ORNL/TM-2018/817

Exploring Flexible Communications for Streamlining DNN Ensemble Training Pipelines

Randall Pittman
Xipeng Shen
Robert M. Patton
Seung-Hwan Lim

Mar 28, 2018

**Approved for public release.
Distribution is unlimited.**

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2018/817

Computer Science and Mathematics Division

Exploring Flexible Communications for Streamlining DNN Ensemble Training Pipelines

Randall Pittman[1], Xipeng Shen[2], Robert M. Patton, Seung-Hwan Lim

Date Submitted: Mar, 2018

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

[1] North Carolina State University, rbpittma@ncsu.edu
[2] North Carolina State University, xshen5@ncsu.edu

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

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.

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. Existing ensemble training pipelines can 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, Titan supercomputer at Oak Ridge National Lab. Our results show that 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.

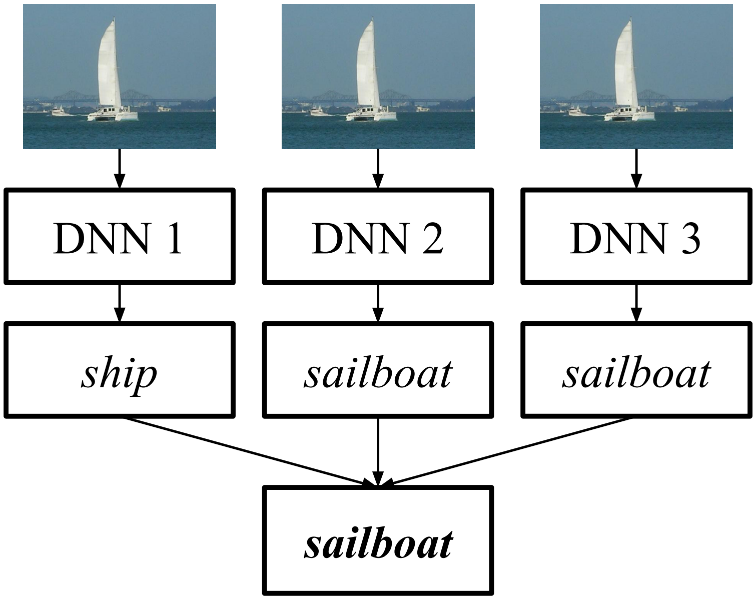


Figure 1. Illustration of the inference stage of a DNN ensemble of size 3 for an image of a sailboat.

1. Introduction

Machine learning enables the discovery of actionable knowledge from large quantities of data. In the training of machine learning models, including Deep Neural Networks (DNNs), machine learning algorithms process a few samples of data in each training iteration for multiple iterations to cover the entire data set until convergence [13]. As machine learning techniques evolve, more advanced usages have appeared such as the *ensemble* of machine learning models [13, 17], where different models learn from the same data set and aggregate their prediction results to produce a more accurate final prediction, as demonstrated in Figure 1. Such ensemble training multiplies the I/O and CPU demand on an already burdened system, since it duplicates 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 prepared 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.

Since each DNN model is trained over the same data, the preprocessing operations (e.g., resizing and cropping) being performed for each DNN are redundant. This stage can be CPU-intensive depending on the type of operations being performed, the dataset being used, and the DNN model to be trained. The result is unnecessarily high CPU usage for every compute node in the ensemble training, slowing-down the DNN training time. It is because the rate of preprocessing on the CPU side cannot keep up with the demand from the GPU to train a model over prepared data. In addition, the excessive CPU usage is likely to increase the power consumption of the ensemble training.

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 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 develop a group-based collective communication addition to the Horovod [21] library. Using this addition with Tensorflow, we examine three pipeline designs that we refer to as All-Shared, Single-Broadcast, and Multi-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 as an initial study of different primitive pipeline communication schemes.

Among these pipelines, the All-Shared scheme provides significantly more efficient parallel ensemble training. Our experimental results show that the preprocessing stage can indeed form a bottleneck for the Alexnet DNN, producing 96% CPU usage with 34% degraded training time on Titan supercomputer, a large scale GPU cluster at Oak Ridge National Lab. Our best optimized pipeline can meet and exceed Alexnet's preprocessing demand by up to 2X. We furthermore reduce CPU usage by 2-11X depending on the DNN being trained. Lastly, we provide experimental results on Titan showing that our best pipeline uses 5-16% less energy during training.

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 3.)

2. It adds into existing DNN ensemble training pipelines with flexible communication controls. (Section 4.)

3. It provides the first known exploration of distributed communication schemes for streamlining parallel ensemble training pipelines in large distributed environments. (Section 4.)

4. It offers a thorough performance analysis of the capabilities of these pipelines on the Titan supercomputer. (Section 6.)

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.

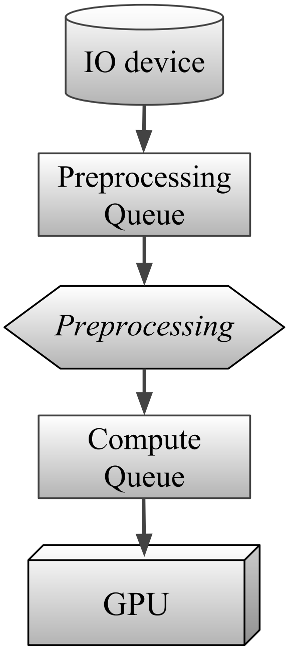


Figure 2. A typical pipeline for DNN training.

2. Backgrounds

2.1 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 2). 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.

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 datatypes, not exclusively images. Both audio [8] and sensor [12] 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.

2.2 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 [6, 22, 25]. 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 [1]. 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 [2]. 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 [10]. 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 copies to GPU memory 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. 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.

3. 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.

3.1 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. We use the Slim module [5] as a starting point, since it

All Tensorflow code is version 1.3.0.

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.

3.2 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.

3.2.1 Workloads

In general, parallel model training can be used as a fast method for hyper-parameter tuning [23], or it can be used to create multiple learners for increased classification accuracy, or to learn an ensemble model [13, 17]. 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 [13], 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.

3.2.2 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 will likely require very little resources and will train quickly. The primary dataset used for this research is a subset of *ImageNet* [20], 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.

3.3 Baseline

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

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

Our subset contains approximately 1.3 million images.

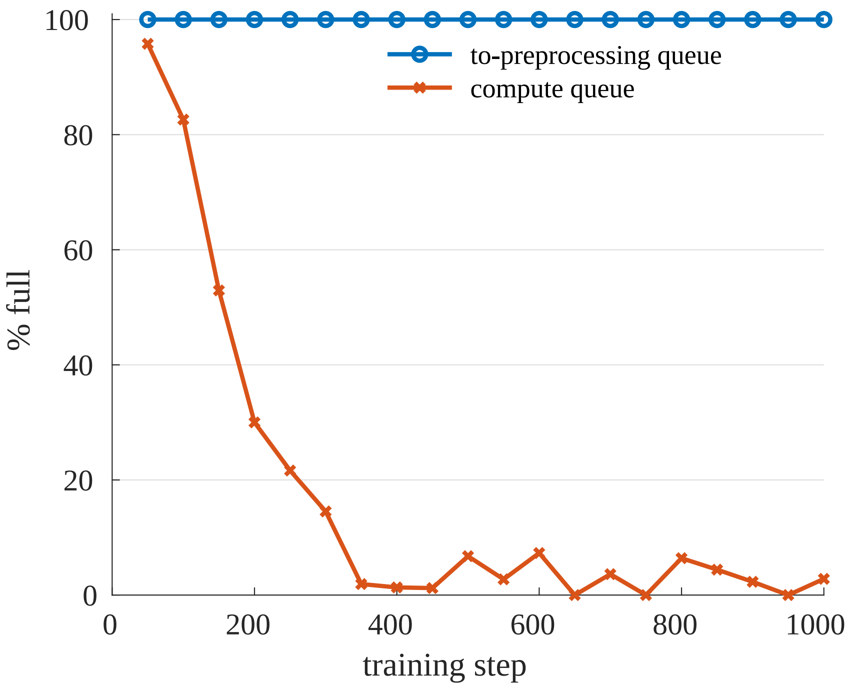


Figure 3. For the default single pipeline, the preprocessing queue is always full, while the compute queue empties quickly. Thus the preprocessing task is the bottleneck.

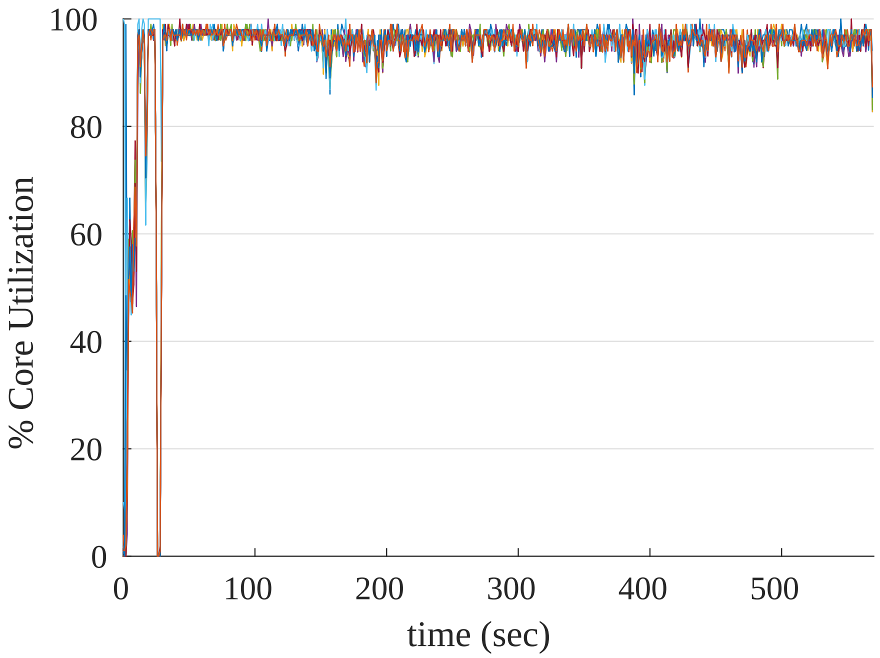


Figure 4. Default single pipeline core utilization for each of the 16 cores on a single Titan node when training Alexnet. The average core utilization over the entire graph is 94.3%. When the startup phase is excluded, the average is 96.0%.

3.3.1 Single Node

We begin with a performance evaluation of the default pipeline shown in Figure 2 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 large batch size of 128 [14].

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 *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.

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 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 3 shows the measured size of the preprocessing and compute queues during the training process. As shown before in Figure 2, 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.

It is important to show that the preprocessing uses the entire CPU. Figure 4 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].

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.

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

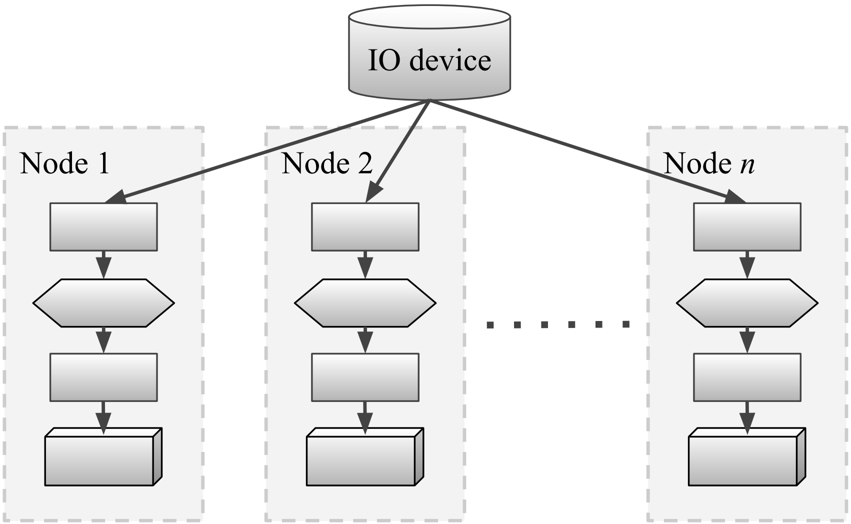


Figure 5. Duplicated pipelines that can be used to concurrently train DNNs.

3.3.2 Multiple Nodes

The natural extension to the single pipeline in Figure 2 is to duplicate each pipeline for each DNN in an ensemble. This duplicated pipeline is shown in Figure 5. 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. High IO usage could in theory lead to scalability problems. Second, the preprocessing operations are redundant since the same data is being modified. While this does not present scalability problems, it does result in unnecessary CPU usage. As shown earlier in Figure 4, 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 1 shows the results of executing 2000 steps of Alexnet on 1000 nodes of Titan in parallel, as well as the results of executing 50 nodes individually. While the 1000 nodes exhibited slightly higher variance in its runtimes, the overall

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.

Table 1. Statistics comparing the total run time for 50 solo runs and a parallel run of 1000 nodes for 2000 Alexnet steps.

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

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.

4. 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. Fortunately, ensemble training provides access to more CPU power for the same data, thereby yielding an opportunity to accelerate the preprocessing stage.

4.1 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. Therefore, the peak preprocessing throughput for each node becomes a function of n, say peak(n) $\leq nr_p$. As n increases, the upper limits of peak(n) depend on the communication pattern among nodes for preprocessing and the network capabilities of the cluster. To address this issue, we need to consider flexible pipeline designs in order to accelerate the progress of the pipeline. 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.

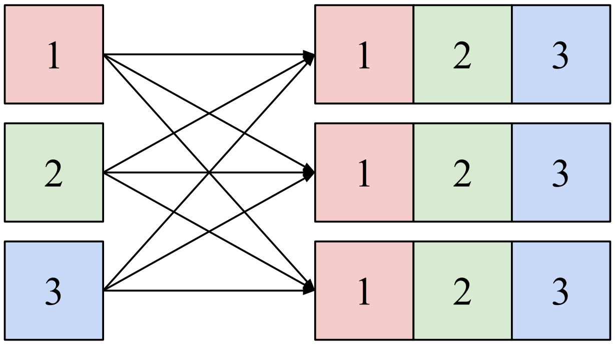


Figure 6. Visualization of the MPI all-gather collective.

4.2 Horovod groups

Horovod [21] is a distributed deep-learning library for Tensorflow. Although distributed Tensorflow [3] provides implicit tensor sends and receives, it does not provide collective operations. 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 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 developed *Horovod Groups* [19]. This modification allows the user to provide a list of groups that should be created upon initialization of the library. Whenever a collective tensor is created, a group index must then be provided indicating which communicator to use for the operation. At present, there are no known constraints on the memberships within these groups. For example, two groups need not be mutually exclusive. A particular rank launches a background MPI thread for each group to which it belongs. Communication can then occur asynchronously using multi-threaded MPI.

4.3 All-Shared

To share preprocessed data with all nodes, one possible approach is to make every node a preprocessor ($n = p$), and share each node's data with all other nodes. The MPI all-gather operation (see Figure 6) is well suited to this purpose. We refer to this as the *All-Shared* (AS) pipeline, as depicted in Figure 7.

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. Our next two designs attempt to take advantage of this fact, thereby increasing their *flexibility*.

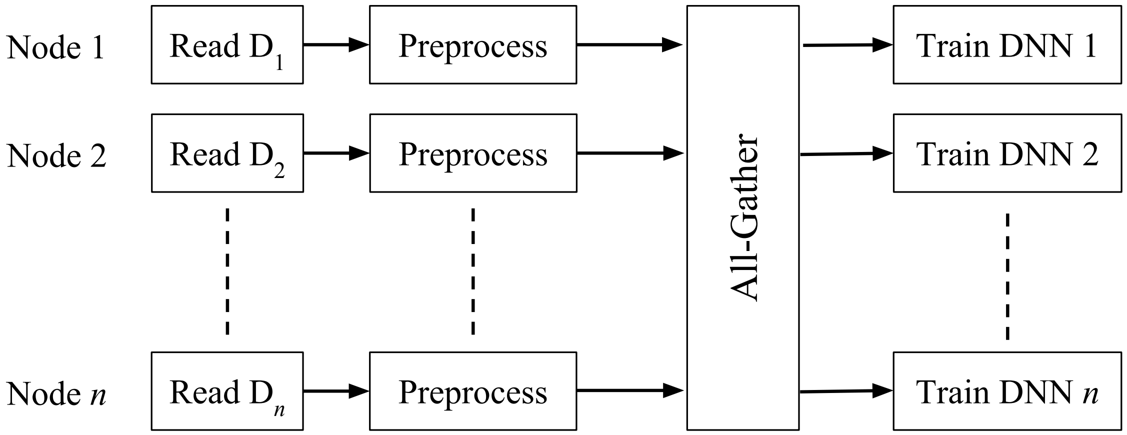


Figure 7. Illustration of the All-Shared (AS) pipeline. The dataset D is divided into n partitions for each reader.

4.4 Single-Broadcast

We now wish to allow the number of preprocessors p to be adjustable. Suppose nodes $1, \ldots, p$ are the preprocessor nodes, and $p + 1, \ldots, n$ are nodes that only contain the GPU's compute stage. Presumably $p < n$, since if $p = n$ we could use the All-Shared technique.

As a first step, we can perform an all-gather between the preprocessor nodes $1, \ldots, p$. Now each of these nodes 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 8, and we refer to this pipeline as *Single-Broadcast* (SB).

The benefit of this pipeline is increased flexibility over AS. We can now control the value of p to adjust the pipeline as necessary to our particular application. The primary downside to this design is potentially degraded performance, since rank p now needs to perform two collective operations. Additionally, Horovod Groups is needed for its custom MPI communicators.

4.5 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 9.

The benefit of this design is its evenly distributed approach. Each preprocessing node has identical work without the extra demand placed on rank p by Single-Broadcast. However, it is limited in the number of preprocessing nodes it can create efficiently, since each broadcast operation needs to occur within its own thread.

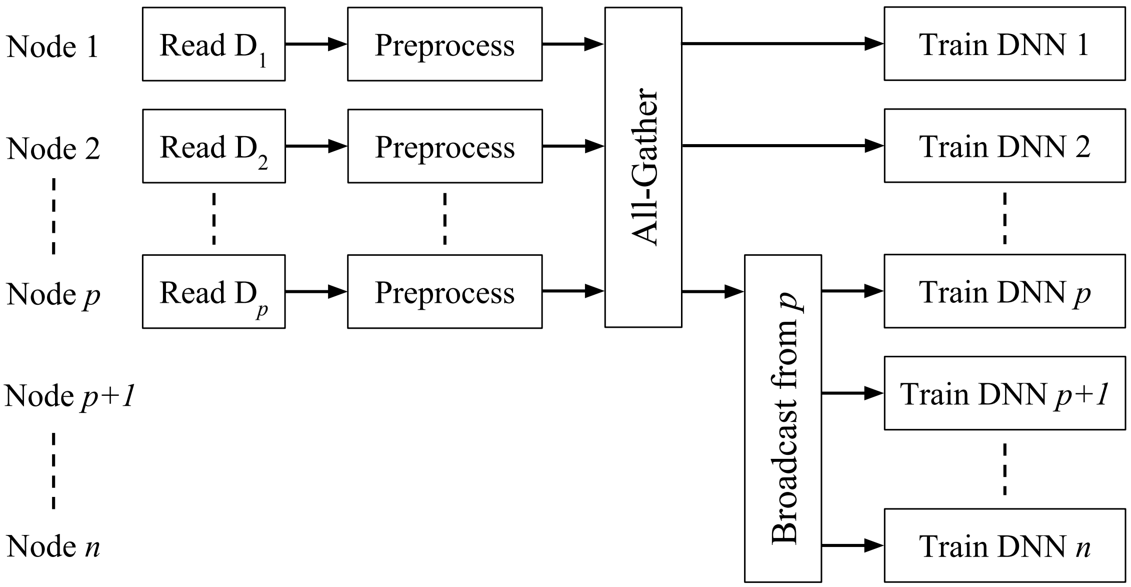


Figure 8. Illustration of the Single-Broadcast (SB) pipeline. The dataset *D* is divided into p partitions for each reader instead of n, since there are now p readers feeding their own preprocessor.

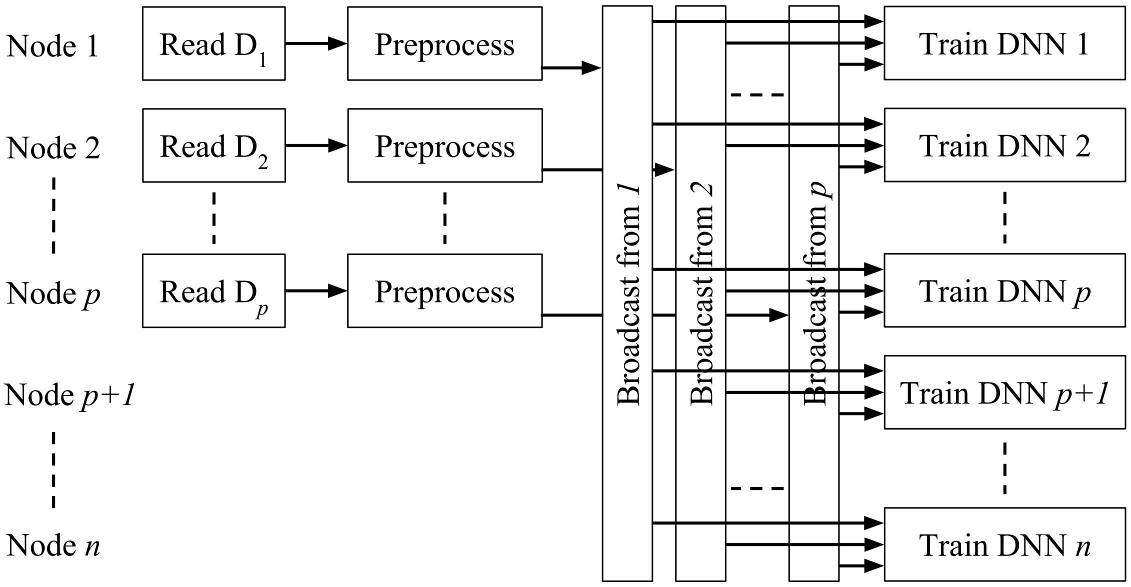


Figure 9. Illustration of the Multi-Broadcast (MB) pipeline. Similar to the Single-Broadcast design, the dataset is divided into p partitions. Each D_i for $1 \leq i \leq p$ is broadcasted to all nodes with the i'th preprocessor node as the root.

5. Methods

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

5.1 Peak Preprocessor Throughput

Previously we defined peak(n) as the function representing the maximum image throughput in a pipeline for a given number of nodes n. 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.

While it is easy to think of peak(n) as a single function, it is actually defined by the throughput at each node in the ensemble. However, it turns out that for a queuing system with finite size, the long term average throughput for each node should be the same. Thus only one function is needed to define the entire pipeline's peak throughput.

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.

5.2 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.

The long term averages are identical simply due to the nature of the collective communication. All nodes in the pipeline receive all data. If node a gets ahead of node b in its computation, the images that b has not processed must be within a queue after the collective communication. Since these queues have finite size, the difference in progress between a and b must be less than this constant. Thus the long term average throughput must be the same.

Table 2. Average core usage when using simulated 3 core allocation on a Titan node.

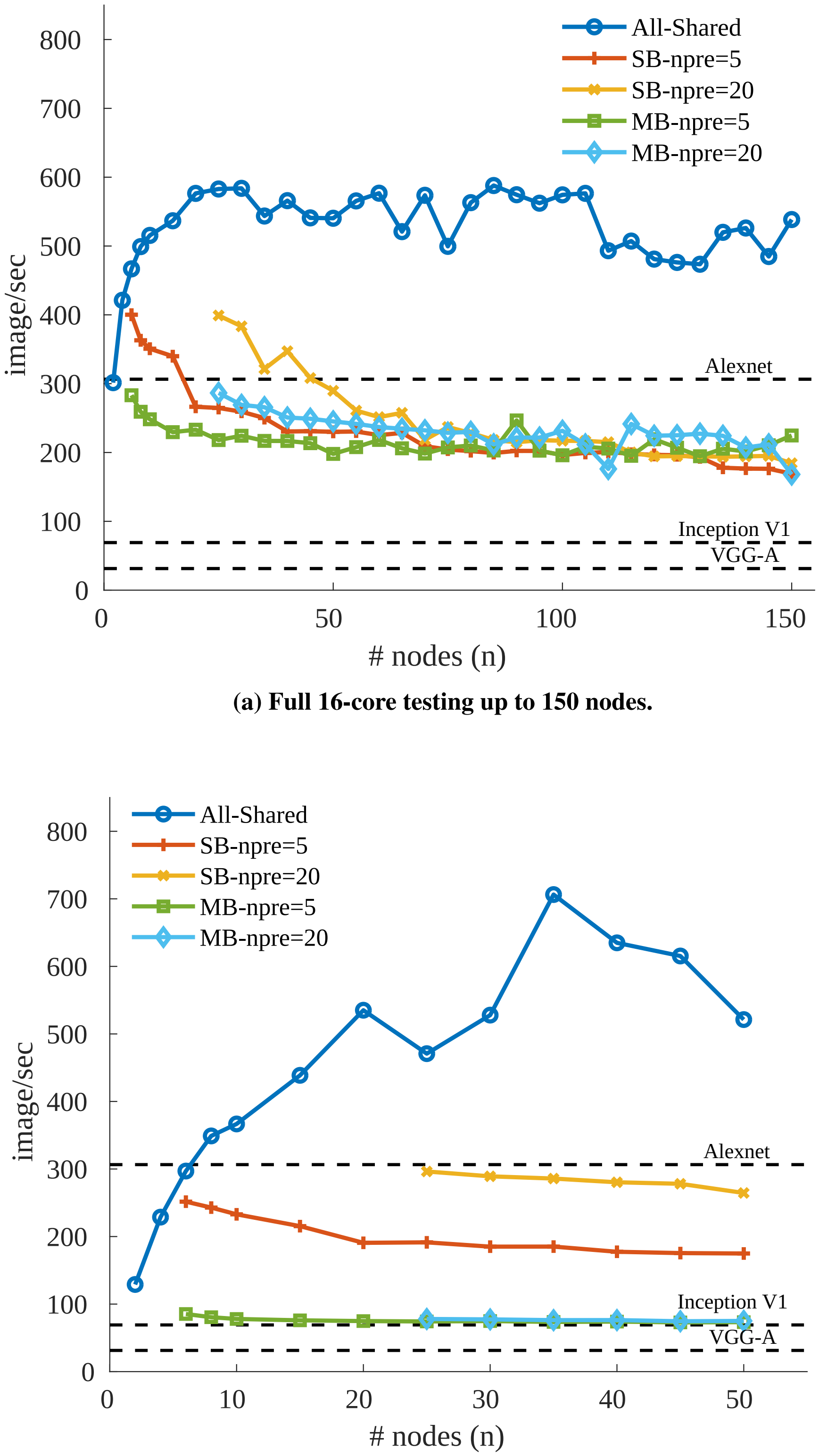
Core ID	Avg % Util	Core ID	Avg % Util
0	94.0181	8	0.0040
1	96.4045	9	0.0080
2	94.5611	10	0.0040
3	0.0436	11	0.0040
4	0.3789	12	0.1432
5	0.0080	13	0.0040
6	0.6070	14	0.0040
7	0.0079	15	0.1352

5.3 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. The charge allocated to the user's account for such a job is typically only charged for the number of cores allocated. In such a case, there is a clear benefit to allocating less cores if the job does not need them. We can therefore test our pipeline by first imposing limits on the number of cores used, and then compare 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. Since each rank is allowed to use an entire node, the number of threads corresponds to the number of CPU cores allowed. Table 2 shows the average core usage when simulating 3 cores allocated on a single Titan node and training Alexnet on a basic pipeline.

5.4 Energy Usage

A secondary benefit from decreased CPU usage is power savings. On Titan, we collect energy consumption data through 2 metered cabinets. 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.



(a) Full 16-core testing up to 150 nodes.

(b) Partial 4-core test up to 50 nodes.

Figure 10. Illustration of peak(n) for each pipeline. The number of preprocessors is changed between 5 and 20 when supported by the pipeline. The horizontal lines indicate the measured compute demand of a GPU on Titan.

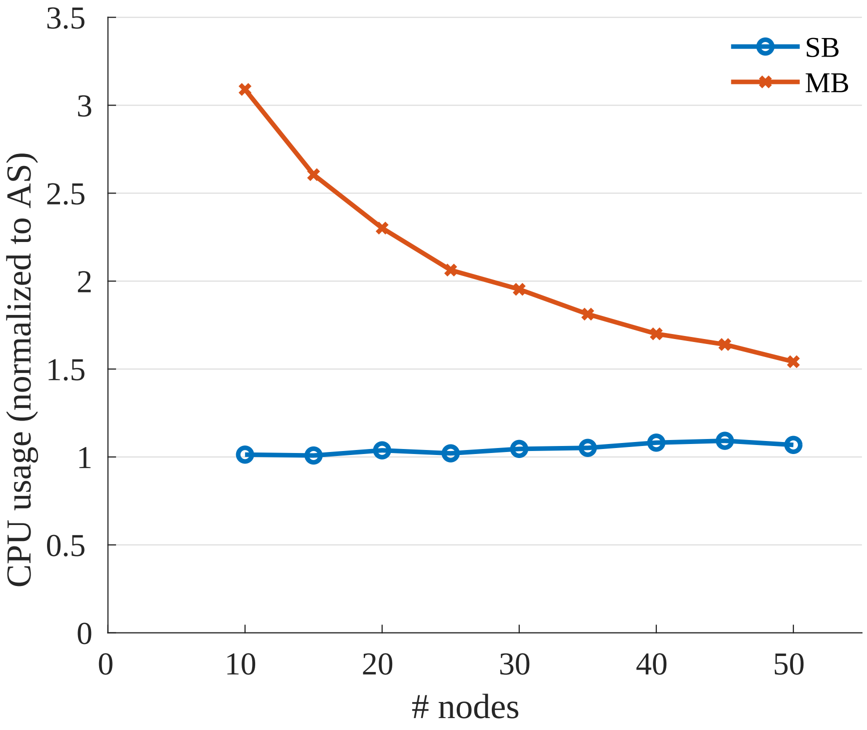


Figure 11. CPU usage for SB and MB on Alexnet normalized to the CPU usage for AS.

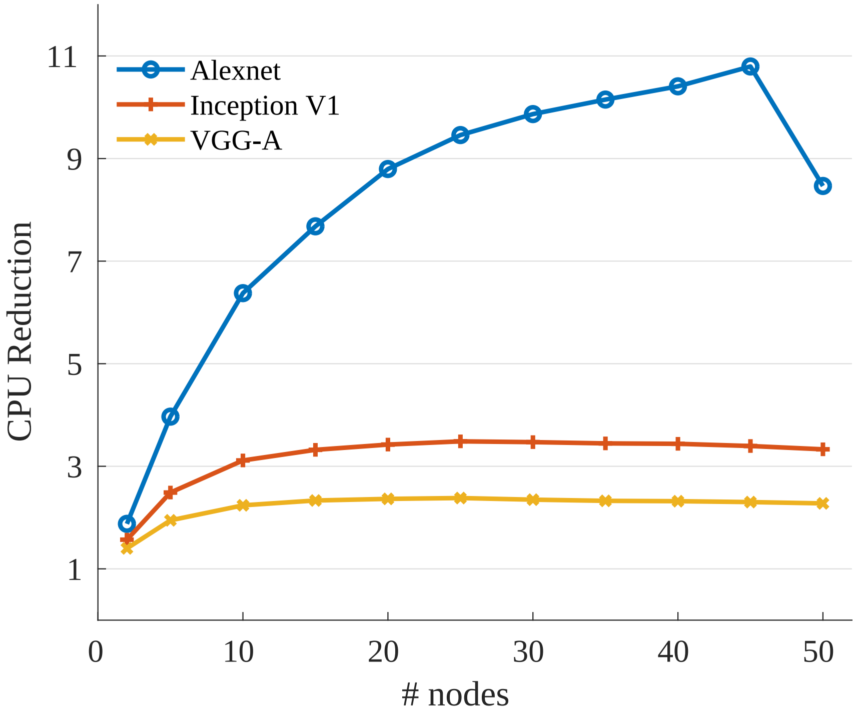


Figure 12. CPU usage reduction for the All-Shared pipeline compared to the baseline. Each network/*n* combination was trained over 1000 steps.

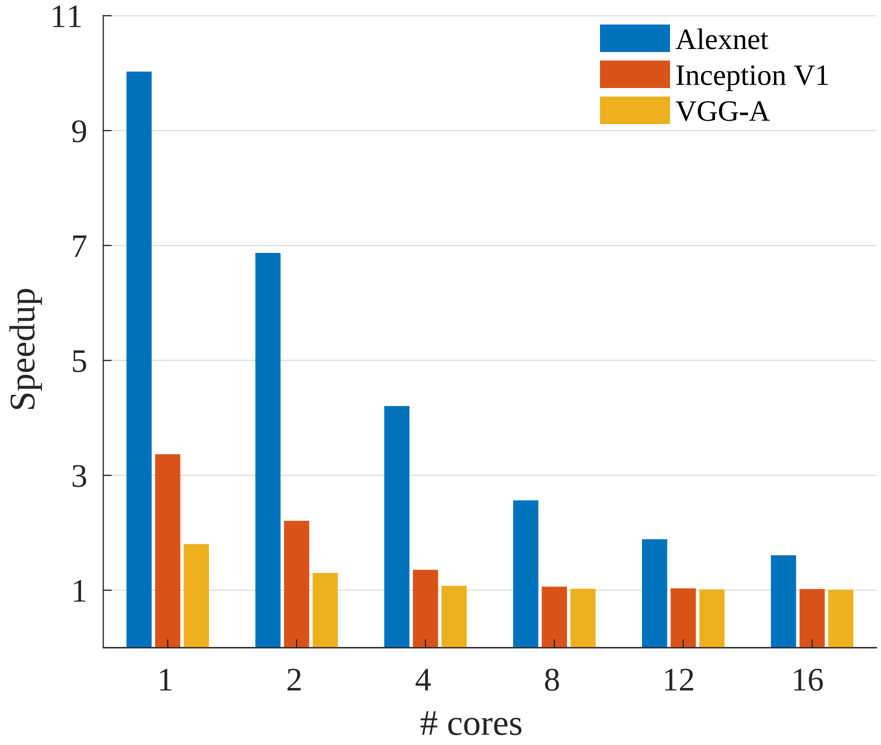


Figure 13. Runtime improvement of AS over the baseline when CPU-core limits are imposed. The ensemble contained 100 networks, each trained over 1000 steps.

6. Results

6.1 Peak 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, and does not include any DNN training.

Figure 10a shows the value of peak(n) for $n <= 150$ for increments of 5 nodes. Since the SB and MB pipelines each also need p preprocessors, technically we need to illustrate peak(n, p). For simplicity, the figure shows peak(n, 5) and peak(n, 20). 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 Alexnet, since they drop below its r_c line. Despite their poor performance, they still provide a viable mechanism to train larger networks, as both Inception and VGG are well within their compute demands.

In order to clarify this data when core usage is restricted, Figure 10b shows peak(n) for up to 50 nodes. 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 10.

As a final test for the broadcasting pipelines, we compare the CPU usage for AS, SB, and MB in Figure 11. We vary 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.

6.2 CPU

Figure 12 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 10. The compute demand is the primary indicator of how much CPU time is needed to preprocess data for the 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 13 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 3 provides information on how each pipeline slows down under core limitations. From this table, we see that

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

Each of these cabinets includes 96 nodes, 8 of which are service nodes, leaving a total of 88 nodes for user jobs.

Table 3. 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	1.06
	VGG	1.02X

Table 4. 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

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.

6.3 Energy Consumption

As seen in Table 4, 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 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 14 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 5 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.

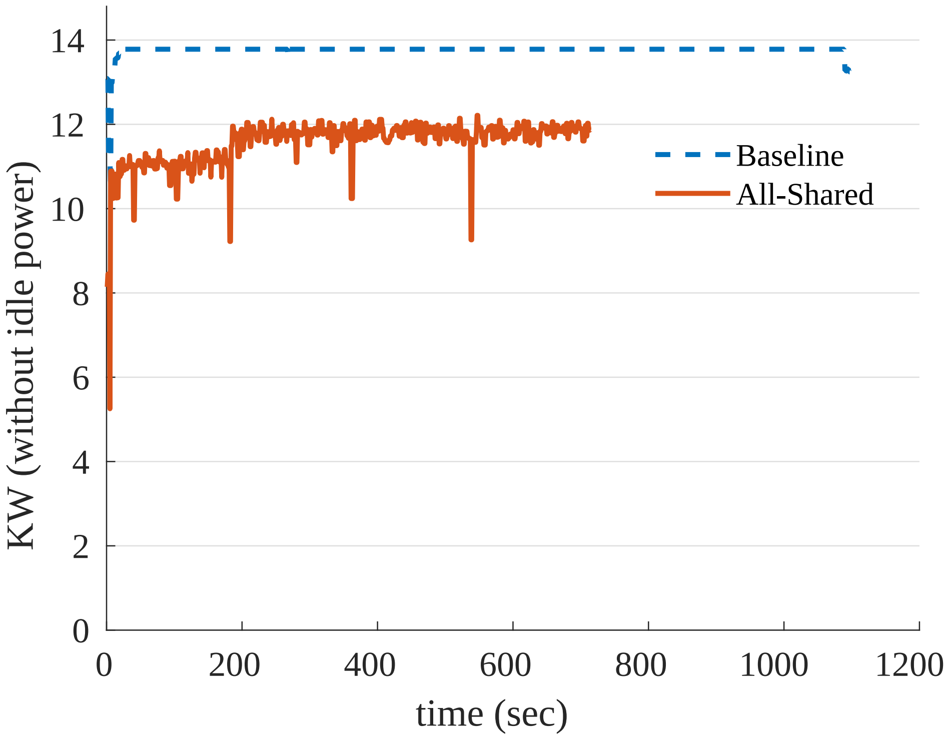


Figure 14. Power draw comparison between AS and the baseline running 80 nodes of Alexnet.

Table 5. 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%

7. 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.

Much research is being done to accelerate the training of larger networks over distributed systems. Google's *DistBelief* framework [11] is an example of this, as it provides a way to scale very large networks over potentially thousands of nodes. Li et al. [16] 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. [9] 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.

Among these studies, Kurth et al. [15] 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, Kurth et al. [15] focused on various communication methods in the context of model parameter updates during the training process. The study also considered various communication methods in distributing training data, including preprocessing and I/O from the storage systems, in the context of *machine learning pipelines*.

Research has also made strides in accelerating networks designed to fit on a single device. Yu et al. [24] take a hardware-oriented approach by customizing weight pruning 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 [7] that has goals similar to [16] 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 [10], this study focuses on the potential gains from more efficient pipelines for multi-DNN systems in distributed GPU environments.

8. 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

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.

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.

The primary limitation of this work is the assumption that DNNs in the ensemble behave similar to one other, with respect to both the training rate and prediction accuracy. In the future, a more in-depth study on the communication methods with CUDA-enabled MPI will be an interesting direction to pursue. We envision that this research will align with the broader goal of creating an adaptive machine learning pipeline that provides portable performance across system architectures.

References

[1] Titan specs: `https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/`.

[2] Summit specs: `https://www.olcf.ornl.gov/summit/`.

[3] Distributed Tensorflow: `https://www.tensorflow.org/deploy/distributed`.

[4] Project source code. (Omitted due to double-blind review).

[5] Tensorflow-slim. `https://github.com/tensorflow/models/tree/master/research/slim`.

[6] 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.

[7] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582, 2014.

[8] Keunwoo Choi, George Fazekas, Kyunghyun Cho, and Mark Sandler. A comparison on audio signal preprocessing methods for deep neural networks on music tagging. *arXiv preprint arXiv:1709.01922*, 2017.

[9] I-Hsin Chung, Tara N Sainath, Bhuvana Ramabhadran, Michael Picheny, John Gunnels, Vernon Austel, Upendra Chauhari, and Brian Kingsbury. Parallel deep neural network training for big data on blue gene/q. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1703–1714, 2017.

[10] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[11] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[12] Davide Figo, Pedro C Diniz, Diogo R Ferreira, and João M Cardoso. Preprocessing techniques for

context recognition from accelerometer data. *Personal and Ubiquitous Computing*, 14(7):645–662, 2010.

[13] Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 171–180. IEEE, 2016.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[15] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail 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*, page 7. ACM, 2017.

[16] 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 *OSDI*, volume 1, page 3, 2014.

[17] Jimmy Lin and Alek Kolcz. Large-scale machine learning at twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 793–804. ACM, 2012.

[18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. *arXiv preprint arXiv:1803.04014*, 2018.

[19] Anonymous (omitted due to double-blind review). Horovod groups, 2018.

[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2012.

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

[22] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey 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*, pages 93–102. IEEE Press, 2013.

[23] Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and Robert M Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, page 4. ACM, 2015.

[24] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 548–560. ACM, 2017.

[25] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng 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*, pages 115–126. ACM, 2010.

APPENDIX A. Artifact Description

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

Code Sources and Dependencies

There are two main code repositories for the project. The primary code scripts for Tensorflow are contained in [4]. In order to run Single-Broadcast or Multi-Broadcast pipelines, Horovod Groups [19] 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.

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.

Reproducibility

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.

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.

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.