2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-9423-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/HiPC56025.2022.00017

AccDP: Accelerated Data-Parallel Distributed DNN Training for Modern GPU-Based HPC Clusters

Nawras Alnaasan, Arpan Jain, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda
Department of Computer Science and Engineering,
The Ohio State University, Columbus, Ohio, USA,
{alnaasan.1, jain.575, shafi.16, subramoni.1}@osu.edu, panda@cse.ohio-state.edu

***Abstract*—Deep Learning (DL) has become a prominent machine learning technique due to the availability of efficient computational resources in the form of Graphics Processing Units (GPUs), large-scale datasets and a variety of models. The newer generation of GPUs are being designed with special emphasis on optimizing performance for DL applications. Also, the availability of easy-to-use DL frameworks—like PyTorch and TensorFlow—has enhanced productivity of domain experts to work on their custom DL applications from diverse domains. However, existing Deep Neural Network (DNN) training approaches may not fully utilize the newly emerging powerful GPUs like the NVIDIA A100—this is the primary issue that we address in this paper. Our motivating analyses show that the GPU utilization on NVIDIA A100 can be as low as 43% using traditional DNN training approaches for small-to-medium DL models and input data size. This paper proposes AccDP—a data-parallel distributed DNN training approach—to accelerate GPU-based DL applications. AccDP exploits the Message Passing Interface (MPI) communication library coupled with the NVIDIA's Multi-Process Service (MPS) to increase the amount of work assigned to parallel GPUs resulting in higher utilization of compute resources. We evaluate our proposed design on different small-to-medium DL models and input sizes on the state-of-the-art HPC clusters. By injecting more parallelism into DNN training using our approach, the evaluation shows up to 58% improvement in training performance on a single GPU and up to 62% on 16 GPUs compared to regular DNN training. Furthermore, we conduct an in-depth characterization to determine the impact of several DNN training factors and best practices—including the batch size and the number of data loading workers— to optimally utilize GPU devices. To the best of our knowledge, this is the first work that explores the use of MPS and MPI to maximize the utilization of GPUs in distributed DNN training.**

***Index Terms*—Deep Neural Networks, Graphics Processing Units, Multi-Process Service, MVAPICH2**

I. Introduction

Today, Deep Learning (DL) is fueling many of the advances in various application areas—including image processing, voice/speech recognition, recommender systems, and natural language processing—and has become the driving engine in pushing the frontiers of Artificial Intelligence (AI). This has been possible due to continuous improvements and increasing efficiency of available DL solutions. While many of the Deep Neural Networks (DNNs) ideas were first proposed in the late 80s and early 90s [1], their success back then was hindered by the limited availability of computational resources, training data, and associated models. Consequently, the recent surge of DL applications can be attributed to the availability of the following: 1) multi-/many-core processing elements like CPUs and Graphics Processing Units (GPUs), 2) HPC systems/cloud platforms that can be used for scaling-up and scaling-out DL applications 3) large-scale and high-quality datasets such as ImageNet [2], and 4) highly accurate DL models from various application domains. These trends have revived the community's interest in DL and led to the development of several software packages and frameworks such as PyTorch [3] and TensorFlow [4]. These frameworks have enabled the true "democratization of AI" by making models and datasets available to the community in a simplified and highly-productive manner.

In order to address more challenging DL problems and achieve better performance, models and datasets have been exponentially growing in complexity and size. As a result, the memory and computational requirements to train such models have increased as well. Since training DNN models inherently involve procedures that can be executed in parallel, GPUs have enabled developers to significantly reduce the training time of DNN models. Recent GPU architectures—such as NVIDIA Ampere—have thousands of cores, large memory capacity, and DL-specific hardware components. For instance, the high-end NVIDIA A100 GPU [5] comes with 80 GB memory, 108 streaming multiprocessors (SMs), 6912 CUDA [6] cores and 4 Tensor Cores per SM. There are mature and efficient software stacks like CUDA and cuDNN [7] that have been optimized to take advantage of the advanced hardware features offered by these devices. Also, it is possible to connect these GPUs via low-latency and high-throughput intra-node and inter-node interconnects like NVLink and InfiniBand. This makes it possible to accelerate DL training beyond the limits of a single device by efficiently utilizing multiple accelerators simultaneously—in parallel—on high-end systems leveraging communication middleware like Message Passing Interface (MPI) [8] or NVIDIA Collective Communication Library (NCCL) [9]. Several parallelization techniques including data/model/hybrid parallelism [10] exist in the literature for parallel execution of DNN training workloads

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002.

2640-0316/22/$31.00 ©2022 IEEE
DOI 10.1109/HiPC56025.2022.00017

A. Motivation

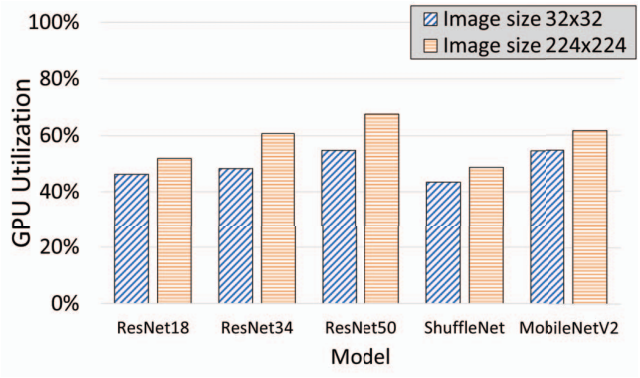


Fig. 1. NVIDIA A100 GPU utilization during DNN training of different models with different input sizes.

Modern GPUs, like NVIDIA A100, are computational workhorses in modern HPC systems and are used in parallel to reduce DNN training time. However, these accelerators are not fully utilized by DNN training workloads. This is especially true for small-to-medium DL models and/or input size. Figure 1 shows an evaluation conducted using NVIDIA Nsight Systems [11] to support our hypothesis of under-utilization of modern GPUs for certain DNN training workloads. The idea here is to find the average percentage of the resources utilization of NVIDIA A100 GPU during the training phase of different DNN models—including ResNet18, ResNet34, ResNet50, ShuffleNet, and MobileNetv2—with two input sizes of 32×32 and 224×224. We observe under-utilization of the GPU across all models with some variation. We observe lesser GPU utilization for smaller models and input sizes. For example, ResNet18 [12] achieves a 43% utilization for 32×32 pixels input image size. On the other hand, larger models, like ResNet50, can achieve 68% utilization with the larger 224×224 pixels image size. This clearly shows that there are compute cycles available on these GPUs that can be used to optimize DNN training even further. Therefore, the primary motivation of this paper is to explore strategies and methods to increase the occupancy and utilization of GPUs.

B. Problem Statements

In the context of the motivation presented in Section I-A, this paper tackles the following problem statements:

1) What hardware/software techniques can be used to increase the utilization of processing elements like GPUs for DNN training workloads? Can these strategies improve overall hardware efficiency for small-to-medium DL models and/or input sizes?
2) Is it possible to configure or tune hyperparameters, including the training batch size and/or input size, to improve hardware utilization during DNN training? How can we take advantage of architecture-specific features to increase GPU utilization for DNN training?
3) In modern DNN training workloads, the performance of data loading is a sizable portion of the overall DNN training time. What is the impact of the number of data loading workers on the overall performance of DNN training? What is the impact of data loading workers on the utilization of compute hardware?

These problem statements are elaborated further in Section III, which discusses the challenges in maximizing GPU performance for DNN workloads.

C. Overview

In this paper, we explore the potential of leveraging NVIDIA's **Multi-Process Service (MPS)**, which allows running multiple CUDA applications concurrently on the same GPU, coupled with MPI for DNN training. We propose AccDP, a novel data-parallelism-based approach to increase the parallel workload assigned to the GPU in order to accelerate the performance of distributed DNN training. We extensively study the various factors that impact the training throughput of our approach and the challenges that they present including: 1) Analyzing common data loading mechanisms and their bottlenecks. 2) Characterizing the impact of batch size on GPU utilization. 3) Evaluating the impact of varying the number of MPS processes and percentage of active threads on our proposed design. 4) Conducting a comparison between the performance of regular data parallelism and our proposed design on a single NVIDIA A100 GPU which yields up to 58% improvement in training throughput. 5) Conducting a multi-node evaluation which shows up to 62% improvement on 16 NVIDIA A100 GPUs using our proposed design.

D. Contributions

This paper makes the following contributions:

1) Propose a novel data-parallelism-based training approach using MPS and MPI to improve the utilization of GPU in distributed DNN training. *To the best of our knowledge, this is the first work that uses MPS and MPI to accelerate distributed DNN training.*
2) Conduct a comprehensive evaluation of our proposed design on 5 different DNN models and report improvements in training throughput of up to 37%, 58%, 33%, 31%, and 25% for models ResNet18, ShuffleNet, MobileNetv2, ResNet34, and ResNet50 respectively on a single NVIDIA A100 GPU and up to 62% for the ShuffleNet model on 16 A100 GPUs.
3) Provide in-depth analysis of the impact of different DNN training parameters including the mini-batch size, input data size, model size, number of MPS processes per GPU, and percentage of active GPU threads per process on our proposed design.
4) Examine current data loading mechanisms, identify data loading bottlenecks, study the impact of the number of data loading workers on training throughput, and propose guidelines to alleviate the data loading overhead in DNN training.

The rest of the paper is organized as follows: Section II establishes the necessary background for key concepts in the paper. Section III describes the main challenges to improving

GPU utilization for DNN training workloads. Section IV introduces our proposed design and its implementation with existing DL frameworks. Section V provides analysis and guidelines to optimize data loading. Sections VI and VII present a comprehensive evaluation and discussion on the proposed design. Section VIII reviews related works in the literature. Finally, we conclude the paper in Section IX.

II. Background

A. Distributed DNN Training

Deep Neural Networks (DNNs) are neural networks that have at least two hidden layers between the input and output layers. The training of DNNs mainly consists of two phases 1) forward propagation and 2) backward propagation. In the first phase, input is fed into the network and is propagated throughout the layers to generate an output. Based on that output, an error is calculated by comparing the generated and expected outputs. In the backward propagation phase, gradients are calculated for the different layers. Weights are then updated based on the calculated gradient values. A form of the gradient descent algorithm is usually used to optimize DNN models and minimize the loss. There are many variants of DNN models that are intended for different purposes and downstream tasks including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformer models. In this paper, we mainly focus on CNN models.

Distributed DNN training is performed on multiple workers in parallel. Workers can reside in one (intra) or multiple (inter) machines. There are several well-established approaches to distribute DNN training such as a) Data parallelism b) Model parallelism, and c) hybrid parallelism. In data parallelism, DNN models are replicated across all workers and the dataset is divided among workers. The workers simultaneously perform forward and backward pass on their chunk of the datasets. Model instances are synchronized by aggregating gradients across the workers. The Allreduce communication operation is used in order to synchronize these models. In model parallelism, on the other hand, the model is divided into multiple machines or processing units. Communication operations are used in order to implement the forward and backward propagations across the workers.

B. Deep Learning Frameworks

Deep Learning frameworks provide interfaces, building blocks, and primitives to define and design DL models and implement training and validation cycles on various types of accelerators. Most DL frameworks such as TensorFlow [4] and PyTorch [3] are optimized for GPU performance. These frameworks rely on GPU-accelerated libraries such as CUDA [6] and cuDNN [7] to deliver high-performance training and inference.

Furthermore, many frameworks provide support/API for distributed DNN training such as PyTorch Distributed [13], TensorFlow distributed, and Horovod [14]. In this paper, we use Horovod to perform distributed DNN training. Horovod is a popular open-source software framework for distributed deep learning which supports multiple DL frameworks including PyTorch, TensorFlow, MXNet, and Keras. Horovod relies on MPI operations including MPI_Allreduce and MPI_Bcast to enable communication of model parameters and gradients between workers. It provides an interface in Python with a high-level API for users. Horovod supports scaling to multiple GPUs whether they reside on single or multiple machines by taking advantage of collective operations.

C. Multi-Process Service (MPS)

MPS is a client-server runtime implementation of the CUDA API that enables the sharing of GPU resources. It is a logical partitioning mechanism that is designed to allow the execution of multi-process CUDA applications concurrently. MPS started with NVIDIA Kepler-based GPUs and is also extended to Volta-based GPUs. An MPS server can support up to 16 clients (pre-Volta) and 48 clients (Volta-based) CUDA contexts per GPU device. These clients are usually launched in the form of MPI processes. Figure 2 shows the Volta-based MPS

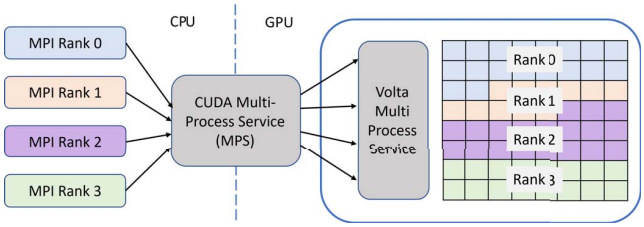


Fig. 2. NVIDIA Volta-based Multi-Process Service (MPS) scenario launching 4 clients concurrently on a single GPU with 30% active threads per client.

workflow. In this scenario, 4 MPI ranks are submitted to MPS from the CPU side. On the GPU side, resources are allocated logically on a need-basis. Each MPS client has its own GPU address space, which is a security improvement from its Kepler-based predecessor. Additionally, users can provision the MPS clients' resource allocation by assigning a value to the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` environment variable. A client may use threads up to the percentage defined; however, it cannot exceed that percentage. The logical allocation scheme allows for oversubscription of resources, shifting the allocation based on the process' needs. As shown in figure 2, in the case of oversubscription where every client has a 30% limit on resource usage, when rank 1 is not using all of its allocation, rank 2 can grow and use available resources.

III. Challenges in Maximizing GPU Performance for Deep Learning Workloads

Maximizing GPU utilization for DNN training is challenging for multiple reasons. First and foremost, there are several interrelated parameters that impact the training performance including the number of available CPU cores, batch size, number of data loading workers on the CPU, training input size, and memory limitations on the GPU. Each of these parameters needs to be studied in isolation. Furthermore, using MPS in the training process introduces more parameters to take into consideration such as the number of MPI processes

to run on a single GPU and the allocated percentage of overall threads per process. In this section, we highlight some of the key challenges in maximizing the GPU performance for DL workloads.

Challenge 1: Is it possible to increase the utilization of a single GPU for DNN training, especially for small-to-medium sized DNNs? Modern GPUs have an outstanding ability to process multiple computations simultaneously due to their great number of cores. Training DNN models can take advantage of these available cores to execute complex matrix operations in parallel. However, the nature of DNN training imposes some sequential order for these operations where the input of one phase depends on the output of the previous one. Therefore, the resources available on highly capable GPUs may not be fully utilized during the training, leaving many cores idle especially for small-to-medium size DNN models. There is potential to utilize the idle cores on the GPU, but we need a new training approach to introduce more parallelism without disrupting the flow of the data in the different training phases.

Challenge 2: How can we identify the impact of varying the batch size and input size on the GPU utilization during DNN training? The batch size in DNN training is a critical parameter for both optimization and performance purposes. In this paper, we purely focus on the training performance in order to achieve maximal GPU utilization and in turn the highest training throughput. While increasing the batch size improves the performance since more data instances are being processed in parallel, it can either reach a saturation point or memory constraints. The input size also has significant impact on the GPU utilization. The smaller the input size, the less parallel workload is assigned to the GPU. Therefore, the impact of both these factors must be taken into consideration to enhance the GPU utilization for DNN training.

Challenge 3: How can we identify the impact of the number of data loading workers on the performance of DNN training? DL frameworks offer multiple tools to hide the overhead of data loading and augmentation during DNN training. However, how can we fully exploit the overlap between data loading and computation in order to accelerate the training process? This is dependent on the method used by the DL framework, the number of available cores on the CPU, and the user-defined number of processes dedicated to performing the data loading. In order to maximize the overlap, we need to first analyze how the data loading mechanism works and conduct an evaluation to determine the best number of data loading workers based on the model and input sizes.

Challenge 4: How can we take advantage of architecture-specific features to increase GPU utilization for DNN training? In order to support a variety of use cases, hardware manufacturers constantly work on developing new features to either supplement the architecture limitations or complement strengths in their designs. As we mentioned in section I-A, GPUs may be underutilized during DNN training where a great portion of the compute resources sits idle. NVIDIA GPUs recently released new features including the multi-process services (MPS). The purpose of this service is to enable concurrent multi-process CUDA applications, balance tasks between CPU and GPU, and address the inefficiency in using GPUs by taking advantage of inter-rank parallelism. This paper explores the potential of exploiting such a feature for DNN training and integrating it with existing DL frameworks.

IV. Proposed Design and Implementation

A. Utilizing MPS and MPI in GPU-Based DNN Training

In this subsection, we present AccDP—our purposed design which utilizes MPI and MPS to improve GPU utilization and reduce overall training time. We first describe the traditional data parallelism approach and then we show our proposed distributed data-parallel training solution that combines MPS and traditional data parallelism to accelerate DNN training.

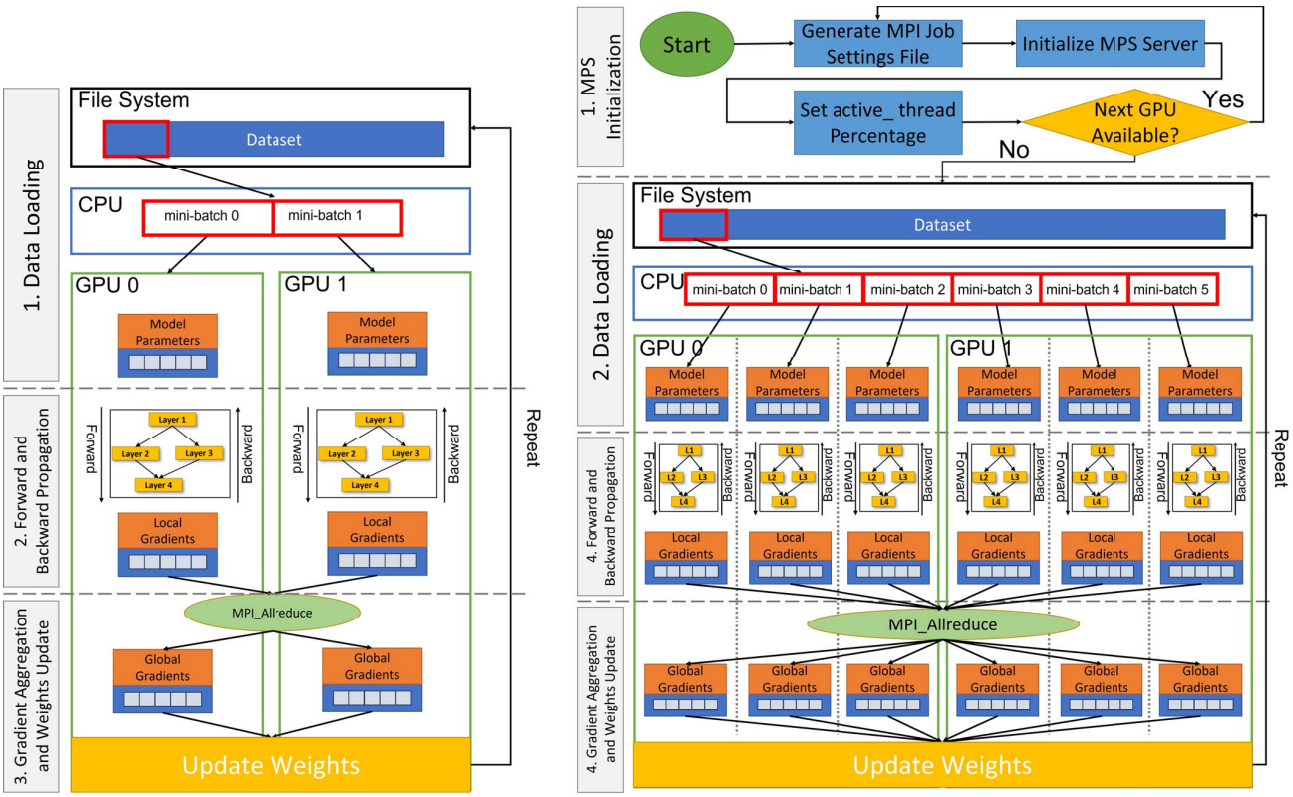
Data parallelism replicates the model and distributes the dataset among all participating processes. It executes forward and backward pass simultaneously on all processes on its data partition. The MPI Allreduce operation is used to reduce local gradients and the accumulated gradients are used to update the model on all processes. Figure 3(a) shows traditional data parallelism on 2 GPUs in which only one replica is created per processing element. Training is executed in a loop over the entire dataset. Each iteration is called a step where a mini-batch (a random subset of dataset) is used to train the model. Broadly, DNN training iteration can be divided into three phases:

1) Data Loading: In this phase, a batch of data is fetched from the file system into the CPU memory. The CPU then applies data augmentation and places data in a ready queue. The GPU fetches data from the queue as it finishes training on the previous mini-batch.

2) Forward and Backward Propagation: At the beginning of training, process 0 broadcasts model parameters to all processing elements. The forward pass is executed by calculating the layers' activations and propagating through the model. The backward pass is executed by calculating the gradients for each layer. Gradients are then averaged for all of the data samples in the current local mini-batch to produce the local gradients.

3) Gradient Aggregation and Weights Update: In this phase, MPI Allreduce aggregates local gradients on GPUs and makes the global gradients available to all the GPUs. Each GPU updates its own version of model parameters using global gradients.

As shown in figure 1, regular data parallelism may not utilize all the available GPU resources due to the sequential nature of DNN training. We improve GPU utilization by extending data parallelism and proposing a new design that creates multiple data-parallel processes per GPU. In proposed method, we replicate the model and distribute the datasets among multiple processes running on a single GPU. To manage the resource allocation, we use NVIDIA's MPS tool. MPS can logically and physically allocate resources on a process-need-basis. Without MPS, different processes cannot share the physical resources efficiently as GPU daemon uses time slicing to share GPU cores. Therefore, the GPU would context



(a) Workflow of traditional data parallelism using two GPU devices.

(b) Workflow of AccDP (proposed design) using data parallelism on two GPU devices and 3 MPS processes per GPU.

Fig. 3. Comparison of the workflow of traditional data parallelism and proposed design on two GPU devices.

switch between the different processes constantly, which has a huge performance overhead and does not improve overall GPU utilization.

Figure 3(b) shows the workflow of AccDP. Each GPU is still responsible for processing the same portion of the dataset. The proposed design consists of the following phases:

1) MPS Initialization: In this phase, we generate a settings file for each GPU. The settings file includes necessary information for MPS to operate optimally including the user ID, number of GPUs, number of MPS processes, GPU IDs, and active thread percentage. The settings file ensures that each MPS process can only see its designated GPU for added security. An MPS server is then initialized for each GPU. It is worth noting that Volta-based GPUs do not require initializing an MPS server; however, it is a necessary step if we want to specify the percentage of active threads. Finally, we launch an MPI job using the settings file to initialize different ranks to their corresponding GPUs.

2) Data Loading: This phase is similar to the traditional data parallelism method, except that we further divide the fetched data. The number of mini-batches corresponds to the number of GPUs multiplied by the number of processes per GPU.

3) Forward and Backward Propagation: At the beginning of this phase, model parameters are broadcasted to all GPUs. Each GPU replicates the model multiple times depending on the number of MPS processes. In figure 3(b), we have three processes per GPU; thus, we replicate the model three times on each GPU. We proceed normally with the forward pass where it is executed by computing the layers' activations and propagating the activations through the model. The backward pass is executed by computing the gradients for each layer. Gradients are then averaged for all of the data samples in the current mini-batch to produces the local gradients.

4) Gradient Aggregation and Weights Update: In this phase, local gradients from each MPI rank are aggregated using MPI_Allreduce to obtain the global gradients. MPI is necessary to facilitate communication for all of the following cases: 1) between nodes 2) between GPUs on the same node 3) between MPS processes on a single GPU. Model parameters for each MPI rank are then updated separately using the obtained global gradients. The training then moves on to the next step (i.e. iteration).

B. Integration of MPS with Existing Distributed Training Frameworks

We implement our proposed design using a combination of shell scripts and Python code and integrate it with the Horovod code in order to automate the process of initializing MPS servers, setting appropriate runtime parameters, assigning sufficient workload to GPUs, and launching MPI

jobs. MPS servers are first initialized on each GPU device on the different nodes. While Volta-based MPS does not require initializing MPS servers, doing so can allow us to set the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`, which in turn leads to optimized performance as shown later in the evaluation section. After the initialization of the servers, we isolate GPU visibility on each set of processes for added security and restrict processes to a single GPU. This guarantees that processes are submitted to their designated GPUs correctly and does not use other GPUs on the node. Finally, we launch the MPI job with the number of processes equal to:

$$\#Procs = \#Nodes \times GPUs/Node \times Clients/GPU$$

V. Analysis and Guidelines to Optimize the Data Loading Mechanism in PyTorch

Data loading is a critical component of DNN training that can lead to severe performance bottlenecks. If not handled properly, data loading can drastically limit any effort to improve DNN training performance. Therefore, in order to alleviate the overhead caused by data loading, we provide in-depth analysis and suggest guidelines to overcome data loading limitations. In this section, we first dissect the data loading mechanism used by common deep learning frameworks such as PyTorch. Then we identify data-loading-related bottlenecks using the benchy profiling tool [15]. Finally, we propose guidelines to minimize the data loading overhead.

A. Analyzing the workflow of the data loading mechanism used in PyTorch

The PyTorch framework contains several useful tools to ensure smooth data loading and augmentation. The `torch.utils.data.Dataloader` class offers compatibility and flexibility to work for both map-style and iterable-style datasets. This class is used to fetch data from disk to host memory and from host memory to GPU if GPUs are used in the training. PyTorch supports two types of data loading schemes: 1) sequential and 2) parallel. In the sequential scheme, for each loop iteration, we wait for the data loader to fetch the data and then we perform the model computations. In the parallel scheme, multiple data loaders work concurrently to fetch the data and overlap it with the model computation on GPUs. Among other options, the user can specify the number of data loading workers when defining the data loader object. Varying the number of workers leads to significant change in performance. In order to select the optimal number of data loading workers, we need to first analyze how PyTorch utilizes the multi-process scheme to improve data loading.

In figure 4, we analyze the workflow of the multi-process data loading scheme in PyTorch. The figure is divided into the following three sections:

1) Application-Level Code: The application-level code is the part that is exposed to the user. The user is responsible for defining the data sampler and the data loader objects. Inside the training loop, the user expects the data loader to fetch the data for each loop iteration.

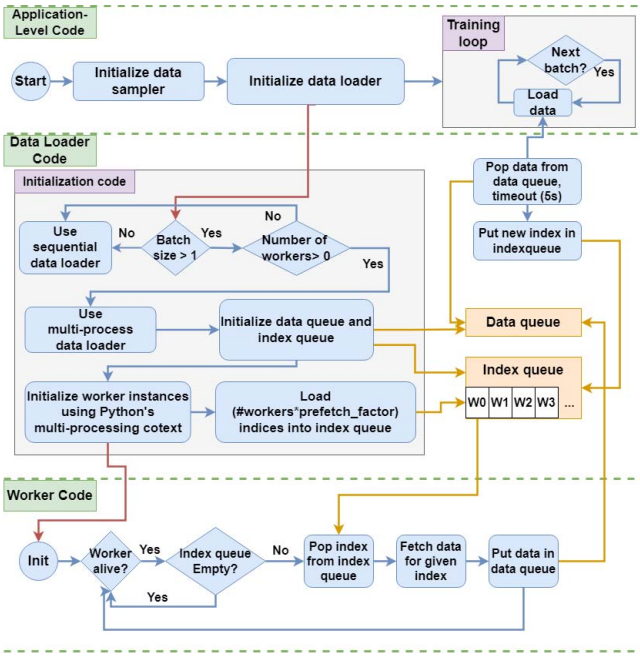


Fig. 4. Multi-process data loading workflow with the PyTorch framework.

2) Data Loader Code: The data loader code is responsible for initializing the data queue, the index queue, and the data loading workers. Data loading module checks if the batch size is greater than one and if the number of workers is greater than zero. If either is not true, the code uses the sequential data loader. If both conditions are satisfied, the multi-process scheme is used. Two queues are initialized in the next step; 1) The data queue where the workers store the data instances and are later popped by the training loop and 2) The index queue which stores the next set of indices to be fetched by the data loading workers and stored in the data queue. In the next step, each worker is initialized. Finally, indices are loaded into the index queue. This process is done once during the initialization phase and the number of initial indices to be loaded is determined by the number of workers multiplied by a prefetch factor.

3) Worker-Level Code: This part of the code defines the logic of the independent data loading workers initialized by the data loader. Each worker keeps checking for entries in the index queue. If entries are found, the worker fetches the new data instances (or mini-batch) and places them in the data queue. This process is repeated as long as the data loading worker is alive.

B. Identifying data loading bottlenecks

Data loading bottlenecks are caused when the average time needed to load a data instance takes longer than the average time needed to run forward and backward computations on it. In order to identify such cases, we use the benchy profiling tool [15] to run three sets of experiments measuring throughput for 1) Full training which is the throughput of the regular DNN training, 2) Synthetic training which is the training throughput

on dummy data (That is no data loading is involved in these experiments), and 3) I/O which is the throughput of loading data from the file system to the host memory to the GPU memory.

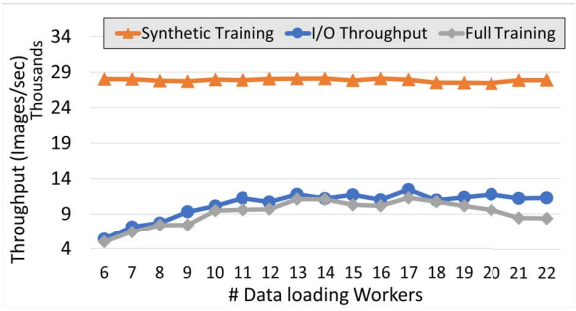


Fig. 5. Analysis of data loading impact on DNN training with ResNet18.

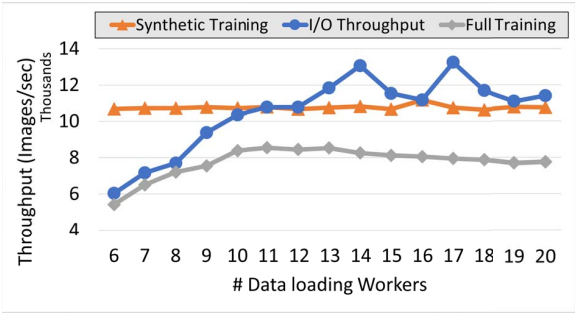


Fig. 6. Analysis of data loading impact on DNN training with ResNet50.

Figure 5 shows the three throughput measures discussed earlier for training the ResNet18 model with 32×32 input size on two NVIDIA A100 GPU. The reported numbers are per GPU. In this experiment, we can see the full training performance is bottlenecked by the I/O throughput. Synthetic training represents the potential capability of the GPU to perform forward and backward computations if not capped by the data loading performance. However, in this case, the full training throughput cannot exceed the I/O throughput; therefore, we observe this kind of bottleneck. Still, we can improve the performance by picking the number of workers that maximizes the I/O. Figure 6 shows the same set of experiments but for the ResNet50 model. Since ResNet50 is a larger model, it achieves lower synthetic training throughput. The I/O throughput catches up with the synthetic training throughput at around 11 data loading workers. However, we can see an overhead between the full and synthetic training due to the partial overlap between the data loading and computation.

C. Guidelines to select the optimal number of data loading workers

As we have seen in figures 5 and 6, the number of data loading workers has a significant effect on the full training throughput. Increasing the number of workers beyond a certain point exhausts the CPU resources which is undesirable especially when the CPU is responsible for data loading and augmentation for more than one GPU. In subsection V-B, we identified two types of bottlenecks 1) I/O bottleneck and 2) Computation bottleneck. To maximize the performance in the first case, we simply pick the number of workers that gives the best I/O throughput. In the second case, we keep increasing the number of workers until the I/O throughput exceeds or matches the synthetic training throughput. By running a few preliminary iterations before the actual training, we can find the optimal number of data loading workers. We use this approach throughout the evaluation section to optimize both the regular training and proposed design runs.

TABLE I
HARDWARE CONFIGURATION FOR NODES USED IN THE EVALUATION

GPU	Two NVIDIA A100 with 40GB of GPU memory
GPU Interconnect	PCIe
CPU	Two AMD EPYC 7713 64-core processors @3.7GHz (128 cores)
Memory	256 GB of RAM
Storage	Lustre file system
Interconnect	Mellanox ConnectX-6 InfiniBand EDR 100Gb/s Adapter

VI. EVALUATION OF PROPOSED DESIGN

This section provides a comprehensive evaluation of the proposed design and comparison with regular distributed data-parallel DNN training both on a single GPU and multiple nodes. First, we describe the experimental setup we used on the software and hardware levels. Then we explore the effects of the training batch size on the GPU utilization. We then evaluate the impact of the number of MPS processes launched on a single GPU and the percentage of active threads per process on the training throughput. Next, we provide a comparison to highlight the benefits of the proposed design on a single GPU. Finally, we evaluate the scalability of the proposed design and compare it with the regular distributed DNN training.

A. Experimental Setup

1) Software: Our experimental environment uses Python 3.9.7 [16], PyTorch 1.10.2 [3], Torchvision 0.11.3 [17], Horovod 0.23.0 [14] with CUDA 11.3 [6] and cuDNN 8.2.0 [7]. For communication, we use MVAPICH2-GDR v2.3.7 [18]. The system runs on Ubuntu SMP Linux kernel 4.18.0-348.2.1. We use two ImageNet-like datasets each consisting of 400,000 images with sizes 32×32 pixels and 224×224 pixels respectively.

2) Hardware: Our experimental testbed utilizes an HPC system with 16 compute nodes each consisting of two EPYC AMD CPUs with 64 cores each and two NVIDIA A100 GPUs with 40GB of graphic memory. Table I shows the hardware configuration per node.

B. Impact of Batch Size on GPU Utilization

Our evaluation starts by analyzing the impact of the training batch size on the GPU utilization with regular DNN training. This experiment is repeated for all future evaluations where we pick the batch size that yields the best overall performance.

We expect to see an increase in GPU utilization as we increase the batch size. This is because we are assigning more parallel workload to the GPU. However, at a certain point, the GPU utilization either saturates or we run out of GPU memory. Figure 7 shows the GPU utilization for different batch sizes

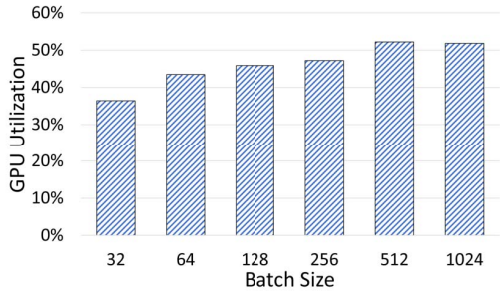


Fig. 7. Impact of the training batch size on GPU utilization for the ResNet18 model with 224×224 input size.

with the ResNet18 model and input size of 224×224. The GPU reaches a max of 52% utilization where it peaks and saturates at batch size 512. It is also worth noting that the GPU runs out of memory at batch size 2048.

C. Impact of Number of MPS Processes and Percentage of Active Threads on DNN Training Performance

In this subsection, we evaluate the impact of varying the MPS initialization options on our proposed design. The purpose of these experiments is to highlight the importance of selecting the appropriate number of MPS processes per GPU and the percentage of active threads. Again, these experiments are repeated for all future evaluations for the different models where the final goal is to maximize the GPU utilization and in turn the DNN training throughput. Figure 8 shows

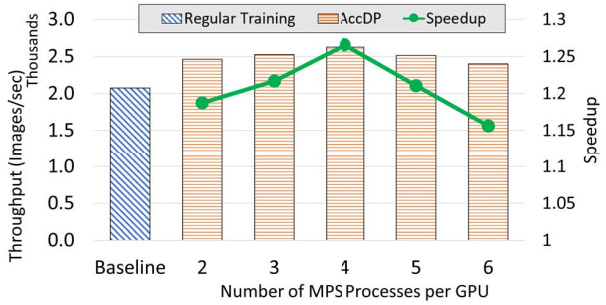


Fig. 8. Impact of number of MPS processes on DNN training throughput for the ResNet18 model with 224×224 pixels input size.

the throughput in images per second of the proposed design while varying the number of MPS processes per GPU for the ResNet18 model with 224×224 input size. We include the best performance from the regular training as a baseline for reference. We observe that using 4 MPS processes per GPU yields the highest throughput of around 2,700 images per second. We also observe that increasing the number of MPS processes beyond 4 decreases the performance.

Figure 9 shows the throughput in images per second of the proposed design while varying the percentage of active threads per MPS process for a total of 4 MPS processes. The model

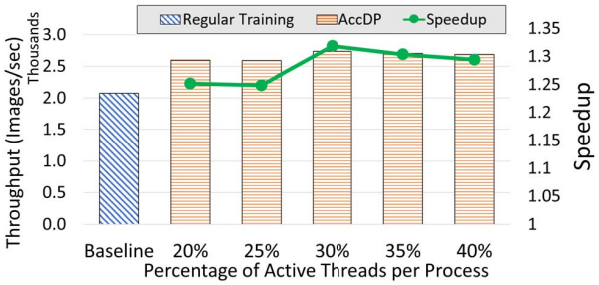


Fig. 9. Impact of varying the percentage of active GPU threads per MPS process on the ResNet18 model and 224×224 for input size.

used for training is ResNet18 with 224×224 input size. We observe the best performance when we slightly oversubscribe the GPU resources at 30% per process. This gives each MPS process the flexibility to use more or fewer resources with a 5% margin of the overall available GPU cores. Loosening this constraint further at 35% and 40% affects the performance negatively.

In fact, we empirically find that the settings shown above at 4 MPS processes per GPU and 30% active threads per MPS process deliver the best performance for the majority of the upcoming training scenarios whether we are running on a single GPU or scaling out to more number of nodes.

D. Single GPU Evaluation of proposed design

In this subsection, we evaluate our proposed design on single GPU using 5 different CNN models: 1) ResNet18, 2) ResNet34, 3) ResNet50 [12], 4) ShuffleNet [19], and 5) MobileNetV2 [20]. We use regular DNN training on a single GPU as a baseline for comparison. To ensure a fair comparison, we run multiple regular training experiments while incrementing the batch size and number of data loading workers. We choose the best performance and include it in the evaluation as we are purely interested in training throughput. We use a similar method with the proposed design where we increment the number of MPS processes, batch size, and number of data loading workers to get the best performance. We perform these experiments for two different image sizes 1) 32×32 pixels and 2) 224×224 pixels.

Figure 10 shows throughput in thousands of images per second for both the regular training and proposed design for the 5 different CNN models. We observe a max of 58% improvement using the proposed design over regular training with the ShuffleNet model. The improvement varies between 25%-58% based on the model. We notice that for relatively smaller models like ResNet18 and ShuffleNet we get better improvement. Looking back at figure 1, this behavior can be attributed to the fact that the GPU utilization for the smaller models is lower than the mid-sized models.

Figure 11 shows throughput in thousands of images per second for the regular training and proposed design but with a larger input size of 224×224 pixels. We can still observe up to 42% improvement with the ShuffleNet model. The improvement varies between 7%-42% based on the model size. By also looking at figure 1, we can see that the GPU

utilization for the input size of 224×224 pixels is higher across all models. This explains the slightly smaller improvement compared to the 32×32 pixels input size.

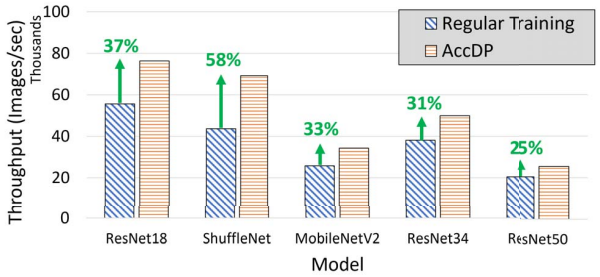


Fig. 10. Training throughput comparison between regular training and AccDP (proposed design) for different DNN models on single GPU with 4 MPS clients and 32×32 pixels image size.

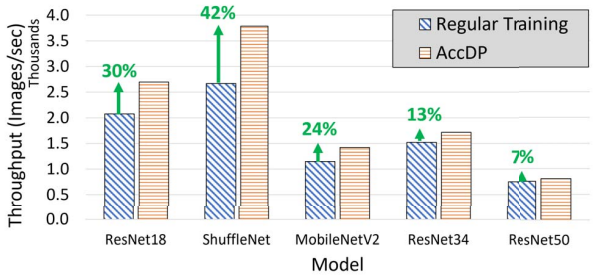


Fig. 11. Training throughput comparison between regular training and AccDP (proposed design) for different DNN models on single GPU with 4 MPS clients and 224×224 pixels image size.

E. Multi-node Evaluation of proposed design

In this subsection, we expand on the single GPU experiments by comparing the regular training to the proposed design on up to 16 GPUs. We run this evaluation on three models: 1) ResNet18 2) ResNet34 and 3) ShuffleNet. We fix the input size to 224×224 pixels to highlight the scaling potential on a larger image size similar to what is commonly used in DNN applications. We choose the optimal values for the batch size and number of data loading workers to report the best possible performance for both the proposed design and regular training. Figure 12 shows the training throughput

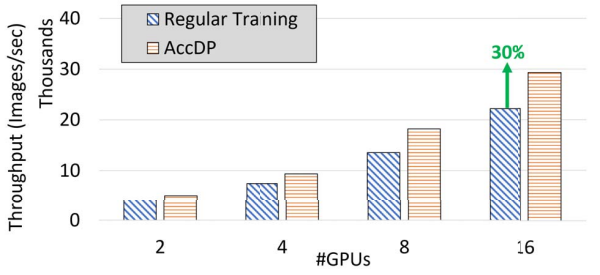


Fig. 12. ResNet18 training throughput comparison between regular training and AccDP (proposed design) for different DNN models on up to 8 nodes 2 GPUs per node (16 GPUs) with 4 MPS clients per GPU.

in thousands of images per second for the ResNet18 model. We can observe the benefits of using the proposed design across different number of GPUs. On 16 GPUs, we observe a 30% improvement which is consistent with the single node experiments.

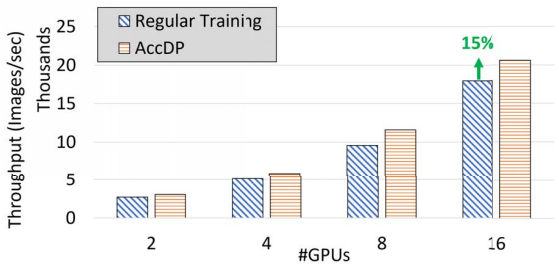


Fig. 13. ResNet34 training throughput comparison between regular training and AccDP (proposed design) for different DNN models on up to 8 nodes 2 GPUs per node (16 GPUs) with 4 MPS clients per GPU.

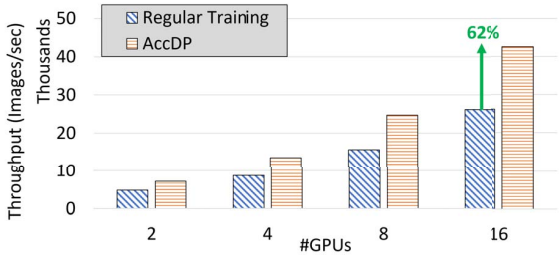


Fig. 14. ShuffleNet training throughput comparison between regular training and AccDP (proposed design) for different DNN models on up to 8 nodes 2 GPUs per node (16 GPUs) with 4 MPS clients per GPU.

Figure 13 shows the training throughput for the regular training and proposed design with the ResNet34 model. We also see consistent improvement regardless of the number of used GPUs. We report an improvement of 15% on 16 GPUs. Figure 14 shows the scaling of the ShuffleNet model. We see a considerable benefit of 62% on 16 GPUs.

VII. Discussion and Summary of Results

In figure 1, we have shown that the GPU may be underutilized during regular DNN training across different model sizes and input sizes. Figure 7, shows that increasing the batch size leads to better utilization, but at a certain point, the improvement can either saturate or be limited by available GPU memory. In figures 8 and 9, we show the impact of the number of MPS processes per GPU and the percentage of active threads per MPS process. We empirically find that running 4 MPS process and oversubscribing the GPU resources at 30% active thread percentage lead to the best performance across different models and input sizes. In figures 10 and 11, we observe the improvement gained by using AccDP with different models and input sizes. We report a max improvement of 58% in training throughput for input size 32×32 and 42% for input size 224×224 with the ShuffleNet model. In figures 12, 13, and 14, we evaluate the performance of AccDP on up to 16 GPUs for different models with input size 224×224. For the multi-node experiments, we observe a max of 62% improvement for the ShuffleNet model. Throughout all of these experiments, we take into consideration the optimal number

of data loading workers following the guidelines defined in section V-C.

VIII. Related Work

While our work focuses on improving the *training* workload in DL applications by enhancing the GPU resource utilization, there are several works that explore improving the *inference* throughput in DL applications. Jain et al. [21] propose a dynamic space-time scheduling technique that combines batch-level, temporal (CUDA context switching), and spatial (CUDA Hyper-Q [22]) multiplexing to improve GPU inference performance. Dhakal et al. [23] propose GSLICE which is a platform to support cloud-based low-latency inference applications that builds on top of CUDA MPS. It provides a dynamic management scheme to appropriate GPU resources across different Inference Functions (IFs) and multiplex multiple instances of them on the GPU. Chen et al. [24] propose EUGE, which takes advantage of CUDA MPS to improve the GPU utilization for DNN-based video analysis applications. They employ model sharing to save GPU memory since DNN inference does not alter the state of the model like training does. Yu et al. [25] conduct a survey on multi-tenant Deep Learning Inference on GPU. They explore different techniques that can be used to enhance the GPU utilization for DL inference applications including NVIDIA Hyper-Q and MPS. Gray et al. [26] explore using MPS with GROMACS [27] which is a simulation package for biomolecular systems. They propose running multiple simulations concurrently on the GPU to increase the application throughput.

IX. Conclusions and Future Work

In this paper, we proposed AccDP, a data-parallelism based design to improve the GPU utilization for DNN training by leveraging CUDA MPS and MPI. Our motivating analysis shows that modern GPUs, like NVIDIA A100, may not be fully utilized by DNN training workloads. Our design extends data parallelism to instantiate multiple concurrent processes on the same GPU. Our evaluation yields up to 37%, 58%, 33%, 31%, and 25% training throughput improvements for models ResNet18, ShuffleNet, MobileNetv2, ResNet34, and ResNet50 respectively on a single NVIDIA A100 GPU and up to 30%, 62%, and 15% for the ResNet18, ShuffleNet, and ResNet34 models on 16 A100 GPUs. Additionally, we examine the data loading mechanism used in PyTorch, identify data loading bottlenecks, and propose guidelines to reduce the data loading overhead and optimize our design. Furthermore, we study the impact of varying the batch size, input size, number of MPS processes per GPU, and percentage of active threads per GPU. In future work, we would like to explore combining our design with other resource management tools such as Hyper-Q and Multi-GPU Instance (MIG). To the best of our knowledge, this is the first work that uses MPS and MPI to accelerate distributed DNN training.

References

[1] H. Wang and B. Raj, "On the origin of deep learning," 2017.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[4] 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 Systems, 2015," *Software available from tensorflow. org*, 2016.

[5] NVIDIA, "Nvidia a100 tensor core gpu." https://www.nvidia.com/en-us/data-center/a100/.

[6] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020.

[7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 2014.

[8] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Mar 1994.

[9] NVIDIA, "Nvidia collective communication library (nccl)." https://developer.nvidia.com/nccl.

[10] A. Jain, A. Shafi, Q. Anthony, P. Kousha, H. Subramoni, and D. K. Panda, "Hy-fi: Hybrid five-dimensional parallel dnn training on high-performance gpu clusters," (Berlin, Heidelberg), Springer-Verlag, 2022.

[11] NVIDIA, "NVIDIA Nsight Systems." https://developer.nvidia.com/nsight-systems.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[13] PyTorch, "torch.distributed." https://pytorch.org/docs/stable/distributed.html, 2021. [Online; accessed November 15, 2022].

[14] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," 2018.

[15] Romero, Josh, "Benchy profling tool." https://github.com/romerojosh/benchy.

[16] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[17] S. Marcel and Y. Rodriguez, "Torchvision the machine-vision package of torch," in *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, (New York, NY, USA), p. 1485–1488, Association for Computing Machinery, 2010.

[18] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, vol. 52, p. 101208, 2021. Case Studies in Translational Computer Science.

[19] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," 2017.

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2018.

[21] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica, "Dynamic space-time scheduling for gpu inference," 2018.

[22] NVIDIA, "Nvidia hyper-q." https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[23] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "Gslice: Controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, (New York, NY, USA), p. 492–506, Association for Computing Machinery, 2020.

[24] Q. Chen, G. Ding, C. Xu, W. Qian, and A. Zhou, "Euge: Effective utilization of gpu resources for serving dnn-based video analysis," in *Web and Big Data* (X. Wang, R. Zhang, Y.-K. Lee, L. Sun, and Y.-S. Moon, eds.), (Cham), pp. 523–528, Springer International Publishing, 2020.

[25] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, and X. Chen, "A survey of multi-tenant deep learning inference on gpu," 2022.

[26] A. Gary and S. Páll, "Maximizing gromacs throughput with multiple simulations per gpu using mps and mig." https://developer.nvidia.com/blog/maximizing-gromacs-throughput-with-multiple-simulations-per-gpu-\\using-mps-and-mig, Nov 2021.

[27] P. Bauer, B. Hess, and E. Lindahl, "Gromacs 2022.2 manual," June 2022.