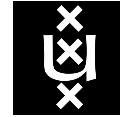
Vrije Universiteit Amsterdam

Universiteit van Amsterdam



Master Thesis

Modeling and Simulation of the Google TensorFlow Ecosystem

Author: Wenchen Lai (2643117)

1st supervisor: prof. dr. ir. Alexandru Iosup
2nd reader: dr. ir. Animesh Trivedi

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

November 2, 2020

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Recently, many powerful machine learning (ML) systems or ecosystems have been developed to render ML solutions feasible for more complex applications. Google TensorFlow ecosystem is one of the most famous and popular machine learning ecosystem. Because of some emerging technologies, such as big data, Internet of Things (IoT), high-performance computing (HPC), the power of datacenters are expected to be applied in Artificial Intelligence (AI) field. However, when performing ML tasks in datacenters, new challenges and issues arise, such as data management. Understanding the behaviors of the Google TensorFlow ecosystem is our main objective.

We adopt the reference architecture method and extend our reference architecture created in our literature survey. We add additional deeper layers and identify more than 10 new components to enrich our reference architecture. Based on the reference architecture, we create a predictive model of Google TensorFlow and integrate it into a discrete event simulator OpenDC. We design simulation experiments to validate our model and evaluate the performance of the TensorFlow ecosystem in HPC environments.

Contents

List of Figures iii

List of Tables v

1 Introduction 1

1.1 Context and Objective 1

1.2 Research Questions 2

1.3 Research Methods 3

1.4 Main Contributions 4

1.5 Outline 4

2 Background 5

2.1 A Reference Architecture for Datacenters 5

2.2 Simulators for Distributed Systems 9

2.3 Performance Modeling of Distributed DL Systems 11

2.4 Performance Evaluation and Analysis of Distributed DL Systems 13

3 Reference Architecture 15

3.1 Overview 15

3.2 Reference Architecture for TensorFlow Ecosystem 17

3.3 TensorFlow Core 21

3.4 Discussion 22

4 Modeling Google TensorFlow Ecosystem 25

4.1 Requirements 25

4.2 Overview 26

4.3 TensorFlow Model 28

4.4 Discussion 31

5 Experiments on the Performance of TensorFlow 35
5.1 Overview 35
5.2 Experiment Setup 36
5.3 Model Validation and Calibration 39
5.4 Results and Analysis for New Experiments 42
5.5 Discussion 46

6 Conclusion and Directions for Future Research 49
6.1 Conclusion 49
6.2 Directions for Future Research 50

References 53

List of Figures

2.1 A reference architecture for datacenters 6

3.1 Reference architecture for Google TensorFlow ecosystem. 16

4.1 Architecture of the model. 27
4.2 Domain model of a computational graph. 28
4.3 An example of dataflow computational graph. 29
4.4 Execution model. 30
4.5 Time diagram for communication tasks. 32
4.6 Sequence diagram: communication among entities. 33

5.1 KTH calibration and validation experiment. 40
5.2 IBM "Minsky" calibration and validation experiment. 42
5.3 Strategy comparison on the A1 workload. 43
5.4 Strategy comparison on the Fathom workloads. 43
5.5 Scalability study on the A1 workload. 45
5.6 Scalability study on the Fathom workload. 45

List of Tables

2.1 Simulators for distributed systems. 12

5.1 An overview of all experiments. 37
5.2 An overview of the workloads. 37
5.3 Types of computing devices. 38
5.4 An overview of HPC environments. 38
5.5 Time breakdown of KTH calibration and validation experiment. 40
5.6 Time breakdown of IBM calibration and validation experiment. 41
5.7 Total power consumption. 46
5.8 Total usage. 47

LIST OF TABLES

1

Introduction

1.1 Context and Objective

Recently, machine learning (ML) and deep learning (DL) have gained much attention due to their great potential in numerous areas like speech recognition [1], medical image analysis [2], product recommendations [3] and self-driving cars [4]. The goal of ML is to build a suitable model for prediction, classification, or decision making by learning from a (large) amount of data. To render ML solutions feasible for more complex applications, many powerful ML systems or ecosystems (such as TensorFlow [5], [6], PyTorch [7], and MXNet [8]) have been developed.

Google TensorFlow [5], [6] is one of the most prominent and representative ML/DL frameworks. Inspired by dataflow systems and parameter server architectures, TensorFlow aims to operate at a large scale and in heterogeneous environments.

The objective of this thesis is to understand the behaviors and performance of Google TensorFlow ecosystem.

Reference Architecture

What is a reference architecture? Muller [9] defines:

> A Reference Architecture captures the essence of the architecture of a collection of systems.

Reference architectures could provide guidelines for project managers, software developers, system architects to design a new system or extend an existing system. A reference architecture for Google TensorFlow ecosystem could help us to understand its behaviours.

There exist various AI reference architectures and they are designed for different proposes. Some target at specific domains, such as ACR AI-LAB hospital reference architecture [10] and a functional reference architecture for autonomous driving [11]. They usually include domain-specific elements, such as Electronic Medical Records(EMRs) systems in the hospital. Another kind of reference architectures is released by big corporations to help customers define, design, and deploy AI solutions. These reference architectures are used for their own AI systems, such as AI infrastructure reference architecture for IBM systems [12] and Microsoft Azure Reference Architectures. The reference architecture for datacenters of Iosup et al. [13], [14] is more general. It has been studied that it is well-designed for big data ecosystems and several well-known industry ecosystems [14]. In this project, we investigate it on TensorFlow ecosystems.

Predictive Model

Predictive model is a model that can predict the output of a system by analyzing data and patterns. Regarding the level of detail, there are three types of model: black-box(empirical) model, grey-box(hybrid) model, and white-box(mechanistic) model [15] [16]. A pure black-box model simply describes a system in terms of its inputs and outputs, without any knowledge of internal mechanisms [17]. It usually consists of a set of mathematical equations and rules. The black box model is efficient and useful when the purpose of the model is to represent the processing trends and to provide a quick and approximate output. However, the black model is unable to inspect any inner components or logic of the system. Our goal is to understand the behaviors of TensorFlow in datacenters, so the black-box model is unsuitable. The opposite of the black-box model is the pure white-box model, which describes as much system detail as possible. It is hard to create a pure white box model which is essentially a replica of the real-world system [18]. Besides, the white box model will consume a great deal of computing power. The gray box model is more generic and commonly used in the simulation. It combines physical knowledge with statistical representation to model a complex system. Considering the interpretability and simplicity, we choose the gray box model to capture the behaviors of TensorFlow.

1.2 Research Questions

Our main research question is *How to use discrete-event simulation to investigate the performance of the Google TensorFlow ecosystem?* To clarify the direction of this study, we elaborate on three research questions as follows.

RQ1. How to design a *deep* reference architecture for the Google TensorFlow ecosystem?

RQ2. How to create a predictive model of the Google TensorFlow ecosystem within a discrete-event simulator?

RQ3. What is the performance of the Google TensorFlow ecosystem in High-Performance Computing (HPC) environments?

1.3 Research Methods

To address RQ1, we extend our reference architecture created in our literature survey. We have mapped at least 10 components of the TensorFlow ecosystem. In this work, we add additional deeper layers for the TensorFlow programming model and execution engine. We identify more components and map to our reference architecture, as discussed in Chapter 3.

To address RQ2, we apply the AtLarge design process [14] to design the model of Google TensorFlow. This structured process contains eight steps: We (1) *formulate the requirements* and (2) *understand alternatives* in Section 4.1. The (3) *bootstrap design* is inspired by the literature survey on distributed machine learning systems, and background information and related work on performance analysis of Google TensorFlow ecosystem as shown in Chapter 2. We also make use of the reference architecture for datacenters of Iosup et al. [13], [14] in the design. The (4) *high-level and low-level design* of the model are presented in Section 4.2 and 4.3. Based on the design, we (5) *implement* the model, and integrate it with OpenDC [19] codebase using Kotlin programming language. After implementation, we refer back to the requirements and discuss the (6) *conceptual analysis of the design* in Section 4.4. The (7) *experimental analysis of the design* presents in Chapter 5. We (8) *summarize the design and the result* of the model in Chapter 4.

To address RQ3, we design experiments on the performance of TensorFlow in Chapter 5, following the general process for computer systems of performance analysis proposed by Jain [20]. There are four types of experiments in computer systems: real-world experiments, benchmarking, emulation, and simulation. We choose the simulation method which simplifies both workload and environments. The simulation method can overcome the difficulties of executing repeatable and reproducible experiments. In Section 5.1, We state the experiments goals. Section 5.2 describes the setup of the experiments, including the selected workload, configured environment, and selected metrics. We validate our model in Section 5.3 by comparing the results from published articles. We present and analyze the experimental results in Section 5.4.

1.4 Main Contributions

The main contributions of this thesis are:

1) A *deep* reference architecture for the Google TensorFlow ecosystem (Chapter 3).

2) A predictive model of Google TensorFlow ecosystem (Chapter 4).

3) An experiment to assess the performance of Google TensorFlow ecosystem (Chapter 5).

1.5 Outline

This thesis is structured as follows: Chapter 2 introduces background information on the Google TensorFlow ecosystem and related work on performance analysis of distributed machine learning systems. To address RQ1, Chapter 4 discusses the design of the model. To answer RQ2, Chapter 5 presents the experimental evaluation on the TensorFlow model and the performance of TensorFlow. In Chapter 6, we discusses the final remarks.

2

Background

This Chapter presents background information of this work. Section 2.1 introduces a reference architecture for datacenters of Iosup et al. [13]. Section 2.2 compares seven simulators for distributed systems. We discuss related work on performance modeling of distributed deep learning systems in Section 2.3 and performance evaluation and analysis of distributed DL systems in Section 2.4.

2.1 A Reference Architecture for Datacenters

In order to explore the generalization of artificial intelligence(AI), machine learning(ML) or deep learning(DL) solutions, we investigate a reference architecture for datacenter, proposed by Iosup et al. [13], [14]. In this reference architecture (Figure 2.1), there are five core layers from top to bottom: (5) *Front-end* (Section 2.1) is used for application-level functions; (4) *Back-end* (Section 2.1) manages tasks, resources and services on behalf of the application; (3) *Resources* (Section 2.1) performs task, resource, and service management on behalf of the Datacenter operator; (2) *Operations Service* (Section 2.1) is for basic services typically associated with (distributed) operating systems; and (1) *Infrastructure* (Section 2.1) manages physical and virtual resources. An orthogonal layer, (6) *DevOps* (Section 2.1), covers functions (such as monitoring, logging, and benchmarking), that is essential to operating the datacenter but orthogonal to the providing services.

Development(Front-end)

Front-end is for users or programmers to deal with a task with functions. Three sub-layers (*Applications*, *High Level Language* and *Programming Model*) help to capture systems or ecosystems with a finer granularity.

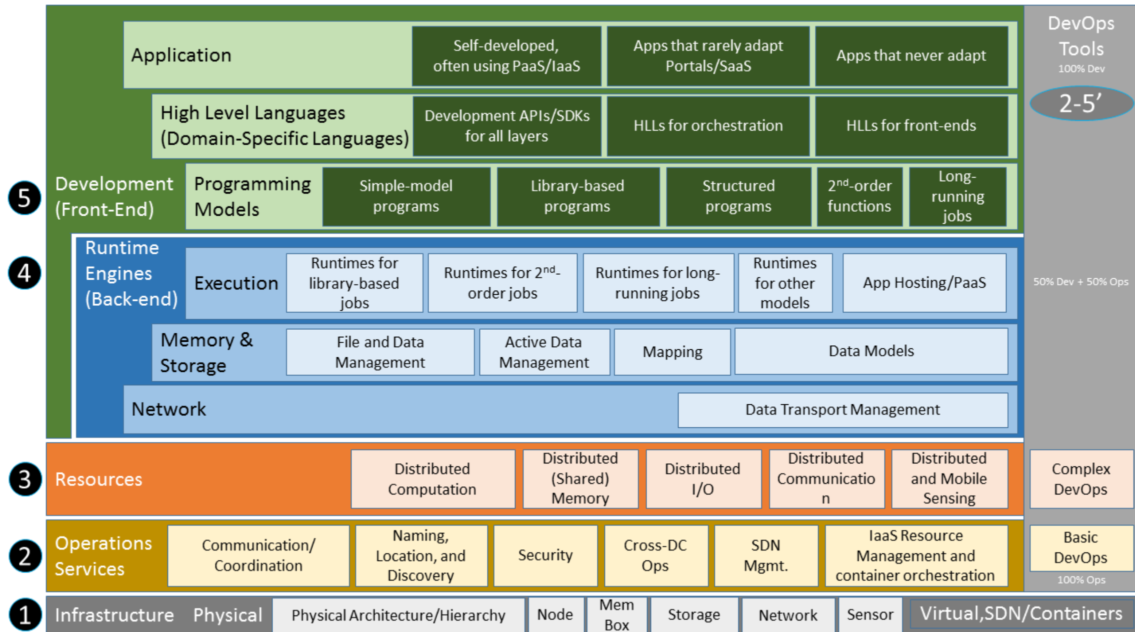


Figure 2.1: The reference architecture for datacenters of Iosup et al. [13] [14]

Applications *Applications* are specific software programs designed for users. According to dependence on Portals or SaaS[1], applications can be categorized as following three types: Self-developed Apps, often using PaaS/SaaS, usually are a program written in Python, Java, Go, Erlang, SQL or C#; Apps that rarely adapt Portals/SaaS includes Google Docs/Sheets, R-studio and Matlab; Apps that never adapt Portals/SaaS usually are legacy, especially with native codes and statically linked libraries.

High-Level Languages *High-Level Languages*(HLLs) are designed for human beings, not machines. Their principles are user-friendliness and easy to understand. High-Level Languages for AI usually provide high-level building blocks for ML/DL algorithms, feature engineering methods, parameters tuning methods, and tools for ML pipelines.

Development APIs/SDKs for all layers

HLLs for orchestration are tools that support automatically configure and manage a pipeline(such as ML pipeline or data pipeline). Examples are Google Cloud Dataflow, Apache Airflow, and KubeFlow Pipelines.

[1]In a cloud computing context, there are three levels of cloud services defined by National Institute of Standards and Technology(NIST): SaaS, Software as a Service; PaaS, Platform as a Service; and IaaS, Infrastructure as a service

HLLs for front-ends usually provide high-level building blocks for applications, such as Apache Pig, Apache Hive, BigQuery, and Impala.

Programming Models *Programming Model* is an abstraction that indicates how the code can be organize or reuse. Programming Model for AI/ML/DL usually is related to data abstraction or computation model.

Simple-model programs refers to functional programming, procedural programming, object-oriented programming, logic/rule-based programming, semantic network and so on.

Library-based programs provide a library of functions for implementing computation. They usually adopt functional or procedural programming. Examples are scikit-learn [21], Spark MLlib [22] and CUDA [23].

Structured programming models contain parallel programming model SPMD (single program multiple data) and MPMD (multiple program multiple data), bags of tasks, Directed Acyclic Graph(DAG) workflows, dataflow and services. Dataflow model is the most popular programming model for AI/ML systems such as TensorFlow [5], [6], MXNet [8], PyTorch [7] and Spark [24].

Second-order and higher-order functions are functions that use functions as input parameters or return value. An example is MapReduce [25].

Long-running jobs refer to streaming or event-based processing. Examples are Spark Streaming, Apache Beam [26] and Apache Flink.

Runtime Engines(Back-end)

Back-end manipulates low level operations for applications. This layer also has three sub-layers (*Execution*, *Memory* & *Storage* and *Network*).

Execution *Execution* is designed to interpret and execute a series of functions sequentially or concurrently with dependencies. Components in this level manage tasks, resources, and services for the applications and deal with the problems about synchronization, communication, scheduling, and fault tolerance. Corresponding to the programming models layer, there are five types of Executions engine: Runtimes for library-based jobs, such as CORBA, MPICH, and Microsoft High-Performance Computing platform; Runtimes

for second-order jobs, such as Hadoop, Spark, and Pregel [27]; Runtimes for long-running jobs, such as Apache Storm, Amazon S4, and Google MillWheel; Runtimes for other models, such as Apache Oozie; and App Hosting/Paas which refers to cloud application platforms, such as Google App Engine, Heroku, and MS Azure.

Memory and Storage To support the variety and velocity of data used and produced by AI, *memory, and storage* engine are important. Four components in these layers are described as follows:

File and Data Management refers to data replica, data recovery, and Content Delivery Network(CDN).

Active Data Management usually facilitates the process of data management, such as Extract-Transform-Load(ETL) data management and Active Storage which makes it simple to upload and attach files to cloud storage services.

Mapping usually refers to tools for Object-relational mapping(ORM) or other mapping using schema.

Data Models includes old, new or NoSQL database, key-value store, key-map store, document, graph, catalogs, and objects store.

Network *Network* engine requires to enable more reliable and high-performance data transmission. This layer does not have many components and one is Data Transport Management, such as GridFTP.

Resources

Resources Layer manage tasks, resources and services for Datacenter operators. This layer contains modules for Distributed Computation in a cluster, a node, or heterogeneous processors; Distributed (Shared) Memory architecture; Distributed I/O ; Distributed Communication with network links, routers, or Top of rack (ToR) switch architecture; and Distributed and Mobile Sensing with smart IoT(Internet of Things) networks or sensing networks.

Operation Services

Operation Services are some basic services associated with (distributed) operating systems. They are related to several aspects: Communication or Coordination, such as peer-to-peer(P2P) communication, queuing, grouping and locking; Naming, Location, and Discovery, such as universally unique identifiers(UUIDs), Directories, Replicas, and Gossiping; Security, such as Authentication, Authorization and Accounting(AAA) Security, access control lists(ACLs), demilitarized zone(DMZ) and single sign-on; Cross-DC Ops, such as Cross Datacenter Replication service and congestion control; SDN Management, that is Software-Defined Networking Management; and IaaS Resource Management and container orchestration, including public cloud providers, private cloud managers(such as OpenNebula), container orchestration(such as Docker).

Infrastructure

Infrastructure is the bottom layer in this RA. It consists of physical hardware and virtual or containerized machines, including Physical Architecture/Hierarchy for Datacenter, Room/Container, Rack, Cluster or Pod/Partition; Node; Memory Box; Storage Device (such as tape robot); Network Device; and Sensors.

DevOps Tools

DevOps, a combination of development and operations, is a set of processes or methods that promote communication, collaboration, and integration between different layers or different components. Basic DevOps are testing, benchmarking, configuring, and deploying tools. Complex DevOps includes monitoring, diagnostics tools and optimizers.

2.2 Simulators for Distributed Systems

In the field of computer systems, simulation is a popular, important, and cost-effective method [20]. It imitates the behaviors and operations of a system, to gain insight into the properties of the system under study and investigate the effects of alternative conditions. It simplifies both workloads and environments of distributed systems. One advantage of the simulation method is that it can overcome the difficulties of executing repeatable and reproducible experiments. Considering the simulation models of the systems which evolve, there are two types of simulators: (1) *continuous simulators*, where the state of the system changes continuously; and (2) *discrete event simulators*, where the state of the

system changes at discrete time points [28]. Discrete-event simulators are more preferable in computer systems since time is generally a discrete event (tick or cycle).

Many powerful discrete-event simulators for distributed systems has been developed, such as CloudSim [29], iCanCloud [30] and OpenDC [19]. CloudSim [29] is a well-known cloud computing simulator that can model the core functionality of clouds, such as the creation of private/public cloud environments, communication between different cloud entities, and resource management policies. Users define workloads by creating instances of *cloudlets*. *Cloudlets* are submitted to and processed by virtual machines (VMs) deployed in the cloud. Users can customize various resource management policies, such as VM allocation policies, VM scheduling policies, and *cloudlet* scheduling polices. iCanCloud [30] is similar to CloudSim, but it supports parallel simulation and provides a full graphical user interface (GUI). OpenDC [19] is a simulation framework, aiming at serving a diverse set of stakeholders. It provides basic resource models (CPUs, VMs, and memories), step-based workflow models, and failure models. It has a GUI for users to interact with the simulators. The above three simulators can model the environment and simulate different workloads running on them, but they lack comprehension of different computing models and applications.

Researchers start to extend simulators to support various applications, such as NetworkCloudSim [31] for parallel applications, MRSim [32] for MapReduce model, CEPSim [33] for complex event processing systems, and IOTSim [34] for Internet of Things(IoT) applications. NetworkCloudSim [31] extends CloudSim with a network model and a stage-based application model that can represent communicating processes. A task in an application contains some computation and communication phases. It allows more accurate evaluation of resource management policies to optimize the performance of parallel applications in a Cloud infrastructure. CEPSim [33] proposes a query model based on Directed Acyclic Graphs (DAGs) to represent Complex Event Processing (CEP) and Streaming Processing (SP) applications. It can be used to analyze and evaluate the performance and scalability of CEP queries with various query processing strategies. MRSim [32] and IOTSim [34] both model MapReduce applications as a set of Map tasks and Reduce tasks. MRSim focuses on the Hadoop environment, while IOTSim focuses on the IoT field. We cannot find any existing simulators that can effectively model ML/DL applications.

To analyze the properties of simulators for distributed systems, Ulita [35] proposes a taxonomy with six branches: (1)model instance; (2)model lifecycle; (3)simulation quality; (4)simulation capabilities; (5)simulator interaction; (6)simulation execution. Each branch

has 4 to 6 leaves that represent functional, non-functional, or stakeholder-related properties. Because this thesis focuses on the application layer, we compare seven mentioned simulators in terms of workload, resources, parallel execution, language, GUI support, and code availability in Table 2.1. OpenDC [19] is not the most mature simulator, but it provides attractive proprieties, such as failure model, energy consumption, and GUI support. Therefore, we choose OpenDC and create a predictive model for TensorFlow applications on top of it in Chapter 4.

2.3 Performance Modeling of Distributed DL Systems

Performance modeling is a method to model the behaviour of a system and predict various performance metrics (such as execution time) of the system with a given workload and setup. Modeling the performance of parallel computing and distributed systems is commonly used for scalability analysis [36], resource allocation [37], and capacity planning [38]. Most of prior work is targeted on specific distributed systems or parallel computing applications, such as MPI-based applications [39], MapReduce [40] [41], and stream processing applications [42].

More recently, several performance models have been developed for distributed deep learning systems. Yan et al. [43] develops a mathematical model for scalability estimation and optimization of distributed deep neural network (DNNs) models training. It predicts the training time of DNNs and quantifies the performance impact of data and model partitioning and system provisioning decisions. Similarly, Hashemi et al. [44] proposes a performance model of DNNs and asynchronous stochastic gradient descent algorithm. They focus on the system scalability of the CNTK framework. They conclude that poor I/O utilization degraded the whole execution time. Shi et al. [45] and Alqahtani et al. [46] model DNNs training with synchronous stochastic gradient descent algorithm. Shi et al. [45] focus on GPU servers and clusters, while Alqahtani et al. [46] focus on different system architectures, such as parameter server (PS), peer to peer (P2P), and ring-allreduce (RA). Qi et al. [47] introduce an analytical performance model of DNNs training called *PALEO* at a very fine granularity. It explores the design space of a deep neural network application the choice of network architecture, hardware, software, communication schemes, and parallelization strategies. Our predictive model in Chapter 4 covers the design choices supported by these prior studies.

Table 2.1: Simulators for distributed systems.

Simulator	Year	Workload	Resources	Parallel Execution	Language	GUI Support	Availability
CloudSim [29]	2011	Miscellaneous	DCs, VMs, Storage, Network	No	Java	No	Open Source
iCanCloud [30]	2012	Miscellaneous	CPUs, Network, Storage	Yes	C++	Yes	Open Source
OpenDC [19]	2017	Workflows	VMs, CPUs, GPUs	Yes	C++/Kotlin	Yes	Open Source
NetworkCloudSim [31]	2011	Parallel	DCs, VMs, Storage, Network	No	Java	No	Open Source
MRSim [32]	2010	MapReduce	DCs, VMs, Storage, Network	No	Java	No	Open Source
CEPSim [33]	2016	Streaming	VMs, Network	No	Java	No	Open Source
IOTSim [34]	2017	MapReduce	VMs, Storage, Network	No	Java	No	No

2.4 Performance Evaluation and Analysis of Distributed DL Systems

Apart from performance modeling, empirical performance analysis is a widely used approach to study the performance of distributed systems. There are several studies conducting machine learning workload characterization to improve resource utilization and workload performance in ML clusters.

Several recent work [48] [49] aims at identifying and understanding the performance bottlenecks of deep learning training workloads. Guignard et al. [48] conduct several experiments and analyze the performance of the Fathom [50] workloads on the IBM "Minsky" platform. They focus on the computational behaviors. Wang et al. [49] establish an analytical characterization framework to investigate the performance of deep learning training workloads from the Platform of Artificial Intelligence (PAI) in Alibaba.

Chien et al. [51] characterize the I/O performance and scaling of DL workloads in TensorFlow and focus on its interaction with the storage system. They design a microbenchmark to measure TensorFlow reads, and then use a TensorFlow mini-application based on AlexNet to measure the performance cost of I/O and check-pointing in TensorFlow. Their work shows that TensorFlow provides high-performance I/O on a single node in the data-ingestion phase and that burst-buffer can be an effective technique for fast check-pointing.

There are also some studies [52] [53] for a multi-tenant cluster where multiple ML applications (tenants) share a set of resources. Park et al. [52] analyze the inference workloads in a Facebook datacenter, pointing out limitations of the current ML infrastructure [53] and providing suggestions for future general-purpose/accelerated inference hardware. Jeon et al. [54] present a detailed workload characterization of a two-month trace from a multi-tenant GPU cluster in Microsoft and focused on resource utilization, job scheduling, and failure analysis.

3

Reference Architecture for Google TensorFlow Ecosystem

Reference architectures could help to conquer the complexity and manage the diversity when designing or analyzing systems. In this chapter, we address our first research question: How to design a ***deep*** reference architecture for the Google TensorFlow ecosystem?

3.1 Overview

In this chapter, we extend the reference architecture from our literature survey. We identify more components for the TensorFlow ecosystem and map them to the reference architecture in Section 3.2. We also add additional depth to TensorFlow programming model and execution model based on TensorFlow architecture [5] in Section 3.3. Figure 3.1 presents our results. Those components marked with red diamonds are new in this work.

Method Our process is mainly based on TensorFlow scientific publications and documentation or tutorials from their official sites. Other materials such as source code may also be considered if available. The first step of our mapping process is to extract components of a system. Different components usually have different functionalities. Then we map each component to each layer. It is also possible that one layer has several components and one component maps to several layers. Further, we consider other components that do not belong to this system but are required or supported by this system.

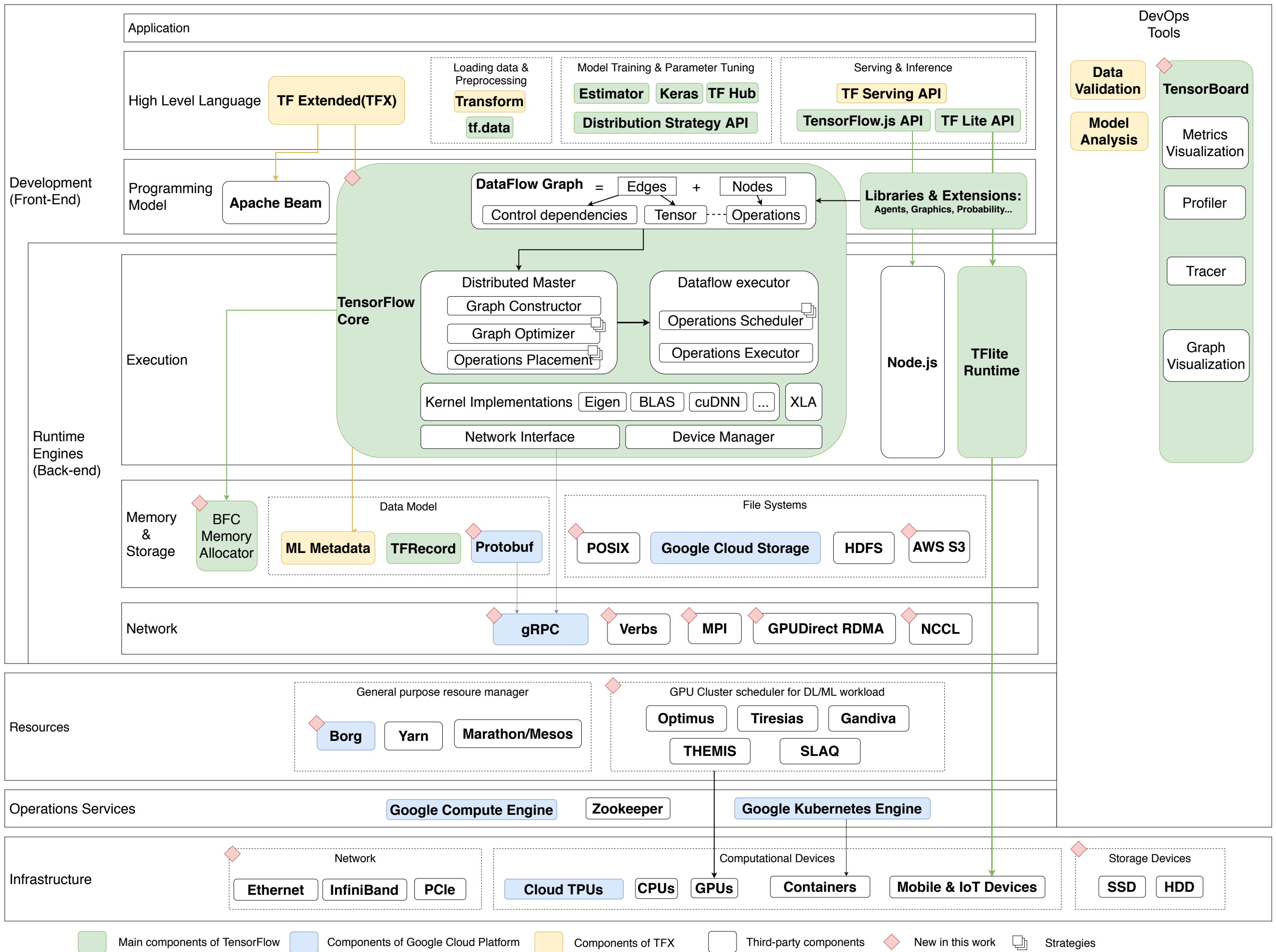


Figure 3.1: Reference architecture for Google TensorFlow ecosystem.

3.2 Reference Architecture for TensorFlow Ecosystem

There are six layers in our reference architecture (Figure 3.1): development, runtime engines, resources, operations services, infrastructure, and DevOps. In this section, we introduce each layer.

3.2.1 Development

Development layer is for users or programmers to deal with a task with functions. There are three sub-layers (*Applications*, *High Level Language*, and *Programming Model*).

3.2.1.1 High-Level Language

An ML/DL workflow contains three stages: loading & preprocessing data, model training & parameter tuning, and serving & inference. Most components in this layer can be placed into these three stages. One component in TensorFlow is used to orchestrate ML pipelines. If not mentioned specifically, components have the most complete and stable support for Python and they also support JavaScript, C++, Java, Go, and Swift language.

Loading and Preprocessing Data Two components (*tf.data* and *Transform*) mapped into this stage are both high level APIs. The *tf.data* API enables to build input pipelines. An input pipeline usually is an Extract-Transform-Load(ETL) process. The Extract phrase reads input data and constructs a *Dataset*, an abstraction represented a sequence of elements that have the same structure, from data stored in memory or file systems. It supports a variety of data formats, including text, CSV, images, and videos. The Transform phase performs data preprocessing, such as shuffling, time-series windowing, and other operations with the map function. The final phase loads the transformed data to available accelerators (e.g., GPU or TPU) to train models.

Transform is a component offered by TensorFlow Extended(TFX) [55]. It is a standalone library based on Apache Beam, which is an advanced unified programming model for batch and streaming data processing. Using *Transform*, a *processing function* will be defined to describe a series of operations that transform raw data into the data used to train a model and then be converted into a Beam pipeline to transform data.

Model Training and Parameter Tuning This stage is essential and contains four components. *Keras* [56] is a high level, user-friendly, functional API for Neural Network. It is written in Python and runs on top of TensorFlow. TensorFlow implements *tf.keras* to

train models and support TensorFlow-specific functionality (such as *tf.data* pipelines). The advantages of Keras are simplicity, modularity, and extensibility. It minimizes the number of user actions required for common use cases and provides standalone, fully-configurable modules (such as neural layers, cost function, and activation function) that can be plugged together with a few restrictions. New modules are also easy to add as existing modules provide enough examples.

Estimators [57] is also a high-level API for specifying, training, and evaluating ML models. Compared to *tf.keras*, Estimators support for parameter server-based training and full TFX integration. The Keras model can be converted to Estimators to access Estimators' strength.

TensorFlow Hub is a repository for transfer learning, providing pre-trained models that can be reused to solved new tasks with less training time and training data.

Distribution Strategy is a relatively low-level API for *Execution* layer to distribute training across multiple machines, GPUs, or TPUs and can be used with high-level APIs such as Keras. There are different types of strategies, including mirrored strategy with an all-reduce algorithm for synchronous, parameter server strategy for asynchronous training, and other strategies for different hardware platforms.

Serving and Inference The trained and saved models can be run directly or be deployed to provide services. There are three components for deployment: *TensorFlow Serving*, *TensorFlow.js* and *Tensorflow Lite*. *TensorFlow Serving* is a production-level flexible and high-performance serving system. It supports RESTful API and gRPC API. *TensorFlow.js* allows training and deploying models in a JavaScript environment, such as in a web browser or a server using Node.js as back-end. *TensorFlow Lite* is a lightweight solution for mobile and embedded systems. It consists of two main components: a converter which converts models into an efficient form, and an interpreter which runs specially optimized models on different hardware types. It supports Android systems with Java, iOS systems with Swift or Objective-C, and Linux-based embedded systems (such as Raspberry Pi and Edge TPU) with Python.

ML pipelines *TensorFlow Extended(TFX)* [55] is a high level language for orchestration. TFX supports for continuous training and serving with production-level reliability and provides reusable built-in components (Data Analysis, Data Transformation, Data Validation, Trainer, Model Evaluation and Validation, and Serving). A TFX pipeline requires Apache Beam as a programming model and it is also possible to use Apache Airflow or KuberFlow

Pipelines as orchestrators to author, schedule, and monitor ML workflows. ML Metadata is an integral component of TFX to retrieve and record metadata associated with ML workflows (such as *Execution* metadata, which records a run in an ML workflow and the runtime parameters). The storage backend of ML Metadata can be SQLlite(in-memory or disk) or MySQL.

3.2.1.2 Programming Model

Apache Beam [26] is required when using TFX components. It is a unified programming model for batch and stream data processing. It belongs to long-running jobs using streaming and structured programs using DAG workflows or dataflow. It supports multiple distributed processing back-ends, such as Apache Spark, Apache Flink, and Google Cloud Dataflow. Java, Python, and Go languages can be used to implement a Beam model.

TensorFlow Core is libraries-based and a structured dataflow programming model. All TensorFlow applications could be expressed as a dataflow computation graph. We discuss more detail about this component in Section 3.3.1.

Libraries & *Extensions* are libraries-based programs and built on TensorFlow. They help to build advanced models or domain-specific applications, such as Agents for reinforcement learning, Graphics for computer graphics functionalities, and Probability for probabilistic reasoning and statistical analysis.

3.2.2 Runtime Engines

Back-end manipulates low level operations for applications. This layer also has three sub-layers (*Execution*, *Memory and Storage*, and *Network*).

3.2.2.1 Execution

TensorFlow Execution belongs to runtimes for second-order jobs. It is written in C and consists of client, master, and worker. The client program uses *Session* to execute a graph and interact with the master and workers. The master schedules tasks over workers and workers execute tasks. No parameter servers are in TensorFlow, instead, ps tasks are generated for parameter updates. If the application runs on multiple devices, a node placement algorithm will be run to map the computation onto a set of available devices (such as CPUs and GPUs). Fault tolerance is handled by user-level checkpoints. We discuss more detail about this component in Section 3.3.2.

Node.js is runtime for TensorFlow.js and is a JavaScript runtime. It executes a JavaScript code on a browser or a server. *TFlite Runtime* is runtime for TF lite library. It deploys ML models on mobiles and IoT devices.

3.2.2.2 Memory and Storage

Memory TensorFlow runtime is responsible for memory allocation and garbage collection [58]. It adopts best-first with coalescing (BFC) memory allocator.

Data Model *ML Metadata* is a library for recording and retrieving metadata associated with ML workflows. Data models that it uses contains *Artifact*(a component or a step in an ML workflow), *Execution*(a record of a run and the runtime parameters), *Events*(a record of the relationship between an Artifact and Execution) and other records. It supports storage back-end SQLite(in-memory or disk) and MySQL.

TFRecord is a file format for storing a sequence of binary records. It helps to read large data efficiently and makes use of memory and storage. *tf.data* the module provides functions to read and write TFRecord files.

Protobuf is a mechanism for serializing structured data and is similar to JSON and XML. It is designed by Google. gRPC uses Protobuf for communication.

File Systems TensorFlow supports to save data in cloud storage (such as Google Cloud Storage and AWS S3), POSIX (Portable Operating System Interface) file systems, and distributed file systems (such as HDFS).

3.2.2.3 Network

gRPC, developed by Google, is a high-performance universal Remote Procedure Call (RPC) framework. It is the main network engine for TensorFlow. TensorFlow also supports Verbs protocol, MPI, GPUDirect RDMA, and NCCL.

3.2.3 Resources, Operation Services and Infrastructure

Resources Layer manage tasks, resources and services for Datacenter operators. In this layer, we identify two types of resource managers. One type is general purpose resource manager, including *Borg* [59], *Hadoop Yarn* [60] and *Apache Mesos* [61]. The other type is GPU cluster scheduler designed for ML/DL workload, including *Optimus* [62], *Tiresias* [63], *Gandiva* [64], *SLAQ* [65], *THEMIS* [66].

In *Operation Services* layer, *Google Kubernetes Engine* and *Google Compute Engine* are IaaS resource management and container orchestration. *Zookeeper* [67] can provide basic services for maintaining configuration information, naming, and grouping.

In *Infrastructure* layer, TensorFlow supports various computational devices (including CPUs, GPUs, Cloud TPUs, containers, and mobile and IoT devices) and different types of networks to connect computing nodes (such as InfiniBand, Ethernet, and PCIe).

3.2.4 DevOps

Data Validation(TFDV) and *Model Analysis*(TFMA) are components of TFX. They are DevOps tools for *High-level language* layer. TFDV can check and analyze data by computing descriptive statistics, inferring a schema, and detecting data anomalies. TFMA allows evaluating the model on a large amount of data. They both support to be visualized in Jupyter notebooks.

TensorBoard is a complex DevOps tool provided by TensorFlow for logging training metrics (such as loss and accuracy), visualizing the model graphs, tuning hyper-parameters, tracing, monitoring and profiling a program.

3.3 TensorFlow Core

As Figure 3.1 shows, we add additional depth to TensorFlow programming model and execution model based on TensorFlow architecture [5]. In this section, we illustrate the details of TensorFlow programming model (Section 3.3.1) and execution model (Section 3.3.2).

3.3.1 Programming Model

A computation is abstracted as a stateful dataflow graph. Each node represents an operation(e.g., Add, Matrix Multiply, or Sigmoid function). Tensors, data with N-dimensions, flow along normal edges in the graph. The graph also has some special edges, called control dependencies, which enforce orderings of operations. Most tensors cannot survive after execution, so variables are a special kind of operations that can handle a persistent mutable tensor to survive across executions of a graph. In ML applications, model parameters are usually stored in variables type.

3.3.2 Execution

Distributed Master contains three steps. It constructs the computational graph and prunes it for optimization. There are various graph optimizer (such as Grappler [68]) and optimization algorithms. Graph optimizer aims to improve TensorFlow performance through graph simplification and improve hardware utilization by optimizing the mapping of graph nodes to compute resources [68]. It is an optional step, so we will not discuss graph optimizer in our modeling work. Graph partitioner partitions the graph to obtain graph pieces for each participating device and distributes the sub-graph to workers. The distributed master also caches these pieces so that they may be re-used in subsequent steps.

Dataflow executor schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, etc). It also performs send and receive operations to communicate with other devices.

Kernel Implementations perform the mathematical computation for each operation. It contains various linear algebra libraries (such as Eigen and BLAS) and GPU-accelerated libraries (such as cuDNN).

XLA(Accelerated Linear Algebra) is a domain-specific optimizing compiler for linear algebra. It provides an alternative mode to run an ML model, aiming at improving execution speed, memory usage, and portability.

Network Interface TensorFlow supports multiple communication protocols, including gRPC over TCP, and RDMA over Converged Ethernet.

Device Manager manage multiple devices (CPUs, GPUs, or TPUs) with a registration mechanism. TensorFlow has special naming rules for devices. The name of a device is the unique identifier of the device.

3.4 Discussion

In this chapter, we extend the reference architecture from our literature survey. Two main contributions of this chapter are as following:

1. We identify more components at several layers, such as *Resource* layer and *Network* layer. These components are well mapped to our reference architecture and enrich our reference architecture.

2. We expand the TensorFlow programming model and execution engine with the additional depth based on TensorFlow architecture [5]. This extension of the reference architecture helps to design our predictive model in Chapter 4.

Threats to Validity

In this chapter, *single reviewer bias* happens in the process of creating the reference architecture. We only have one single person to perform the identifying and mapping components manually. The result is based on the comprehension of relevant materials. Although we try to utilize as many materials (relevant papers, tutorials, and source code, etc.) as possible, only one person may miss some additional materials or misunderstand the functionalities of some components. Then the result is not reliable. An alternative way is to use a pair-reviewing or multiple-reviewing system [69]. Several reviewers perform the identifying and mapping process independently. We collect the results from all the reviewers. Then we merge the results using statistical methods or other methods (such as showing additional evidence) to reach an agreement.

4

Modeling Google TensorFlow Ecosystem

The reference architecture provides a guideline for architectural design and analysis. Based on the reference architecture from Chapter 3, and considering the interpretability and simplicity of a model, we address our second research question in this chapter: How to create a predictive model of the Google TensorFlow ecosystem within a discrete-event simulator?

The remainder of this chapter is structured as follows. Section 4.1 states the requirements of the model. Section 4.2 describes the overview of the model. In Section 4.3, we introduce our TensorFlow predictive model in details. In Section 4.4, we refer back to the requirements of our model and discuss potential threats to validity.

4.1 Requirements

The model is used to evaluate the performance of the Google-TensorFlow ecosystem. The model should be able to predict the TensorFlow workload, imitate its execution in clusters, and monitor the states of applications and clusters. We analyze the requirements of the model layer by layer in this section.

R1. Application: The application model is expected to fulfill the generality requirements, that is, it could represent all TensorFlow applications.

R2. Execution: TensorFlow expresses numerical computations as a dataflow graphs. The execution of a graph mainly contains graph optimization, graph partitioning on different devices, and local scheduling on each device (i.e., the selection of next operations

to be executed). The model needs to capture these behaviors of the execution engine of the TensorFlow system.

R3. Data Management: Task processing includes data access(read/write). ML/DL jobs need to handle I/O operations such as loading training or validation samples from files, updating model parameters, and saving trained models or checkpoints to a storage system. When a replica is located at remote resources, communication overhead may happen. The model is required to provide an interface for storage systems and manage data sizes, locality, and transfer process.

R4. Heterogeneous Resources: In TensorFlow, there are different kinds of heterogeneity, including the memory capacity, the computational speed, and the network bandwidth. It is expected to provide models for heterogeneous resources and support to manage these resources with various policies.

R5. Datacenter Topology should be configurable. The purpose is to make it suitable for a different design of the physical and logical layout of the resources and their connectivity in different datacenters or HPC environments.

R6. Metrics should be collected for both TensorFlow users and cluster administrators. TensorFlow users should be able to monitor their jobs' status and extract data from allocated devices. With R1-R3, users can study the scalability of their jobs and identify performance bottleneck, which can help to make better decisions on distributed strategies, synchronization protocols, and the specifications of devices. Multi-tenant clusters administrators can monitor the status of the cluster and the utilization of different machines or units. With R4 and R5, administrators can study the impact of heterogeneous resources management and cluster topology, to improve clusters utilization and capacity to run more ML/DL jobs.

4.2 Overview

To address the requirements, we design the architecture of our model in Figure 4.1.

The OpenDC simulator We choose OpenDC simulator [19] as our base simulator. The OpenDC simulator provides a basic discrete event simulation framework and two main components: OpenDC model and OpenDC simulation engine. The simulation engine controls the main simulation loop and schedules the simulation events. The model is used

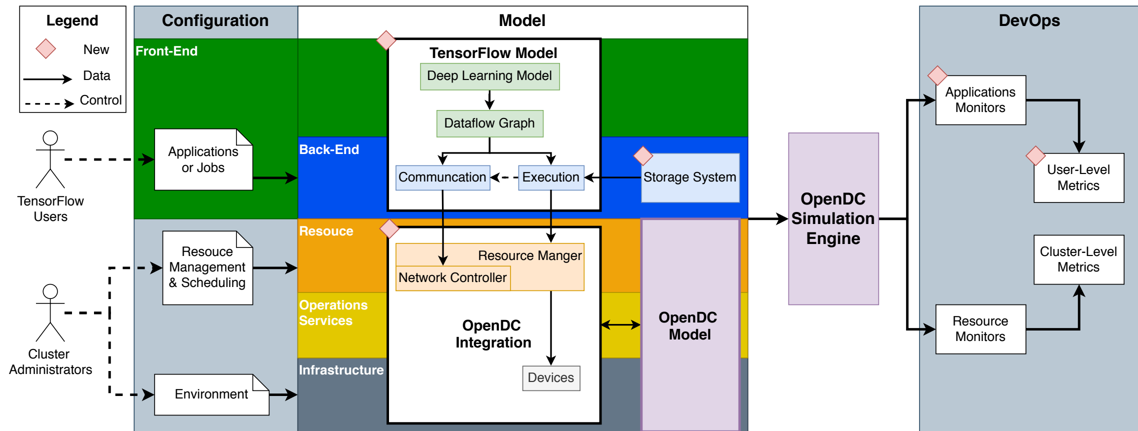


Figure 4.1: Architecture of the model.

to define the physical datacenter environments, the policies that control the dynamics of the datacenter, and various workloads running on the datacenter. Following list the interesting OpenDC features that could be reused on this project.

Bare Metal Provisioning determines how to map requested devices to available nodes in the datacenter. Datacenter providers can customize allocation and scheduling policies to maximize resource utilization.

Workflow Workload Model determines how a workflow job schedule and run on nodes. A workflow job consists of a set of tasks with inter-dependencies. Andreadis et al. [70] also propose a reference architecture for datacenter scheduling using this workflow workload model.

OpenDC Integration We implement the pieces necessary to integrate the OpenDC simulator with the TensorFlow model logic. The main classes created for the integration are depicted in the following:

Resource Manager extends the workflow scheduler to handle ML applications. It is used for resource allocation and deallocation. There are various allocation policies, such as first-fit policy and random policy.

Network Controller controls network flow along with machines. Because there is no existing network model in the OpenDC simulator, this is a minimal form of network model for datacenters. We assume all machines are connected. Then we

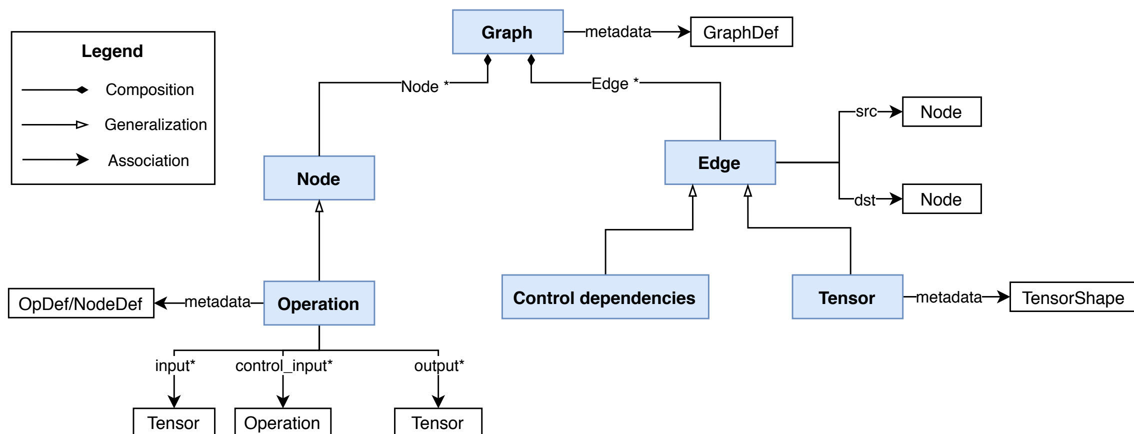


Figure 4.2: Domain model of a computational graph.

create a bandwidth lookup table and a network latency lookup table. The duration of a single network flow can be computed as: $network_delay = latency + message_size/bandwidth$. As Figure 4.5 shows, the source device sends a message to the network controller. The network controller checks the destination device calculates the network delay, and then sends the message to the destination device.

Devices extends bare-metal devices to execute ML tasks. To support heterogeneous resources, we add a flag for GPU awareness. We also implement several functions for communication tasks.

4.3 TensorFlow Model

In this section, we illustrate our TensorFlow model in details: application model (Section 4.3.1), execution model (Section 4.3.2), communication model (Section 4.3.3)

4.3.1 Application Model

Considering the generality, an ML application is modeled as a dataflow computational graph as Figure 4.2 shows. A graph consists of a set of nodes and edges. *GraphDef* is a serialized version of a graph in ProtoBuf format. Each node represents an operation (such as add, matrix multiply, or sigmoid function). The metadata of an operation is held by *OpDef* and *NodeDef* in ProtoBuf format. *OpDef* describes the static information of operation, such as operation type. *NodeDef* describes dynamic information, such as

operation name and device information. Tensors ow along normal edges and some special edges control dependencies. Figure 4.3 is an example of data ow computational graph.

Figure 4.3: An example of data ow computational graph.

However, the data ow computational graph is a relatively low-level application model and it is too complex to model a real-world ML application using the data ow model. For convenience, we also create a high-level deep learning model. A layer is a general and central abstract of deep learning. It contains a set of operations. We model each layer in the neural network as a node and the connections between layers as edges.

4.3.2 Execution Model

Based on Section 3.3.2, we design the execution model in Figure 4.4. Distributed Master coordinates the execution of an application. It could accept di erent strategies to orchestrate jobs. There are di erent aspects to consider these strategies. One is the parallelism method, including data parallelism and model parallelism. When we train a DL model using data parallelism, each worker will have an entire copy of the DL model. The training data is divided into several non-overlapping chunks. Each worker loads a chunk of training data and trains its data. After one or more iterations, all the workers will update and synchronize model parameters. When we train a DL model using model parallelism, the DL model will be split. Each worker holds a di erent part of the DL model. The workers holding the input layer of the DL model will load the training data. There are two phases of DL training: forward propagation and backward propagation. In the forward propagation, the workers compute the output and propagate the output to the workers holding the next layer. In the backward propagation, the workers holding the output layer of the DL model will compute the gradients and propagate the gradients to the workers holding the input layers.

Another aspect is synchronization, including synchronous training and asynchronous training. In synchronous training, all workers train on di erent chunks of input data, and update model parameters at each step [6]. In asynchronous training, all workers are independently training on the input data and updating model parameters asynchronously [6].

Figure 4.4: Execution model.

Users can customize their own distributed strategies with the device manager and operation placer. Following are three common strategies currently supported by TensorFlow:

Default Strategy is no distribution strategy. It places all variables and computations on a single device. A distributed master also acts as a data ow executor to execute a computational graph.

Parameter Server Strategy supports training on multiple devices. There are two types of devices: workers and parameter servers. The model parameters are placed on parameter servers. Workers will perform computational tasks and communicate with parameter severs to updates the model asynchronously or synchronously.

Mirrored Strategy supports synchronous distributed training on multiple GPUs. By

default, it uses e cient ring-allreduce algorithms [6] to transfer model variable updates across the devices.

The default operation scheduling algorithm is Depth-First-Search (DFS). It is possible to customize the operation scheduling algorithm. Mayer et.al. [71] have tried Highest Path Computation Time rst (PCT) scheduling algorithm to reduce the execution time of the critical path on the graph and Maximum Successor Rank rst (MSR) scheduling algorithm to optimize the resource utilization of the devices.

Operations executor executes two types of operations: mathematical computation operations and communication operations (Send or Receive operations). A computation operation processes a batch size of data and could be calculated as the information of FLOPs (oating-point operations). FLOPs calculator is mapped to kernel implementations. The calculation of FLOPs refers to literature in Section 2.3.

4.3.3 Communication

It is necessary to consider the execution of data communication tasks when evaluating distributed DL applications. Once the graph is partitioned into a set of subgraphs by graph partitioner, any cross-device edge (i.e. the source node and destination node are on di erent devices) will be removed and replaced by an edge to a new Send node and an edge to a corresponding Receive node. When the operation executor gets this task, the source device encodes its message and sends a message to the network controller as Figure 4.5 shows. The network controller searches the destination device in the lookup table, calculates the network delay, and then sends the message to the destination device. Figure 4.5 presents an asynchronous non-blocking communication.

4.4 Discussion

In this chapter, We create a predictive model for the TensorFlow ecosystem based on the requirements listed in Section 4.1.

1. We create the application model (R1) and execution model (R2) of TensorFlow based on our analysis from the reference architecture in Chapter 3.

2. We create a minimal form of storage and network model to ful ll the requirement R3.

Figure 4.5: Time diagram for communication tasks.

3. Our model is embedded in the OpenDC simulator. We leverage its existing features and extend it to support heterogeneous resources (R4), the con guration of clusters (R5), and some cluster-level metrics (R6).

4. Several user-level metrics, such as execution time, are also generated to meet the requirement R6.

Threats to Validity

In this chapter, one potential threat to validity is the veri cation of the predictive model. The design of our model is derived from the reference architecture of the TensorFlow ecosystem in Chapter 3. It is hard to verify our model because we create it based on a conceptual analysis of the reference architecture. One possible method is to perform formal analysis, but it is out of our scope. Another threat to validity is the validation of the model. It depends on whether the implementation of the model works. We will discuss more details on the experiments of model validation in the next chapter (Chapter 5).

Figure 4.6: Sequence diagram: communication among entities.

5

Experiments on the Performance of TensorFlow

In this chapter, we address our third research question: What is the performance of Google TensorFlow ecosystem in High-Performance Computing (HPC) environments? Our approach is to design simulation experiments to evaluate the performance of the TensorFlow ecosystem in HPC environments. We create our simulation model in Chapter 4. We first validate and calibrate our model through simulations set up to match the real-world experiments of Chien et al. [51] and Guignard et al. [48]. Then, we use our model to analyze the performance of the TensorFlow ecosystem in HPC environments.

We focus in this chapter on a modern interpretation of performance, which includes not only conventional performance metrics (e.g., response time) but also how well the system performs in terms of power consumption.

The remainder of this chapter is structured as follows. Section 5.1 states the goals of the experiments. Section 5.2 describes the setup of experiments, including the selected workload, configured environment, and selected metrics. In Section 5.3, we validate and calibrate our predictive model. In Section 5.4, we present and analyze the experimental results.

5.1 Overview

The goals of the experiments are (1) to calibrate and validate our predictive model designed in Chapter 4; (2) to use our predictive model, embedded in a simulator, to explore the performance of TensorFlow workloads in HPC environments. Table 5.1 depicts an overview

of all experiments, whose setup we summarize in Section 5.2. We elaborate on each of these two goals in the following, in turn.

To calibrate all features in our model, we reproduce two experiments from peer-reviewed papers [48] and [51] as follows:

1. Chien et al. [51] conduct experiments on characterizing Deep Learning (DL) I/O workloads in TensorFlow and focus on its interaction with the storage system. We will use their results and descriptions to validate the storage component included in our simulation setup.

2. Guignard et al. [48] port Fathom [50] workloads to the IBM Minsky platform, to study its performance. Experiments from [48] can help validate the communication processes between devices. We use this information for our validation experiments.

We describe the details and results of calibration and validation experiments in Section 5.3.

Although the environments and workloads we study admit many heterogeneous con g-urations, in this chapter we focus on four main aspects. First, the selection of a strategy (de ned in Section 4.3) a ects the dynamic operation of the system. Second, HPC environments face challenges when scaling under ML workloads. Third, we are interested to understand how the system becomes used and thus how its utilization evolves over time.

To evaluate the performance of Google TensorFlow ecosystem in HPC environments, we design four types of experiments: strategy comparison (Section 5.4.1), scalability study (Section 5.4.2), and cluster utilization study (Section 5.4.3).

5.2 Experiment Setup

In this section, we illustrate the setup of our experiments. The experimental setup includes the selected workload, con gured environment, and selected metrics. We describe the process of recreating workloads (Section 5.2.1) and environments (Section 5.2.2) from two peer-reviewed articles for calibration purposes.

5.2.1 Workload

Table 5.2 describes an overview of the workloads used in this work. We recreate each workload used in this thesis from the description presented in peer-reviewed articles, as follows:

Table 5.1: An overview of all experiments. The unique elements of each experiment are emphasizedin bold . Strategies: D: default strategy, PS: parameter server strategy, M: mirrored strategy; Metrics: T_{exec} : execution time, NET: normalized execution time, PC: power consumption. For details ofWorkload, see Table 5.2. For details ofEnvironment, see Table 5.1 and 5.3.

x	Goal	Workload	Environment	Strategies	Metrics
5.3	Calibration and Validation	A1 Fathom	HPC at KTH IBM Minsky platform	D D/ PS	T_{exec} NET
5.4.1	Strategy Comparison	A1 Fathom	HPC at KTH IBM Minsky platform	PS/ M	T_{exec} NET
5.4.2	Scalability Study	A1 Fathom	HPC at KTH IBM Minsky platform	PS/M	Speedup
5.4.3	Cluster Utilization Study	A1 Fathom	HPC at KTH IBM Minsky platform	D/PS/M	Usage, PC

Table 5.2: An overview of the workloads.

Name	Model	NumParams	FLOPs	Dataset	#samples	Batch size	Iterations
A1 [51]	AlexNet	61 M	0.7 G	Caltech 101	9144	16, 32, 64, 128	142
Fathom [50]	VGG-19	138 M	16 G	ImageNet	14 M	8	200
	AlexNet	61 M	0.7 G	ImageNet	14 M	64	200

For the A1 workload of Chien et al. [51], we consider a single TensorFlow mini-application, which computes the AlexNet model and uses the Caltech 101 dataset. The dataset contains 9,144 images with the average image size of around 14kB. The application is executed 142 iterations. The batch size is varied between 16, 32, 64 and 128.

For the Fathom workloads [50], we consider two applications as Table 5.2 shows. For each application, we execute 200 iterations. The arrival pattern of these seven applications has no relation with the results, so we test them one by one.

5.2.2 Environment

Table 5.4 describes an overview of HPC environments. We recreate HPC environments used in this thesis from the description presented in peer-reviewed articles as follows:

For HPC at KTH, we leveraged already existing computing devices components in OpenDC and modi ed the performance characteristics as in Table 5.3. Each node

Table 5.3: Types of computing devices. Mem_Bw: memory Bandwidth; Max PC: maximum power consumption; Min PC: idle power consumption.

Types	Speci cation	Cores	Frequency	Memory	Mem_Bw	Max PC	Min PC
CPU1	Intel E5-2690v3 Haswell	24	3498 MHz	512 GB	68 GB/s	135 Wh	84 Wh
CPU2	IBM POWER8	10	3690 MHz	160 GB	115 GB/s	190 Wh	130 Wh
GPU1	NVIDIA Tesla K80	4992	824 MHz	24 GB	480 GB/s	334 Wh	90 Wh
GPU2	NVIDIA Tesla P100	3584	1190 MHz	16 GB	720 GB/s	250 Wh	125 Wh

Table 5.4: An overview of HPC environments. For details of devices, see Table 5.3.

Environments	Storage System	Devices per Node	Network
HPC at KTH [51]	Lustre	1 CPU1+1 GPU1	12 GB/s EDR In niband
IBM Minsky platform [48]	not used	2 CPU2+4 GPU2	12 GB/s EDR In niband

has one CPU and one GPU. KTH GPU cluster has nine thin nodes. The parallel le system used is Lustre. The nodes are connected by 12GB/s EDR In niBand network. We con gured the data transfer rate of the Lustre storage system as summarized in Chien et al. [51, Table 1].

For IBM Minsky platform, we added to the set of components supported by OpenDC computing devices with the con guration summarized in Table 5.3. Each node has two CPUs and four GPUs. IBM Minsky platform has twelve nodes. The nodes are connected by 12GB/s EDR In niBand network. The experiments in Guignard et al. [48] do not use the storage sub-system; instead, all data is in-memory. We con gured the network matrix in our network controller model with the Minsky network topology diagram from Guignard et al. [48, Figure 1].

5.2.3 Metrics

In this chapter, we use the following metrics:

- Execution time T_{exec} of an application.
- Communication time T_{comm}, computation time T_{comp}, and memory access time T_{mem} of an application. Memory access time is a part of computation time.
- Normalized execution time [48]: $NET = T_{measured} = T_{default}$, where $T_{measured}$ is the predictive execution time of the application and $T_{default}$ is the predictive execution time of the application using default strategy.

Speedup is the ratio between the execution time with the number of computing workers equal to 1 and the execution time with the number of computing workers equal to n: $T_{exec}(1)=T_{exec}(n)$. We only considerstrong scaling, which means the problem size is kept constant, andhorizontal scaling, which means the added resources are computing workers, instead of cores in a single machine.

Usage: the percentage of CPU/GPU device time$T_{comp}=T_{exec}$.

Power Consumption (PC) in Wh of the workload. This metric is provided by OpenDC simulator [19]. It uses a linear model based on machine load [72], with an idle baseline of minimum power consumption and a maximum power draw as Table 5.3 shows.

5.3 Model Validation and Calibration

We validate and calibrate our simulation model in following steps:

1. We recreate the workload and environment from the description presented in peer-reviewed articles [51] [48] as describes in previous Section 5.2.
2. We iterate each experiment 50 times.Uta et al. [73, Figure 3 and Figure 13] nds 50 runs or more could achieve 95% con dence intervals (CIs) for performance measurements. Thus, we choose 50 as the number of repetitions, which is in line with that in peer-reviewed publications.
3. For each iteration, we compare the results with the articles.
4. We check if the trends shown by the model match the results in the articles and the execution time is within expected numerical ranges.
5. If we get an undesirable outcome, errors may occur in the workload and environment con guration, the system model, and the calculation of metrics. We analyze the undesirable result and review each component of the model and its parameters.
6. If we re ne our predictive model or the calculation of metrics, we should rerun all validation and calibration experiments.
7. When the outcomes of all validation and calibration experiments are expected, we regard our result as valid.

Figure 5.1: KTH calibration and validation experiment: compare with the execution time from the article [51, Figure 3]

Table 5.5: Time breakdown of KTH calibration and validation experiment.

Batch size	T_{comm}	T_{comp}	T_{mem}	T_{exec}
16	0 s	59.64 s	2.03 s	59.64 s
32	0 s	51.40 s	1.48 s	51.40 s
64	0 s	48.99 s	1.20 s	48.99 s
128	0 s	46.86 s	1.06 s	47.14 s

Results

Figure 5.1 shows our simulation results for capturing the execution time of the A1 workload with 16, 32, 64, 128 batch size compared to the runtime from the article [51, Figure 3]. Our estimation time is a bit higher than the actual time reported in the article. However, we still can see that there is a trend, which shows when the batch size of an application is higher, the execution time gets lower. Table 5.5 presents the time breakdown of the KTH calibration and validation experiment. The article does not provide more details on the application. We only can ensure the communication time is valid in this table because the application with the default strategy is executed in one device.

Figure 5.2 shows our simulation results for capturing the normalized execution time (NET) of Fathom workloads of AlexNet and VGG-19 compare to the runtime from the arti-

Table 5.6: Time breakdown of IBM calibration and validation experiment.

Model	Strategy	T_{comm}	T_{comp}	T_{mem}	T_{exec}
AlexNet	Default	0 s	190.60 s	11.95 s	199.00 s
	ParameterServer	3.84 s	66.80 s	1.12 s	70.64 s
VGG-19	Default	0 s	733.7 s	62.71 s	781.43 s
	ParameterServer	7.82 s	109.00 s	2.49 s	117.21 s

cle [48, Figure 7 and Table 2]. The normalized execution time is calculated as $NET = T_{estimated}/T_{default}$, where $T_{estimated}$ is the overall predictive execution time using predened strategy and $T_{default}$ is the overall predictive execution time of the application using default strategy. Figure 5.2 shows our estimated NET is lower than the actual NET report in the article. Table 5.6 presents the time breakdown of the IBM calibration and validation experiment. In this experiment, the application using the default strategy is executed on one CPU device, while the application using the parameter server strategy is running on one GPU device and CPU device. Thus, we can also ensure the communication time for the default strategy is zero. The AlexNet application using the parameter server strategy in this experiment uses 64 batch sizes and one GPU device as one worker. We can compare this result with the KTH experiment that uses 64 batch size and default strategy with one GPU device in Table 5.1. The time breakdown of these two applications is similar. The execution time of the application in this experiment is a little higher, because of the communication time and more iterations.

The articles do not provide more detailed information on the experiments. The reasons for our estimation inaccuracy of IBM experiments might be ourminimum form of network model. We do not model network overhead and the network delay is only based on the bandwidth and the data size of the message. Thus, the estimated communication time might be lower than the actual. Then, the execution time of the application using the parameter server strategy will be lower accordingly.

We not only validate our model with metrics but also inspect the running of our simulator step by step to ensure the functionalities (such as the communication model and execution model) are valid. Overall, although the prediction might be not so accurate, our model could produce several valid results.

Figure 5.2: IBM calibration and validation experiment: compare with the normalized execution time from the article [48, Figure 7 and Table 2].

5.4 Results and Analysis for New Experiments

In this section, we present and analyze the results of four new experiments: strategy comparison (Section 5.4.1), scalability study (Section 5.4.2), and cluster utilization study (Section 5.4.3).

5.4.1 Strategy Comparison

Figure 5.3 and Figure 5.4 presents the strategy comparison on the A1 workload in KTH environment and on the Fathom workloads in IBM Minsky environment. The applications with parameter server strategy and mirrored strategy use four GPU workers. We can see in Figure 5.3 the trend of batch size and execution time is similar to that in Figure 5.1. For the AlexNet applications in two experiments, the execution time or NET of the application using the mirrored strategy is lower than that using parameter strategy. The mirrored strategy is more e cient for the AlexNet application. In Figure 5.4, the VGG-19 application performs similar to parameter server strategy and mirrored strategy. A possible reason for this is that the VGG-19 application is computational-intensive. The computation time of VGG-19 accounts for a higher percent of execution time.

Figure 5.3: Strategy comparison on the A1 workload.

Figure 5.4: Strategy comparison on the Fathom workloads.

5.4.2 Scalability Study

Figure 5.5 and Figure 5.6 shows the scalability study on the A1 workload using mirrored strategy and Fathom workloads using mirrored strategy and parameter strategy, respectively. We can see that unusual super-linear speedup happens in the AlexNet application using a batch size of 128 and mirrored strategy with 8 workers (Figure 5.5) and AlexNet application using a batch size of 64 and mirrored strategy with more than 16 workers (Figure 5.6). We discuss the unusual super-linear results as following:

1. Yan et al. [43] have investigated the scalability results of ImageNet-22K. They observed that model parallelism speedup is super-linear, while data and parameter server parallelism speedups are roughly linear [43]. Thus, most of our results are roughly linear and mirrored strategy scale better than the parameter server strategy. These are reasonable. However, the super-linear results are abnormal.

2. The super-linear results happen in the application using a batch size of more than 64 and a mirrored strategy with more than 8 workers. Our application size is kept constant in this experiment. Adding more workers means each worker will get less computational jobs. We might drop some jobs because the jobs cannot be divided equally to each worker. Then, it will cause our computational time lower than expected.

3. The communication time might be not accurate. We have mentioned our network model is simple and when using mirrored strategy there is less communication overhead. Thus, some communication time might not be counted because of low precision.

5.4.3 Cluster Utilization Study

Table 5.7 presents the results of the total power consumption of the A1 workload. The power consumption, provided by OpenDC, uses a linear model based on machine load [72], with an idle baseline of minimum power consumption 90 Wh and maximum power draw 334 Wh as Table 5.3 shows. The power consumption varies signi cantly in a di erent setup. Compared to the default strategy with one worker, other settings have a higher power consumption. And we also can see that when batch size equals 64, the application consumes the lowest power compared to other batch sizes. It is hard to validate and explain these results in detail. However, the default batch size of AlexNet is 64, which means AlexNet with a batch size of 64 usually can get better performance. It is consistent with

Figure 5.5: Scalability study on the A1 workload.

Figure 5.6: Scalability study on the Fathom workload.

Table 5.7: Total power consumption.

Strategy	Batch size	# Workers			
		1	2	4	8
Default		4.24 kWh			
Parameter Server	16	240.9 kWh	241.0 kWh	241.1 kWh	241.5 kWh
	32	120.5 kWh	120.5 kWh	120.7 kWh	119.4 kWh
	64	60.2 kWh	60.3 kWh	59.7 kWh	58.3 kWh
	128	150.6 kWh	148.5 kWh	144.5 kWh	136.4 kWh
Mirrored	16	240.9 kWh	241.0 kWh	241.1 kWh	241.5 kWh
	32	120.5 kWh	120.5 kWh	120.7 kWh	119.4 kWh
	64	60.2 kWh	60.3 kWh	59.7 kWh	58.3 kWh
	128	150.6 kWh	148.5 kWh	144.5 kWh	136.4 kWh

our results. We conclude that our model enables us to estimate the power consumption of the TensorFlow ecosystem.

Table 5.8 presents the results of the total usage of the A1 workload. It shows when adding more workers, the device usage is degraded for parameter server strategy and almost remains the same for mirrored strategy. For the parameter server strategy, when there are more workers, the parameter server is much easier to be the bottleneck. All the workers have to communicate with the parameter server and update model parameters. However, the mirrored strategy overcomes this issue. Thus, the usage almost remains the same when increasing workers. We also can observe that when increasing the batch size, device usage is higher. This is because if the batch size of the application is larger, the size of a task will be larger and the number of tasks for each worker will be less. Then each worker will spend more time computing a task rather than communicating with other machines.

5.5 Discussion

In Chapter 5, we conduct simulation experiments to evaluate the performance of the TensorFlow ecosystem in HPC environments. We rst validate and calibrate our predictive model created in Chapter 4 through simulations set up to match the real-world experiments of Chien et al. [51] and Guignard et al. [48]. Then, we use our simulation model to evaluate the performance of the TensorFlow ecosystem in HPC environments. We focus on three aspects: strategy selection, scalability, and cluster utilization.

The key ndings of these experiments are as following:

Table 5.8: Total usage.

Strategy	Batch size	# Workers			
		1	2	4	8
Default		1.0			
Parameter Server	16	0.847	0.793	0.708	0.584
	32	0.905	0.869	0.808	0.707
	64	0.948	0.927	0.888	0.822
	128	0.965	0.954	0.932	0.893
Mirrored	16	0.856	0.857	0.857	0.856
	32	0.912	0.911	0.909	0.908
	64	0.951	0.952	0.951	0.949
	128	0.968	0.968	0.967	0.967

1. There exists a trend for a xed-size machine learning application that when the batch size is larger, the execution time is lower.
2. For AlexNet application, the mirrored strategy is more e cient than the parameter server strategy.
3. Mirrored strategy scales better than the parameter server strategy and its speedup is roughly linear.
4. Our model can estimate the power consumption of the TensorFlow ecosystem.
5. When adding more workers, the device usage is degraded for parameter server strategy and almost remains the same for mirrored strategy.

Threats to Validity

We identify several threats to validity for our experiments. First potential threat is that the abnormal super-linear speeduphappens in scalability study experiment. It means our model might not be accurate. We should gure out why it happens and x it in the future. Another potential threat is that our data management model is simpleWe do not present a detailed understanding of complex interactions between devices. The data management model is expected to be improved in the future.Reproducing experimentsis also a threat to validity. It is hard to make sure all the settings of the experiments are the same as that in the articles, including the setup of environments, the setup of workloads, and the calculation of metrics. Thus, there is much work on experiments left to be done.

6

Conclusion and Directions for Future Research

In this chapter, we conclude the main contributions of this thesis and propose several potential directions for future research.

6.1 Conclusion

In this thesis, we address three main research questions on understanding the behaviors and performance of Google TensorFlow ecosystem:

RQ1. How to design adeepreference architecture for the Google TensorFlow ecosystem?
In Chapter 3, we extend our reference architecture created in our literature survey. We add additional deeper layers for the TensorFlow programming model and execution engine. We identify more components and map to our reference architecture.

RQ2. How to create a predictive model of the Google TensorFlow ecosystem within a discrete-event simulator?
In Chapter 4, we create a multi-layer predictive model of the Google TensorFlow ecosystem within the OpenDC simulator. An application is modeled as a data ow computational graph. We design the execution model with three strategies (default strategy, parameter strategy, and mirrored strategy) which are currently supported by TensorFlow. We also create a simple communication model for data transfer.

RQ3. What is the performance of the Google TensorFlow ecosystem when running in High-Performance Computing (HPC) environments?

In Chapter 5, we design simulation experiments to evaluate the performance of the TensorFlow ecosystem in HPC environments. We rst validate and calibrate our predictive model through simulations set up to match the real-world experiments of Chien et al. [51] and Guignard et al. [48]. Then, we use our model to analyze the performance of the TensorFlow ecosystem in HPC environments. We focus on three aspects: strategy selection, scalability, and cluster utilization. The key ndings of these experiments are as following:

(a) There exists a trend for a xed-size machine learning application that when the batch size is larger, the execution time is lower.

(b) For AlexNet application, a mirrored strategy is more e cient than a parameter strategy.

(c) Mirrored strategy scales better than the parameter strategy and its speedup is roughly linear.

(d) Our model enables us to estimate the power consumption of the TensorFlow ecosystem.

(e) When adding more workers, the device usage is degraded for parameter server strategy and almost remains the same for mirrored strategy.

6.2 Directions for Future Research

In this section, we propose several potential directions for future research on understanding the behaviours of machine learning ecosystems in datacenters.

Improve data management model. In this work, we create a minimal form of network and storage model for data transfer. We do not present a detailed understanding of complex interactions between devices, including computational devices (such as CPUs and GPUs) and communication devices (such as switches and routers). Data management is a challenge for machine learning workloads [74]. It relates to loading training data from storage or memory, saving checkpoint les or model parameters data, moving data to other devices, and other data management-related problems. Chien et al. [51] nd data prefetching is a key factor to improve the performance of the input pipeline in TensorFlow. Chishti et al. [75] show memory hierarchy is becoming a performance bottleneck for scaling deep learning. To better capture the behaviors of data management in TensorFlow, future work could design a more detailed network, storage, and memory model. It not only could help

the prediction more accurate, but also could provide insights to optimize data systems for machine learning workloads.

Conduct real-world experiments to validate the model. In Chapter 5 Section 5.3, we validate and calibrate our model by reproducing two experiments from peer-reviewed articles. Reproducing experiments is a challenge [76]. It is hard to make sure all the settings of the experiments are the same as that in the articles, including the setup of environments, the setup of workloads, and the calculation of metrics. Thus, we consider to conduct real-world experiments and compare the results from real-world experiments with that from simulation to validate our model. Then, our model would be more reliable.

Explore scheduling policies in multi-tenant heterogeneous GPU clusters. Modern HPC environments host not only a single job, but rather the combined workloads of multiple concurrent tenants. Multi-tenant cluster scheduling for machine learning workloads has been studied by a number of recent works, such as Gandiva [64], Optimus [62], and Themis [66]. These schedulers, mentioned in Chapter 3 Section 3.2.3, target at di erent objectives such as high e ciency, fairness, cluster utilization, and resource isolation. Future research could focus on exploring resource scheduling and cluster management in multi-tenant heterogeneous GPU clusters.

Study how to provision HPC hardware for machine learning workloads. There are two opportunities for future research on how to provision HPC hardware for machine learning workloads. First, we could explore di erent types of hardware, including traditional hardware and ML/big-data hardware (e.g., SSD). Second, we could study capacity planning for machine learning workloads in HPC environments. Andreadis extends OpenDC and designs a capacity planning system, *Capelin*, for cloud infrastructure. Using Capelin tool to investigate machine learning workloads in HPC environments has the potential to help resource provisioning and planning.

Investigate the performance and energy e ciency of TensorFlow in HPC environments with failures. Failures could a ect cluster utilization and energy e ciency. Failures are caused by various reasons, such as runtime errors, network timeouts, incorrect inputs, job preempted or invalid memory access [54] [77]. In this work, we have studied the performance and power consumption of TensorFlow in HPC environments without failures. OpenDC has provided failure features and realistic energy models are being developed. It

is possible to extend this work to perform failure analysis of TensorFlow jobs in shared clusters. It could help to study the impact of failures on cluster usage and to improve failure handling ability for machine learning workloads.

References

[1] Li Deng and Xiao Li. Machine Learning Paradigms for Speech Recognition: An overview. IEEE Transactions on Audio, Speech, and Language Processing, 21(5):1060 1089, 2013. 1

[2] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep Learning in Medical Image Analysis. Annual Review of Biomedical Engineering, 19:221 248, 2017. 1

[3] Ivens Portugal, Paulo Alencar, and Donald Cowan. The Use of Machine Learning Algorithms in Recommender Systems: A Systematic Review. Expert Systems with Applications, 97:205 227, 2018. 1

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. ArXiv, abs/1604.07316, 2016. 1

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In 12th USENIX Symposium on Operating Systems Design and Implementation, pages 265 283, 2016. 1, 7, 15, 21, 22

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. ArXiv, abs/1603.04467, 2016. 1, 7, 29, 31

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems, pages 8024 8035, 2019. 1, 7

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and E cient Machine Learning Library for Heterogeneous Distributed Systems. ArXiv, abs/1512.01274, 2015. 1, 7

[9] Gerrit Muller. A Reference Architecture Primer. https://www.gaudisite.nl/ReferenceArchitecturePrimerSlides.pdf, August 21, 2020. 1

[10] Chris Treml and Brad Genereaux. ACR AI-LAB Hospital Reference Architecture Framework. https://resources.nvidia.com/redefining-medical-imaging/acr-ai-lab-hospital-architecture-ebook, 2019. 2

[11] Sagar Behere and Martin Törngren. A Functional Reference Architecture for Autonomous Driving. Information and Software Technology, 73:136 150, 2016. 2

[12] Kelvin Lui and Jeff Karmiol. AI Infrastructure Reference Architecture. https://www.ibm.com/downloads/cas/W1JQBNJV, June, 2018. IBM systems. 2

[13] Alexandru Iosup, Alexandru Uta, Laurens Versluis, Georgios Andreadis, Erwin Van Eyk, Tim Hegeman, Sacheendra Talluri, Vincent Van Beek, and Lucian Toader. Massivizing Computer Systems: a Vision to Understand, Design, and Engineer Computer Ecosystems through and beyond Modern Distributed Systems. In 38th IEEE International Conference on Distributed Computing Systems, pages 1224 1237, 2018. 2, 3, 5, 6

[14] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. The AtLarge Vision on the Design of Distributed Systems and Ecosystems. In 39th IEEE International Conference on Distributed Computing Systems, pages 1765 1776, 2019. 2, 3, 5, 6

[15] Paul A. Fishwick, editor. Handbook of Dynamic System Modeling. Chapman and Hall/CRC, 2007. 2

[16] R. H. A. Hellsen, Georgo Z. Angelis, M. J. G. van de Molengraft, A. G. de Jager, and J. J. Kok. Grey-box Modeling of Friction: An Experimental Case-study. Eur. J. Control, 6(3):258 267, 2000. 2

[17] Octavio Loyola-González. Black-Box vs. White-Box: Understanding Their Advantages and Weaknesses From a Practical Point of View. IEEE Access, 7:154096 154113, 2019. 2

[18] Martin Tarr. MODELLING-Choosing a model. http://www.ami.ac.uk/courses/topics/0199_mod/index.html, 2003. 2

[19] Alexandru Iosup, Georgios Andreadis, Vincent Van Beek, Matthijs Bijman, Erwin Van Eyk, Mihai Neacsu, Leon Overweel, Sacheendra Talluri, Laurens Versluis, and Maaike Visser. The OpenDC Vision: Towards Collaborative Datacenter Simulation and Exploration for Everybody. In 16th International Symposium on Parallel and Distributed Computing, pages 85 94, 2017. 3, 10, 11, 12, 26, 39

[20] Raj Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley & Sons, 1990. 3, 9

[21] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12(Oct):2825 2830, 2011. 7

[22] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine Learning in Apache Spark. The Journal of Machine Learning Research, 17(1):1235 1241, 2016. 7

[23] Shane Cook. CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Newnes, 2012. 7

[24] Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, 10:10, 2010. 7

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simpli ed Data Processing on Large Clusters. Communications of the ACM, 51(1):107 113, 2008. 7

[26] Apache Beam. https://beam.apache.org/documentation/, 2016. 7, 19

[27] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In Proceedings of the ACM SIGMOD International Conference on Management of data, pages 135 146, 2010. 8

[28] Oliver Ullrich and Daniel Lückerath. An Introduction to Discrete-Event Modeling and Simulation. Simul. Notes Eur., 27(1):9 16, 2017. 10

[29] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. Software: Practice and Experience, 41(1):23 50, 2011. 10, 12

[30] Alberto Núñez, Jose L Vázquez-Poletti, Agustin C Caminero, Gabriel G Castañé, Jesus Carretero, and Ignacio M Llorente. iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator. Journal of Grid Computing, 10(1):185 209, 2012. 10, 12

[31] Saurabh Kumar Garg and Rajkumar Buyya. NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations. In IEEE 4th International Conference on Utility and Cloud Computing, pages 105 113, 2011. 10, 12

[32] Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah Khalid Alham, and Zelong Liu. MRSim: A Discrete Event based MapReduce Simulator. In Seventh International Conference on Fuzzy Systems and Knowledge Discovery, 6, pages 2993 2997, 2010. 10, 12

[33] Wilson A Higashino, Miriam AM Capretz, and Luiz F Bittencourt. CEP-Sim: Modelling and Simulation of Complex Event Processing Systems in Cloud Environments. Future Generation Computer Systems, 65:122 139, 2016. 10, 12

[34] Xuezhi Zeng, Saurabh Kumar Garg, Peter Strazdins, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, and Rajiv Ranjan. IOTSim: A Simulator for Analysing IoT Applications. Journal of Systems Architecture, 72:93 107, 2017. 10, 12

[35] Paweª Ulita. A Survey of Simulators for Distributed Systems. unpublished. 10

[36] Prasad Jogalekar and C. Murray Woodside. Evaluating the Scalability of Distributed Systems. IEEE Transactions on Parallel and Distributed Systems, 11(6):589 603, 2000. 11

[37] Brian J. Watson, Manish Marwah, Daniel Gmach, Yuan Chen, Martin F. Arlitt, and Zhikui Wang. Probabilistic performance modeling of virtualized resource allocation. In Manish Parashar, Renato J. O. Figueiredo, and Emre Kiciman, editors, Proceedings of the 7th International Conference on Autonomic Computing, pages 99 108, 2010. 11

[38] Arnd Schröter, Gero Mühl, Samuel Kounev, Helge Parzyjegla, and Jan Richling. Stochastic performance analysis and capacity planning of publish/subscribe systems. In Jean Bacon, Peter R. Pietzuch, Joe Sventek, and Ugur Çetintemel, editors, Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, pages 258 269, 2010. 11

[39] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In Salim Hariri and Kate Keahey, editors, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pages 597 604, 2010. 11

[40] Emanuel Vianna, Giovanni Comarela, Tatiana Pontes, Jussara M. Almeida, Virgílio A. F. Almeida, Kevin Wilkinson, Harumi A. Kuno, and Umeshwar Dayal. Analytical Performance Models for MapReduce Workloads. Int. J. Parallel Program., 41(4):495 525, 2013. 11

[41] Daria Glushkova, Petar Jovanovic, and Alberto Abelló. MapReduce Performance Models for Hadoop 2.x. In Yannis E. Ioannidis, Julia Stoyanovich, and Giorgio Orsi, editors, Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference, 1810 of CEUR Workshop Proceedings. CEUR-WS.org, 2017. 11

[42] Gabriela Jacques-Silva, Zbigniew Kalbarczyk, Bugra Gedik, Henrique Andrade, Kun-Lung Wu, and Ravishankar K. Iyer. Modeling stream processing applications for dependability evaluation. In Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, pages 430 441. IEEE Compute Society, 2011. 11

[43] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul M. Chilimbi. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1355 1364, 2015. 11, 44

[44] Sayed Hadi Hashemi, Shadi A. Noghabi, William Gropp, and Roy H. Campbell. Performance Modeling of Distributed Deep Neural Networks. CoRR, abs/1612.00521, 2016. 11

[45] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. In IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, pages 949 957, 2018. 11

[46] Salem Alqahtani and Murat Demirbas. Performance Analysis and Comparison of Distributed Machine Learning Systems. CoRR, abs/1909.02061, 2019. 11

[47] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A Performance Model for Deep Neural Networks. In 5th International Conference on Learning Representations, 2017. 11

[48] Mauricio Guignard, Marcelo Schild, Carlos S. Bederián, Nicolás Wolovick, and Augusto J. Vega. Performance Characterization of State-Of-The-Art Deep Learning Workloads on an IBM "Minsky" Platform. In

51st Hawaii International Conference on System Sciences, pages 1 8, 2018. 13, 35, 36, 38, 39, 41, 42, 46, 50

[49] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia . Characterizing Deep Learning Training Workloads on Alibaba-PAI . In IEEE International Symposium on Workload Characterization, pages 189 202, 2019. 13

[50] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David M. Brooks . Fathom: reference workloads for modern deep learning methods . In IEEE International Symposium on Workload Characterization, pages 148 157. IEEE Computer Society, 2016. 13, 36, 37

[51] Steven Wei Der Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luís Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure . Characterizing Deep-Learning I/O Workloads in TensorFlow . In 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, pages 54 63, 2018. 13, 35, 36, 37, 38, 39, 40, 46, 50

[52] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications . arXiv preprint arXiv:1811.09886, 2018. 13

[53] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang . Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective . In IEEE International Symposium on High Performance Computer Architecture, pages 620 629. IEEE Computer Society, 2018. 13

[54] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang . Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications . Technical report, Microsoft Research, 2018. 13, 51

[55] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017. 17, 18

[56] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015. 17

[57] Heng-Tze Cheng, Zakaria Haque, Lichan Hong, Mustafa Ispir, Clemens Mewald, Illia Polosukhin, Georgios Roumpos, D Sculley, Jamie Smith, David Soergel, et al. TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1763–1771, 2017. 18

[58] Peng Gu. Memory management for tensorflow. https://github.com/miglopst/cs263_spring2018/wiki/Memory-management-for-tensorflow, May 4, 2018. 20

[59] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 18:1–18:17. ACM, 2015. 20

[60] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013. 20

[61] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 11, page 22, 2011. 20

[62] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an E cient Dynamic Resource Scheduler for Deep Learning Clusters. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference*, pages 3:1–3:14. ACM, 2018. 20, 51

[63] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation*, pages 485–500, 2019. 20

[64] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 595–610, 2018. 20, 51

[65] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404. ACM, 2017. 20

[66] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. THEMIS: Fair and E cient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*, pages 289–304, 2020. 20, 51

[67] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, 2010. 21

[68] Rasmus Munk Larsen and Tatiana Shpeisman. TensorFlow Graph Optimizations, 2019. 22

[69] Erwin Van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst,

Cristina Abad, and Alexandru Iosup. The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms. *IEEE Internet Computing*, 2019. 23

[70] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. A Reference Architecture for Datacenter Scheduling: Design, Validation, and Experiments. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 478–492, 2018. 27

[71] Ruben Mayer, Christian Mayer, and Larissa Laich. The TensorFlow Partitioning and Scheduling Problem: It's the Critical Path! *CoRR*, abs/1711.01912, 2017. 31

[72] Mark Blackburn and Green Grid. Five Ways to Reduce Data Center Server Power Consumption. *The Green Grid*, 42:12, 2008. 39, 44

[73] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan S. Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is Big Data Performance Reproducible in Modern Cloud Networks? In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation*, pages 513–527. USENIX Association, 2020. 39

[74] Arun Kumar, Matthias Boehm, and Jun Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722, 2017. 50

[75] Zeshan Chishti and Berkin Akin. Memory system characterization of deep learning workloads. In *Proceedings of the International Symposium on Memory Systems*, pages 497–505. ACM, 2019. 50

[76] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, José Nelson Amaral, Petr Tuma, and Alexandru Iosup. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. In Michael Felderer, Wilhelm Hasselbring, Rick Rabiser,

and Reiner Jung, editors, *Software Engineerin, Fachtagung des GI-Fachbereichs Softwaretechnik*, P-300 of *LNI*, pages 93–94. Gesellschaft für Informatik e.V., 2020. 51

[77] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference*. 51