ABSTRACT

MOKADAM, LAXMIKANT KISHOR. FLEET: Flexible Efficient Ensemble Training for Heterogeneous Deep Neural Networks. (Under the direction of Dr. Xipeng Shen).

Neural network architecture search and hyper-parameter tuning are the essential steps to select the best fit neural network model for the given learning task. An effective method to these steps is parallel training of an ensemble of Deep Neural Networks (DNN) on a cluster of nodes. Prior efforts have shown that data sharing, where the common preprocessing operation is shared across the DNN training pipelines, saves computational resources and improves pipeline efficiency. Data sharing strategy, however, performs poorly for a heterogeneous set of DNNs where each DNN has varying computational needs and thus different training rate and convergence speed. This project proposes FLEET, a flexible ensemble DNN training framework for efficiently training a heterogeneous set of DNNs. We build FLEET via several technical innovations. We theoretically prove that an optimal resource allocation is NP-hard and propose a greedy algorithm to efficiently allocate resources for training each DNN so that they can reach a similar training rate. We introduce checkpointing into this context to address the issue of different convergence speeds. We integrate data-parallel DNN training into ensemble training to mitigate the differences in training rates. Our experiments show that the proposed ensemble DNN training framework significantly improves the training efficiency of DNN ensembles without compromising the quality of the result.
© Copyright 2019 by Laxmikant Kishor Mokadam

All Rights Reserved
FLEET: Flexible Efficient Ensemble Training for Heterogeneous Deep Neural Networks

by
Laxmikant Kishor Mokadam

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Dr. Frank Mueller

Dr. Guoliang Jin

Dr. Seung-Hwan Lim

External Member

Dr. Xipeng Shen

Chair of Advisory Committee
ACKNOWLEDGEMENTS

I would like to extend my gratitude towards my thesis advisor Dr. Xipeng Shen for his continued support and insights towards this project. I also want to thank Hui Guan for her invaluable advice and guidance for the duration of my thesis research. Also, I would like to acknowledge the contribution of Dr. Seung-Hwan Lim for providing the guidance and resources required to complete this project.
BIOGRAPHY

The author originally hails from Nagpur, India. Prior to joining the graduate program at North Carolina State University, the author earned his Bachelor of Engineering (B.E.) in Computer Science & Engineering from Government College of Engineering, Aurangabad in 2015.
# TABLE OF CONTENTS

List of Tables .................................................................................. v

List of Figures .................................................................................. vi

Chapter 1  INTRODUCTION ............................................................... 1
  1.1  Background ........................................................................ 2
  1.1.1  DNN Training Pipeline .................................................. 3
  1.1.2  Data-Parallel DNN Training .......................................... 3

Chapter 2  FLEET .......................................................................... 5

Chapter 3  Problem Definition ........................................................ 8
  3.1  Complexity Analysis ........................................................... 10

Chapter 4  Greedy Allocation Algorithm ........................................... 12
  4.1  Principles .......................................................................... 13
  4.2  Algorithm ......................................................................... 13
  4.3  Flotilla Creation ................................................................. 15
  4.4  GPU Assignment ............................................................... 15
  4.5  Training ............................................................................ 17
  4.6  Complexity Analysis .......................................................... 19
  4.7  Application in the Local Paradigm ....................................... 19

Chapter 5  Implementation Challenges and Solutions ......................... 21
  5.1  Profiling ........................................................................... 22
  5.2  Communications between Preprocessors and Trainers ............... 23
  5.3  Dataflow Implementation .................................................... 23

Chapter 6  Evaluation ................................................................... 25
  6.1  Experiment Settings ............................................................ 25
  6.2  Measurements and Results .................................................. 27
  6.2.1  Characteristics of Experimental DNNs ............................... 27
  6.2.2  End-to-End Speedups ..................................................... 29
  6.2.3  Overhead ........................................................................ 30

Chapter 7  Related Work ................................................................. 31

Chapter 8  Conclusions ................................................................... 33

BIBLIOGRAPHY ............................................................................. 34
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Notations</td>
<td>9</td>
</tr>
<tr>
<td>6.1</td>
<td>Mean and standard deviation of the running length of DNNs in seconds. (80 GPUs, 100 DNNs)</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>Scheduling and checkpointing overhead</td>
<td>30</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Basic pipeline for DNN training [Pit18]</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>An illustration of data-parallel DNN training [SB]</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>An illustration of the ensemble training pipeline in FLEET. P1 and P2 are preprocessors and T1-T8 are trainers. There are four training groups, (T1), (T2, T3), (T4), (T5, T6, T7, T8), which train the four DNNs D1-D4 respectively. Edges indicate transfers of preprocessed images.</td>
<td>6</td>
</tr>
<tr>
<td>5.1</td>
<td>The profiled training rates (images/sec) of 100 DNNs in an ensemble.</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>Illustration of the dataflow implementation. Two DNNs, D1 and D2, are trained using four GPUs (Ranks 0-3) by two training groups, (T1) and (T2, T3, T4). T1 and T2 are training group masters.</td>
<td>24</td>
</tr>
<tr>
<td>6.1</td>
<td>Training rate of each DNN on single GPU.</td>
<td>27</td>
</tr>
<tr>
<td>6.2</td>
<td>Correlations between model size of a DNN and the training rate and the number of epochs till converge.</td>
<td>28</td>
</tr>
<tr>
<td>6.3</td>
<td>The averaged speedups over the baseline in terms of the end-to-end time for training a 100-DNN ensemble. The error bars show the variations.</td>
<td>28</td>
</tr>
<tr>
<td>6.4</td>
<td>Waiting time per GPU.</td>
<td>29</td>
</tr>
</tbody>
</table>
Recent years have witnessed rapid progress in the development of Deep Neural Networks (DNN) and their successful applications to the understanding of images, texts, and wavelet data from sciences to industry \[\text{Pat18; Mat18; RP12; Sze}\]. An essential step to apply a deep learning algorithm to a new data set is the selection of an appropriate network architecture and hyper-parameters \[\text{You15; ZL16; BB12; Pha18}\]. In this step, one needs to train models with various architectures and configurations until identifying the best model for a particular task. An effective strategy for this is to concurrently train a set of DNNs on a cluster of nodes, which is referred to as ensemble training of DNNs. We refer to an ensemble of DNN models with the same architecture as a \textit{homogeneous DNN ensemble}. Otherwise, the ensemble is called \textit{heterogeneous DNN ensemble}.

A common ensemble training strategy is to duplicate the training pipeline on multiple nodes to train DNNs in parallel. A typical DNN training pipeline is an iterative process including data fetching, preprocessing, and training. For the ease of description, we will refer to data fetching and preprocessing together as preprocessing unless noted differently. In the ensemble training, the training steps are not identical since we train models with different architectures and configurations. However, data preprocessing can be redundant across the pipelines, resulting in unnecessary CPU usage and even poor pipeline performance.

To eliminate the redundancies, Pittman et al. \[\text{Pit18}\] proposed data sharing where the common preprocessing operations are shared across training pipelines of all DNNs in an ensemble. They demonstrated that data sharing is an effective strategy to reduce computational resource utilization.
and improve pipeline efficiency. Their solution, however, assumes relatively homogeneous computational needs for DNNs in an ensemble. It may perform poorly for an ensemble of heterogeneous DNNs due to the variance of DNN model training from two algorithmic characteristics.

The first algorithmic characteristic is varying training rate. Training rate of a DNN is the compute throughput of processing units (such as CPUs and GPUs) used for training the DNN. An ensemble of heterogeneous DNNs contains DNNs with different architectures and configurations. Each DNN in the ensemble could have varying computational needs and thus different training rates (i.e., sample processing speed) with the same computing resources [Can16; Sze17]. If a DNN consumes preprocessed data slower than other DNNs, others will have to wait for the slower one before evicting current set of cached batches when we employ synchronized data fetching for data sharing to ensure that each DNN is trained using the entire dataset. This waiting lowers the utilization of computing resources in the cluster and delays the overall training time of the ensemble.

The second one is varying convergence speed. Due to the differences in network architecture or hyper-parameter settings, some DNNs may require a larger number of epochs (one epoch goes through all data samples once) to converge than others [Kri12; He16; Hua17; ZK16]. There can be scenarios where a subset of DNNs in the ensemble have already converged while the shared preprocessing operations have to keep prepossessed data for the remaining DNNs. Resources allocated to these converged DNNs will be under-utilized until the training of all the DNNs is completed.

To address the issues, we propose FLEET, a flexible ensemble DNN training framework for efficiently training a heterogeneous set of DNNs. We build FLEET via several technical innovations.

First, we formalize the essence of the problem into an Optimal Resource Allocation problem. We analyze the computational complexity of the problem and present an efficient greedy algorithm that groups a subset of DNNs into a unit (named flotilla) and effectively maps DNNs to GPUs in a flotilla on the fly. The algorithm incurs marginal runtime overhead while balancing the progressing pace of DNNs. Second, we develop a set of techniques to seamlessly integrate distributed data-parallel training of DNN, preprocessing sharing, and runtime DNN-to-GPU assignments together into FLEET, the first ensemble DNN training framework for heterogeneous DNNs. FLEET features flexible and efficient communications and effective runtime resource allocations.

Experiments on 100 heterogeneous DNNs on Summit-dev demonstrate that FLEET can speed up the ensemble training by 1.12-1.92X over the default training method, and 1.23-1.97X over the state-of-the-art framework that was designed for homogeneous DNN ensemble training.

1.1 Background

This section provides the necessary background of DNN training pipeline and data-parallel DNN training.
1.1.1 DNN Training Pipeline

As shown in Figure 1.1, a typical DNN training pipeline is an iterative process containing three main stages: data fetching, preprocessing, and training. In each iteration, data is fetched to the main memory and then run through a sequence of preprocessing operations such as decoding, rotation, cropping, and scaling. The preprocessed data is arranged into batches and consumed by the training stage. The batch size is the number of data the network trains simultaneously per step.

The modern high performance computing clusters have evolved into a hybrid structure that contains both CPUs and GPUs on each node. These heterogeneous CPU-GPU clusters are particularly useful for DNN training as CPUs and GPUs can work together to accelerate the training pipeline. Compared to the training stage, preprocessing is usually less computation intensive. To pipeline the preprocessing and DNN training, typically preprocessing is performed on CPUs while training on another batch of data happens simultaneously on GPUs.

1.1.2 Data-Parallel DNN Training

Data-parallel DNN training trains a single DNN using multiple training pipelines where each pipeline handles a different subset of data. As illustrated in Figure 1.2, each pipeline fetches a different subset of data from storage and prepossesses data independently. In the training stage, gradients are calculated by each pipeline and are reduced so that every pipeline has the same averaged gradients. The averaged gradients are used to update the model to make sure each pipeline has the same copy of model parameters.

Pipelines in data-parallel DNN training can run either on the same computing node using intra-node communication (single node multiple GPU training) or different nodes using inter-node communication (multiple node multiple GPU training). For the existing communication interfaces (e.g., MPI), the intra-node communication is usually more efficient than inter-node communication. Thus, it is preferred to allocate pipelines on the same computing node rather than on different nodes. As the existing frameworks (e.g. Tensorflow, Pytorch) do not allow different pipelines to concurrently use a single GPU, the number of GPUs available to train a DNN model practically limits the maximum number of pipelines that can be created in data-parallel DNN training.
Figure 1.2 An illustration of data-parallel DNN training [SB].
This chapter gives an overview of FLEET. FLEET is a flexible pipeline software architecture for efficient ensemble training of heterogeneous DNNs. It provides flexibility for configuring the scheduling of DNNs on nodes and GPUs via separation of preprocessing and training into different processes and a collection of communication schemes. It creates efficiency via heterogeneity-conscious runtime resource allocation and scheduling, plus sharing of preprocessing results among DNNs.

FLEET uses two types of processes, called preprocessor and trainer, to perform preprocessing and training separately. A trainer group contains one or more trainer processes and is responsible for training one DNN in the ensemble. A trainer process uses one GPU for training. When a trainer group contains more than one trainer process, they perform data-parallel DNN training for one DNN. Each trainer group has a trainer as the training group master and zero or more trainers as the training workers. The preprocessors communicate directly with only some master trainers, and those master trainers forward the preprocessed data to other trainers. Figure 2.1 illustrate the ensemble training pipeline in FLEET.

The job of each process is summarized as follows:

**Preprocessor**: fetch data from storage, preprocess the data, and send the preprocessed data to its paired training group master.

**Training Group Master**: receive the preprocessed data from its paired preprocessor, scatter it within its training group, broadcast the data to other training group masters, and train the DNN
Figure 2.1 An illustration of the ensemble training pipeline in FLEET. P1 and P2 are preprocessors and T1-T8 are trainers. There are four training groups, (T1), (T2, T3), (T4), (T5, T6, T7, T8), which train the four DNNs D1-D4 respectively. Edges indicate transfers of preprocessed images.

using the assigned batch of data.

Training Worker: receive the assigned batch of data from its training group master and use it to train the DNN.

Two important features of FLEET are its efficiency and flexibility.

The efficiency of FLEET comes from its novel resource allocation strategy developed for efficient ensemble training. The strategy is powered by some fundamental understanding of this resource allocation problem, and a greedy scheduling algorithm designed specifically to heterogeneous DNN ensemble training. The algorithm seamlessly integrates with the data-parallel distributed training that FLEET introduces into DNN ensemble training. As illustrated in Figure 2.1, different number of GPUs can be allocated to each DNN so that the DNNs can reach a similar training rate, avoiding the pipeline inefficiency caused by the slowest DNNs. It overcomes the NP-hardness of the resource allocation problem through a greedy design, grouping DNNs into multiple flotillas and periodically (re)allocate GPUs to remaining DNNs in a global efficient manner. It further leverages check-pointing to mitigate the issue of varying convergence speeds among DNNs. Together, FLEET is able to achieve efficient ensemble training while enabling data sharing to save CPU usage.

The flexibility of FLEET is in two aspects. First, decoupling preprocessing and training using different processes\(^1\) provides the flexibility in configuring the number of preprocessors such that the preprocessing throughput can match the trainers’ throughput without creating too many preprocessors that may waste computing resource and power. Second, as each trainer is associated with one GPU, resources for training can be allocated in the granularity of GPUs (rather than nodes as in prior work [Pit18]). Each GPU in a node can be assigned independently to DNNs. Each DNN

\(^1\)The reason we used processes instead of threads is due to the Global Interpreter Lock in Python. As FLEET is built on TensorFlow which is in Python, multi-processing brings maximum parallelism into the training pipeline.
in an ensemble can be trained using different numbers of GPUs concurrently, giving flexibility for handling the heterogeneity in DNNs.

**Two-fold Enabling Techniques** The key technical contributions that make FLEET possible are two-fold. The first is theoretical, consisting of a deep understanding of the resource allocation problem and some novel algorithms for assigning DNNs to GPUs. The second is empirical, consisting of a set of solutions to the various challenges for implementing FLEET above the array of complex software components (TensorFlow, Horovod, Python, MPI, etc.) on a heterogeneous Multi-GPU supercomputer like Summit-dev. We present the two-fold contributions in the next two sections respectively.
CHAPTER

3

PROBLEM DEFINITION

The design of FLEET is based on some fundamental understanding of the efficient ensemble training problem. We formalize it as an optimal resource allocation problem and analyze its computational complexity; the understanding motivates our later designs of the practical algorithms and the FLEET architecture. We next start with a formal analysis of the problem.

Efficient ensemble training is essentially an optimal resource allocation problem. The resources involve CPUs and GPUs in the modern high performance heterogeneous computing clusters. Under the context of data sharing, an optimal CPU allocation sets the number of preprocessors to be the one that just meets the computing requirement of training DNNs. GPU allocation, however, is much more complex and determines the pipeline efficiency[18]. So the optimal resource allocation problem is essentially to determine the optimal assignment of GPUs to each DNN in order to obtain the maximum pipeline efficiency.

Let \( D \) be a DNN ensemble that has \( N \) DNNs, \( D = [D_1, \cdots, D_N] \), and \( M \) be the number of GPUs in a cluster. When \( N > M \), the DNNs may have to be trained in multiple rounds, with later rounds reusing the GPUs released by earlier rounds. With data-parallel training allowed, even when \( N < M \), multiple rounds could be still needed as one DNN could occupy multiple GPUs.

There are two possible paradigms for scheduling DNNs on GPUs. A local paradigm assigns a DNN to a GPU immediately when the GPU becomes vacant. A global paradigm periodically examines the remaining DNNs and does a global (re)assignment of the DNNs to all GPUs. The local paradigm is relatively easy to understand; the global paradigm has the potential to avoid the local
optimal but is more difficult to design. Particularly, to effectively realize the global paradigm, several open questions must be answered: Is an optimal scheduling algorithm feasible? If so, what is it? If not, how to efficiently approximate it? This section hence focuses on the global paradigm, and explores these open questions. For easy reference, we put into Table 3.1 the important notations used in the rest of this report.

In this scheduling problem, the entire execution trains $N$ DNNs on $M$ GPUs in $K$ rounds. The beginning of a round is the time for globally (re)scheduling remaining DNNs on GPUs. The set of DNNs being trained in each round is called a flotilla. So there are $K$ flotillas being trained in the execution, one flotilla a round.

Theoretically, a round can be a time period of an arbitrary length. We first focus the problem setting to a simple case where a round finishes when and only when the training of all the DNNs in a flotilla finishes (e.g., converges or the maximum training epochs reached). In this setting, the GPUs that are done with its work in the current flotilla earlier than other GPUs would have some idle waiting time. The simplicity of this setting, however, makes the analysis easy to understand. We will briefly discuss the complexities of the more general settings at the end of the next subsection.

We now give a formal definition of the resource allocation problem in the focused setting. Each DNN in the ensemble are placed into at least one of the flotillas $\mathcal{F}_k \subseteq \mathcal{D}$, $k = 1, \cdots, K$ such that a list of $K$ flotillas $\mathcal{F} = [\mathcal{F}_1, \cdots, \mathcal{F}_K]$ satisfies the following requirements:

- The flotillas cover all the DNNs, $\cup_{k=1}^{K} \mathcal{F}_k = \mathcal{D}$.
- The flotillas do not have overlap\(^1\).
- Each flotilla, $G_k = [D_{1(k)}, \cdots, D_{N_k(k)}]$, contains no more than $M$ DNNs (i.e., $N_k \leq M$) such that the DNNs within each flotilla can be trained concurrently and thus the preprocessing can be shared during the training of these DNNs.

\(^1\)In the later section, we remove this requirement to address the issue of varying convergence speed.

---

**Table 3.1 Notations.**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>the number of DNNs in an ensemble.</td>
</tr>
<tr>
<td>$M$</td>
<td>the number of GPUs available in a cluster.</td>
</tr>
<tr>
<td>$K$</td>
<td>the number of DNN flotillas.</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>the list of DNNs in an ensemble, $\mathcal{D} = [D_1, \cdots, D_N]$.</td>
</tr>
<tr>
<td>$\mathcal{F}$</td>
<td>the list of flotillas of DNNs, $\mathcal{F} = [\mathcal{F}_1, \cdots, \mathcal{F}_K]$.</td>
</tr>
<tr>
<td>$\mathcal{F}_k$</td>
<td>the $k$-th flotilla of DNNs, $\mathcal{F}<em>k = [D</em>{1(k)}, \cdots, D_{N_k(k)}]$.</td>
</tr>
<tr>
<td>$\mathcal{A}$</td>
<td>the list of GPU allocations corresponding to $\mathcal{F}$, $\mathcal{A} = [A_1, \cdots, A_K]$.</td>
</tr>
<tr>
<td>$\mathcal{A}_k$</td>
<td>the GPU allocations for the $k$-th flotilla of DNNs, $\mathcal{A}_k$ is an $N_k$-by-$M$ matrix.</td>
</tr>
<tr>
<td>$a_{i,j}^{(k)}$</td>
<td>whether the $j$-th GPU is assigned to the $i$-th DNN in the $k$-th flotilla (i.e., $D_{i(k)}$).</td>
</tr>
<tr>
<td>$r_i^{(m)}$</td>
<td>the training rate of the $i$-th model in the $k$-th flotilla trained with $m$ GPUs.</td>
</tr>
</tbody>
</table>
Let $\mathcal{A} = [A_1, \cdots, A_K]$ be the GPU assignment for the $K$ flotillas of DNNs. Each assignment $A_k$ is a $N_k$-by-$M$ matrix $(a_{i,j}^{(k)})$ with:

$$a_{i,j}^{(k)} = \begin{cases} 
1, & \text{if the } j\text{-th GPU is assigned to the model } D_i^{(k)} \\
0, & \text{otherwise}
\end{cases}$$

(3.1)

subject to

$$\sum_{i=1}^{N_k} a_{i,j}^{(k)} \leq 1 \quad (j = 1, 2, \cdots, M)$$

(3.2)

$$\sum_{j=1}^{M} a_{i,j}^{(k)} \geq 1 \quad (i = 1, 2, \cdots, N_k)$$

(3.3)

Given the assignment matrix $A_k$, $m_i^{(k)} = \sum_{j=1}^{M} a_{i,j}^{(k)}$ is the number of GPUs assigned to the $i$-th DNN in the $k$-th flotilla. The notations are listed in Table 3.1.

An optimal resource allocation is an allocation strategy of available GPUs in a cluster to DNNs in an ensemble such that the end-to-end training time of the DNNs is minimized. The definition is as follows:

**Definition 1 Optimal Resource Allocation Problem.** Given a DNN ensemble $\mathcal{D}$ and a cluster of nodes with totally $M$ GPUs, let $T(\mathcal{D}|\mathcal{F}, \mathcal{A})$ be the end-to-end time (also called make-span) to finish the training of all the DNNs according to the list $\mathcal{F}$ and the corresponding GPU assignment $\mathcal{A}$. The optimal resource allocation problem is to find a schedule $(K^*, \mathcal{F}^*, \mathcal{A}^*)$ such that

$$K^*, \mathcal{F}^*, \mathcal{A}^* = \arg \min_{K, \mathcal{F}, \mathcal{A}} T(\mathcal{D}|\mathcal{F}, \mathcal{A})$$

(3.4)

$$= \arg \min_{K, \mathcal{F}, \mathcal{A}} \sum_{k=1}^{K} T(\mathcal{F}_k|A_k),$$

(3.5)

where, $T(\mathcal{F}_k|A_k)$ is the time spent on training the DNNs in the $k$-th flotilla with the assignment $A_k$ for a certain number of epochs.

### 3.1 Complexity Analysis

In this part, we argue that the Optimal Resource Allocation Problem is NP-hard in general. The argument comes from the classic results in Parallel Task System Scheduling. As Du and Leung have proved [DL89], finding an optimal nonpreemptive schedule for a Parallel Task System with the precedence constraints consisting of chains is strongly NP-hard for each $n > 2$ ($n$ is the number of processors). And, when the precedence constraints are empty, the problem is strongly NP-hard for each $n \geq 5$. The optimal resource allocation problem can be viewed as a parallel task system.
scheduling problem with each DNN as a task and each GPU as a parallel processor. One subtle aspect is that even though the DNNs are independent, to leverage shared preprocessing data among DNNs, a newly freed GPU does not take on a new DNN until the new round starts. It could be viewed as there are some pseudo precedence constraints between the DNNs in two adjacent rounds. So in general, the optimal solution is unlikely to be found in polynomial time. Recall that our discussion has been assuming that a new round starts only when the training of the DNNs in the previous round is all done. If the condition is relaxed such that a round can be a time period of an arbitrary length, the problem becomes even more complex to solve.
Motivated by the complexity in finding optimal solutions to the problem, we have designed a greedy algorithm for FLEET to assign DNNs to GPUs efficiently. It is worth noting that even though the Resource Allocation Problem connects with the classic Parallel Task System Scheduling problem, several special aspects of it make it unique and deserve new algorithm designs. First, unlike what is often assumed in the classic scheduling problem, the length of a task (DNN training) is hard if ever possible to predict: There is no known method that can accurately predict the number of epochs (and hence the time) needed for a DNN training to converge. Second, the relations among tasks (DNNs) are "fluid". The training processes of two DNNs are theoretically independent: One does not depend on another's data or control. But when they are put into the same flotilla, they would become related: They would share the same preprocessed data and hence would need to keep a similar progressing pace. These special aspects make the problem different from prior problems and call for new algorithms to be designed.

This chapter describes our algorithm. It first introduces several principles we have followed in developing the solution, and then elaborates our greedy algorithm. Note that the solution provided in this section is for the general problem rather than the restricted settings in the previous subsection. We will explain the solution in the context of the global paradigm, but will discuss how it is also applicable to the local paradigm at the end of this section.
4.1 Principles

A resource allocation strategy involves both the grouping the DNNs into flotillas and assigning the DNNs in each flotilla to the GPUs. We develop our solution by following four principles. The core of these principles is to organize tasks with less variation and dependencies at the flotilla level (Principle 1 and 2) and at the node level (Principle 3 and 4).

**Principle 1** DNNs in the same flotilla should be able to reach a similar training rate (e.g., images per sec) by assigning a proper number of GPUs to each of the DNNs.

This principle helps ensure a balanced pace of all GPUs, which help the DNNs in consuming the shared preprocessed data in a similar rate to minimize the waiting time of certain GPUs.

**Principle 2** Packing into one flotilla as many DNNs as possible.

The reason for this principle is two-fold. First, the throughput of multi-GPU training scales sublinearly with the number of GPUs due to the communication overhead of exchanging gradients. The principle is to help maintain good efficiency of the DNNs. Second, it allows more DNNs to share preprocessed data.

**Principle 3** When assigning multiple GPUs to a DNN, try to use GPUs in the same node.

This principle is to reduce the variation in communication latency: inter-node communications are slower and have more variations than intra-node communications.

**Principle 4** Try to assign DNNs that need a small number of GPUs to the same node.

This principle is similar to Principle 2 but at the node level. The rationale is that, although it is hard to reduce the communication overhead of DNNs that need to be trained using multiple nodes, we can minimize the communication overhead of DNNs that need a small number of GPUs by assigning them to the GPUs in the same node.

Based on the above principles, we propose a greedy algorithm to solve the resource allocation problem, as described below.

4.2 Algorithm

The greedy algorithm is shown in Algorithm 1. It uses training rates of the DNNs, \( R = \{r_i(m)\}, \) \( i = 1, \ldots, N, \) \( m = 1, \ldots, M, \) which are attained through a short profiling process (line 1). We propose profiling of fewer than 50 batches of training for each DNN, which also can piggyback the conventional checkpoints to training a DNN model. Typically, a DNN model training checkpoints to recover
Algorithm 1 Greedy Algorithm

Input: \( D, M \)
Output: \( \mathcal{F}, \mathcal{A} \)

1: \( R = \text{profile}(D) \) // Profile training rates of each DNN trained using \( m = 1, \ldots, M \) number of GPUs.
2: \( \mathcal{F}, \mathcal{A}, \text{cands}, k = [], [], D, 1 \)
3: while \(|c| > 0\) do
4: \( \mathcal{F}_k, \mathbf{m}_k = \text{createFlotilla}(\text{cands}, R, M) \) // Step 1: create a new flotilla from candidate DNNs, returns the flotilla of DNNs (\( \mathcal{F}_k \)), GPU count vector (\( \mathbf{m}_k \))
5: \( A_k = \text{getGPUAssignment}(\mathcal{F}_k, \mathbf{m}_k) \) // Step 2: find GPU assignment for the flotilla
6: \( \text{dels} = \text{train}((\mathcal{F}_k, A_k) \) // Step 3: Load latest checkpoint, loss history, and epoch count if available; train DNNs in the flotilla for a number of Epochs; return converged models (\( \text{dels} \)).
7: \( \text{cands} = \text{dels} \) // Remove converged models from candidates (\text{cands})
8: \( \mathcal{F}.\text{append}(\mathcal{F}_k); \mathcal{A}.\text{append}(A_k); k+ = 1 \)
9: end while

from system faults since a DNN training usually requires at least thousands of steps that may take multiple hours or days. We defer the detailed profiling process to Chapter 5.

The greedy algorithm dynamically determines the grouping of the DNNs in an ensemble based on the training status of each DNN in the ensemble (i.e., whether the DNN is converged or not) and the training rate of each DNN. Once a flotilla is created, an optimal GPU assignment can be derived. Initially, all DNNs are considered as candidates (\text{cands}) when a new flotilla needs to be created (line 2). The greedy algorithm then iterates over three main steps, flotilla creation (line 4), GPU allocation (line 5), and training (line 6), until all the DNNs in the ensemble are converged.

In the first step, a set of DNNs is selected from the candidates to form a flotilla such that the DNNs in the flotilla are able to achieve similar training rate by assigning proper amount of GPUs, following Principles 1 and 2. The flotilla creation process produces a DNN flotilla as well as the GPU count vector that specifies the number of GPUs to assign to each DNN in the flotilla. Then the GPUs are allocated to DNNs in the flotilla based on the GPU count vector, following Principles 3 and 4. The third step trains the DNNs in the flotilla on assigned GPUs for a number of epochs and reports the DNNs that are converged during the training. These converged DNNs are removed from the candidate list (line 7). The above three main steps are repeated if there are still DNNs that are not converged (i.e., \text{cands} are not empty).

We next describe the three steps in detail.
4.3 Flotilla Creation

This first step selects a set of DNNs from candidates to create a new flotilla whose DNNs are trained concurrently with data sharing. Algorithm 2 shows the flotilla creation algorithm.

The algorithm first identifies the largest training rate with a single GPU, $r_{fast} = \max\{r_1(1), \ldots, r_{cand}(1)\}$, and the corresponding DNN, $D_{fast}$, from the candidate set of DNNs (line 1). Then, $r_{fast}$ is used as the reference training rate to search for other DNNs that can be placed in the same flotilla. Mathematically, the algorithm searches for the next DNN that can be placed into the flotilla by solving the following optimization problem (lines 4-10):

$$\min_{D_i \in cand, \ D_i \notin F_k, \ m=1, \ldots, M} |r_i(m) - r_{fast}|,$$

subject to

$$|r_i(m) - r_{fast}| \leq \delta,$$

$$M_k + m \leq M.$$  \hspace{1cm} (4.1)

where $\delta$ is the threshold that determines if two training rates are close, and $M_k$ is the total number of GPUs that are already assigned to DNNs. In our experiments, $\delta$ is set to 20 (images/sec). The algorithm stops adding DNNs to a flotilla if no solution exists to above equation.

After a flotilla is formed, if there are still GPUs available, we assign the next GPU to the DNN in the flotilla that has the smallest training rate iteratively until all the GPUs are assigned (lines 12 - 16). The DNN with the smallest training rate determines the pipeline efficiency. Assigning extra GPUs to the slowest DNN is able to improve pipeline efficiency.

The flotilla creation step produces a flotilla of DNNs as well as the GPU count vector that specifies the number of GPUs assigned to each DNN. We next explain how to properly assign GPUs to each DNN based on the GPU count vector and considering GPU locality.

4.4 GPU Assignment

This procedure assigns GPUs to DNNs in a flotilla. The goal of this procedure is to allocate a smaller number of nodes to each DNN in a flotilla so that we reduce communication latency variation from inter-node communication (Principle 3), while assigning DNN models in a flotilla into all available GPUs in the system (Principle 4). In other words, we want to find an assignment $A_k$ to minimize the number of nodes involved to train each DNN. Let $c(.)$ be the function that counts the number of nodes involved in training a DNN given its GPU assignment $a_{i}^{(k)}$, which is the $i$-th row of the
Algorithm 2 createFlotilla

Input: \textit{cands, R, M}
Output: \( \mathcal{F}_k, m_k \)

1: \( D_{fast}, r_{fast} = \text{fastestDNN}(\textit{cands, R}) \) // Find the DNN with the largest training rate with a single GPU
2: \( \mathcal{F}_k, M_k, m_k = [D_{fast}], 1, [1] \)
3: \textbf{while} \( |\mathcal{F}_k| < |\textit{cands}| \) \textbf{do}
4: \( D_{best}, r_{best}, M_{best} = \text{findNext}(r_{fast}, R, \textit{cands, F}_k, M - M_k) \) // Find the next DNN, its training rate and required GPU count
5: \textbf{if} \( D_{best} == -1 \) \textbf{then}
6: \hspace{1em} \textbf{break}
7: \textbf{end if}
8: \( \mathcal{F}_k.\text{append}(D_{best}) \)
9: \( m_k.\text{append}(M_{best}) \)
10: \( M_k += M_{best} \)
11: \textbf{end while}
12: \textbf{while} \( M_k < M \) \textbf{do}
13: \( D_{slow} = \text{slowestDNN}(\mathcal{F}_k, m_k, R) \) // in terms of speed on the currently assigned GPUs
14: \( m_k[slow]++ = 1 \)
15: \( M_k += 1 \)
16: \textbf{end while}
17: \textbf{return} \( \mathcal{F}_k, m_k \)
assignment matrix $A_k$, the GPU assignment is an optimization problem:

$$
\begin{align*}
\min_{A_k} & \sum_{i=1}^{N_k} \frac{c(a^{(k)}_i)}{m^{(k)}_i}, \\
\text{s.t.} & \sum_{j=1}^{M} a^{(k)}_{i,j} = m^{(k)}_i, \quad i = 1, \ldots, N,
\end{align*}
\quad (4.2)
$$

where $\frac{c(a^{(k)}_i)}{m^{(k)}_i}$ is the number of nodes involved to train the $i$-th DNN, scaled by $m^{(k)}_i$, the number of GPUs assigned. The solution space is as large as $M! \prod_{i=1}^{N_k} (m^{(k)}_i)!$.

Instead of exhaustively searching for an optimal solution in the reduced space, we propose a greedy approach that assigns GPUs to each DNN in an incremental fashion. For example, if the $j$-th GPU is already assigned to a DNN, then the next GPU to be assigned to the DNN is the $j+1$-th GPU. The solution space is reduced to the space of possible permutations of the GPU count vector, which contains $N_k!$ solutions.

This algorithm assumes the number of GPUs per node is the same among nodes (GPUsPerNode), which holds in the major supercomputers. It assigns GPUs to DNNs in the following order:

1. allocate GPUs to the DNNs whose required number of GPUs is a multiple of the number of GPUs per node.
2. allocate GPUs to the pairs of DNNs whose sum of the required number of GPUs is a multiple of the number of GPUs per node.
3. allocate GPUs to the remaining DNNs by searching for an optimal assignment of GPUs.

This step assigns GPUs to DNNs in a group. The algorithm is shown in Algorithm 3.

The flotilla creation and GPU assignment steps ensure that DNNs in the same flotilla is able to achieve similar training rate to improve GPU utilization. We next describe how the training step addresses the varying convergence speed issue via check-pointing.

### 4.5 Training

The training step trains the DNNs on their assigned GPUs concurrently with data sharing. Due to the architectural difference of DNNs in a heterogeneous ensemble, these DNNs require a different number of epochs to converge. With data sharing, converged models need to wait for the un-converged models to complete, leading to the waste of computing resources.

We leverage check-pointing to address the varying convergence speed issue. Specifically, each flotilla is trained until only $\alpha \cdot M$ GPUs remain active for training, where $\alpha$ is a non-negative fraction
Algorithm 3 getGPUAssignment

Input: $\mathcal{F}_k, m_k$
Output: $A_k$

1: $j, A_k, remaining, assigned = 1, 0_{N_k,M}, \{1, \ldots, N\}, \{\}$
2: for all $i \in remaining$ do 
3: if $m_i^{(k)} \% \text{GPUsPerNode} == 0$ then
4: assigned.add($i$)
5: $j = \text{assignGPUs}(A_k, i, j, m_i^{(k)})$
6: end if
7: end for
8: remaining -= assigned
9: memo, assigned = {}, {}
10: for all $i \in remaining$ do
11: if $-m_i^{(k)} \% \text{GPUsPerNode}$ not in memo then
12: memo$[-m_i^{(k)} \% \text{GPUsPerNode}] = i$
13: else
14: for $ii = i, memo[-m_i^{(k)} \% \text{GPUsPerNode}]$ do
15: assigned.add($ii$)
16: $j = \text{assignGPUs}(A_k, ii, j, m_i^{(k)})$
17: end for
18: del memo$[-m_i^{(k)} \% \text{GPUsPerNode}]$
19: end if
20: end for
21: remaining -= assigned
22: if $|remaining| > 0$ then
23: $\tilde{m}_k, bestScore, bestA, j_copy, A_copy = [], \infty, j, \text{clone}(A_k)$
24: for all $i \in remaining$ do
25: $\tilde{m}_k$.append($[i, m_i^{(k)}]$)
26: end for
27: for permutation in allPermutations($\tilde{m}_k$) do
28: $j, A_k = j_copy, \text{clone}(A_copy)$
29: for $i, m_i^{(k)}$ in permutation do
30: $j = \text{assignGPUs}(A_k, i, j, m_i^{(k)})$
31: end for
32: score = calculateScore($A_k$) // Score is calculated based on the loss function in Eq. 4.2
33: if score < bestScore then
34: bestScore, best$A = score, A_k$
35: end if
36: end for
37: $A_k = bestA$
38: end if
39: return $A_k$
Algorithm 4 assignGPUs

Input: $A_k, i, j, m^{(k)}_i$
Output: $j$

1: while $m^{(k)}_i > 0$ do
2: $a_{i,j}^k = 1; \ j += 1; \ m^{(k)}_i -= 1$
3: end while
4: return $j$

(set to 0.8 in all our experiments). This is achieved by monitoring whether a model is converged at the end of each epoch. Once a model is converged, it is marked as complete and its GPUs are released. If the total number of GPUs that are not released falls below $\alpha \cdot M$, the training of all the DNNs in the flotilla stops. The parameters, loss history, and epoch count of all the DNNs are checkpointed. The DNNs in the next flotilla will be initialized by its checkpoint if it is available. A DNN that is marked as complete will not be packed into any of the following flotillas.

4.6 Complexity Analysis

Flotilla creation first searches for the reference training rate (time complexity is $\mathcal{O}(N)$), then iteratively finds the best candidate DNN to add in the flotilla (time complexity is $\mathcal{O}(N_k \times N \times M)$), and finally assigns all the remaining GPUs available to the DNNs in the flotilla (time complexity is $\mathcal{O}(N_k \times M)$). So the time complexity of flotilla creation is $\mathcal{O}(N_k \times N \times M)$.

GPU assignment first prunes the factorial solution space by identifying and assigning GPUs to the DNNs whose training rate meets certain requirements in $\mathcal{O}(N_k)$ time complexity. It then searches for the optimal GPU assignment strategy for the remaining DNNs. Let $N'_k$ be the number of remaining DNNs. The solution space is $N'_k!$. Most of the time, $N'_k$ is a small number less than five. However, enumerating all the possible solutions is still in factorial time complexity. We set the maximum number of solutions to explore as 1024, reducing the time complexity to $O(1)$. The time complexity of GPU assignment is thus $\mathcal{O}(N'_k)$.

4.7 Application in the Local Paradigm

Although the discussion has been assuming the global paradigm, the greedy algorithm is applicable to the local paradigm of resource allocation as well. The training proceeds as follows:

(1) At the beginning, the algorithm forms the first flotilla of DNNs in the same way as Algorithm 2, and starts training them.

(2) Whenever a DNN is done, the algorithm fills the released GPUs with new DNNs just as
Algorithm 2 does, except that $M$ is replaced with the GPUs that are currently available. If no DNN remains untrained, terminate when all current training is done.
This chapter describes the challenges and the implementation details of FLEET. The implementation of FLEET faces the following three major challenges:

- The greedy algorithm uses the training rates of DNNs in an ensemble with different number of GPUs, which are attained through a short profiling process. How do we design the profiling process that introduces minimal overhead?

- Recall that FLEET has two types of processes, preprocessor and trainer. The number of preprocessors needs to be set to meet the requirement of trainers’ throughput. Thus, it is necessary for FLEET to support creating different number of processes per rank on a cluster. How do we support the throughput, while enabling flexible communications between preprocessors and trainers?

- With data-parallel DNN training, preprocessed data from a processor is received by its paired training group master, scattered to trainers within the group (including the training group master), and broadcasted to the other training group masters. How do we build an efficient dataflow to enable efficient training pipeline?

We next describe the solutions and the implementation details for these challenges.
5.1 Profiling

To minimize the overhead of profiling, we only profile the training rates of each DNN in the ensemble with the number of GPUs varying from one to $M_t (M_t < M)$. For $m = 1, \cdots, M_t$, we train a DNN for a maximum of 48 batches and use the training time of the last 20 batches to calculate the exact training rate: $r_i(m)$, $i = 1, \cdots, N$. Based on the profiled training rates, we estimate the training rates of each DNN when $m > M_t$. Specifically, the profiling has three steps:

1. Collect the training rates of each DNN on a single GPU, $R(1) = \{r_i(1)\}$, $i = 1, \cdots, N$.

2. Estimate the number of GPUs required to make the DNN that has the smallest training rate on a single GPU achieve the largest single-GPU training rate, $M_a = \lceil \frac{\max(R(1))}{\min(R(1))} \rceil$.

3. Collect the training rates of each DNN with the number of GPUs varying from two to $M_t = \max(M_a, M_b)$, where $M_b = 2 \times \text{GPUs\ Per\ Node}$.

Note that steps 1 and 3 can be done in parallel because the trainings of different DNNs with different number of GPUs are independent. The training rate of the $i$-th DNN with the number of GPUs higher than $M_t$ is estimated via the following equation:

$$r_i(m) = r_i(M_b) \times \left( \frac{r_i(M_b)}{r_i(M_b - 1)} \right)^{m-M_b}, m = M_t + 1, \cdots, M. \tag{5.1}$$

The formula for $M_b$ and Equation 5.1 are the result of performance modeling on our observations.

Figure 5.1 The profiled training rates (images/sec) of 100 DNNs in an ensemble.
on the DNN performance trend as illustrated in Figure 5.1. It achieves a good tradeoff between the profiling cost and the performance prediction accuracy.

The profiling process also measures the throughput of a range of preprocessors (# cores=1, 2, 4, 8, 16, 32) in the pipeline. This step is quick since preprocessing does not exhibit large variations. Based on the profiled information, FLEET calculates the minimum number of preprocessors that can meet the demands of an arbitrary $M$ DNNs (with one running on one GPU), and uses it to set the number of preprocessors.

5.2 Communications between Preprocessors and Trainers

To support creating different number of processes per rank, a preprocessor process is created through the fork operation optionally activated in a trainer group master. The number of preprocessors can be controlled by choosing how many trainer group masters execute the fork operation.

A preprocessor is a child process of a training group master and thus is not able to use MPI functions. We establish the communications between a preprocessor and its paired trainer group master through server process. A server process holds python objects and allows other processes to manipulate them using proxies. A preprocessor sends the processed data to its training group master by writing to a numpy object using the object’s proxy.

5.3 Dataflow Implementation

Figure 5.2 illustrates the dataflow implementation in FLEET. The dataflow contains the following three pipelined steps:

1. Training group masters receive preprocessed data from their paired preprocessor and put the data into a preprocessed queue $Q_p$.

2. Preprocessed data from $Q_p$ are broadcast to all the training group masters through MPI. Each training group master receives all the preprocessed data, but handle the data differently, depending on whether data-parallel training is used. If a training group contains only one trainer (i.e., only one GPU is used to train a DNN), the training group master puts all the data into its trainer queue $Q_r$. Otherwise, the training group master scatters the data to its trainer queue $Q_r$ and the distribution queue $Q_{D^*}$. The data in the distribution queue is sent to the trainer queue $Q_r$ of each training group worker via MPI point-to-point communication in a separate thread.

3. For each trainer, the DNN training queue $Q_D$ reads preprocessed data from the trainer queue $Q_r$ and feeds the data to the DNN model for training. Data-parallel training of a DNN employs Horovod [SB] for gradient synchronizations.
Figure 5.2 Illustration of the dataflow implementation. Two DNNs, $D_1$ and $D_2$, are trained using four GPUs (Ranks 0–3) by two training groups, $(T_1)$ and $(T_2, T_3, T_4)$. $T_1$ and $T_2$ are training group masters.
We conduct a set of experiments to examine the efficacy of FLEET by answering the following questions:

1. How much speedup can FLEET bring to ensemble training of heterogeneous DNNs?

2. How do the pros and cons of the two paradigms in FLEET designs, local and global, play out in handling the variations among DNNs?

3. More specifically, does the greedy scheduling algorithm in FLEET produce favorable schedules? How much waiting time does the round-by-round scheme in FLEET cause, compared to eager scheduling schemes?

4. What is the overhead of runtime profiling, scheduling, and checkpointing in FLEET?

6.1 Experiment Settings

DNNs. The DNNs used in this experiment are derived from six popular DNNs, DenseNet-121, DenseNet-169, DenseNet-201, ResNet-50, ResNet-101 and ResNet-152. The first three are variations of DenseNet [Hua17]. The three variations share the same structure, but differ in the number of DNN layers, indicated by their suffixes. The latter three are variations of ResNet [He16]. Both DenseNet and ResNet are the state-of-the-art network architectures that achieve high performance in various
learning tasks. We select these networks as the basis because, as structural DNNs, they are composed of many Convolutional blocks, which have a standard interface making a block ready to be connected with any other blocks. As a result, it is easy to derive new DNNs from them—one just needs to remove or insert some Convolutional blocks.

Based on the public DNNs, we derive 100 experimental DNNs (50 from DenseNet and 50 from ResNet) by randomly adding and removing blocks from them. Different DNNs have different GPU memory requirements and thus require different batch sizes to maximize GPU utilization. We use the maximum batch size that can fit into GPU's memory. Section 6.2.1 provides more characteristics of the 100 experimental DNNs.

FLEET is built on Tensorflow 1.12 (as the core training engine), Horovod v0.15.2 [SB] (as the basis for distributed DNN training), and mpi4py v3.0.0 (for the pipeline construction).

System All experiments are conducted on SummitDev [Sum], a development machine for Summit supercomputer at Oak Ridge National Lab. Each node is equipped with two IBM POWER8 CPUs and 256GB DRAM, and four NVIDIA Tesla P100 GPUs. Each POWER8 CPU has 10 cores with 8 HW threads each. The default SMT level is set to one unless noted otherwise. The number of cores allocated per GPU is five in all the experiments. NVLink 1.0 is the connection among all GPUs and between CPUs and GPUs within a node. EDR InfiniBand connects different nodes in a full fat-tree. The file system is an IBM Spectrum Scale file system, which provides 2.5 TB/s for sequential I/O and 2.2 TB/s for random I/O. Our experiments show that thanks to the large I/O throughput of the file system, I/O is not the bottleneck of DNN training.

Datasets. The datasets used in the experiments are ImageNet [Den09] and Caltech256 Object Category Dataset [Cal]. ImageNet contains 1,261,406 training images and Caltech256 contains 30,606 training images. We use ImageNet whenever possible (e.g., for throughput comparisons), but use Caltech256 for the measurement of end-to-end ensemble training times such that the training can converge within the maximum 240 minutes limit that SummitDev permits.

Counterparts for Comparisons.

- **Baseline** The baseline uses the default TensorFlow to train each DNN on one GPU independently. A GPU randomly picks one yet-to-be-trained DNN whenever it becomes free until there are no DNN left.

- **Homogeneous Training** This is the state-of-the-art framework recently published [Pit18] for ensemble DNN training. This framework allows the DNNs that get trained at the same time to share the preprocessed data. But it is designed for homogeneous DNN training, considering no variations among DNNs or the situation where the number of DNNs are no greater than the

\[\text{inter_op_parallelism_threads}\] and \[\text{intra_op_parallelism_threads}\] to # logical cores for parallel TensorFlow operations on CPU.
number of GPUs. In our experiments, when there are more DNNs than GPUs, the framework randomly picks a subset of the remaining DNNs to train, one DNN per GPU with shared preprocessed data. After that subset is done, it picks another subset and repeats the process until all DNNs are done.

- **FLEET-G** This is FLEET in the global paradigm.
- **FLEET-L** This is FLEET in the local paradigm as described in Section 4.7. Its difference from FLEET-G is that as soon as a DNN is done, the released GPUs are immediately used to train some remaining DNNs; which DNNs are picked is determined by the greedy algorithm as in FLEET-G, but only locally (for the newly released GPUs) rather than globally.

### 6.2 Measurements and Results

#### 6.2.1 Characteristics of Experimental DNNs

The 100 DNNs used in our experiments have a range of model sizes, from 232 MB to 1.19GB. Figure 6.1 shows the distribution of their training rates on a single GPU which vary from 21 to 176 images/sec.

Figure 6.2 outlines the relations between the training rates and model sizes of the DNNs, as well as the relations between convergence rates (i.e., the number of epochs needed for the DNNs to converge) and their model sizes. As model size increases, the training rate tends to drop as more computations are involved in the DNN, but there are no clear correlations with the convergence rate. It is the reason that the resource allocation algorithm in FLEET primarily considers training rate explicitly, while relies on the periodical (re)scheduling to indirectly adapt to the variations of DNNs in the converging rates.
Figure 6.2 Correlations between model size of a DNN and the training rate and the number of epochs till converge.

Figure 6.3 The averaged speedups over the baseline in terms of the end-to-end time for training a 100-DNN ensemble. The error bars show the variations.
6.2.2 End-to-End Speedups

Figure 6.3 reports the speedups of the three methods over the baseline method, in terms of the end-to-end ensemble training time of the 100 DNNs. All runtime overhead for FLEET is included. We repeat each measurement multiple times and report the average and error bars.

It shows the results in eight settings. The prior homogeneous framework shows large slowdowns in the first four settings where the number of GPUs is less than the number of DNNs. The slowdowns are due to the waiting of other GPUs for the slowest DNN to finish in each round, shown in Figure 6.4. In the other four settings, the homogeneous framework performs similarly as the baseline does: As there are more GPUs than DNNS, there is only one round, in which, the two methods use resource similarly. The sharing of preprocessing in the homogeneous framework does not generate speedups for these DNN trainings because the preprocessing is not the bottleneck for them.

FLEET-G gives the best overall performance, producing 1.13x-1.92x speedups over the baseline. The primary reason for the speedups come from its better resource allocation to the DNNs. The bottom of Table 6.1 reports the mean and standard deviations of the running lengths of DNNs in the first five flotillas in FLEET-G (80GPU,100DNN). In comparison to the data in the baseline and FLEET-L (top rows in Table 6.1), the DNNs show much smaller variations in length, which indicate the effectiveness of the GPU allocations in FLEET-G in evening out the differences among DNNs. At the beginning, we thought that the catch to FLEET-G is the waiting time of some GPUs after they are done with their work in a round. Our experiments however show the opposite effects. As Figure 6.4
Table 6.1 Mean and standard deviation of the running length of DNNs in seconds. (80 GPUs, 100 DNNs)

<table>
<thead>
<tr>
<th>Technique</th>
<th>Flotilla ID</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>-</td>
<td>10372.2</td>
<td>4178.87</td>
</tr>
<tr>
<td>FLEET-L</td>
<td>-</td>
<td>6213</td>
<td>3560.00</td>
</tr>
<tr>
<td>FLEET-G</td>
<td>0</td>
<td>2067.9</td>
<td>5.47E+01</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>335.6</td>
<td>4.84E+01</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2291.9</td>
<td>2.60E+02</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>415.5</td>
<td>5.19E+01</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1072.2</td>
<td>3.64E+02</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2322.3</td>
<td>2.16E+02</td>
</tr>
</tbody>
</table>

Table 6.2 Scheduling and checkpointing overhead.

<table>
<thead>
<tr>
<th>(#GPU, #DNN)</th>
<th>Total Training Time (in sec)</th>
<th>Scheduling Overhead</th>
<th>Checkpointing Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>(20,100)</td>
<td>55200.1</td>
<td>20.1 0.037</td>
<td>1496.0 2.7</td>
</tr>
<tr>
<td>(40,100)</td>
<td>30204.8</td>
<td>15.8 0.054</td>
<td>1156.0 3.8</td>
</tr>
<tr>
<td>(60,100)</td>
<td>24495.0</td>
<td>14.0 0.060</td>
<td>986.0 4.0</td>
</tr>
<tr>
<td>(80,100)</td>
<td>21891.0</td>
<td>12.0 0.057</td>
<td>816.0 3.7</td>
</tr>
<tr>
<td>(100,100)</td>
<td>18359.1</td>
<td>10.1 0.058</td>
<td>782.0 4.3</td>
</tr>
<tr>
<td>(120,100)</td>
<td>15323.9</td>
<td>9.9 0.068</td>
<td>680.0 4.4</td>
</tr>
<tr>
<td>(140,100)</td>
<td>13366.3</td>
<td>9.3 0.073</td>
<td>680.0 5.1</td>
</tr>
<tr>
<td>(160,100)</td>
<td>11825.2</td>
<td>10.2 0.092</td>
<td>748.0 6.3</td>
</tr>
</tbody>
</table>

shows, the average waiting time per GPU is smallest for FLEET-G. The reason is that the the other methods all suffer long waiting time at the end; because of their suboptimal resource allocation, some GPUs have to work long after others to finish up the last few DNNs. FLEET-L gives notable but less speedups for its less favorable decisions at resource allocation due to the local view.

6.2.3 Overhead

Table 6.2 reports the breakdown of the runtime overhead of FLEET-G. The overhead of scheduling and checkpointing is at most 0.1% and 6.3% of the end-to-end training time in all the settings. Recall that, due to wall-clock-time limitation of SummitDev, we have used the small Caltech256 dataset. For large datasets (e.g., ImageNet), the overhead would be quickly become negligible. The profiling overhead is independent of dataset size and solely depends on ensemble size. Recall that, profiling needs the DNN to train for only a few steps in parallel. Its overhead is marginal for typical DNN trainings on large datasets that take hours or days to train.
CHAPTER

7

RELATED WORK

Much research has been done to accelerate the training of a single DNN over distributed systems. DeepBelief and the recent Tensorflow from Google [Dea12; Aba16] and Project Adams from Microsoft [Chi14] are examples of this that support to train a DNN over thousands of machines. FireCaffe [Ian16] scales DNN training across a cluster of GPUs with Infiniband and Cray interconnects. Shrivastava et al. [Shr17] proposed a framework to support both data-parallel and model-parallel distributed training of DNNs over a cluster of CPUs in Apache Spark. Chung et al. [Chu17] designed a data-parallel algorithm that scales well on the IBM Blue Gene/Q computer system, which contains a large number of loosely coupled processors. Harlap et al. [Har18] proposed PipeDream, a DNN training system for efficient pipeline-parallel training of large DNNs. Huang et al. [Hua18] presented GPipe, a scalable pipeline parallelism library that enables learning of giant DNNs with thousands of machines.

All those studies have focused on improving the training speed of an individual DNN rather than ensemble training. Recently, ensemble training starts drawing more attention. Pittman et al. [Pit18] explored flexible communication strategies for training DNN ensembles and proposed data sharing to eliminate pipeline redundancy and improve pipeline efficiency. That work is designed for the DNNs that are homogeneous in architecture and hence computational needs. There are some other efforts [Gar18; LH16] on ensemble training, but they focus on designing lightweight methods to form high-performing ensembles instead of improving pipeline efficiency of ensemble training.

Another line of research that is relevant to this work is task scheduling on clusters. The scheduling
of a set of tasks or workloads on clusters or multiprocessor systems has been extensively studied in the literature [Tur92; Shm95; Urg02; Aug11; Zah08; Gra15; Cho16; Xu18]. For example, Shmoys et al. [Shm95] considered the problem of scheduling jobs on parallel machines when jobs come online and the processing requirement of a job is unknown when it starts processing. The problem shares some similarities with the problem addressed in this work because the completion time of training a DNN is unknown until the model converges. Turek et al. [Tur92] targets at a multiprocessor scheduling problem whose goal is to find the number of processors and a schedule assigned to each task to minimize the latest task completion task. Efficient heterogeneous DNN ensemble training with data sharing poses novel challenges as mentioned at the beginning of Section 4. To the best of our knowledge, the greedy algorithm inside FLEET is the first dynamic task scheduling algorithm that aims to improve heterogeneous DNN ensemble training efficiency on heterogeneous clusters.
This project presents a systematic exploration on enabling flexible efficient ensemble training for heterogeneous DNNs. It addresses two-fold challenges. First, it formalizes the essence of the problem into an optimal resource allocation problem, analyzes its computational complexity, and presents an efficient greedy algorithm to effectively map DNNs to GPUs on the fly. Second, it develops a set of techniques to seamlessly integrate distributed data-parallel training of DNN, preprocessing sharing, and runtime DNN-to-GPU assignments together into a software framework, FLEET. FLEET features flexible and efficient communications, and effective runtime resource allocations. Experiments on 100 heterogeneous DNNs on SummitDev demonstrate that FLEET can speed up the ensemble training by 1.12-1.92X over the default training method, and 1.23-1.97X over the state-of-the-art framework that was designed for homogeneous DNN ensemble training.
BIBLIOGRAPHY


