The Thesis Committee for **Snehil Verma**
certifies that this is the approved version of the following thesis:

Deep Learning Training at Scale: Experiments with MLPerf on Multi-GPU and Multi-TPU Hardware

APPROVED BY

SUPERVISING COMMITTEE:

Lizy Kurian John, Supervisor

Mattan Erez

Deep Learning Training at Scale: Experiments with MLPerf on Multi-GPU and Multi-TPU Hardware

by

Snehil Verma

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

Dedicated to my parents.

Acknowledgments

First and foremost, I wish to thank my advisor Prof. Lizy K. John for nurturing me in a unique field at the intersection of Computer Architecture and Machine Learning. I would also like to extend my gratitude to fellow LCA lab members, especially, Qinzhe Wu and Bagus Hanindhito, and other collaborators: Gunjan Jha, Prof. Eugene B.John, and Dr. Ramesh Radhakrishnan. Most of all, I am forever grateful to my Mom and Dad for their endless love and unwavering support to all my decisions in life.

This research was supported in part by the National Science Foundation under grant numbers 1725743, 1745813, and 1763848. I would also like to acknowledge computational resources from Texas Advanced Computing Center (TACC), Dell EMC Corporation, Google Cloud, and TensorFlow Research Cloud (TFRC) program. Any opinions, findings, conclusions or recommendations are those of the author and not of the National Science Foundation or other sponsors.

Deep Learning Training at Scale: Experiments with MLPerf on Multi-GPU and Multi-TPU Hardware

Snehil Verma, M.S.E.
The University of Texas at Austin, 2020

Supervisor: Lizy Kurian John

Training deep learning (DL) models is a highly compute-intensive task since it involves operating on massive datasets and tuning weights until the model meets the desired accuracy. Compute clusters paired with deep learning accelerators are typically employed in training complex DL models to reduce the training time and achieve the desired accuracy. MLPerf, an emerging machine learning benchmark suite, strives to cover a broad range of machine learning applications. Utilizing the training workloads from the MLPerf benchmark suite, this thesis studies their behavior on industry-grade multi-GPU (on-premise) and multi-TPU (cloud) hardware. The training suite of MLPerf contains a diverse set of models that allows unveiling various bottlenecks in training hardware. Based on the findings, dedicated low latency interconnect between GPUs in multi-GPU systems is crucial for optimal distributed deep learning training. Significant variation in scaling efficiency between various MLPerf training benchmarks (ranging from 2.3× to 7.8× on an 8-GPU cluster and 1.1× to 9.2× on an 8-TPU cluster) is also observed. The variation exhibited by the different models highlight the importance of smart scheduling strategies for distributed training. A speedup of up to 1.7× is seen on using TPU v3 over TPU v2. Furthermore, host CPU utilization increases with an increase in the number of GPUs or TPUs used for training, suggesting the need for powerful CPUs. Corroborating prior work, improvements possible by compiler optimizations and mixed-precision training using Tensor Cores on the GPUs are also quantified. Similarly, the performance gain on using the bfloat16 data type on multi-TPU runs is also highlighted in this work.

In addition, a study on the characteristics of MLPerf training benchmarks and how they differ from previous deep learning benchmarks such as DAWNBench and DeepBench is also presented. MLPerf benchmarks are seen to exhibit moderately high memory transactions per second and moderately high compute rates, while DAWNBench creates a high-compute benchmark with low memory transaction rate, and DeepBench provides low compute rate benchmarks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The recent advances in machine learning have led to an evolution of a myriad of applications, revolutionizing scientific, industrial, and commercial fields. Machine learning, primarily deep learning, is the state-of-the-art in providing models, methods, tools, and techniques for developing autonomous and intelligent systems.

There are two parts to machine learning: training and inference. Training refers to the process where the neural network learns a new capability based on existing data. While, inference utilizes the capabilities of a trained neural network to make useful predictions. Among the two, training is the long-running task. This is not only because of the massive datasets needed for high accuracy but also because the weights in the neural network need to be iteratively tuned until the model meets the desired quality. As the system's compute power plays a significant role in how fast the neural network learns, training is usually done using high-performance compute clusters attached with deep learning accelerators like GPUs and TPUs. On the other hand, inference is usually performed inside the end-user hardware, such as edge devices, where energy efficiency is also an important design consideration.

1.1 Upbringing of the MLPerf benchmarks

Evaluation of training capability necessitates benchmarks that encompass the training requirements of modern DL models from different domains. MLPerf [3] is an emerging consortium that provides separate benchmark suites for machine learning training and inference. The training suite helps to measure the performance of machine learning frameworks, hardware accelerators, and cloud platforms [18, 29, 56]. The major contributors to the benchmarks include Google, NVIDIA, Baidu, Intel, and other commercial vendors, as well as universities such as Harvard, Stanford, and the University of California, Berkeley. MLPerf's initial release `v0.5` in 2018 consisted of benchmarks only for training, but inference benchmarks were added in June 2019. MLPerf training benchmark suite covers the areas of computer vision, product recommendation, and other key areas where deep learning models have shown success, and the datasets are available publicly. This thesis solely focuses on the MLPerf training benchmark suite.

Young [56] rightly pointed out five main attributes that a good machine learning benchmark suite should possess, grouped together as five "R"s. One of them was *Representative workloads*, with regards to which he wrote:

> *A good benchmark suite is both diverse and representative, where each workload in the suite has unique attributes and the suite collectively covers a large fraction of the application space.*

In a talk [28] at FastPath, ISPASS-19 on "MLPerf design challenges", Mattson highlighted that the current set of training benchmarks cover a wide range of applications. Later in the year 2019, whitepapers on MLPerf training [29] and inference [39] benchmarks were also made available on the arXiv.

1.2 Thesis Contributions

This thesis[1] evaluates the MLPerf training benchmarks with experiments on diverse hardware platforms. Specifically, deep learning accelerator (GPU and TPU) based systems are studied. Note that GPUs have evolved over generations to support the acceleration of deep learning applications, while TPUs are specifically designed for this purpose. Additionally, this thesis investigates whether the execution characteristics of these benchmarks point out sufficient dissimilarities, or they are mostly similar in spite of diverse domains. The objective of this work is to unfold the answers to following enigmas:

- How well does the training performance scale with increasing the number of GPUs or TPUs? Is there a point beyond which increasing the number of accelerators[2] is not rewarding enough?
- What is an efficient way for a user to operate on a multiple-accelerator system to train several models simultaneously: should they run dis-

[1]Some contents of this thesis are hosted on-line at arXiv [47] and some are published at the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) in 2020 [48]. I am the principal author of the aforementioned works. Part of the same is also presented at NVIDIA's GPU Technology Conference (GTC) 2019 [37].

[2]One GPU device (referred to as GPU) and one TPU core (referred to as TPU) is considered as an accelerator.

tributed jobs one-by-one on all accelerators, should they run jobs assigning one model to each accelerator, or is there any other better solution?

- How well are the CPUs, accelerators, and interconnects utilized? Especially for a multi-GPU system, is there a significant performance impact from the high-bandwidth GPU interconnects like NVLink?
- What is the performance differential that can be obtained by mixed precision training exploited by NVIDIA's tensor cores or by Google's TPUs using their native data type, bfloat16?
- How much speedup is obtained on using the newer version of the TPUs (v3) when compared to the older version (v2)?
- How different are the MLPerf benchmarks from the prior deep learning benchmarks? How different are the MLPerf benchmarks from each other? If the hardware designers do not have time budget and resources to evaluate all the benchmarks, can they use a subset of the benchmarks?

This thesis presents many key observations, as summarized in the left column of Table 1.1. These observations lead to significant architectural and resource scheduling implications, as listed on the right side of Table 1.1.

1.3 Thesis Organization

The thesis is organized as follows: Chapter 2 introduces the emerging MLPerf [3] benchmark suite as well as some prior deep learning benchmarks

Table 1.1: Summary of key insights from the work.

Observation	**Proof**	**Insight/Explanation**
Every benchmark in MLPerf benchmark suite is on the boundary of the workload space.	Fig. 3.2b	There is a great diversity existing in MLPerf benchmark suite, e.g., in terms of the scaling efficiency. This information is helpful for resource scheduling in systems with multiple devices, such as data centers and cloud platforms.
Different benchmarks scale up differently, and by exploiting these differences, the optimal scheduling can save hours of training on multi-GPU and multi-TPU systems.	Tab. 3.2 Fig. 3.7 Tab. 4.3	
Data points representing machine learning workloads are close to the slanted roof line.	Fig. 3.4	It's easy to exploit the abundant parallelism in ML applications and finally end up being bound by hardware resources.
Mixed precision in combination with Tensor Cores earns significant speedup on MLPerf.	Fig. 3.5	Hardware support for reduced precision arithmetic is important, especially for machine learning workloads.
Performance gain is observed when multiplying in bfloat16 on TPU's Matrix Units.	Tab. 4.4	
With XLA enabled, Res50_TF converges to the same accuracy as no XLA, while the time reduces by 40%	Fig. 3.6	Compiler optimizations, especially kernel fusion, provides for a lot of potential for performance improvement.

(e.g., DAWNBench [11] and DeepBench [7]). Chapter 3 expands on the analysis performed on the systems with GPUs. It includes the configurations and topologies, on which various experiments are performed. It presents measurements on system scalability and resource utilization to provide insights on CPU's, GPU's, and interconnect's impact on machine learning training performance as well as the memory requirement to store the dataset during processing. This chapter also presents the performance impact of compiler

Table 1.1: Summary of key insights from the work (continued).

Observation	**Proof**	**Insight/Explanation**
When scaling to more GPUs, many benchmarks have a super-linear increase in PCIe / NVLink utilization.	Tab. 3.3	Machine learning applications can become communication-heavy workloads, so it is worth paying attention to the buses in ML processor designs. Direct connections between GPUs facilitate better performance in machine learning workloads.
Training time: GPU-system with NVLink enabled $<$ GPU-system with PCIe switch enabled $<$ system with GPUs connected using CPU PCIe ports.	Tab. 3.1 Fig. 3.1 Fig. 3.8	
MLPerf benchmark suite has a disjoint envelope from DAWNBench and DeepBench.	Fig. 3.2a	MLPerf, DAWNBench, and DeepBench suite stress HBM2 memory at different levels, and are optimized to different extents. Throughput and arithmetic intensity: DAWNBench $>$ MLPerf $>$ DeepBench.
DeepBench, MLPerf, and DAWNBench are located in different regions in the roofline graph.	Fig. 3.4	

optimizations and mixed precision training, supported by Tensor Cores. It also examines various benchmark characteristics and presents the similarity of various benchmarks. Chapter 4 showcases a similar analysis but using systems enabled with TPUs. It encapsulates the methodology of the experimentation and studies the trends for system scalability, TPU utilization, and host-side activity. It also quantifies the performance gains on training using bfloat16 compared to float32 data type and v3 TPUs compared to v2 TPUs. Chapter 5 concludes the thesis with a summary of the contributions.

Chapter 2

Background

This chapter introduces MLPerf (Training) [3, 29], DAWNBench [11], and DeepBench [7] benchmarks for machine learning. With research in the field of deep learning, various other benchmarks have also appeared in the past, such as Fathom [5], Training Benchmark for DNNs (TBD) [57], etc., but this thesis is focused on studying MLPerf, DAWNBench, and DeepBench.

2.1 MLPerf Training Benchmarks

The MLPerf [3] benchmark suite includes workloads from image classification, object detection, translation, recommendation, and reinforcement learning. It aims is to accelerate progress in machine learning via fair and suitable measurement and enforce replicability to ensure reliable results. It enables fair comparison of competing systems yet encourage innovation to push the limits of ML, meanwhile keeping the benchmarking effort affordable so all can participate. In order to do so, MLPerf takes two approaches: closed model division and open model division. The MLPerf closed model division postulates the model to be used and restricts the values of hyperparameters, such as batch size and learning rate, with the emphasis on fair comparisons of the hardware and software systems. On the contrary, in the open model division, the same problem is required to be solved using the same dataset but with fewer restrictions, with the emphasis on advancing the state-of-the-art of ML [3].

Table 2.1 displays a summary of the various workloads of MLPerf `v0.5` release, including respective models as well as the datasets used. MLPerf uses the time taken to reach a specified accuracy or quality target as the metric for evaluation and comparison, which is also listed in Table 2.1 for each benchmark. MLPerf benchmark implementations provided by the submitters currently include frameworks such as PyTorch [36], MXNet [9] and TensorFlow [4]. Many of the workloads consume days of training time on powerful GPUs, as shown in Figure 2.1, for MLPerf's reference machine that has an NVIDIA Tesla P100 GPU.

2.1.1 Image Classification

Image Classification is a typical deep learning application that identifies the object classes present in the image. This benchmark uses ResNet-50 [20,21] model. ResNet-50 signifies a 50-layered residual network, which effectively overcomes the problem of degradation of training accuracy with depth and is easier to optimize.

Table 2.1: Summary of benchmarks in MLPerf (top), DAWNBench (middle), and DeepBench (bottom) used in this study. MLPerf benchmarks are from `v0.5` suite unless specified otherwise.

MLPerf Training Benchmark					
Abbreviation	**Domain**	**Model**	**Framework (Submitter)**	**Dataset**	**Quality Target**
MLPf_Res50_TF	Image Classification	ResNet-50	TensorFlow (Google)	ImageNet	Accuracy: 0.749
MLPf_Res50_MX			MXNet (NVIDIA)		
MLPf_SSD_TF (v0.6)	Object Detection	SSD (light-weight)	TensorFlow (Google)	Microsoft COCO	mAP: 0.23
MLPf_SSD_Py			PyTorch (NVIDIA)		mAP: 0.212
MLPf_MRCNN_TF (v0.6)		Mask RCNN (heavy-weight)	TensorFlow (Google)		Box mAP: 0.377, Mask mAP: 0.339
MLPf_MRCNN_Py			PyTorch (NVIDIA)		
MLPf_XFMR_Py	Translation	Transformer	PyTorch (NVIDIA)	WMT17	BLEU score (uncased): 25
MLPf_GNMT_TF (v0.6)		RNN GNMT	TensorFlow (Google)		Sacre BLEU score (uncased): 24.0
MLPf_GNMT_Py			PyTorch (NVIDIA)		Sacre BLEU score (uncased): 21.80
MLPf_NCF_Py	Recommendation	Neural Collaborative Filtering	PyTorch (NVIDIA)	MovieLens 20-million	Hit rate @ 10: 0.635

DAWNBench					
Abbreviation	**Domain**	**Model**	**Framework (Submitter)**	**Dataset**	**Quality Target**
Dawn_Res18_Py	Image Classification	ResNet-18 (modified)	PyTorch (bkj)	CIFAR10	Test accuracy: 94%
Dawn_DrQA_Py	Question Answering	DrQA	PyTorch (Yang et al.)	SQuAD	F1 score: 0.75

DeepBench				
Abbreviation	**Operation**	**Parameters**		**Targeted Application**
Deep_GEMM_Cu	Dense Matrix Multiply	all specified in the repository		N/A
Deep_Conv_Cu	Convolution	all specified in the repository		N/A
Deep_RNN_Cu	Vanilla Recurrent	Units=1760	N=16	DeepSpeech
	GRU Recurrent	Units=2816	N=32	
	GRU Recurrent	Units=1024	N=32	Speaker ID
	LSTM Recurrent	Input=512	N=16	Machine Translation
	LSTM Recurrent	Input=4096	N=16	Language Modeling
	LSTM Recurrent	Input=256	N=16	Character Language Modeling
Deep_Red_Cu	Communication (AllReduce)	all specified in the repository		N/A

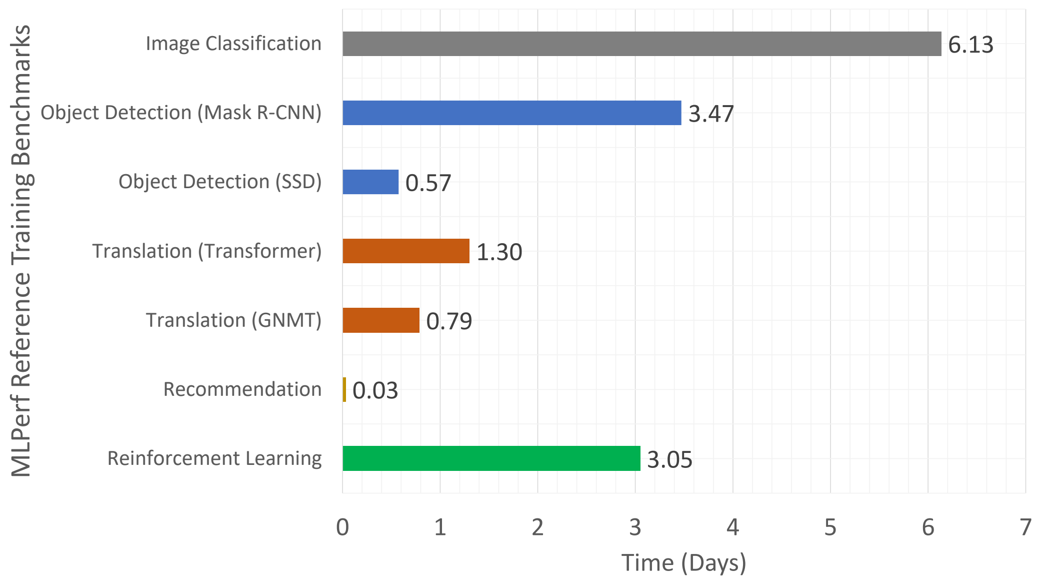


Figure 2.1: Training time of MLPerf reference implementations of the Training suite on MLPerf's reference machine (consisting one NVIDIA Tesla P100 GPU).

2.1.2 Object Detection

Object Detection is a technology that classifies individual objects and localizes each using a bounding box. MLPerf's training benchmark suite includes two models in this domain:

2.1.2.1 Heavy-weight: Mask R-CNN

Mask R-CNN [19] adds a branch for predicting segmentation masks on each Region of Interest (RoI), along with the existing branch for classification and bounding box regression. In Mask R-CNN, the additional mask output is distinct from the class and box outputs, as it extracts a finer spatial layout of an object.

2.1.2.2 Light-weight: SSD

Single Shot Detection (SSD) [27] discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. The SSD model completely eliminates proposal generation and subsequent pixel or feature resampling stage and encapsulates all computation in a single network. This makes SSD easy to train and integrate into systems that require a detection component.

2.1.3 Translation

Translation is the task of converting an input text from one language to another. There are two models for Translation included in the MLPerf Training benchmark suite:

2.1.3.1 Non-recurrent: Transformer

The model architecture - Transformer [46], avoids recurrence and relies on an attention mechanism to generate global dependencies between input and output. The attention weights apply to all symbols in the sequences.

2.1.3.2 Recurrent: GNMT

Google's Neural Machine Translation system (GNMT) [53] model uses residual connections as well as attention connections. GNMT provides a decent balance between the flexibility of “character”-delimited models and the efficiency of “word”-delimited models, and handles translation of rare words.

2.1.4 Recommendation

Recommendation is a task accomplished by a recommendation system that predicts the “rating” or “preference” to an item. This benchmark uses Neural Collaborative Filtering model (NCF) [22] that can express and generalize matrix factorization under its framework. In order to supercharge NCF modeling with non-linearities, a multi-layer perceptron can be utilized in this model to learn the user-item interaction function.

2.1.5 Reinforcement Learning

Reinforcement Learning is associated with how software agents should take actions in an environment to maximize the notion of cumulative reward. This benchmark[1] is based on a fork of the mini-go project [1], inspired by DeepMind’s AlphaGo algorithm [40, 42]. There are four phases in this benchmark, repeated in order: selfplay, training, target evaluation, and model evaluation. Moreover, this architecture is also extended for Chess and Shogi [41].

[1]The focus of evaluation is on MLPerf `v0.5` on GPU platforms and `v0.6` on TPU platforms. The only GPU code of *Reinforcement Learning* is the reference one, which spends more time on the CPU than the GPU, and there is no implementation available for the TPU. Hence, *Reinforcement Learning* is excluded from the rest of the work.

2.2 DAWNBench

DAWNBench [11], developed by Stanford University in 2017, evaluates deep learning systems across different optimization strategies, model architectures, software frameworks, clouds, and hardware. It supports benchmarking of *Image Classification* on CIFAR10 [25] and ImageNet [13], and *Question Answering* on SQuAD [38]. DAWNBench assesses the performance based on four metrics: training time to a specified validation accuracy, cost (in USD) of training, average latency of performing inference, and the cost (in USD) of inference. It provides reference implementations and seed entries, implemented in two popular deep learning frameworks: PyTorch [36] and TensorFlow [4]. The hyperparameters that DAWNBench considers for optimizations are optimizer for gradient descent, minibatch size, and regularization.

2.3 DeepBench

DeepBench, released in 2016 [7], and updated in 2017 [6], primarily uses the neural network libraries to benchmark the performance of basic operations on different hardware. The performance characteristics of models built for various applications are different from each other. DeepBench essentially benchmarks the underlying operations such as dense matrix multiplication, convolutions, recurrent layers, and communication. For training, DeepBench specifies the minimum precision requirements as 16 and 32 bits for multiplication and addition, respectively [7]. The benchmarks are written in CUDA and, thus, are more fundamental than any deep learning framework or model implementation. Additionally, there is no concept of a quality target.

Chapter 3

Experiments on GPUs

This chapter presents the analysis performed on multi-GPU systems. First, Section 3.1 describes the system configurations, benchmarks, and tools that were used, followed by a characterization of the benchmarks in Section 3.2. Finally, Section 3.3 showcases various insights from studying, for example, performance gain using mixed-precision training, scalability of the benchmarks, system utilization trends, and performance variation on systems with different topologies.

3.1 Methodology

3.1.1 System configurations

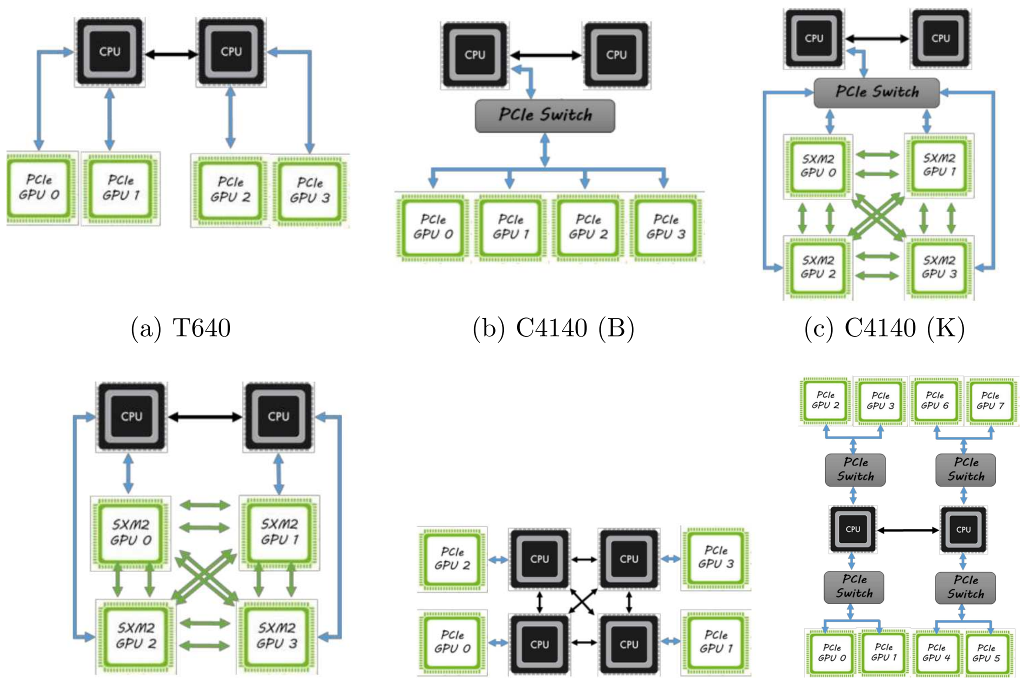
Various multi-GPU system configurations are used for experimentation in this chapter. Hardware specifications for the same are highlighted in Table 3.1 and topologies are shown in Figure 3.1. All the systems, except C4140 (B), operate on Ubuntu 16.04.4 LTS. The operating system on C4140 (B) is CentOS Linux 7.

3.1.2 Benchmarks

The benchmarks chosen to conduct research on are:

Table 3.1: Hardware specifications of Dell PowerEdge multi-GPU systems for experimentation. (UPI - Ultra Path Interconnect)

Systems	**T640**	**C4140 (B)**	**C4140 (K)**	**C4140 (M)**	**R940 XA**	**DSS 8440**
CPUs (**Intel Xeon Gold**)						
Model #	6148	6148	6148	6148	6148	6142
Base freq.	2.40GHz	2.40GHz	2.40GHz	2.40GHz	2.40GHz	2.60GHz
Memory (**Samsung/Micron DDR4**)						
# DIMM	12	12	12	24	24	12
Size	16GB	16GB	16GB	16GB	16GB	32GB
GPUs (**NVIDIA Tesla V100**)						
Form Factor	PCIe Full Height/ Length	PCIe Full Height/ Length	SXM2	SXM2	PCIe Full Height/ Length	PCIe Full Height/ Length
Inter-connect	PCIe & UPI	PCIe	NVLink	NVLink	UPI	PCIe & UPI
# GPUs	4	4	4	4	4	8
Memory	32GB HBM2	16GB HBM2	16GB HBM2	16GB HBM2	32GB HBM2	16GB HBM2



(a) T640 (b) C4140 (B) (c) C4140 (K)

(d) C4140 (M) (e) R940 XA (f) DSS 8440

Figure 3.1: Multi-GPU system topologies.

- GPU submissions of the **MLPerf** [3] `v0.5` training benchmarks, which are made by Google (cloud) and NVIDIA (on-premise). The submitted source codes were optimized for performance on their respective hardware. Among the various submissions, Google's submission on `8x Volta V100` and NVIDIA's submission on `DGX-1` are picked as I had access to platforms with a maximum of 8 GPUs. Note that, as there was no GPU submission for *Reinforcement Learning* benchmark (one of the MLPerf training benchmarks), this benchmark is excluded from the study.

- From **DAWNBench** [11], for *Image Classification (CIFAR10)* training, ResNet-18 implementation [8] provided by `bkj` is selected, and for *Question Answering (SQuAD)* training, the DrQA implementation [55] submitted by Yang et al. is chosen.

- In the case of **DeepBench** [7], four NVIDIA training benchmarks: `gemm_bench`, `conv_bench`, `rnn_bench`, and `nccl_single_all_reduce` are used. The MPI version of `all_reduce` is omitted as training on different nodes is not the focus of this work. The aggregated numbers are used for all the kernels with different sizes, except for `rnn_bench`, for which only six configurations are chosen because of long profiling time taken by the benchmark.

Note that the hyperparameters like batch size and learning rate are scaled accordingly to ensure that the run[1] completed successfully on the ex-

[1] "A run is a complete execution of an implementation on a system, training a model

perimental setup.

3.1.3 Measurement tools

3.1.3.1 nvprof

The `nvprof` profiler from CUDA-toolkit is used to profile the Region of Interest (ROI) in the benchmarks. Information collected are: invocation and duration of kernels, floating-point operation counts, and memory read/write transactions. This information is used to add data points as the representatives of machine learning workloads to the roofline plot.

3.1.3.2 dstat

`dstat` [50] is used to obtain real-time statistics of the system resource usage such as CPU usage, memory usage, disk activity, and network traffic. In UNIX platform, `dstat` gives more flexibility that combines `vmstat` (virtual memory statistics) [45], `iostat` (storage input/output statistics) [43], and `netstat` (network statistics) [44]. The statistics are then exported to comma-separated values for further analysis. Moreover, the functionality of `dstat` can be extended by adding plugins such as one to measure NVIDIA GPU Utilization [49].

from initialization to the quality target." - MLPerf [3]

3.1.3.3 dmon

`dmon`, which is available in NVIDIA System Management Interface (`nvidia-smi`) [35], is utilized to get individual GPU usage statistics that include GPU streaming multiprocessor usage, GPU memory usage, temperature, frequency, and PCI Express bus usage. A feature to measure the NVLink bus utilization using hardware counters is also employed in `nvidia-smi`.

3.2 Benchmark Comparison

The experiments and analysis in this thesis are based mainly on the MLPerf benchmarks because it has the most active community and is backed by many companies as well as academic institutions. On the other hand, several other deep learning benchmarks were proposed in the past. It is worth knowing the distinction of MLPerf benchmarks from the prior benchmarks and the contrast within the suite, so this section[2] presents a characterization on the benchmarks from MLPerf, DAWNBench, and DeepBench.

3.2.1 Similarity/Dissimilarity analysis

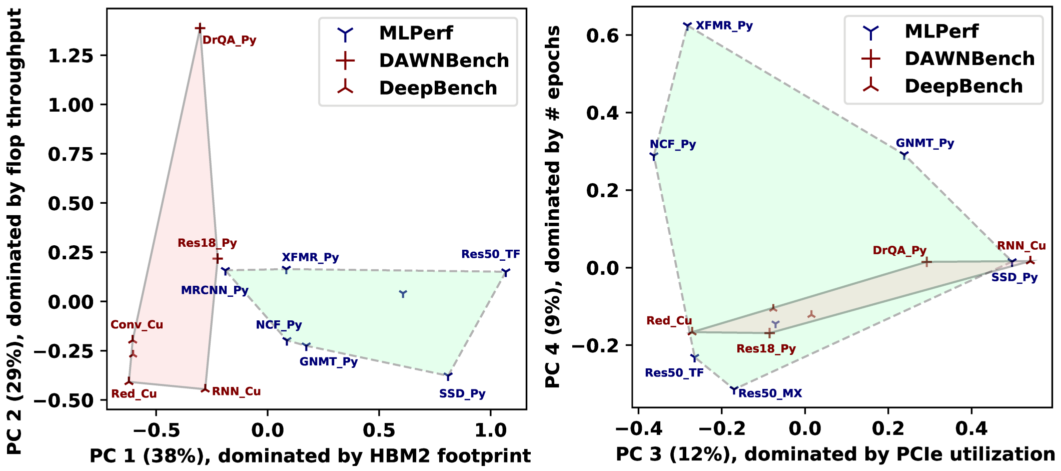
Principal Component Analysis (PCA) is performed on 8 collected workload characteristics (namely, PCIe utilization, GPU utilization, CPU utilization, DDR memory footprint, HBM2 footprint, flop throughput, memory throughput, and number of epochs), and the distribution of the targeted machine

[2]Co-investigated with co-authors of Verma et al. (2020) [48]. Included for completeness with approval from the co-authors.

learning benchmarks is visualized in the workload space. This analysis helps in understanding how similar and different these benchmarks are. In addition, a dendrogram is generated in order to help users pick the most representative benchmarks of a certain number according to their time budget and available resources.

As shown in Figure 3.2a, MLPerf benchmarks are so different from DeepBench kernels as well as DAWNBench benchmarks on PC1, that they become two isolated clusters (with outliers labeled) sitting on two sides. PC1 is dominated by GPU memory footprint. The location in the space is actually a reflection of the fact that DeepBench kernels and DAWNBench benchmarks are working on relatively smaller datasets, and they cannot stress GPU memory as much as MLPerf benchmarks can. On the PC2 axis, MLPerf benchmarks have a shorter span than other benchmarks do, mainly because MLPerf benchmarks are optimized end-to-end applications, having a stable floating point operation throughput, while more diversity exists in the other benchmarks (e.g., the communication kernel Deep_Red_Cu even has zero floating-point operations). MLPerf benchmarks are more sparsely-spread on the PC3-PC4 plane (Figure 3.2b), and cover what other benchmarks cover. The intra-suite diversity is exposed in Figure 3.2 as well. For PC1 to PC4 (covering 88% variance), each MLPerf benchmark gets at least one chance to extend the boundary, and there are no two MLPerf benchmarks that are very close to each other.

The dendrogram shown in Figure 3.3 presents the result of linkage-distance-based hierarchical clustering, where each benchmark starts as a leaf



(a) PC1 - PC2 (b) PC3 - PC4

Figure 3.2: The distribution of MLPerf, DAWNBench, and DeepBench in the dominant principal component workload space. The dominant metric is the one with the greatest absolute value in the eigenvector of a principal component.

node, then the two benchmarks closest to each other (i.e., most similar) are linked first, for instance, MLPf_Res50_TF and MLPf_Res50_MX. A dendrogram is more useful than just presenting the similarities between benchmarks; it facilitates the benchmarks selections for users who do not want or cannot run all the benchmarks due to time or cost limitation. For example, in Figure 3.3, the dashed line crossing 4 vertical lines filters out 4 most representative subsets for people can only evaluate with 4 benchmarks. The user is supposed to use Dawn_DrQA_Py, MLPf_SSD_Py, one of MLPf_Res50_TF and MLPf_Res50-_MX (take MLPf_Res50_TF for example), and another from the purple group (with all the benchmarks left, take Deep_Red_Cu for example). As a validation for the subsetting, the range of 8 metrics covered by the 4 selected benchmarks with respect to all is reported: PCIe utilization 0.3%˜100%, GPU utilization 0%˜95.6%, CPU utilization 0%˜100%, DDR memory footprint 0%˜100%, HBM2 footprint 0%˜100%, flop throughput 0%˜100%, memory throughput 8.0%˜100%, and number of epochs 0%˜98.4%.

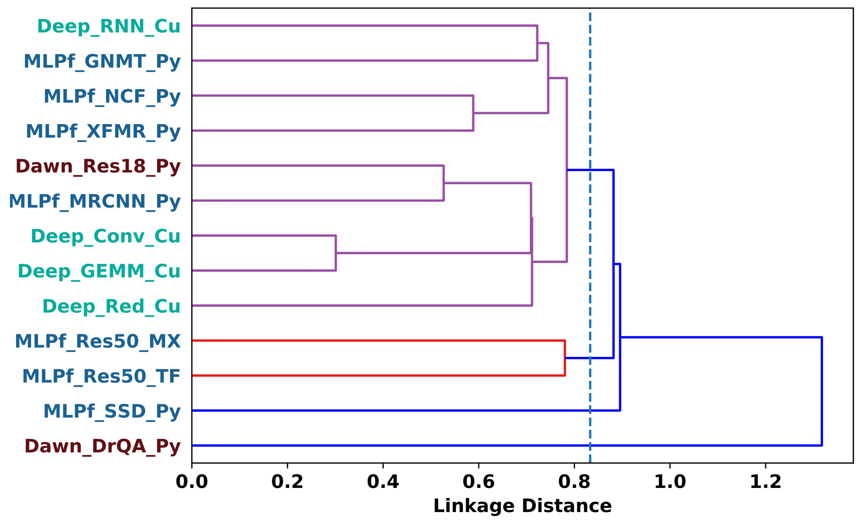


Figure 3.3: Dendrogram of MLPerf, DAWNBench and DeepBench benchmarks. If a subset of 4 is desired, pick one from each cluster intercepted by the vertical line at linkage distance around 0.8.

3.2.2 Roofline analysis

In a broader sense, the computation performance of a particular workload is determined by two key factors:

(i) how fast the computation is performed inside the processing core (e.g., CPU vs. GPU), and

(ii) how quickly the data that needs to be computed is fed into the processing core (e.g., memory bandwidth limitation).

A workload is said to be compute-bound when the workload can maximize the usage of available processing core capability; thus, its performance solely depends on how fast the processing core is. On the other hand, a workload is said to be memory-bound when the workload spends most of the time moving data back and forth between the processing core and the memory; thus, its performance is determined by how fast the memory can operate.

A roofline model [51] is a visual representation of the maximum attainable performance for a given workload in a given hardware by combining the processing core performance, memory bandwidth, and the data locality.

Figure 3.4 presents the roofline model for a single Tesla V100 GPU and machine learning workloads that are studied. The runs are carried out on the T640 system, invoking just one GPU. The vertical axis represents the compute capability that can be expressed, usually in a unit of Floating Point Operations per Second (FLOPs/sec). Meanwhile, the horizontal axis denotes the arithmetic intensity, which is the ratio between floating-point operations and data amount, using Floating Point Operations per Byte (FLOPs/Byte) as the unit. Memory-bound workloads have lower arithmetic intensity, hence their performance is limited by memory bandwidth (corresponding to the slope of the slash lines in Figure 3.4). Compute-bound workloads have high enough arithmetic intensities, so their performance is limited by the computational resources (the horizontal lines in Figure 3.4). "741.7 GB/s", "541.0 GB/s",

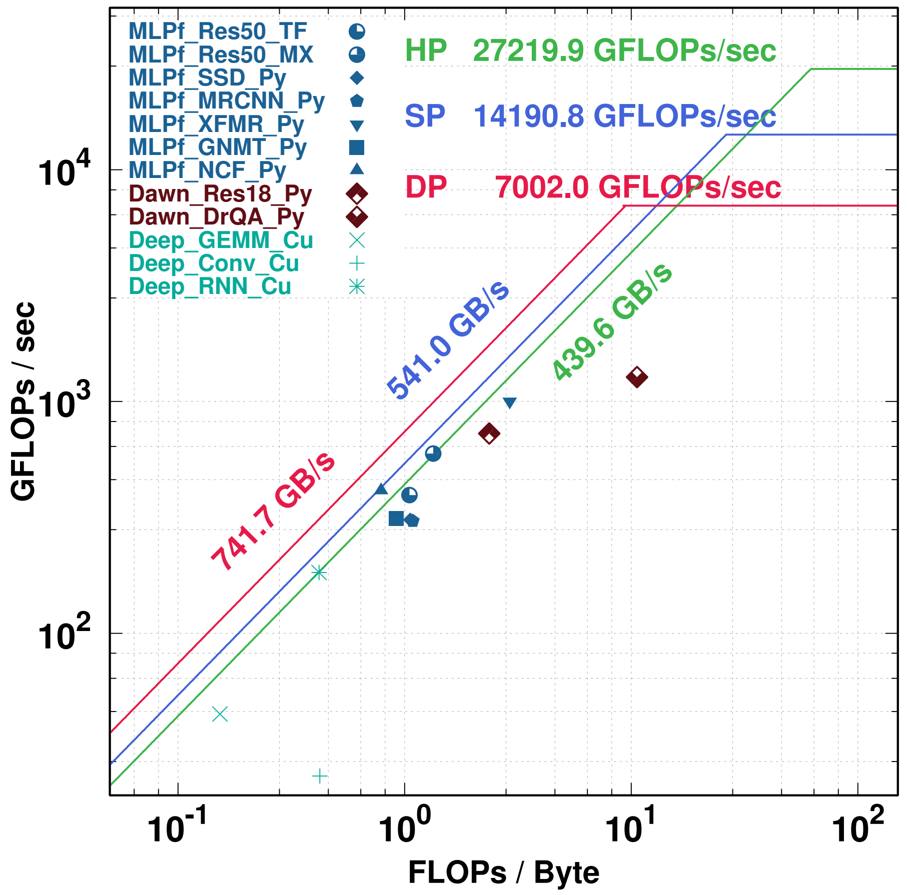


Figure 3.4: NVIDIA V100 roofline model. Red, blue, green polylines show the empirical limitations (from available memory bandwidth and computational resources) for V100 to perform double, single, and half-precision floating point operations (measured with Empirical Roofline Toolkit [54]). MLPerf benchmarks are labeled in blue shapes, DAWNBench programs labeled in red shapes, and DeepBench programs labeled in cyan shapes.

and "439.6 GB/s" as shown in Figure 3.4 are the maximum memory throughput values that can be achieved by the microbenchmark [54] with different precision settings, rather than the theoretical peak throughput. The locations of different machine learning workloads are indicated with points in different shapes. Workloads from the same benchmark suite are assigned the same color. As can be seen from the Figure 3.4, MLPerf benchmarks are more optimized than DeepBench kernels so that there is more data reuse, achieving higher arithmetic intensity, while the two DAWNBench workloads show even higher arithmetic intensities with higher throughput. Nevertheless, all the workloads are memory-bound (have not crossed the turning point, and touch the horizontal lines). This observation implies that memory is the system bottleneck for machine learning workloads, and more resources should be dedicated to memory interface for a well-balanced system.

3.3 Results

This section showcases the observations made on training hardware infrastructure with GPUs using a variety of performance metrics. It also presents results on sensitivity to interconnect topology, scheduling algorithms in multi-GPU training, compiler optimizations, and the effectiveness of mixed-precision training using Tensor Cores.

The analysis is presented on the optimized codes submitted by Google and NVIDIA to MLPerf unless specified otherwise. It may be noted from the MLPerf website that only three vendors (Google, NVIDIA, and Intel) have submitted results to MLPerf `v0.5`, and no vendor has submitted results for all benchmarks. The effort to run MLPerf codes on the systems, as shown in Figure 2.1, is non-trivial, and some of the benchmarks are omitted from some studies due to difficulties with runs. A statistic of kernels is available on-line [47].

The system-level utilization studies[3] are performed with `dstat` and `dmon` in order to better understand the impact of running MLPerf workloads and system requirements for the different models. This experimentation is performed on C4140 (K) system by appropriately regulating the number of GPUs.

3.3.1 Sensitivity of MLPerf models to Mixed Precision Training

Prior work [17,23,30] suggests that deep learning benefits from reduced precision in the following ways:

- Lowering on-chip memory requirement for the neural network models.
- Reducing the memory bandwidth requirement by accessing less or equal bytes compared to single precision.
- Accelerating the math-intensive operations, especially on GPUs with Tensor Cores.

[3]Co-investigated with co-authors of Verma et al. (2020) [48]. Included for completeness with approval from the co-authors.

Typically, only some pieces of data employ reduced precision, leading to mixed precision implementations. Moreover, employing mixed precision for training is getting easier for programmers with the release of NVIDIA's Automatic Mixed Precision (AMP) [34] feature on different frameworks like TensorFlow [4], PyTorch [36], and MXNet [9]. Figure 3.5 shows the speedup observed in different MLPerf training benchmarks by employing half-precision along with single-precision when tested on DSS 8440 using 8 GPUs. The speedup observed is in the range of 1.5× in MRCNN_Py to 3.3× in Res50_TF. Thus, it can be inferred that MLPerf, an end-to-end benchmark suite, is capable of testing the reduced precision support of processors. For example, Tensor Cores are tested here.

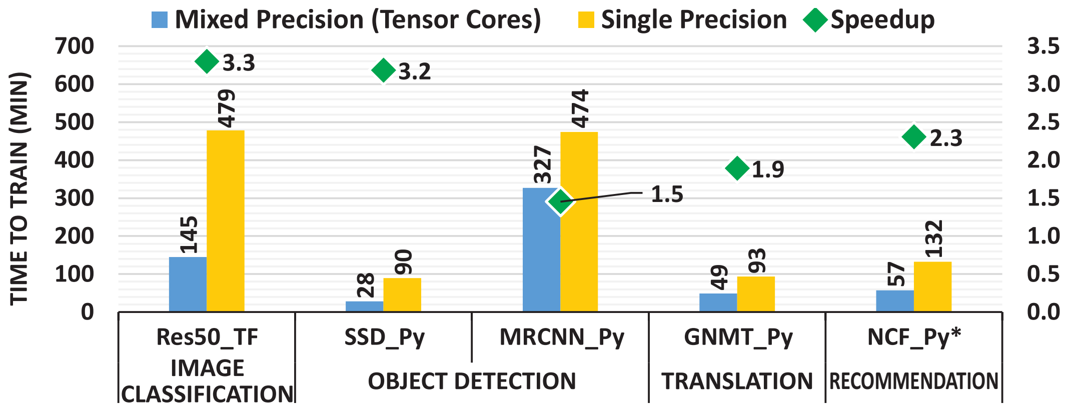


Figure 3.5: Mixed Precision training (supported by Tensor Cores) results in 1.5× to 3.3× speedups over single precision. (Note, the time of NCF_Py is in seconds)

3.3.2 Compiler optimization impact on Benchmark performance

Deep learning frameworks offer building blocks for designing, training, and validating deep neural networks through a high-level programming interface. They rely on GPU-accelerated libraries such as cuDNN [10] and NCCL [32] to deliver high-performance for single as well as multi-GPU accelerated training. From Figure 3.6, it can be seen that MLPf_Res50_TF takes around 270 minutes. These experiments are performed on C4140 (K) using all 4 GPUs. It is interesting to note that the TensorFlow/XLA JIT (just-in-time) compiler [2] optimizes TensorFlow computations and reduces the execution time by about 40% for this use case. XLA uses JIT compilation techniques to analyze and optimize the TensorFlow subgraphs created by the user at runtime. Some optimizations are specialized for the target device. The compiler then fuses multiple operators (kernel fusion) together and generates efficient native machine code for the device. This results in the reduction of execution time and the required memory bandwidth for the application.

3.3.3 Scalability of the benchmarks

The scalability study is performed on a system with 8 GPUs, the DSS 8440, where the number of GPUs employed to train the model is controlled. Ideally, performance speedup of using 2 GPUs, 4 GPUs, and 8 GPUs over 1 GPU should be 2×, 4×, and 8×, respectively. Table 3.2 shows the scalability trends for every MLPerf benchmark except for GNMT_Py. The training time using a single GPU is also added to provide a better understanding. Some of

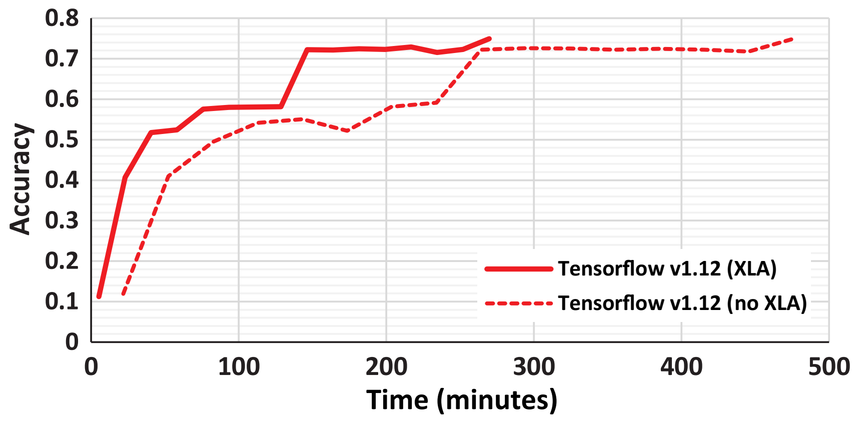


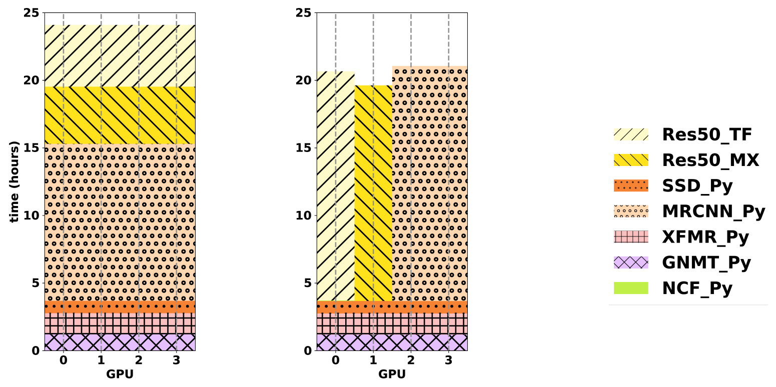
Figure 3.6: Image Classification (Res50_TF) reaches desired accuracy in 60% time if compiler uses XLA optimization.

Table 3.2: Scaling efficiency on multi-GPU systems.

Benchmark	**Training Time (min)**		**Scalability (speedup)**			
	1×P100	**1×V100**	**P-to-V**	**1-to-2**	**1-to-4**	**1-to-8**
Res50_TF	8831.3	1016.9	8.68×	1.92×	3.84×	7.04×
Res50_MX	8831.1	957.0	9.23×	1.92×	3.76×	5.92×
SSD_Py	827.7	206.1	4.02×	1.94×	3.72×	7.28×
MRCNN_Py	4999.5	1840.4	2.72×	1.76×	2.64×	5.60×
XFMR_Py	1869.8	636.0	2.94×	1.42×	2.92×	5.60×
NCF_Py	46.7	2.2	21.23×	1.88×	2.16×	2.32×

the benchmarks like Res50_TF, Res50_MX, and SSD_Py scale well with the number of GPUs, while for others increasing the number of GPUs beyond a certain point is not rewarding enough. For instance, in the case of Res50_TF, when the number of GPUs is increased from 1-to-2, from 1-to-4, and from 1-to-8, training time improves by approximately 1.9×, 3.8×, and 7×, respectively. On the contrary, for NCF_Py the speedup achieved over a single GPU is 1.9×, 2.2×, and 2.3× when the number of GPUs is increased to 2, 4, and 8, respectively. This data does not justify increasing the number of GPUs beyond 2 for training *Recommendation* benchmark. I believe the small dataset (MovieLens 20-million) causes this behavior for the benchmark. A small dataset limits the maximum batch size, which, as a result, restricts the scalability of the benchmark. Few other benchmarks, such as XFMR_Py and MRCNN_Py fall between the most and the least scalable ones, providing a scale-up by a factor of roughly 1.6×, 2.8×, and 5.6× for 2, 4, and 8 GPUs, respectively.

Such differences in scalability between different workloads give users hints to schedule a mix of machine learning training tasks. The naive scheduling scheme, that sequentially distribute every workload to all resources at once, avoids fragmentation and keeps the resources busy all the time. However, it may not be the most efficient way in terms of total training time, because users having multiple GPUs can choose to distribute some scalable workloads, while they decide to run workloads with poor scalability in sets simultaneously on fewer GPUs. Thus, the system administrators associated with super computing clusters might be interested in finding an effective algorithm to schedule various kinds of machine learning training jobs submitted from researchers, developers, and all other kinds of machine learning users. In order to show the potential benefit, a search through all permutations of scheduling 7 MLPerf benchmarks on multiple GPUs is performed, and Figure 3.7 presents 4-GPU scheduling for illustration. In each subfigure, available GPUs are listed along the x-axis, with vertical dashed lines as their timelines. Different color shades under the timeline correspond to the executions of the 7 different MLPerf workloads. Figure 3.7b shows the shortest scheduling of the 7 MLPerf benchmarks on 4 GPUs. Compared with the naive scheduling in Figure 3.7a, it saves about 3 hours to finish all the training tasks. In the optimal scheduling, the workloads chosen to be distributed on 4 GPUs, namely XFMR_Py and SSD_Py, are the scalable benchmarks, as observed above. MRCNN_Py gets two GPUs to execute due to its medium scalability. Two *Image Classification* workloads, Res50_MX and Res50_TF, are assigned to single GPUs separately to achieve faster training time. Note that two similar workloads running in parallel provides lower training time than running them in a distributed fashion even if they are highly scalable. Similarly, optimal scheduling could save around 4.1 hours and 0.4 hours for 2-GPU and 8-GPU settings, respectively. It is worth mentioning that this performance gain is without any effort in optimizing the software or adding costly hardware.



(a) (b)

Figure 3.7: Scheduling a mix of MLPerf workloads on 4 GPUs: (a) naive scheduling, which distributes one benchmark on all the GPUs one by one; (b) optimal scheduling, found by searching through the possible space, saves 3.0 hours.

3.3.4 CPU utilization across different workloads

The previous section presents the scalability of each benchmark for 1, 2, 4, and 8 GPUs runs. Although most of the computation is offloaded to the GPUs, it is worthwhile to know how the CPU is utilized during the execution of the benchmarks. Each workload is run on C4140 (K) platform and configure accordingly to use 1, 2, or all 4 GPUs available on that platform and sample the CPU usage with `dstat`.

The average CPU usage while running 1, 2, and 4 GPUs is summarized in Table 3.3. Note that, the average CPU usage includes the operating system (e.g., kernel, low-level driver) usage as well as that used by the user programs. In general, as the number of GPUs used to run the workloads doubles, the CPU utilization roughly doubles. This trend is observable for all submissions to MLPerf, which indicates that the CPU must have adequate performance to keep all GPUs busy; otherwise, it can become a bottleneck during the run.

Among the MLPerf submissions, MLPf_Res50_TF has the highest CPU utilization, followed by MLPf_Res50_MX. This is because, compared to other workloads, both *Image Classification* benchmarks require CPU to perform more packaging of the data before dispatching them to the GPUs and post-processing the data after the GPUs finish the requested tasks. Moreover, the dataset used for *Image Classification* benchmark is significantly bigger (around 300GB) compared to datasets for other benchmarks. Since it is not feasible to store such a big chunk of data on GPU memory, the CPU has to coordinate small parts of the dataset that can be stored in GPU memory at one time. The GPU can then perform a partial computation. This copying back and forth between CPU memory and GPU memory also increases the utilization of CPU. MLPf_NCF_Py shows the lowest CPU utilization followed by MLPf_GNMT_Py and MLPf_XFMR_Py. The Object Detection workloads are in the middle in terms of CPU utilization.

Another observation stems from Dawn_DrQA_Py. Although this benchmark runs on a single GPU, it has the highest CPU usage of all the workloads included in the Table 3.3. However, this benchmark also shows least GPU utilization among all the workloads, around 20%, which indicates that a major part of the computation is performed on the CPU with only a few tasks offloaded to the GPU.

Table 3.3: System resource usage statistics on C4140 (K). Utilization and footprint increase with use of more GPUs.

# GPUs	Utilization		Memory Footprint		Bus Utilization	
	CPU (%)	**GPU** (%)	**System** (MB)	**GPU** (MB)	**PCIe** (MBps)	**NVLink** (MBps)
MLPf_Res50_TF						
1xV100	10.76	85.84	17,922	15,927	1,251	0
2xV100	16.25	188.08	18,521	31,896	2,609	967
4xV100	29.06	372.43	19,970	62,214	4,269	2,867
MLPf_Res50_MX						
1xV100	4.56	85.84	7,091	10,343	1,251	0
2xV100	9.16	190.90	14,924	20,605	6,913	1,871
4xV100	18.12	378.94	28,781	40,959	11,480	21,755
MLPf_SSD_Py						
1xV100	3.89	96.13	4,100	15,406	4,720	0
2xV100	7.21	180.58	10,305	30,772	6,998	509
4xV100	13.69	334.84	20,273	60,539	9,791	1,500
MLPf_MRCNN_Py						
1xV100	2.45	62.46	7,208	4,762	258	0
2xV100	4.83	144.40	13,561	15,933	2,219	2,472
4xV100	10.39	283.88	24,923	33,935	3,444	6,547
MLPf_XFMR_Py						
1xV100	1.80	91.14	3,992	14,926	47	0
2xV100	3.35	189.30	7,167	29,493	123	11,247
4xV100	6.39	376.91	14,244	58,229	249	35,862
MLPf_GNMT_Py						
1xV100	1.91	89.94	7,210	12,098	2,743	0
2xV100	3.32	185.71	13,561	24,479	4,609	1508
4xV100	6.41	360.89	24,923	46,016	7,692	33,262
MLPf_NCF_Py						
1xV100	0.76	96.39	1,550	13,870	42	0
2xV100	2.41	194.44	3,077	24,847	110	17,887
4xV100	5.69	333.11	5,978	39,634	200	75,051
Dawn_Res18_Py						
1xV100	4.67	76.90	2,670	2,056	176	0
Dawn_DrQA_Py						
1xV100	48.84	20.30	6,721	2,657	52	0
Deep_GEMM_Cu						
1xV100	1.80	99.60	333	1,067	13	0
Deep_Conv_Cu						
1xV100	1.73	99.10	948	783	13	0
Deep_RNN_Cu						
1xV100	1.80	94.80	994	2,536	3,747	0
Deep_Red_Cu						
1xV100	0.75	91.30	313	631	27	0
2xV100	0.96	193.20	430	994	86	77,992
4xV100	1.68	366.24	1123	2320	134	404,376

3.3.5 GPU utilization for different workloads

This section looks at how each benchmark makes use of the streaming multiprocessors (SMs) available on the GPUs. Streaming multiprocessor (SM) is a part of NVIDIA GPU where the computations are performed, and each NVIDIA GPU has multiple SMs, depending on the GPU model. Each SM contains a large number of registers, SRAM arrays, scheduler(s), and, of course, the execution units. A good GPU program would offload the tasks to the GPU and distribute them across multiple SMs, thus giving high GPU utilization. Moreover, for a multi-GPU run, the program should also distribute the work across SMs on multiple GPUs efficiently by maximizing computation time in each GPU and minimizing communication time between GPUs.

The GPU utilization, as given in Table 3.3, is the sum of the utilization of every GPU that is used during the runtime. Therefore, single-, dual-, and quad-GPU run have maximum utilization of 100%, 200%, and 400%, respectively. For *Image Classification* workloads, both MLPf_Res50_TF and MLPf_Res50_MX, show near-identical GPU utilization with around 85% GPU usage for single-GPU run, around 190% GPU usage (i.e., around 95% utilization per GPU) for dual-GPU run, and around 375% GPU usage (i.e., around 93.5% utilization per GPU) for quad-GPU run.

Most of the submissions to MLPerf show a similar trend for single-GPU and dual-GPU runs. Moreover, MLPf_NCF_Py shows decreasing individual GPU usage for quad-GPU run compared to dual-GPU run. This observation agrees with the one mentioned in Section 3.3.3 that due to the limited scope of increase in the batch size for the workload, it is unable to utilize the GPUs efficiently. Increasing communication cost for multi-GPU run can impact individual GPU utilization as confirmed by Deep_Red_Cu benchmark from DeepBench. Another indicator to see the communication cost between GPUs is to look at the NVLink bus utilization, which is shown in Section 3.3.7.

3.3.6 CPU and GPU memory footprint

The system memory is mostly used to store the dataset that is used for the training as well as the intermediate data required between computations. In the case when the dataset is too large to fit in the GPU memory, the system memory acts as a buffer to store the dataset. The user program moves the data back and forth between the system and GPU memory to perform partial calculations. Moreover, in an extreme case, the dataset can be too large to be stored inside the system memory. Thus the disk storage (e.g., hard disk drive, solid-state drive) is used to store them, and the CPU is responsible for coordinating the switching between each part of the dataset.

From Table 3.3, it can be observed that the system memory footprint roughly doubles every time the number of GPUs is doubled. The GPU memory footprint is the total memory footprint for every GPU used during the run. Note that the footprint of GPU memory depends on the batch size, and the batch sizes for the experiments are scaled accordingly from the original submissions, as mentioned in Section 3.1.2.

Although the table only shows the memory footprint of each benchmark, I would like to emphasize that the heterogeneity of the medium where the dataset is stored may become a bottleneck, especially for memory-bounded applications that perform data exchange frequently. In this case, the interconnect bandwidth between each storage medium and the intelligence of the program to overlap the data transfer just before the next computation and to manage the locality of the data can play a crucial factor.

In C4140 (K) platform, for example, each CPU has 96GB of memory consisting of six 16GB DDR4-2666 DIMMs in hexa-channel memory configuration. The theoretical unidirectional memory bandwidth available to each CPU is around 128 GBps [14]. In comparison, Intel's proprietary Ultra Path Interconnect (UPI) that links two CPUs has only unidirectional theoretical bandwidth of 20.8 GBps [31]. In a case when a CPU needs a part of the dataset stored in other CPU's memory, the performance of data transfer will be significantly reduced (i.e., 128 GBps direct access for local DRAM v.s. 20.8 GBps neighbor DRAM access via UPI).

The same thing happens with a GPU that has a more limited dedicated memory. In C4140 (K) platform, each NVIDIA Tesla V100 is equipped with 16GB HBM2 stacked memory, which is capable of 450 GBps unidirectional bandwidth. In the case that the dataset cannot be fully stored inside the GPU memory, the CPU should bring a part of the dataset from the system memory into the GPU memory. This data exchange uses PCIe 3.0 bus that connects the CPU and GPU and is able to provide theoretical unidirectional bandwidth of 15.8 GBps for x16 lanes, which limits the performance of data transfer.

3.3.7 System and GPU bus utilization

Earlier sections highlight that interconnection bus between CPU-GPU and between GPU-GPU may play an important role in determining the overall system performance. Moreover, they mention that the choice interconnection topology between CPU and GPU should be considered carefully. This section explains more details about how the performance is impacted by the interconnection bus based on the data on Table 3.3.

Modern microprocessor systems use PCI Express (PCIe) bus as the interconnection standard between CPU and external peripheral that requires high-speed data communication. PCIe 3.0 standard, introduced in 2010, has been widely adopted by most computer system products available in today's market. PCIe 3.0 provides theoretical unidirectional bandwidth up-to 984.6 MBps per lane and up-to 15.8 GBps per PCIe 3.0 compatible device connected using 16 PCIe 3.0 lanes (PCIe 3.0 x16). This massive bandwidth, in theory, should be sufficient for most of the peripheral devices, including GPU, network interface card, and non-volatile memory storage.

Usually, a GPU is connected to the CPU using PCIe 3.0 x16 to assure that there is plenty of bandwidth between them. High bandwidth is easy to achieve for a single-GPU system, but more complicated for a multi-GPU system since the number of PCIe 3.0 lanes that the CPU has are limited. High-end Intel Xeon may have up to 48 lanes of PCIe 3.0, which are then allocated to various devices. With this constraint, each GPU on a four GPU system, for example, may only be assigned eight PCIe 3.0 lanes. While it depends on how the GPU is used and how intense the data exchange happens between the CPU and GPU, some applications like gaming may find PCIe 3.0 x8 already provides plenty of bandwidth. On the other hand, this much bandwidth may not be optimal for deep learning training.

Alternatively, a PCIe switch, such as those manufactured by PLX Technology, can be used to provide additional PCIe lanes; thus, each GPU can have PCIe 3.0 x16 lanes. This switch will be useful for GPU-to-GPU communication since the data exchange will only take place on the switch without going over to the CPU. However, the switch will not be beneficial to improve the bandwidth between CPU and all GPUs on the system as the effective CPU-to-GPU bandwidth is still limited by what the CPU has. The interconnection topology and how it affects the performance is discussed in Section 3.3.8.

Furthermore, apart from CPU-to-GPU communication, PCIe bus can be used for GPU-to-GPU communication for a multi-GPU system. Although each GPU can be allocated with PCIe 3.0 x16 lanes, the available bandwidth may not be sufficient for some workloads that require intensive data exchange between the GPUs. Therefore, an additional bus specifically for GPU-to-GPU communication has been developed, such as NVLink, which is high-speed proprietary interconnect system in NVIDIA GPUs. Each NVLink lane provides 25 GBps theoretical unidirectional bandwidth. The NVIDIA Tesla V100 GPU in SXM2 form factor has six NVLink lanes that are capable of transferring data with theoretical unidirectional bandwidth of 150 GBps. This is significantly faster than what PCIe 3.0 x16 can offer.

Besides, NVLink can also be used for CPU-to-GPU interconnect, replacing the PCIe 3.0 bus. This feature is available on Power8 and Power9 CPU from IBM. However, there is no x86 CPU that features NVLink interface; thus, the CPU-to-GPU connection still uses the PCIe 3.0 bus.

Table 3.3 shows the PCIe 3.0 bus utilization between CPU and GPU available on the system as well as NVLink utilization between GPU and GPU. The value presented in the table is the sum of PCIe 3.0 bidirectional PCIe bus utilization for each GPU that is used during the run, and the sum of NVLink lane utilization from each GPU used during the run. As can be seen from the table, the data transfer rate over NVLink bus increases as more GPUs are added for the run. Deep_Red_Cu and MLPf_NCF_Py use the highest bandwidth of NVLink, which means that the data exchanges between GPU for these benchmarks are intensive. On the other hand, the utilization of PCIe 3.0 bus increases as more GPUs are added, which is expected. In a multi-GPU system equipped with NVLink, the PCIe 3.0 bus is used only for communication between CPU and each GPU because the GPU to GPU communication has been offloaded into the higher speed NVLink.

3.3.8 Impact of GPU-Interconnect Topology

In order to reduce the training time, it is becoming increasingly common to scale deep learning (DL) training across multiple GPUs within a system. There are multiple ways in which the GPUs can be connected within the system. Primarily there are two options available - using a PCIe based interconnect (which may include PCIe switches if the number of lanes from the CPU is not sufficient) and using NVIDIA's proprietary interconnect like NVLink. The theoretical bandwidth of an NVLink interconnect is 10× higher than PCIe (300 GBps vs. 32 GBps) [33]. Additionally, communication libraries like NCCL from NVIDIA are optimized to perform GPUDirect peer-to-peer (P2P) direct access when NVLink is available between GPUs, which can lower training times if there is significant peer-to-peer communication during model training. GPUDirect P2P is also feasible in certain PCIe topology designs where GPUs are the same PCIe domain (single root complex). Using MLPerf, a performance evaluation study of five different 4-GPU platforms is conducted, each of them with a unique GPU interconnect topology. Figure 3.1 shows how the GPUs are interconnected for the servers used in this study.

Two of the five servers, C4140 (M) and C4104 (K), include the high-speed proprietary NVLink interconnect to provide 100 GBps bandwidth between any two GPUs. The difference between the two NVLink based designs is the use of a PCIe switch in the C4140 (K) to aggregate the PCIe connections to the GPUs. The remaining three systems use PCIe based interconnects. They use very different approaches in how the GPUs are connected to the CPUs and in how they communicate with other GPUs. One system C4140 (B), uses a 96-lane PCIe switch that allows for 4 GPUs to be hosted in a single PCIe domain where it can perform GPUDirect peer-to-peer (P2P) between the GPUs using the PCIe switch. This is not feasible in the other two PCIe based interconnect platforms - T640, where two GPUs are hosted per CPU, and R940 XA, which is a 4 CPU platform with each GPU connected directly using the PCIe lanes of the CPU.

The training times for the different servers are plotted in Figure 3.8, which illustrates the impact of GPU interconnect topology on DL training times. As expected, due to lack of GPUDirect P2P capability between any of the GPUs, the T640 and R940 XA take the longest time to train all the MLPerf models. Conversely, the two servers that use NVLink interconnect (the C4140 (M) and (K) systems) show the best training times across all the MLPerf models. However, the performance improvements differ depending on the model that is being trained and ranges from 42% and 17% for the *Translation* benchmarks, 30% for MLPf_MRCNN_Py to 11% for the *Image Classification* benchmarks. The C4140 (B), which uses a PCIe topology but can perform GPUDirect P2P between GPUs due to all GPUs connected to a PCIe switch, shows performance parity to the NVLink platform for the *Image Classification* benchmarks and better performance than the R940 XA and T640 servers for remaining benchmarks. This platform provides a mix of flexibility that is available when using PCIe based GPU cards in addition to higher performance over PCIe based designs that do not support GPUDirect P2P transactions between GPUs.

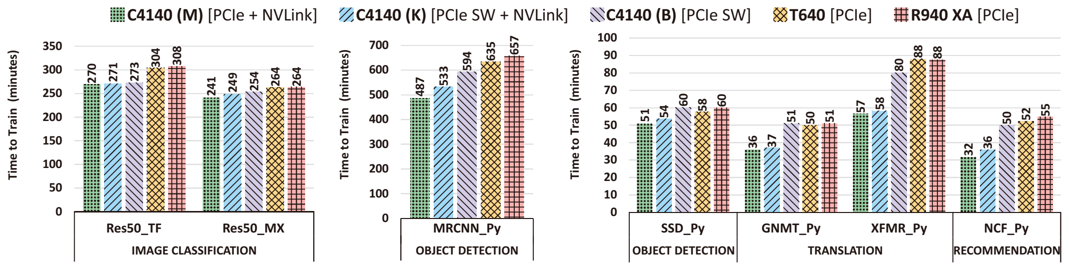


Figure 3.8: Training time on 4-GPU systems (topologies shown in Figure 3.1). Time on systems with NVLink interconnect (the first 2 bars) is less than training time on the remaining systems. (Note that the time of NCF_Py is in seconds)

3.3.9 Impact of job types on system utilization

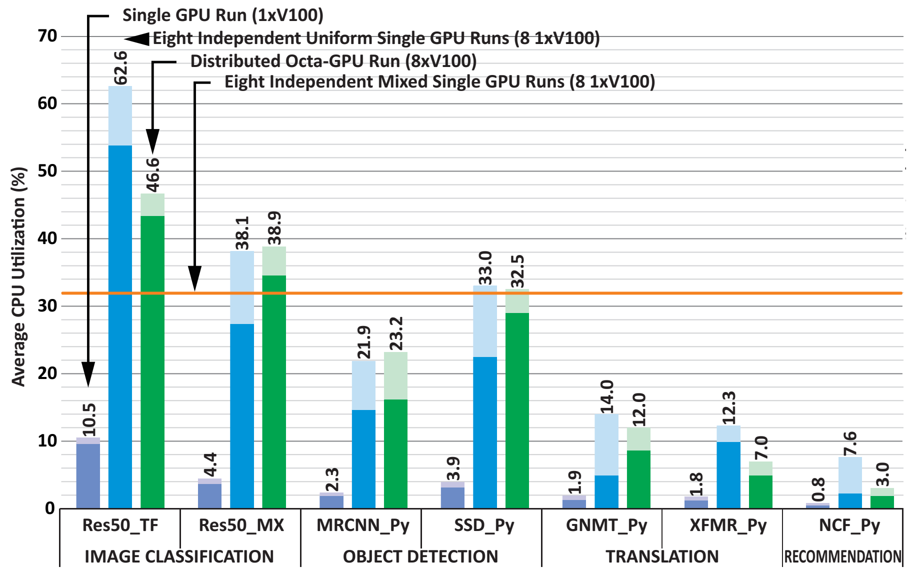
GPU infrastructure in cloud or on-premise data centers typically hosts different classes of training jobs with different purposes:

- **Distributed-run**: to train a large complex model over a large training dataset across multiple GPUs for fast time-to-solution.
- **Multiple-run**: to sweep hyper-parameter space of the same model, typically having one training run with different settings of hyper-parameters on each GPU on the same test dataset.
- **Mixed-run**: different users submit different jobs that are training smaller models using single GPU each on a cluster

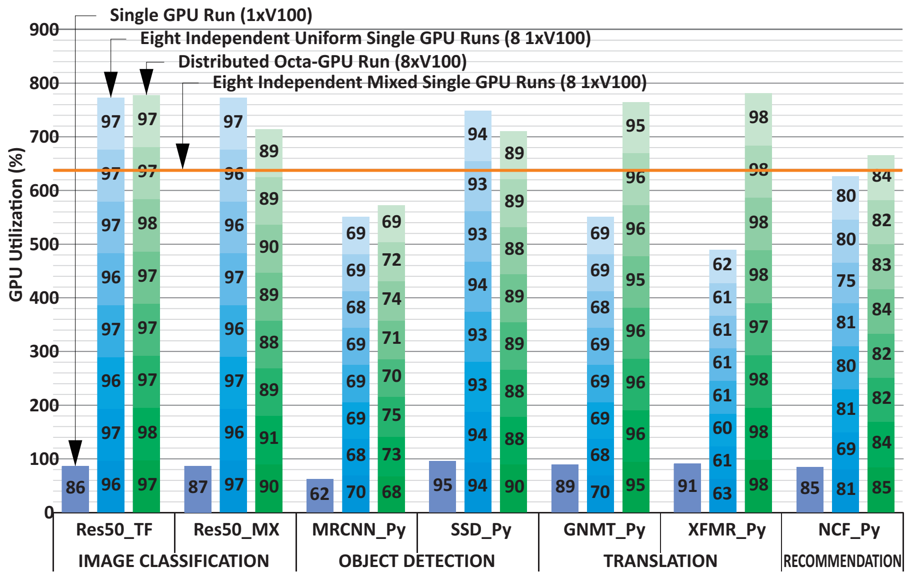
This section compares the system resource utilization of these three methods of running machine learning workloads on a multi-GPU system (the 8-GPU DSS 8440).

Figure 3.9 shows the CPU and GPU utilization for each method. In general, running multiple instances of the same benchmark requires higher CPU utilization compared to a single instance on multiple GPUs (distributed-run). It is because for multiple-run, each instance has its own host (CPU) program that performs pre-processing, controls the GPU computation, and collects the computation from the GPU. Thus, CPU is required to handle each host program, hence, leading to higher CPU utilization. On the other hand, the mixed-run CPU utilization is roughly the same as the sum of CPU utilizations of a single GPU run for each workload.

In GPU utilization, the topology of how the GPUs are connected to the CPU plays an important role. MLPf_NCF_Py, MLPf_XFMR_Py, GNMT_Py, MLPf_MRCNN_Py, and the MLPf_Res50_TF have higher GPU utilization for the distributed eight-GPU run compared to eight independent uniform single GPU runs. It turns out that their usage of PCIe bus for the distributed run is significantly higher compared to the independent run. During the distributed-run, total data transfer rate for MLPf_NCF_Py, MLPf_XFMR_Py, MLPf_GNMT_Py, MLPf_MRCNN_Py, and MLPf_Res50_TF over PCIe bus reaches 58.95 GBps, 51.90 GBps, 39.1 GBps, 19.51 GBps, and 16.97 GBps, respectively. Meanwhile, for the multiple-run, they only use 276 MBps, 606 MBps, 2.07 GBps, 2.07 GBps, and 12.39 GBps, respectively. I suspect that most of GPU utilization is coming from communicating between GPUs and as there is no NVLink for GPU-GPU communication on this system. Each GPU competes for the PCIe bus as well as for the UPI link. On the other hand,



(a) CPU utilization



(b) GPU utilization

Figure 3.9: Utilization of CPU (a) and GPU (b) for single-GPU run, 8 independent uniform single-GPU runs, distributed 8-GPU run, and 8 independent mixed single-GPU runs. Higher CPU utilization in multiple-run vs distributed in 5 of 7 cases.

MLPf_Res50_MX and MLPf_SSD_Py have the opposite behavior. Here the total data transfer rate for the distributed-run is smaller than the multiple-run.

3.3.10 Impact of GPU clock frequency

Most GPUs are packaged in the form of a PCIe card defined on the standard established by PCI-SIG. This form factor allows the card to be inserted on the PCIe slots available on the system. On the other hand, as an addition to the PCIe card form factor, NVIDIA developed their proprietary form factor called SXM2. The NVIDIA Tesla V100 is one of the GPU from NVIDIA that is available in the form of PCIe card and SXM2 form factor.

The PCIe card version of NVIDIA Tesla V100 has full-length, double-height size with a passive cooling system that offers more versatility because of its compatibility with all of the chassis that supports PCIe card. This version only features PCIe 3.0 x16 bus for CPU-to-GPU interconnect as well as GPU-to-GPU interconnect. With a TDP of 250W, the GPU is clocked at up to 1380 MHz resulting up to 7 TFLOPs/sec of double precision, 14 TFLOPs/sec of single precision, and 112 TFLOPs/sec of mixed precision (utilizing Tensor Cores).

Meanwhile, the SXM2 version of NVIDIA Tesla V100 has a mezzanine-like form factor that requires support from the chassis to accept their connector. Because of the custom form-factor, NVIDIA is able to run the GPU at slightly higher clock speed by increasing the thermal envelope to 300W. Clocked at up-to 1530MHz, depending on how good the cooling solution is, it can perform up to 7.8 TFLOPs/sec of double precision, 15.7 TFLOPs/sec of single precision, and 125 TFLOPs/sec of mixed precision (utilizing Tensor Cores). Moreover, with the SXM2 form factor, NVIDIA is able to integrate six lanes of high-speed NVLink as an addition to PCIe 3.0 bus with a total of 300 GBps bidirectional bandwidth via NVLink and 31.6 GBps bidirectional bandwidth via PCIe 3.0. The only drawback of this form-factor is that it is less flexible in changing this GPU to other GPU or accelerator because most of them are packaged in the PCIe card form factor while the chassis can only accept SXM2 form factor.

NVIDIA Tesla V100 in PCIe card form factor and NVIDIA Tesla V100 in SXM2 form factor are compared in terms of MLPerf benchmark runtime, as shown in the Figure 3.10. While the differences in runtime are not that significant, the SXM2 version of NVIDIA Tesla V100 is faster in all MLPerf benchmark submissions. The noticeable differences are in NCF_Py benchmark that has high utilization of NVLink for GPU-to-GPU communication, as suggested by Table 3.3, and thus SXM2 form factor has a significant benefit on it.

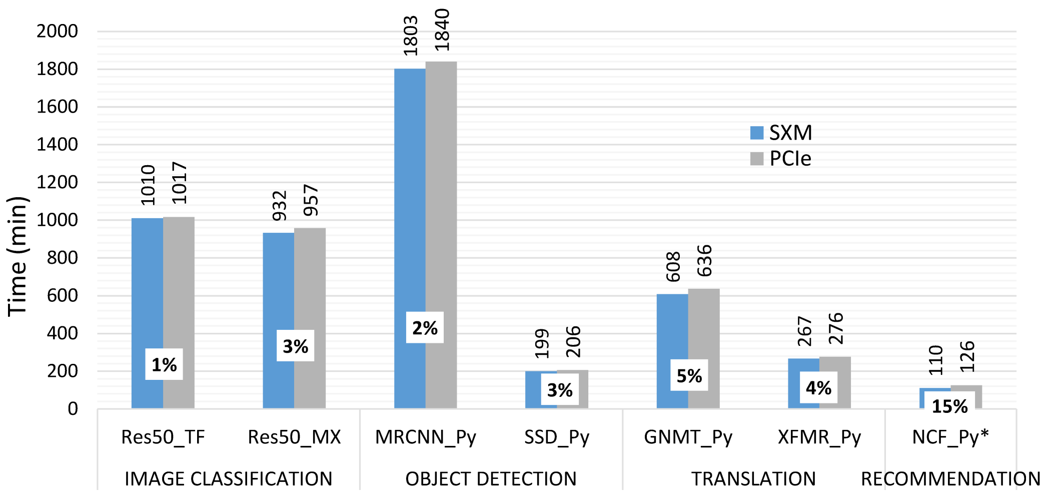


Figure 3.10: Comparison of PCIe and SXM2 form factor on one NVIDIA Tesla V100 (16 GB) for the MLPerf training benchmark submissions. (Note that the time of NCF_Py is in seconds)

Chapter 4

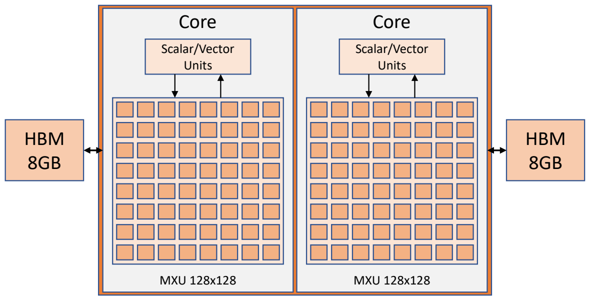
Experiments on TPUs

This chapter talks about various studies performed on TPU enabled systems. First, it presents the system configurations, benchmarks, and tools that were used in Section 4.1; followed by some insights on the scaling efficiency of the benchmarks, performance comparison of TPU v3 and TPU v2, benefits of using bfloat16 data type in training, and the variation in host-side activity in Section 4.2.

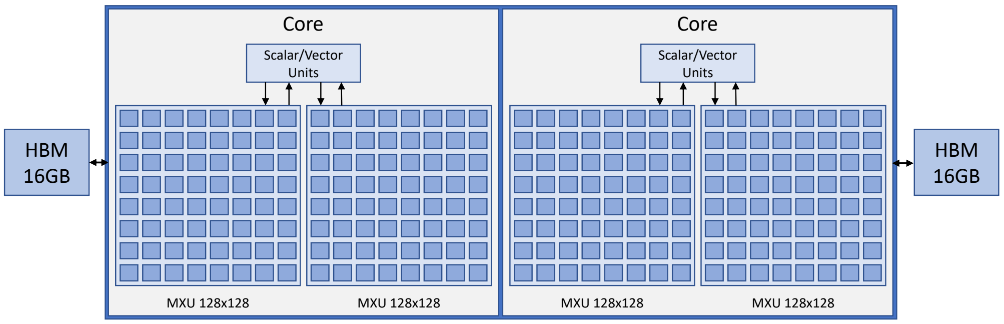
4.1 Methodology

4.1.1 System configurations

Various Google cloud N1 (standard) systems attached to multiple TPUs are used for experimentation in this chapter. Hardware specifications for the same are highlighted in Table 4.1 and Figure 4.1. It is evident from the table that the systems have access to 8 TPU cores; however, TensorFlow TPU's experimental API, device placement, is utilized in order to control the number of TPU cores used in the run. For brevity, the following nomenclature is used: `v{x}-{y}` refers to a system using `x` version of TPU (can be 2 or 3) using `y` number of cores (can be 1, 2, 4 or 8). Note that the host memory capacity



(a) TPU v2



(b) TPU v3

Figure 4.1: Illustration of different TPU versions.

Table 4.1: Hardware specifications of systems for experimentation. (MUX - TPU Matrix Units)

Systems	**v2-8**	**v3-8**
CPUs (**Intel Xeon**)		
Model #	63	85
Base freq.	2.30GHz	2.00GHz
Memory		
# DIMM	2	2
Size	30GB	30GB
TPUs (**Google**)		
# Cores	8	8
# MXUs/core	1	2
Memory/core	8GB HBM	16GB HBM

is dependent on the specific type of machine instance; specifically, the table shows the available memory for `n1-standard-8` instance. All the systems operate on Debian GNU/Linux 9 (stretch).

4.1.2 Benchmarks

Google's TPU submissions of the **MLPerf** [3] `v0.6` training benchmarks are chosen for experimentation in this chapter. The submitted source codes were optimized for performance on hardware systems with at least 32 TPUs. So, the benchmark implementations for `tpu-v3-32` is picked and scaled accordingly to run on a cluster with a maximum of 8 TPUs. Note that, as there was no TPU submission for *Recommendation* and *Reinforcement Learning* benchmarks, these benchmarks are excluded from the study. Additionally, *Image Classification* and *Translation (Transformer)* benchmarks are excluded due to the inability to replicate the runs on 8 (or less) TPUs.

4.1.3 Measurement tools

Google's **Cloud TPU Profiler** is used to capture a profile for an epoch of the benchmark's run. Information collected are: utilization of the TPU matrix unit (systolic array), top-10 kernels that run on the TPUs, host-side peak memory usage, host-side input operations, and TensorFlow operations that account for the overall floating-point operations. This information is used to analyze the benchmark's performance in TensorFlow's TensorBoard console.

4.2 Results

In contrast to Section 3.3, this section highlights observations made on training hardware infrastructure attached to TPUs. In particular, it brings to light the scalability of the benchmarks, benefits of the bfloat16 data type, speedup gained on using TPU v3 over TPU v2, and some insights on the host-side activity. The analysis is presented on the optimized codes submitted by Google to MLPerf unless specified otherwise. Note that some runs are omitted due to the fact that they cannot be replicated on hardware with less number of TPUs ($<$ 32 cores) due to limited TPU memory. Additionally, the runs did not always converge to the desired quality target; however, this would not affect the results and analysis presented.

4.2.1 Scalability of the benchmarks

The scalability study is performed by varying the number of TPUs used for different runs. The average epoch time is used to investigate the same. The analysis presented will hold true for the complete runs (those who reach the desired quality target) assuming the number of epochs taken by the system remains the same, which is generally the case and is achieved by manipulating the hyperparameters. On adding more TPU cores, ideally, the performance should improve in the same proportion, i.e., using 2, 4, and 8 TPU cores should give a speedup of 2×, 4×, and 8× over 1 TPU core, respectively. In a non-ideal case, several overheads caused by using multiple TPUs can reduce this benefit. However, the speedup can be more than the ideal case as well. This can happen if the benchmark overcomes a critical bottleneck as a result of using multiple TPUs.

The time taken by various systems running different benchmarks to complete an epoch is shown in Table 4.2. The speedups for the same are presented in Table 4.3. As the efficiency is observed along the row, each row contains a value of 1× corresponding to which the rest of the elements belonging to the same row are evaluated. From Table 4.3, it is evident that the SSD_TF and GNMT_TF are highly scalable benchmarks when evaluated on TPUs. For example, training SSD_TF (performing compute in float32 data type) shows 2×, 3.99×, and 7.99× speedup on using 2, 4, and 8 v3 TPUs over 1 v3 TPU. SSD_TF training using bfloat16 data type is also observed to have super-linear scaling efficiency. One possible reason for this behavior is that the runs taking advantage of only one TPU core are found to have top (most time consuming) TensorFlow operation(s) on TPU that is very inefficient in terms of the throughput (FLOPS) they provide. This finding agrees with what observed in Section 3.3.3. On the other hand, surprisingly, MRCNN_TF, whose PyTorch implementation from `v0.5` is classified as a mediocre scalable benchmark, shows very low scalability on TPUs, i.e., less than 2× speedup even on using 8 TPUs over 1 TPU. This observation can be supported by the fact that the batch size was not constant when increasing the number of TPUs. This profoundly affects the epoch duration for MRCNN_TF (as highlighted in Section 4.2.2). The variability of the scaling efficiencies observed in this section also motivates the need smart scheduling techniques for multi-TPU systems, similar to what is proposed in Section 3.3.3.

4.2.2 Performance enhancement of v3 over v2

In the year 2017, the basic architecture of Google's custom deep learning accelerator for the datacenter, TPU, was revealed [24]. Similar to many production chips, TPUs improved over the years. Major architectural differences between the two versions of the TPUs used in this study are highlighted in Table 4.1 and Figure 4.1. The TPU paper [24] says that one TPU core has a single MXU of size 256×256 (supporting 64K MACs per cycle), while the newer designs, v2 and v3, has MXU(s) each of size 128×128 (supporting 16K MACs per cycle). According to the official documents [16], the high compute power and on-chip memory of v3 TPUs can benefit the deep learning

Table 4.2: Epoch duration on various TPU-enabled systems for different MLPerf training benchmarks.

Data Type	**Epoch Time (min)**			
bfloat16	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	13.03	5.70	3.26	1.42
MRCNN_TF	22.49	18.77	15.75	14.59
GNMT_TF			31.30	15.12
bfloat16	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	5.77	2.95	1.25	0.63
MRCNN_TF	32.13	29.88	28.74	28.20
GNMT_TF	70.77	29.91	15.96	9.00
float32	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	13.48	6.79	3.40	1.70
MRCNN_TF	21.76	15.80	12.91	11.39
GNMT_TF				18.09
float32	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	5.50	2.75	1.38	0.69
MRCNN_TF	21.96	19.12	17.74	17.10
GNMT_TF		51.09	19.17	10.72

Table 4.3: Scaling efficiency on multi-TPU systems.

Data Type	**Speedup**			
bfloat16	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	1×	2.29×	3.99×	9.14×
MRCNN_TF	1×	1.20×	1.43×	1.54×
GNMT_TF			1×	2.07×
bfloat16	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	1×	1.96×	4.63×	9.19×
MRCNN_TF	1×	1.08×	1.12×	1.14×
GNMT_TF	1×	2.37×	4.43×	7.87×
float32	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	1×	1.98×	3.96×	7.93×
MRCNN_TF	1×	1.38×	1.69×	1.91×
GNMT_TF				1×
float32	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	1×	2×	3.99×	7.99×
MRCNN_TF	1×	1.15×	1.24×	1.28×
GNMT_TF		1×	2.66×	4.77×

workloads in the following cases:

(i) when the workload is compute-bound.

(ii) when the workload is memory-bound on v2 TPUs, but is not when using v3 TPUs.

(iii) when v2 TPUs do not meet the on-chip data storage requirement, but v3 TPUs meet the same.

(iv) when using cutting edge models or using larger mini-batch sizes, which cannot be supported by v2 TPUs.

This section attempts to quantify the gains provided by the v3 design when compared to the v2 design on the MLPerf training benchmarks. Figure 4.2 illustrates the same using the numbers derived from the Table 4.2. The figure suggests that the speedups observed are highly dependent on the benchmarks itself. For example, SSD_TF achieves a geomean speedup of more than 2.25× for both the precision types (one using bfloat16 and another using float32; the difference regarding which is discussed in Section 4.2.3). On the other hand, it seems that MRCNN_TF does not enjoy any benefit from v3 TPUs. Additionally, the trend shows that on increasing the number of TPUs, the performance of v2 TPUs on this benchmark improves at a higher rate than v3 TPUs. This behavior for MRCNN_TF is attributed to the fact that maximum batch sizes supported on different versions of the TPUs differ by a factor of 4. As the on-chip memory capacity on v2 and v3 TPUs are

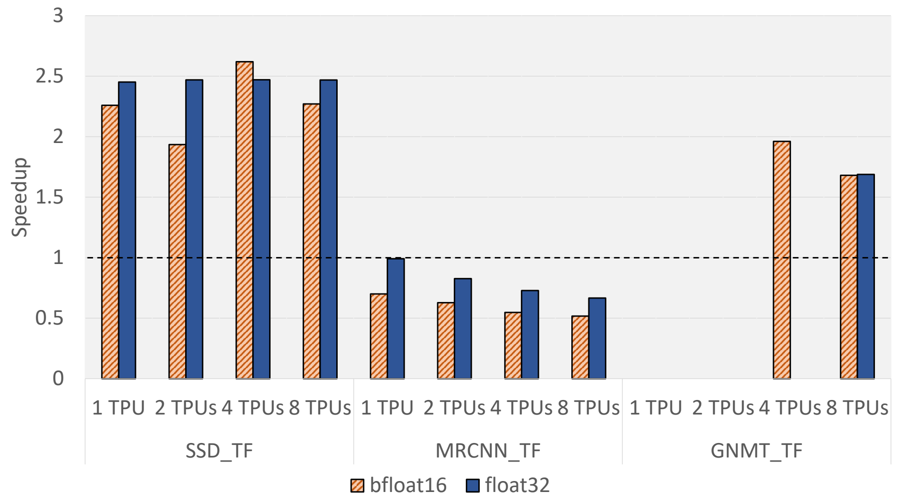


Figure 4.2: Speedup provided by v3 TPUs when compared to v2 TPUs.

different, the runs are made with different batch sizes such that the on-chip memory is utilized to the fullest. It is found that when the batch size of 64 (used for training MRCNN_TF on v3-8 with bfloat16 data type) is reduced by a factor of 2, 4, and 8, the speedup observed in terms of the epoch time is 1.88×, 2.69×, and 3.30× respectively. Therefore, if the comparison is made such that batch size is the same for training on both v2 and v3 TPUs, the speedup observed for MRCNN_TF is greater than 1 (on 8 TPUs using batch size 16 with bfloat16 data type, the speedup is 1.39×).

4.2.3 Significance of training in bfloat16

Deep learning practitioners found that there are several benefits to using reduced precision for training and inference (highlighted in Section 3.3.1). As a result, instead of using the IEEE standard half-precision format (float16), deep learning researchers came up with other compact representations explicitly designed for deep learning applications such as Brain Floating Point (bfloat16) [15], Dynamic Fixed Point [12,52], and Flexpoint [26]. bfloat16 was proposed by Google and hence is supported in their machine learning framework, TensorFlow [4], and their custom deep learning accelerator, TPU. It is a 16-bit format containing 1 sign bit, 8 exponent bits, same as in the IEEE standard single-precision format (float32), and 7 mantissa bits. Therefore, the bfloat16 format enjoys a greater dynamic range than float16. This section quantifies the performance improvements observed when moving to bfloat16 format for MLPerf training benchmarks. Note that the inputs and outputs to MXUs are always in float32 format whereas the multiplication is performed in the bfloat16 format.

Figure 4.3 shows that GNMT_TF achieves a speedup ranging from 1.19× to 1.71× using bfloat16 over float32. Mixed precision training of SSD_TF on v2 TPUs is always profitable; however, it appears sometimes to suffer slowdown when trained on v3 TPUs. The benchmark shows a performance increase of up to 1.19× on v2 TPUs. Training of MRCNN_TF using bfloat16 is severely impacted by the variation of batch sizes, as discussed earlier. Up to 39% performance degradation is observed in bfloat16 experiments; however, a smaller batch size used in the float32 experiment is the reason for the reduced runtime. Similarly, there are some caveats which concern the validity of the apparent slowdowns in SSD_TF. Note that some of the runs do not converge to the same accuracy. Additionally, the batch size used for training is not the same. The

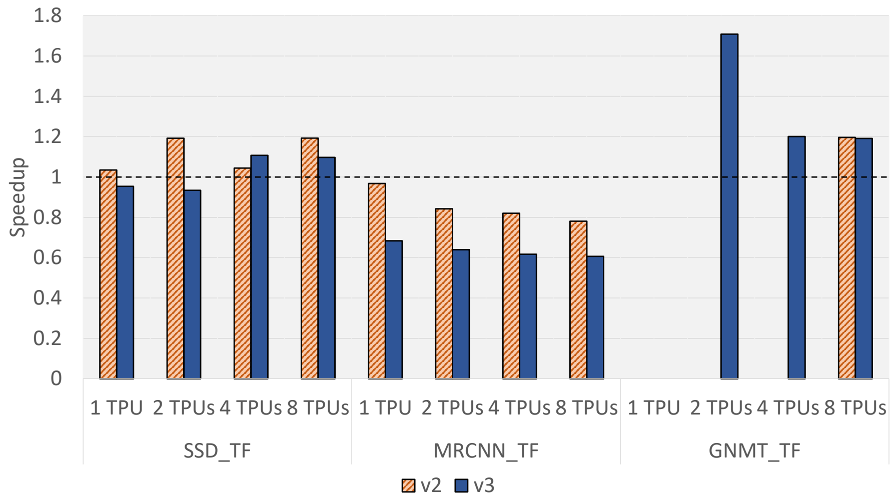


Figure 4.3: Performance gain when using bfloat16 data type over float32 on MLPerf training benchmarks.

maximum batch sizes supported using bfloat16 and float32 are seen to differ by up to a factor of 4.

4.2.4 TPU matrix unit utilization for different workloads

TPU cores consist of scalar, vector, and matrix units (MXU). MXUs are responsible for the high compute horsepower delivered by the TPUs. Each MXU is capable of performing 16k MACs per cycle [16]. Utilization of the MXUs when training the MLPerf benchmark suite on various TPU-enabled systems is exposed in this section.

Table 4.4 shows the percentage utilization of the Matrix Units while training MLPerf benchmarks. It is observed that as the number of TPU cores are added, the utilization increases almost linearly. This is because the number

Table 4.4: Utilization of Matrix Units on various TPU-enabled systems for different MLPerf training benchmarks.

Data Type	**MXU Utilization (%)**			
bfloat16	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	9	18.3	37.2	75.4
MRCNN_TF	4.1	8.6	17.8	36.5
GNMT_TF			28.7	62.1
bfloat16	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	7.5	15	31.7	62.8
MRCNN_TF	3.2	6.5	12.9	26.2
GNMT_TF	4.6	10.9	21.4	42.5
float32	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	7.8	16.7	34.3	70.7
MRCNN_TF	2.8	5.9	12.1	25.5
GNMT_TF				51.6
float32	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	6.2	13.9	28.6	58.1
MRCNN_TF	2.6	5.5	11.3	22.9
GNMT_TF		6.5	17.5	34.6

of TPU cores are enabled/disabled using the device placement, which makes the total number of MXUs constant in the systems. Furthermore, as more TPU cores are enabled, more MXUs can be utilized. Another observation is that the MXU utilization for the v3 TPUs are lower when compared to the v2 TPUs, ranging from 7% to 33% reduction. This is expected as the v3 TPU cores have twice the number of MXUs as the v2 TPU cores and thus makes the denominator for calculating the utilization twice as large. Moreover, using reduced precision training increases the MXU utilization by up to 47%.

Additionally, diversity in the utilization values can highlight the contrast among the benchmarks. Average TPU FLOPS utilization by SSD_TF is found to be 55%, by MRCNN_TF is found to be 22%, and by GNMT_TF is found to be 41% of the TPU peak FLOPS.

4.2.5 Host activity across different workloads

Section 3.3.4 showcased the importance of CPUs when the MLPerf benchmarks are trained on multi-GPU platforms. This section explores a similar idea concerning multi-TPU systems by characterizing the percentage of active host time (percentage of the time when the host is not idle). The TPU profiler characterizes work of the host into five sets:

- Reading data from files on demand.
- Reading data from files in advance (includes caching, prefetching, interleaving).

- Data preprocessing (like image decompression).
- Enqueuing data to an infeed queue to be transferred to the device
- Other data reading or processing.

Table 4.5 shows the percent time spend by the host doing some useful work. Note that the number of virtual CPU cores used by each benchmark during the experimentation varies. SSD_TF benchmark runs make use of 4 cores, MRCNN_TF uses 8 cores, and GNMT_TF uses 2 cores. It can be seen that as the number of TPU cores used for training increases, the active host time increases. This increase is observed to show a sub-linear trend for training SSD_TF benchmark. While for MRCNN_TF the trend seems very arbitrary. Additionally, for GNMT_TF a super-linear increase is observed moving from 2 to 4 TPU cores and a sub-linear increase when moving from 4 to 8 TPU cores. This behavior can be reasoned by the observation that while training GNMT_TF using 4 TPU cores, the proportion of the host activity involved in data preprocessing increases significantly. In general, for all the benchmarks, it is seen that the host-side activity is dominated by enqueuing of the data. A significant portion of the remaining activity is occupied with operations performing data preprocessing and reading data ahead of time. It can be said that the benchmark implementations are well optimized as the on-demand reads are rarely observed.

Table 4.5: Percentage of the time when the host is active on various TPU-enabled systems.

Data Type	**Active Host Time (%)**			
bfloat16	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	4.9	10.1	15	28.2
MRCNN_TF	2.9	4	5.8	17.9
GNMT_TF			14.5	29.6
bfloat16	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	7.9	10.6	20.1	50.5
MRCNN_TF	2.7	8.7	23.6	44.6
GNMT_TF	4.4	8.9	23.4	30.4
float32	**v2-1**	**v2-2**	**v2-4**	**v2-8**
SSD_TF	5.1	7.1	11.6	25.2
MRCNN_TF	7.4	8.1	10.7	10.9
GNMT_TF				29.6
float32	**v3-1**	**v3-2**	**v3-4**	**v3-8**
SSD_TF	6.7	14.7	26.6	33.3
MRCNN_TF	3.9	4.4	13.2	37.2
GNMT_TF		10.2	25.7	31.4

Chapter 5

Conclusion

This thesis studies the training of MLPerf benchmarks, an emerging suite of deep learning workloads, on multi-GPU and multi-TPU platforms. The experiments point towards (i) the importance of powerful interconnects in multi-GPU systems, (ii) the variation in scalability exhibited by different ML models, (iii) the opportunity for smart scheduling strategies in distributed training exploiting the variability in scaling efficiency, (iv) the significance of mixed precision training, (v) performance improvement from utilizing the newer generation of TPUs, and (vi) the need for powerful CPUs (as hosts) when the number of GPUs or TPUs increases.

The uniqueness and coverage of MLPerf benchmarks in the performance spectrum are also examined. This thesis presents the dissimilarity of the MLPerf benchmarks to other benchmarks in the suite (intra-suite dissimilarity) and dissimilarity against other suites such as DAWNBench and DeepBench (inter-suite dissimilarity). MLPerf provides benchmarks with moderately high memory transactions per second and moderately high compute rates. In contrast, DAWNBench creates a high-compute benchmark with a low memory transaction rate, whereas DeepBench provides low compute rate benchmarks.

The uniqueness of the MLPerf benchmarks is evident in the high NVLink utilization in NCF_Py, low NVLink utilization in SSD_Py, near-perfect scalability with increasing GPU counts in Res50_TF and SSD_Py, and low scalability in NCF_Py. MRCNN_Py makes only 1.5× improvement with tensor cores and reduced precision, whereas Res50_TF makes 3.3× improvement. Our characterization indicates a diverse set of benchmarks inside the MLPerf suite. I expect these benchmarks to stress ML training hardware for years to come and spawn research in hardware and software optimizations.

Bibliography

[1] Minigo: A minimalist Go engine modeled after AlphaGo Zero, built on MuGo. `https://github.com/tensorflow/minigo`.

[2] XLA (accelerated linear algebra). `https://www.tensorflow.org/xla/jit`.

[3] MLPerf. `https://mlperf.org/`, 2018.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.

[5] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference Workloads for Modern Deep Learning Methods, 2016.

[6] Baidu. An update to DeepBench with a focus on deep learning inference, 2017.

[7] Baidu. DeepBench: Benchmarking Deep Learning operations on different hardware, 2017.

[8] bkj. Resnet18 + minor modifications (submission at DAWNBench). `https://github.com/bkj/basenet/tree/49b2b61e5b9420815c64227c5a10233267c1fb14/examples`, 2018.

[9] 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 efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning, 2014.

[11] Cody A. Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. DAWNBench : An End-to-End Deep Learning Benchmark and Competition. In *NIPS ML Systems Workshop*, 2017.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications, 2014.

[13] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.

[14] FUJITSU. White Paper FUJITSU Server PRIMERGY & PRIMEQUEST Memory Performance of Xeon scalable processor(Skylake-SP) based Systems. `https://sp.ts.fujitsu.com/dmsp/Publications/public/wp-skylake-memory-performance-ww-en.pdf`, 2018.

[15] Google. The bfloat16 floating-point format. `https://cloud.google.com/tpu/docs/bfloat16#the_bfloat16_floating-point_format`.

[16] Google. TPU System Architecture. `https://cloud.google.com/tpu/docs/system-architecture`.

[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision, 2015.

[18] Linley Gwennap. AI Benchmarks Remain Immature. *Microprocessor Report*, January 28, 2019.

[19] K. He, G. Gkioxari, P. Dollar, and R. Girshick. Mask R-CNN. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2018.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, 2015.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks, 2016.

[22] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural Collaborative Filtering, 2017.

[23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations, 2016.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Er-

ick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`, 2009.

[26] Urs Köster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Oğuz H. Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks, 2017.

[27] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector, 2015.

[28] Peter Mattson. MLPerf Design Challenges. *FastPath 2019, ISPASS*, 2019.

[29] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta,

Kim Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark, 2019.

[30] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training, 2017.

[31] Microway. Performance Characteristics of Common Transports and Buses. `https://www.microway.com/knowledge-center-articles/performance-characteristics-of-common-transports-buses/`, 2019.

[32] NVIDIA. NVIDIA Collective Communications Library (NCCL). `https://developer.nvidia.com/nccl`.

[33] NVIDIA. NVIDIA Tesla V100 GPU Accelerator. `https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf`, 2018.

[34] NVIDIA. Automatic Mixed Precision (AMP). `https://developer.nvidia.com/automatic-mixed-precision`, 2019.

[35] NVIDIA Corporation. NVIDIA System Management Interface program. `https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf`, 2016.

[36] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[37] Ramesh Radhakrishnan, Snehil Verma, Qinzhe Wu, Bagus Hanindhito, Gunjan Jha, Eugene B. John, and Lizy K. John. Demystifying Hardware Infrastructure Choices for Deep Learning Using MLPerf. *NVIDIA GPU Technology Conference (GTC)*, 2019.

[38] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text, 2016.

[39] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan,

Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark, 2019.

[40] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[41] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017.

[42] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 10 2017.

[43] The FreeBSD Project. Iostat: I/O statistics tool. `https://www.freebsd.org/cgi/man.cgi?query=iostat&manpath=FreeBSD+12.0-`

RELEASE+and+Ports.

[44] The FreeBSD Project. Netstat: Network status and statistics tool. https://www.freebsd.org/cgi/man.cgi?query=netstat&sektion=1&manpath=FreeBSD+12.0-RELEASE+and+Ports.

[45] The FreeBSD Project. Vmstat: Virtual memory statistics tool. https://www.freebsd.org/cgi/man.cgi?query=vmstat&sektion=8&manpath=FreeBSD+12.0-RELEASE+and+Ports.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2017.

[47] Snehil Verma, Qinzhe Wu, Bagus Hanindhito, Gunjan Jha, Eugene B. John, Ramesh Radhakrishnan, and Lizy K. John. Demystifying the MLPerf Benchmark Suite. 2019.

[48] Snehil Verma, Qinzhe Wu, Bagus Hanindhito, Gunjan Jha, Eugene B. John, Ramesh Radhakrishnan, and Lizy K. John. Demystifying the MLPerf Training Benchmark Suite. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2020.

[49] Vasilis Vryniotis. NVIDIA GPU Utilization plugin for dstat. https://raw.githubusercontent.com/datumbox/dstat/master/plugins/dstat_nvidia_gpu.py, 2017.

[50] Dag Wieërs. Dstat: Versatile resource statistics tool. `http://dag.wiee.rs/home-made/dstat/`.

[51] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[52] D. Williamson. Dynamically scaled fixed point arithmetic. In *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*, pages 315–318 vol.1, 1991.

[53] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016.

[54] Charlene Yang. Berkeley CS Roofline Toolkit. `https://bitbucket.org/berkeleylab/cs-roofline-toolkit`.

[55] Runqi Yang, Facebook-ParlAI, and Brett Koonce. DrQA (submission at DAWNBench). `https://github.com/hitvoice/DrQA`, 2018.

[56] Cliff Young. Why Machine Learning Needs Benchmarks. *Computer Architecture Today, ACM SIGARCH*, 2018.

[57] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. TBD: Benchmarking and Analyzing Deep Neural Network Training, 2018.