Profiling DNN Workloads on a Volta-based DGX-1 System

Saiful A. Mojumder[1], Marcia S Louis[1], Yifan Sun[2], Amir Kavyan Ziabari[3*],
José L. Abellán[4], John Kim[5], David Kaeli[2], Ajay Joshi[1]
[1]ECE Department, Boston University; [2]ECE Department, Northeastern University; [3]Advanced Micro Devices;
[4]CS Department, UCAM; [5]School of EE, KAIST;
{msam, marcia93, joshi}@bu.edu, {yifansun, kaeli}@ece.neu.edu,
amirkavyan.ziabari@amd.com, jlabellan@ucam.edu, jjk12@kaist.edu

***Abstract*—High performance multi-GPU systems are widely used to accelerate training of deep neural networks (DNNs) by exploiting the inherently massive parallel nature of the training process. Typically, the training of DNNs in multi-GPU systems leverages a data-parallel model in which a DNN is replicated on every GPU, and each GPU performs Forward Propagation (FP), Backward Propagation (BP) and, Weight Update (WU). We analyze the WU stage that is composed of collective communication (e.g., allReduce, broadcast), which demands very efficient communication among the GPUs to avoid diminishing returns when scaling the number of GPUs in the system. To overcome this issue, different data transfer mechanisms and libraries have been introduced by NVIDIA, and adopted by high-level frameworks to train DNNs. In this work, we evaluate and compare the performance of peer-to-peer (P2P) data transfer method and NCCL library-based communication method for training DNNs on a DGX-1 system consisting of 8 NVIDIA Volta-based GPUs. We profile and analyze the training of five popular DNNs (GoogLeNet, AlexNet, Inception-v3, ResNet and LeNet) using 1, 2, 4 and 8 GPUs. We show the breakdown of the training time across the FP+BP stage and the WU stage to provide insights about the limiting factors of the training algorithm as well as to identify the bottlenecks in the multi-GPU system architecture. Our detailed profiling and analysis can help programmers and DNN model designers accelerate the training process in DNNs.**

I. Introduction

GPUs have become the most commonly used devices for training Deep Neural Networks (DNNs) [5], [16], [21], [31]. Thanks to their massively parallel computing capabilities, GPUs can train a DNN several hundred times faster than CPUs [7], [22], [34], [45]. As neural networks grow deeper and training datasets become larger, a single GPU requires several days (sometimes even weeks) to train a DNN. Hence, multi-GPU systems are being introduced to achieve faster training of DNNs [24], [41].

Training a DNN on a multi-GPU system introduces new challenges. First, the programmer has to distribute the data (input data and network model data) among multiple GPUs. The programmer can either distribute the input data onto multiple GPUs while replicating the network model in each of the GPUs [41] (referred to as data parallelism), or assign different parts of the neural network model to distinct GPUs (referred to as model parallelism) [41]. Both approaches require data to be transferred and synchronized across GPUs. The input data is fed to the GPUs as mini-batches. Each mini-batch consists of a certain number of unique inputs chosen by the programmer from the dataset. The mini-batch size has implications on training time, GPU memory usage and training accuracy. Recent works [15], [37], [43] have shown that batch size can be increased without losing accuracy. Hence, in our work we do not consider accuracy as a limiting factor for increasing batch size, rather we analyze the effects of increasing batch size on training time, GPU-to-GPU communication and GPU memory usage. Finally, although we can parallelize the computation required for training DNNs, the GPUs still need to communicate with each other during the different phases of training. High-end multi-GPU systems support methods and libraries for communication. Depending on the size of neural networks, communication can pose significant bottlenecks. To minimize the communication time, both hardware-level (i.e. NVLinks) and software-level (i.e. NCCL library) solutions have been introduced. In our work, we evaluate the effectiveness of these solutions.

Between the two parallelism models mentioned above, data parallelism is more suitable for a network with more convolution layers than fully-connected layers, while model parallelism is more suitable for networks with more fully-connected layers than convolution layers [20]. As DNN models grow deeper, convolution layers dominate execution in the model, while the fully-connected layers are only used in the last few layers.[1] Hence, data parallelism is the frequently used model for distributing a workload across multiple GPUs. According to the survey by Fox et al. [12], all the DNN frameworks support data parallelism, while only half (4 out of 8 frameworks) support model parallelism. Therefore, in our work, we focus on the data-parallel approach.

We present the limits and opportunities for training the following five DNN workloads: GoogLeNet, AlexNet, Inception-v3, ResNet and LeNet, on NVIDIA's Volta-based DGX-1 system. These workloads represent a wide mixture of computation and communication behavior. We present a thorough analysis of how well the training of various DNNs scales with GPU count on the DGX-1 multi-GPU system. We identify hardware-level

*This author completed the majority of this work before joining AMD.

[1]In this work, we focus on DNNs equipped with convolution layers, which are integral parts of the popular DNNs used in both academia and industry [26], [39].

(i.e. computation, communication and memory capacity) and software-level (i.e. NCCL library, the programming framework for DNNs, etc.) bottlenecks present in the training of the various DNNs using multi-GPU systems.

We have chosen MXNet [6] out of the several DNN training frameworks (such as Caffe [17], Tensorflow [2], and Torch [8]) that support training on multi-GPU systems. Since all the frameworks use standard CUDA libraries (such as cuDNN [28], cuBLAS [30], NCCL [1], etc.), they achieve similar performance for training DNN workloads on multi-GPU system [36].

In our analysis, we profile the three stages– Forward Propagation (FP), Backward Propagation (BP), and Weight Update (WU) of the DNN training process. In prior work [11], [36], researchers have profiled and characterized the FP and BP stages in detail. However, to the best of our knowledge, we are the first to profile DNN workloads on an NVIDIA Volta-based DGX-1 system and analyze the WU stage. The WU stage is where GPUs communicate using a data-parallel model, and that is where the communication bottlenecks mainly arise [41]. In this work, we provide detailed measurements of the time spent in each stage of training, as well as the memory transfers that are involved in these stages by looking into both the peer-to-peer (P2P) data transfer method and NCCL library based communication method. The information about the behavior of the DGX-1 system in each stage can help us identify the bottlenecks and provide guidance on how to improve the performance of each stage.

The contributions of this work include:

- We compare the impact of P2P and NCCL based communication methods on the training time of DNN workloads (LeNet, AlexNet, GoogLeNet, ResNet and Inception-v3) on NVIDIA's Volta-based DGX-1 system. We profile these workloads to isolate and quantify the computation-intensive and the communication-intensive portion of the training process to identify the software and hardware-level bottlenecks.
- Our evaluation shows that multi-GPU communication latency cannot be hidden by simply increasing the computation-intensiveness of the workloads or compute capability of the GPUs. We also show that only increasing the bandwidth (BW) of the interconnect network in the multi-GPU system cannot completely eliminate the communication bottleneck. We also need an efficient implementation of DNN algorithms to take advantage of the high BW interconnect.
- We quantify the impact of growing network size and increasing batch size on memory usage and identify memory bottleneck to be a key limiting factor that hinders the speedup of the training of DNNs on multi-GPU systems.

II. Background

A. DNN

A DNN is a computation system that has multiple layers of neurons. Neurons in a layer are connected to the neighboring layers by weighted edges. Each layer applies a set of mathematical operations, such as dot-product of vectors, convolution, max-pooling or sigmoid to the layer's inputs.

A DNN can be trained to classify input data samples with high accuracy. Training a DNN is an iterative process of updating the parameters (weights) of each layer. The iterative process performs the following stages in each iteration: 1) Forward Propagation (FP), 2) Backward Propagation (BP), 3) Weight Update (WU), and 4) Metric Evaluation (ME). In FP, each layer performs a set of linear and non-linear operations on the input data. The common type of layers in a model include: the convolution layers, the fully connected layers, and the activation layers. The observed output is then compared with the expected output. The difference between the two is then fed back into the network, from the last layer back to the first layer. This is the BP stage. The output of the BP stage is the local gradients of the network parameters, suggesting how each parameter should change its value to reduce the difference between observed and expected output i.e. improve neural network classification accuracy. After a complete backward pass, the gradients are used to update the weights. This process of updating weights using gradients is based on the Stochastic Gradient Descent (SGD) algorithm. During the ME stage, performance metrics such as training accuracy are calculated. This is performed for each batch of data. Since our evaluation only focuses on performance rather than algorithm efficiency and the ME stage only adds a fixed amount of execution time, we do not include the ME stage in our study.

The FP, BP, and WU stages are repeated multiple times until the output error rate is less than a desired value. For deep neural networks with large training datasets it is expensive to update the weights after performing FP and BP for each input-output pair of the whole training set. Hence, training data is randomly sampled into mini-batches. All inputs within a mini-batch go through the FP and BP stages. The gradients are accumulated for all input-output pairs within a mini-batch and WU is performed only once.

B. Multi-GPU DNN Training

Multi-GPU systems provide faster DNN training compared to single-GPU systems. In multi-GPU systems, the training is distributed and parallelized across multiple GPUs. In this work, we do not dive deep into how the DNN algorithm works. We focus on how the data is managed and moved in a typical DNN training process. Although the exact stages of the training process differ from framework to framework, the overall approach is the same.

The timeline for training DNNs using the synchronous SGD algorithm with four GPUs is shown in Figure 1. When the algorithm starts, the CPU randomly generates the internal parameters of the network model (not shown in the figure, as this is a one time process). The network model is broadcasted to all the GPUs. The CPU also loads k mini-batches of the training data, where k equals to the number of GPUs in the system, and sends one mini-batch to each GPU (see the left-most arrows in the figure). All the GPUs perform FP and BP to

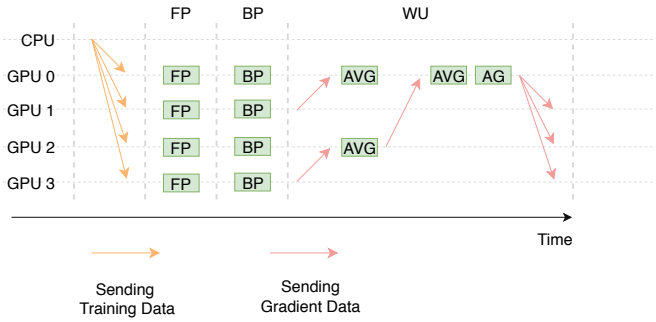

Fig. 1: The timeline of an epoch during multi-GPU DNN training using the data-parallelism approach with synchronous SGD. FP, BP, AVG, and AG represent forward propagation, backward propagation, averaging, and add gradients, respectively. (This figure is not drawn to scale.)

calculate the gradients. The size of the gradient data should be approximately equal to the size of data in the neural network model [14].

The gradients calculated by each GPU is not the same and needs to be averaged. The average is calculated with a *reduction* approach. For example, if four GPUs are used, the gradients calculated by GPU1 will be moved to GPU0 and GPU0 takes the average of the gradients from GPU0 and GPU1. Simultaneously, GPU2 collects the gradients from GPU3 and calculates the average. Finally, GPU0 collects the averaged result from GPU2 and then calculates the average. GPU0, at this time, has the averaged gradients. GPU0 updates the neural network data with averaged gradients and then, it broadcasts the updated network model to all four GPUs again. Once all the GPUs have the next mini-batch of the training set sent from the CPU, the next iteration will start. The process is repeated for a specific number of epochs.[2] Here, the number of epochs depends on the desired accuracy of training and convergence of the training algorithm.

MXNet uses a parameter server (PS) based method [9], [23] to train DNNs using multi-GPU systems. In this method, one of the GPUs, i.e. GPU0 in the aforementioned example, works as a server to store the weights and after each update of weights, all other GPUs pull the updated weights from the server. MXNet supports pipelining of WU and BP stages to overlap computation and communication to hide the communication latency. But the weights are updated using the synchronous SGD algorithm, which means GPU0 has to wait to receive the gradients from all the GPUs before updating weights.

An Asynchronous SGD (ASGD) algorithm [10], [32], [46] can also be used for training DNNs with multiple GPUs. An ASGD algorithm allows each GPU to update the weights asynchronously in the server, and continue training with the updated weights and a new mini-batch of data. This method suffers from the well-known "delayed gradient problem", which decreases the accuracy of training and affects convergence. The delayed gradient problem occurs when a GPU adds its gradients to the server, but the weight update using those gradients is delayed until the server updates the weights based on all the gradients it has received previously from other GPUs. Addressing the delayed gradient problem is still an open research problem and a number of works aim at addressing this problem [18], [38], [42], [47], [48].

[2]An epoch is a complete pass through all the data in the training dataset. Each epoch involves processing of multiple mini-batches of data.

C. Inter-GPU communication

NVIDIA provides CUDA, which is a parallel computing platform and programming model for GPUs. CUDA allows memory copy from CPU to GPU and GPU to CPU with the `cudaMemcpy` API. Alongside the support for copying data between the CPUs and the GPUs, CUDA also provides support for peer-to-peer (P2P) direct transfer and P2P direct access. For the rest of the paper, 'P2P' refers to P2P direct transfer, unless otherwise specified. In P2P direct transfer, the programmer uses the `cudaMemcpy` function which initiates a DMA copy from one GPU memory to another GPU memory. In this case, the granularity of data transfer is specified by the programmer. If P2P direct access is used, one GPU reads from or writes to another GPU's memory without performing data copy to its own memory while executing GPU instructions. In P2P direct access, the accessing granularity is usually smaller than a cache line (typically 64 bytes). Using P2P direct access simplifies the program and can also avoid stopping the kernel during memory copy operation. In the case of large data transfers, high throughput is achieved by pipelining communication using P2P direct access and hiding the latency [33].

For inter-GPU communication, NVIDIA also provides NVIDIA Collective Communications Library (NCCL) [1]. NCCL includes collectives such as ReduceKernels, BroadcastKernels, etc., that are optimized for inter-GPU communication and synchronization. Programmers can move the data easily from one GPU to another by using the collective algorithms provided by NCCL.

The two collective communication functions that are provided by NCCL are especially useful in the context of DNN training. The first function is `Broadcast`, where the DNN frameworks can send the updated model data from one GPU to all other GPUs at the end of the WU stage (see Figure 1). The second function is `AllReduce`, which copies the data from multiple GPUs to a single GPU. DNN frameworks can use it to calculate the average of the gradients at the beginning of the WU stage.

III. Related Work

There are a variety of prior studies that profile DNN workloads on a GPU and multi-GPU systems in order to gain insights into the performance bottlenecks that occur during the time-consuming training process. For instance, Dong et al. [11] extensively analyze the training process of a Convolutional Neural Network (CNN) model by profiling it on a layer-by-layer basis. Based on their thorough characterization, the authors evaluate a number of optimization approaches such as kernel fusion and cache bypassing to accelerate the training process. Zhu et al. [49] propose a new benchmark

suite for DNN training that consists of eight state-of-the-art DNN models from different types of learning– supervised, non-supervised and reinforcement learning. The DNN models are implemented in TensorFlow, MXNet and CNTK [44]. They show a variety of performance analyses and profiling statistics for the different hardware configurations, especially considering one node with a single-/multi-GPU system and also multi-node systems. However, their evaluation mainly focuses on GPU compute utilization, memory profiling, and throughput. Our main focus is to precisely account for the communication bottlenecks of the training in the new Volta-based DGX-1 system. To this end, we delve into the FP, BP and WU stages of the training process, and identify that the WU stage is a truly limiting factor as we increase the number of GPUs in the DGX-1 system. As part of our evaluation, we explore P2P direct transfer and NCCL-based communication schemes as they are commonly used in DNN frameworks.

Support for training on multiple GPUs is now standard in DNN frameworks. Shi et al. [36] explore a selection of these frameworks including Caffe [17], CNTK, MXNet, TensorFlow, and Torch, using the three most popular types of DNNs– Fully-Connected Networks, Convolutional Neural Networks, and Recurrent Neural Networks, on two CPU platforms and three GPU platforms. The authors provide guidelines for selecting appropriate combinations of the hardware platforms and software tools. Bahrampour et al. [4] present a comparative study of Caffe, Neon [25], TensorFlow, Theano, and Torch, from three different aspects, namely extensibility, hardware utilization, and performance. Kim et al. [19] analyze the GPU performance characteristics of Caffe, CNTK, TensorFlow, Theano, and Torch from the perspective of a representative CNN model. Since most of the frameworks rely on the cuBLAS and the cuDNN library, the computing performance is close among all the frameworks. In our work, we focus on data movement approaches and fully study how data movement impacts the scalability of DNN training in multi-GPU systems.

In the recent past, researchers have also started to study and reduce the communication overhead in multi-GPU systems during DNN training. Awan et al. [3] perform a study comparing the performance of the system when using MPI versus NCCL in multi-node multi-GPU DNN training. Tallent et at. [40] compare DNN training performance in PCIe-based connection vs. NVLink. Gawande et al. [13] compare a Pascal-based DGX-1 system that uses NVLink interconnects with Intel Knights Landing (KNL) CPUs interconnected with Intel Omni-Path. To the best of our knowledge, there is no study that focuses on the training of DNNs when using NCCL and P2P communication approaches in a multi-GPU system that is equipped with NVLink. Our work will study the complex communication pattern in DNN training on a multi-GPU system that is equipped with NVLink.

IV. Evaluation Methodology

To understand the behavior of DNN workloads on the Volta-based DGX-1 multi-GPU system, we choose five workloads (GoogLeNet, AlexNet, Inception-v3, ResNet and LeNet) that represent different mixtures of computation and communication. To gain insights into the factors affecting the speedup of DNN training in a multi-GPU environment, we compare two most popular inter-GPU communication methods. With the help of a profiler, we isolate the computation-intensive and the communication-intensive portion of the workloads to identify the computation and communication bottlenecks in the underlying hardware.

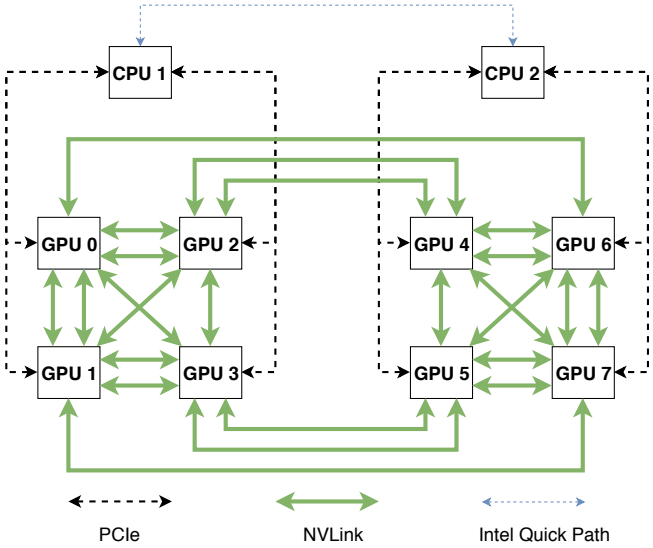

Fig. 2: Network Topology in a DGX-1 System.

A. Evaluation Platform

Our evaluations are performed on NVIDIA's Volta-based DGX-1 system [27]. The DGX-1 system has two 20-Core Intel Xeon E5-2698 v4 CPUs and eight Tesla V100 GPUs. Each Tesla V100 GPU incorporates 80 streaming multiprocessors (SMs) and delivers 17.7 TFLOPs single precision computing capability. The Tesla V100 GPU also features eight tensor cores that serve as dedicated hardware that can accelerate matrix operations. With the tensor cores, a Tesla V100 GPU can achieve a 125 TFLOPs, which is 7× faster than only using the traditional single precision computing devices. Since we focus on the DNN workloads where matrix multiplication is the most common operation, the tensor cores are utilized to accelerate the training of the DNNs.

Figure 2 shows a high-level view of the network topology of the Volta-based DGX-1 system. As depicted in Figure 2, the CPUs in the DGX-1 system use the PCIe bus to communicate with the GPUs, while the GPUs are connected with high-bandwidth peer-to-peer NVLink connections. Each NVLink connection delivers 25 GB/s data transfers in each direction. For GPU pairs that have more than one connection, NVLink can aggregate the connections and provide a 50 GB/s virtual connection. Each CPU has direct access to only four GPUs. To access the other four GPUs it needs the help of the other CPU. Some GPUs have two direct connections between them (e.g. between GPU 0 and GPU 2), while some GPUs have only one direct connection between them (e.g. between GPU

2 and GPU 3). Moreover, some GPUs may not have a direct connection between them (e.g. between GPU 3 and GPU 4), and they require the help of another GPU or two CPUs for communication. A maximum of one intermediate node (two hops) is required to connect any pair of GPUs.

B. Framework and Tools

For our evaluation, we use the NVIDIA container image of MXNet, release 18.04 and CUDA 9.0.176. The DNN frameworks run on the cuDNN 7.1.1 [28] and cuBLAS 9.0.333 [30]. The MXNet framework uses `Broadcast` and `AllReduce` communication collectives from NCCL 2.1.15. We collect the profiling data using the `nvprof` [29] profiler. We use the NVIDIA System Management Interface `nvidia-smi` to monitor memory usage of GPUs.

As discussed in Section II-A, we only execute and profile the training of DNNs for FP, BP, and WU stages. As FP, BP, and WU stages are repeatedly executed, the accuracy of the network will improve. However, the time spent during each of the three stages within an epoch will remain the same.

C. Workloads and Datasets

For our evaluation, we use five popular neural networks used for image classification. Table I specifies the number of layers and parameters in each neural network. Layers in LeNet and AlexNet are connected serially and consist of 3×3 to 11×11 kernels in the Convolution layers for feature extraction. LeNet and AlexNet have a higher number of parameters because of their relatively larger number of fully connected layers compared to other neural networks in our evaluation.

GoogLeNet and Inception-v3 networks have both Convolution layers and Inception layers for feature extraction. Typical Inception layers consist of small parallel convolution kernels (1×1 to 5×5) followed by concatenation layer to concatenate features extracted from the parallel convolution kernels. Inception layers allow the network to use both local features (small convolution kernels) as well as highly abstracted features (larger convolution kernels). GoogLeNet and Inception-v3 require a smaller number of parameters compared to AlexNet because of the inception layers.

ResNets are very deep neural networks with residual blocks. A residual block is created by combining the output of the current layer and the output(s) from one or more previous layers using a shortcut connection (i.e. a direct connection skipping any layers between the current layer and the previous layer(s).) Residual block allows training very deep networks without degrading the initially extracted features. It also requires a smaller number of parameters compared to other neural networks.

We consider both strong scaling and weak scaling in our evaluation. Strong scaling of the training of DNNs is the speedup in training time as we increase the number of GPUs while keeping the size of the input dataset fixed. Weak scaling is the speedup in training as we increase the number of GPUs as well as the size of the input dataset by a factor equal to the GPU count. We use 256K images from Imagenet dataset to train our networks for evaluating the strong scaling. The input image dimension is $299 \times 299 \times 3$ for Inception-v3 and ResNet, and $224 \times 224 \times 3$ for the other networks. For evaluating weak scaling, we use 256K, 512K, 1024K and 2048K images for 1, 2, 4 and 8 GPUs, respectively.

TABLE I: Description of the networks. (Conv = Convolution, Incep = Inception, and FC = Fully Connected)

Network	Layers	Conv Layers	Incep Layers	FC Layers	Weights
LeNet	5	2	0	2	60K
AlexNet	8	5	0	3	60M
GoogLeNet	22	3	9	1	4M
Inception-v3	48	7	11	1	24M
ResNet	110	107[1]	0	1	55M

[1] Conv layers with residual input from previous layers

V. Evaluation

In this section, we present our evaluation and analysis of the training of 5 DNN workloads using NVIDIA's Volta-based DGX-1 multi-GPU system. Here, we discuss the comparison of the P2P and the NCCL communication methods, quantify the NCCL overhead, show the breakdown of training time into FP+BP and WU stages, determine memory usage, and provide analysis on weak scaling for training the DNNs. We also provide in-depth insights and guidance for future endeavors for accelerating the training of DNN workloads using multi-GPU systems.

A. Comparison of P2P and NCCL

In this subsection, we compare the training time for the DNN workloads with the batch sizes of 16, 32 and 64 using the P2P and the NCCL communication methods and identify the factors that affect the training of the DNNs on the DGX-1 multi-GPU system.

Figure 3 shows the total training time for 5 different workloads using 1, 2, 4 and 8 GPUs. To analyze the results, first we start with the smallest network (LeNet) and discuss the impact of increasing the number of GPUs for a given batch size on LeNet using the P2P communication method. Then we discuss the impact of increasing batch size for training LeNet with P2P for a given GPU count. Afterwards, we discuss the effect of increasing both the batch size and the number of GPUs on the 5 workloads. Then, we compare the impact of the two communication methods for all the workloads, different batch sizes and different number of GPUs. Finally, we provide insights obtained from our analysis.

For training LeNet with a batch size of 16, as we increase the number of GPUs, the overall training time decreases for both P2P and NCCL. With P2P we can speed up the training time by factors of 1.62×, 2.37× and 3.36× for 2, 4 and 8 GPUs, respectively. On the other hand, with NCCL we achieve speedup factors of 1.56×, 2.27× and 2.77× for 2, 4 and 8 GPUs, respectively. This is understandable because LeNet is a very small network with only 2 convolution layers. GPUs do not have sufficient computation for this workload to hide the latency of communication. As a result, communication time

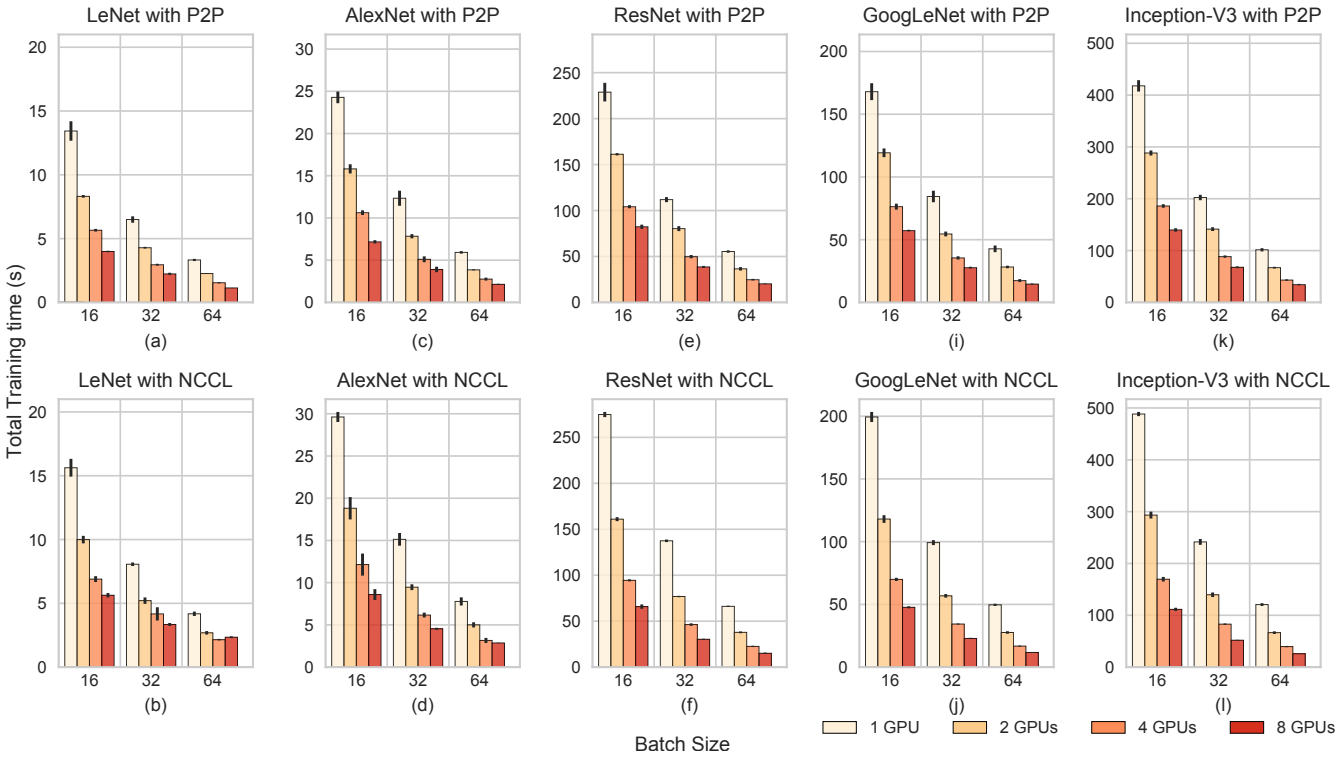

Fig. 3: Training time per epoch for 5 different workloads on the Volta-based DGX-1 system using the P2P and the NCCL-based communication. Each bar represents the mean training time of 5 repetitions. The standard deviation is shown by the black line on top of each bar.

dominates the training time. Hence, the training time does not decrease linearly for this workload as we increase the number of GPUs. P2P outperforms NCCL for this workload due to the overhead associated with incorporating NCCL into MXNet. In Section V-B we quantify this overhead.

For all GPU counts, we observe that the time spent in training LeNet decreases almost linearly as we increase the batch size from 16 to 32 to 64. For instance, for training LeNet using 4 GPUs with P2P, as we increase the batch size from 16 to first 32 and then 64, the training time decreases by a factor of 1.92× and 3.67×, respectively. With the increased batch size, two factors affect computation. The first factor is the total number of batches that the GPU needs to process decreases for a fixed dataset. The second factor is that the number of images each GPU needs to process increases. While a decrease in the number of batches helps reduce computation time, an increase in the number of images per batch leads to increased utilization of GPU compute cores, provided that the cores are not already saturated. If the compute cores become saturated, increasing the batch size may lead to a computation bottleneck. Moreover, as the number of batches decreases with a larger batch size, the frequency of inter-GPU communication decreases. However, the amount of data that needs to be transferred for each WU from one GPU to another remains constant. This is because the number of gradients and weights is independent of the batch size, and depends solely on the DNN. Hence, increasing the batch size helps decrease the communication time.

Using the P2P communication method, the training time for all workloads under study decreases as the batch size increases. As we increase the number of GPUs from 1 to 2, for all the workloads, we observe up to a 1.8× speedup in the training time. However, we do not get the same rate of speedup when using 4 and 8 GPUs. This is because P2P memcopy becomes increasingly communication heavy as we increase the number of GPUs. For instance, consider the 4 GPU case. In the MXNet implementation using P2P, the gradients are aggregated on GPU0. Hence, the other 3 GPUs transfer their locally computed gradients to GPU0 using a reduction operation, as discussed in Section IIB. Then, GPU0 updates the weights using the gradients and transfers the updated weights to all the GPUs. The transfer of the gradients and the weights is parallelized using an asynchronous data transfer between GPUs. Note that the DGX-1 system provides asymmetric interconnects between different pairs of GPUs. This can cause some of the GPUs to become idle during DNN training. The BW for communication between GPU0 and GPU1, and GPU0 and GPU2, is twice the BW rate between GPU0 and GPU3 (see Figure 2). As a result, after updating weights, GPU3 has to wait longer than GPU1 and GPU2 to receive the updated weights. This causes GPU1 and GPU2 to remain idle until GPU3 receives the updated weights.[3]

[3]It is possible that GPU1 and GPU2 execute FP and BP immediately after receiving the updated weights. In that case, before synchronizing the gradients, GPU1 and GPU2 have to remain idle after executing FP and BP, until GPU3 finishes executing FP and BP.

For the training with 8 GPUs, the situation is even worse because of the lack of direct connectivity using NVLink between all the GPUs. For instance, GPU0 has direct NVLink connections with GPU1, GPU2, GPU3, and GPU6. When weights are transferred from GPU0 to GPU4, GPU5, and GPU7, direct P2P memory transfers cannot be used. Instead, weights are transferred using a device-to-host (DtoH) memory copy followed by a host-to-device (HtoD) memory copy over the slow PCIe interconnect.[4] Hence, the communication time can be significantly longer for training a DNN using 8 GPUs if the number of weights is large. MXNet tries to overcome this issue by performing multi-stage transfers through NVLink. More precisely, since GPU1 has a direct NVLink connection with GPU7, GPU0 first transfers the weights to GPU1 and then GPU1 transfers the weights to GPU7. This multi-stage transfer requires some additional time, which results in a non-linear speedup as we increase the GPU count from 2 to 4 to 8.

For training LeNet with a batch size of 16, P2P achieves a better speedup of training time than NCCL as we increase the GPU count. P2P achieves better training time than NCCL even if we increase the batch size for training LeNet. This applies for training AlexNet as well, because AlexNet has only 5 convolution layers and a large number of weights ($\sim$60M). This implies that if the number of computation-intensive layers is small, the overhead associated with incorporating the NCCL library cannot be amortized and P2P will outperform NCCL. Note that DNNs with a small number of computationally-intensive layers (i.e., Lenet, AlexNet) achieve non-linear speedup as we increase the GPU count to 4 and 8. This is because the amount of computation is not sufficient to hide the latency of communication, as well as synchronization, among the GPUs. However, for workloads with a larger number of computation-intensive layers, NCCL continuously outperforms P2P, as we increase the batch size for 4 and 8 GPUs. For a batch size of 16, training of GoogleNet is 1.1$\times$ and 1.2$\times$ faster when using NCCL as compared to P2P for 4 and 8 GPUs, respectively. For both ResNet and Inception-v3, the training with a batch size of 16 is 1.1$\times$ and 1.25$\times$ faster using NCCL than using P2P with 4 and 8 GPUs, respectively. This is because NCCL pipelines the data transfer for updating and transferring the weights using `AllReduce` and `Broadcast` operations, respectively. This implies that the process of pipelining data transfers can amortize the NCCL overhead if there are sufficient number of data transfers [5].

Our evaluation based on training time provides the following insights:

- Increasing batch size reduces training time for an epoch linearly for all the workloads we evaluated in this work. Hence, to accelerate DNN training hardware support is needed to facilitate training with larger batch sizes.
- Whether increasing the number of GPUs to train a particular network will lead to faster training depends on the computation-intensity of the workload and the communication method.
- With the increase of computation-intensive layers in DNN workloads, although the overall training time increases, we can reduce training time by increasing the number of GPUs.
- With NCCL, training time decreases significantly for 4 and 8 GPUs if the DNN workload has a sufficiently large number of computation-intensive layers.
- NCCL implementation has additional overhead compared to P2P implementation. NCCL should be used for training with more than 4 GPUs if the DNN network model is large, otherwise, P2P is sufficient.

[4]This is a limitation of the design of DGX-1. The "routers" within each GPU do not have the ability to route a packet to another node, and thus, the communication for all non-1-hop packets go through the CPU. This not a fundamental limitation, but instead, a design decision.

[5]More layers with weights mean more number of data transfers.

B. NCCL Overhead

In the previous section, we explained that using NCCL for communication comes with additional overhead. When NCCL is used as the communication method, MXNet uses different source code (i.e. variables, functions, and kernels) compared to P2P even if only one GPU is used. MXNet uses `AllReduce` and `Broadcast` collectives available in NCCL for aggregating gradients and transferring updated weights to the GPUs, respectively. In particular, two kernels (`ReduceKernel` and `BroadcastKernel`) are used when NCCL is used as the communication method. These kernels leverage the P2P direct memory access where one GPU can directly use the data on another GPUs memory without transferring the data to its own memory. Note that the P2P communication method that we compare with NCCL uses P2P direct data transfers which is different from P2P direct memory access. Since the kernels executed by the GPUs are different for NCCL and P2P method, the CUDA runtime API overhead varies as data is accessed using different methods. For DNN training, the overhead for NCCL is larger than the overhead for P2P. However, NCCL overcomes this overhead by pipelining the data transfer as we increase the GPU count. In this section, we measure that additional NCCL overhead by comparing the training times on a single GPU for P2P and NCCL. This result explains why the use of NCCL cannot help reduce the training time for all types of workloads.

Table II shows the NCCL overhead over P2P for training the 5 DNNs with different batch sizes on a single GPU. The percent overhead varies by a value of less than 3.6 for large networks (i.e. GoogLeNet, Inception-v3 and ResNet) with the increase of batch size while the percentage of overhead increases with batch size for smaller networks (LeNet and AlexNet). This is because as the batch size increases, the overall computation time reduces significantly for these smaller workloads and the NCCL overhead becomes more significant. This additional overhead is also present in the multi-GPU training. In the MXNet implementation, the P2P memcopy method requires hundreds of GBs of data copy per epoch from 3 GPUs (for training with 4 GPUs) or 7 GPUs (for training with 8 GPUs) to one of the GPUs' (GPU0) memory, whereas using NCCL, one GPU simply reads another GPU's memory and directly uses

TABLE II: NCCL overhead compared to P2P for the workloads executed on a single GPU.

Network	Batch Size	NCCL Overhead (%)
LeNet	16	16.4
LeNet	32	24
LeNet	64	26.7
AlexNet	16	21.8
AlexNet	32	21.8
AlexNet	64	31.8
ResNet	16	20.1
ResNet	32	22.9
ResNet	64	19.3
GoogLeNet	16	18.7
GoogLeNet	32	17.5
GoogLeNet	64	16.2
Inception-v3	16	16.9
Inception-v3	32	19.4
Inception-v3	64	18.9

the data for computation. As NCCL reduces the communication time by leveraging the pipelining in data transfer, training a network using 2 GPUs cannot benefit much from the pipelining, rather it suffers from additional NCCL overhead. However, if a network is large, NCCL benefits significantly from the pipelining and overcomes the NCCL overhead when training the network with 4 and 8 GPUs. Hence, with 4 and 8 GPUs, we observe a better speedup in the training time of ResNet, GoogleNet and, Inception-v3 using NCCL compared to training time using P2P.

C. Training Time Breakdown

In this section, we show the breakdown of the total training time into computation (FP+BP) and communication (WU) time. The dataset contains a fixed set of 256K images from the Imagenet dataset for this experiment. Figure 4 shows the breakdown of the total training time for the 5 workloads for different batch sizes and GPU counts when using NCCL-based communication. Since our comparison between P2P and NCCL shows that NCCL has the potential to significantly decrease the training time and achieve larger speedup compared to P2P as we increase the network size and GPU count, in this subsection, we only consider NCCL-based communication. We use the `nvprof` profiler for this analysis.

During the FP stage of DNN training, the outputs (i.e. feature maps) of different layers are generated for an input batch of data (i.e. images). During the BP stage, the error at the final output layer is calculated and back-propagated to compute the gradients. Hence, FP and BP are the compute-intensive portions of DNN training. During the WU stage, the gradients are aggregated and synchronized using `AllReduce` operations from the NCCL library and the updated weights using the aggregated gradients are broadcasted to all GPUs using `Broadcast` operations from the NCCL library. Hence, the amount of computation is negligible in the WU stage. So, we make the assumption that the time spent in the WU stage is primarily for communication.[6] Note that for the single GPU case the WU is nearly two orders of magnitude lower than the FP and BP stage [35] because updating weights is simple matrix addition operation (i.e. $Y = aX + B$, where a is scalar and Y and B are vectors) and does not involve any inter-GPU communication. Hence, in our evaluation, we do not report the time spent in the WU stage for single GPU training. To analyze the results, first, we discuss the impact of increasing the number of GPUs on the FP+BP and WU stages for training LeNet with a given batch size. Then, we discuss the effect of increasing the batch size on the FP+BP and WU stages for training LeNet for a given GPU count. We discuss the effect of both batch size and GPU count across all the workloads. Finally, we present the insights obtained from the breakdown of training time into FP+BP and WU stages,

TABLE III: `cudaStreamSynchronize` API overhead for training LeNet with a batch size of 16, 32 and 64 using 1, 2, 4 and 8 GPUs.

Batch Size	GPU Count	Time (%)
16	1	89.2
	2	94.1
	4	86.7
	8	76.4
32	1	86.7
	2	91.9
	4	78.6
	8	68.8
64	1	81.6
	2	86.1
	4	69.8
	8	54.4

For the training of LeNet with a batch size of 16, we observe more than two-fold improvement for FP+BP time as the number of GPUs increases from 1 to 2. This is because the training with 1 GPU suffers from 21.8% additional NCCL overhead that we have shown in Section V-B. However, as the number of GPUs further increases, the time required for FP+BP decreases non-linearly. Our profiling results show that the `cudaStreamSynchronize` API consumes most of the time among all APIs.[7] Table III shows the percent overhead of `cudaStreamSynchronize` for training LeNet with a batch size of 16 using 1, 2, 4 and GPUs. LeNet with a compute utilization of only 18.3% fails to amortize this CUDA API overhead, and so we observe a non-linear scaling of time spent

[6]Since MXNet allows overlap of BP and WU, some of the communication latency can be hidden. The WU stage takes into account the hidden latency. Hence, the actual communication time is larger than the time required for the WU stage.

[7]While training DNNs using GPUs, the training process is conducted with the help of multiple CUDA streams. Each stream is responsible for a unique set of tasks (i.e., one CUDA stream performs FP, while another performs BP). All the tasks assigned to a particular stream execute sequentially, but the different streams can be executed in parallel. The `cudaStreamSynchronize` API is used to maintain synchronization of the streams with the CPU or host thread. It holds off execution in the CPU or host thread until all the CUDA tasks assigned to the stream referenced by `cudaStreamSynchronize` finish execution. This overhead can be amortized by assigning more tasks to the stream before synchronization.

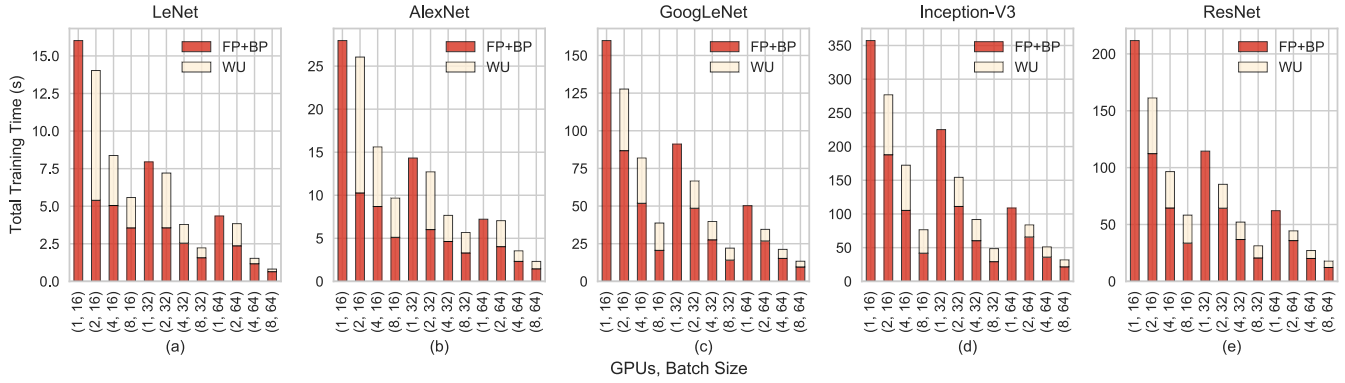

Fig. 4: Breakdown of training time into computation (FP stage and BP stage) time and communication (WU stage) time. The X-axis represents (GPU count, Batch Size).

in the FP+BP stages. The time spent in the WU stage decreases almost linearly as we increase the number of GPUs from 2 to 4 to 8, for a batch size of 16.

As we increase the batch size for training LeNet, both the time for FP+BP stage and time for WU decreases linearly. This is expected because doubling the batch size halves not only the number of batches each GPU processes, but also the number of times each GPU needs to communicate with other GPUs for a fixed dataset. Since LeNet is neither a computation-intensive (only 2 convolution layers) nor a communication-intensive (only ∼60k parameters) workload, batch sizes of 32 and 64 cannot saturate the compute cores or the NVLink BW. Nonetheless, Table II shows that as we increase the batch size, percentage of time spent for `cudaStreamSynchronize` decreases. This is because with the increased batch size, each CUDA stream performs more computations or tasks while the number of times synchronization of streams is required reduces.

As the number of computation-intensive layers in the workload increases, we observe that time spent in FP+BP stage reduces almost linearly for all GPU count (for Inception-v3, we achieve near ideal linear scaling for the batch sizes of 16 and 32). However, the time spent in the WU stage achieves ideal linear scaling only for AlexNet which has ∼60M weights and only 8 layers. As we increase the GPU count from 2 to 4 to 8, although for other workloads we do not obtain linear scaling, we observe that from ResNet→GoogLeNet→Inception-v3, the WU stage achieves better speedup. From our observation, it is evident that workloads with more weights per layer (for layers that contain weights) show better speedup in the WU stage. This is because transferring a small amount of data is a waste of NVLink BW if the layers have a small number of weights.

Based on our evaluation using `nvprof`, in this section we provide the following insights:

- Computation time for FP+BP dominates the training time as we increase the number of GPUs for the workloads under study.
- In order to achieve close to ideal linear scaling, FP+BP stages need to utilize GPU compute cores efficiently (i.e. we can increase GPU compute utilization by increasing the amount of work in each GPU by increasing the batch size and correspondingly, reduce the number of data transfers).

D. Memory Usage Analysis

During our evaluation, we observe that the memory capacity of GPUs limits the maximum batch size that can be used to train a DNN. Hence, in this section, we perform an in-depth evaluation of memory usage by GPUs prior to the start of the training (pre-training) and during training. During the pre-training stage, the network model is transferred to the GPUs from the CPU and during the training stage additional GPU memory space is required to house the feature maps and temporary results. We varied the batch size to see the impact of batch size on both the pre-training and the training stage memory usage for 5 DNN workloads. During training, since one of the GPUs (typically GPU0) is responsible for coordinating the other GPUs, it requires additional memory.

Table IV shows the memory usage for 4 GPUs during the pre-training and training phase. Note that all the GPUs require the same amount of memory during the pre-training phase. Additionally, there is a less than 5% difference in the memory usage of P2P memcopy and NCCL based communication methods. Hence we are only reporting memory usage for NCCL-based communication method. We observe that during the training of DNNs, all the GPUs except GPU0 consume the same memory irrespective of the number of GPUs used for training. Furthermore, for training using 2, 4 or 8 GPUs, GPU0 consumes almost the same amount of memory. Hence, the 4 GPU memory results are representative of the memory usage for training with 2, 4 and 8 GPUs.

As we increase the batch size, the memory usage increases for all workloads. While the increase in the pre-training memory usage is insignificant, the memory usage increases significantly during training. For instance, increasing the batch size from 16 to 64 increases the GPU memory consumption by a factor of 1.83× for Inception-v3. This is because with an increased batch size, GPUs produce more feature maps or intermediate results. For all the workloads, GPU0 uses more memory than the other GPUs used in training. This is because MXNet uses GPU0's memory for gradient aggregation and weight update. As we increase batch size, the increase in the

TABLE IV: Memory usage when using the NCCL-based communication method during the pre-training stage and the training stage of DNNs when using 4 GPUs. The memory usage of all GPUs is the same for the pre-training stage. GPUz refers to the memory usage of a GPU during the pre-training, where z can take any value from 0 to 3. GPU0 refers to the memory usage of the GPU0 during training while GPUx refers to memory usage of the remaining GPUs, where x can take any value from 1 to 3.

Network	Batch Size	Pre-training GPUz (GB)	Training GPU0 (GB)	Training GPUx (GB)	Additional Mem. Usage in GPU0 w.r.t. GPUx (%)	Increase in Mem. Usage w.r.t. the Batch Size of 16 (%)
LeNet	16	1.37	2.76	1.96	41.1	–
LeNet	32	1.38	2.84	2.04	39.4	3.0
LeNet	64	1.40	2.89	2.36	22.7	4.8
AlexNet	16	1.24	2.15	1.55	39.2	–
AlexNet	32	1.25	2.36	1.76	34.5	9.9
AlexNet	64	1.27	2.97	2.37	25.6	38.2
ResNet	16	1.08	3.62	3.29	10.1	–
ResNet	32	5.98	5.66	5.63	6.2	56.1
ResNet	64	11.06	9.48	9.15	3.5	161.5
GoogLeNet	16	0.92	2.35	2.24	4.7	–
GoogLeNet	32	0.94	3.64	3.55	2.5	55.2
GoogLeNet	64	0.97	6.17	6.07	1.6	162.8
Inception-v3	16	1.04	3.89	3.60	7.9	–
Inception-v3	32	1.06	6.70	6.06	10.5	72.3
Inception-v3	64	1.09	11.01	10.78	2.4	183.3

memory required for feature maps is significantly large. But the additional memory required for gradients does not increase proportionately. Hence, the percentage of additional memory usage by GPU0 decreases with increased batch size.

As the number of feature maps increases with the increase in the number of layers, the memory usage increases significantly. Note that the increase in feature maps do not necessarily depend on the number of layers, rather it depends on total nodes (neurons) in different layers. For a batch size of 64, GPU0 requires a memory usage of 2.37GB to train AlexNet while GPU0 requires 11GB of memory to train Inception-v3. Memory required for feature maps can only be reduced by making algorithm-level changes.

During our evaluation, we also tried to evaluate batch sizes larger than 64 per GPU for all the workloads. However, we could not train Inception-v3 and ResNet with a batch size larger than 64 per GPU, and we could not train GoogleNet with a batch size larger than 128 per GPU, due to GPU memory limitations. Hence, future research should focus on both increasing memory capacity while preserving the memory BW from the hardware-level, as well as more efficient memory mapping from the software-level.

Our evaluation in this subsection provides the following insights:

- While increasing the batch size reduces the training time of DNNs for each epoch, the amount of GPU memory limits the maximum batch size that can be used for training DNN workloads.
- For larger workloads (i.e. ResNet, GoogleNet, and Inception-v3), the memory required for intermediate outputs at different layers far exceeds the memory required for the network model.

E. Weak Scaling

We evaluated the weak scaling trends of the training time of the 5 DNN workloads by increasing the number of images in the dataset as we increase the number of GPUs. We use 256k, 512k, 1024k, and 2048k images for 1, 2, 4, and 8 GPUs, respectively. Figure 5 shows the average time for training with 256K images for both P2P and NCCL using 1, 2, 4 and 8 GPUs, as well as the total training time for evaluating weak scaling. Based on our evaluation of weak scaling we provide the following insights:

- The speedup for training LeNet with weak scaling is larger than that with strong scaling for all the batch sizes with both P2P and NCCL. As discussed in Section V-C, the overhead associated with CUDA APIs affects training time of LeNet. As the dataset size is increased for weak scaling, the overhead associated with creating and synchronizing CUDA streams gets amortized, which leads to a slightly improved training time over strong scaling training time for LeNet.
- When using weak scaling, AlexNet shows better training time for the batch sizes of 32 and 64 compared to strong scaling. With a small number of computation-intensive layers, AlexNet suffers from the overhead of CUDA APIs for creating and synchronizing streams. As we increase the number of batches, it amortizes some of the overheads. Since AlexNet has a large number of weights per layer, it utilizes the high BW of NVLink more efficiently than LeNet.
- In case of the relatively more computation-intensive workloads i.e. ResNet, GoogLeNet, and Inception-v3, when using weak scaling we achieve speedups that are less than 17% higher as compared to speedups with strong scaling using NCCL for all the batch sizes.

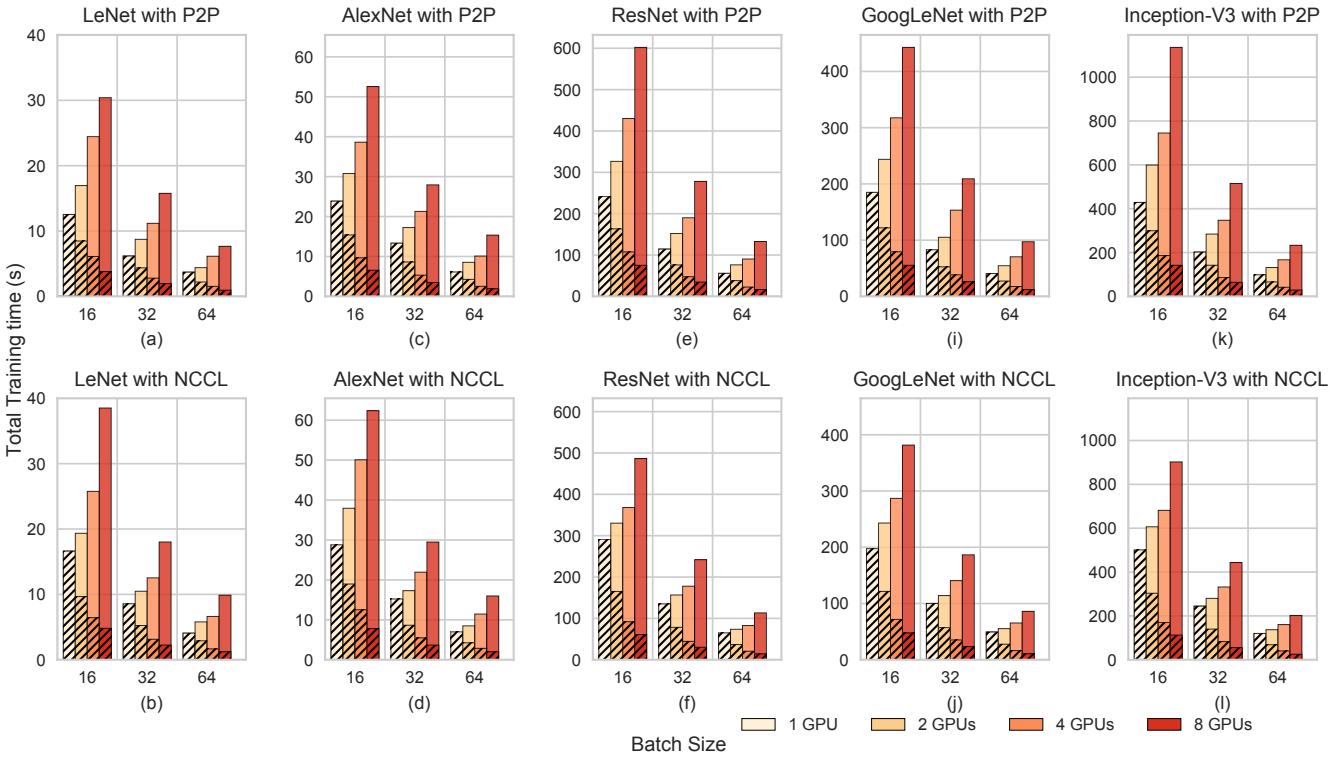

Fig. 5: Weak scaling evaluation for the 5 workloads. The height of the 'entire bar' represents the total time per epoch for training with 256k, 512k, 1024k and, 2048k images using 1, 2, 4 and, 8 GPUs, respectively. The height of the 'hatched bar' represents the average time to train with 256k images. This facilitates the comparison between the training time for weak scaling with that for strong scaling.

F. Accelerating Training of DNNs

Based on our evaluation, we make the following suggestions to accelerate DNN training:

- In Section V-A and Section V-B, we have shown that NCCL does not always perform better than P2P because of additional overhead associated with NCCL. This overhead needs to be reduced. Apart from that, frameworks such as MXNet should be improved to leverage the best available communication method automatically for a given DNN workload.
- Increasing the GPU count does not improve the training time for smaller workloads as shown in Section V-A. Hence, the size of the workload (i.e. number of compute-intensive layers, number of weights, etc.) should be taken into account to choose the proper GPU count.
- Our evaluation in Section V-C shows that for FP+BP stages do not achieve ideal linear speedup as we increase the batch size for a number of computation-intensive workloads. GPUs with more tensor cores and compute cores can help accelerate the FP+BP stage.
- Our memory analysis in Section V-D showed that GPU memory capacity can be a severe bottleneck for training DNNs as larger networks need to be trained with larger batch sizes to reduce training time. The maximum batch size that can be used to train a network is limited by GPU memory capacity for data parallel implementation of training. Hence, significant improvement in the memory technology is required to increase GPU memory capacity.[8]
- Our evaluation show that inefficiency in the implementation of high-level frameworks such as MXNet, may lead to under-utilization of GPU resources. For instance, additional memory consumption of GPU0 compared to other GPUs causes under-utilization of available GPU memory. This can be solved by a more efficient distribution of data.
- CUDA API overheads for maintaining synchronization consume a significant amount of training time. Faster synchronization mechanism needs to be developed to utilize the GPU resources more efficiently.

VI. Conclusion

In this work, we performed a comprehensive analysis to understand the computation and communication pattern of training DNN workloads on a multi-GPU system. We used Volta-based DGX-1 multi-GPU system and characterized data movement and memory usage of five DNN workloads (GoogLeNet, AlexNet, Inception-v3, ResNet and LeNet). We used the MXNet framework for training DNN workloads using data parallelism approach. We compared NCCL library based com-

[8]The memory required for training a DNN using GPUs can depend on how a framework is implemented. Hence, different frameworks may need a different amount of memory for training the same DNN. But the memory required for the output at each layer must be the same for all frameworks, for a given DNN.

munication with P2P memcopy based communication among GPUs.

Based on our evaluation, we found that the multi-GPU scalability heavily depends on the neural network architecture, batch size, and the GPU-to-GPU communication method. We conclude that workloads scale better with NCCL based communication than P2P for 4 and 8 GPUs. NCCL introduces significant overhead to DNN training, especially when using 1 or 2 GPUs for training. Finally, we provide suggestions to accelerate DNN training. As future work, we will use the results from this paper to study the distribution of data to better utilize the memory across all the GPUs and to customize the communication between GPUs to use the asymmetric NVLink topology more efficiently.

ACKNOWLEDGEMENT

We sincerely thank the paper shepherd, Dr. Sherief Reda and the anonymous reviewers for their useful feedback. This work was supported in part by NSF CNS-1525474 and MINECO TIN2016-78799-P.

REFERENCES

[1] Nvidia collective communications library (nccl), May 2018.
[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
[3] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda. Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: Mpi or nccl? *arXiv preprint arXiv:1707.09414*, 2017.
[4] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*, 2015.
[5] L. Brown. Deep learning with gpus. *Larry Brown Ph. D., Johns Hopkins University*, 2015.
[6] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.
[7] X.-W. Chen and X. Lin. Big data deep learning: challenges and perspectives. *IEEE access*, 2:514–525, 2014.
[8] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.
[9] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.
[10] 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*, pages 1223–1231, 2012.
[11] S. Dong, X. Gong, Y. Sun, T. Baruah, and D. Kaeli. Characterizing the microarchitectural implications of a convolutional neural network (cnn) execution on gpus. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 96–106, New York, NY, USA, 2018. ACM.
[12] J. Fox, Y. Zou, and J. Qiu. Software frameworks for deep learning at scale. *Internal Indiana University Technical Report*, 2016.
[13] N. A. Gawande, J. B. Landwehr, J. A. Daily, N. R. Tallent, A. Vishnu, and D. J. Kerbyson. Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 399–408, 2017.
[14] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.
[15] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.
[16] J. Hagerty, R. J. Stanley, and W. V. Stoecker. Medical image processing in the age of deep learning. 2017.
[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.
[18] J. Keuper and F.-J. Pfreundt. Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, page 1. ACM, 2015.
[19] H. Kim, H. Nam, W. Jung, and J. Lee. Performance analysis of cnn frameworks for gpus. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pages 55–64. IEEE, 2017.
[20] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.
[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
[22] D. Li, X. Chen, M. Becchi, and Z. Zong. Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus. In *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on*, pages 477–484. IEEE, 2016.
[23] 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*, volume 14, pages 583–598, 2014.
[24] Y. Miao, H. Zhang, and F. Metze. Distributed learning of multilingual dnn feature extractors using gpus. 2014.
[25] I. NervanaSystems. The neon deep learning framework, 2017.
[26] H. Noh, S. Hong, and B. Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.
[27] NVidia. Nvidia dgx-1 with tesla v100 system architecture.
[28] NVidia. Nvidia cudnn, 2018.
[29] NVidia. Profiler user's guide, 2018.
[30] C. Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.
[31] K.-S. Oh and K. Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.
[32] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *arXiv preprint arXiv:1312.6186*, 2013.
[33] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.
[34] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
[35] S. Shi and X. Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. *arXiv preprint arXiv:1711.05979*, 2017.
[36] S. Shi, Q. Wang, P. Xu, and X. Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104, Nov 2016.
[37] S. L. Smith, P.-J. Kindermans, and Q. V. Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.
[38] A. Srinivasan, A. Jain, and P. Barekatain. An analysis of the delayed gradients problem in asynchronous sgd. 2018.
[39] S. Stabinger, A. Rodríguez-Sánchez, and J. Piater. 25 years of cnns: Can we compare to human abstraction capabilities? In *International Conference on Artificial Neural Networks*, pages 380–387. Springer, 2016.
[40] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie. Evaluating on-node gpu interconnects for deep learning workloads. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 3–21. Springer, 2017.
[41] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato. Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853*, 2013.
[42] Q. Yao, X. Liao, and H. Jin. Training deep neural network on multiple gpus with a model averaging method. *Peer-to-Peer Networking and Applications*, 11(5):1012–1021, 2018.
[43] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. 100-epoch imagenet training with alexnet in 24 minutes. *ArXiv e-prints*, 2017.
[44] D. Yu, A. Eversole, M. Seltzer, K. Yao, Z. Huang, B. Guenter, O. Kuchaiev, Y. Zhang, F. Seide, H. Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*, 2014.
[45] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*, 2015.
[46] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6660–6663. IEEE, 2013.
[47] S.-Y. Zhao and W.-J. Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *AAAI*, pages 2379–2385, 2016.
[48] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. Asynchronous stochastic gradient descent with delay compensation. *arXiv preprint arXiv:1609.08326*, 2016.
[49] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Tbd: Benchmarking and analyzing deep neural network training. *arXiv preprint arXiv:1803.06905*, 2018.