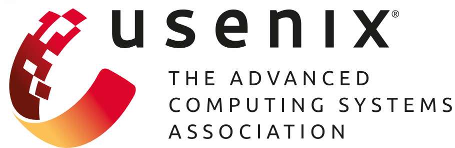


HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism

Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, and Seungmin Lee, *UNIST;* Jaesik Choi, *KAIST;* Sam H. Noh and Young-ri Choi, *UNIST*

https://www.usenix.org/conference/atc20/presentation/park

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism

Jay H. Park[1], Gyeongchan Yun[1], Chang M. Yi[1], Nguyen T. Nguyen[1], Seungmin Lee[1], Jaesik Choi[2], Sam H. Noh[1], and Young-ri Choi[1]

[1]*UNIST* [2]*KAIST*

Abstract

Deep Neural Network (DNN) models have continuously been growing in size in order to improve the accuracy and quality of the models. Moreover, for training of large DNN models, the use of heterogeneous GPUs is inevitable due to the short release cycle of new GPU architectures. In this paper, we investigate how to enable training of large DNN models on a heterogeneous GPU cluster that possibly includes whimpy GPUs that, as a standalone, could not be used for training. We present a DNN training system, *HetPipe* (*Het*erogeneous *Pipe*line), that integrates pipelined model parallelism (PMP) with data parallelism (DP). In HetPipe, a group of multiple GPUs, called a *virtual worker*, processes minibatches in a pipelined manner, and multiple such virtual workers employ data parallelism for higher performance. We also propose a novel parameter synchronization model, which we refer to as Wave Synchronous Parallel (WSP) to accommodate both PMP and DP for virtual workers, and provide convergence proof of WSP. Our experimental results on a given heterogeneous setting show that with HetPipe, DNN models converge up to 49% faster compared to the state-of-the-art DP technique.

1 Introduction

Deep Neural Networks have been popularly used to solve various problems such as image classification [16,29], speech recognition [17], topic modeling [3], and text processing [10]. The size of DNN models (i.e., the number of parameters) have continuously been increasing in order to improve the accuracy and quality of models and to deal with complex features of data [19,47,54,55]. The size of input data and batches used for training have also increased to achieve higher accuracy and throughput [19,26].

For training large DNN models, data parallelism [4, 31, 32, 50], which employs multiple workers using parameter servers or AllReduce communication, and model parallelism [12,28,30], which divides the network layers of a DNN model into multiple partitions and assigns each partition to a different GPU, have commonly been leveraged. Furthermore, to mitigate the critical issue of low GPU utilization of naive model parallelism, pipelined model parallelism, where minibatches are continuously fed to the GPUs one after the other and processed in a pipelined manner, has recently been proposed [19,38].

Table 1: Heterogeneous GPUs

	Year	Archi.	CUDA Core	Boost Clock (MHz)	Memory Size (GB)	Memory BW (GB/sec)
TITAN V	2017	Volta	5120	1455	12	653
TITAN RTX	2018	Turing	4608	1770	24	672
GeForce RTX 2060	2019	Turing	1920	1680	6	336
Quadro P4000	2017	Pascal	1792	1480	8	243

For training DNN models, the use of GPU clusters is now commonplace. In such an environment, the use of heterogeneous GPUs is inevitable due to the short release cycle of new GPU architectures [24]. Moreover, several types of GPUs targeted for high-end servers, workstations, and desktops are being released for purchase [39–42]. Due to their cost-effectiveness, less expensive GPUs targeted for desktops and workstations, rather than high-end servers are also commonly used for machine learning training, especially for small and medium size clusters [14, 21, 49, 56, 57, 59]. Due to the same reason, spot instances with different types of GPUs that are offered by cloud service providers are being used [2,24,36]. Table 1 shows the hardware specifications for four different types of GPUs, along with their market release years, that we have purchased in our institution in the short span of the last three years. Each, at the time of purchase, was (close to) state-of-the-art affordable with what budget we could muster. With technology advancing in such rapid pace, these systems have become outdated. Some of the systems have become old technologies that, individually, are unable to run large DNN models that are common today. Such situations with clusters of heterogeneous GPUs should now be commonplace.

There are benefits to enabling DNN training with heterogeneous resources. First, it allows for large model training with lower-class GPUs. While unable to train individually due to their limited resources, aggregated together, they may be used for training. These GPUs, which likely would have been retired, become usable, possibly used to create (virtual) workers that show similar performance as high-class GPUs. Second, low-class GPUs can be used to improve the performance of even high-class GPUs by incrementally adding on the resources of the (old) lower class systems to the (new) high-class systems. We call a group of aggregated GPUs that could satisfy the resource constraint and be used for training a *virtual worker*. Internally, such a virtual worker could leverage pipelined model parallelism (PMP) to process a minibatch,

while externally, a number of virtual workers could leverage data parallelism (DP) for higher performance.

In this paper, we explore the integration of PMP and DP to maximize the parallelism of DNN model training. In particular, we investigate a DNN model training system, which employs both PMP and DP, for a heterogeneous GPU cluster that possibly includes whimpy GPUs that, as a standalone, could not be used for training large models. Integrating DP to PMP may sound trivial, but in fact, it is quite challenging. In this setting, each virtual worker is continuously processing multiple minibatches in a pipelined manner and thus, all the virtual workers can be in different states. Thus, the key question here is, what weight version should be used by each virtual worker to synchronize with other virtual workers? Numerous questions need to be answered to answer this question: 1) How many new minibatches can start being processed while waiting for global updates from the parameter server? 2) Can synchronization occur at any point of processing the minibatches? 3) How can convergence be guaranteed when such synchronization occurs? 4) What version of parameters is used for the next minibatch while previous minibatches are still executing within each virtual worker? (This question is also considered to some extent in a prior work [38].) And so on. Furthermore, there are also many challenges that need to be overcome to ideally leverage a heterogeneous GPU cluster for DNN training: How are the heterogeneous GPUs to be divided and allocated into virtual workers? How do we reduce virtual worker stragglers when we consider DP? How do we partition the model to maximize the performance of PMP using heterogeneous GPUs?

While DP [4, 31, 32, 50], PMP [19, 38], and heterogeneity [24, 25, 33] for training have been considered separately, to the best of our knowledge, this is the first paper that tackles these issues together in attempting to answer *some* of the aforementioned questions. In this work, we design a DNN training system, *HetPipe* (*Het*erogeneous *Pipe*line), that integrates PMP of a virtual worker, which is composed of multiple (possibly whimpy) heterogeneous GPUs, with DP of virtual workers using parameter servers to enable and also speed up training of large models. HetPipe can aggregate heterogeneous resources from multiple GPUs to form a virtual worker such that the performance of each virtual worker is similar to each other, reducing the straggler problem. For HetPipe, we propose a novel parameter synchronization model, which we refer to as Wave Synchronous Parallel (WSP). WSP is adapted from the Stale Synchronous Parallel (SSP) model [18] to accommodate both PMP and DP for multiple virtual workers unlike existing synchronization models. We also prove the convergence of WSP. Note that while HetPipe would work in a homogeneous GPU cluster in training a large model that cannot be loaded into the memory of a single GPU, with the rapid turnaround of newer GPU architectures, it is more likely that one will end up with a cluster of heterogeneous GPUs. This is the environment that we target.

We implement HetPipe by modifying TensorFlow, a commonly used machine learning training system. We evaluate the performance of HetPipe for two DNN models using a heterogeneous GPU cluster composed of four different types of GPUs. Our experimental results demonstrate that the performance of HetPipe is better than that of the state-of-the-art DP via Horovod [50] that uses AllReduce communication [45]. This is because HetPipe mitigates the straggler problem, and also because it enables each virtual worker and the parameter server to intra-communicate for all parameter updates, significantly reducing communication overhead. Compared to Horovod, the convergence of VGG-19 with a large parameter set to a desired accuracy becomes 49% faster, and that of ResNet-152 which is too big to be loaded in four whimpy GPUs in our cluster becomes 39% faster by using all the GPUs (including whimpy ones).

Strategies to leverage PMP have been explored in previous studies [7, 19, 27, 38]. Compared to these, our study makes forward strides in three aspects. First, we generalize PMP of a virtual worker to be used together with DP of virtual workers, increasing the parallelism of DNN model training. Consequently, this results in speeding up training. Second, we consider a heterogeneous GPU cluster, which allows the use of GPUs, which otherwise, could not be used for training. Finally, we present a parameter synchronization model that guarantees convergence, of which we provide a proof, for training models using PMP with DP. We provide a more in-depth comparative discussion on these studies in Section 2.2.

2 Background

2.1 Data Parallelism

Training of a DNN model is processed by a *forward pass* followed by a *backward pass* for each *minibatch*, which is a subset of training samples, in a popularly used *stochastic gradient descent* (SGD) method. For each minibatch, the weight updates, i.e., *gradients*, are computed to update weights (or parameters) w of the model.

Data parallelism (DP) utilizes multiple workers to speed up training of a DNN model. It divides the training dataset into subsets and assigns each worker a different subset. Each worker has a replica of the DNN model and processes each minibatch in the subset, thereby computing the weight updates. Therefore, if a DNN model cannot be loaded into the memory of a single GPU, DP cannot be used.

Among the multiple workers, the parameters are synchronized using parameter servers [31] or AllReduce communications [32, 50]. For *Bulk Synchronous Parallel* (BSP) [1, 35], each worker must wait for all other workers to finish the current minibatch p before it starts to process the next minibatch $p+1$ so that it can use an updated version of the weights for minibatch $p+1$. For *Asynchronous Parallel* (ASP) [1, 48], each worker need not wait for other workers to finish minibatch p, possibly using a stale version of the weights. With BSP, which is possible for both the parameter servers and

AllReduce communications, the system may suffer from high synchronization overhead, especially in a heterogeneous GPU cluster where each worker with a different GPU provides different training performance [33]. On the other hand, while ASP, which is possible for the parameter servers, has no synchronization overhead, it is known that ASP does not ensure convergence [48, 58].

A method that takes the middle ground between BSP and ASP is *Stale Synchronous Parallel* (SSP) [18]. With SSP, each worker is allowed to proceed the training of minibatches using a *stale* version of the weights that may not reflect the most recent updates computed by other workers. Thus, workers need not synchronize with other workers whenever it finishes the processing of a minibatch. As such, parameter staleness can occur. However, this staleness is bounded as defined by the user and referred to as the *staleness threshold*. As SSP is beneficial when worker performance is varied, it has been explored especially in the context of heterogeneous systems [24].

In SSP, each worker periodically pushes the weight updates to the parameter server. This synchronization interval is called a *clock*. Thus, each worker increases its local clock by one for every iteration, which is the training period of a minibatch. For a given staleness threshold s where $s \geq 0$, each worker with clock c is allowed to use a stale version of the weights, which includes all the updates from iteration 0 to $c-s-1$ and, possibly, more recent updates past iteration $c-s-1$. That is, a worker can continue training of the next minibatch with parameters whose updates may be missing from up to the s most recent minibatches.

2.2 Model Parallelism and Pipeline Execution

Model parallelism (MP) is typically exploited for large DNN models that are too large to be loaded into memory of a single GPU. In particular, a DNN model composed of multiple layers is divided into k partitions and each partition is assigned to a different GPU. Each GPU executes both the forward and backward passes for the layers of the assigned partition. *Note that it is important to execute the forward and backward passes of a partition on the same GPU* as the activation result computed for the minibatch during the forward pass needs to be kept in the GPU memory until the backward pass of the same minibatch for efficient convergence, as similarly discussed by Narayanan and others [38]. Otherwise, considerable extra overhead will incur for managing the activation through either recomputation or memory management.

In the basic form of MP, k GPUs, individually, act as one *virtual worker* to process a minibatch as follows: For each minibatch, execution of the forward pass starts from GPU_1 up to GPU_k. When each GPU_i, where $1 \leq i < k$, completes the forward pass of the assigned partition, it sends the computed activations of *only the last layer in its partition* to GPU_{i+1}. Once GPU_k finishes the forward pass of its partition, the backward pass of the minibatch is executed from GPU_k down to GPU_1. When each $GPU_{i'}$, where $1 < i' \leq k$, finishes the backward pass, it sends the computed local gradients of *only the first layer in its assigned partition* to $GPU_{i'-1}$. This basic form of MP results in low GPU utilization as only one GPU is actively executing either the forward or backward pass. Nonetheless, MP allows execution of large DNN models that are too large for a single GPU.

Table 2: Comparison of HetPipe with GPipe and PipeDream

	GPipe	PipeDream	HetPipe
Heterogeneous Cluster Support	No	No	Yes
Target Large Model Training	Yes	No	Yes
Number of (Virtual) Workers	1	1	N
Data Parallelism	Extensible	Partition	Virtual Workers
Proof of Convergence	Analytical	Empirical	Analytical

To improve utilization of the GPUs in a virtual worker, minibatches can be processed in a pipelined manner. The subsequent minibatches are fed into the first GPU in MP (i.e., GPU_1) one by one once the GPU completes the processing of the previous minibatch. This allows for multiple GPUs to simultaneously execute either the forward or backward pass of their assigned layers for different minibatches. This is referred to as Pipelined Model Parallelism (PMP).

This PMP strategy has been investigated in previous studies [19, 38]. PipeDream exploits PMP of a single virtual worker to avoid the parameter communication overhead of DP [38]. Considering only homogeneous GPUs, when PipeDream partitions a model into stages to maximize pipeline performance, it does not take into account the memory requirement of a GPU that depends on the stage of a pipeline. Thus, PipeDream processes a limited number of minibatches, which is large enough to saturate the pipeline, to reduce memory overhead. PipeDream also provides a form of DP, but it considers DP within a virtual worker to speed up the execution of lagging layers. No proof of single pipeline convergence is provided in PipeDream. Note that without a parameter synchronization model such as WSP, it is not possible to properly run DP over multiple PipeDream virtual workers via parameter servers or AllReduce communication.

GPipe is a scheme that leverages PMP of a single virtual worker to support large DNN models, also in a homogeneous GPU cluster [19]. In GPipe, a minibatch is divided into multiple *microbatches* that are injected into the pipeline. Using the same weights, GPipe executes the forward passes for all the microbatches, and then executes the backward passes for them. When the backward pass of the last microbatch is done, it updates the weights all together for the minibatch. GPipe incurs frequent pipeline flushes, possibly resulting in low GPU utilization [38]. In GPipe, DP of multiple virtual workers can be done using existing synchronization schemes like BSP as a virtual worker processes one minibatch at a time. GPipe saves on GPU memory by recomputing the activations again in the backward pass instead of keeping the activations computed in the forward pass in memory. We do not use this optimization though there are no fundamental reasons forbidding it. A comparison of HetPipe with previous studies is given in Table 2.

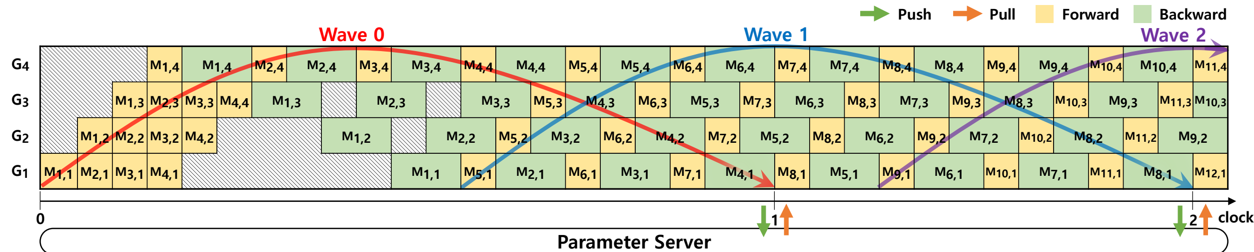


Figure 1: Pipeline execution of minibatches where $M_{p,k}$ indicates the execution of a minibatch p in partition k, which is executed in GPU$_k$ and the yellow and green colors indicate the forward and backward passes, respectively.

3 System Overview

The system that we propose focuses on training a large DNN model in a *heterogeneous GPU cluster* composed of various types of GPUs that have different computation capability and memory capacity. In such settings, for some types of GPUs in the cluster, the DNN model of interest may be too large to be loaded into the memory of a single GPU. The system that we propose in this paper leverages both pipelined model parallelism (PMP) and data parallelism (DP) to enable training of such large DNN models and, in the process, enhance performance as well as the utilization of the heterogeneous GPU resources of the cluster.

Figure 2 shows the architecture of the proposed cluster system composed of H nodes. Each node comprises a homogeneous set of GPUs, but the GPUs (and memory capacity) of the nodes themselves can be heterogeneous. Two key novelties exist in this architecture. First, DP is supported through a notion of a *virtual worker* (VW), which consists of k, possibly heterogeneous, GPUs, and encapsulates the notion of a worker in typical DNN systems. That is, a virtual worker is used to train the DNN model. In Figure 2, note that there are N virtual workers with 4 GPUs each, that is, $k = 4$, and that the GPUs comprising the virtual worker may be different for each virtual worker. While in this paper we consider k to be constant for each virtual worker, our design does not restrain it to be so; this is simply a choice we make for simplicity. The key aspect here is that a virtual worker allows DP by aggregating GPUs possibly even when individual GPUs may be resource limited.

The second novelty is that each virtual worker processes each minibatch based on model parallelism, in a pipelined manner, to fully utilize the GPU resources, as shown in Figure 1, to accommodate large DNN models. While PMP has been proposed before (which we compare in Section 2.2), to the best of our knowledge, we are the first to present PMP in a heterogeneous setting. We refer to our system as *HetPipe* as it is *het*erogeneous, in GPUs, across and, possibly, within virtual workers and makes use of *pipe*lining in virtual workers for resource efficiency.

To train DNN models based on pipelined model parallelism in virtual workers, the *resource allocator* first assigns k GPUs to each virtual worker based on a resource allocation policy

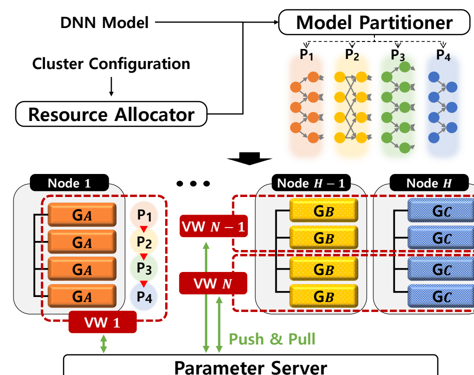


Figure 2: System architecture (VW: Virtual Worker)

(which will be discussed in Section 8.1). Note that for allocating the heterogeneous GPUs to the virtual workers, the resource allocation policy must consider several factors such as the performance of individual GPUs as well as the communication overhead caused by sending activations and gradients within a virtual worker, and synchronizing the weights among the virtual workers and the parameter server. Then, for the given DNN model and allocated k GPUs, the *model partitioner* divides the model into k partitions for the virtual worker such that the performance of the pipeline executed in the virtual worker can be maximized.

As any typical DP, multiple virtual workers must periodically synchronize the global parameters via parameter servers or AllReduce communication; in HetPipe, parameter servers are used to maintain the global weights. Each virtual worker has a local copy of the global weights and periodically synchronizes the weights with the parameter server. Evidently, when managing the weights within a virtual worker and across virtual workers, two types of staleness, *local staleness* and *global staleness*, need to be permitted to improve the performance of DNN training. Local staleness refers to staleness within a virtual worker. As each virtual worker processes minibatches in a pipelined manner, there are multiple minibatches that are being processed in parallel. Thus, staleness is inevitable as weights seen by a minibatch may not reflect the updates of all of its previous minibatches.

Global staleness, on the other hand, is similar to the staleness notion introduced by Ho et al. [18]. That is, the system needs to reduce communication overhead between the param-

eter server and (virtual) workers, and, in our case, also mitigate the synchronization overhead caused by possibly heterogeneous virtual workers. Therefore, similarly to SSP [18], each virtual worker should be allowed to proceed training without querying the global weights for every minibatch, unless its local copy is so old such that there are too many missing recent updates made by other virtual workers. Note that such staleness condition is set by the user [18].

For our system, we propose the *Wave Synchronous Parallel* (WSP) model to synchronize the weights. A *wave* is a sequence of minibatches that are processed concurrently in a virtual worker. Let the number of minibatches in a wave be N_m. Within a wave, processing of the i-th minibatch is allowed to proceed without waiting for the preceding minibatchs i' to be completed, where $1 < i \leq N_m$ and $1 \leq i' < i$. That is, there is no dependency among the weights used by minibatches in the same wave. As the virtual worker does not enforce the updates even from the first minibatch in a wave to be reflected in the weights used by the last minibatch, the local staleness threshold in WSP is $N_m - 1$. Moreover, *each virtual worker only pushes the aggregated updates from all the minibatches in a wave, instead of for every minibatch, to the parameter server.* This results in considerable reduction in communication overhead.

As it is important that the results generated through our proposed system configuration are correct [18, 24, 60], we show the convergence of our methodology in Section 6.

Note that HetPipe uses parameter servers, which may incur synchronization and communication overhead. However, HetPipe mitigates such overhead by permitting global staleness among virtual workers and executing the pipeline in each virtual worker such that it continues to process minibatches that have already been injected while waiting for the parameter update. We believe HetPipe can be further optimized by taking decentralized approaches, but leave this for future work.

4 Pipelined Model Parallelism Within a VW

Number of Minibatches in the Pipeline: In our system, each virtual worker processes up to N_m minibatches concurrently in a pipeline manner so that the executions of the minibatches can overlap. Given a DNN model and k GPUs, the maximum number of minibatches executed concurrently in the virtual worker, Max_m, is basically determined by the memory requirement for training the model. For a model that requires a huge amount of memory for output activations and weights, Max_m may be less than k. Note that in such cases, the utilization of each GPU is unlikely to be high.

N_m, the actual number of minibatches in the pipeline will be $N_m \leq Max_m$ and basically determined by considering the throughput of the pipeline. Note that N_m must be the same in every virtual worker, and thus, N_m is set to the minimum Max_m among all the virtual workers. N_m will affect the local staleness that we discuss later in this section.

Model Partitioning: To train a DNN model, a set of k GPUs is allocated to a virtual worker by a resource allocation policy, which we discuss in Section 8.1. For now, let us assume that k, the number of possibly heterogeneous GPUs, and N_m are given. Then, a partitioning algorithm is employed to divide multiple layers of the model into k partitions, assigning them to the k different GPUs. The goal of the partitioning algorithm is to maximize the performance of the pipeline, while satisfying the memory requirement of each partition to process N_m minibatches.

In particular, in this study, for memory, we consider the fact that the actual memory requirement will vary depending on the stage of the pipeline that the GPU is used for. For example, contrast GPU_4 and GPU_1 in Figure 1. GPU_4, the GPU that handles the last stage of the pipeline, handles only one minibatch at a time and is immediately done with the minibatch as exemplified by the yellow (forward pass) and green (backward pass) $M_{i,4}$ pairs for $i = 1, 2, ...$, that are side-by-side. In contrast, for GPU_1, the yellow and green $M_{i,1}$ pairs are far apart, meaning that the forward pass $M_{i,1}$ needs to hold up memory until the backward pass $M_{i,1}$ is finished with its execution. Thus, with GPU_1, the memory requirement is high as it needs to hold on to the results of the forward pass for all stages of the pipeline. This variance in memory requirement is considered in partitioning the layers.

Execution time must also be considered when partitioning the layers. To do so, we calculate the execution time of a partition to be the sum of the computation time of all the layers in the partition and the communication time needed for receiving the activations (in the forward pass) and local gradients (in the backward pass). Our partitioning algorithm attempts to minimize the maximum execution time of the partitions within the bounds of satisfying the memory requirement.

Partition Scheduling: Once the partition is set, the partitions need to be scheduled for each of the GPUs. Each GPU_q responsible for partition q may have multiple forward pass and backward pass tasks to schedule at a time. Each GPU schedules a task by enforcing the following conditions:

1. A forward pass task for a minibatch p will be executed only after a forward pass task for every minibatch p' is done where $1 \leq p' < p$.
2. Similarly, a backward pass task for a minibatch p will be executed only after a backward pass task for every minibatch p' is done where $1 \leq p' < p$.
3. Among multiple forward and backward pass tasks, a FIFO scheduling policy is used.

Note that in the last partition, for a minibatch, processing a forward pass immediately followed by a backward pass is executed as a single task.

Considering Staleness: Given the description of pipelining, the question of staleness of weights used needs to be considered. That is, as a minibatch is scheduled, it may be that the layers are not using the most up-to-date weights. For example, in Figure 1, when the forward pass $M_{2,1}$, the second minibatch, begins to be processed, it must use stale weights as

the first minibatch has not completed and hence, the changes in the weights due to the first minibatch have not yet been appropriately reflected, which is in contrast with typical processing where minibatches are processed one at a time. We now discuss how this staleness issue is considered.

Let *local staleness* be the maximum number of missing updates from the most recent minibatches that is allowed for a minibatch to proceed in a virtual worker. As training with N_m minibatches can proceed in parallel in a virtual worker, the local staleness threshold, s_{local}, is determined as $N_m - 1$, where $1 \leq N_m \leq Max_m$. If $N_m = 1$, the behavior is exactly the same as naive model parallelism. Larger N_m may improve the performance (i.e., throughput) of the pipeline as a larger number of concurrent minibatches are executed, but local staleness increases, possibly affecting the convergence of training. In a real setting, typically, N_m will not be large enough to affect convergence as it will be bounded by the total amount of GPU memory of a virtual worker.

Such local staleness also exists in PipeDream [38]. As PipeDream basically employs weight stashing that uses the latest version of weights available on each partition to execute the forward pass of a minibatch, a different version of weights is used across partitions for the same minibatch. Unfortunately, PipeDream only shows empirical evidence of convergence when weight stashing is used. Note that PipeDream also discusses vertical sync, which is similar to HetPipe, but it excludes vertical sync in its evaluations [38].

Now let w_p be the weights used by minibatch p. Then, initially, we can assume that w_0, the initial version of weights, is given to the virtual worker. Then, the first $(s_{local}+1)$ minibatches are processed in a pipelined manner with $w_0 = w_1 = \cdots = w_{s_{local}} = w_{s_{local}+1}$.

To accommodate staleness in our system, when processing of minibatch p completes, the virtual worker updates the local version of the weights, w_{local} as $w_{local} = w_{local} + u_p$, where u_p is the updates computed by processing minibatch p. Therefore, in HetPipe, weights are not updated layer by layer and w_{local} is a consistent version of weights across partitions. When the virtual worker starts to process a new minibatch, it makes use of the latest value of w_{local} without waiting for the other minibatches to update their weights. For example, once the virtual worker is done for minibatch 1 and updates w_{local} with u_1, it will start to process minibatch $s_{local}+2$ by using the updated weights without waiting for minibatches 2 up to $s_{local}+1$ to be completed. Similarly, when the virtual worker is done with minibatch $s_{local}+1$ and updates w_{local} with $u_{s_{local}+1}$, it will start to process minibatch $2 \times (s_{local}+1)$ without waiting for the previous most recent s_{local} minibatches to be completed. Therefore, except for the initial minibatches 1 to $s_{local}+1$, for minibatch p the virtual worker will use the version of the weights that reflects (at least) all the local updates from minibatches 1 to $p-(s_{local}+1)$. Note that for every minibatch p, w_p must be kept in GPU memory until the backward pass for p is executed.

Note that staleness in SSP is caused by the different processing speed of minibatches among multiple workers. Thus, in SSP, staleness is used as a means to reduce the synchronization and communication overhead. However, local staleness in HetPipe is caused inherently as minibatches are processed in a pipelined manner within a virtual worker.

5 Data Parallelism with Multiple VWs

In this section, we discuss data parallelism (DP) with virtual workers. The first and foremost observation of DP being supported with virtual workers is that the virtual workers may be composed of (whimpy) heterogeneous GPUs. While it is well known that DP helps expedite DNN execution, DP, in typical systems, is not possible if individual GPUs, that is, workers, do not have sufficient resources to handle the DNN model, in particular, large DNNs. By allowing a virtual worker to be composed of multiple GPUs that are lacking in resources, our system allows DP even with whimpy GPUs. The other key observation in properly supporting DP with virtual workers is that each virtual worker now retains local staleness as discussed in Section 4. Making sure that, despite such individual staleness, we understand and show that the results obtained from DP among virtual workers (globally) converge is an important issue that must be addressed. The rest of the section elaborates on this matter.

Workings of WSP: As stated in the system overview, HetPipe uses parameter servers. We assume that such synchronization occurs in *clock* units, a notion taken from SSP [18]. Precisely, a clock unit is defined as the progress of completing one wave. Recall from Section 3 (and Figure 1) that a wave is a sequence of $s_{local}+1$ minibatches concurrently executed such that a virtual worker is allowed to process a later minibatch in a wave without updates from an earlier minibatch in the same wave.

Similarly to SSP (which, however, considers the staleness of weights only in DP), each virtual worker maintains a local clock c_{local}, while the parameter server maintains a global clock c_{global}, which holds the minimum c_{local} value of all the virtual workers. Initially, the local clocks and the global clock are 0. At the end of every clock c, each virtual worker completes the execution of all the minibatches in wave c. At this point, the virtual worker computes the aggregated updates from minibatch $c \times (s_{local}+1)+1$ to minibatch $(c+1) \times (s_{local}+1)$ and pushes the updates $\tilde{u}$ to the parameter server. We see that, similar to in SSP [18], $\tilde{u}$ is synchronized with a clock value c. For example, as shown in Figure 1 where $s_{local} = 3$, at the end of clock 0, the virtual worker pushes the aggregated updates of wave 0, which is composed of minibatches from 1 to 4, and at the end of clock 1, the aggregated updates of wave 1, which is composed of minibatches from 5 to 8, and so on. It is important to note that in WSP, the virtual worker pushes $\tilde{u}$ to the parameter server for every wave, instead of pushing $\tilde{u}$ for every minibatch, which will significantly reduce the communication overhead.

When the parameter server receives the updates ũ from the virtual worker, the parameter server updates the global version of the weights as $w_{global} = w_{global} + \tilde{u}$. Note that the parameter server updates its c_{global} to $c+1$ only after every virtual worker has pushed the aggregated updates of wave c.

In WSP, each virtual worker is allowed to proceed training without retrieving the global weights for every wave. Thus, the virtual worker may use a weight version that, from a global standpoint, may be stale, as the most recent updates received by the parameter servers may not be reflected in its local version of the weights. We discuss how global staleness among the virtual workers is bounded.

Global Staleness Bound: Let *clock distance* be the difference in c_{local} between the fastest and slowest virtual workers in the system. Therefore, a virtual worker with local clock c, where $c \geq D+1$, must use a version of the weights that includes all the (aggregated) updates from wave 0 up to $c-D-1$. Also, the weight version may include some recent global updates from other virtual workers and some recent local updates within the virtual worker beyond wave $c-D-1$.

When a virtual worker pulls the global weights at the end of clock c to maintain this distance, it may need to wait for other virtual workers to push their updates upon completion of wave $c-D$. However, while a virtual worker waits for other virtual workers to possibly catch up at the end of clock c, local processing is allowed to proceed with s_{local} minibatches of wave $c+1$ as the minibatches are executed in a pipelined manner. Take, for example, the case when $D=0$ and $s_{local}=3$ in Figure 3 (and Figure 1). As a virtual worker, VW1, completes minibatch 4, it computes the aggregated updates ũ for wave 0 (composed of minibatches 1 to 4) and pushes ũ to the parameter server. VW1 now waits for the other virtual workers to complete wave 0 before proceeding with minibatch 8. However, note that as shown in the figure, VW1 has already started to process minibatches 5, 6 and 7, which belong to wave 1, while its local clock is still 0. Similarly, once it completes minibatch 8, it pushes the aggregated updates ũ for wave 1 (composed of minibatches 5 to 8) to the parameter server; in the meantime, it has already started processing minibatches 9, 10, and 11, which belong to wave 2, while its clock is still 1.

Note that this processing of local minibatches in the virtual worker does not violate the local staleness bound. Note also that when $D=0$, each virtual worker must wait for each other at the end of every clock to synchronize the weights for every wave, which is BSP-like behavior with pipelined execution in each virtual worker.

Now let us define the *global staleness bound*, s_{global}, to be the maximum number of missing updates from the most recent minibatches, *globally computed by all the other virtual workers in the system*, that is allowed for a minibatch to proceed in a virtual worker. We want to identify s_{global} based on our discussion so far. This will allow each virtual worker to determine whether it can proceed with its current minibatch.

Initially, all virtual workers start processing the first $(D+1)$

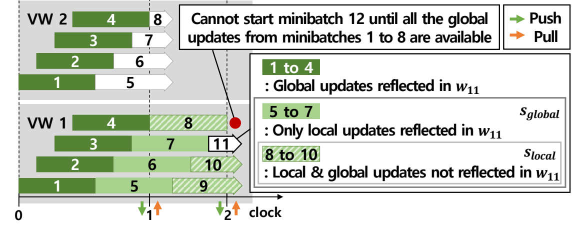


Figure 3: Local and global staleness with WSP

waves without querying the global weights from the parameter server. Furthermore, they can start to process up to s_{local} minibatches of the next wave before receiving the global weights that include the recent updates as discussed above. Therefore, for those initial minibatches, the virtual worker uses w_0 or a weight version that may include some recent local updates.

For any minibatch p thereafter, that is, where $p > (D+1) \times (s_{local}+1) + s_{local}$, p must use a weight version that reflects, at the very least, all the global updates from all the other virtual workers from minibatch 1 to minibatch $p - (s_{global}+1)$, where $s_{global} = (D+1) \times (s_{local}+1) + s_{local} - 1$. The first term of this equation is due to the fact that a virtual worker is allowed to proceed with the next $(D+1)$ waves (i.e., $(D+1) \times (s_{local}+1)$ minibatches), and the second term is due to the additional s_{local} minibatches that can be started because of pipelined execution. Continuing with the example in Figure 3, where $D=0$ and $s_{local}=3$, VW1 proceeds the training of minibatch 11 without the global and/or local updates from wave 1 (minibatches 5 to 8) or the two local updates from minibatches 9 and 10 (i.e., having $s_{global}=6$). Thus, it must have a version of the weights that includes all the global updates from minibatches 1 to 4. Actually, the weight version used for minibatch 11 includes three local updates from minibatches 5, 6, and 7, along with all the global updates from wave 0. In case of minibatch 12, it cannot start the training until global updates up to minibatch 8 are received.

6 Convergence Analysis

In this section, we discuss the convergence property of the WSP model. Let N be the number of virtual workers and $u_{n,p}$ be the update of worker n at minibatch execution p. Let $s_g = s_{global}$ and $s_l = s_{local}+1$, and following the analysis of [18], the noisy weight parameter[1] $\tilde{w}_{n,p}$ for worker n at minibatch execution p, is decomposed into

$$\tilde{w}_{n,p} = w_0 + \left[\sum_{n'=1}^{N} \sum_{p'=1}^{p-s_g-1} u_{n',p'}\right] + \left[\sum_{p' \in \mathcal{C}_{n,p}} u_{n,p'}\right] + \left[\sum_{(n',p') \in \mathcal{E}_{n,p}} u_{n',p'}\right]. \quad (1)$$

[1] In this section, we use the term 'weight parameter' to denote all weights of a network. Thus, the weight parameters refer to a set of weights of networks.

Here w_0 refers to the initial parameter, and the noisy weight parameter has three other terms which respectively include

1. updates of all workers (guaranteed to be included) to process minibatch execution p,
2. $\mathcal{C}_{n,p} \subseteq [p-s_g, p-1]$: the index set of the latest updates of the querying worker n in the range of current global staleness bound, and
3. $\mathcal{E}_{n,p} \subseteq ([1,N]\backslash\{n\}) \times [p-s_g, p+s_g+s_l]$: the index set of extra updates of other workers in the range of the current global staleness bound. When execution p is not at a synchronization point, $\mathcal{E}_{n,p} = \emptyset$.

We define $\{u_t\}$ as the sequence of updates of each virtual worker after processing each minibatch and $\{w_t = w_0 + \sum_{t'=0}^{t-s_lN} u_{t'}\}$ as the reference sequence of weight parameters, where $u_t := u_{t \bmod N, \lfloor t/N \rfloor + t \bmod s_l}$, in which we loop over the workers ($t \bmod N$) and over each update after a minibatch execution inside a worker ($\lfloor t/N \rfloor + t \bmod s_l$). Here, s_lN ($= s_l \times N$) is the number of total minibatch updates in one wave from all virtual workers. Since a virtual worker uses a version of the weight parameter that reflects all the local updates from minibatch 1 to $p - s_l$ for worker p, the reference and noisy sequences at iteration t are updated up to $t - s_lN$. The set $\mathcal{E}_t$ and the noisy weight parameter $\tilde{w}_t$ are defined similarly and the difference between w_t and $\tilde{w}_t$ is $\tilde{w}_t = w_t - \left[\sum_{i\in\mathcal{R}_t} u_i\right] + \left[\sum_{i\in\mathcal{Q}_t} u_i\right]$ where $\mathcal{R}_t$ is the index set of missing updates in the reference weight parameter but not in noisy weight parameter, and $\mathcal{Q}_t$ is the index set of extra updates in the noisy weight parameter but not in reference weight parameter.

After T updates, we represent the target function as $f(w) := \frac{1}{T}\sum_{t=1}^{T} f_t(w)$, the regret of two functions with $\tilde{w}_t$, the parameter learned from the noisy update, and w^*, the parameter learned from the synchronized update is $R[W] := \frac{1}{T}\sum_{t=1}^{T} f_t(\tilde{w}_t) - f(w^*)$.

Thus, when we bound the regret of the two functions, we can bound the error of the noisy updates incurred by the distributed pipeline staleness gradient descent. We first bound the cardinality of $\mathcal{R}_t$ and $\mathcal{Q}_t$ in the following lemma.

Lemma 1. *The following two inequalities,* $|\mathcal{R}_t| + |\mathcal{Q}_t| \leq (2s_g + s_l)(N-1)$ *and* $\min(\mathcal{R}_t \cup \mathcal{Q}_t) \geq \max(1, t - (s_g + s_l)N)$*, hold.*

Proof. Since $\mathcal{Q}_t \subseteq \mathcal{E}_t$ and $\mathcal{R}_t \subseteq \mathcal{E}_t \backslash \mathcal{Q}_t$, $|\mathcal{R}_t| + |\mathcal{Q}_t| \leq |\mathcal{E}_t| \leq (2s_g + s_l)(N-1)$. The second claim follows from $\mathcal{E}_t \supseteq \mathcal{R}_t \cup \mathcal{Q}_t$. □

With the following two assumptions, the proof of convergence generally follows Qirong et al. [18][2]

Assumption 1. ***(L-Lipschitz components)*** *For all t, the component function f_t is convex and has bounded subdifferential $\|\nabla f_t(w)\| \leq L$, in which $L > 0$ is a constant.*

Assumption 2. ***(Bounded distances)*** *For all w, w', the distance between them is bounded $D(w\|w') \leq M$, in which $M > 0$ is a constant.*

[2]The full proof is omitted due to space, but can be found in [44].

We also denote $\frac{1}{2}\|w - w'\|^2$ as $D(w\|w')$. Then, we can bound the regret of the function trained with our noisy distributed, pipeline update as in Theorem 1.

Theorem 1. *Suppose w^* is the minimizer of $f(w)$. Let $u_t := -\eta_t \nabla f_t(\tilde{w}_t)$ where $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{M}{L\sqrt{(2s_g+s_l)N}}$, in which M, L are the constants defined in the assumptions. Then the regret is bounded as $R[W] \leq 4ML\sqrt{\frac{(2s_g+s_l)N}{T}}$.*

Our theoretical results are similar with existing work on non pipelined version of staleness update [18, 24]. However, we reflect the new characteristics of distributed pipeline staleness update in Lemma 1, and thus in Theorem 1.

7 Partitioning Algorithm

Recall that the goal of our partitioning algorithm is to minimize the maximum execution time of the partitions within the bounds of satisfying the memory requirement. To obtain a performance model to predict the execution time of each layer of a model in a heterogeneous GPU, we first profile the DNN model on each of the different types of GPUs in a cluster, where we measure the computation time of each layer of the model. For GPU memory usage, we measure the usage of each layer (by using the logging feature of TensorFlow) on only one GPU type (as it is roughly the same for all GPU types). For profiling the memory usage on a whimpy node, we measure the memory usage of each layer using a small batch size and then multiply it for the target batch size. To compute the memory requirement for a given partition, we take into account the total memory usage to store the data to process the layers as well as the maximum number of minibatches concurrently assigned to the partition.

For communication time between layers in the model, we first derive the amount of input data for each layer in the forward and backward pass from the model graph. For the given data size, we predict intra-node communication based on the PCI-e bandwidth, then multiply it by a scaling-down constant (which is similarly done in Paleo [46]), since in practice, it is not possible to utilize the peak bandwidth. The scaling-down constant is derived by running a synthetic model that sends various sizes of data from one GPU to another GPU in the same node. For inter-node communication (via InfiniBand), we use linear regression to estimate the communication time for the given data size. To build a prediction model, we collect 27 samples by training two DNN models, used in our experiments, with arbitrary partitions. Note that in this work, the heterogeneity of network performance such as slow network links is not considered (as in [33]). However, for such cases, we can extend our partitioning algorithm to consider different network performance between two nodes when estimating the communication time. Also, a model that estimates the memory requirement for each stage more accurately will be helpful in partitioning a DNN model in a more balanced manner.

To find the best partitions of a DNN model, we make use of CPLEX, which is an optimizer for solving linear programming problems [20]. The memory requirement for each partition on the pipeline to support N_m concurrent minibatches is provided as a constraint to the optimizer. The algorithm will return partitions for a model with a certain batch size only if it finds partitions that meet the memory requirement for the given GPUs. Also, the optimizer checks all the different orders of the given heterogeneous GPUs for a single virtual worker to partition and place layers of the DNN model on them.

8 Experimental Results

8.1 Methodology

Heterogeneous GPU cluster: In our experiments, we use four nodes with two Intel Xeon Octa-core E5-2620 v4 processors (2.10 GHz) connected via InfiniBand (56 Gbps). Each node has 64 GB memory and 4 homogeneous GPUs. Each node is configured with a different type of GPU as shown in Table 1. Thus, the total number of GPUs in our cluster is 16. Each GPU is equipped with PCIe-3×16 (15.75 GB/s). Ubuntu 16.04 LTS with Linux kernel version 4.4 is used. We implement HetPipe based on the WSP model by modifying TensorFlow 1.12 version[3] with CUDA 10.0 and cuDNN 7.4.

DNN models and datasets Our main performance metric is throughput (images/second) of training a DNN model. We use ResNet-152 [16], and VGG-19 [51] with ImageNet [13]. For each DNN model, batch size of 32 is used. For all other hyperparameters, we use the default settings as specified in the benchmark [52] of ResNet-152 and VGG-19.

Resource allocation for virtual workers: Given any heterogeneous GPU cluster, there can be many ways of allocating the resources to the multiple virtual workers. For our experiments, we consider allocation policies within the bounds of our platform. Thus, given the 16 GPUs, HetPipe employs four virtual workers, each of which is configured with four GPUs, along the following three allocation policies.

Node Partition (NP): This policy assigns a node per virtual worker. Thus, each virtual worker is composed of homogeneous GPUs. Consequently, as the nodes are heterogeneous, partitioning of layers for a DNN model is different for each virtual worker. NP results in minimum communication overhead within each virtual worker as communication between GPUs occurs within the same node via PCI-e, rather than across multiple nodes where communication is via InfiniBand. On the other hand, as the performance of each virtual worker varies, a straggler may degrade performance with DP.

Equal Distribution (ED): This policy evenly distributes GPUs from each node to every virtual worker. Thus, every virtual worker is assigned four different GPUs, but every virtual worker has the exact same resources. Thus, model partitioning is the same, and thus, performance will be the same across the virtual workers, which mitigates the straggler problem. However, ED results in high communication overhead within each virtual worker.

[3] Modified LOC is ∼1.5K in the TensorFlow framework and TensorFlow benchmark codes, where most features are added as independent functions.

Table 3: Resource allocation for the three policies considered

	Node Partition	Equal Distribution	Hybrid Distribution
VW1	VVVV	VRGQ	VVQQ
VW2	RRRR	VRGQ	VVQQ
VW3	GGGG	VRGQ	RRGG
VW4	QQQQ	VRGQ	RRGG

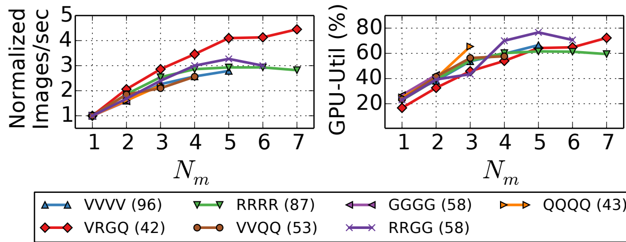
Hybrid Distribution (HD): This policy is a hybrid of NP and ED. For our cluster, a combination of two GPU types are allocated to each virtual worker such that their performances in terms of aggregated computation capability and amount of GPU memory are similar to each other. This choice is made to mitigate the straggler problem while reducing the communication overhead within each virtual worker. As, in terms of computation power, V > R > G > Q and, in terms of the amount of the GPU memory, R > V > Q > G, two virtual workers are allocated VVQQ, while the other two are allocated RRGG, where V, R, G and Q refers to TITAN V, TITAN RTX, GeForce RTX 2060, and Quadro P4000, respectively.

Table 3 shows the resource allocation of each virtual worker for the three resource allocation policies.

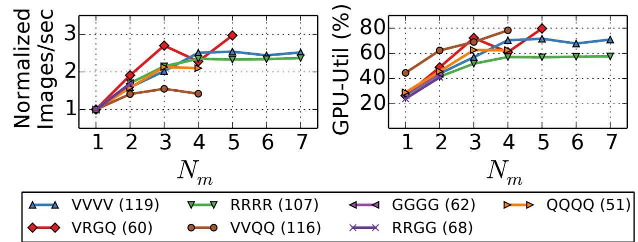
Parameter Placement: In our experiments, for DP, we locate the parameter servers, each of which only handles a portion of the model parameters, over all the nodes. For the *default* placement policy, which can be used with all three of our resource allocation policies, we place layers of the model in round-robin fashion over all the parameter servers as in TensorFlow [53]. For ED, however, another policy is possible, which we refer to as 'ED-local'. With 'ED-local', we place the layers of a partition on the parameter server running on the same node, incurring no actual network traffic across the nodes for parameter synchronization. This is possible as the same partition of the model can be assigned *locally* to the GPU on the same node for every virtual worker. For all results reported hereafter, the 'default' policy is used, except for 'ED-local'.

8.2 Performance of a single virtual worker

We first investigate the performance of the 7 different individual virtual workers that are possible according to the allocation schemes in Table 3. Figure 4 shows the throughput over various values of N_m, which is the number of minibatches executed concurrently, in the virtual worker normalized to that of when $N_m = 1$ and the maximum average GPU utilization among the four partitions for ResNet-152 and VGG-19. The numbers shown (in the box) along with the allocation policy are the absolute throughput (images/sec) when $N_m = 1$. Note that some results for larger N_m are not shown. This is because the GPU memory cannot accommodate such situations and hence, cannot be run.

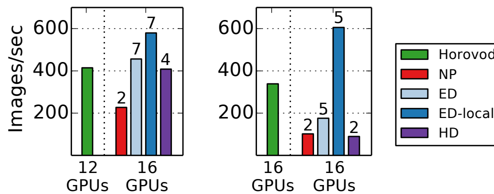


(a) ResNet-152



(b) VGG-19

Figure 4: Normalized throughput and the maximum average GPU utilization among partitions in a single virtual worker for various resource allocation policies as N_m is varied. The number in parenthesis is absolute throughput (images/sec) when $N_m = 1$ (which is equivalent to the naive MP) for each policy.



(a) ResNet-152 (b) VGG-19

Figure 5: Performance with the three allocation policies when D=0 (The number on bar represents N_m)

From the results, we can see that as N_m increases, normalized throughput of a virtual worker as well as the maximum GPU utilization generally increases. Note that, though not shown, the total GPU memory utilization tends to increase as N_m increases. However, depending on the resource allocation scheme (which results in different partitions of a model) as well as the DNN model, the effect of having larger N_m varies. When a virtual worker is configured with homogeneous GPUs, the average GPU utilization of each partition is similar to each other. However, when it is configured with heterogeneous GPUs, there is a tendency that the GPU utilization of the first or last partition is higher than those of the other partitions. For this configuration, different computation capabilities and memory capacity of the GPUs are considered when partitioning a model. As it is possible that only a small number of layers are assigned to some GPUs, the overall GPU utilization may turn out to be low.

8.3 Performance of multiple virtual workers

Figure 5 shows the throughput of training each model with the three resource allocation policies, where "Horovod" indicates the state-of-the-art DP via Horovod that uses AllReduce communication[4]. In these experiments, for each resource allocation policy, N_m is set such that performance is maximized while every virtual worker uses the same value of N_m as this is the assumption behind HetPipe. For ResNet-152, the whole model is too large to be loaded into a single GPU with G type, and thus, Horovod uses only 12 GPUs.

[4]We use the same minibatch size for all workers of Horovod as the minibatch size is one of the critical factors to the final performance of a trained DNN and adaptive batch sizing will affect convergence [5].

Table 4: Performance improvement of adding whimpy GPUs (The number in parenthesis presents the total number of concurrent minibatches in HetPipe)

Model	Single GPU [V]	Method	4 GPUs 4[V]	8 GPUs 4[VR]	12 GPUs 4[VRQ]	16 GPUs 4[VRQG]
VGG-19	159	Horovod	164	205	265	339
		HetPipe	300(5)	530(16)	572(20)	606(20)
ResNet-152	112	Horovod	233	353	415	X
		HetPipe	256(5)	516(20)	538(24)	580(28)

The results in Figure 5 show that the performance of DNN training is strongly affected by how heterogeneous GPUs are allocated to virtual workers. From the results, we can make the following observations: First, for VGG-19 whose parameter size is 548MB, the performance of Horovod, which reduces communication overhead for parameter synchronization, is better than those of NP, ED, and HD. However, for ResNet-152 whose parameter size is 230MB, ED and HD, which utilize virtual workers with similar performance, show a bit better or similar performance to Horovod (with 12 GPUs). Second, with NP, training performance of ResNet-152 and VGG-19 is low as N_m is bounded by the virtual worker with the smallest GPU memory. Third, with ED-local, intra-communication occurs between each GPU and the parameter server, significantly reducing communication overhead across the nodes, especially for VGG-19, the model with a large parameter set. For VGG-19, the amount of data transferred across the nodes per minibatch with ED-local (i.e., 103MB) is much smaller than that with Horovod (i.e., 515MB). Thus, the performance of ED-local (which also mitigates the straggler problem) is 1.8× higher than Horovod. For ResNet-152, the amount of data transferred with ED-local (i.e., 298MB) is larger than that with Horovod (i.e., 211MB) because the sizes of output activations to be sent between partitions are large, even though the parameter size is relatively small. However, the throughput of ED-local is still 40% higher than Horovod. This is because Hetpipe allows each virtual worker to process a large number of minibatches concurrently. Compared to NP and HD, ED-local (or ED) usually has larger N_m in each virtual worker, improving throughput.

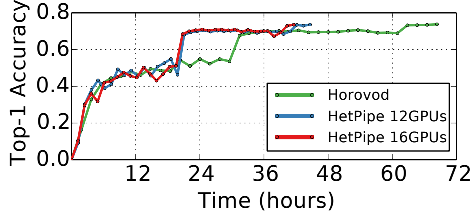


Figure 6: ResNet-152 top-1 accuracy

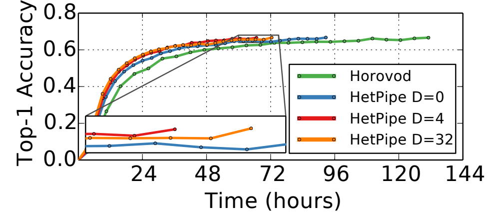


Figure 7: VGG-19 top-1 accuracy

Next, we investigate how the throughput is improved when whimpy GPUs are additionally used for training. Table 4 shows the throughput of VGG-19 and ResNet-152 when DP via Horovod and HetPipe with ED-local are used over different sets of heterogeneous GPUs, and also when a single V GPU is used. For these experiments, HetPipe is configured to use four virtual workers, except for '4 GPUs' where a single virtual worker is used. In the table, the number and type of GPUs used for each experiment are also given. From the results, we can see that the performance of both Horovod and HetPipe increases when additional whimpy GPUs are used for training. With additional GPUs, HetPipe can increase the total number of concurrent minibatches processed, having up to 2.3 times speedup. This scenario can be thought of as an answer to when new, higher end nodes are purchased, but one does not know what to do with existing nodes. The results show that making use of the whimpy systems allows for faster training of larger models.

8.4 Convergence

Our HetPipe based on the WSP model is guaranteed to converge as proven in Section 6. In this section, we analyze the convergence performance of HetPipe with ED-local using ResNet-152 and VGG-19. For our experiments, the desired target accuracy of ResNet-152 and VGG-19 is 74% and 67%, respectively.

Figure 6 shows the top-1 accuracy of ResNet-152 with Horovod (12 GPUs), HetPipe (12 GPUs), and HetPipe (16 GPUs), where D is set to 0 for HetPipe. For the experiments with 12 GPUs, the 4 G type GPUs are not used. When the same set of GPUs are used, convergence with HetPipe is 35% faster than that of Horovod by reducing the straggler problem in a heterogeneous environment and exploiting both PMP and DP. Furthermore, by adding four more whimpy G GPUs, HetPipe improves training performance even more, converging faster than Horovod by 39%.

Figure 7 shows the top-1 accuracy of VGG-19 with Horovod and HetPipe as we vary D to 0, 4, and 32. For the experiments, all 16 GPUs are used. The figure shows that convergence with the BSP-like configuration (i.e., $D = 0$) of HetPipe is roughly 29% faster than that with Horovod. As we increase D to 4, the straggler effect is mitigated and the communication overhead due to parameter synchronization is reduced. Thus, convergence is faster by 28% and 49% compared to $D = 0$ and Horovod, respectively. In this experiment with ED-local (where the training speed of each virtual worker is similar), when D becomes very large (i.e., 32), the throughput remains similar but the convergence performance degrades by 4.7%, compared to $D = 4$. This is because it is unlikely that the clock distance between the fastest and slowest virtual workers becomes as large as 32, but higher global staleness can degrade the convergence performance (similarly discussed in [18]). Note that though not shown, using larger D has a greater effect for HetPipe with NP, ED and HD resource allocation, and the different resource allocations only affect the set of heterogeneous GPUs used for each virtual worker and do not affect the convergence behavior.

We also analyze the synchronization overhead as D is varied. We find that as D increases, the waiting time of a virtual worker to receive the updated global weights decreases. In our experiments, the average waiting time with $D = 4$ is found to be 62% of that with $D = 0$. Furthermore, the actual idle time is only 18% of the waiting time as the virtual worker can continue to proceed in the pipeline while waiting.

9 Discussion

Comparison to PipeDream PipeDream [38], which is the closest related study, optimizes PMP of a single virtual worker, only employing DP for lagging layers within a virtual worker in homogeneous environments. To be adapted to heterogeneous environments, its partitioning algorithm must be extended to consider the different performance and memory sizes of heterogeneous GPUs, various orders of heterogeneous nodes used for a pipeline, and the memory requirement of the GPUs for partitions.

We run the training of ResNet-152 using PipeDream, which is implemented on PyTorch [37], in our heterogeneous GPU cluster described in Section 8.1. Since the partitioning algorithm does not consider heterogeneous GPUs, for each GPU type, we profile ResNet-152, then generate partitions of the model assuming that our cluster is configured with homogeneous GPUs with that type, and finally, measure the throughput of PipeDream with the partitions. All the computed configurations of the pipeline result in a large number of (i.e., 12 or 14) partitions. For example, with Q, the configuration is 4-2-1-1-1-1-1-1-1-1-1-1 indicating that the model is divided into 12 partitions where the first partition is executed by four GPUs with DP, the second one is executed by two GPUs with DP, and so on. For these configurations, we run experiments with various orders of the four different nodes

and test using several batch sizes. (Note that we could not run training for some configurations due to out of memory errors.) The best throughput measured using PipeDream is 158. Recall that the throughputs of Horovod (with 12 GPUs) and HetPipe are 415 and 580, respectively. In this case, the performance of PipeDream for ResNet-152 is found to be low as a large number of partitions cause high network overhead, in addition to the sub-optimal partitions. Therefore, with PMP alone (i.e., single virtual worker), the performance benefit may become limited when a model is divided into numerous partitions. Instead of increasing partitions, running DP with multiple virtual workers like HetPipe can improve the parallelism of training and further improve performance in such cases.

Effect of imbalanced partitions Our partitioning algorithm attempts to balance partitions while satisfying the memory requirements. However, depending on the DNN model, computed partitions may be imbalanced. For example, for a model composed of a small number of layers, if one layer takes much longer to execute compared to other layers, the partitions may end up having different execution times. In this case, the performance of the pipeline will be degraded as in any other pipeline-based systems. Note that running DP for the slow partition to have a similar processing rate across all the partitions like PipeDream [38] will be a possible extension of HetPipe.

10 Related Work

Pipelining has been leveraged to improve the performance of machine learning systems [6, 7, 19, 32, 38]. A pipelining scheme is employed to handle expensive backpropagation [7]. Pipe-SGD pipelines the processing of a minibatch to hide communication time in AllReduce based systems [32]. A weight prediction technique is proposed to address the staleness issue in pipelined model parallelism [6]. Detailed comparisons of HetPipe with PipeDream [38] and GPipe [19] are provided in Section 2.2. Note that the feature of overlapping computation and communication, presented in PipeDream [38], will also improve the performance of our system. PipeDream employs the one-forward-one-backward scheduling algorithm for pipeline execution. Sophisticated schedulers that consider various factors such as heterogeneous configurations, the number of partitions, and the number of concurrent minibatches within a virtual worker, can potentially improve the performance of HetPipe. Techniques to optimize learning rates have been studied [15], which can also be applied to HetPipe to help converge faster.

Decentralized training systems that consider heterogeneous environments have also been studied [33, 34]. However, these techniques do not consider integration of DP with PMP, which allows support for large models that do not fit into single GPU memory. In AD-PSGD, once a mini-batch is processed, a worker updates the parameters by averaging them with only one neighbor which is randomly selected [33]. This is done asynchronously, allowing faster workers to continue. In theory, the convergence rate of AD-PSGD is the same as SGD. In principle, the contribution of AD-PSGD is orthogonal with the contributions of HetPipe in that we can extend our HetPipe further by adapting the idea of asynchronous decentralized update in AD-PSGD when there is a bottleneck in the parameter server. When it comes to the experimental evaluations, the performance of AD-PSGD is evaluated for DNN models whose sizes are 1MB, 60MB, and 100MB, which are smaller than the models we consider in HetPipe. For a decentralized training system, Hop [34] considers the bounded staleness and backup workers, and uses CIFAR-10 for performance evaluation on a CNN model.

There have been earlier efforts to employ DP and/or MP for model training. Project Adam uses both DP and MP to train machine learning models on CPUs [8]. Pal et al. combine DP and MP in a similar way as our system, but do not consider pipelining nor heterogeneous GPUs [43]. STRADS leverages MP to address the issues of uneven convergence of parameters and parameter dependencies [27]. FlexFlow considers utilizing parallelism in various dimensions such as sample, operator, attribute and parameters to maximize parallelization performance [23]. Bounded staleness has been explored where Jiang et al. present heterogeneity-aware parameter synchronization algorithms based on the SSP model [24], while Cui et al. analyze the effects of bounded staleness [11].

Hierarchical AllReduce performs the AllReduce operation in two levels [22]. This technique does not solve the straggler problem in a heterogeneous GPU cluster, as master GPUs in the second level will have different GPU types. BlueConnect is an efficient AllReduce communication library considering heterogeneous networks [9]; unfortunately, it also cannot handle stragglers caused by heterogeneous GPUs.

11 Conclusion

In this paper, we presented a DNN training system, HetPipe, that integrates pipelined model parallelism with data parallelism. Leveraging multiple virtual workers, each of which consists of multiple, possibly whimpy, heterogeneous GPUs, HetPipe makes it possible to efficiently train large DNN models. We proved that HetPipe converges and presented results showing the fast convergence of DNN models with HetPipe.

Acknowledgments

We would like to thank our shepherd Saurabh Bagchi and the anonymous reviewers for their invaluable comments. This work was partly supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-14 and Institute for Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-00118 and No.2019-0-00075). Young-ri Choi is the corresponding author.

References

[1] 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. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Amazon. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/.

[3] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 2003.

[4] Léon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of COMPSTAT*, 2010.

[5] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 2018.

[6] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform. *arXiv preprint arXiv:1809.02839*, 2018.

[7] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. Pipelined Back-Propagation for Context-Dependent Deep Neural Networks. In *Proceedings of the Annual Conference of the International Speech Communication Association*, 2012.

[8] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System . In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[9] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[10] Ronan Collobert and Jason Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2008.

[11] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[14] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. CatBoost: gradient boosting with categorical features support. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[17] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing Magazine*, 2012.

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

[19] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2019.

[20] IBM. CPLEX-Optimizer. https://www.ibm.com/analytics/cplex-optimizer/.

[21] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[22] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shi Shaohuai, and Chu Xiaowen. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[24] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware Distributed Parameter Servers. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017.

[25] Wenbin Jiang, Geyan Ye, Laurence T Yang, Jian Zhu, Yang Ma, Xia Xie, and Hai Jin. A Novel Stochastic Gradient Descent Algorithm Based on Grouping over Heterogeneous Cluster Systems for Distributed Deep Learning. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2019.

[26] Tian Jin and Seokin Hong. Split-CNN: Splitting Window-based Operations in Convolutional Neural Networks for Memory System Optimization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[27] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

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

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[30] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On Model Parallelization and Scheduling Strategies for Distributed Machine Learning. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2014.

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

[32] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[33] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[34] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. Hop: Heterogeneity-aware Decentralized Training. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[35] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zahria, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 2016.

[36] Microsoft. Microsoft Azure Pricing. https://azure.microsoft.com/en-us/pricing/.

[37] msr-fiddle. PipeDream: Generalized Pipeline Parallelism for DNN Training. https://github.com/msr-fiddle/pipedream/.

[38] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[39] NVIDIA. GeForce RTX 2060. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2060/.

[40] NVIDIA. Quadro P4000. https://www.nvidia.com/en-us/design-visualization/quadro-desktop-gpus/.

[41] NVIDIA. TITAN RTX. https://www.nvidia.com/en-us/titan/titan-rtx/.

[42] NVIDIA. TITAN V. https://www.nvidia.com/en-us/titan/titan-v/.

[43] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training. *IEEE Micro*, 2019.

[44] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. *arXiv preprint arXiv:2005.14038*, 2020.

[45] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 2009.

[46] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the Conference on International Conference on Learning Representations (ICLR)*, 2017.

[47] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2019.

[48] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2011.

[49] Muhammad Saqib, Sultan Daud Khan, Nabin Sharma, and Michael Blumenstein. A Study on Detecting Drones Using Deep Convolutional Neural Networks. In *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2017.

[50] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

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

[52] TensorFlow. TensorFlow benchmarks. https://github.com/tensorflow/benchmarks/.

[53] TensorFlow. tf.train.replica_device_setter. https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/replica_device_setter/.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017.

[55] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[56] Tian Wang, Yang Chen, Mengyi Zhang, Jie Chen, and Hichem Snoussi. Internal Transfer Learning for Improving Performance in Human Action Recognition for Small Datasets. *IEEE Access*, 2017.

[57] FengLi Yu, Jing Sun, Annan Li, Jun Cheng, Cheng Wan, and Jiang Liu. Image Quality Classification for DR Screening Using Deep Learning. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2017.

[58] Shen-Yi Zhao and Wu-Jun Li. Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-Free Approach with Convergence Guarantee. In *Proceedings of the Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2016.

[59] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and Analyzing Deep Neural Network Training. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[60] Martin Zinkevich, John Langford, and Alex J Smola. Slow Learners are Fast. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2009.