one-forward-one-backward* (1F1B) schedule. 1F1B ensures that every GPU is occupied with a minibatch in a balanced pipeline, with each stage producing outputs in aggregate at roughly the same rate. It also ensures backward passes from inputs are applied at regular intervals of time.

Figure 4 shows the corresponding compute timeline for a pipeline with 4 stages. The NOAM for this configuration is 4. In the startup phase, the input stage admits exactly four minibatches that propagate their way to the output stage. As soon as the output stage completes its forward pass for the first minibatch, it performs its backward pass for the same minibatch, and then starts alternating between forward and backward passes for subsequent minibatches. As the backward pass starts propagating to earlier stages in the pipeline, every stage starts alternating between its forward and backward passes for different minibatches. As shown in the figure, every worker is performing either a forward or backward pass for some minibatch in steady state.

When a stage is run in a data-parallel configuration (replicated across multiple GPUs), we use deterministic round-robin load balancing based on a minibatch identifier to spread work across the replicas. Such deterministic load-balancing ensures that each minibatch is routed to the same worker for both the forward and backward passes of the stage, which is important since parameters and intermediate outputs from the forward pass are needed for the backward pass. This mechanism, which we call *one-forward-one-backward-round-robin* (1F1B-RR), is a *static* policy that is executed without expensive distributed coordination. Figure 8 shows this mechanism in action for a simple 2-1 configuration, with the first stage replicated twice, and the second stage un-replicated. In the first stage, all inputs with even minibatch IDs are processed by worker 1, while inputs with odd minibatch IDs are processed by worker 2. Worker 3 in the second stage processes all inputs. All workers perform a forward pass followed by a backward pass on a different input minibatch.

For 1F1B-RR to be effective, it is *not necessary* for the forward pass to take as long as the backward pass. In fact, we observe that the backward pass is always larger than the forward pass in practice. 1F1B-RR remains an effective scheduling mechanism, as highlighted in Figure 4.[4]

3.3 Challenge 3: Effective Learning

In a naively pipelined system, each stage's forward pass for a minibatch is performed using one version of parameters and its backward pass is performed using a different version of parameters. Figure 4 illustrates this using a partitioning with four workers and no stage replication. In stage 1, the forward pass for minibatch 5 is performed after the updates from minibatch 1 are applied, whereas the backward pass for minibatch 5 is performed after updates from minibatches 2, 3, and 4 are applied. As a result, in the backward pass for minibatch 5 on stage 1, the gradient is computed using a different set of weights than the ones used in the corresponding forward pass; this discrepancy in weight versions results in invalid gradients and can prevent model convergence.

Solution: PipeDream uses a technique called *weight stashing* to avoid a fundamental mismatch between the version of weights used in the forward and backward pas. Weight stashing maintains multiple versions of the weights, one for each active minibatch. Each stage processes a minibatch using the latest version of weights available in the forward pass. After completing the forward pass,

[4] 1F1B-RR produces a full steady state pipeline even for cases where the ratio of backward- to forward-pass time is not an integer (e.g., 3 to 2).

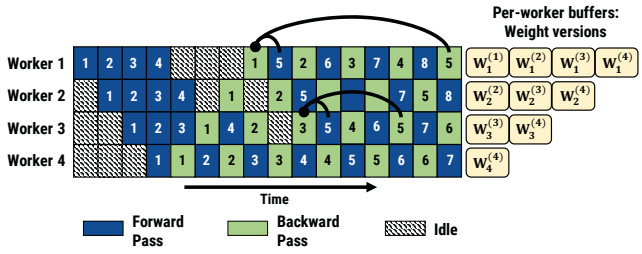


Figure 9: Weight stashing as minibatch 5 flows across stages. Arrows point to weight versions used for forward and backward passes for minibatch 5 at the first and third stages.

PipeDream stores the weights used for that minibatch. The *same weight version* is then used used to compute the weight update and upstream weight gradient in the minibatch's backward pass.

Weight stashing ensures that *within a stage*, the same version of model parameters are used for the forward and backward pass of a given minibatch. For example, in Figure 9, minibatch 5 uses parameter updates from minibatch 1 on machine 1 and from 2 on machine 2. Weight stashing does not guarantee the consistency of parameter versions used for a given minibatch *across* stages.

Vertical Sync. Vertical sync is an optional technique in PipeDream that eliminates the potential inconsistency *across stages*. For example, in Figure 4, minibatch 5 uses parameters updated by minibatch 1 on all workers for both its forward and backward passes when using vertical sync. Each minibatch (b_i) that enters the pipeline is associated with the latest weight version ($w^{(i-x)}$) seen at the input stage. This information is propagated along with the activations and gradients as the minibatch b_i flows through the pipeline in the forward direction. Across all stages, the forward pass for b_i uses the stashed weights $w^{(i-x)}$ as opposed to the latest weight update. After performing the backward pass for b_i (using stashed weights $w^{(i-x)}$), each stage independently applies weight updates to create the latest weights ($w^{(i)}$), and can then delete $w^{(i-x)}$. This coordination across stages is asynchronous.

The semantics of vertical sync are different from GPipe (and data parallelism). In particular, gradients are not aggregated over all in-flight minibatches in the system – vertical sync merely ensures that the same weight versions are used to compute gradients across different workers (but the weight versions to which gradients are applied are different from those used to compute the corresponding gradients).

Staleness. We can now formalize the degree of staleness of weight updates for each of these techniques. For this discussion, we assume a straight pipeline (i.e., no stage replication) with the model split into n stages; the weights in each stage are represented as w_1, w_2, and so on. In addition, we denote $w_l^{(t)}$ as the weights w_l after t minibatches.

Now, after every minibatch, we compute $\nabla f(w_1, w_2, \ldots, w_n)$, which is the gradient averaged over all samples in the minibatch. Vanilla minibatch SGD (f is the loss function, ν is the learning rate) has the following gradient update:

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t)}, w_2^{(t)}, \ldots, w_n^{(t)})$$

With weight stashing, gradients in stage 1 are computed with weights that are n steps delayed, gradients for stage 2 are computed with weights that are $n-1$ steps delayed, etc. Mathematically, this means the weight update looks like:

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+2)}, \ldots, w_n^{(t)})$$

Without weight stashing, the weight update is not a valid gradient of the loss function f for any vector $w_1, \ldots, w_n$.

Adding vertical sync alters the weight update to:

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+1)}, \ldots, w_n^{(t-n+1)})$$

This is semantically similar to data parallelism with BSP synchronization on n workers (with the same per-worker minibatch size), with the same staleness (but gradients averaged over a minibatch of size B instead of nB).

Memory Overhead. Pipelining does not significantly increase per-worker memory usage relative to data parallelism, even with weight stashing. Consider a straight pipeline (no data-parallel stages), where a model is divided across n workers, with each worker holding $1/n$ of the weights. With non-pipelined model-parallel training, each worker would need $1/n$ of the memory compared to data parallel training. Admitting n inputs into the pipeline, as PipeDream does, increases this by at most a factor of n, because a version of <weights, activations> is needed for each in-flight minibatch. Thus, PipeDream's peak per-worker memory usage is on par with data parallelism.

PipeDream's memory footprint can be further reduced by using existing techniques: efficient encoding or compression of intermediate data [30], gradient aggregation where weight gradients are added to a single buffer at a stage for m minibatches before performing a weight update, and trading computation time for activation-stash memory by discarding them in the forward pass and recomputing them as needed during the backward pass [14].

PipeDream's default semantics exclude vertical sync as it requires more metadata to be stored at every stage in the pipeline. PipeDream's default semantics (weight stashing but no vertical sync) are between regular minibatched SGD on a single worker, and data parallelism with BSP synchronization [19, 27]. Our evaluation demonstrates its effectiveness across models, datasets, and hardware configurations.

4 IMPLEMENTATION

The interface to PipeDream is implemented as a standalone Python library of ~3000 LOC that manages device memory, schedules work, and handles communication. PipeDream uses PyTorch [5] for auto-differentiation and to execute operators; however, PipeDream is extensible and can work with other ML frameworks such as Tensorflow [8], MXNet [13], and CNTK [45]. As a proof of concept, we also integrated PipeDream with Caffe [32].

PipeDream first profiles the model on a single GPU with a subset of inputs from the training dataset (Figure 6). It then runs the optimization algorithm described in § 3.1 to partition the DNN model into stages, with some stages possibly replicated.

PipeDream's optimizer returns an annotated operator graph, with each model layer mapped to a stage ID. PipeDream performs a BFS traversal of this graph and generates code for each stage as a separate `torch.nn.Module`, ordering operators in each stage to make sure their input-output dependencies from the original PyTorch model graph are respected. The PipeDream runtime then assigns each stage (including replicas for replicated stages) to a single worker according to its 1F1B-RR schedule.

Parameter State. PipeDream maintains all parameters associated with the layers assigned to the stage directly in GPU memory. PipeDream applies updates to the most recent parameter version when the weight update becomes available if the stage is not replicated. The weight updates are synchronized across replicas prior to being applied if the stage is replicated. When a newer version of the parameters becomes available, the prior version is *not* immediately discarded. Parameters are discarded only once a backward pass that uses fresher parameters is performed.

Intermediate State. Each stage's input and output data is assigned a unique blob ID. Upon receiving intermediate data from the prior stage (or from disk in the case of the input stage), PipeDream copies the intermediate data to GPU memory and places a pointer to the associated buffer in a work queue. Intermediate data from the forward pass is not discarded until the associated minibatch completes that stage's backward pass. Intermediate data from the backward pass is freed as soon as the worker finishes using it, and if necessary, after it is sent to the next stage.

Stage Replication. PipeDream uses PyTorch's `DistributedDataParallel` library [6] to synchronize parameters for layers of data-parallel stages. Using wait-free back propagation, weight gradients are communicated to servers as soon as they are computed, rather than waiting for computation to finish for all layers. Since we support replication of individual stages, data-parallel training is effectively a special case in our framework – we represent this as a single stage that contains all the layers of the DNN model, and replicate the stage across all available GPUs. We use the NCCL communication backend [3] for data-parallel baselines as we find it to be faster than Gloo [1] for the large tensors exchanged in DP. We also find that Gloo is faster than NCCL for small tensors that are exchanged across the pipeline, such as activations and gradients. PipeDream defaults to using Gloo for all inter-GPU communication when performing pipeline-parallel training because we are unable to use both Gloo (across the pipeline) and NCCL (across replicated stages) at the same time in a stage.

Checkpointing. PipeDream supports periodic checkpointing of model parameters for fault tolerance, with default checkpoints made across stages at the end of every epoch. Checkpoints don't require expensive global coordination. Each stage dumps its model parameters locally when it performs the backward pass for the last minibatch in an epoch. Restarting a run due to failures entails starting from the last successfully created checkpoint for all stages.

5 EVALUATION

This section evaluates the effectiveness of PipeDream for seven different DNNs on three different clusters. The results of our experiments support a number of important findings: 1) PipeDream

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.

Cluster name	Server SKU	GPUs per server	Interconnects Intra-, Inter-server
Cluster-A	Azure NC24 v3	4x V100	PCIe, 10 Gbps
Cluster-B	AWS p3.16xlarge	8x V100	NVLink, 25 Gbps
Cluster-C	Private Cluster	1 Titan X	N/A, 40 Gbps

Table 2: Characteristics of servers used in experiments.

achieves significant speedups in time-to-target-accuracy across a wide range of different learning tasks on different hardware deployments, 2) PipeDream is more efficient than other recently proposed inter-batch approaches, 3) PipeDream greatly reduces overheads of communication and does not significantly increase memory footprint compared to data-parallel training, and 4) combining pipelining, model parallelism, and data parallelism outperforms model-, data-, or hybrid-parallelism in isolation.

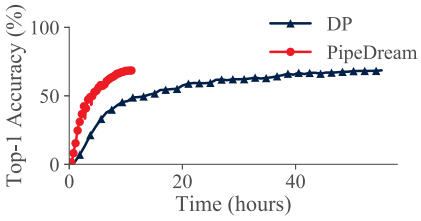
5.1 Experimental Setup

Tasks and Datasets. We use four tasks and four datasets in our experiments: 1) Image Classification, using the ImageNet-1K (ILSVRC12) [44] dataset; 2) Translation, using the WMT16 English to German dataset for training, and the "newstest2014" dataset for validation; 3) Language Modeling, using the Penn Treebank (PTB) [41] dataset; and 4) Video Captioning (S2VT), using the Microsoft Video description corpus (MSVD) [11].

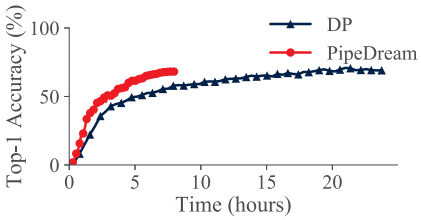
Clusters. We use three different clusters in our experiments, summarized in Table 2. *Cluster-A* has servers with 4 NVIDIA V100 GPUs each (Microsoft Azure NCv3 instances), with 16 GB of GPU device memory, and a 10 Gbps Ethernet interface. *Cluster-B* has servers with 8 V100s each (AWS EC2 p3.16xlarge instances), with 16 GB of GPU device memory, and a 25 Gbps Ethernet interface. GPUs within servers are connected via a shared PCIe interconnect on Cluster-A, and via point-to-point `NVLink` on Cluster-B. All servers run 64-bit Ubuntu 16.04 with CUDA toolkit 10.0 and cuDNN v7.4. *Cluster-C* has servers with 1 NVIDIA Titan X GPU and 12 GB of GPU device memory, connected via 40 Gbps Ethernet. Unless otherwise stated, all our experiments are run on multi-GPU servers (Cluster-A and Cluster-B).

Models. We use seven different DNN models in our experiments across the four applications: 1) VGG-16 [48], 2) ResNet-50 [26], 3) AlexNet [37], 4) Google Neural server Translation (GNMT) with 8 LSTM layers [55], 5) GNMT with 16 LSTM layers, 6) AWD Language Model (LM) [40], and 7) the S2VT [54] sequence-to-sequence model for video transcription.

Batch Sizes and Training Methodology. We use the largest per-GPU minibatch that fits in one GPU's memory – anything larger yields out-of-memory exceptions. This ensures that we hit peak achievable FLOPs on a single device. Unless otherwise stated, we report per-GPU minibatch sizes (G); for data-parallel runs with n workers, the global minibatch size (BS) is $n \times G$. The global minibatch sizes we use are consistent with those used by the ML community and reported in the literature for these models. We use a per-GPU minibatch size of 64 per GPU for VGG-16, 256 for AlexNet, 128 for ResNet-50 (e.g., $BS = 1024$ for 8 GPUs), 64 for GNMT, 80 for S2VT, and batch size of 80 for LM. We train the VGG-16, ResNet-50, Language Modeling, and S2VT models using SGD with an initial learning rate of 0.01, 0.1, 30.0, and 0.01 respectively. For GNMT, we

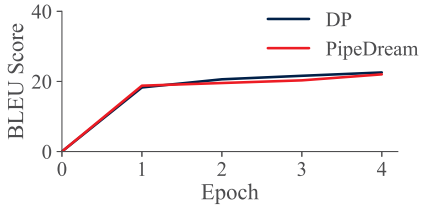


(a) Cluster-A.

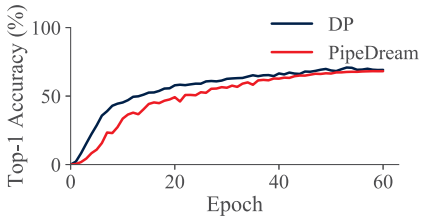


(b) Cluster-B.

Figure 10: Accuracy vs. time for VGG-16 using 16 GPUs. Each circle or triangle represents two epochs of training.



(a) GNMT-16



(b) VGG-16

Figure 11: Accuracy vs. epoch using 16 GPUs on Cluster-B.

use the Adam optimizer [35] with an initial learning rate of 0.0003. We use full (`fp32`) precision in all our experiments.

For all experiments (other than AlexNet), we measure the time taken to train to a target validation accuracy: top-1 accuracy of 68% for VGG-16 [7], top-1 accuracy of 75.9% for ResNet-50, BLEU score of 21.8 for GNMT, a validation perplexity of 98 for LM, and a METEOR [22] score of 0.294 for S2VT. Guided by prior work, we adjust the learning rate during training to converge to the desired result faster [35, 49] and utilize learning rate warm-up for large global batch sizes [24]. We use the same learning rate schedules for PipeDream and data-parallel training. For AlexNet we use synthetic data (otherwise, data loading is the bottleneck) and measure throughput.

5.2 Comparison to Data Parallelism

Table 1 summarizes results comparing PipeDream with data-parallel training (DP). The table shows PipeDream's auto-generated configurations and their speedups in training time-to-accuracy over corresponding data-parallel training configurations.[5]

PipeDream Configurations. As described in § 3.1, given a DNN model and a set of servers, PipeDream's optimizer automatically chooses to partition the model into stages, while also deciding the optimal replication factor for each stage. Although most prior research has focused on improving data-parallel training, our results indicate that the best configurations for many models is not data parallelism, despite the use of many important optimizations such as wait-free back propagation. In all but one of our experiments, the best PipeDream configuration combines model parallelism, pipelining, and sometimes data parallelism; each of these configurations outperforms data-parallel training, highlighting the importance of combining inter-batch pipelining with intra-batch parallelism. PipeDream's optimizer recommends data parallelism for ResNet-50 because its weight representations are small and its outputs are large. PipeDream's optimizer, besides determining the optimal configuration, also automatically decides where to partition the DNN training graph; these partitioning decisions are not shown in Table 1.

Image Classification. We compare PipeDream and DP time-to-accuracy for VGG-16 using 4 servers in Cluster-A (4x4 (A) in Table 1. PipeDream reaches target accuracy 5.28× faster than DP on a single server, due to a reduction in inter-server communication. Figure 10 (a) shows this comparison as the DNN is trained over time. In the 4-server configuration, PipeDream's optimizer (§ 3.1) recommends a 15-1 configuration – in this case, VGG-16's convolutional layers are replicated, while the large fully connected layers are not, reducing communication overhead. Moreover, pipelining across the two stages helps keep all workers busy.

Compared to Cluster-A, which has 4 GPUs per server connected via PCIe, Cluster-B has 8 GPUs per server connected over faster peer-to-peer NVLink interconnects. On 2 servers on Cluster-B (16 GPUs total), PipeDream reaches target accuracy 2.98× faster than DP when training VGG-16. Due to the faster interconnects on Cluster-B, both PipeDream and DP reach target accuracy faster than on Cluster-A (see Figure 10).

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. For AlexNet, we compare throughput of PipeDream on Cluster-A and Cluster-B. On Cluster-A, PipeDream achieves a time-per-epoch speedup of 4.92× with 4 servers. On Cluster-B, PipeDream achieves a speedup of 2.04× when using 16 GPUs.

Translation. We show results for the GNMT model with 8 LSTM layers (GNMT-8) and 16 LSTM layers (GNMT-16) in Table 1). Using 1 server on Cluster-A, PipeDream reaches target accuracy ~ 1.5×

[5] A configuration indicates how layers are partitioned into stages amongst workers.

Model	Scale (# V100s)	Cluster-B / official MLPerf v0.5
GNMT-8	256	1.94×
SSD	64	3.29×
Mask R-CNN	64	2.32×

Table 3: Increase in per-epoch times for data-parallel training when moving from dedicated clusters used in official MLPerf v0.5 entries to public clouds like Cluster-B. The *same* code is used for both sets of runs.

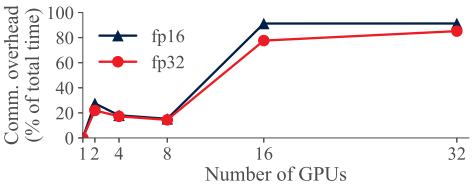


Figure 12: Communication overhead of data-parallel training using different server instances using PyTorch 1.1 and NCCL [3] for a GNMT-8 model with `fp16` and `fp32` precision.

faster than DP for GNMT-8 and GNMT-16. When using 4 servers (16 GPUs) on Cluster-A, PipeDream reaches target accuracy 2.92× (GNMT-8) and 2.95× (GNMT-16) faster than DP. We show in § 5.5 that PipeDream significantly reduces communication compared to DP, thus reducing its time to target accuracy.

On 2 servers (16 GPUs) of Cluster-B, PipeDream reaches target accuracy 3.14× faster than DP for GNMT-16, choosing a "straight" configuration (no stage replication). For GNMT-8, PipeDream falls back to data parallelism, since the smaller model has lower communication overhead on servers with fast `NVLink` interconnects between GPUs on the same server, and GNMT-8 does not have enough layers for a 16-deep straight pipeline.

Language Modeling. This model is made up of six LSTM layers that contain a large number of model parameters (0.41GB), making data-parallel training inefficient. Using a single server on Cluster-A, PipeDream reaches target accuracy 4.25× faster than DP. PipeDream chooses a "straight" configuration that reduces communication by 88% compared to DP.

Video Captioning. PipeDream chooses to use a 2-1-1 configuration for the S2VT on Cluster-C, reducing communication by 85% compared to DP, which in turn allows it to reach target accuracy 3.01× faster than DP.

Comparison to MLPerf v0.5. For ResNet-50 and GNMT-8, we observe that our data-parallel baseline on a single server with 8 GPUs in Cluster-B is comparable to the MLPerf v0.5 entry that uses a similar hardware configuration. However, we observe that per-epoch times on public cloud servers are slower than official MLPerf v0.5 entries for multi-server DP deployments, since slower communication links on public cloud servers (compared to dedicated clusters used in the MLPerf entries) make `all_reduce` communication slower. We cannot measure this difference in time-to-accuracy at the scales used by the MLPerf entries as it is cost prohibitive, but Table 3 compares the advertised training throughput of official MLPerf v0.5 [2]

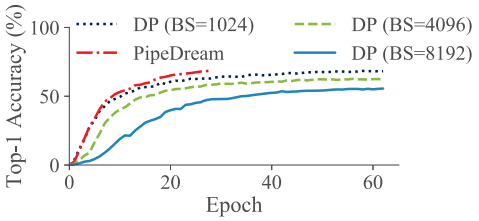


Figure 13: Statistical efficiency (accuracy vs. epoch) using LARS (VGG-16, 8 GPUs).

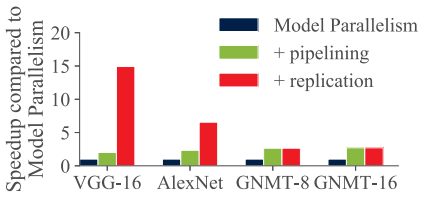
entries with data-parallel runs on `p3.16xlarge` instances using the *same code*. Coleman et al. observed similar results [17], both for official DAWNBench and MLPerf entries.

Furthermore, with 8 GPUs, for GNMT-8, while full precision is slower than the entry using mixed precision, we use a `fp32` baseline to be consistent with the rest of the evaluation in this paper. Figure 12 shows that communication overheads for data parallelism with mixed precision are higher than with full precision, and thus the speedups we highlight with pipeline parallelism should carryover (or improve) with mixed precision training.

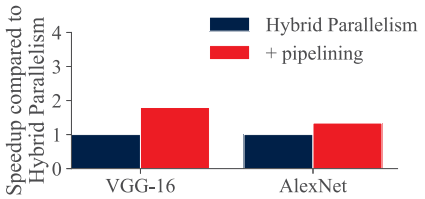
Comparison to DP with large minibatches. Recent work has demonstrated that using large minibatches is effective for training ResNet-50 and AlexNet models, especially when combined with Layer-wise Adaptive Rate Scaling (LARS). [24, 31, 56]. LARS uses different learning rates for each layer based on the ratio of the weight norm to the gradient norm. Large minibatches decrease the frequency of communication, reducing the communication overhead for data parallelism. Figure 13 shows 8-server results for data-parallel training of VGG-16 using LARS and large minibatches on Cluster-C. Minibatches of 1024 had the fastest time-to-target-accuracy, while minibatches of 4096 and 8192 failed to reach target accuracy, highlighting the lack of generality of such approaches. PipeDream still reaches target accuracy over 2.4× faster than the fastest data-parallel option (1024 with LARS).

Comparison to Asynchronous Parallelism (ASP). ASP can reduce communication overhead in data-parallel training. Unlike BSP, which synchronizes parameters after every minibatch, ASP has no synchronization overheads, and workers use the most recent parameter data *available*. The result is often poor statistical efficiency. For example, when training VGG-16 on 4 Cluster-B servers, ASP data-parallel takes 7.4× longer than PipeDream to reach a 48% accuracy (when we terminate ASP for taking too long to converge), even though ASP has minimal communication delays. Similar results have been shown by Chen et al. [12].

Statistical Efficiency. Figure 11 shows accuracy vs. epoch for VGG-16 and GNMT-16 on Cluster-B. We do not show accuracy vs. epoch graphs for other experiments due to space constraints. However, we consistently observe that PipeDream reaches target accuracy in a similar number of epochs as DP (as can be seen by the fact that TTA and epoch time speedups are the same for many rows in Table 1). This highlights the fact that PipeDream's weight stashing mechanism is able to achieve statistical efficiency comparable to data parallelism, and that PipeDream's speedups are due to better system performance.



(a) Model Parallelism.



(b) Hybrid Parallelism.

Figure 14: Comparison of PipeDream (red) to non-DP intra-batch techniques for 4-GPU configurations on Cluster-A.

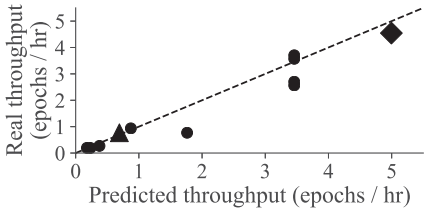


Figure 15: Real vs. optimizer's predicted throughput for VGG-16 with 16 workers. Each symbol represents a different partition, including the triangle for vanilla data-parallelism and the diamond for the optimizer's selection.

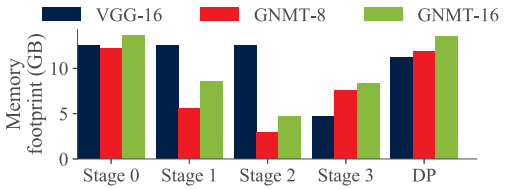


Figure 16: Memory footprint for various models using 4 GPUs. Per-GPU memory footprint is shown for data parallelism, and is identical on all GPUs.

5.3 Comparison to Other Intra-batch Parallelism Schemes

This section compares PipeDream to other intra-batch parallelization techniques besides data parallelism.

Model Parallelism. Figure 14a compares model parallelism (blue bars), straight pipelines without replication (green bars), and pipelining with stage replication (red bars). For all four models, pipelining alone increases throughput by 2× or more. For GNMT-8 and GNMT-16, PipeDream's optimizer chooses not to replicate any stages, resulting in identical configurations for the green and red bars. For VGG-16 and AlexNet, PipeDream replicates the first stage, leading to speedups of 14.9× and 6.5× compared to model parallelism.

Hybrid Parallelism. Figure 14b shows that pipelining for a configuration that combines data and model parallelism (similar to those proposed by Krizhevsky et al. [36] and FlexFlow [33, 34]) increases throughput by as much as 80%. In running FlexFlow for AlexNet on Cluster-B (not shown in Figure 14b), we observe that PipeDream is 1.9× faster; a speedup due to pipelining over hybrid parallelism. Note that the same number of bytes are being communicated across workers with and without pipelining. Speedups are achieved by overlapping compute and communication, and consequently better utilization of compute resources.

5.4 Comparison to Inter-batch Parallelism

We compare training GNMT-16 using PipeDream and our implementation of GPipe using 16 GPUs on Cluster-A and Cluster-B. GPipe does not provide an algorithm for partitioning work across stages, so we use the same partitions as PipeDream. GPipe also does not provide an algorithm for how many items should be permitted into the "pipeline" (pipeline depth). When we set the pipeline depth to be equivalent to "NOAM" in PipeDream (§ 3.2), GPipe experiences 55% and 71% throughput slowdowns compared to PipeDream on Cluster-A and Cluster-B, respectively. Setting the pipeline depth for GPipe to the largest number that does not cause an out-of-memory exception, leads to throughput slowdowns of 35% and 42% on Cluster-A and Cluster-B, respectively. These throughput slowdowns are largely due to more frequent pipeline flushes compared to PipeDream (Figures 3 and 4).

5.5 Microbenchmarks

We evaluate PipeDream's optimizer, its communication overhead and memory footprint, and the effect of pipeline depth on throughput and memory footprint.

Optimizer. PipeDream's optimizer is efficient, generating optimal training configurations in under 8 seconds for all models and hardware deployments evaluated. As one example, Figure 15 shows real vs. predicted throughputs for various configurations for VGG-16 with 16 workers. Predicted and real throughputs are strongly linearly correlated, and the optimizer picks the best configuration among those tested.

Memory Footprint. Figure 16 shows the per-stage memory footprint of PipeDream for 4-stage configurations for three different models. PipeDream's worst-case memory footprint is on par with that of data parallelism, even though PipeDream stashes multiple weight and activation versions. This is because each stage in PipeDream is responsible for only a fraction of the total number of weights and activations in the model. As PipeDream scales to include more stages, the memory footprints remain consistent as discussed in § 3.3.

Communication Overhead. Figure 17 shows the amount of communication performed *per training sample* in the best non-DP configuration compared to the amount of communication performed in data-parallel training. For GNMT-8, GNMT-16, and VGG-16, the communication overhead for the best non-DP configuration is far less than the communication overhead for the DP configuration.

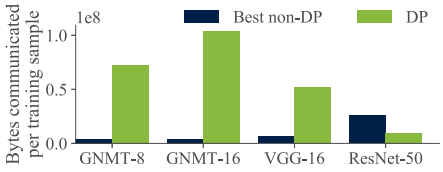
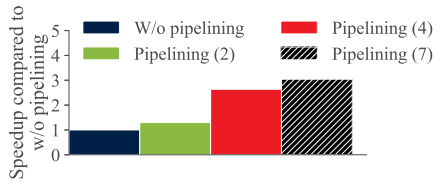
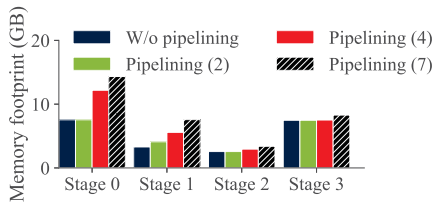


Figure 17: Bytes communicated per training sample by data-parallel (DP) and the best non-DP configurations for 4 GPUs on Cluster-A.



(a) Throughput.



(b) Memory Overhead.

Figure 18: Effect of pipeline depth on throughput and memory overhead for GNMT-8 on 4 V100s in Cluster-A.

For ResNet-50, the amount of communication for the best *non-data-parallel* configuration is higher than the DP configuration, thus explaining why PipeDream's optimizer chooses to perform ResNet-50 training using a data-parallel configuration.

Effect of Pipeline Depth. Figure 18 shows the effect of varying pipeline depth on throughput and memory overhead for GNMT-8. We make three observations: 1) Memory footprint with no pipelining is different across stages, since PipeDream's optimizer tries to load balance compute and communication, and *not* memory footprint (working set still fits comfortably in GPU memory). 2) As the pipeline depth increases from 2 to 7, memory footprint increases because the number of weights and activations that need to be stashed increases proportionally. 3) In our experiments, a pipeline depths of 4 (NOAM) and 7 give the highest throughput. While the working set of stages fits in GPU memory (16 GB), if required, pipeline depth can be decreased to trade throughput for reduced memory footprint. Throughput increases as pipeline depth increases since communication can be more easily hidden as the number of inputs in the pipeline increases, reducing pipeline stalls and thus improving resource utilization.

6 CONCLUSION

Pipeline-parallel DNN training helps reduce the communication overheads that can bottleneck intra-batch parallelism. PipeDream automatically partitions DNN training across workers, combining *inter-batch pipelining* with intra-batch parallelism to better overlap computation with communication while minimizing the amount of data communicated. Compared to state-of-the-art approaches, PipeDream completes training up to 5.3× faster across a range of DNNs and hardware configurations.

ACKNOWLEDGMENTS

PipeDream is part of Project Fiddle at MSR. We thank the MSR Lab LT, especially Ricardo Bianchini and Donald Kossmann, for their enthusiastic and unwavering support of Project Fiddle, and for their generous support in procuring the many resources required to develop and evaluate PipeDream.

We also thank our shepherd, Christopher J. Rossbach, the anonymous SOSP reviewers, Zeyuan Allen-Zhu, Jakub Tarnawski, Shivaram Venkataraman, Jack Kosaian, Keshav Santhanam, Sahaana Suri, James Thomas, Shoumik Palkar, and many of our MSR colleagues for their invaluable feedback that made this work better. We acknowledge the support of the affiliate members of the Stanford DAWN project (Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, SAP, Teradata, and VMware), Amazon Web Services, and Cisco. We also thank the members and companies of the CMU PDL Consortium (Alibaba, Amazon, Datrium, Facebook, Google, HPE, Hitachi, IBM, Intel, Micron, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma) and VMware for their feedback and support. This research was also supported in part by NSF CAREER grant CNS-1651570, NSF Graduate Research Fellowship grant DGE-1656518, NSF grant CCF-1725663, and the Intel STC on Visual Cloud Systems (ISTC-VCS). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] 2019. Gloo. https://github.com/facebookincubator/gloo.
[2] 2019. *MLPerf*. https://www.mlperf.org/.
[3] 2019. NCCL. https://developer.nvidia.com/nccl.
[4] 2019. NVLink. https://www.nvidia.com/en-us/data-center/nvlink/.
[5] 2019. PyTorch. https://github.com/pytorch/pytorch.
[6] 2019. PyTorch DDP. https://pytorch.org/docs/stable/_modules/torch/nn/parallel/distributed.html.
[7] 2019. VGG-16 target accuracy using Caffe model. https://gist.github.com/ksimonyan/211839e770f7b538e2d8#gistcomment-1403727.
[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA, 265–283. https://www.tensorflow.org/
[9] Baidu Inc. 2017. Bringing HPC Techniques to Deep Learning. http://research.baidu.com/bringing-hpc-techniques-deep-learning/
[10] Léon Bottou and Olivier Bousquet. 2008. The Tradeoffs of Large Scale Learning. In *Advances in Neural Information Processing Systems*. 161–168.

[11] David L Chen and William B Dolan. 2011. Collecting Highly Parallel Data for Paraphrase Evaluation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*. Association for Computational Linguistics, 190–200.

[12] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). http://arxiv.org/abs/1512.01274

[14] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174* (2016).

[15] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. 2012. Pipelined Back-Propagation for Context-dependent Deep Neural Networks. In *Interspeech*.

[16] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Vol. 14. 571–582.

[17] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2019. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Operating Systems Review* 53, 1 (2019), 14–25.

[18] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. *NIPS ML Systems Workshop* (2017).

[19] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX Annual Technical Conference*. 37–48.

[20] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 4.

[21] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*. 1223–1231.

[22] Michael Denkowski and Alon Lavie. 2014. Meteor Universal: Language Specific Translation Evaluation for Any Target Language. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*. 376–380.

[23] DGX-1 [n. d.]. NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/.

[24] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677* (2017).

[25] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv preprint arXiv:1806.03377* (2018).

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[27] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems*. 1223–1231.

[28] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[29] Zhouyuan Huo, Bin Gu, Qian Yang, and Heng Huang. 2018. Decoupled Parallel Backpropagation with Convergence Guarantee. *ICML-18, arXiv preprint arXiv:1804.10574* (2018).

[30] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient Data Encoding for Deep Neural Network Training. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*.

[31] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205* (2018).

[32] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[33] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML '18)*.

[34] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd SysML Conference, SysML '19*. Palo Alto, CA, USA.

[35] Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[36] Alex Krizhevsky. 2014. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997* (2014).

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[38] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Vol. 1. 3.

[39] Dominic Masters and Carlo Luschi. 2018. Revisiting Small Batch Training for Deep Neural Networks. *arXiv preprint arXiv:1804.07612* (2018).

[40] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and Optimizing LSTM Language Models. *arXiv preprint arXiv:1708.02182* (2017).

[41] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent Neural Network Based Language Model. In *Eleventh Annual Conference of the International Speech Communication Association*.

[42] Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yuefeng Zhou, Naveen Kumar, Rasmus Larsen, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. https://arxiv.org/abs/1706.04972

[43] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*. 693–701.

[44] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[45] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. New York, NY, USA, 2135–2135. https://github.com/Microsoft/CNTK

[46] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-Bit Stochastic Gradient Descent and its Application to Data-Parallel Distributed Training of Speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[47] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. On Parallelizability of Stochastic Gradient Descent for Speech DNNs. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE SPS.

[48] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[49] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. 2015. Automating Model Search for Large Scale Machine Learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 368–380.

[50] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[51] Uber Technologies Inc. 2017. Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow. https://eng.uber.com/horovod/

[52] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990).

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[54] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. 2015. Sequence to sequence-video to text. In *Proceedings of the IEEE International Conference on Computer Vision*. 4534–4542.

[55] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016).

[56] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large Batch Training of Convolutional Networks. *arXiv preprint arXiv:1708.03888* (2017).

[57] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 181–193.