Exploring Flexible Communications for Streamlining DNN Ensemble Training Pipelines

Randall Pittman*, Hui Guan*, Xipeng Shen*, Seung-Hwan Lim† and Robert M. Patton†
*North Carolina State University, Raleigh, NC
†Oak Ridge National Laboratory, Oak Ridge, TN
Emails: {rbpittma, hguan2, xshen5}@ncsu.edu, {lims1, pattonrm}@ornl.gov

***Abstract*—Parallel training of a Deep Neural Network (DNN) ensemble on a cluster of nodes is a common practice to train multiple models in order to construct a model with a higher prediction accuracy, or to quickly tune the parameters of a training model. Existing ensemble training pipelines perform a great deal of redundant operations, resulting in unnecessary CPU usage, or even poor pipeline performance. In order to remove these redundancies, we need pipelines with more communication flexibility than existing DNN frameworks can provide. This project investigates a series of designs to improve pipeline flexibility and adaptivity, while also increasing performance. We implement our designs using Tensorflow with Horovod, and test it using several large DNNs in a large scale GPU cluster, the Titan supercomputer at Oak Ridge National Lab. Our results show that with the new flexible communication schemes, the CPU time spent during training is reduced by 2-11X. Furthermore, our implementation can achieve up to 10X speedups when CPU core limits are imposed. Our best pipeline also reduces the average power draw of the ensemble training process by 5-16% when compared to the baseline.**

I. Introduction

Recent years have witnessed rapid progress in the development of Deep Neural Networks (DNNs). Ensemble training of DNNs refers to the use of many computing nodes to concurrently train a number of DNNs on a dataset. It has two purposes. The first is to search for the best configurations of DNN hyperparameters, such as the number of layers, the number of filters at each layer, the appropriate learning rates, and so on. This is called *hyperparameter tuning*, which is essential for finding the DNN that works the best for a particular task [1], [2]. The second is to produce a collection of DNNs that work together (e.g., through majority voting) to make more accurate predictions than each individual DNN can provide [3], [4]. For either purpose, such ensemble training multiplies the I/O and CPU demand on an already burdened system, since it duplicates the model training pipeline across different nodes.

The DNN training pipeline is an iterative process that consists of reading, preprocessing, and computing/training stages. Data is first read from a storage system, then preprocessed for training using the CPU, and lastly is sent to the GPU for DNN training. A common implementation of an ensemble training pipeline is constructed by duplicating this pipeline onto multiple machines to train models in parallel. Such a simple parallelization scheme inherently creates redundancies, particularly for the preprocessing stage.

Preprocessing is CPU-intensive and can form the bottleneck of the pipeline. In ensemble training, each DNN model is often trained over the same data,[1] and hence much redundancy exist in the preprocessing operations (e.g., resizing and cropping) being performed for each DNN. The result is that while preprocessing takes unnecessarily high CPU usage and power consumptions on every compute node in the ensemble training, the rate of preprocessing may remain behind the demand from the GPU for training a model over prepared data.

The difficulty in resolving these redundant operations is the lack of flexibility in model training pipelines. Since most present frameworks (e.g. TensorFlow, Caffe, and Torch) focus on training a single model, they do not provide sufficient flexibility to allow pipelines to fit the demands of *parallel* ensemble training in distributed environments. The Horovod library [5], an addon to Tensorflow, provides better distributed training capabilities, but is still too rigid in design to support ensemble training.

The overarching goal of the research direction presented here is to add flexibility into existing DNN frameworks to enable customizable communications in parallel ensemble training, and further to identify the communication schemes that best suite DNN ensemble training in both training time and power consumption.

In this study, we analyze a series of queues used to buffer data between each stage in the machine learning pipeline, allowing us to isolate potential bottlenecks. We discover a bottleneck in the preprocessing stage that can hinder DNN training speed. To add flexibility to present frameworks, we modify the Horovod [5] library to support arbitrary MPI group allocation. Using this addition with Tensorflow, we examine three pipeline designs that we refer to as All-Shared, Single-Broadcast, and Multi-Broadcast. These pipelines are constructed from existing MPI collective operations, such as all-gather and broadcast.

The All-Shared pipeline shares the preprocessing step across all members in the ensemble, whereas Single-Broadcast and Multi-Broadcast share within a subset of the ensemble. Single-Broadcast elects a leader to broadcast the preprocessed data to other nodes, while Multi-Broadcast performs asynchronous broadcasts from multiple nodes. We examine these three cases

[1]Some ensemble trainings use different segments of a large dataset to train different models. That scenario is beyond the focus of this work.

as an initial study of different primitive pipeline communication schemes.

Among these pipelines, the All-Shared scheme provides significantly more efficient parallel ensemble training. Using the Titan supercomputer at Oak Ridge National Laboratory, we show that the preprocessing stage can indeed form a bottleneck for some DNN; for Alexnet, it takes 96% CPU usage while still limiting the GPU training stage at only 66% of its peak throughput. Our best optimized pipeline can meet and exceed Alexnet's preprocessing demand by up to 2X, while reducing CPU usage by 2-11X. When the available CPU cores are limited, the less CPU demands of the improved pipeline yields up to 10X faster training rates than the default pipeline does. Lastly, we provide experimental results on Titan showing that during training, our best pipeline uses 5-16% less energy than the default does.

In summary, we present the following key contributions:

1) To our best knowledge, this is the first work that systematically characterizes performance issues present in parallel DNN ensemble training in large distributed environments. (Section III)
2) It adds into existing DNN ensemble training pipelines with flexible communication controls. (Section IV)
3) It provides the first known exploration of distributed communication schemes for streamlining parallel ensemble training pipelines in large distributed environments. (Section IV)
4) It offers a thorough performance analysis of the capabilities of these pipelines on the Titan supercomputer. (Section VI)

We will first introduce DNN pipelines, showing how they can be extended for ensemble training. After seeing the shortcomings of more simplistic designs, we will present our alternative pipelines. Lastly, we will provide detailed experiments to show the benefits our optimizations yield.

II. Backgrounds

A. Deep Neural Network Training Pipeline

A typical deep neural network training pipeline contains three stages: reading the data from storage systems, preprocessing the data, and training the model (see Figure 1). Data is first read into a queue, and is then run through various transformations known as *preprocessing*. Afterwards, the data is queued again and arranged into *batches*. The *batch size* is the number of data the network trains simultaneously per step. When training DNNs, it is important not to overfit to a particular dataset. Preprocessing typically helps with this goal by modifying input data to be more generic.

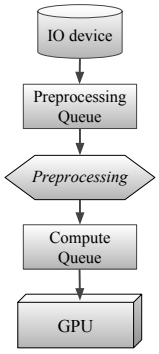


Fig. 1: A typical pipeline for DNN training.

The first stage in preprocessing raw data is decoding, perhaps from a compressed state. For example, *jpg* files are heavily compressed and need to be decoded before use. The second stage, known as *data augmentation*, transforms the data with a series of random operations to be more generic.

Many expensive data preprocessing operations are performed on a point by point basis. For example, image preprocessing allows us to flip, rotate, blur, and resize images to allow for more general cases than what is being provided by the dataset. While this increases the computational complexity of the input pipeline, it also increases the generality of the final DNN. Many other preprocessing techniques exist for other data types, not exclusively images. Both audio [6] and sensor [7] data have a wide range of preprocessing techniques that can be applied.

Preprocessing techniques can further be divided into online and offline preprocessing. In the offline case, preprocessed data is saved to storage, then loaded directly into the pipeline when training begins. On the other hand, online preprocessing techniques are used every time the dataset is loaded. Online is particularly useful when randomized preprocessing techniques are used. Images may be flipped, rotated, or cropped in random ways, allowing a single image to provide a vast array of possible inputs to a DNN. Online preprocessing is the method commonly used in DNN training as its dynamic nature makes it more effective in preventing overfitting a dataset, allowing much more general applications for the network. In modern implementations of DNN training, online preprocessing typically serves as one stage in the training pipeline; the pipeline structure helps hide its runtime overhead.

B. Heterogeneous GPU-CPU cluster for DNN training pipeline

The modern high performance computing cluster has evolved into a hybrid architecture that houses CPUs and GPUs on each node in order to handle other computationally heavy workloads with high energy efficiency [8]–[10]. One of the most prominent large-scale examples of such an architecture is the Titan supercomputer located at Oak Ridge National Laboratory. Each of Titan's 18,688 nodes features both a 16-Core AMD CPU and a K20X Nvidia GPU [11]. The next supercomputer that will soon be replacing Titan is called Summit, which is anticipated to be ready for researchers in 2018. Summit will contain 2 IBM Power9 CPUs and 6 Nvidia Volta GPUs [12]. With this level of computing power, researchers can use each node to either train larger networks, or train smaller networks faster using techniques such as batch parallelism.

Heterogeneous GPU-CPU clusters are particularly well suited towards DNN training, since the CPU and GPU can

work together to accelerate the training pipeline. In heterogeneous GPU-CPU clusters, the GPU is generally given the training task and the CPU is in charge of reading and preprocessing data into batches that the GPU can quickly use, ideally with as little idle time as possible. Such a division of pipeline steps is largely to achieve maximal training throughput on the GPU, since GPUs are generally able to process machine learning kernels to train DNN models with a higher throughput than multi-core CPUs [13]. To achieve maximal training throughput, it is generally best to preserve cache and memory states on the GPU. If the GPU were to attempt preprocessing as well as training, the CPU would need to perform additional data copies (TF records) to GPU memory; kernel switches will add extra overhead, depending on how well the fusion of the preprocessing and training stages is performed. Furthermore, extra memory would need to be allocated on the GPU for the preprocessing stage, which constricts the maximum batch size that can be used on a large network. Thus, in general, performing preprocessing on GPU does not give performance benefits. The general goal is to make the GPU's training stage as efficient as possible, while the preprocessing on the CPU side attempts to saturate GPU resources.

III. Ensemble Performance

In this section we discuss the scheme of the typical DNN ensemble training pipelines used in existing work. We refer to such pipelines as the *duplicated pipelines* scheme, and provide a Tensorflow implementation that is used to test and analyze its performance. We later use this implementation as a baseline against which other schemes may be compared.

A. Duplicated Pipelines and the Implementation

DNN ensemble training consists of the training of a number of DNN variants. These variants are independent from one another. The scheme commonly used in existing work, *duplicated pipeline* scheme, launches N duplicated pipelines with each running on one (or more) nodes training one DNN variant in the ensemble.

We implement the scheme based on Tensorflow.[2] We use the Slim module [14] as a starting point, since it includes the implementations of several popular networks, such as Inception, Alexnet, and VGG. Furthermore, Slim provides a robust set of preprocessing operations by default for the Imagenet dataset, which proved quite useful for our tests. In our experiments, each DNN runs on one Titan node.

B. Settings for Testing

We describe the settings used in our performance testing of various ensemble training schemes as follows. Some of these choices are designed to draw out problems of interest that may arise from an ensemble of DNNs.

[2]All Tensorflow code is version 1.3.0.

1) Workloads: In general, parallel model training can be used as a fast method for hyper-parameter tuning [15], or it can be used to create multiple learners for increased classification accuracy, or to learn an ensemble model [3], [4]. An ensemble model is most effective when each DNN serves a useful and probably unique testing purpose, and has been modified appropriately to suit that purpose. As discussed earlier, the final result is intended to be more diverse than any single classifier could be. Towards this goal, our study investigates the system efficiency of parallel ensemble training.

The more complicated case arises when the differences between each DNN are substantial enough to cause *significant* changes in performance. For example, each model may contain varying numbers of hidden layers or different numbers of nodes within each layer. Since the number of layers in a network is a primary factor influencing training time [4], such changes could cause significant differences in training times between the members of the ensemble. While this area may be an interesting point for a future optimization study, managing the burst computational requirements from many concurrent model training pipelines poses the most urgent problem. In light of this, our experiments focus on the DNN variants in an ensemble that are of the same structure but differ in their learning rates, initial filter values, or other non-structural parameters.

2) Datasets: When considering the effects of preprocessing, computation, IO usage, network traffic, etc., it is reasonable to require that the input dataset dimension and the number of elements be large. Smaller datasets such as MNIST or Cifar-10[3] will likely require very little resources and will train quickly. The primary dataset used for this research is a subset[4] of *ImageNet* [16], where the entire dataset contains over 14 million images of size 224×224. With this dataset it is much easier to investigate the performance effects of DNN ensembles. For a given dataset, we also expect that every image will need to be processed by every DNN in an ensemble.

C. Baseline

In this section we provide data that characterizes the performance of individual DNNs, as well as DNN ensembles.

1) Single Node: We begin with a performance evaluation of the default pipeline shown in Figure 1 on a single node. Since the primary goal of the pipeline is to saturate the GPU with prepared data, we present a scenario in which the GPU can process data quickly. We use Alexnet for this purpose since it is a smaller network that uses a larger batch size of 128 [17].

Since preprocessing occurs on the CPU, it is important to allow parallelism over all CPU cores. Multi-core execution can drastically speed up preprocessing, and can sometimes utilize all CPU resources for the task. The DNN computation is affected little by the high CPU usage since it executes on the GPU. Tensorflow allows such CPU parallelism by default, but in our case we needed to manually change the number of parallelism threads. We set *inter_op_parallelism_threads* and

[3]Both datasets contain 60000 images. MNIST images have size 28×28, and Cifar-10 images have size 32×32

[4]Our subset contains approximately 1.3 million images.

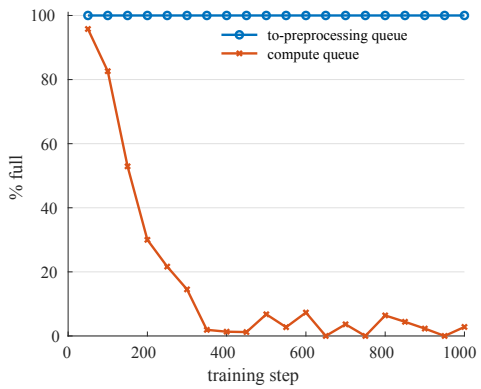


Fig. 2: For the default single pipeline, the preprocessing queue is always full, while the compute queue empties quickly. Thus the preprocessing task is the bottleneck.

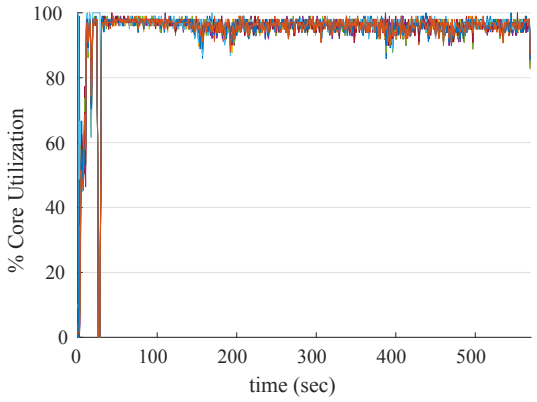


Fig. 3: Default single pipeline core utilization for each of the 16 cores on a single Titan node when training Alexnet. The average core utilization over the entire graph is 94.3%. When the startup phase is excluded, the average is 96.0%.

intra_op_parallelism_threads to 16 in order to maximize the usability of the 16-core CPUs available on each Titan node. The former enables parallelism between multiple operations, while the latter parallelizes individual operations if supported. We also needed to set a flag when launching the Titan job that enabled multi-core usage for each node[5].

Since Tensorflow training requires that the graph be constructed symbolically, and is only executed within an API session call, it is difficult to obtain direct performance diagnostics at runtime. Thus we use Tensorboard[6] summaries on the various queue sizes in the pipeline to determine where bottlenecks might be occurring. Since the operation that saves summaries in Tensorflow can affect training performance, we save summaries every 20 steps and disable certain costly summary operations, such as preprocessed image viewing. We run Alexnet for 1000 steps on the ImageNet dataset, then analyze the relevant queues in Tensorboard.

Figure 2 shows the measured size of the preprocessing and compute queues during the training process. As shown before in Figure 1, the preprocessing queue is the data that is about to be preprocessed, and the compute queue is the preprocessed data being fed to the DNN. In this case, the preprocessing queue fills up quickly enough that the summary data for this queue reports that it is always full. On the other hand, the compute queue fills up during the startup phase, then empties out in the first few hundred steps. In Tensorflow, the first step of the training process is typically many times slower than the rest. This is primarily due to various initialization and optimization routines that are being executed at runtime. The result is that the batch queue has time to fill while the first step is executing, but cannot keep up after the first step. The bottleneck in this case is therefore the preprocessing stage.

[5]Titan jobs are executed using the *aprun* command. Passing the number of allowed threads using the option *-d* allows multiple cores to be used by a single task. We used *-d16* to enable all cores to be used for each Tensorflow session.

[6]Tensorboard is a diagnostic tool designed to parse and display summary data produced during a Tensorflow training session.

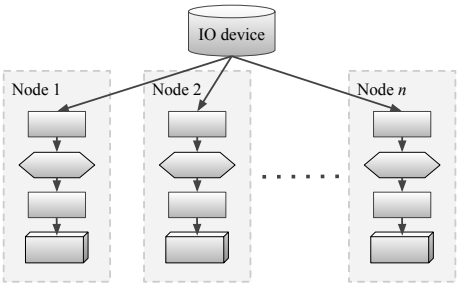


Fig. 4: Duplicated pipelines that can be used to concurrently train DNNs.

It is important to show that the preprocessing uses the entire CPU. Figure 3 shows the utilization level for each of the 16 cores in our default single pipeline test. Once Tensorflow has finished initializing, we see the utilization reach peak levels and remain there. The average measured utilization for this test was 96.0% after startup. From these series of tests, we conclude that a heavy preprocessing load with a smaller DNN is capable of shifting the bottleneck from the model training to the preprocessing. More computationally intense models (e.g., GoogleNet with Inception modules) can also create similar issues on newer hardwares like NVIDIA V100 with TensorCore technology, where processing rate for deep learning workloads is 90 times improved [18].

2) Multiple Nodes: The natural extension to the single pipeline in Figure 1 is to duplicate each pipeline for each DNN in an ensemble. This duplicated pipeline is shown in Figure 4. In theory, each DNN could be an arbitrary network, but our present tests use the same network for the sake of analyzing optimization potential.

We first note two main concerns arising from the duplicated

pipeline. First, each node reads its own copy of the dataset, which is highly redundant and places unnecessary strain on the storage systems[7]. High IO usage could in theory lead to scalability problems. Second, the preprocessing operations are redundant since the same data is being modified. [8] While this does not present scalability problems, it does result in unnecessary CPU usage. As shown earlier in Figure 3, the CPU usage could actually be quite high. This presents some opportunities for pipeline optimization.

In order to test potential scalability issues, we perform a test of the duplication pipeline on 1000 nodes of Titan and compare overall training time to that of nodes run individually. Table I shows the results of executing 2000 steps of Alexnet on 1000 nodes of Titan in parallel, as well as the results for an isolated node running by itself. We execute the single-node test 50 times for a more precise result. While the 1000 nodes exhibited slightly higher variance in its runtimes, the overall runtime was not affected. This demonstrates that the storage systems in Titan did not suffer performance issues caused by the high number of data requests.

IV. Optimized Pipelines

Keeping in mind the issues with the duplicated pipeline discussed in the previous section, we establish three objectives for designing pipelines to increase system efficiency:

1) Eliminate pipeline redundancies through data sharing.
2) Enable sharing by increasing pipeline flexibility.
3) Use increased flexibility to accelerate the pipeline.

Towards these goals, we focus on balancing the computational demand for preprocessing and model training. The key is in making the pipeline more intelligently take advantage of the computing resource in a cluster of nodes to both minimize redundant preprocessing and speeding it up.

A. Problem statement

Let n be the total number of DNNs being trained. Since each DNN uses a single compute node, n is also the number of nodes being used for the ensemble training. Let p be the number of nodes performing preprocessing operations, where $p \leq n$. Suppose the i'th preprocessor produces a data-block D_i. When designing a new pipeline, the goal is to have every node contain $D = [D_1, D_2, ..., D_p]$ after the communication stage. Note that for simplicity of notation, D refers to the dataset at any stage of the pipeline, either before or after being preprocessed.

Given a particular DNN and hardware system, let r_c be the GPU's compute throughput, and let r_p be the CPU's preprocessing throughput. Both can be measured in units of *images/second*. In order to achieve maximum training speed, we need $r_p \geq r_c$. However, this may not be the case, as we have already shown with Alexnet on Titan. A solution to this challenge is to share preprocessing steps across n machines for each data partition, which can raise the throughput of preprocessing up to $nr_p \geq r_c$.

Taking this approach, the number of machines, n needed to satisfy $nr_p \geq r_c$ was relatively small for our test cases. For example, our tests revealed that $n = 2$ is theoretically sufficient to saturate Alexnet's compute rate. If more advanced preprocessing techniques are used to enhance model training, the computational requirements on the CPU will increase and may require larger n to satisfy the condition.

In practice, nr_p is only an upper bound on the possible preprocessing rate. After the data has been prepared, it must be shared over the cluster's network to each training node. We define peak(n) as the peak rate at which images can be received by each of the n nodes in such a cluster. Due to the communication overhead, peak$(n) \leq nr_p$. The gap between peak(n) and nr_p tends to increase as n grows. It is hence important to consider flexible pipeline designs to better utilize the capabilities of the cluster without adding too much overhead. Later on, we will show detailed empirical results in this regard.

In the remainder of this section, we introduce our method for improving pipeline flexibility, and further explore different communication patterns as alternatives to the baseline of the duplicated pipelines.

B. Horovod groups

Horovod [5] is a distributed deep-learning library for Tensorflow. Although distributed Tensorflow [19] provides implicit tensor sends and receives, it does not provide efficient collective operations (e.g. all-gather). Horovod fills the gap by supporting collective operations, including all-gather, broadcast, and all-reduce. Thus, it allows *tensor* objects to be sent through MPI collectives.

However, one limitation in Horovod is its master-worker communication structure. It is designed to train individual large DNNs in a batch-parallel manner. This task needs only one global communicator. More specifically, it is designed to operate in "ticks", each consisting in a series of operation requests to the master, followed by a *done* message. Such a structure forces all communication to occur on a global scale, specifically, using MPI_COMM_WORLD as the communicator for MPI messages. When designing custom pipelines, we need the ability to use MPI collectives within a subset of ranks.

To solve this issue, we added functionality to this library to create *Horovod Groups*. This addition allows the user to provide a list of groups that should be created upon initialization of the library. Whenever a collective operation is created, a group index must be provided indicating which MPI

[7]Unless the file-system uses caches and each node is reading data from the same files in such a way that the cache scores successive hits.

[8]The data decoding portion of preprocessing is always redundant. On the other hand, data augmentation operations could be considered as different in order to feed each DNN with unique final data. We do not consider this special case, but instead treat these operations as redundant.

TABLE I: Statistics comparing the total run time in seconds for 50 solo runs and a parallel run of 1000 nodes for 2000 Alexnet steps.

	Avg (sec)	Std Dev (sec)	Min (sec)	Max (sec)
Solo	1132.3	1.429	1129.7	1134.5
Parallel	1132.2	1.962	1125.0	1139.0

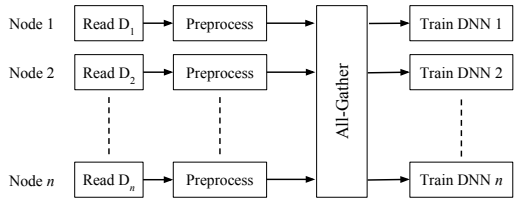


Fig. 5: Illustration of the All-Shared (AS) pipeline. The dataset D is divided into n partitions for each reader.

communicator to use for the operation. [9] This allows data to be broadcasted or all-gathered between subsets of MPI ranks.

In addition to normal Horovod requirements, the user must provide a list of groups to create within MPI. For example, providing the list $[[0, 1, 2], [2, 3, 4]]$ creates two groups: the first contains ranks $0, 1, 2$, and the second contains $2, 3, 4$. Any call to a library function, such as *all-gather*, will then require a group index, 0 or 1, specifying which group the operation belongs to.

C. All-Shared

To share preprocessed data with all nodes, one possible approach is to make every node a preprocessor ($n = p$) with each using an exclusive portion of the dataset, and then share each node's data with all other nodes. The MPI all-gather operation is well suited to this purpose. We refer to this as the *All-Shared* (AS) pipeline, as depicted in Figure 5.

The primary benefit of this design is to maximally share all the preprocessing across all the compute nodes. The limitation, however, is the lack of flexibility. For training more computationally heavy neural network models, it seems unnecessary to require that every node instantiate a data reader and preprocessing stage, when a small number of nodes could provide enough preprocessed data to training models. By using less nodes for preprocessing, perhaps we can use less CPU. Our next two designs attempt to take advantage of this fact, thereby increasing their *flexibility*.

D. Single-Broadcast

We now wish to allow the number of preprocessors p to be adjustable. Suppose the GPUs of n nodes are used for the GPU compute stage of the pipeline, while preprocessing happens on the CPUs of the first p $(1, \ldots, p)$ of those n nodes.

As a first step, we can perform an all-gather among the preprocessor nodes $1, \ldots, p$. Now each node has access to all the data, but the remaining $n-p$ nodes have none. One method to resolve this is to elect node p to broadcast its data out to nodes $p+1, \ldots, n$. This process is shown in Figure 6, and we refer to this pipeline as *Single-Broadcast* (SB).

E. Multi-Broadcast

Each node i in $1, \ldots, p$ has its own data item D_i. Instead of running an all-gather between preprocessors, each i could broadcast its D_i to all other nodes. In Multi-Broadcast, we avoid the initial all-gather by performing asynchronous broadcasts from each preprocessor, as shown in Figure 7.

[9] The groups created need not be mutually exclusive.

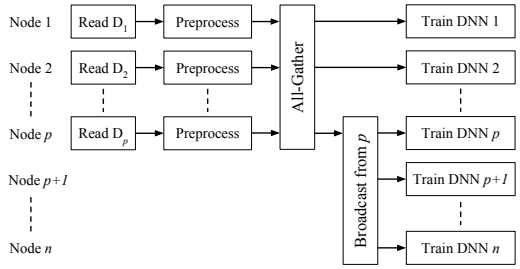


Fig. 6: Illustration of the Single-Broadcast (SB) pipeline. The dataset D is divided into p partitions for each reader instead of n, since there are now p readers feeding their own preprocessor.

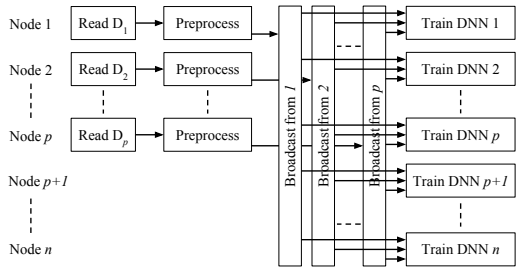


Fig. 7: Illustration of the Multi-Broadcast (MB) pipeline. Similar to the Single-Broadcast design, the dataset is divided into p partitions. Each D_i for $1 \leq i \leq p$ is broadcast to all nodes with the i'th preprocessor node as the root.

From a high level perspective, we can observe that AS (all-to-all) is a specific instance of MB (some-to-all) where $p = n$. However the implementation of MB uses a series of asynchronous parallel broadcasts to implement the some-to-all operation. This differs from the built-in MPI all-gather collective, and may use much more CPU. Thus we do not treat AS as a special case of MB, though it may be viewed as an MPI-optimized routine for the case $p = n$.

V. Metrics

In this section we introduce the DNNs and metrics used to compare the baseline and our alternate pipeline designs.

A. Peak Preprocessor Throughput

Previously we defined peak(n) as the function representing the maximum image preprocessing throughput in a pipeline for a given number of nodes n used for the pipeline. The peak function is a good method to measure the scalability of a pipeline, and provides the best mechanism for speed comparison to other pipelines.

Note that peak(n) is only a measure of *preprocessing* image throughput, and does not involve any DNN training. In order to test the value of this function for a specific pipeline, we construct the compute queue that stores batches ready to be trained, then we dequeue a batch. Repeating this operation

quickly enough causes the pipeline to reach peak image throughput.

The importance of measuring peak(n) is apparent when the compute throughput r_c is also considered. As mentioned before, we must have peak(n) $\geq r_c$ in order to saturate GPU resources. Towards this end, we additionally gather the value of r_c for each DNN in our tests. To obtain r_c, we calculate the average step duration for a specific DNN, while also pausing between steps to allow all queuing systems to catch up. This ensures that the GPU will have data ready to be dequeued when the next step is timed. By averaging the seconds per step, we then invert and multiply by the batch size to obtain images per second, or r_c.

B. CPU Usage

As a standard, it is important for the optimized version to run with at least the same training rate as the baseline. However, it is not expected for the optimized pipeline to train DNNs faster than the baseline under normal conditions. We previously established that it is possible for preprocessing to form a bottleneck, but this is a more unusual case. If preprocessing is not a problem, our optimized pipeline should not increase the training rate. In most of our tests, the GPU performance was the limiting factor. Recall that this may change when Summit becomes available, since there are many more GPUs on the new node architecture.

To measure overall CPU load, we use the *mpstat* command to obtain CPU utilization statistics on each compute node in 4 second intervals. After training is complete, we integrate CPU utilization statistics over time to obtain CPU usage for the job.

C. Core Usage Limits

Another useful metric is the runtime of the training process when a CPU core limit is imposed. Some cluster systems allow nodes to be shared by users who have requested few CPU cores for their job. For such systems, the charge allocated to the user's account for such a job is typically only charged for the number of cores allocated.

Unfortunately, node sharing and custom CPU core allocation is not allowed on Titan. In practice, however, a cluster's allocation scheme may be more flexible (e.g., Amazon AWS). By limiting the cores used by ensemble training, these cores can be left in an idle state to save power, or they can be used for other jobs. Overall, there may be good reason to impose core limits on ensemble training. We therefore test our pipeline under limited core conditions, while comparing the overall runtime to the baseline under the same limits.

Since Titan does not support node sharing nor partial core allocation, we simulate a limited CPU environment by controlling the number of threads allocated to each MPI rank[10]. Since each rank is allowed to use an entire node, the number of threads corresponds to the number of CPU cores allowed. As an example, when simulating a 3 core limitation training Alexnet on a basic pipeline, cores 0-2 average 95% utilization while cores 3-15 use at most an average of 0.6%.

[10]The number of MPI threads per rank is controlled by the *-d* option passed to the *aprun* command.

D. Energy Usage

A secondary benefit from decreased CPU usage is power savings. On Titan, we collect energy consumption data through 2 metered cabinets. Each of these cabinets includes 96 nodes, 8 of which are service nodes, leaving a total of 88 nodes for user jobs. One limitation of these cabinets is that they only record the consumption of the *entire* cabinet, so distinguishing between the power usage of different devices within the cabinet is impossible. Thus the results we report are the power consumption of all devices in the cabinet, not just the CPU. In order to eliminate possible power variances due to jobs executing on different systems, we reserved only one cabinet for all jobs. We submit each ensemble training job sequentially, with approximately 2 minute breaks between the job's end and the next launch. We record measurements from two runs for each type of job.

We observed that intensive tasks can cause the cabinet to use a flat peak power of 32.767KW. Since we had 88 nodes available, we decided to use only 80 of them for ensemble training tests in order to lessen the impact of this maximum output on the quality of the results.

VI. Results

A. Peak Preprocessing Throughput

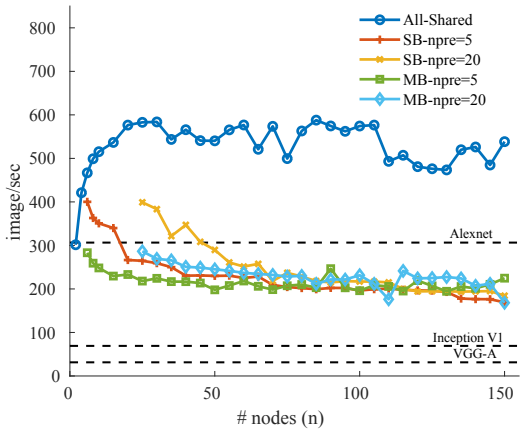
In order to effectively compare each pipeline to find the best, we first observe differences in peak throughput of the preprocessing stage, or peak(n). Recall that this function is a measure of the steady state image throughput for the preprocessing stage; the compute stage is stripped such that it only receives the preprocessed images but and does not do any DNN training.

Figure 8a shows the value of peak(n) for $n \leq 150$ for increments of 5 nodes. *All-Shared* pipeline uses all nodes for both preprocessing and the stripped compute stage; *SB-npre=5* and *SB-npre=20* are Single-Broadcast pipelines with 5 and 20 of the nodes for preprocessing respectively; *MB-npre=5* and *MB-npre=20* are the corresponding Multi-Broadcast pipelines. We observe that changing the number of preprocessors between 5 and 20 does little to affect the throughput as n increases. Furthermore, the SB and MB pipelines are incapable of saturating the compute queue for Alexnet, since they drop below its the throughput of its compute stage. They however can still meet the demands of the compute stages in larger networks (e.g., Inception and VGG).

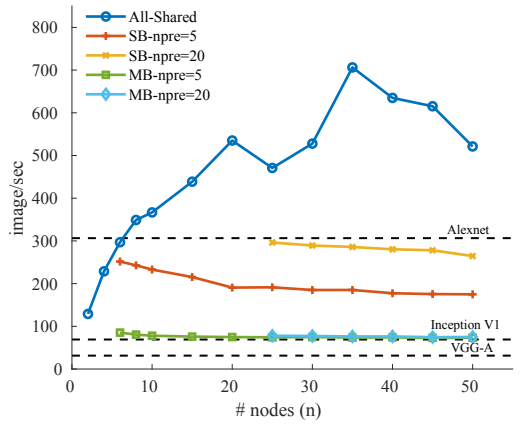
In order to clarify this data when core usage is restricted, Figure 8b shows peak(n) for up to 50 nodes when only 4 cores are used in each node. The performance for SB and MB is markedly decreased, while AS remains unchanged. This confirms that AS is better in terms of peak throughput for both full-core and partial core training.

For SB and MB, these results point towards the broadcast operation as a performance problem. As n increases while p remains constant, the broadcast size also increases. This correlates to the slow decrease in throughput seen in Figure 8.

As a final test for the broadcasting pipelines, we compare the CPU usage for AS, SB, and MB in Figure 9. We vary



(a) There are 16 CPU cores for use in each node.



(b) There are 4 CPU cores for use in each node.

Fig. 8: The curves show the peak preprocessing throughputs (peak(n)) when n, the number of nodes for the stripped DNN pipeline (i.e., the compute stage receives but does not actually use the preprocessed images for DNN training), increases. *All-Shared* pipeline uses all nodes for both preprocessing and the stripped compute stage; *SB-npre=5* and *SB-npre=20* are Single-Broadcast pipelines with 5 and 20 of the nodes for preprocessing respectively; *MB-npre=5* and *MB-npre=20* are the corresponding Multi-Broadcast pipelines. The horizontal dashed lines indicate the measured demand in images/sec of the actual (unstripped) compute stages of three DNN models on a Titan GPU.

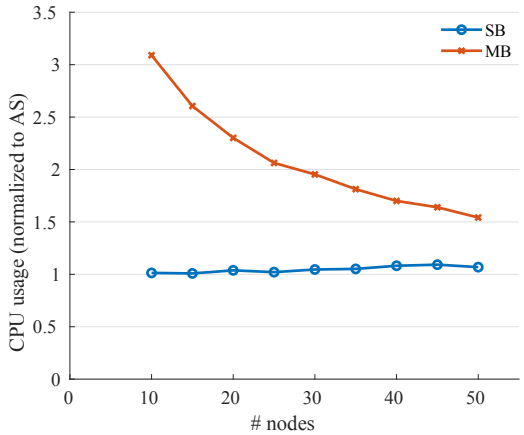


Fig. 9: CPU usage for SB and MB on Alexnet normalized to the CPU usage for AS.

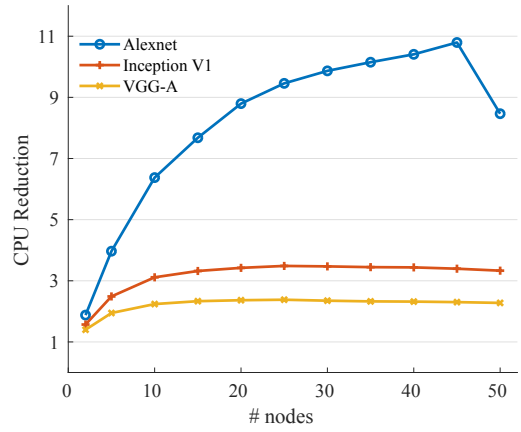


Fig. 10: CPU usage reduction for the All-Shared pipeline compared to the baseline. Each network/n combination was trained over 1000 steps.

the number of nodes in the ensemble between 10 and 50 and normalize the resulting CPU usage to the AS pipeline. The SB pipeline uses marginally more CPU than AS, while MB uses far more. This indicates that both of these pipelines are inferior to AS in both preprocessor throughput and CPU usage. Thus, our next series of tests are only performed on AS.

TABLE II: Specifications of the DNNs used for testing.

DNN	Batch Size	# Layers	# Params
Alexnet	128	8	60M
Inception V1	32	22	6.8M
VGG-A	32	11	133M

B. CPU

We use the Alexnet, Inception, and VGG models for DNN tests (see Table II). These models are commonly used for DNN benchmarking ([20]–[25]). For our tests, their different training speeds place different levels of stress on our pipelines.

Figure 10 shows the reduced CPU usage provided by the AS pipeline. We see that the usage is reduced by up to 10.8X, 3.5X, and 2.4X for Alexnet, Inception, and VGG, respectively. We observe that the reduction is inversely proportional to the compute demand of the network, as shown by the dotted lines in Figure 8. The compute demand is the primary indicator of how much CPU time is needed to preprocess data for the

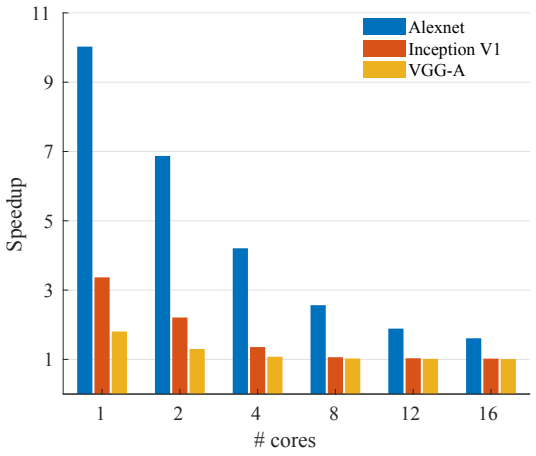


Fig. 11: Runtime improvement of AS over the baseline when CPU-core limits are imposed. The ensemble contained 100 networks, each trained over 1000 steps.

GPU. Higher demanding networks like Alexnet will cause the preprocessing stage to use much more CPU, while Inception and VGG will use less. Thus we see smaller reductions for larger/slower networks.

Aside from measuring CPU usage reduction, we also test training time when CPU limits are imposed. Figure 11 shows the speedup that AS provides when both AS and the baseline are subjected to core restrictions. Recall that each Titan node has a 16 core CPU.

Alexnet sees a speedup of up to 10X for 1 core allocation on the AS pipeline. To understand this, Table III provides information on how each pipeline slows down under core limitations. From this table, we see that Alexnet's speedup is due primarily to the dramatic slowdown that the baseline incurs (9.6X) from this limitation, since it relies on additional CPU power to preprocess data. In contrast, the AS pipeline only incurs a 54% slowdown due to the severe core limitation. While the AS pipeline's large number of individual processor cores should in theory be able to handle the necessary preprocessing, having only 1 core limits other systems as well from executing efficiently, thus causing the slowdown. However, the AS pipeline is able to train Inception and VGG on 1 core incurring only a 6% and 2% slowdown, respectively. Since less preprocessing is needed for these networks, less competition for CPU resources is present, allowing near-full-speed training. As with the CPU-reduction results, the potential speedups under core limitations is inversely proportional to the size of the DNN being trained. To reiterate, this is simply because larger networks need less CPU for preprocessing since they train slowly.

C. Energy Consumption

As seen in Table IV, the minimum energy for the Titan metered cabinet was found to be roughly 19KW, while the maximum energy observed for the most intensive job was

TABLE III: Slowdowns under a 1-core limitation, measured relative to the 16-core performance of the same DNN and pipeline.

Pipeline	DNN	1-core slowdown
Baseline	Alexnet	9.61X
	Inception	3.49X
	VGG	1.83X
All-Shared	Alexnet	1.54X
	Inception	1.06X
	VGG	1.02X

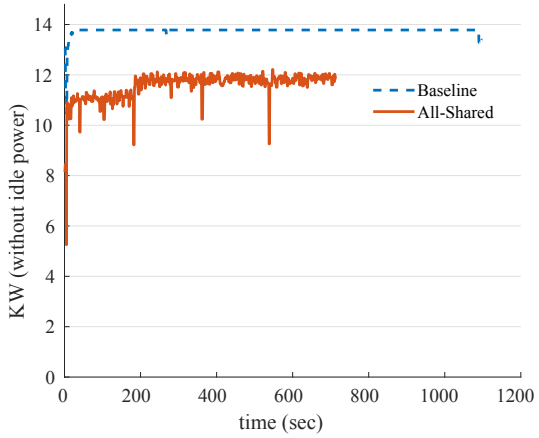


Fig. 12: Power draw comparison between AS and the baseline running 80 nodes of Alexnet.

32.767KW. Since the idling power is a significant 58% of the maximum power, savings will be reported based on the relative increase above the idling power.

Figure 12 shows the power consumption of the AS pipeline compared to the baseline when training 80 nodes of Alexnet. Since the baseline suffers from performance issues in its preprocessing, it takes more time to train its DNNs, and this is reflected in the figure.

Table V shows the average energy consumption in Kilo-Watts (KW) during training for Alexnet, Inception, and VGG on the baseline and AS pipelines. The energy demand for AS over the idle usage was 4.5%-15.8% less than for the baseline.

TABLE IV: The minimum energy usage for one of Titan's metered cabinets, averaged over 45 minutes of idle time with 1-second interval sampling.

Minimum power	Variance	Max observed power
18.985KW	5.355×10^{-4}	32.767KW

VII. Related Work

Recent work has tried to increase the scalability of machine learning algorithms in distributed environments. When discussing scalability, it is important to distinguish between a *single* network vs. *many* networks in distributed environments.

TABLE V: Average energy consumption during training for each of AS and the baseline on Alexnet, Inception, and VGG.

Pipeline	DNN	KW	KW (without idle)	Savings %
Baseline	Alexnet	32.745	13.760	
	Inception	31.318	12.333	
	VGG	31.509	12.524	
All-Shared	Alexnet	30.576	11.591	15.8%
	Inception	30.084	11.099	10.0%
	VGG	30.940	11.955	4.5%

Much research is being done to accelerate the training of a single large network over distributed systems. Google's *DistBelief* framework [26] is an example of this, as it provides a way to scale a very large network over potentially thousands of nodes. Li et al. [27] create a framework that maintains a set of global parameters while distributing data and workloads to a set of worker nodes. More recent work in this area has focused on specific cluster architectures and algorithms. Chung et al. [28] create an implementation of a data-parallel training algorithm that is designed specifically to scale well on a large number of loosely connected processors. They test their implementation on the IBM Blue Gene/Q cluster and find linear performance scaling up to 4096 processes with no accuracy loss.

Apache Spark [29] is also often employed to train either a single large model or many independent models, through the packaged MLLib library [30] or external libraries like TensorFlow [20] and H2O [31]. However, Spark provides limited functionality in controlling communication mechanisms to distribute the dataset among computational tasks.

The work by Kurth and others [32] is the most closely related to this study, where the authors considered DNN training in high performance computing environments. However, this study was performed on a cluster of Xeon-Phi processors, while our work used a large scale GPU cluster. In addition, that prior study focused on communication methods in the context of model parameter updates during the training process. Our study considers various communication methods in distributing training data, including preprocessing and I/O from the storage systems, in the context of DNN data pipelines.

Research has also made strides in accelerating networks designed to fit on a single device. Yu et al. [33] take a hardware-oriented approach by customizing weight pruning [11] to fit the underlying hardware being used. They note that hardware devices such as microcontrollers, CPUs, and GPUs have different execution patterns that are most efficient. They take advantage of this by carefully choosing when to prune out nodes or weights from the network. This results in 1.25-3.54X speedups depending on the hardware used.

Little work has been performed in the area of ensemble DNN training, as most researchers have focused on training a large DNN in distributed environments. However, Microsoft researchers have produced a tool called Adam [34] that has goals similar to [27] and partially relates to our work. Again, the tool primarily deals with accelerating larger networks over distributed nodes, but their pipeline has some elements in common with our current work. They mention concerns with heavy preprocessing tasks caused by complex image transformations. They similarly offload these tasks to a set of nodes dedicated to queuing preprocessed data to feed worker nodes more efficiently. Nevertheless, their work optimizes the training of a single DNN over multiple CPU-based machines. As GPU-based distributed systems can train similar models with much smaller cluster configuration than CPU-based systems [13], this study focuses on the potential gains from more efficient pipelines for multi-DNN systems in distributed GPU environments.

[11] Weight pruning involves analyzing a network during the training process to see if any of the network's nodes or weights are redundant or useless. Pruning can result in smaller memory footprint and faster training, but can also potentially reduce accuracy.

VIII. Conclusion

This research investigated the performance properties of DNN ensemble pipelines. We modified the Horovod library to provide additional communication flexibility to Tensorflow that is not present in other Deep Learning frameworks. Leveraging this tool, we developed a series of pipelines which eliminated redundant preprocessing operations. The best of these was selected based upon its ability to supply the most preprocessed data while requiring minimal CPU resources.

The All-Shared pipeline was able to reduce CPU usage by 2-11X when more than 5 nodes were present in the ensemble, while providing nearly twice the throughput that Alexnet demanded. Under CPU core restrictions, the AS pipeline was able to achieve up to 10X speedups over the baseline. Lastly, this pipeline uses 5-16% less energy on Titan than our baseline used.

This work assumes that DNNs in the ensemble behave similar to one other, with respect to both the training rate and prediction accuracy, and every model fits in one compute node. A future direction to extend is to consider the more complicated cases where models vary in size and training speeds, and some may even require parallel training on multiple nodes. Ultimately, we envision this research aligning with the broader goal of creating an adaptive machine learning pipeline that provides portable performance across system architectures.

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grants No. CCF-1455404, CCF-1525609, CNS-1717425, CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

REFERENCES

[1] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *arXiv preprint arXiv:1707.06342*, 2017.

[2] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[3] J. Lin and A. Kolcz, "Large-scale machine learning at twitter," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 793–804.

[4] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 2016, pp. 171–180.

[5] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[6] K. Choi, G. Fazekas, K. Cho, and M. Sandler, "A comparison on audio signal preprocessing methods for deep neural networks on music tagging," *arXiv preprint arXiv:1709.01922*, 2017.

[7] D. Figo, P. C. Diniz, D. R. Ferreira, and J. M. Cardoso, "Preprocessing techniques for context recognition from accelerometer data," *Personal and Ubiquitous Computing*, vol. 14, no. 7, pp. 645–662, 2010.

[8] A. Bland, W. Joubert, D. Maxwell, N. Podhorszki, J. Rogers, G. Shipman, and A. Tharrington, "Titan: 20-petaflop cray xk6 at oak ridge national laboratory," *Contemporary High Performance Computing: From Petascale Toward Exascale, CRC Computational Science Series. Taylor and Francis*, 2013.

[9] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring hybrid memory for gpu energy efficiency through software-hardware co-design," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 93–102.

[10] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping," in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010, pp. 115–126.

[11] Titan specs: https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/. [Online]. Available: https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/

[12] Summit specs: https://www.olcf.ornl.gov/summit/. [Online]. Available: https://www.olcf.ornl.gov/summit/

[13] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[14] "Tensorflow-slim," https://github.com/tensorflow/models/tree/master/research/slim.

[15] S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton, "Optimizing deep learning hyper-parameters through an evolutionary algorithm," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2015, p. 4.

[16] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2012.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[18] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," *arXiv preprint arXiv:1803.04014*, 2018.

[19] Distributed Tensorflow: https://www.tensorflow.org/deploy/distributed. [Online]. Available: https://www.tensorflow.org/deploy/distributed

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[21] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, vol. 472, p. 473, 2016.

[22] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[23] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on*. IEEE, 2015, pp. 5353–5360.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[25] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[26] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[27] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 1, no. 10.4, 2014, p. 3.

[28] I.-H. Chung, T. N. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, and B. Kingsbury, "Parallel deep neural network training for big data on blue gene/q," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1703–1714, 2017.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[30] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[31] H2O, open source machine learning platform. [Online]. Available: https://www.h2o.ai/

[32] T. Kurth, J. Zhang, N. Satish, E. Racah, I. Mitliagkas, M. M. A. Patwary, T. Malas, N. Sundaram, W. Bhimji, M. Smorkalov *et al.*, "Deep learning at 15pf: supervised and semi-supervised classification for scientific data," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 7.

[33] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 548–560.

[34] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system." in *OSDI*, vol. 14, 2014, pp. 571–582.

[35] "Project source code," https://github.com/rbpittman/ensemble.

[36] R. Pittman, "Horovod groups," https://github.com/rbpittman/horovod, 2018.

APPENDIX
ARTIFACT DESCRIPTION

In this section we provide a rough outline of our codebase deployment procedure, as well as some of the details of implementation that could affect the reproducibility of the project.

A. Code Sources and Dependencies

There are two main code repositories for the project. The primary code scripts for Tensorflow are contained in [35]. In order to run Single-Broadcast or Multi-Broadcast pipelines, Horovod Groups [36] is needed. For the All-Shared pipeline, Horovod is needed. To swap out these Horovod installations, modify your `PYTHONPATH` to include the correct installation directory depending on the pipeline being used.

Horovod Groups requires that the MPI installation can run in multi-threaded mode. That is, it initializes

MPI using `MPI_Init_thread`, and requests `MPI_THREAD_MULTIPLE` for its thread support. In order to run tests with the SB and MB pipelines using Horovod Groups, you will need to verify that your system supports multi-threaded MPI.

Our Python code calls several system commands, including the `mkdir`, `date`, and `mpstat` commands. These are used for log directory creation for various ranks, timestamp retrieval (for the energy tests), and CPU usage statistics. These commands will need to be installed on the system.

Our cluster job scripts all use the `.pbs` extension. These scripts are specific to our system/directory configuration on Titan, and cannot be used in other locations. Since each cluster has its own method for running various MPI/Tensorflow code, you will need to construct a new set of job scripts for your cluster.

Note that our implementation on Titan used a Singularity installation of Tensorflow, through the recommendation of OLCF staff. Therefore, many of our job scripts contain singularity command wrappers. Additionally, the Titan compute nodes and default Tensorflow Singularity image do not feature the `mpstat` command. Thus, we built our own Singularity image to include this command.

B. Compilation

The only compilation needed for this project is Horovod and Horovod Groups, which can both be built using the same commands. Since no Tensorflow modifications were made for this project, your own installation of Tensorflow should work. Note that since Tensorflow is rapidly changing, it is quite possible that installations newer than 1.3.0 will have unexpected errors.

C. Reproducibility

1) Preprocessing Throughput Tests: We ran a series of tests on the capabilities of each pipeline, measuring what we refer to in this paper as the peak(n) function for the pipeline. Depending on your MPI installation and particularly on your cluster's network architecture, you will likely see differences in each pipelines overall performance. However, we do expect that the relationships between AS, MB, and SB will remain the same.

2) Multi-Core CPU Tests: This work relies heavily on multi-core Tensorflow execution. It will be important to ensure that your MPI/cluster configuration will allow MPI ranks to use multiple CPU cores.

Our various CPU tests also rely on the ability of the system to restrict core usage. This might be accomplished by reserving only a subset of the CPU cores within your cluster, but this was not possible on Titan since users can only reserve entire compute nodes. We used the `-d` flag to limit the number of cores MPI ranks could use for our tests. Whatever the means, reproducing the CPU core limitations tests will require this capability.

The Slim module by default includes some preprocessing capabilities. This module also provides a `fast_mode` flag for preprocessing. If this flag is true, the module will select a random resizing algorithm to shrink the raw input image to the correct DNN dimensions. Some of these algorithms are more costly than others. We left `fast_mode` disabled to provide the best preprocessing capabilities for DNN training. This also increases CPU usage, and is therefore important to more precisely reproduce our results.

The largest speedup reported in this paper is about 10X for Alexnet. This value depends on the cluster's individual GPU and CPU performance. On Titan, we observed that the 16-core CPU is not able to keep up with the GPU for the Alexnet DNN, experiencing a 34% slower training speed than the GPU can handle. A system with higher CPU to GPU power would see less performance degradation, and thus the reported 10X speedup would be less. However, a system with more GPU power would see higher CPU usage and slower training due to preprocessing, resulting in greater speeups. Overall, the CPU to GPU power ratio will be different for your system, and will likely produce different speedups. Nevertheless, we expect that the AS pipeline will typically be capable of producing speedups for the 1-2 core case.

3) Energy Tests: Our energy tests required extended communication and assistance from OLCF staff. For our tests, we ensured that the 80-node job allocations were sent to only 1 metered cabinet. Our reported Kilo-Watt values are for the *entire* cabinet, which contained 96 nodes. Since we discovered a significant 58% idle power usage for this cabinet, we decided to factor this out of our savings reports. Thus, in order to get an accurate reproduction of these results, the idle power consumption for your testing system will need to be obtained.