

© 2020 Cheng Li

PERFORMANCE BENCHMARKING, ANALYSIS AND OPTIMIZATION OF DEEP
LEARNING INFERENCE

BY

CHENG LI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei Hwu, Chair
Assistant Professor Christopher W. Fletcher
Professor David A. Padua
Dr. Wei Tan, Citadel

ABSTRACT

The world sees a proliferation of deep learning (DL) models and their wide adoption in different application domains. This has made the performance benchmarking, understanding, and optimization of DL inference an increasingly pressing task for both hardware designers and system providers, as they would like to offer the best possible computing system to serve DL models with the desired latency, throughput, and energy requirements while maximizing resource utilization. However, DL faces the following challenges in performance engineering.

Benchmarking — While there have been significant efforts to develop benchmark suites that evaluate widely used DL models, developing, maintaining, and running benchmarks takes a non-trivial amount of effort, and DL benchmarking has been hampered in part due to the lack of representative and up-to-date benchmarking suites.

Performance Understanding — Understanding the performance of DL workloads is challenging as their characteristics depend on the interplay between the models, frameworks, system libraries, and the hardware (or the HW/SW stack). Existing profiling tools are disjoint, however, and only focus on profiling within a particular level of the stack. This largely limits the types of analysis that can be performed on model execution.

Optimization Advising — The current DL optimization process is manual and ad-hoc that requires a lot of effort and expertise. Existing tools lack the highly desired abilities to characterize ideal performance, identify sources of inefficiency, and quantify the benefits of potential optimizations. Such deficiencies have led to slow DL characterization/optimization cycles that cannot keep up with the fast pace at which new DL innovations are introduced.

Evaluation and Comparison — The current DL landscape is fast-paced and is rife with non-uniform models, hardware/software (HW/SW) stacks, but lacks a DL benchmarking platform to facilitate evaluation and comparison of DL innovations, be it models, frameworks, libraries, or hardware. Due to the lack of a benchmarking platform, the current practice of evaluating the benefits of proposed DL innovations is both arduous and error-prone — stifling the adoption of the innovations.

This thesis addresses the above challenges in DL performance engineering. First we introduce DLBricks, a composable benchmark generation design that reduces the effort of developing, maintaining, and running DL benchmarks. DLBricks decomposes DL models into a set of unique runnable networks and constructs the original model’s performance using the performance of the generated benchmarks. Then, we present XSP, an across-stack profiling design that correlates profiles from different sources to obtain a holistic and hier-

archical view of DL model execution. XSP innovatively leverages distributed tracing and accurately capture the profiles at each level of the HW/SW stack in spite of profiling overhead. Next, we propose Benanza, a systematic DL benchmarking and analysis design that guides researchers to potential optimization opportunities and assesses hypothetical execution scenarios on GPUs. Finally, we design MLModelScope, a consistent, reproducible, and scalable DL benchmarking platform to facilitate evaluation and comparison of DL innovations. This thesis also briefly discusses TrIMS, TOPS, and CommScope which are developed based on the needs observed from the performance benchmarking and optimization work to solve relevant problems in the DL domain.

To my family, for their love and support.

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor Professor Wen-mei Hwu for his support of my Ph.D. study. Five years ago, Wen-mei kindly admitted me to University of Illinois at Urbana-Champaign (UIUC). I am honored to be a member of the IMPACT (Illinois Microarchitecture Project using Algorithms and Compiler Technology) Research Group ever since. Wen-mei provided me with valuable guidance on my research. He also encouraged and sponsored me to attend many conferences and events to present my work and get inspired. Without Wen-mei, I could not have made it here and become a researcher.

Besides my advisor, I would also like to thank the rest of my doctoral committee: Assistant Professor Christopher W. Fletcher, Professor David A. Padua, and Dr. Wei Tan, for their insightful comments and suggestions on my thesis work.

Next I would like to thank Abdul Dakkak, my colleague and collaborator, for teaching me a lot of research/engineering skills that I will benefit from for the rest of my life. As a senior member of the research group, Abdul did a great job mentoring me. I am also deeply grateful to other group members for their support and companionship in this journey.

My sincere thanks also go to my collaborator Jinjun Xiong at IBM Research. Jinjun is a very good researcher and gives useful feedback on many of my research projects. He is always willing to help. There were many times when I worked on paper submissions with tight deadlines and Jinjun helped as much as he could until the last minute.

Finally, I would like to acknowledge with gratitude, the love and support of my family — my parents, Ruihai and Huazhi; my brother and his wife, Bo and Xiaoxu. They are always there for me no matter what happens. Without their encouragement, I would not have overcome the challenges I encountered during my study. This thesis would not have been possible without them. I am extremely lucky to have them in my life.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	DL INFERENCE	5
CHAPTER 3	DLBRICKS: COMPOSABLE BENCHMARK GENERATION TO REDUCE DEEP LEARNING BENCHMARKING EFFORT	6
3.1	Motivation	8
3.2	Design	14
3.3	Evaluation	17
3.4	Related Work	20
3.5	Discussion and Future Work	21
3.6	Conclusion	23
CHAPTER 4	XSP: UNDERSTANDING DL PERFORMANCE ACROSS STACK	24
4.1	ML Profiling on GPUs and Related Work	26
4.2	XSP Design and Implementation	28
4.3	Evaluation	40
4.4	Conclusion	48
CHAPTER 5	BENANZA: AUTOMATIC μ BENCHMARK GENERATION TO COMPUTE “LOWER-BOUND” LATENCY AND INFORM OPTIMIZATIONS OF DEEP LEARNING MODELS	49
5.1	Motivation	51
5.2	Benanza Design and Implementation	55
5.3	Evaluation	62
5.4	Related Work	72
5.5	Conclusion	73
CHAPTER 6	MLMODELSCOPE: THE DESIGN AND IMPLEMENTATION OF A SCALABLE DL BENCHMARKING PLATFORM	74
6.1	Design Objectives	76
6.2	MLModelScope Design and Implementation	78
6.3	Evaluation	88
6.4	Related Work	94
6.5	Conclusion	95
CHAPTER 7	OTHER RELEVANT WORKS	96
7.1	TrIMS: Transparent and Isolated Model Sharing for DL Inference	96
7.2	TOPS: Accelerating Reduction and Scan Using Tensor Core Units	99
7.3	CommScope	101

CHAPTER 8 CONCLUSION	103
REFERENCES	104

CHAPTER 1: INTRODUCTION

The past few years have seen a spur of deep learning (DL) innovations. These innovations span from DL models to software stack optimizations (e.g., frameworks such as MXNet or PyTorch, libraries such as cuDNN or MKL-DNN) and hardware stack improvements (e.g. CPU, GPU, FPGA). Among all the innovations, however, DL models are the most rapidly evolving and prolific. This is true in both academia [1] and industry [2], where models are tweaked and introduced on a weekly, daily, or even hourly basis. There have been numerous impressive advances in applying DL in many application domains such as image classification, object detection, machine translation, etc.

This has resulted in a surge of interest in deploying these DL models within various computing platforms/devices including commodity servers, accelerators, reconfigurable hardware, mobile and edge devices. Therefore, there is an increasing need for hardware providers, computer architects, and system/chip designers to benchmark, understand and optimize DL model inference performance (throughput, latency, system resource utilization, etc.) across different computing systems. However, DL inference performance engineering faces the following challenges, which stifle the adoption of DL innovations.

Developing, maintaining, and running DL benchmarks — For each DL task of interest, benchmark suite authors select a small subset (or one) out of tens or even hundreds of candidate models. Deciding on a representative set of models is an arduous effort as it takes a long debating process to determine what models to add and what to exclude. For example, it took over a year of weekly discussion to determine and publish MLPerf v0.5 inference models, and the number of models was reduced from the 10 models originally considered to 5. Given that DL models are proposed or updated on a daily basis [1, 2], it is very challenging for benchmark suites to be agile and remain representative of real-world DL model usage. Moreover, only publicly available models are considered for inclusion in benchmark suites. Proprietary models are trade secrets or restricted by copyright and cannot be shared externally for benchmarking. Thus, proprietary models are not included or represented within benchmark suites. Due to these issues, DL benchmarking has been hampered in part due to the lack of representative and up-to-date benchmarking suites.

Understanding DL performance across the hardware/software stack — DL model inference is complex and its performance is impacted by the interplay between different levels within the hardware/software (HW/SW) stack — frameworks, system libraries, and hardware. An example is shown in Figure 1.1. Due to the complexity of model execution, to be able to identify bottlenecks and locate their sources, one needs a holistic view

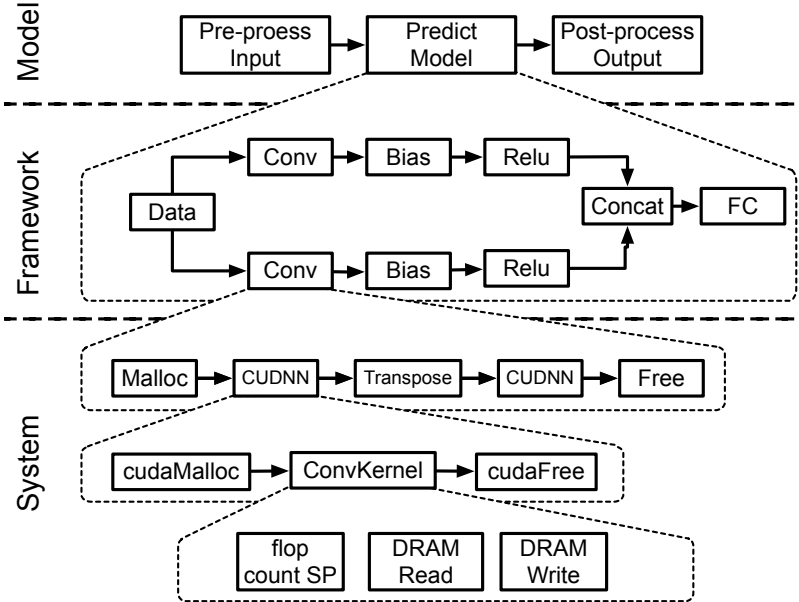


Figure 1.1: DL model performance is impacted by the interplay between different levels within the HW/SW stack - frameworks, system libraries, and hardware.

of the model execution. However, existing profiling tools or methods only provide partial views of model execution.

Interpreting DL benchmarking results into possible optimizations — Both industry and academia have invested heavily in developing benchmarks to characterize DL models and systems [3–7]. The characterization is followed by optimization to improve the model performance. However, there is currently a gap between the benchmarking results and possible optimizations to perform. Researchers use profilers, such as nvprof [8], Nsight [9], and VTune [10], to profile DL model execution and get low-level GPU and CPU information. With ample knowledge of how models execute and utilize system resources, researchers manually identify bottlenecks and inefficiencies within model execution by examining the profiling results. Researchers then make hypotheses of solutions and try out different ideas to optimize the model execution — which may or may not pan out. This manual and ad-hoc process requires a lot of effort, expertise, and guesswork, and slows down the turnaround time for model optimization and system tuning. Thus, there is a need for a systematic DL benchmarking and subsequent analysis design that can guide researchers to optimization opportunities and assess hypothetical execution scenarios.

Consistent, reproducible, and scalable DL experimentation — The DL landscape is fast-paced and is rife with non-uniform models, HW/SW stacks, but lacks a DL experimentation platform to facilitate the evaluation and comparison of DL innovations, be it models, frameworks, libraries, or hardware. To consistently evaluate two DL benchmarks requires one

to use the same evaluation code and HW/SW environment. However, DL benchmarks are often developed independently as a set of ad-hoc scripts. Thus, a fair comparison requires a non-trivial amount of effort. Furthermore, DL benchmarking often requires evaluating models across different combinations of HW/SW stacks. As HW/SW stacks are increasingly being proposed, there is an urging need for a DL benchmarking platform that consistently evaluates and compares different DL models across HW/SW stacks, while coping with the fast-paced and diverse landscape of DL.

The thesis addresses the above challenges through novel DL benchmarking, analysis, and optimization designs, and is organized as follows:

- Chapter 2 describes DL inference in detail.
- Chapter 3 presents DLBricks, a composable benchmark generation design that decomposes DL models into a set of unique runnable networks and constructs the original model’s inference performance using the performance of the generated benchmarks. DLBricks reduces the effort to develop, maintain, and run DL benchmarks.
- Chapter 4 presents XSP, an across-stack profiling design that innovatively leverages distributed tracing to construct a holistic and hierarchical view of DL model execution without modification to frameworks. XSP accurately captures the profiles at each level of the stack in spite of the profiling overhead incurred from the profilers. XSP addresses the challenge of understanding DL performance across the HW/SW stack and provides insights that are difficult to discern without it.
- Chapter 5 presents Benanza, a systematic DL benchmarking and analysis design to inform DL inference optimizations on GPUs. Benanza automatically generates micro-benchmarks given a set of models, computes their “lower-bound” latencies using the benchmark data, and informs optimizations of their executions on GPUs. Benanza guides researchers to optimization opportunities and assesses hypothetical execution scenarios on GPUs.
- Chapter 6 presents MLModelScope, a consistent, reproducible, and scalable DL experimentation platform to facilitate evaluation and comparison of DL innovations. MLModelScope offers a unified and holistic way to evaluate, compare and introspect DL inference, and provides an automated analysis and reporting workflow to summarize the results.
- Chapter 7 discusses several other works relevant to DL performance which include TrIMS — removing the model loading overhead from the DL inference by exploiting

sharing of models across the memory hierarchy in the cloud, TOPS — leveraging Tensor Core Units to accelerate non-GEMM primitives that are common in DL operators, and CommScope — understanding memory transfer behavior across different data placement and exchange scenarios.

- Chapter 8 offers concluding remarks and points to future directions.

CHAPTER 2: DL INFERENCE

A DL *model* is defined by its graph topology and its weights. The graph topology is defined as a set of nodes where each node is a function operator with the implementation provided by a *framework* (e.g. TensorFlow, MXNet, PyTorch). A DL inference pipeline includes the pre-processing, prediction, and post-processing steps. Pre-processing is the process of transforming the user input into a form that can be consumed by the model and post-processing is the process of transforming the model’s output to compute metrics. If we take image classification shown in Figure 2.1 as an example, the pre-processing step decodes the input image into a tensor of dimensions $[batch, height, width, channel]$ ($[N, H, W, C]$), then performs resizing, normalization, etc. The image classification model’s output is a tensor of dimensions $[batch * numClasses]$ which is sorted to get the top K predictions (label with probability).

In the model prediction step, the framework acts as a “runtime” and maps the function operators into *system library* calls. The layers executed by a framework are pipelines of system library calls. The system libraries, in turn, invoke a chain of primitive kernels that impact the underlying hardware counters. As can be observed, this inference pipeline is intricate and has many *levels* of abstraction — frameworks, system libraries, and hardware, as summarized in Figure 1.1. DL inference performance is impacted by the interplay between these different HW/SW stack levels. When a slowdown is observed, any one of them can be suspect.

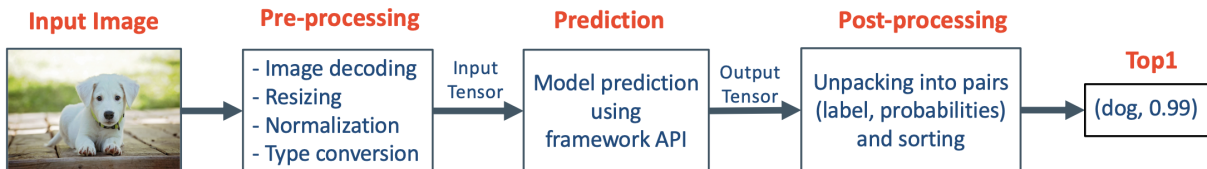


Figure 2.1: Image classification model inference pipeline.

CHAPTER 3: DLBRICKS: COMPOSABLE BENCHMARK GENERATION TO REDUCE DEEP LEARNING BENCHMARKING EFFORT

This chapter presents DLBricks, a composable benchmark generation design that reduces the effort of developing, maintaining, and running DL benchmarks. DLBricks decomposes DL models into a set of unique runnable networks and constructs the original model’s performance using the performance of the generated benchmarks. DLBricks can keep up-to-date with the latest proposed models, relieving the pressure of selecting representative DL models.

The recent progress made by Deep Learning (DL) in a wide array of applications, such as autonomous vehicles, face recognition, object detection, machine translation, fraud detection, etc., has led to increased public interest in DL models. Benchmarking these trained DL models before deployment is critical, as DL models must meet target latency and resource constraints. Hence, there have been significant efforts to develop benchmark suites that evaluate widely used DL models [3, 4, 11, 12]. An example is MLPerf [3], which is formed as a collaboration between industry and academia and aims to provide reference implementations for DL model training and inference.

However, developing, maintaining, and running benchmarks takes a non-trivial amount of effort. For each DL task of interest, benchmark suite authors select a small representative subset (or one) out of tens or even hundreds of candidate models. Deciding on a representative set of models is an arduous effort as it takes a long debating process to determine what models to add and what to exclude. For example, it took over a year of weekly discussion to determine and publish MLPerf v0.5 inference models, and the number of models was reduced from the 10 models originally considered to 5. Figure 3.1 shows the gap between the number of DL papers [13] and the number of models included in recent benchmarking efforts. Given that DL models are proposed or updated on a daily basis [1, 2], it is very challenging for benchmark suites to be agile and representative of real-world DL model usage. Moreover, only public available models are considered for inclusion in benchmark suites. Proprietary models are trade secrets or restricted by copyright and cannot be shared externally for benchmarking. Thus, proprietary models are not included or represented within benchmark suites.

To address the above issues, we propose DLBricks — a composable benchmark generation design that reduces the effort to develop, maintain, and run DL benchmarks. Given a set of DL models, DLBricks parses them into a set of atomic (i.e. non-overlapping) unique layer sequences based on the user-specified *benchmark granularity* (G). A *layer sequence* is a chain of layers. Two layer sequences are considered the *same* (i.e. not *unique*) if they are

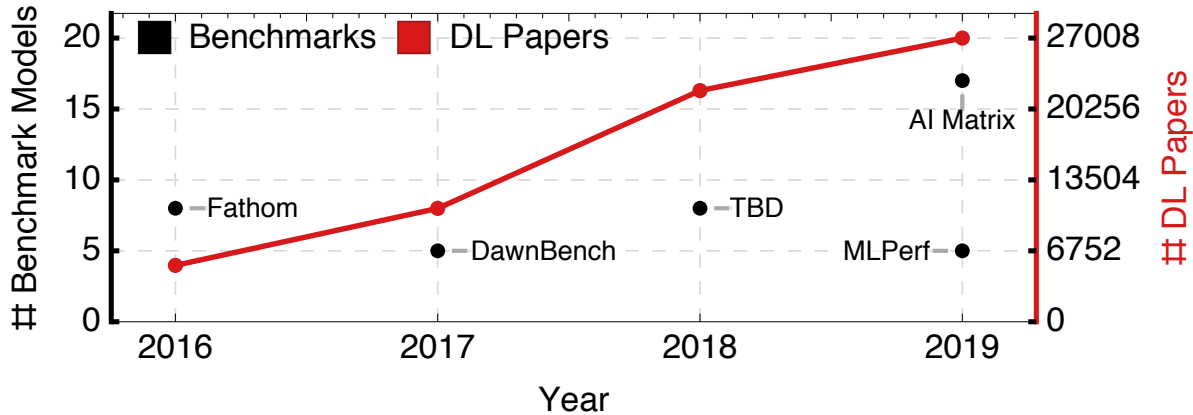


Figure 3.1: The number of DL models included in the recent published DL benchmark suites (Fathom [11], DawnBench [6], TBD [12], AI Matrix [4], and MLPerf [3]) compared to the number of DL papers published in the same year (using Scopus Preview [13]).

identical ignoring their weight values. DLBricks then generates unique *runnable networks* (i.e. subgraphs of the model with at most G layers that can be executed by a framework) using the layer sequences’ information, and these networks form the representative set of benchmarks for the input models. Users run the generated benchmarks on a system of interest and DLBricks uses the benchmark results to construct a performance estimate on that system.

DLBricks leverages two key observations on DL inference: ❶ Layers are the performance building blocks of the model performance. ❷ Layers (considering their layer type, shape, and parameters, but ignoring the weights) are extensively repeated within and across DL models. DLBricks uses both observations to generate a representative benchmark suite, minimize the time to benchmark, and estimate a model’s performance from layer sequences.

Since benchmarks are generated automatically by DLBricks, benchmark development and maintenance effort are greatly reduced. DLBricks is defined by a set of simple consistent principles and can be used to benchmark and characterize a broad range of models. Moreover, since each generated benchmark represents only the nodes of the input model, the input model’s topology does not appear in the output benchmarks. This, along with the fact that “fake” or dummy models can be inserted into the set of input models, means that the generated benchmarks can represent proprietary models without the concern of revealing proprietary models.

In summary, this work makes the following contributions:

- We perform a comprehensive performance analysis of 50 state-of-the-art DL models on CPUs and observe that layers are the performance building blocks of DL models, thus a model’s performance can be estimated using the performance of its layers (Section 3.1.1).

- We also perform an in-depth DL architecture analysis of the DL models and make the observation that DL layers with the same type, shape, and parameters are repeated extensively within and across models (Section 3.1.2).
- We propose DLBricks, a composable benchmark generation design that decomposes DL models into a set of unique runnable networks and constructs the original model’s performance using the performance of the generated benchmarks (Section 3.2).
- We evaluate DLBricks using 50 MXNet models spanning 5 DL tasks on 4 representative CPU systems (Section 3.3). We show that DLBricks provides a tight performance estimate for DL models and reduces the benchmarking time across systems. The composed model latency is within 95% of the actual performance while up to $4.4\times$ reduction in benchmarking time is achieved on the Amazon EC2 `c5.xlarge` system.

This chapter is structured as follows. First, we detail two key observations that enable our design in Section 3.1. We then propose DLBricks in Section 3.2 and describe how it provides a streamlined benchmark generation workflow which lowers the effort to benchmark. Section 3.3 evaluates using 50 models running on 4 systems. In Section 3.4 we describe different benchmarking approaches previously performed. We then describe future work in Section 3.5 before we conclude in Section 3.6.

3.1 MOTIVATION

DLBricks is designed based on two key observations presented in this section. To demonstrate and support these observations, we perform a comprehensive performance and architecture analysis of state-of-the-art DL models. The evaluations in this section use 50 MXNet models of different DL tasks (listed in Table 3.1) and were run with MXNet (v1.5.1 MKL release) on an Amazon `c5.2xlarge` instance (as listed in Table 3.2). We focus on latency sensitive (batch size = 1) DL inference on CPUs.

3.1.1 Layers as the Performance Building Blocks

A DL model is a directed acyclic graph (DAG) where each vertex within the DAG is a layer (i.e. operator, such as convolution, batchnormalization, pooling, element-wise, softmax) and an edge represents the transfer of data. For a DL model, a *layer sequence* is defined as a simple path within the DAG containing one or more vertices. A *subgraph*, on the other hand, is defined as a DAG composed of one or more layers within the model (i.e. subgraph is a superset of layer sequence, and may or may not be a simple path). We are

Table 3.1: The 50 MXNet models [14] used for evaluation, including Image Classification (IC), Image Processing (IP), Object Detection (OD), Regression (RG) and Semantic Segmentation (SS) tasks.

ID	Name	Task	Num Layers
1	Ademxapp Model A Trained on ImageNet Competition Data	IC	142
2	Age Estimation VGG-16 Trained on IMDB-WIKI and Looking at People Data	IC	40
3	Age Estimation VGG-16 Trained on IMDB-WIKI Data	IC	40
4	CapsNet Trained on MNIST Data	IC	53
5	Gender Prediction VGG-16 Trained on IMDB-WIKI Data	IC	40
6	Inception V1 Trained on Extended Salient Object Subitizing Data	IC	147
7	Inception V1 Trained on ImageNet Competition Data	IC	147
8	Inception V1 Trained on Places365 Data	IC	147
9	Inception V3 Trained on ImageNet Competition Data	IC	311
10	MobileNet V2 Trained on ImageNet Competition Data	IC	153
11	ResNet-101 Trained on ImageNet Competition Data	IC	347
12	ResNet-101 Trained on YFCC100m Geotagged Data	IC	344
13	ResNet-152 Trained on ImageNet Competition Data	IC	517
14	ResNet-50 Trained on ImageNet Competition Data	IC	177
15	Squeeze-and-Excitation Net Trained on ImageNet Competition Data	IC	874
16	SqueezeNet V1.1 Trained on ImageNet Competition Data	IC	69
17	VGG-16 Trained on ImageNet Competition Data	IC	40
18	VGG-19 Trained on ImageNet Competition Data	IC	46
19	Wide ResNet-50-2 Trained on ImageNet Competition Data	IC	176
20	Wolfram ImageIdentify Net V1	IC	232
21	Yahoo Open NSFW Model V1	IC	177
22	AdaIN-Style Trained on MS-COCO and Painter by Numbers Data	IP	109
23	Colorful Image Colorization Trained on ImageNet Competition Data	IP	58
24	ColorNet Image Colorization Trained on ImageNet Competition Data	IP	62
25	ColorNet Image Colorization Trained on Places Data	IP	62
26	CycleGAN Apple-to-Orange Translation Trained on ImageNet Competition Data	IP	94
27	CycleGAN Horse-to-Zebra Translation Trained on ImageNet Competition Data	IP	94
28	CycleGAN Monet-to-Photo Translation	IP	94
29	CycleGAN Orange-to-Apple Translation Trained on ImageNet Competition Data	IP	94
30	CycleGAN Photo-to-Cezanne Translation	IP	96
31	CycleGAN Photo-to-Monet Translation	IP	94
32	CycleGAN Photo-to-Van Gogh Translation	IP	96
33	CycleGAN Summer-to-Winter Translation	IP	94
34	CycleGAN Winter-to-Summer Translation	IP	94
35	CycleGAN Zebra-to-Horse Translation Trained on ImageNet Competition Data	IP	94
36	Pix2pix Photo-to-Street-Map Translation	IP	56
37	Pix2pix Street-Map-to-Photo Translation	IP	56
38	Very Deep Net for Super-Resolution	IP	40
39	SSD-VGG-300 Trained on PASCAL VOC Data	OD	145
40	SSD-VGG-512 Trained on MS-COCO Data	OD	157
41	YOLO V2 Trained on MS-COCO Data	OD	106
42	2D Face Alignment Net Trained on 300W Large Pose Data	RG	967
43	3D Face Alignment Net Trained on 300W Large Pose Data	RG	967
44	Single-Image Depth Perception Net Trained on Depth in the Wild Data	RG	501
45	Single-Image Depth Perception Net Trained on NYU Depth V2 and Depth in the Wild Data	RG	501
46	Single-Image Depth Perception Net Trained on NYU Depth V2 Data	RG	501
47	Unguided Volumetric RG Net for 3D Face Reconstruction	RG	1029
48	Ademxapp Model A1 Trained on ADE20K Data	SS	141
49	Ademxapp Model A1 Trained on PASCAL VOC2012 and MS-COCO Data	SS	141
50	Multi-scale Context Aggregation Net Trained on CamVid Data	SS	53

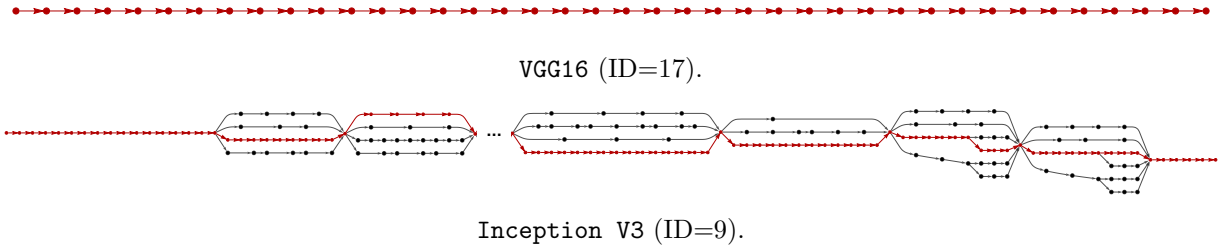


Figure 3.2: The model architecture of VGG16 (ID=17) and Inception V3 (ID=9). The critical path is highlighted in red.

only interested in network subgraphs that are runnable within frameworks and we call these runnable subgraphs *runnable networks*.

DL models may contain layers that can be executed independently in parallel. The network made of these data-independent layers is called a *parallel module*. For example, Figure 3.2a shows the VGG16 [15] (ID=17) model architecture. VGG16 contains no parallel module and is a linear sequence of layers. Inception V3 [16] (ID=9) (shown in Figure 3.2b), on the other hand, contains a mix of layer sequences and parallel modules.

DL frameworks such as TensorFlow, PyTorch, and MXNet execute a DL model by running the layers within the model graph. We explore the relation between layer performance and model performance by decomposing each DL model in Table 3.1 into layers. We define a model’s *critical path* to be a simple path from the start layer to the end layer with the highest latency. For a DL model, we add all its layers’ latency and refer to the sum as the *sequential total layer latency*, since this assumes all the layers are executed sequentially by the DL framework. Theoretically, data-independent paths within a parallel module can be executed in parallel, thus we also calculate the *parallel total layer latency* by adding up the layer latencies along the critical path. The critical path of both VGG 16 (ID=17) and Inception V3 (ID=9) is highlighted in red in Figure 3.2. For models that do not have parallel modules, the sequential total layer latency is equal to the total layer latency.

For each of the 50 models, we compare both sequential and parallel total layer latency to the model’s end-to-end latency. Figure 3.3 shows the normalized latencies in both cases. For models with parallel modules, the parallel total layer latencies are much lower than the model’s end-to-end latency. The difference between the sequential total layer latencies and the models’ end-to-end latencies are small. The normalized latencies are close to 1 with a geometric metric mean of 91.8% for the sequential case. This suggests the current software/hardware stack does not exploit parallel execution of data-independent layers or overlapping of layer execution, we verified this by inspecting the source code of popular frameworks such as MXNet, PyTorch, and TensorFlow.

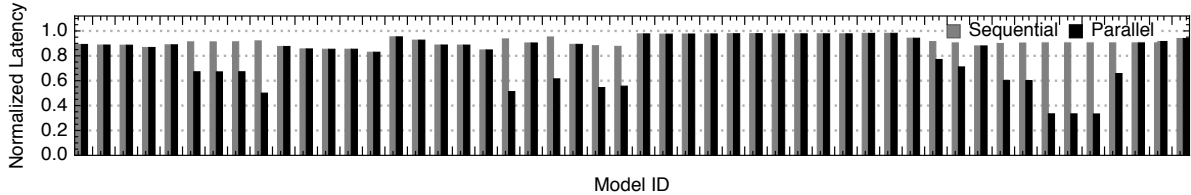


Figure 3.3: The sequential and parallel total layer latency normalized to the model’s end-to-end latency using batch size 1 on `c5.2xlarge` in Table 3.2.

The difference between a model’s end-to-end latency and its sequential total layer latency is due to the complexity of model execution within DL frameworks and the underlying software/hardware stack. We identified two major factors that may affect this difference: framework overhead and memory caching. Executing a model within frameworks introduced an overhead that is roughly proportional to the number of the layers. This is because frameworks need to perform bookkeeping, layer scheduling, and memory management for model execution. Therefore, the measured end-to-end performance can be larger than the total layer latency. On the other hand, both the framework and the underlying software/hardware stack can take advantage of caching to decrease the latency of data-dependent layers. For memory-bound layers, this can achieve significant speedup and therefore the measured end-to-end performance can be lower than the total layer latency. Depending on which factor is dominant, the normalized latency can be larger or smaller than 1. Based on this, we formulate the **1** observation:

Observation 3.1: *DL layers are the performance building blocks of the model performance, therefore, a model’s performance can be estimated using the performance of its layers. Moreover, a simple summation of layer-wise latency is an effective approximation of the end-to-end latency given the current DL software stack (no parallel execution of data-independent layers or overlapping of layer execution) on CPUs.*

3.1.2 Layer Repeatability

From a model architecture point of view, a DL layer is identified by its type, shape, and parameters. For example, a convolution layer is identified by its input shape, output channels, kernel size, stride, padding, dilation, etc. Layers with the same type, shape, parameters (i.e. only differ in weights) are expected to have the same performance. We inspected the source code of popular frameworks and verified this, as they do not perform any special optimizations for weights. Thus in this paper we consider two layers to be the *same* if they have the same type, shape, parameters, ignoring weight values, and two layers are *unique* if they are not the same.

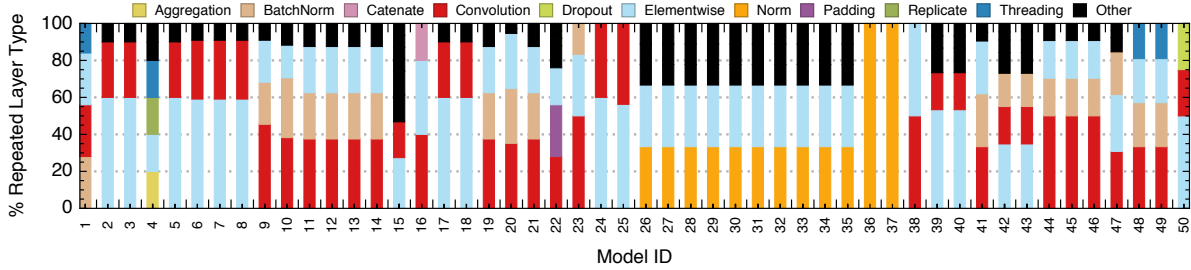


Figure 3.6: The type distribution of the repeated layers.

each model and Figure 3.6 shows their type distribution. As we can see Convolution, Elementwise, BatchNorm, and Norm are the most repeated layer types in terms of intra-model layer repeatability. If we consider all 50 models in Table 3.1, the total number of layers is 10,815, but only 1,529 are unique (i.e. 14% are unique).

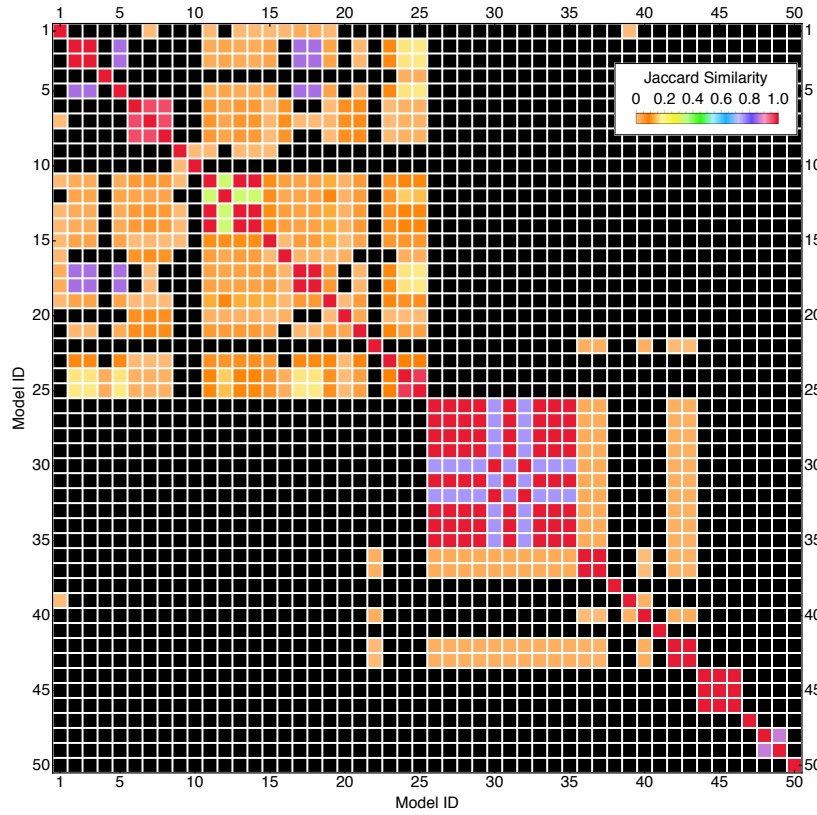



Figure 3.7: The Jaccard Similarity grid of the models in Table 3.1. Solid red indicates two models have identical layers, and black means there is no common layer.

We illustrate the layer repeatability across models by quantifying the similarity of any two models listed in Table 3.1. We use the Jaccard similarity coefficient; i.e. for any two models M_1 and M_2 the Jaccard similarity coefficient is defined by $\frac{|\mathcal{L}_1 \cap \mathcal{L}_2|}{|\mathcal{L}_1 \cup \mathcal{L}_2|}$ where \mathcal{L}_1 and \mathcal{L}_2 are the

layers of M_1 and M_2 respectively. The results are shown in Figure 3.7. Each cell corresponds to the Jaccard similarity coefficient between the models at the row and column. As shown, models that share the same base architecture but are retrained using different data (e.g. **CycleGAN*** models with IDs 26 – 35 and **Inception V1*** models with IDs 6 – 8) have many common layers. Layers are common across models within the same family (e.g. **ResNet***) since they are built from the same set of modules (e.g. **ResNet-50** is shown in Figure 3.4), or when solving the same task (e.g. the image classification task category). Based on this, we formulate the  observation:

Observation 3.2: *Layers are repeated within and across DL models. This enables us to decrease the benchmarking time since only a representative set of layers need to be evaluated.*

The above two observations suggest that if we can decompose models into layers, and then take the union of them to produce a set of representative runnable networks, then benchmarking the representative runnable networks is sufficient to construct the performance of the input models. Since we only look at the representative set, the total runtime is less than running all models directly, thus DLBricks can be used to reduce benchmarking time. Since layer decomposition elides the input model topology, models can be private while their benchmarks can be public. The next section (Section 3.2) describes how we leverage these two observations to build a benchmark generator while having a workflow where one can construct a model’s performance based on the benchmarked layer performance. We further explore the design space of benchmark granularity and its effect on performance construction accuracy.

3.2 DESIGN

This section presents the design of DLBricks , a composable benchmark generation system for DL models. The design is motivated by the two observations discussed in Section 3.1. DLBricks explores not only layer level model composition, but also sequence level composition where a *layer sequence* is a chain of layers. The *benchmark granularity* (G) specifies the maximum numbers of layers within any layer sequence in the output generated benchmarks. G is introduced to account for the effects of model execution complexity (e.g. framework overhead and caching as discussed in Section 3.1.1). Thus, a larger G is expected to increase the accuracy of performance construction. On the other hand, a larger G might decrease the layer repeatability across models. Therefore, a balance needs to be struck (by the user) between performance construction accuracy and benchmarking time speedup.

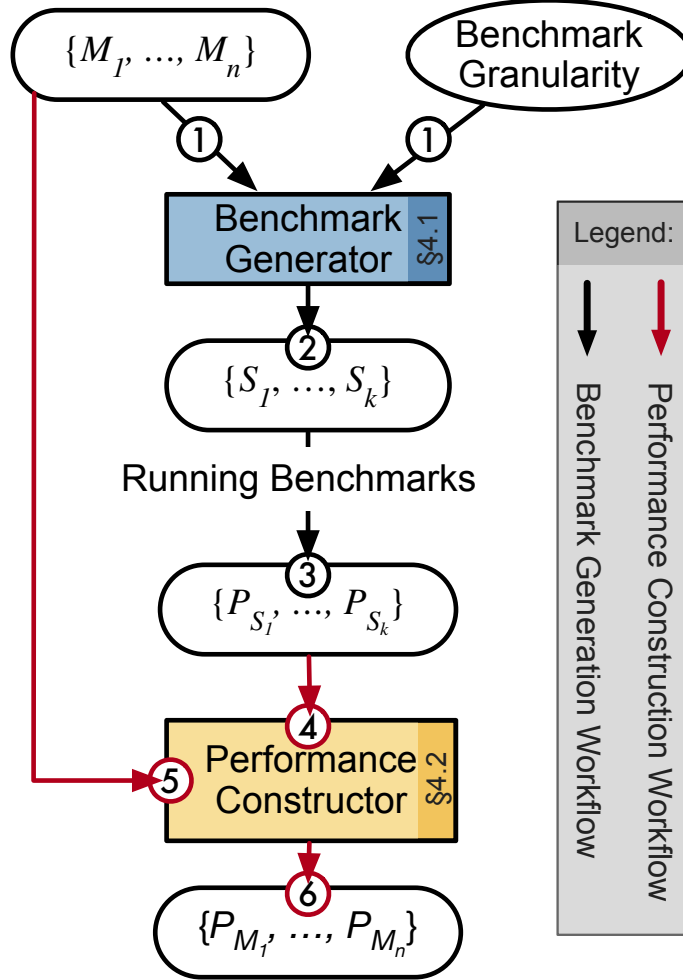


Figure 3.8: DLBricks design and workflow.

The design and workflow of DLBricks is shown in Figure 3.8. DLBricks consists of a benchmark generation workflow and a performance construction workflow. To generate composable benchmarks, one uses the *benchmark generation workflow* where: ① the user inputs a set of models (M_1, \dots, M_n) along with a target benchmark granularity. ② The benchmark generator parses the input models into a representative (unique) set of non-overlapping layer sequences and then generates a set of runnable networks (S_1, \dots, S_k) using these layer sequences' information. ③ The user evaluates the set of runnable networks on a system of interest to get each benchmark's corresponding performance $(P_{S_1}, \dots, P_{S_k})$. The benchmark results are stored and ④ are used within the *performance construction workflow*. ⑤ To construct the performance of an input model, the performance constructor queries the stored benchmark results for the layer sequences within the model, and then ⑥ computes the model's estimated performance $(P_{M_1}, \dots, P_{M_k})$. This section describes both workflows in detail.

3.2.1 Benchmark Generation

The benchmark generator takes a list of models M_1, \dots, M_n and a benchmark granularity G . The *benchmark granularity* specifies the maximum sequence length of the layer sequences generated. This means that when $G = 1$, each generated benchmark is a single-layer network, whereas when $G = 2$ each generated benchmark contains at most 2 layers.

To split a model with the specified benchmark granularity, we use `FindModelSubgraphs` (Algorithm 3.1). The `FindModelSubgraphs` takes a model and a maximum sequence length and iteratively generates a set of non-overlapping layer sequences. First, the layers in the model are sorted topologically and then call the `SplitModel` function (Algorithm 3.2) with the desired begin and end layer offset. This `SplitModel` tries to create a runnable DL network (i.e., a valid DL network) using the range of layers desired, if it fails (e.g., a network which cannot be constructed due to input/output layer shape mismatch¹), then `SplitModel` creates a network with the current layer and shifts the begin and end positions. The `SplitModel` returns a list of runnable DL networks (S_i, \dots, S_{i+j}) along with the end position to `FindModelSubgraphs`. The `FindModelSubgraphs` terminate when no other subsequences can be created.

Algorithm 3.1 The `FindModelSubgraphs` algorithm.

Input: M (Model), G (Benchmark Granularity)

Output: $Models$

```
1:  $begin \leftarrow 0, Models \leftarrow \{\}$ 
2:  $verts \leftarrow \mathbf{TopologicalOrder}(\mathbf{ToGraph}(M))$ 
3: while  $begin \leq \mathbf{Length}(verts)$  do
4:    $end \leftarrow \mathbf{Min}(begin + G, \mathbf{Length}(verts))$ 
5:    $sm \leftarrow \mathbf{SplitModel}(verts, begin, end)$ 
6:    $Models \leftarrow Models + sm["models"]$ 
7:    $begin \leftarrow sm["end"] + 1$ 
8: end while return  $Models$ 
```

The benchmark generator applies the `FindModelSubgraphs` for each of the input models. A set of representative (i.e. *unique*) runnable DL networks (S_1, \dots, S_k) is then computed. We say two sequences S_1 and S_2 are the same if they have the same topology along with the same node parameters (i.e. they are the same DL network modulo the weights). The unique networks are exported to the frameworks' network format and the user runs them

¹An example invalid network is one which contains a `Concat` layer, but does not have all of the `Concat` layer's required input layers.

with synthetic input data based on each network’s input shape. The performance of each network is stored ($P_{S_i} \dots, P_{S_k}$) and used by the performance construction workflow.

Algorithm 3.2 The `SplitModel` algorithm.

Input: $verts, begin, end$

Output: $\langle \text{“models”}, \text{“end”} \rangle$ ▷ Hash table

- 1: $vs \leftarrow verts[begin : end]$
 - 2: $m \leftarrow \text{CreateModel}(vs)$ ▷ Creates a valid model **return**
 $\langle \text{“models”} \rightarrow \{m\}, \text{“end”} \rightarrow end \rangle$ ▷ Hash table with keys: **“model”** and **“end”**
 ModelCreateException
 - 3: $m \leftarrow \{\text{CreateModel}(\{verts[begin]\})\}$ ▷ Creates a model with a single node
 - 4: $n \leftarrow \text{SplitModel}(verts, begin + 1, end + 1)$ ▷ Recrusively split the model **return**
 $\langle \text{“models”} \rightarrow m + n[\text{“models”}], \text{“end”} \rightarrow n[\text{“end”}] \rangle$
-

3.2.2 DL Model Performance Construction

DLBricks uses the performance of the layer sequences to construct an estimate to the end-to-end performance of the input model M . To construct a performance estimate, the input model is parsed and goes through the same process ① in the Figure 3.8. This creates a set of layer sequences. The performance of each layer sequence is queried from the benchmark results (P_{S_i}, \dots, P_{S_k}). DLBricks supports both sequential and parallel performance construction. Sequential performance construction is performed by summing up all the resulting queried results, whereas parallel performance construction sums up the results along the critical path of the model. Since current frameworks exhibit a sequential execution strategy (from Section 3.1.1), sequential performance construction is used within DLBricks by default. Other performance constructions can be easily added to DLBricks to accommodate different framework execution strategies.

3.3 EVALUATION

This section demonstrates that DLBricks is valid in terms of performance construction accuracy and benchmarking time speedup. We explore the effect of benchmark granularity on the constructed performance estimation as well as the benchmarking time. We evaluated DLBricks with 50 DL models (listed in Table 3.1) using MXNet (v1.5.1 using MKL v2019.3) on 4 different Amazon EC2 instances. These systems are recommended by Amazon [21]

Table 3.2: Evaluations are performed on the 4 Amazon EC2 systems listed. The `c5.*` systems operate at 3.0GHz, while the `c4.*` systems operate at 2.9GHz. The systems are ones recommended by Amazon for DL inference.

Instance	CPUS	Memory (GiB)	\$/hr
<code>c5.xlarge</code>	4 Intel Platinum 8124M	8GB	0.17
<code>c5.2xlarge</code>	8 Intel Platinum 8124M	16GB	0.34
<code>c4.xlarge</code>	4 Intel Xeon E5-2666 v3	7.5GB	0.199
<code>c4.2xlarge</code>	8 Intel Xeon E5-2666 v3	15GB	0.398

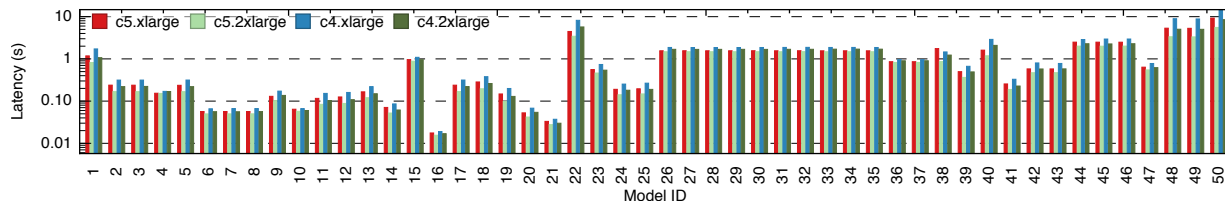


Figure 3.9: The end-to-end latency of all models in log scale across systems.

for DL inference and are listed in Table 3.2. To maintain consistent CPU evaluation, the systems are configured to disable CPU frequency scaling, turbo-boosting, scaling-governor, and hyper-threading. Each benchmark is run 100 times and the 20th percentile trimmed mean is reported.

3.3.1 Performance Construction Accuracy

We first ran the end-to-end models on the 4 systems to understand their performance characteristics, as shown in Figure 3.9. Then, using DLBricks, we constructed the latency estimate of the models based on the performance of their layer sequence benchmarks. Figure 3.10 shows the constructed model latency normalized to the model’s end-to-end latency for

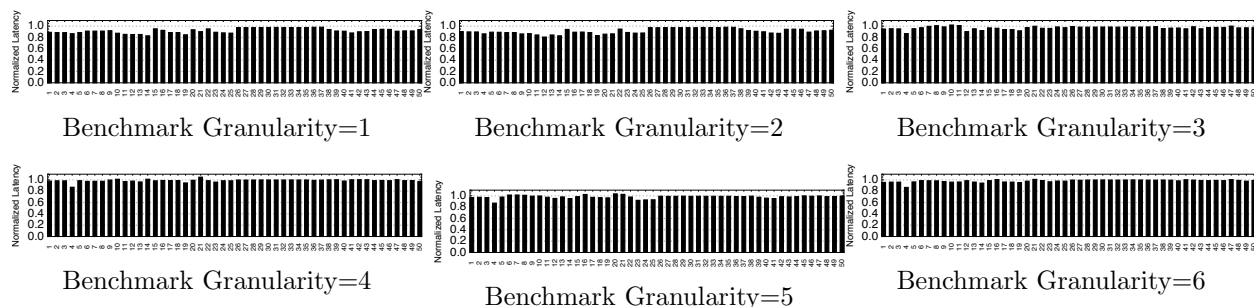


Figure 3.10: The constructed model latency normalized to the model’s end-to-end latency for the 50 model in Table 3.1 on `c5.2xlarge`. The benchmark granularity varies from 1 to 6. Sequence 1 means each benchmark has one layer (layer granularity).

all the models with varying benchmark granularity from 1 to 6 on `c5.2xlarge`. We see that the constructed latency is a tight estimate of the model’s actual performance across models and benchmark granularities. E.g., for benchmark granularity $G = 1$, the normalized latency ranges between 82.9% and 98.1% with a geometric mean of 91.8%.

As discussed in Section 3.1.1, the difference between a model’s end-to-end latency and its constructed latency is due to the combinational effect of model execution complexity such as framework overhead and caching, thus the normalized latency can be either below or above 1. For $G = 1$ (layer granularity model decomposition and construction), where a model is decomposed into the largest number of sequences, the constructed latency is slightly less accurate compared to other G values. Using the number of layers in Table 3.1 and the model end-to-end latency in Figure 3.9, we see no direct correlation between the performance construction accuracy, number of model layers, or end-to-end latency.

Figure 3.11 shows the geometric mean of the normalized latency (the constructed latency normalized to the end-to-end latency) of all the 50 models across systems and benchmark granularities. Model execution in a framework is system-dependent, thus the performance construction accuracy is not only model-dependent but also system-dependent. Overall, the estimated latency is within 5% (e.g., $G = 3, 5, 9, 10$) to 11% ($G = 1$) of the model end-to-end latency across systems. This demonstrates that DLBricks provides a tight estimate to the input models’ actual performance across systems.

3.3.2 Benchmarking Time Reduction

DLBricks decreases the benchmarking time by only evaluating the unique layer sequences within and across models. Recall from Section 3.1.2 that for all the 50 models, the total number of layers is 10,815, but only 1,529 are unique (i.e. 14% are unique). Figure 3.12 shows the speedup of the total benchmarking time across systems as benchmark granularity varies. The benchmarking time speedup is calculated as the sum of the end-to-end latency of all models divided by the sum of the latency of all the generated benchmarks. Up to $4.4\times$ benchmarking time speedup is observed for $G = 1$ on the `c5.xlarge` system. The speedup decreases as the benchmark granularity increases. This is because as the benchmark granularity increases, the chance of having repeated layer sequences within and across models decreases.

Figure 3.11 and Figure 3.12 suggest a trade-off exists between the performance construction accuracy and benchmarking time speedup and the trade-off is system-dependent. For example, while $G = 1$ (layer granularity model decomposition and construction) produces the maximum benchmarking time speedup, the constructed latency is slightly less accurate

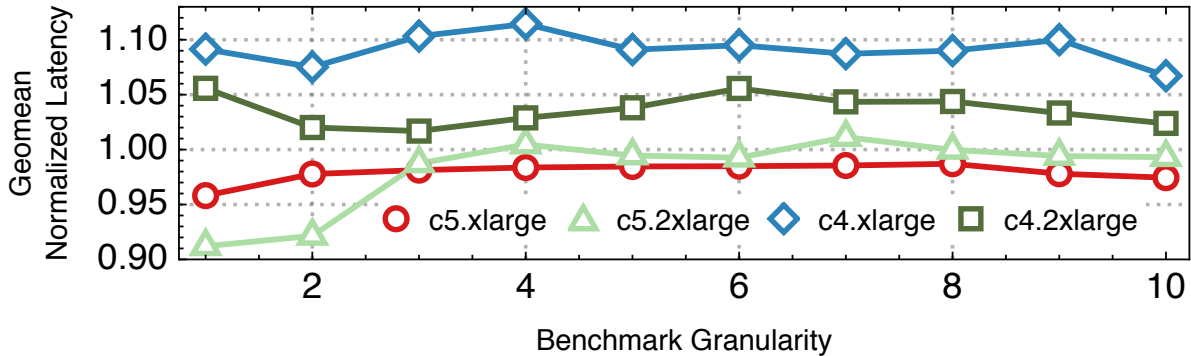


Figure 3.11: The geometric mean of the normalized latency (constructed vs end-to-end latency) of all the 50 models on the 4 systems with varying benchmark granularity from 1 to 10.

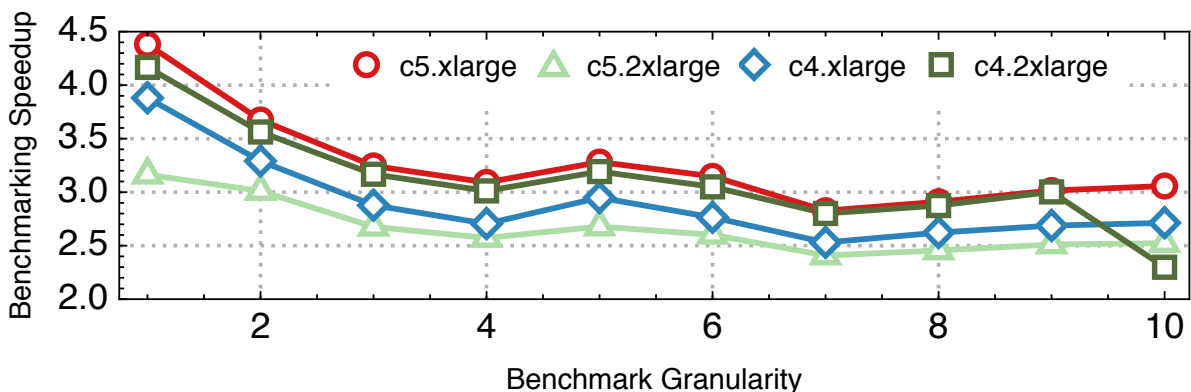


Figure 3.12: The speedup of total benchmarking time for all the models across systems and benchmark granularities.

comparing to other G values on the systems. Since this accuracy loss is small, overall $G = 1$ is a good choice of benchmark granularity configuration for DLBricks given the current DL software stack on CPUs.

3.4 RELATED WORK

To characterize the performance of DL models, both industry and academia have invested in developing benchmark suites that characterize models and systems. The benchmarking methods are either end-to-end benchmarks (performing user-observable latency measurement on a set of representative DL models [3, 4, 6]) or are micro-benchmarks [4, 5, 22] (isolating common kernels or layers that are found in models of interest). The end-to-end benchmarks target end-users and measure the latency or throughput of a model under a specific workload scenario. The micro-benchmark approach, on the other hand, distills models to their basic atomic operations (such as dense matrix multiplies, convolutions, or communication rou-

tines) and measures their performance to guide hardware or software design improvements. While both approaches are valid and have their use cases, their benchmarks are manually selected and developed. As discussed, curating and maintaining these benchmarks requires significant effort and, in the case of lack of maintenance, these benchmarks become less representative of real-world models.

DLBricks complements the DL benchmarking landscape as it introduces a novel benchmarking methodology which reduces the effort of developing, maintaining, and running DL benchmarks. DLBricks relieves the pressure of selecting representative DL models and copes well with the fast-evolving pace of DL models. DLBricks automatically decomposes DL models into runnable networks and generates micro-benchmarks based on these networks. Users can specify the benchmark granularity. At the two extremes, when the granularity is 1 a layer-based micro-benchmark is generated, whereas when the granularity is equal to the number of layers within the model then an end-to-end network is generated. To our knowledge, there has been no previous work solving the same problem and we are the first to propose such a design.

Previous work [23] also decomposed DL models into layers, but uses the results to guide performance optimization. DLBricks focuses on model performance and aims to reduce benchmarking effort. DLBricks shares similar spirit to synthetic benchmark generation [24]. However, to the authors’ knowledge, there has been no previous work on synthetic benchmark generation for DL.

3.5 DISCUSSION AND FUTURE WORK

Generating Overlapping Benchmarks — The current DLBricks design only considers non-overlapping layer sequences during benchmark generation. This may inhibit some types of optimizations (such as layer fusion). A solution requires a small tweak to Algorithm 3.1 where we increment the *begin* by 1 rather than the end index of the `SplitModel` algorithm (line 7). A small modification is also needed within the performance construction step to pick the layer sequence resulting in the smallest latency. Future work would explore the design space when the generated benchmarks can overlap.

Adapting to Framework Evolution — The current DLBricks design is based on the observation that current DL frameworks do not execute data-independent layers in parallel. Although DLBricks supports both sequential and parallel execution (assuming all data-independent layers are executed in parallel as described in Section 3.2.2), as DL frameworks start to have some support of parallel execution of data-independent layers, the current

design may need to be adjusted. To adapt DLBricks to this evolution of frameworks, one can adjust DLBricks to take user-specified parallel execution rules. DLBricks can then use the parallel execution rules to make a more accurate model performance estimation.

Sparse Models — The current DLBricks design assumes the input models are dense models. In a sparse model where the layers’ weights are sparse tensors², the sparsity pattern (i.e., the distribution of non-zeros) of a layer’s weights affects the performance of the layer. To adapt DLBricks to sparse models, we need to consider the sparsity pattern of layer weights when identifying layers as performance building blocks (Section 3.1.1) or exploring their repeatability (Section 3.1.2). Recall that the sparsity of model layers is fixed in inference, we can use the sparsity pattern signature and encode it within the layer description. This would allow us to avoid unifying layers with different sparsity patterns and correctly reflect the sparsity effects and their influence on the layer performance. Future work would explore this encoding.

Other Systems — While this work focuses on CPUs, we expect the design to hold for GPUs and other AI chips. As stated in Observation 1 in Section 3.1.1, a simple summation of layer-wise latency is an effective approximation of the model’s end-to-end latency given the current DL software stack on CPUs. On GPUs and AI chips, a summation of layer-wise latency may no longer be effective due to the more aggressive DL optimizations done on these systems. Thus, to accommodate DLBricks to GPUs or AI chips, the execution strategy used in the benchmark generation and performance construction needs to be modified to reflect the DL model execution on those systems. Future work would explore the design for GPUs and other AI chips.

Other Use Cases — We are also interested in other use cases that are afforded by the DLBricks design — model/system comparison and advising for the cloud. For example, it is common to ask questions such as, *given a DL model, which system should I use?* or *given a system and a task, which model should I use?* Using DLBricks, system providers can curate a continuously updated database of the generated benchmarks results across different system offerings. The system providers can then perform a static performance estimate of the user’s DL model (without running it) and give suggestions as to which system to choose. This catalog of benchmark performance can also be shared to the public through secured APIs.

²input feature maps are still treated as dense tensors

3.6 CONCLUSION

The fast-evolving landscape of DL poses considerable challenges in the DL benchmarking practice. While benchmark suites are under pressure to be agile, up-to-date, and representative, we take a different approach and propose a novel benchmarking design — aimed at relieving this pressure. Leveraging the key observations that layers are the performance building block of DL models and the layer repeatability within and across models, DLBricks automatically generates composable benchmarks that reduce the effort of developing, maintaining, and running DL benchmarks. Through the evaluation of state-of-the-art models on representative systems, we demonstrated that DLBricks provides a trade-off between performance construction accuracy and benchmarking time speedup. As the benchmark generation and performance construction workflows in DLBricks are fully automated, the generated benchmarks and their performance can be continuously updated and augmented as new models are introduced with minimal effort from the user. Thus DLBricks copes with the fast-evolving pace of DL models.

CHAPTER 4: XSP: UNDERSTANDING DL PERFORMANCE ACROSS STACK

This chapter proposes XSP — an across-stack profiling design that gives a holistic and hierarchical view of ML model execution. XSP leverages distributed tracing to aggregate and correlate profile data from different sources. XSP introduces a leveled and iterative measurement approach that accurately captures the latencies at all levels of the HW/SW stack in spite of the profiling overhead.

Machine learning/deep learning (ML) models are increasingly being used to solve problems across many domains such as image classification, object detection, machine translation, etc. This has resulted in a surge of interest in optimizing and deploying these models on many hardware types including commodity servers, accelerators, reconfigurable hardware, mobile/edge devices, and ASICs. As a result, there is an increasing need to profile and understand the performance of ML models.

Characterizing ML model inference is complex as its performance depends on the interplay between different levels of the HW/SW stack — frameworks, system libraries, and hardware platforms. Figure 4.1 shows an example model inference pipeline on GPUs. At the top, there is the ❶ model-level evaluation pipeline. Components at the model-level include input pre-processing, model prediction, and output post-processing. Within the model prediction step are the ❷ layer-level components — layer operators including convolution (Conv), batch normalization (BN), softmax, etc. Within each layer are the ❸ GPU kernel-level components — a sequence of CUDA API calls or GPU kernels invoked by the layer. Because of the complexities of model inference, one needs a holistic view of the execution to identify and locate performance bottlenecks.

Existing profiling tools or methods only provide a partial view of model execution. To capture a holistic view of model execution, one has to switch between an array of tools. Take the current ML profiling on GPUs for example. To measure the model-level latency, one inserts timing code around the model prediction step of the inference pipeline. To capture the layer-level information, one uses the ML framework’s profiling capabilities [25, 26]. And, to capture GPU kernel information, one uses GPU profilers such as NVIDIA’s nvprof [8] or Nsight [9]. The output profiles from the different tools are disjoint; e.g., the GPU kernels are not correlated with the layers. As a result, one cannot construct Figure 4.1 and identify that the three GPU kernels shown come from the first Conv layer, for example. This same issue exists when profiling ML model execution on CPUs.

To correlate profiled events with model layers, vendors modify ML frameworks and instru-

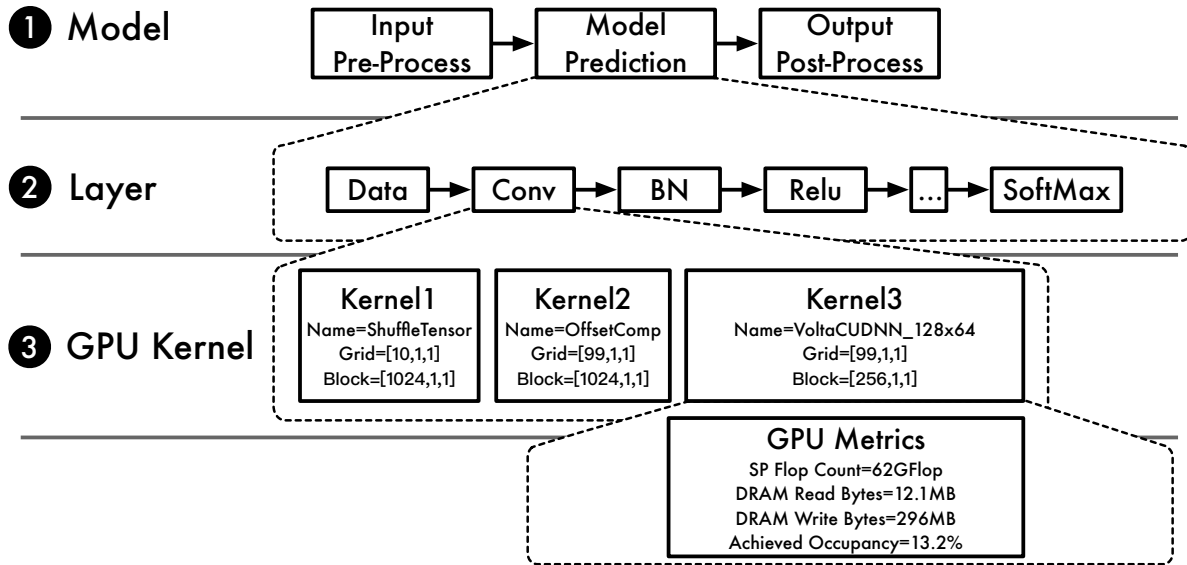


Figure 4.1: The model-, layer-, and GPU kernel-level profile of MLPerf_ResNet50_v1.5 (Table 4.8) on Tesla_V100 (Table 4.7) with batch size 256 using NVIDIA GPU Cloud TensorFlow v19.06. The layers executed are data (Data), convolution (Conv), batch normalization (BN), relu (Relu), etc. The 3 GPU kernels from the first Conv layer are shown along with the GPU metrics of Kernel 3.

ment them to work with their profilers. For example, NVIDIA GPU Cloud [27] (NGC) hosts frameworks which are instrumented with NVTX [28] markers. The NVTX markers are added around each layer in the framework and are captured along with GPU events by Nvidia’s nvprof and Nsight profilers. However, this approach only annotates GPU kernel-level information with layer names and lacks the layer-level profiling reported by the framework. Moreover, using these instrumented frameworks creates vendor lock-in — making the profiling and analysis dependent on the vendor’s frameworks and profilers. This is not an option for ML models developed or deployed using customized or non-vendor supported frameworks.

To address the above issue, we propose XSP [29] — an across-stack profiling design along with a leveled experimentation methodology. XSP innovatively leverages distributed tracing to aggregate and correlate the profiles from different sources into a single timeline trace. Through the leveled experimentation methodology, XSP copes with the profiling overhead and accurately captures the profiles at each HW/SW stack level. Users can use XSP to have a smooth hierarchical step-through of model performance at different levels within the HW/SW stack and identify bottlenecks. Unlike existing approaches, XSP requires no framework modifications. We implement the profiling design for GPUs and couple it with an across-stack analysis pipeline. The analysis pipeline consumes the across-stack profiling trace and performs 15 types of automated analyses (Table 4.1). These analyses allow us to characterize ML models and their interplay with frameworks, libraries, and hardware. The

consistent profiling and automated analysis workflows in XSP enable systematic comparisons of models, frameworks, and hardware.

This chapter makes the following contributions:

- We propose XSP, an across-stack profiling design that innovatively leverages distributed tracing to aggregate profile data from different profiling sources and construct a holistic view of ML model execution.
- We introduce a leveled experimentation methodology that allows XSP to accurately capture the profile at each HW/SW stack level despite the profiling overhead.
- We implement the design for GPU ML model inference and couple it with an analysis pipeline that performs 15 types of automated analyses to systematically characterize ML model execution.
- We conduct comprehensive experiments to show the utility of XSP. We use 65 state-of-the-art ML models from MLPerf Inference, AI-Matrix, and TensorFlow and MXNet model zoos. We evaluate the models on 5 representative systems that span the past 4 GPU generations (Turing, Volta, Pascal, and Maxwell) and present performance insights that would otherwise be difficult to discern absent XSP.

4.1 ML PROFILING ON GPUS AND RELATED WORK

Researchers leverage different tools and methods to profile ML model execution at each specific level of the HW/SW stack on GPUs. Figure 4.1 illustrates the model-, layer-, and GPU kernel-level profiling levels on GPUs.

❶ **Model-level** profiling measures the steps within the model inference pipeline. There exist active efforts by both research and industry to develop benchmark suites [3, 4] to measure and characterize models under different workload scenarios. For model-level profiling, researchers manually insert timing code around inference steps such as input pre-processing, model prediction, and output post-processing. Researchers then use the results as reference points to compare models or systems.

❷ **Layer-level** profiling measures the layers executed by the ML framework using the framework’s profilers [25, 26]. These framework profilers are either built-in to the framework or are community-contributed framework plugins. The layer index, name, latency, and memory allocations are captured by the framework profiler as it is executing the layers. Researchers explicitly enable the framework’s profiler in their code to get the layer-level profile in a framework-specific format.

❸ **GPU kernel-level** profiling measures the low-level GPU information. Using NVIDIA’s nvprof and Nsight profilers, researchers capture the executed GPU kernels information such

as their name, latency and metrics. NVIDIA’s nvprof and Nsight profilers are built on top of the NVIDIA CUPTI library [30], which provides an API to capture CUDA API, GPU kernel, and GPU metric information.

The disconnect between the above profiling levels prohibits researchers from being able to have a holistic view of model execution — thus, limiting the types of analysis which can be performed. Take the `MLPerf_ResNet50_v1.5` model in Figure 4.1 for example. One can use the aforementioned profiling tools to get the most time-consuming layer (the 208th layer which is named `conv2d_48/Conv2D`) and the most time-consuming GPU kernel (`volta_scudnn_128x64_relu_interior_nn_v1`). However, because of the lack of correlation between the GPU kernels and the layers, no other useful analysis can be performed. E.g, one cannot figure out the GPU kernels invoked by the most time-consuming layer, or correlate the most time-consuming GPU kernel to a specific layer within the model. Knowing the correlation between layers and GPU kernels enables more meaningful analyses and informs more optimization opportunities.

Currently, other than modifying framework source code, no tool or method exists to correlate the GPU kernel-level profile to the layer-level profile. For example, to be able to correlate GPU kernels to a certain layer, researchers manually instrument the framework’s source code with NVTX markers to annotate layers [31]. The NVTX markers are captured by the nvprof or Nsight profilers and kernels within the markers’ ranges belong to the annotated layers. Since the correlation between GPU kernels and layers is highly desired, NVIDIA provides modified versions of frameworks as Docker containers (NGC) where the frameworks are already instrumented with NVTX markers. While the profile captured in this approach correlates GPU kernels with layers, it lacks critical layer-level profiling (such as memory allocations performed by a framework for a layer). Furthermore, current implementations [31] introduce barriers which inhibit frameworks from performing certain optimizations (such as layer-fusion) since the NVTX layer marking is performed by surrounding each layer with a “start NVTX marker” layer and an “end NVTX marker” layer. Finally, using vendor frameworks is not an option for profiling ML models developed with customized frameworks — a common practice when using user-defined layers.

To overcome the unknown correlation between layers and GPU kernels without vendor lock-in, there have been efforts [5, 22] to develop fine-grained micro-benchmarks of representative layers. These micro-benchmarks target convolution or RNN layers and are purposely built for algorithm developers, compiler writers, and system researchers. Using layer parameters of popular models, these micro-benchmark measure each layer in isolation. Thus, they do not reflect how layers are executed by frameworks. At best, micro-benchmarks give a lower-bound estimate of how layers would perform in an ideal scenario. This lower-bound

can be used to pinpoint potential optimizations in the HW/SW stack [23]. Recent benchmark suites take a multi-tier approach [4, 7] and provide a collection of benchmarks that cover both end-to-end model and layer benchmarking.

We believe a profiling design which captures ML model executions at different HW/SW stack levels and correlates profile data from the different sources — coupled with automated analyses of the results — would boost the productivity of researchers and help understand the model/system performance and identify the bottlenecks. The authors are unaware of any previous work on the aforementioned across-stack profiling. Hence, we design XSP.

4.2 XSP DESIGN AND IMPLEMENTATION

4.2.1 Across-Stack Profiling Through Distributed Tracing

To incorporate profile data from different sources and to create a holistic hierarchical view of ML model execution, XSP leverages distributed tracing [32–34]. This section presents XSP’s across-stack profiling design.

Distributed tracing is a technique originally conceived for distributed applications, e.g., the ones built using a micro-service architecture. In distributed tracing terminology, a timed operation representing a piece of work is referred to as a *span*. Each span contains a unique identifier (used as its reference), start/end timestamps, and user-defined annotations such as name, key-value tags, and logs. A span may also contain a *parent reference* to establish a parent-child relationship. Each service in a distributed application has a *tracer* — some code to create and publish spans. Spans are published to a *tracing server* which is run on a local or remote system. The tracing server aggregates the spans published by the different tracers into one application timeline trace.

We observe similarities between distributed tracing and across-stack profiling. Based on this observation, we propose XSP, an across-stack profiling design. Profiling across stack levels can be represented using the distributed tracing terminology by: ① each profiler within a stack is turned into a tracer, ② the profiled events each form a span, ③ each span is tagged with its stack level, and ④ the parent-child relationship is encoded using a parent reference. The conversion from the profiled events to spans can be performed online while the profiler is running, or can be performed off-line by processing the output of the profiler. The published spans across the stack levels are aggregated by a tracing server into a single timeline trace. Multiple tracers (or profilers) can exist within a stack level, e.g. both CPU and GPU tracers can co-exist at system library or hardware level. As a feature supported by distributed tracing, tracers can be enabled or disabled at runtime.

During span creation, we can, in some cases, associate it with a parent (e.g. map the layer-level spans to the model prediction span). In other cases, because of the use of disjoint profilers, manually associating the child span with its immediate parent is not possible (e.g. map the GPU kernel-level spans to the CPU layer-level spans). To reconstruct the missing parent-child relationship of the profiled events captured by different profilers, XSP’s profile analysis builds an interval tree [35] and populates it with intervals corresponding to the spans’ start/end timestamps. Using the interval tree, XSP reconstructs the parent-child relationship by checking for interval set inclusion (if the interval span s_1 contains the interval span s_2 and the level of s_1 is one level higher than the level of s_2 , then s_1 is a parent of s_2). It is possible that there are parallel events where it may be ambiguous to determine a span’s parent. In those cases, XSP requires another profiling run where the parallel events are serialized to get the missing correlation information. This can be performed by specifying environment variables without modifications to the application — e.g. setting either `CUDA_LAUNCH_BLOCKING=1` for GPUs using CUDA or `OMP_NUM_THREADS=1` for CPUs using OpenMP.

To profile asynchronous functions, XSP captures two spans for each asynchronous function denoting their asynchronous launch (called a *launch span*) and future execution (called an *execution span*). XSP correlates the two spans using a correlation identifier which is inserted as a span tag during span creation. XSP uses the launch span’s parent as the parent of the asynchronous function and uses the execution span to get the performance information or find child spans. E.g., to profile asynchronous GPU kernels, XSP captures both the kernel launch and execution spans (as detailed in Section 4.2.2).

4.2.2 Across-stack Profiling on GPUs

While the across-stack profiling design presented above is general, this paper focuses on the profiling of ML models on GPUs across the model, layer, and GPU kernel level:

❶ **Model-level profiling** — To profile at the model granularity, XSP provides tracing APIs — `startSpan` and `finishSpan` — which can be placed within the inference code to measure code regions of interest. For example, to measure the time spent running the model prediction using the framework C APIs, one places the tracing APIs around the calls to `TF_SessionRun` for TensorFlow or `MXPredForward` for MXNet. This only requires adding two extra lines in the user’s inference code.

❷ **Layer-level profiling** — To profile at the layer granularity, XSP uses the ML framework’s existing profiling capability. During runtime, XSP enables the framework profiler, converts the profile results into spans, and publishes them to the tracing server. In Ten-

TensorFlow, enabling layer profiling requires calling the framework’s prediction function with the profiling option enabled. This option is controlled by the `RunOptions.TraceLevel` setting which is passed to the `TF.SessionRun` function in TensorFlow. In MXNet, the `MXSetProfilerState` function enables and disables layer profiling. Similar mechanisms exist for other frameworks such as Caffe, Caffe2, PyTorch, and TensorRT. The layer spans are set to be the children of the model prediction span, and hence each layer are directly correlated to the model prediction step. Since XSP leverages the existing framework’s profiling capabilities, profiling at the layer level require no modification to the framework’s source code.

🕒 **GPU kernel-level profiling** — To obtain the GPU profile, XSP uses NVIDIA’s CUPTI library [30]. The CUPTI library captures the CUDA API calls, GPU activities (GPU tasks such as kernel executions and memory copies), and GPU kernel metrics (low-level hardware counters such as GPU achieved occupancy, flop count, and memory read/write for GPU kernels). Similar to Nsight or nvprof (which are built on top of CUPTI), one can specify with XSP which CUDA APIs, GPU activities, or metrics to capture. At runtime, XSP converts the captured CUPTI information into spans and publishes them to the tracer server (asynchronously to avoid added overhead). If profiling GPU metrics is enabled, the metrics are added as metadata to the corresponding kernel’s span.

GPU kernels are often launched asynchronously by the ML frameworks or libraries. Therefore, for each kernel two spans are created within the XSP timeline. The CUPTI Callback API allows one to register a callback function when the code being profiled calls a CUDA function. XSP uses the CUPTI callback API to capture the CUDA API `cudaLaunchKernel` as the launch span. The CUPTI Activity API allows one to asynchronously collect a trace of the GPU activity. XSP uses the CUPTI activity API to capture the effective kernel duration as the execution span. XSP uses the kernel launch span to associate it with the parent layer span and use the execution span to get the kernel performance information. The two spans are correlated by the `correlation_id` provided by CUPTI. Since this correlation can potentially be expensive, we perform correlation during profile analysis which aggregates the information from two GPU kernel spans.

4.2.3 Dealing with Profiling Overhead through Leveled Experimentation

Profiling always comes with overhead. We observe that creating spans online adds negligible overhead per span (and no overhead exists if the profile is converted offline). Thus, XSP incurs only the profiling overhead introduced by the integrated profilers. For example, layer-level profiling adds overhead to the model prediction depending on how many layers

are executed. And as with the existing NVIDIA profilers, the GPU-level profiling incurs overhead, which can be substantial depending on if GPU metric profiling is enabled and the types of GPU metrics to capture. GPU memory metrics are especially expensive to profile and can slow down execution by over 100×. This is due to the limited number of GPU hardware performance counters, which require GPU kernels to be replayed multiple times to capture the user-specified metrics.

M: Model-level Profiling L: Layer-level Profiling G: GPU Kernel-level Profiling

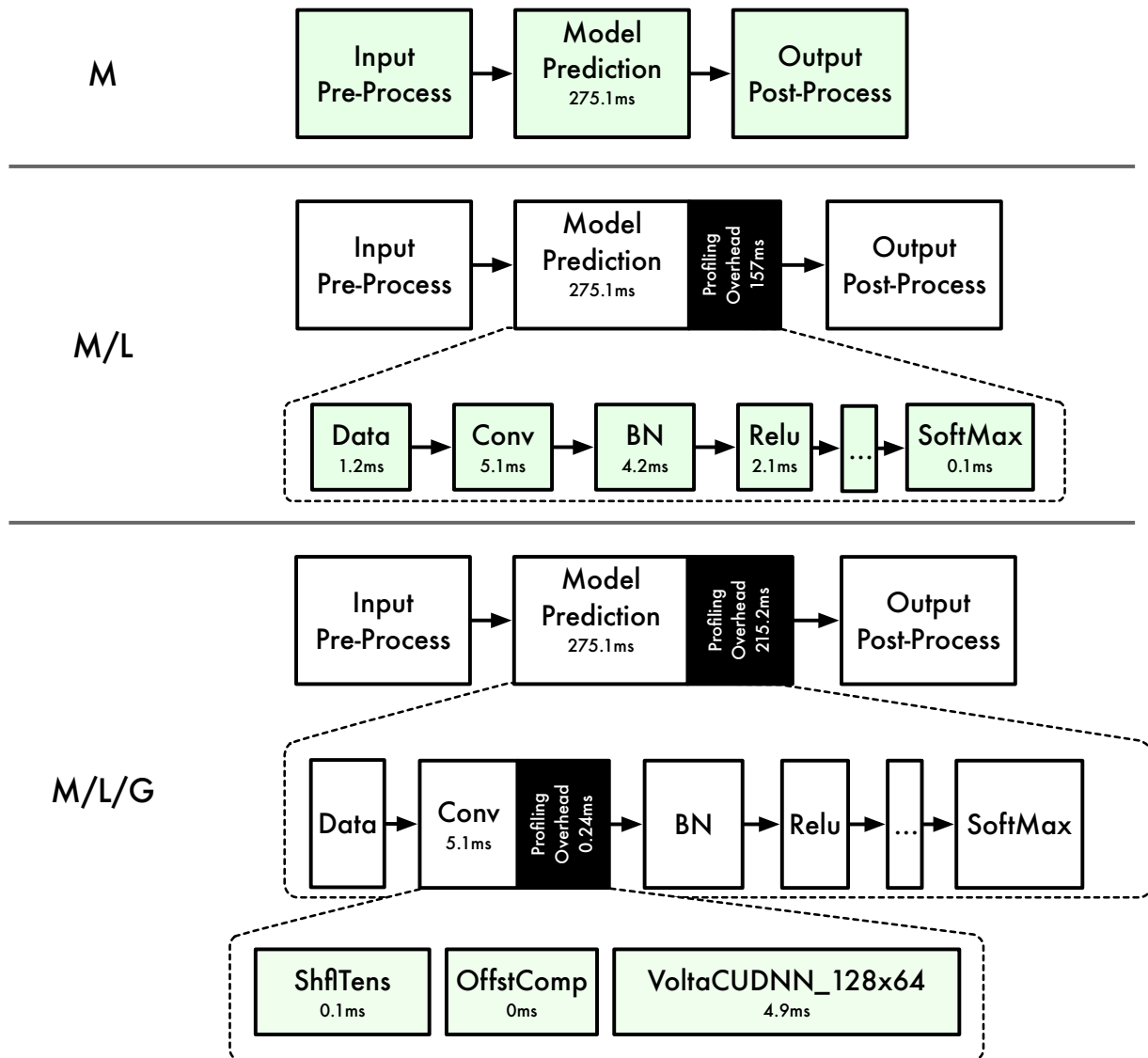


Figure 4.2: XSP profiles for MLPerf_ResNet50_v1.5 with batch size 256 on Tesla_V100 (Table VI) with the model-level (M), model-/layer-level (M/L), and model-/layer-/GPU kernel-level (M/L/G) profiling. At each level, the green components correctly measure the latency whereas the rest incur profiling overhead.

Profilers at a specific stack level accurately capture the events within that level. And, since tracers in XSP can be enabled or disabled depending on the characterization target, the profiling overhead can be controlled by picking the profiling level. For an event at level n (where level 1 is the model level), the profiling overhead introduced at level $n + 1$ can be quantified by subtracting the latency of the event when profilers up to level n are enabled from the latency when profilers up to level $n + 1$ are enabled. We refer to the profiling practice which uses traces from multiple runs with different profiling levels as *leveled experimentation*. Through leveled experimentation, XSP gets the accurate timing of the profiled events at all stack levels.

To demonstrate the profiling overhead and the leveled experimentation, we use the `MLPerf_ResNet50_v1.5` model running on the `Tesla_V100` system (Table 4.7) as an example. Figure 4.2 shows the model’s XSP profiles at different profiling levels. We can enable the model-level profiling (M) to get the baseline model prediction latency of $275.1ms$. To further measure the latency of each layer, we enable both the model- and layer-level profiling (M/L). While the layer-level profiling adds overhead to the model prediction latency, it accurately captures the latency of each layer. We can quantify this overhead by subtracting the model prediction latency in the model-level profile from the model prediction latency in the model-/layer-level profile. We find that the layer-level profiling introduces a $157ms$ overhead. We can further perform the GPU kernel-level profiling along with the model-/layer-level profiling to get a hierarchical view of the model execution (M/L/G). Enabling the GPU kernel-level profiling adds extra overhead to the model prediction latency — making the model prediction step (with the added overhead) take $490.3ms$. If we look at the first convolution layer, the GPU profiling of the 3 child kernels incurs a $0.24ms$ overhead. We verified the layer and GPU kernel latencies measured by XSP against what framework and NVIDIA’s profilers report.

4.2.4 Extensibility

Care was taken to ensure that XSP’s design is extensible. Other profiling tools or methods can be integrated into XSP by implementing XSP’s tracer interface. Thus, XSP can be extended with more tracers at each stack level or extended to capture more stack levels. For example, one can integrate CPU profilers into XSP to capture both CPU and GPU information within the same timeline. One can also add an ML library profiling level between the layer- and GPU kernel-level to measure the cuDNN API calls. Adding an application profiling level above the model level to measure whole applications (possibly distributed and using more than one ML model) is naturally supported by XSP as it uses distributed

Table 4.1: The 15 analyses performed by MLModelScope. The analyses require profiling information from one or more levels (**M**: model-level profile, **L**: layer-level profile, and **G**: GPU kernel-level profile.)

Analysis	Profiling Provider	End-to-End Benchmarking	Framework Profilers	NVIDIA Profilers	MLModelScope
Analysis 1	M	✓	✗	✗	✓
Analysis 2	L	✗	✓	✗	✓
Analysis 3	L	✗	✓	✗	✓
Analysis 4	L	✗	✓	✗	✓
Analysis 5	L	✗	✓	✗	✓
Analysis 6	L	✗	✓	✗	✓
Analysis 7	L	✗	✓	✗	✓
Analysis 8	G	✗	✗	✓	✓
Analysis 9	G	✗	✗	✓	✓
Analysis 10	G	✗	✗	✓	✓
Analysis 11	L/G	✗	✗	✗	✓
Analysis 12	L/G	✗	✗	✗	✓
Analysis 13	L/G	✗	✗	✗	✓
Analysis 14	L/G	✗	✗	✗	✓
Analysis 15	M/L/G	✗	✗	✓	✓

tracing. As new profilers are introduced into XSP, one can add more types of analyses to the automated analysis pipeline.

4.2.5 Integration within MLModelScope Runtime

We integrated XSP within MLModelScope [36], an open-source framework and hardware agnostic, extensible, and customizable framework for evaluating ML models at scale. For distributed tracing, we use Jaeger [37] — a production grade [38] distributed tracing library. XSP uses the frameworks’ C-level API directly to avoid the added overhead introduced by scripting languages. Consequently, the model inference latency captured at the model level is as close to the bare metal performance as possible. We wrap the C API calls with tracing points to capture the model latency, pass the required options for the framework’s layer-level profiling, and extend XSP to use the CUPTI library.

We also modified the user interface of MLModelScope. Users control the profiling granularity (model, framework, GPU API and activity, and GPU metrics) of the model evaluation through MLModelScope’s command line, library, or web interface. We also added a profile ingestion pipeline within XSP which is described in detail in Section 4.2.6.

4.2.6 Across-Stack Analysis

We couple XSP with an automated analysis pipeline which consumes the profiling traces published to the tracing server. We define 15 analyses that capture across-stack characteristics of ML model execution on GPUs as listed in Table 4.1. The 15 analyses are grouped into

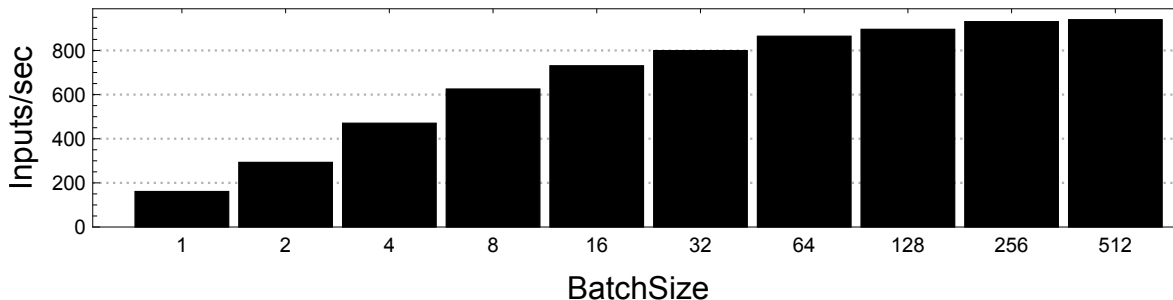


Figure 4.3: The throughput of `MLPerf_ResNet50_v1.5` across batch sizes on `Tesla_V100`.

3 categories based on the profiling information required. Since meaningful characterization requires multiple runs, the pipeline takes traces from a user-defined number of evaluations, correlates the information, and computes the trimmed mean value (or other user-defined statistical summaries) for the same performance value (e.g. latency) across runs. This automated analysis pipeline allows users to systematically and efficiently characterize and compare ML models.

To illustrate the analyses, we use the TensorFlow `MLPerf_ResNet50_v1.5` model (ID = 7 in Table 4.8) from the MLPerf Inference v0.5 release. The model is run within the NGC TensorFlow container v19.06 on an AWS P3 [39] instance (`Tesla_V100` in Table 4.7). The P3 instance is equipped with a Tesla V100-SXM2 GPU and achieves a peak throughput of 15.7 TFlops and 900 GB/s global memory bandwidth. Batch size 256 is used in Sections 4.2.6 and 4.2.6, since the model achieves maximum throughput at that batch size. Using XSP, one can perform analyses that are either difficult or impossible using existing tools or methods.

Using Model-level Profile

Both model throughput and latency are important to researchers who want to understand a model’s end-to-end performance. Using only the model-level profiling, XSP automates the computation of a model’s throughput and latency across batch sizes and generate a **Analysis 1** model information table. XSP then computes the model’s optimal batch size given a user-defined metric (e.g. a latency target). By default XSP computes the optimal batch size by evaluating the model across batch sizes and selecting the batch size where doubling it does not increase the model’s throughput by more than 5%. Figure 4.3 shows the throughput of `MLPerf_ResNet50_v1.5` across batch sizes. XSP computes the optimal batch size as 256, where the model achieves a maximum throughput of 930.7 images/second. The corresponding batch latency is $275.05ms$. Absent XSP, researchers insert timing functions

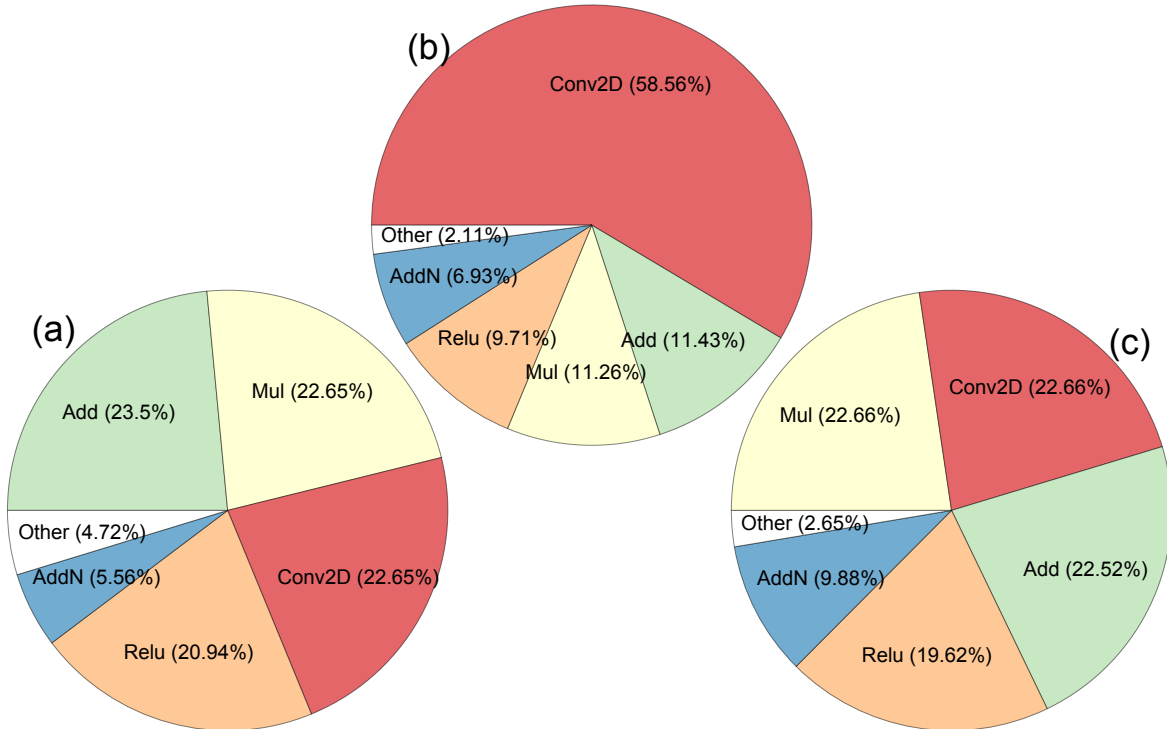


Figure 4.4: Layer statistics for MLPPerf_ResNet50_v1.5 on Tesla_V100: (a) **Analysis 5** layer type distribution, (b) **Analysis 6** layer latency aggregated by type, (c) **Analysis 7** layer memory allocation aggregated by type.

around the model prediction code, perform multiple evaluations, and write scripts to compute the model’s throughput, latency, and optimal batch size.

Using Model- and Layer-level Profiles

Using both the model- and layer-level profiles enables the characterization of layers executed by the ML framework. The measured layers may be different from the ones statically defined in the model graph, since a framework may perform model optimization at runtime.

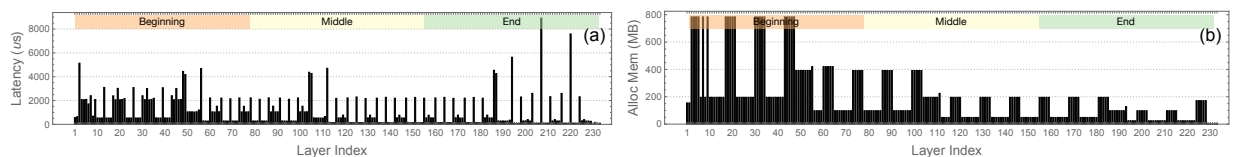


Figure 4.5: The (a) **Analysis 3** latency and (b) **Analysis 4** memory allocation for each layer in MLPPerf_ResNet50_v1.5 with batch size 256 on Tesla_V100. To understand the performance trend, we divide the model execution into 3 intervals based on the layer index: beginning, middle, and end.

Using the data captured, XSP generates a **Analysis 2** layer information table reporting index, name, shape, latency, and allocated memory of all the layers. For example, Table 4.2 shows the top 5 most time-consuming layers for `MLPerf_ResNet50_v1.5`.

XSP further uses the profile data to visualize both the **Analysis 3** latency per layer and **Analysis 4** allocated memory per layer in layer execution order. Figure 4.5 shows the two analyses for `MLPerf_ResNet50_v1.5` at the optimal batch size. We observe that a layer latency and memory allocation trend exists — the model latency can be mostly attributed to the early executed layers. Similarly, the memory allocation is high for the early stage of the model execution, and less so during the middle and end stages. This is because the early stage of `MLPerf_ResNet50_v1.5` requires more compute and memory (e.g. dimensions of tensors are larger, operators are more expensive, etc.).

We can group the layer information by layer type to derive useful layer execution statistics such as **Analysis 5** the number of times each layer type is executed (Figure 4.4a), the **Analysis 6** layer latency aggregated by type (Figure 4.4b), and the **Analysis 7** layer memory allocation aggregated by type (Figure 4.4c). We observe that `MLPerf_ResNet50_v1.5` mostly comprises of `Add`, `Conv2D`, `Mul`, and `Relu` layers. This is because of the ResNet modules which have the pattern of `Conv` → `BN` → `Relu`. The ResNet modules get executed by TensorFlow as a `Conv2D` → `Mul` → `Add` → `Relu` layer sequence. This same group of layers dominates both latency and memory allocation, with `Conv2D` being the most time-consuming layer type.

Absent XSP, researchers use the framework profiler to gather layer-level information. Through manually parsing and aggregating the profiling output across runs, researchers can perform **Analysis 2-7**. However, since the output format of a framework profiler is framework-dependent, the analysis scripts developed in this case are also framework-specific.

Using Model-, Layer-, and GPU Kernel-level Profiles

To distill fine-grained performance information, XSP uses model-, layer- and GPU kernel-level profiles to generate a **Analysis 8** GPU kernel information table summarizing all the kernels in the model prediction. An example is shown in Table 4.3 where the top 5 most time consuming GPU kernel calls for `MLPerf_ResNet50_v1.5` are listed. The 5 kernels perform either matrix multiplication or convolution. All the GPU metrics supported by the NVIDIA profiling tools [40] can be captured through XSP, here we focus on `flop_count_sp`, `dram_read_bytes`, `dram_write_bytes`, and `achieved_occupancy`:

- `flop_count_sp` — the total number of single-precision floating-point operations exe-

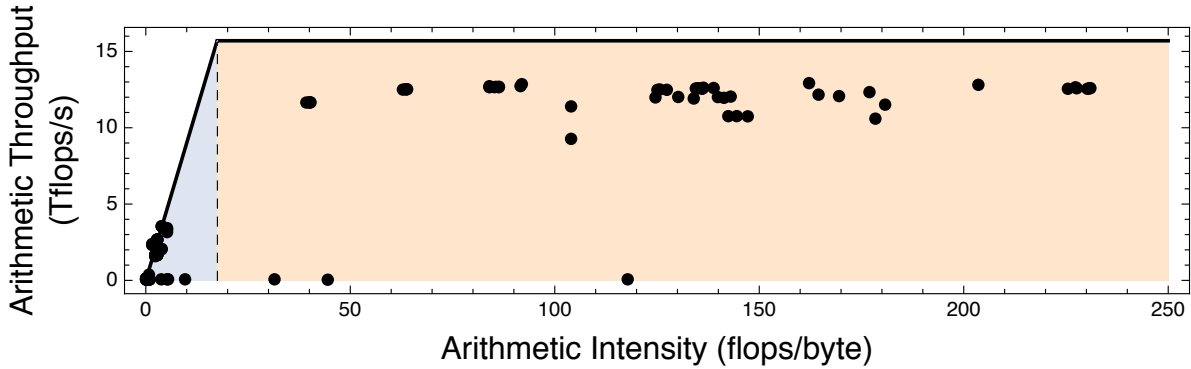


Figure 4.6: The **Analysis 9** roofline analysis for the GPU kernels in MLPerf_ResNet50_v1.5 with batch size 256 on Tesla_V100. Kernels within the blue region are memory-bound, whereas the ones within the orange region are compute-bound.

cuted by a kernel.

- `dram_read.bytes` — the total number of bytes read from the GPU’s DRAM to its L2 cache in a kernel.
- `dram_write.bytes` — the total number of bytes written from the GPU’s L2 cache to its DRAM in a kernel.
- `achieved_occupancy` — the ratio of the average active warps per active cycle to the maximum number of warps per streaming multiprocessor. The `achieved_occupancy` is an indicator to the level of parallelism for a kernel.

Using both the kernel flop and memory access metrics, XSP calculates the kernel arithmetic intensity and arithmetic throughput. These parameters are used to perform GPU kernel roofline [41] analysis. A kernel’s arithmetic intensity is the ratio between the number of flops and the number of memory accesses: $\text{arithmetic_intensity} = \frac{\text{flop_count_sp}}{\text{dram_read.bytes} + \text{dram_write.bytes}}$. A kernel’s arithmetic throughput is the ratio between the number of flops and the latency: $\text{arithmetic_throughput} = \frac{\text{flop_count_sp}}{\text{kernel_latency}}$. Using the GPU’s theoretical FLOPS and memory bandwidth, we compute the ideal arithmetic intensity using the equation: $\text{ideal_arithmetic_intensity} = \frac{\text{peak_FLOPS}}{\text{memory_bandwidth}}$. The Tesla_V100 GPU, for example, has a peak throughput of 15.7 TFLOPS and a global memory bandwidth of 900 GB/s, hence an ideal arithmetic intensity of $\frac{15.7 \text{ TFLOPS}}{900 \text{ GB/s}} = 17.44$ flops/byte. A kernel is *memory-bound* if its arithmetic intensity is less than the GPU’s ideal arithmetic intensity (blue region) and is *compute-bound* otherwise (orange region). **Analysis 9** visualizes the roofline analysis of all the GPU kernels (shown in Figure 4.6). As expected, the most time-consuming kernels are convolution kernels which are all compute-bound.

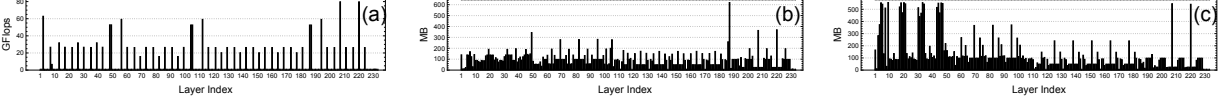


Figure 4.7: The **Analysis 12** total GPU kernel (a) flops, (b) DRAM reads, and (c) DRAM writes per layer for MLPerf_ResNet50_v1.5 with batch size 256 on Tesla_V100.

XSP creates a table of **Analysis 10** GPU kernel information aggregated by name, as shown in Table 4.4. The aggregated kernel latency, flops, and DRAM reads and writes are calculated as the sum of all the kernel instances with the same name. The aggregated kernel achieved occupancy is calculated as the weighted sum (by kernel latency) of achieved occupancy of all the kernel instances with the same name. The aggregated kernel arithmetic intensity and throughput are calculated using the aggregated flops and memory accesses. For MLPerf_ResNet50_v1.5, we observe that the most time consuming GPU kernel is `volta_scudnn_128x64_relu_interior_nn_v1` from the cuDNN [42] library, which is compute-bound and takes 30.87% of the overall model prediction latency. The 2nd and 3rd most time consuming kernels are `scalar_product_op` and `scalar_sum_op` and are defined by the Eigen [43] library, are memory-bound, and take 10.33% and 9.59% of the model inference latency, respectively.

Since each GPU kernel can be correlated to the layer that invokes it, XSP aggregates the information of GPU kernels within each layer and builds a table of **Analysis 11** GPU kernel information aggregated by layer. A layer’s kernel latency, flops, DRAM reads and writes are calculated by adding the corresponding values of all the kernels invoked by that layer. The layer’s achieved occupancy is calculated as the weighted sum (by kernel latency) of the achieved occupancy of all the kernels within the layer. As an example, Table 4.5 shows the aggregated GPU kernel information for the top 5 most time-consuming layers in MLPerf_ResNet50_v1.5.

Using this data, XSP visualizes the **Analysis 12** total flops, DRAM reads and writes per layer (shown in Figure 4.7 (a), (b) and (c) respectively). Subtracting a layer’s total GPU kernel latency from the its overall latency computes the **Analysis 13** time not spent performing GPU computation. We call this difference the layer’s *non-GPU latency*. Figure 4.8 shows the layer’s GPU and non-GPU latency normalized to the overall layer latency for MLPerf_ResNet50_v1.5. The layer arithmetic intensity and throughput are calculated using the layer’s total flops and memory accesses. A **Analysis 14** roofline analysis of all the layers is performed in Figure 4.9. We observe that the Conv2D layers are the most compute and memory intensive. The Conv2D, MatMul, BiasAdd, and Softmax layers are compute-bound, whereas the other layers (Add, Mul, and Relu) are memory-bound.

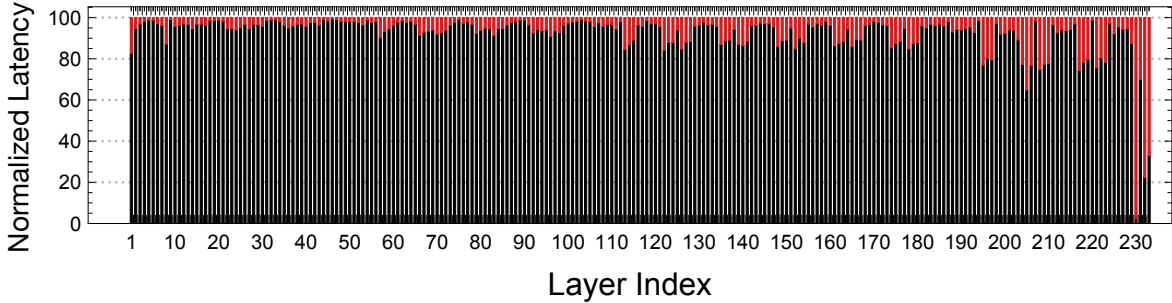


Figure 4.8: The **Analysis 13** normalized GPU and Non-GPU latency per layer for `MLPerf_ResNet50_v1.5` with batch size 256 on `Tesla_V100`.

XSP aggregates all the GPU kernel information within a model and computes a table of the **Analysis 15** total GPU kernel latency, flop, and memory access information for the model (shown in Table 4.6). Similar to the layer aggregation, the model kernel latency, flops, DRAM reads and writes are calculated as the sum of all kernels invoked by the model. XSP computes the model’s achieved occupancy as the weighted sum (by kernel latency) of the achieved occupancy of all the kernels invoked. The model’s arithmetic intensity and throughput are calculated using the model’s total flops and memory accesses. This information is used to classify the entire model as either compute- or memory-bound.

Figure 4.10 visualizes the roofline analysis for `MLPerf_ResNet50_v1.5` across batch sizes on `Tesla_V100`. We see that the model is compute-bound except for batch sizes 16 and 32 where it is memory-bound. Looking into the data in **Analysis 2,8,10** we find that the kernels invoked for the convolution layers sometimes vary across batch sizes. This is because the cuDNN library relies on heuristics to choose the algorithm used for a convolution layer. The heuristics depend on the layer input parameters, available memory, etc. For batch sizes less than 16, the cuDNN convolution API uses the `IMPLICIT_GEMM` algorithm and invokes the GPU kernel `cuda::detail::implicit_convolve_sgemm`. This kernel has high arithmetic intensity and dominates the model’s latency. For batch sizes greater than 16, the cuDNN convolution API chooses a different algorithm — `IMPLICIT_PRECOMP_GEMM` algorithm, which invokes the GPU kernel `volta_scudnn.128x64.relu.interior.nn.v1`. Although this kernel is compute-bound, for batch sizes less than 64 it has a relatively low arithmetic intensity. Thus, for both batch sizes 16 and 32, this kernel’s arithmetic intensity is not high enough to compensate for the effects of the other memory-bound kernels. The result is that the overall model is memory-bound for batch sizes 16 and 32. We also observe that the overall GPU achieved occupancy for the model increases as the batch size approaches the optimal batch size.

Analysis 8 and **Analysis 10** are currently the most common types of analyses performed

Table 4.2: The top 5 most time consuming layers in [Analysis 2](#) for MLPerf_ResNet50_v1.5 with batch size 256 on Tesla_V100. In total, there are 234 layers of which 143 take less than 1 ms.

Layer Index	Layer Name	Layer Type	Layer Shape	Latency (ms)	Alloc Mem (MB)
208	conv2d_48/Conv2D	Conv2D	$\langle 256, 512, 7, 7 \rangle$	7.59	25.7
221	conv2d_51/Conv2D	Conv2D	$\langle 256, 512, 7, 7 \rangle$	7.57	25.7
195	conv2d_45/Conv2D	Conv2D	$\langle 256, 512, 7, 7 \rangle$	5.67	25.7
3	conv2d/Conv2D	Conv2D	$\langle 256, 64, 112, 112 \rangle$	5.08	822.1
113	conv2d_26/Conv2D	Conv2D	$\langle 256, 256, 14, 14 \rangle$	4.67	51.4

Table 4.3: The top 5 most time-consuming kernels in [Analysis 8](#) for MLPerf_ResNet50_v1.5 on Tesla_V100. In total, 375 kernels are invoked of which 284 take less than 1ms.

Kernel Name	Layer Index	Layer Kernel Latency (ms)	Kernel Gflops	Kernel DRAM Reads (MB)	Kernel DRAM Writes (MB)	Kernel Achieved Occupancy (%)	Kernel Arithmetic Intensity (flops/byte)	Kernel Arithmetic Throughput (Tflops/s)	Memory Bound?
volta_cgemm_32x32.tn	221	6.04	77.42	40.33	43.86	12.18	876.97	12.82	\times
volta_cgemm_32x32.tn	208	6.03	77.42	43.93	43.81	12.19	841.59	12.83	\times
volta_scudnn_128x128_relu_interior_nn_v1	195	5.48	59.20	27.71	8.40	15.49	1,563.30	10.80	\times
volta_scudnn_128x64_relu_interior_nn_v1	3	4.91	62.89	11.55	283.05	13.20	203.58	12.81	\times
volta_scudnn_128x128_relu_interior_nn_v1	57	4.56	59.24	34.83	37.64	15.15	779.55	12.99	\times

by researchers using NVIDIA’s profilers. Less common, but still possible, analyses without XSP are roofline analyses [Analysis 9](#) and [Analysis 15](#) as they require non-trivial scripts. The scripts parse and aggregate the GPU profilers’ outputs across multiple model evaluations to compute the roofline model. Analyses [Analysis 11-14](#) cannot be performed using existing tools as they require both the layer- and GPU kernel-level profiles and their results to be correlated.

4.3 EVALUATION

We profile and characterize 55 state-of-the-art TensorFlow ML models (Table 4.8) selected from the MLPerf Inference [3], AI-Matrix [4], and TensorFlow model zoo [44–46]. The models solve computer vision tasks including image classification, object detection, instance segmentation, semantic segmentation, and super resolution. To compare TensorFlow against MXNet, we select an additional 10 MXNet models from the MXNet Gluon model

Table 4.4: The top 5 most time-consuming kernels in [Analysis 10](#) for MLPerf_ResNet50_v1.5 on Tesla_V100. 30 unique kernels are invoked in total.

Kernel Name	Kernel Count	Kernel Latency (ms)	Kernel Latency Percentage	Kernel Gflops	Kernel DRAM Reads (MB)	Kernel DRAM Writes (MB)	Kernel Achieved Occupancy (%)	Kernel Arithmetic Intensity (flops/byte)	Kernel Arithmetic Throughput (Tflops/s)	Memory Bound?
volta_scudnn_128x64_relu_interior_nn_v1	34	84.95	30.87	1,053.63	4,429.64	5,494.22	22.58	101.25	12.40	\times
Eigen::TensorCwiseBinaryOp<scalar_product_op>	52	28.43	10.33	2.85	4,181.23	6,371.12	49.72	0.26	0.10	\checkmark
Eigen::TensorCwiseBinaryOp<scalar_sum_op>	51	26.38	9.59	2.64	4,063.49	6,052.22	49.69	0.25	0.10	\checkmark
Eigen::TensorCwiseBinaryOp<scalar_max_op>	48	24.71	8.98	0	3,773.84	5,699.95	98.39	0	0	\checkmark
volta_scudnn_128x128_relu_interior_nn_v1	4	23.02	8.37	276.64	671.68	335.01	15.96	262.08	12.02	\times

Table 4.5: The top 5 most time-consuming layers in **Analysis 11** for MLPerf_ResNet50_v1.5 on Tesla_V100.

Layer Index	Layer Latency (ms)	Kernel Latency (ms)	Layer Gflops	Layer DRAM Reads (MB)	Layer DRAM Writes (MB)	Layer Achieved Occupancy (%)	Layer Arithmetic Intensity (flops/byte)	Layer Arithmetic Throughput (Tflops/s)	Memory Bound?
208	7.59	7.45	79.74	362.67	548.50	19.43	83.46	10.70	✗
221	7.57	7.43	79.74	368.11	551.70	19.43	82.68	10.73	✗
195	5.67	5.55	59.20	36.51	17.99	15.80	1,036.10	10.67	✗
3	5.08	4.91	62.89	11.55	284.21	13.23	202.78	12.80	✗
113	4.67	4.57	59.22	76.65	21.36	15.31	576.17	12.94	✗

Table 4.6: The **Analysis 15** GPU kernel information aggregated within MLPerf_ResNet50_v1.5 across batch sizes on Tesla_V100.

Batch Size	Model Latency (ms)	Kernel Latency (ms)	Model Gflops	Model DRAM Reads (MB)	Model DRAM Writes (MB)	Model Achieved Occupancy (%)	Memory Bound?
1	6.21	5.01	7.94	192.49	194.16	22.65	✗
2	6.83	5.93	16.08	290.41	354.54	22.47	✗
4	8.51	7.68	30.95	659.11	720.15	26.39	✗
8	12.80	11.60	60.66	1,676.07	1,496.81	31.97	✗
16	21.90	20.14	118.04	3,969.19	3,024.09	35.58	✓
32	40.03	37.14	232.78	7,711.50	5,823.97	38.76	✓
64	74.03	67.72	429.08	10,932.22	9,268.27	43.18	✗
128	142.89	131.79	873.63	16,071.32	16,105.40	44.48	✗
256	275.05	254.25	1,742.39	23,185.11	31,095.45	43.15	✗

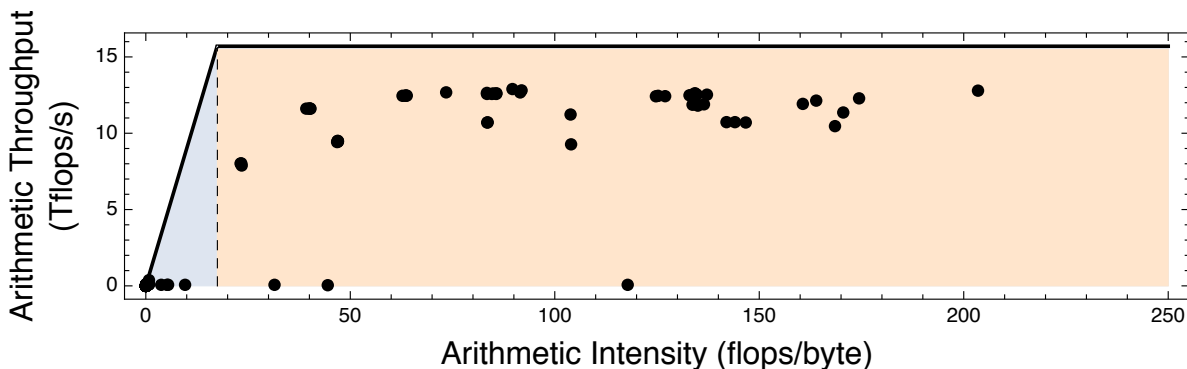


Figure 4.9: The **Analysis 14** roofline analysis for all the layers in MLPerf_ResNet50_v1.5 with batch size 256 on Tesla_V100.

Table 4.7: Five systems with Turing, Volta, Pascal, and Maxwell GPUs are selected for evaluation. We calculate the ideal arithmetic intensity of each system using the theoretic Flops and memory bandwidth reported by NVIDIA.

Name	CPU	GPU	GPU Architecture	Theoretical FLOPS (TFLOPS)	Memory Bandwidth (GB/s)	Ideal Arithmetic Intensity (flops/byte)
Quadro_RTX	Intel Xeon E5-2630 v4 @ 2.20GHz	Quadro RTX 6000	Turing	16.3	624	26.12
Tesla_V100 (AWS P3)	Intel Xeon E5-2686 v4 @ 2.30GHz	Tesla V100-SXM2-16GB	Volta	15.7	900	17.44
Tesla_P100	Intel Xeon E5-2682 v4 @ 2.50GHz	Tesla P100-PCIE-16GB	Pascal	9.3	732	12.70
Tesla_P4	Intel Xeon E5-2682 v4 @ 2.50GHz	Tesla P4	Pascal	5.5	192	28.34
Tesla_M60 (AWS G3)	Intel Xeon E5-2686 v4 @ 2.30GHz	Tesla M60	Maxwell	4.8	160	30.12

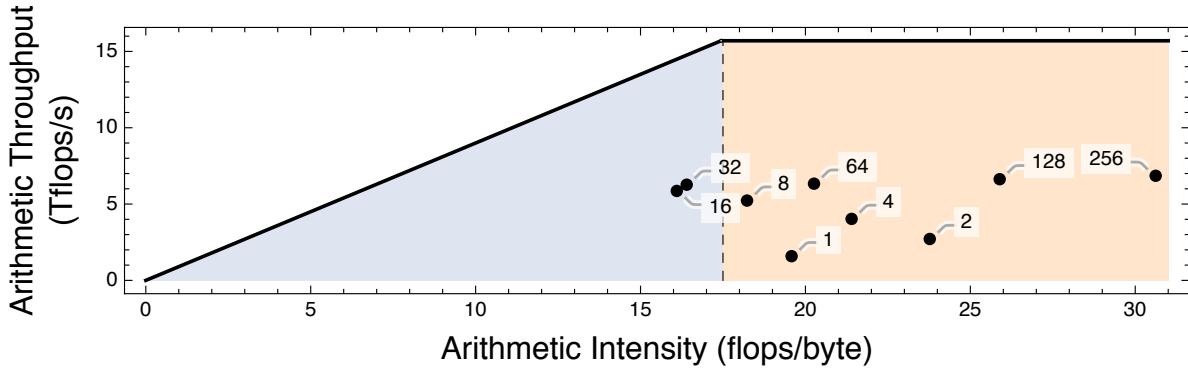


Figure 4.10: The roofline analysis for MLPerf_ResNet50_v1.5 across batch sizes on Tesla_V100 using `Analysis 15`.

zoo [47] (Table 4.10) that are comparable to the TensorFlow models. We evaluated the models using NGC TensorFlow container v19.06, and NGC MXNet container v19.06 on 5 representative GPU systems listed in Table 4.7. This section presents insights about the models, frameworks, and GPU systems using the XSP’s analyses described in Section 4.2.6.

4.3.1 Model Evaluation

Using the model- and layer-level profiling data, we look at all 55 TensorFlow models in Table 4.8. Models solving the same task are clustered together and are then sorted by their reported accuracy. The table shows each model’s accuracy, model graph size, online latency (batch size is 1), maximum throughput, optimal batch size (described in Section 4.2.6), and percentage of latency attributed to convolution layers.

Model latency percentage of convolution layers — Using the model- and layer-level profile data, we calculate the percentage of model latency attributed to convolution layers (Tensorflow’s `Conv2D` and `DepthwiseConv2dNative` layers) with each model’s optimal batch size on Tesla_V100. This is shown in the last column of Table 4.8. We observe that: ① the convolution layer latency percentage ranges between 36.3% and 80.2% for image classification models. This suggests that convolution layers still dominate (but not exclusively) the latency of image classification models — even on recent GPUs. This is not true for ② object detection models, which (except for `Faster_RCNN_NAS`) attribute only 0.6% to 14.9% of latency to convolution layers. For these models, the dominating layer type is `Where`, which reshapes a tensor with respect to a user-defined operator. For ③ instance segmentation models, convolution layers dominate the model latency; except for `Mask_RCNN_Inception_v2` whose latency is also dominated by `Where` layers. For ④ semantic segmentation models, the model latency is affected by both the convolution layers and the memory-bound layers (such as

Table 4.8: We use 55 TensorFlow models from MLPerf, AI-Matrix, and TensorFlow Slim, Detection Zoo, DeepLab for evaluation. These models are sorted by the reported accuracy and solve different tasks: Image Classification (**IC**), Object Detection (**OD**), Instance Segmentation (**IS**), Semantic Segmentation (**SS**), and Super Resolution (**SR**). We measured the peak throughput achieved on **Tesla_V100** and find the optimal batch size for each model. Online latency is defined as the model latency for batch size 1. Graph size is the size of the frozen graph for a model.

ID	Name	Task	Accuracy	Graph Size (MB)	Online Latency (ms)	Max Throughput (Inputs/Sec)	Optimal Batch Size	Convolution Percentage (%)
1	Inception_ResNet_v2	IC	80.40	214	23.24	346.6	128	68.8
2	Inception_v4	IC	80.20	163	17.29	436.7	128	75.7
3	Inception_v3	IC	78.00	91	9.85	811.0	64	72.8
4	ResNet_v2_152	IC	77.80	231	14.05	466.8	256	60.5
5	ResNet_v2_101	IC	77.00	170	10.39	671.7	256	60.9
6	ResNet_v1_152	IC	76.80	230	13.70	541.3	256	69.6
7	MLPerf_ResNet50_v1.5	IC	76.46	103	6.22	930.7	256	58.7
8	ResNet_v1_101	IC	76.40	170	10.01	774.7	256	69.9
9	AI_Matrix_ResNet152	IC	75.93	230	14.61	468.0	256	61.8
10	ResNet_v2_50	IC	75.60	98	6.23	1,119.7	256	58.1
11	ResNet_v1_50	IC	75.20	98	6.19	1,284.6	256	67.5
12	AI_Matrix_ResNet50	IC	74.38	98	5.99	1,060.3	256	57.9
13	Inception_v2	IC	73.90	43	6.45	2,032.0	128	68.2
14	AI_Matrix_DenseNet121	IC	73.29	31	12.80	846.4	32	49.3
15	MLPerf_MobileNet_v1	IC	71.68	17	3.15	2,576.4	128	52.0
16	VGG16	IC	71.50	528	21.33	687.5	256	74.7
17	VGG19	IC	71.10	548	22.10	593.4	256	76.7
18	MobileNet_v1.1.0_224	IC	70.90	16	3.19	2,580.6	128	51.9
19	AI_Matrix_GoogleNet	IC	70.01	27	5.35	2,464.5	128	62.9
20	MobileNet_v1.1.0_192	IC	70.00	16	3.11	3,460.8	128	52.5
21	Inception_v1	IC	69.80	26	5.30	2,576.6	128	63.7
22	BVLC_GoogLeNet_Caffe	IC	68.70	27	6.53	951.7	8	55.1
23	MobileNet_v1.0.75_224	IC	68.40	10	3.18	3,183.7	64	51.1
24	MobileNet_v1.1.0_160	IC	68.00	16	3.01	4,240.5	64	55.4
25	MobileNet_v1.0.75_192	IC	67.20	10	3.05	4,187.8	64	51.8
26	MobileNet_v1.0.75_160	IC	65.30	10	2.81	5,569.6	64	53.1
27	MobileNet_v1.1.0_128	IC	65.20	16	2.91	6,743.2	64	55.9
28	MobileNet_v1.0.5_224	IC	63.30	5.2	3.55	3,346.5	64	63.0
29	MobileNet_v1.0.75_128	IC	62.10	10	2.96	8,378.4	64	55.7
30	MobileNet_v1.0.5_192	IC	61.70	5.2	3.28	4,453.2	64	63.3
31	MobileNet_v1.0.5_160	IC	59.10	5.2	3.22	6,148.7	64	63.7
32	BVLC_AlexNet_Caffe	IC	57.10	233	2.33	2,495.8	16	36.3
33	MobileNet_v1.0.5_128	IC	56.30	5.2	3.20	8,924.0	64	64.1
34	MobileNet_v1.0.25_224	IC	49.80	1.9	3.40	5,257.9	64	60.6
35	MobileNet_v1.0.25_192	IC	47.70	1.9	3.26	7,135.7	64	61.2
36	MobileNet_v1.0.25_160	IC	45.50	1.9	3.15	10,081.5	256	68.4
37	MobileNet_v1.0.25_128	IC	41.50	1.9	3.15	10,707.6	256	80.2
38	Faster_RCNN_NAS	OD	43	405	5079.32	0.6	4	85.2
39	Faster_RCNN_ResNet101	OD	32	187	91.15	14.67	4	13
40	SSD_MobileNet_v1_FPN	OD	32	49	47.44	33.46	8	4.8
41	Faster_RCNN_ResNet50	OD	30	115	81.19	16.49	4	10.8
42	Faster_RCNN_Inception_v2	OD	28	54	61.88	22.17	4	4.7
43	SSD_Inception_v2	OD	24	97	50.34	32.26	8	2.5
44	MLPerf_SSD_MobileNet_v1_300x300	OD	23	28	47.49	33.51	8	0.8
45	SSD_MobileNet_v2	OD	22	66	48.72	32.4	8	1.3
46	MLPerf_SSD_ResNet34_1200x1200	OD	20	81	87.4	11.44	1	14.9
47	SSD_MobileNet_v1_PPN	OD	20	10	47.07	33.1	16	0.6
48	Mask_RCNN_Inception_ResNet_v2	IS	36	254	382.52	2.92	4	29.2
49	Mask_RCNN_ResNet101_v2	IS	33	212	295.18	3.6	2	42.4
50	Mask_RCNN_ResNet50_v2	IS	29	138	231.22	4.64	2	40.3
51	Mask_RCNN_Inception_v2	IS	25	64	86.86	17.25	4	5.7
52	DeepLabv3_Xception_65	SS	87.8	439	72.55	13.78	1	49.2
53	DeepLabv3_MobileNet_v2	SS	80.25	8.8	10.96	91.27	1	42.1
54	DeepLabv3_MobileNet_v2_DM0.5	SS	71.83	7.6	9.5	105.21	1	41.5
55	SRGAN	SR	-	5.9	70.29	14.23	1	62.3

Table 4.9: In-depth characterization of the 37 image classification models listed in Table 4.8 at the optimal batch sizes on `Tesla_v100`. The model execution is partitioned into beginning (*B*), middle (*M*), and end (*E*) intervals based on layer index. The most intensive stages for latency, memory allocation, flops and memory access are shown.

ID	Batch Latency (ms)	GPU Latency Percentage (%)	GPU Gflops	GPU DRAM Read (GB)	GPU DRAM Write (GB)	GPU Achieved Occupancy (%)	Arithmetic Intensity (Flops/byte)	Arithmetic Throughput (TFlops)	Memory Bound?	Latency Stage	Allocated Memory Stage	flops Stage	Memory Access Stage
1	400.06	94.77	2,910.44	50.64	38.74	39.74	32.56	7.68	✗	M	M	M	M
2	324.49	93.92	2,492.92	27.25	24.48	33.79	48.19	8.18	✗	M	M	M	M
3	86.39	88.05	552.22	10.54	8.18	34.6	29.50	7.26	✗	M	M	M	B
4	593.97	96.32	3,954.06	58.90	65.44	43.51	31.80	6.91	✗	E	E	M	E
5	412.37	94.90	2,725.14	39.08	44.62	42.88	32.56	6.96	✗	E	E	M	E
6	517.11	95.90	3,947.38	51.17	54.77	42.78	37.26	7.96	✗	E	E	M	E
7	275.05	92.43	1,742.39	24.40	32.61	43.15	30.62	6.85	✗	B	E	M	E
8	360.90	94.29	2,720.62	33.87	37.12	42.19	38.32	7.99	✗	E	E	M	E
9	591.47	96.29	4,034.74	63.70	72.16	43.9	29.70	7.08	✗	B	M	B	M
10	245.07	91.74	1,480.10	21.84	28.29	42.96	29.52	6.58	✗	E	E	M	E
11	213.52	90.42	1,477.33	18.79	22.76	42.29	35.56	7.65	✗	E	E	M	E
12	257.80	91.89	1,561.76	24.86	33.39	44.26	26.81	6.59	✗	B	M	B	M
13	68.27	83.62	363.33	9.67	7.32	40.23	21.38	6.36	✗	B	B	M	B
14	40.24	93.32	150.02	10.13	7.93	44.94	8.30	4.00	✓	B	B	B	B
15	51.57	79.76	148.18	7.08	6.81	52.58	10.67	3.60	✓	M	M	M	M
16	399.31	94.98	2,655.39	24.38	33.23	26.14	46.10	7.00	✗	B	B	M	E
17	464.47	95.61	3,207.02	26.44	37.65	24.91	50.04	7.22	✗	B	B	M	E
18	51.59	79.73	148.18	6.97	6.75	52.59	10.80	3.60	✓	M	M	M	M
19	56.08	80.20	259.14	7.63	6.18	42.16	18.76	5.76	✗	M	B	M	B
20	38.48	79.55	108.93	6.51	6.19	52.32	8.58	3.56	✓	M	M	M	B
21	53.35	79.43	252.06	7.21	5.61	41.74	19.67	5.95	✗	M	B	M	B
22	9.08	80.00	20.26	0.73	0.84	33.87	12.97	2.79	✓	E	B	E	B
23	20.82	73.14	45.10	4.86	4.11	52.73	5.03	2.96	✓	M	M	M	M
24	14.92	78.26	38.17	3.24	2.88	48.92	6.23	3.27	✓	M	M	M	M
25	15.69	72.61	33.10	3.52	3.08	52.02	5.01	2.91	✓	M	M	M	M
26	11.30	71.86	23.14	2.31	2.17	51.01	5.17	2.85	✓	M	M	M	M
27	9.86	77.23	24.39	1.90	1.84	47.78	6.54	3.20	✓	M	M	M	M
28	20.00	71.93	52.03	2.99	2.85	43.87	8.91	3.62	✓	B	M	B	M
29	7.75	71.35	14.80	1.26	1.35	47.12	5.68	2.68	✓	M	M	M	M
30	15.07	71.75	38.22	2.08	2.09	43.27	9.17	3.53	✓	B	M	B	M
31	10.91	71.38	26.62	1.29	1.42	41.43	9.83	3.42	✓	B	M	B	M
32	6.52	68.69	15.36	0.76	0.51	37.31	12.11	3.43	✓	B	B	B	B
33	7.44	70.48	17.05	0.71	0.88	39.88	10.73	3.25	✓	B	M	B	M
34	11.95	53.93	14.79	1.25	1.42	44.25	5.52	2.30	✓	B	M	B	M
35	9.09	53.68	10.87	0.84	1.02	43.46	5.82	2.23	✓	B	M	B	M
36	25.36	60.78	36.75	3.26	3.09	42.39	5.79	2.38	✓	B	M	B	M
37	23.71	70.01	23.81	1.87	2.31	39.8	5.69	1.43	✓	M	M	B	M

Transpose, Add, and Mul). Finally, ⑤ the super resolution model SRGAN is dominated by convolution layers.

GPU latency, flops and memory accesses — Using the model-, layer-, and GPU kernel-level profiling, we perform an in-depth analyses of the 37 image classification models at their optimal batch sizes on `Tesla_V100`. Table 4.9 shows the model’s latency at the optimal batch size, GPU latency percentage (i.e. the latency due to GPU kernel execution normalized to the model latency), GPU metrics, and arithmetic intensity and throughput. It also shows the most intensive stage for latency, memory allocation, GPU flops, and memory access throughout the model execution. We find that across the models the GPU latency percentage varies from 53.68% to 95.61% and is roughly proportional to the number of flops and memory accesses (the sum of GPU DRAM reads and writes). We also observe that models with high batch latency tend to have a high GPU latency percentage. This either suggests that the GPU saturates for these models or that the models are not well optimized for GPU execution. The low GPU latency percentage for some models shows that the time

Table 4.10: Characterization of 10 MXNet models, which are comparable to the TensorFlow ones listed in Table 4.8 (labeled with the same ID). The online latency is measured at batch size 1 and the others are measured at the model’s optimal batch size on `Tesla.V100`. The online latency and maximum throughput are normalized to TensorFlow’s.

ID	Name	Normalized Online Latency	Optimal Batch Size	Normalized Maximum Throughput	GPU Latency Percentage	GPU Gflops	GPU DRAM Read (GB)	GPU DRAM Write (GB)	GPU Achieved Occupancy (%)	Arithmetic Intensity (Flops/byte)	Arithmetic Throughput (TFlops)	Memory Bound?
4	ResNet_v2.152	1.76	256	1.03	97.00	4,116.42	49.05	52.62	46.91	38.61	7.95	✗
5	ResNet_v2.101	1.59	256	1.02	96.77	2,882.65	32.33	36.16	46.38	40.14	7.96	✗
6	ResNet_v1.152	1.68	256	0.90	96.20	3,828.11	51.29	55.00	49.40	34.35	7.54	✗
8	ResNet_v1.101	1.60	256	0.91	95.67	2,589.76	33.93	37.84	49.57	34.42	7.45	✗
10	ResNet_v2.50	1.41	256	1.03	97.10	1,636.10	17.03	22.60	46.98	39.37	7.60	✗
11	ResNet_v1.50	1.32	256	0.96	94.90	1,339.50	18.37	24.04	51.97	30.12	6.76	✗
18	MobileNet_v1.1.0.224	1.00	256	1.54	93.75	298.38	6.91	8.29	63.53	18.71	4.96	✗
23	MobileNet_v1.0.75.224	0.95	64	1.76	79.49	45.00	3.47	2.73	63.38	6.92	4.08	✓
28	MobileNet_v1.0.5.224	0.87	64	1.35	81.01	51.47	1.99	1.82	48.68	12.88	4.49	✓
34	MobileNet_v1.0.25.224	0.93	64	1.64	64.32	13.77	0.81	0.90	50.57	7.64	2.88	✓

spent within non-GPU code (framework overhead, GPU stalls due to synchronization, etc.) is high.

Batch size vs GPU achieved occupancy — The GPU achieved occupancy is a partial indicator of GPU utilization. Table 4.6 shows that as a model’s batch size approaches the optimal, its overall achieved GPU occupancy increases.

Roofline analysis — Figure 4.12 shows the roofline analysis for all 37 image classification models with their optimal batch sizes on `Tesla.V100`. Out of 37 models, 20 are memory-bound. Models with low compute and memory requirements tend to be memory-bound and have lower accuracy, e.g. some variants of `MobileNet` which target edge devices. All models achieve at most 52% of the theoretical peak throughput, suggesting that there is room for optimizations.

Latency, memory allocation, flops, and memory access trend — To understand the performance trend within model execution, we divide the model execution into 3 intervals, beginning, middle, and end, based on the layer index. Each stage includes one third of the total layers. We then compute the total latency, flops, and memory accesses within each interval and identify which interval dominates. The last 4 columns in Table 4.9 show the results of the 37 image classification models on `Tesla.V100`. The demanding intervals vary across models and suggest that one can potentially interleave multiple model executions to increase GPU utilization.

4.3.2 ML Framework Evaluation

To compare ML frameworks, 10 MXNet models are selected from the MXNet model zoo [47]. We choose 6 variants of `ResNet` which are compute-intensive and are compute-bound (at the optimal batch size), and 4 variants `MobileNet` which are less compute-intensive and are memory-bound. The models (shown in Table 4.10) are comparable to the TensorFlow

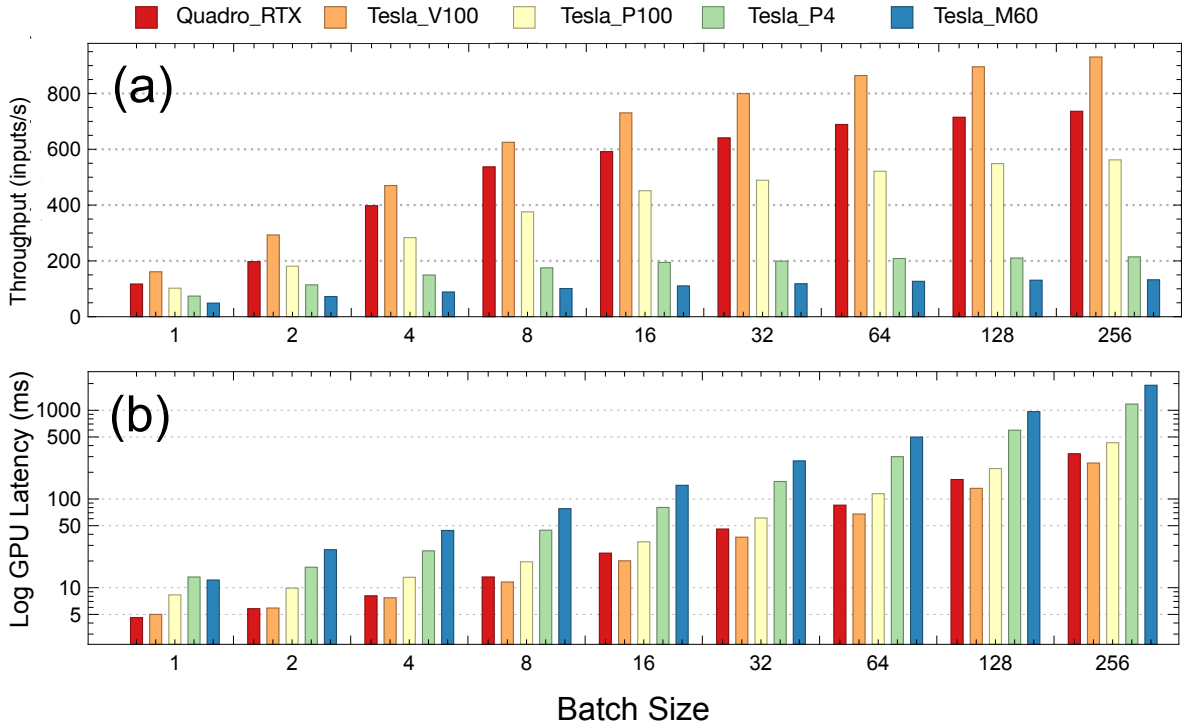


Figure 4.11: The throughput and latency (log scale) of MLPerf_ResNet50_v1.5 across batch sizes and systems.

models. We perform the comparison between the TensorFlow and MXNet frameworks on Tesla_V100. The online latency and maximum throughput in the Table 4.10 are normalized to the corresponding values using TensorFlow. We use XSP to compute the optimal batch size for each MXNet model. Except for model 18, the optimal batch size for all MXNet models match the corresponding TensorFlow models.

Compute-bound models — Table 4.10 shows that the online latency (batch size 1) of MXNet ResNets is higher than that of the corresponding TensorFlow model. After looking into the analysis results, we find that while the total GPU kernel latencies of TensorFlow and MXNet ResNets are about the same, the MXNet ResNets have a much higher non-GPU latency. MXNet ResNet_v1.50, for example, has a non-GPU latency of 4.44ms (55.1% of the total online latency) whereas it is only 2.18ms for TensorFlow ResNet_v1.50 (35.3% of the total). We observe that as the batch size increases (and the model becomes more compute-bound) the percentage of the non-GPU latency decreases and MXNet ResNets achieve about the same maximum throughput as TensorFlow ResNets. At the optimal batch size, TensorFlow and MXNet ResNets have comparable GPU latency percentage, flops, memory accesses, achieved occupancy, and roofline results. This suggests that MXNet incurs a fixed overhead for model execution which is more pronounced for small batch sizes.

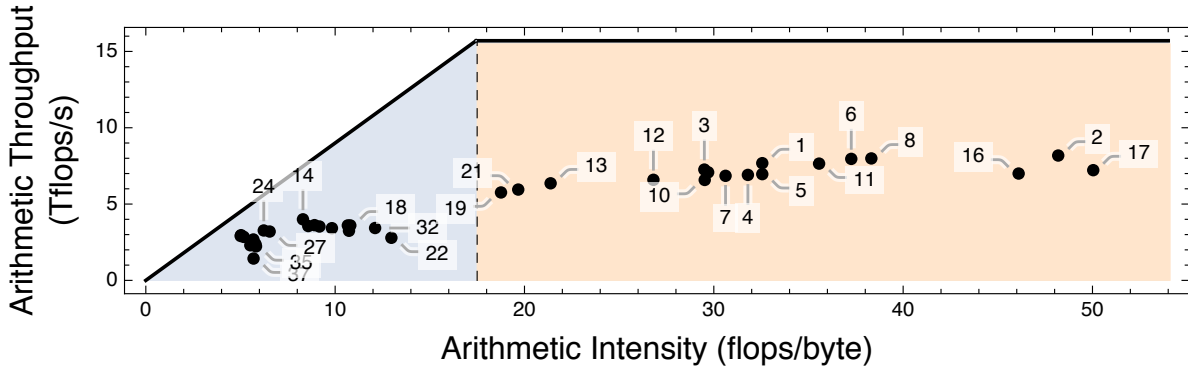


Figure 4.12: The roofline analysis for the 37 image classification models with their the optimal batch sizes on `Tesla_V100`.

Memory-bound models — For the less compute-intensive `MobileNets`, we observe that `MXNet` achieve the same online latency as the corresponding `TensorFlow` model. However, as the batch size increases (and the models become memory-bound). we find that `MXNet MobileNets` has fewer memory accesses and therefore a higher achieved GPU occupancy compared to the `TensorFlow` models. As a result, `MXNet MobileNets` achieve between 35% and 74% more throughput at their optimal batch sizes (shown in Table 4.10). Further GPU kernel-level analysis attributes the cause to the `Eigen` library. The `Eigen` library is used by `TensorFlow` (but not `MXNet`) for element-wise layers and it incurs excessive DRAM reads and writes. This becomes a performance-limiting factor for memory-bound models.

4.3.3 System Evaluation

We use `XSP` to evaluate `MLPerf_ResNet50_v1.5` on all 5 GPU systems in Table 4.7 using the `NGC TensorFlow` container. We fix the software stack (`TensorFlow`, `cuDNN`, `cuBLAS`, `CUDA` version, etc.) on all 5 systems to be the same. Figure 4.11a shows the throughput across systems and batch sizes. Figure 4.11b shows the GPU latency (the total latency of all the GPU kernel calls) in log scale for the 5 systems across batch sizes. Although the `Quadro_RTX` GPU has a slightly higher peak FLOPS compared to `Tesla_V100`, it has a much lower memory bandwidth. Hence, `Quadro_RTX` struggles on memory-bound layers and performs slightly worse when compared to `Tesla_V100`. We observe that the performance at each batch size differs across systems. The performance also scales differently across systems with respect to the batch size.

Looking at the GPU kernel-level profile for each system, we find that the GPU kernels invoked are system-dependent — even with the same batch size and software stack. Both `Quadro_RTX` and `Tesla_V100` call the same set of GPU kernels, while the other 3 systems use

a different set of GPU kernels. This is because the same cuDNN API may use different GPU kernels for different GPU systems. For example, the convolution layers for batch size 256 on `Tesla_P100`, `Tesla_P4`, and `Tesla_M60` invoke the `maxwell_scudnn_*` kernels, whereas on `Quadro_RTX` and `Tesla_V100` the `volta_scudnn_*` kernels are invoked. This implies that cuDNN uses optimized kernels for GPU generations after Volta. Furthermore, because of the cuDNN algorithm selection heuristics, the distribution of the kernel calls differs across systems. For example, `Tesla_V100` calls the `volta_scudnn_128x64_relu_interior_nn_v1` kernel 34 times whereas `Quadro_RTX` calls it 18 times (the other 16 being dispatched to the `volta_scudnn_128x128_relu_interior_nn_v1` kernel).

4.4 CONCLUSION

A big hurdle in optimizing and deploying ML workloads is understanding their performance characteristics across the HW/SW stack. The analyses currently performed on ML models and systems are largely limited by the lack of correlation between profiles from different profiling tools or methods. This paper proposes XSP, an across-stack profiling design that aggregates profile data from different sources and correlates them to construct a holistic and hierarchical view of ML model execution. While the across-stack profiling design is general, this paper focuses on how it enables in-depth automated profiling and characterization of ML models on GPUs. We use XSP’s profiling and analysis capabilities to systematically characterize 65 state-of-the-art ML models. Through the 15 types of analysis introduced, we derive meaningful insights that would otherwise be difficult to discern without XSP. We show that XSP helps researchers understand the sources of inefficiency in ML models, frameworks, and systems.

CHAPTER 5: BENANZA: AUTOMATIC μ BENCHMARK GENERATION TO COMPUTE “LOWER-BOUND” LATENCY AND INFORM OPTIMIZATIONS OF DEEP LEARNING MODELS

This chapter presents Benanza, a sustainable and extensible benchmarking and analysis design that speeds up the characterization/optimization cycle of DL models on GPUs. Benanza consists of four major components: a model processor that parses models into an internal representation, a configurable benchmark generator that automatically generates micro-benchmarks given a set of models, a database of benchmark results, and an analyzer that computes the “lower-bound” latency of DL models using the benchmark data and informs optimizations of model execution. The “lower-bound” latency metric estimates the ideal model execution on a GPU system and serves as the basis for identifying optimization opportunities in frameworks or system libraries.

Both industry and academia have invested heavily in developing benchmarks to characterize DL models and systems [3–7]. Characterization is followed by optimizations to improve the model performance. However, there is currently a gap between the benchmarking results and possible optimizations to perform. Researchers use profilers, such as nvprof [8], Nsight [9], and VTune [10], to profile and get low-level GPU and CPU information. With ample knowledge of how models execute and utilize system resources, researchers manually identify bottlenecks and inefficiencies within model execution using the profilers. Researchers then make hypotheses of solutions, and try out different ideas to optimize the model execution — which may or may not pan out. This manual and ad-hoc process requires a lot of effort and expertise and slows down the turnaround time for model optimization and system tuning.

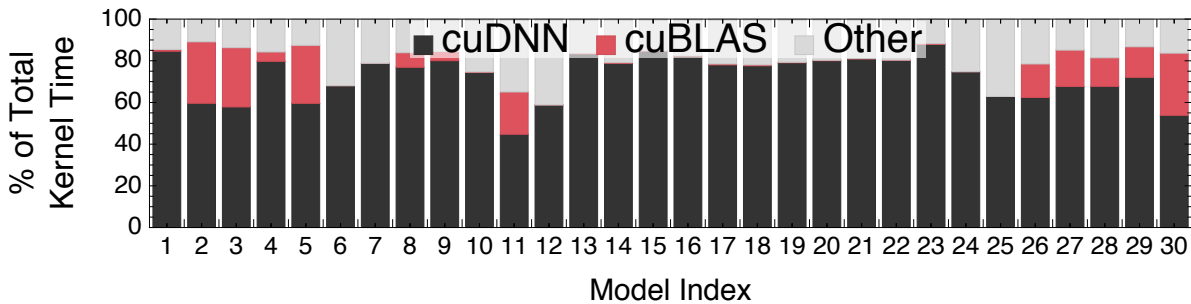


Figure 5.1: The GPU kernel time breakdown for all 30 models (listed in Table 5.1) on Tesla_V100 (Table 5.3) using batch size 1. Both cuDNN and cuBLAS invoke child GPU kernel(s) asynchronously in the model executions, we therefore measure the time of kernels launched by cuDNN and cuBLAS APIs rather than the time of the API itself.

Thus there is a need for a systematic DL benchmarking and subsequent analysis design that can guide researchers to potential optimization opportunities and assess hypothetical execution scenarios. Since for GPUs model execution latency is determined by the hardware, framework, and system libraries (primarily cuDNN [42] and cuBLAS [48] for DL), answers to the following questions are highly desired by researchers: **Question 1** what is the potential latency speedup if optimizations are performed? **Question 2** Are independent layers executed in parallel? **Question 3** Are convolution layers using the optimal convolution algorithms? **Question 4** Are there any inefficiencies or unexpected behavior in a framework? Does the execution **Question 5** fuse layers or **Question 6** leverage Tensor Cores, and what are the benefits? We motivate our design by answering these 6 questions, while ensuring the sustainability and extensibility of the design.

To answer these questions, we first propose a new benchmarking metric: “*lower-bound*” *latency*. The “lower-bound” latency estimates the ideal latency of a DL model given a software and hardware stack, and is based on the following observations: (1) DL models are executed as layers in frameworks and thus layers form the performance building blocks of DL models. (2) Frameworks delegate execution of common layers to either cuDNN or cuBLAS (shown in Figure 5.1). The “lower-bound” latency is defined in terms of the latencies of the cuDNN and cuBLAS API functions invoked by model layers (framework overhead and memory transfers are ignored). We refine the “lower-bound” latency and define it under *sequential execution mode* (all layers are executed sequentially) and *parallel execution mode* (data-independent layers are executed asynchronously).

This chapter presents Benanza (pronounced bonanza) — an sustainable and extensible benchmarking and analysis design. Benanza consists of a set of modular components: (1) a model processor to process input ONNX models into a set of *unique layers* (layers are considered the same if they have the same layer type, shape, and parameters), (2) a benchmark generator to automatically generate parameterized cuDNN and cuBLAS micro-benchmarks from the unique layers, (3) a performance database to store historical benchmark results, and (4) an analyzer to compute the “lower-bound” latency of DL models and inform potential optimizations (**Question 1-6**).

Benanza is architected to be sustainable. The benchmarking workflow of Benanza is highly automated and minimizes the benchmark development and maintenance effort. Benanza uses the observation that DL models have repeated layers (i.e. non-unique) within and across models to decrease the time to benchmark. When a new model is introduced, only the newly un-benchmarked layers that do (not in the performance database) need to be benchmarked. Although the focus of the chapter is on NVIDIA GPUs using cuDNN and cuBLAS, the design proposed is extensible and users can incorporate other benchmark

runtimes that target other software libraries or hardware such as: frameworks’ API or MKL-DNN for CPUs.

In summary, this paper makes the following contributions:

- We propose a “lower-bound” latency metric for DL models based on the observation that the latency of a DL model is bounded by the latencies of the cuDNN and cuBLAS API calls corresponding to the model layers. The “lower-bound” latency metric estimates the ideal latency of a model given a specific GPU hardware and software stack.
- We present Benanza, a novel benchmarking and analysis system designed to automatically generate micro-benchmarks given a set of models; compute their “lower-bound” latencies using the benchmark data; and inform optimizations of their execution on GPUs. Benanza is sustainable and extensible to cope with the fast evolution of DL innovations.
- Using Benanza, we characterized the “lower-bound” latencies of 30 ONNX models (shown in Table 5.1) using MXNet, ONNX Runtime, and PyTorch on 7 systems (shown in Table 5.3). We performed a comprehensive “lower-bound” latency analysis as we vary the model, execution mode, batch size, and system. E.g., when using parallel execution mode, up to $2.87\times$ (with a geometric mean of $1.32\times$ across models) latency speedup could be made to MXNet using batch size 1 on the `Tesla_V100` system.
- We identified optimization opportunities through Benanza in cuDNN convolution algorithm selection (up to $1.32\times$ geometric mean speedup across models), inefficiencies within MXNet (up to $1.15\times$ speedup across models) and PyTorch (up to $2.3\times$ speedup using batch size 1) frameworks, and layer fusion and Tensor Cores (up to $1.09\times$ and $1.72\times$ speedup for `ResNet50-v1` respectively). We further demonstrated that when performed jointly, these optimizations achieve up to $1.95\times$ speedup for `ResNet50-v1` across systems and batch sizes.

5.1 MOTIVATION

5.1.1 DL Model Execution and ONNX Format

A DL model is an execution graph where each vertex is a layer operator (e.g. convolution, activation, normalization, pooling, or softmax). These layer operators (or *layers* for short) are functions defined by a DL framework. A framework executes a model by traversing the model graph in topological order and enqueueing the layers into an execution queue. Although sequential evaluation is always valid, frameworks strive to execute data-independent layers within the queue in parallel. Through careful execution scheduling, a framework can overlap communication with computation, increase utilization, etc. Regardless of the execution strategy, however, layer execution latency is the limiting factor for model execution. As

such, layers are not only the building blocks by which developer define models, but are also the atomic components that define a model’s performance characteristics.

Each framework provides its own API, layer definition semantics, model storage format, and model executing strategy. To increase interoperability between frameworks, there have been concerted efforts [49, 50] to standardize layer definitions and model exchange format. A leading effort is the Open Neural Network Exchange Format (ONNX), which has wide industry and framework backing. Frameworks such as Caffe2, CNTK, MXNet, Paddle, PyTorch, and TensorRT readily support ONNX, and converters exist for other frameworks such as TensorFlow and Caffe. To perform a fair comparison between frameworks (by evaluating them using the same ONNX model), and more importantly, to make Benanza framework-agnostic, we choose ONNX as the model input format for Benanza. ONNX hosts all their models publicly [51] and, we select 30 vision models out of the 32 models available at the time of writing for evaluation (the 2 models not selected are non-vision models). The selected models cover an array of tasks and are listed in Table 5.1. We refer to these models by their IDs throughout the paper.

5.1.2 cuDNN and cuBLAS

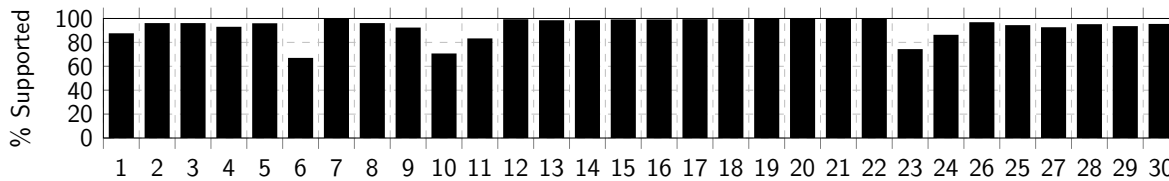


Figure 5.2: The percentage of layers supported by cuDNN and cuBLAS (also covered by Benanza) for each model in Table 5.1.

Much like BLAS or LAPACK are the backbone of HPC computing, cuDNN and cuBLAS are the backbones of the GPU software stacks for DL. cuDNN is a GPU-accelerated library and provides highly tuned implementations of DL layers such as convolution, pooling, normalization, activation. cuBLAS is a GPU-accelerated BLAS library and provides fast implementations of GEMM and GEMV. The DL layers supported by each API are listed in

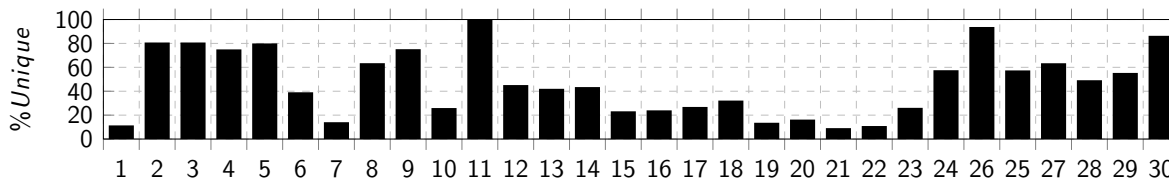


Figure 5.3: The percentage of unique layers within the 30 models

Table 5.1: The 30 ONNX models used are vision models which encompass image classification (**IC**), object detection (**OD**), face recognition (**FR**), emotion recognition (**ER**), semantic segmentation (**SS**), or hand digit recognition (**HR**) tasks.

ID	Name	Task	MACs	# Layers	Year
1	Arcface [52]	FR	12.08G	412	2018
2	BVLC-Alexnet [53]	IC	656M	24	2012
3	BVLC-Caffenet [53]	IC	721M	24	2012
4	BVLC-Googlenet [54]	IC	1.59G	143	2014
5	BVLC-RCNN-ILSVRC13 [55]	IC	718M	23	2013
6	Densenet-121 [56]	IC	2.87G	910	2016
7	DUC [57]	SS	34.94G	355	2017
8	Emotion Ferplus [58]	ER	877M	52	2016
9	Inception-v1 [59]	IC	1.44G	144	2015
10	Inception-v2 [60]	IC	2.03G	509	2015
11	LeNet [61]	HR	796K	12	2010
12	MobileNet-v2 [62]	IC	437M	155	2017
13	Resnet18-v1 [63]	IC	1.82G	69	2015
14	Resnet18-v2 [64]	IC	1.82G	69	2016
15	Resnet34-v1 [63]	IC	3.67G	125	2015
16	Resnet34-v2 [64]	IC	3.67G	125	2016
17	Resnet50-v1 [63]	IC	3.87G	175	2015
18	Resnet50-v2 [64]	IC	4.10G	174	2016
19	Resnet101-v1 [63]	IC	7.58G	345	2015
20	Resnet101-v2 [64]	IC	7.81G	344	2016
21	Resnet152-v1 [63]	IC	11.30G	515	2015
22	Resnet152-v2 [64]	IC	11.53G	514	2016
23	Shufflenet [65]	IC	127M	203	2015
24	Squeezenet-v1.1 [66]	IC	352M	66	2016
25	Tiny Yolo-v2 [67]	OD	3.13G	32	2016
26	Vgg16-BN [15]	IC	15.38G	54	2014
27	Vgg16 [15]	IC	15.38G	41	2014
28	Vgg19-bn [15]	IC	19.55G	63	2014
29	Vgg19 [15]	IC	19.55G	47	2014
30	Zfnet512 [68]	IC	1.48G	22	2013

Table 5.2. And, while there is a wide array of DL frameworks, common between them is the reliance on these primitives defined by cuDNN and cuBLAS. In fact, all major DL frameworks, such as MXNet, PyTorch, ONNX Runtime, and TensorFlow, rely on cuDNN/cuBLAS API functions for the implementation of common layers.

Figure 5.2 shows the percentage of layers supported by cuDNN and cuBLAS for each model in Table 5.1. Most layers within DL models are covered by the cuDNN and cuBLAS API. The layers that are not supported are non-compute operators (such as concatenate, which joins two tensors across a specified axis) or datatype manipulations (such as reshape, which changes the dimensions of a tensor). For example, the cuDNN and cuBLAS functions support 70% of the layers within `Inception-v2` (ID = 10). This is because `Inception-v2`

Table 5.2: Eleven layer types are supported by cuDNN and two layer types are supported by cuBLAS. Each API may have auxiliary functions to setup its arguments (e.g. `cudaSetTensor4dDescriptor` to specify a tensor’s dimensions and `cudaSetConvolution2dDescriptor` to configure the convolution API). The convolution, RNN, and GEMM APIs have Tensor Core support.

Layer Type	cuDNN / cuBLAS API	Tensor Core Support
Convolution	<code>cudaConvolutionForward</code>	✓
Activation	<code>cudaActivationForward</code>	✗
BatchNorm	<code>cudaBatchNormalizationForwardInference</code>	✗
Conv+Bias+Activation	<code>cudaConvolutionBiasActivationForward</code>	✓
RNN	<code>cudaRNNTensorForwardInference</code>	✓
Dropout	<code>cudaDropoutForward</code>	✗
Pooling	<code>cudaPoolingForward</code>	✗
Softmax	<code>cudaSoftmaxForward</code>	✗
Add	<code>cudaAddTensor</code>	✗
Element-wise	<code>cudaOpTensor</code>	✗
Rescale	<code>cudaScaleTensor</code>	✗
GEMM	<code>cublas*Gemm / cublasGemmEx</code>	✓
GEMV	<code>cublasSgemv</code>	✗

makes heavy use of `unsqueeze` — a tensor reshape layer; 27% of the layers in `Inception-v2` are `unsqueeze` layers.

Given a specific DL software stack (e.g. framework, cuDNN, cuBLAS, and other CUDA libraries) and GPU hardware, the cuDNN and cuBLAS functions invoked by a model are fixed. Most common layers are supported by cuDNN and cuBLAS and the latency attributed to cuDNN and cuBLAS functions is significant with respect to the model’s end-to-end latency. Figure 5.1 shows that for the 30 vision models, the time spent within the cuDNN and cuBLAS API calls dominates the model execution time. The “other” time is due to either memory operations, synchronization, the framework’s choice of not using cuDNN API for certain operations, or other framework code that is neither cuDNN nor cuBLAS.

Based on the above observations, we propose a “lower-bound” latency metric for DL models. The “lower-bound” metric is defined by the latencies of the cuDNN and cuBLAS functions executed for the model layers within a specific software/hardware stack. The “lower-bound” latency is computed under different execution scenarios to determine if optimizations can be made, pinpoint where optimizations are, and quantify the potential benefits of optimizations, as detailed in Section 5.2.

5.2 BENANZA DESIGN AND IMPLEMENTATION

Benanza consists of four main components: Model Processor, Automatic Benchmark Generator, Performance Database, and Analyzer. The components are shown in Figure 5.4 and are used in the benchmarking and analysis workflows:

- **Benchmarking workflow:** ① The Model Processor takes ONNX models, parses them, performs shape inference, and finds the set of unique layers within the models. Two layers are considered the same (non-unique) if they have the same operator type and parameters (i.e. **only differ in weight values**). ② The Automatic Benchmark Generator then generates micro-benchmarks for each unique layer. The generated micro-benchmarks measure the latency (or the GPU kernel metrics if profiling mode is enabled) of the corresponding cuDNN or cuBLAS function calls for the layers. ③ The micro-benchmarks are then run on systems of interest and the results are stored in the Performance Database.
- **Analysis workflow:** ④ The user runs the target model using a framework on a system of interest with utilities provided by Benanza to get the model execution profile (i.e. the end-to-end latency, cuDNN and cuBLAS logs, and Nsight profile). ⑤ The user then specifies the model and system to Benanza. The model is parsed into layers and the Analyzer queries the latencies of each layer from the Performance Database (using the layers and system information provided) to compute the **Question 1** “lower-bound” latency under different execution scenarios. By analyzing the model execution profile and the computed “lower-bound”, the Analyzer informs optimizations in: **Question 2** parallel execution of independent layers, **Question 3** convolution algorithm selection, **Question 4** framework inefficiency, **Question 5** layer fusion, and **Question 6** Tensor Core usage.

5.2.1 Benanza Model Processor

The ① Model Processor parses ONNX models into Benanza’s internal representation (IR). The IR wraps around the ONNX Protobuf and has the same layer coverage. Since ONNX models do not have layer shapes information embedded (except for the input layers), shape inference [69] is performed to determine the shape of each layer. Layers in the IR (referred to as *layers* and correspond to the ONNX nodes) are annotated with the inferred shapes. Benchmarks are generated for each layer using its type, shape (i.e. all input dimensions), and parameters information ¹.

We observe that layers with the same type, shape, and parameters (i.e. **only differ in weight values**) are repeated extensively within and across models. Figure 5.3 shows that

¹The current design focuses on dense layers. Refer to Section 3.5 for discussion on sparse layers.

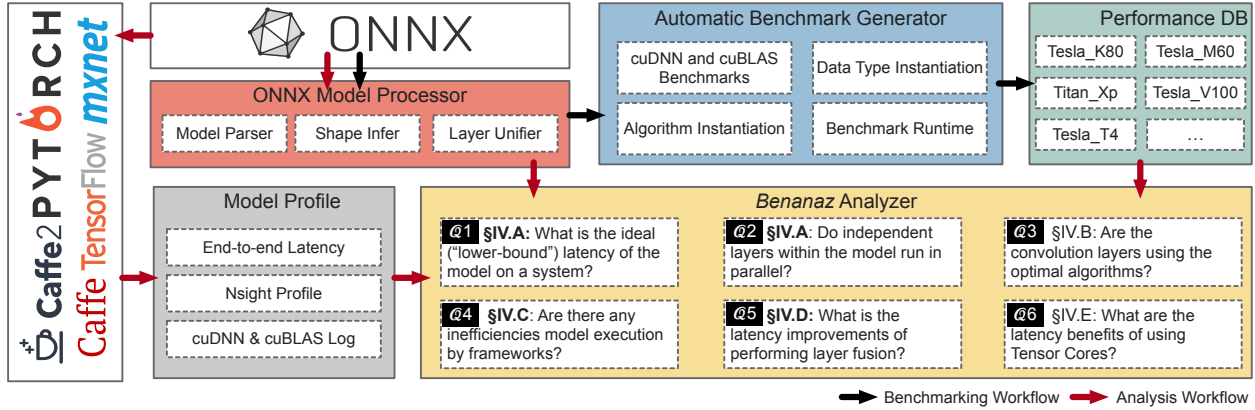


Figure 5.4: The Benanza design and workflow. **Question 1-6** are represented as **Q1-6**.

most models have a low percentage of unique layers — indicating that layers are repeated extensively within the model. For example, `ResNet50-v1` (ID=17) has 175 layers but only 47 (26.9%) are unique. The number of unique layers across models of similar architecture is also low. The `ResNet*-v1` models (ID=13, 15, 17, 19, 21) are built from the same modules and have a total of 1229 layers, of which only 60 (5.6%) are unique. Across all 30 models, the total number of layers is 5754, but only 1031 (18%) are unique. We exploit this layer repeatability to optimize the benchmark generation and minimize the time to benchmark. Thus, the Model Processor unifies the repeated layers across the input models and produces a set of unique layers. The time saved can be used to explore other algorithms and data types (Sections 5.2.2 and 5.2.2) benchmarks.

5.2.2 Automatic Benchmark Generator

The ② Automatic Benchmark Generator uses the set of unique layers (produced by the Model Processor) and generates C code to invoke the benchmark runtime using each layer’s type, shape, and parameters information.

The Benchmark Runtime

Benanza provides a benchmark runtime that measures the latency of the cuDNN or cuBLAS API required to execute each layer (as shown in Table 5.2). The runtime also sets up the function arguments for each API. The setup time is not included in the latency measurement. The runtime uses the Google Benchmark [70] library — a micro-benchmarking support library. The Google Benchmark library dynamically determines the number of iterations to run each benchmark and ensures that the reported latency results are statistically

stable. Generated benchmarks are linked with the cuDNN/cuBLAS libraries, and are run on systems of interest.

Algorithm Instantiation

The convolution layers map to the `cudaConvolutionForward` API (Table 5.2). The convolution API takes one of the following 8 algorithms as an argument: Implicit GEMM (IGEMM), Implicit PreComputed GEMM (IPGEMM), GEMM, Direct (DRCT), FFT, Tiled FFT (TFFT), Winograd (WING), and Winograd Non-Fused (WINGNF). These algorithms have different compute and memory characteristics [71, 72]. The optimal algorithm to use depends on the system, layer shape, and layer parameters (e.g. filter size, stride, dilation, etc.) [42]. For inference, most frameworks (e.g. MXNet, PyTorch, TensorFlow) rely on the cuDNN provided heuristic function (`cudaGetConvolutionForwardAlgorithm`) to choose the convolution algorithm. The heuristic function suggests an algorithm given the layer’s shape, parameters, data type, system, etc. To explore the design space of algorithm selection, by default, for each layer Benanza generates benchmarks using all algorithms applicable to the layer.

Data Type Support

Benanza can be configured to generate micro-benchmarks that target different data types. Both `float16` and `float32` are generated by default, but benchmarks can be instantiated for other data types. The `float16` benchmarks use Tensor Cores when the API function (see Table 5.2) and system (see Table 5.3) supports it.

Layer Fusion Support

Benanza can be configured to generate micro-benchmarks that target the cuDNN fused API (`cudaConvolutionBiasActivationForward`) to perform the convolution, bias, and activation layer sequence. Two fusion pattern rules are currently handled by Benanza: `Conv→Bias→Activation` and `Conv→Bias`. The `Conv→Bias→Activation` maps directly to the fused API. Fusing `Conv→Bias` is implemented through the fused API using `CUDNN_ACTIVATION_IDENTITY` as the activation function and requires cuDNN version ≥ 7.1 . For older cuDNN versions, the `Conv→Bias` is implemented as two calls — a `cudaConvolutionForward` followed by a `cudaAddTensor`. Users can extend Benanza’s fusion support by registering new fusion patterns as the cuDNN fused API evolves.

Integration with CUPTI

Benanza can be configured to generate benchmarks that integrate with low-level GPU profiler libraries such as NVIDIA’s CUPTI [30]. This allows Benanza to capture detailed GPU metrics [40] of benchmarks such as flops, memory transfers, etc. In this mode, the user specifies the metrics of interest, the number of benchmark iterations for warm-up, and the number of iterations to measure. Benanza does not use the Google Benchmark in this mode since a fixed, small number of profiling runs suffice for statistically stable measurement of the metrics. The profiling outputs (name, timing, and metric values of GPU kernels) are stored as metadata to the corresponding benchmark entry in the Performance Database.

5.2.3 Performance Database

The ③ benchmarking results are collected and published to Benanza’s Performance Database. Each entry within the database is indexed by the system, data type, and layer (type, shape, and parameter information). The Analyzer queries the database to get the benchmark latencies. If a query is a miss, then a warning with the information about the missing benchmark is issued to the user and the user is asked if they wish the Automatic Benchmark Generator to generate the missing benchmarks.

5.2.4 Benanza Analyzer

The ④ user runs the target model using a framework on a system of interest with utilities provided by Benanza to get the *model execution profile*. The model execution profile contains information about the model’s end-to-end latency, cuDNN and cuBLAS logs, and Nsight profile (which contains cuDNN/cuBLAS API calls and function backtrace information). Capturing the model end-to-end latency requires the user to place the provided timing functions within their application code. To capture the usage of cuDNN and cuBLAS functions within a framework, Benanza launches the user code with the `CUDNN_LOGINFO_DBG` and `CUBLAS_LOGINFO_DBG` environment variables. These environment variables enable the cuDNN and cuBLAS loggers respectively. Utilities to run the user code using NVIDIA’s Nsight profiler are also provided. The results from Nsight are parsed and correlated with the cuDNN and cuBLAS logs.

The ⑤ user then inputs the model execution profile along with the ONNX model, system, data type. The model is parsed by the Model Processor into layers. Then, the Benanza Analyzer queries the Performance Database for the benchmark latencies of each layer

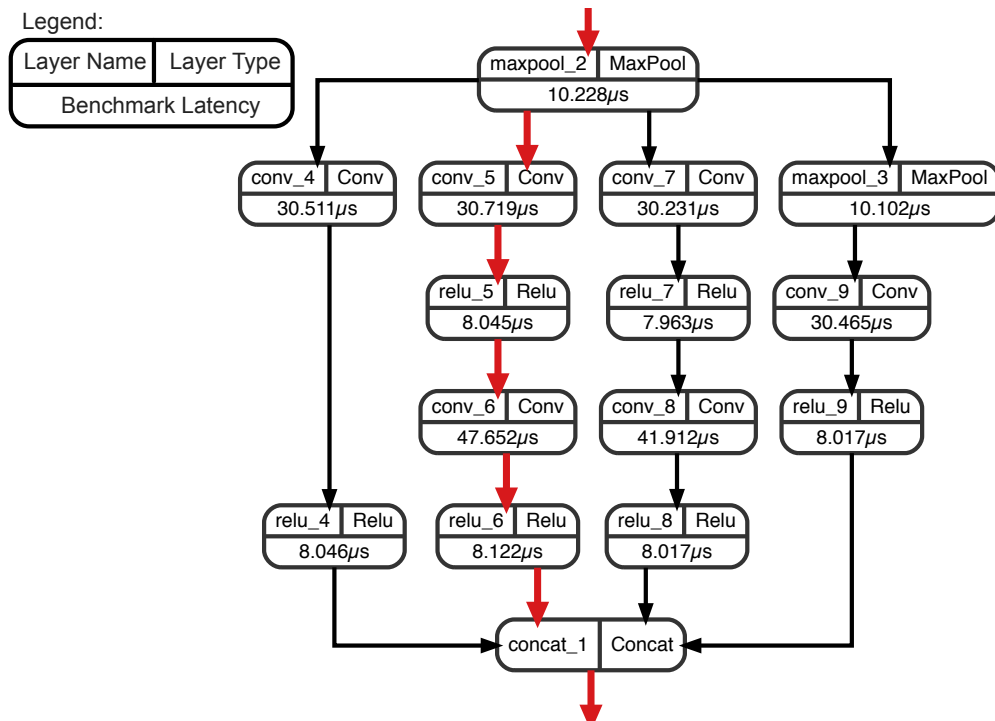


Figure 5.5: The first parallel module of Inception-v1 in Figure 5.7 visualized by the Benanza Analyzer. The layers are annotated with the name, type, and latency used for the “lower-bound” calculation. The critical path used in the parallel mode is highlighted in red.

using the user-specified system and data type (by default `float32`). Due to algorithm (Section 5.2.2) instantiation, multiple benchmarks may exist for a layer. The Analyzer, therefore, selects the benchmark result achieving the lowest latency. The following analyses are then performed:

Sequential and Parallel “Lower-Bound” Latency (**Question 1,2**)

DL models may contain layer sequences which can be executed independently in parallel. The sub-graph formed by these data-independent layer sequences is called a *parallel module*. For example, a parallel module in Inception-v1 is shown in Figure 5.5. A framework may execute the independent paths within the parallel module either sequentially or in parallel. Thus, the Analyzer computes the “lower-bound” latency of a model using two execution modes: sequential and parallel.

The sequential mode assumes that independent layers are executed sequentially, and therefore is defined as the sum of each layer’s benchmark latency. The parallel strategy assumes that data-independent layers are executed in parallel. Therefore, the parallel “lower-bound” latency is defined by the model’s *critical path* — the simple path from the start to the end

layer with the highest latency. Finding the critical path of a graph is a longest path problem and is NP-hard. Since a DL model forms a directed acyclic graph (DAG), the critical path can be framed as a shortest path problem [73]. To compute the critical path we construct a weighted DAG from the model graph where the edge weight between two nodes (layers) is negative of the latency of the layer at the tail of the edge. Computing the shortest path from the start to the end layer of the constructed weighted DAG produces the critical path of the model. The parallel “lower-bound” latency is the sum of layers latencies along the critical path. Benanza visualizes the critical path of the model (e.g. Figure 5.5), and the difference between the sequential and parallel “lower-bound” latencies indicates the profit of executing independent layers in parallel. Other analyses performed by Benanza leverage the sequential and parallel “lower-bound” latencies, and the benefits can be calculated in terms of either sequential or parallel mode.

Convolution Algorithm Selection (**Question 3**)

The Analyzer uses the parsed cuDNN log in the model execution profile to determine if the cuDNN algorithm used by the framework for each layer is optimal (recall from Section 5.2.2 that benchmark results using all available algorithms for layers exist in the Performance Database). Cases where the algorithm choice is sub-optimal are reported to the user along with how much end-to-end latency improvement could be gained if algorithm selection was ideal. The user can act upon these suggestions by forcing the framework to use specific algorithms.

Framework Inefficiency Inspection (**Question 4**)

The expected cuDNN and cuBLAS API calls are known to the Analyzer from the “lower-bound” latency computation. The Analyzer compares the model execution profile against the expected execution to pinpoint inefficiencies within the framework. The user is presented with any deviation observed in cuDNN or cuBLAS API invocation’s parameters or their execution order. CUDA API functions and CUDA kernels executed between cuDNN or cuBLAS API calls, are also presented to the user — along with their backtraces.

Layer Fusion Analysis (**Question 5**)

If the user enables the benchmark generation for layer fusion (as described in Section 5.2.2), then the Analyzer can be used to determine the potential profitability if layer fusion is em-

ployed. The Analyzer traverses the model layers and looks for the fusion pattern rules (listed in Section 5.2.2). If one of these patterns is found, then the corresponding fused operation’s latency is queried from the database and is used in the “lower-bound” computation (in either sequential or parallel model). If the benchmark is unavailable, or failed to run, then the latencies of the non-fused layers are used. The difference between the non-fused “lower-bound” latency and the fused “lower-bound” latency determines the profitability of layer fusion.

Tensor Core Analysis (**Question 6**)

The Analyzer determines if the target model execution utilizes Tensor Cores by looking at kernel names in the model execution profile.² Kernel names that match the `_[ish]\d+*` Regular-expression use Tensor Cores. E.g., kernels with names `trt_volta_int8_i8816cudnn_*` use Tensor Cores. By default, benchmarks targeting both `float16` and `float32` are generated. When benchmarks are run on systems with Tensor Core support, the difference between the “lower-bound” latency of `float32` and `float16` informs the profitability of using Tensor Cores and `float16`.

5.2.5 Sustainability and Extensibility

Sustainability of Benanza is ensured by providing an automated benchmark generation and analysis workflow design along with a continuously updated Performance Database. Benchmarking requires limited effort, as the micro-benchmarks are automatically generated, and the user only needs to compile and run the generated code on systems of interest. The Performance Database is continuously updated with new benchmark results. A big insight of the proposed design is that there is ample layer repeatability within and across models. This keeps the number of unique layers and thus the number of Performance Database entries in check over time. For new models, only the newly introduced unique layers are benchmarked.

For example, consider a scenario where all models in Table 5.1 except for `ResNet*-v2` have already been benchmarked and the results are in the Performance Database. Using our design, benchmarking the `ResNet*-v2` models requires measuring all the `ResNet*-v2` layers that are not within the Performance Database. Evaluating this hypothetical scenario results in a 75% reduction (30 minutes) in benchmarking time on the `Tesla_V100` system for batch size 32. The saving would be even larger on slower systems. By storing and reusing the

²Similar to DLProf [74], determining the Tensor Core utilization from the kernel names can identify cuDNN kernels that use Tensor Cores, but will not identify custom kernels or kernels outside of cuDNN.

micro-benchmark results in the Performance Database we minimize the time cost of running micro-benchmarks.

Benanza is extensible. As shown in Figure 5.4, Benanza is designed as a set of modular components. As new cuDNN functions are introduced, users update the Benanza runtime accordingly. For example, if a new cuDNN convolution algorithm is added, then the user can just add it to the list of algorithms to instantiate in the convolution benchmark implementation. If a new cuDNN/cuBLAS API or a fused API is added, then a user needs to add the benchmark implementation for the new API using the templates provided by Benanza as a basis. Users can also extend the Automatic Benchmark Generator to support other runtimes that target other software libraries or hardware, and leverage most of the other analysis components unmodified. These runtimes can target the frameworks’ Python or C++ API or other DL libraries (e.g. MIOpen [75] on AMD GPUs, or MKL-DNN [76] on CPUs). Through the novel benchmarking and analysis design, Benanza copes well with the fast evolving pace of DL innovations.

5.3 EVALUATION

We implemented Benanza and evaluated its design by answering **Question 1-6**. We evaluated 30 ONNX models (listed in Table 5.1) in the MXNet (v1.5.1), ONNX Runtime (v0.5.0), and PyTorch (v1.3) frameworks. Experiments were run on the 7 systems listed in Table 5.3. All systems use Ubuntu 18.04.3 LTS, CUDA 10.1.243, cuDNN Version 7.6.3, and CUDA Driver 430.26. The micro-benchmarks were compiled with GCC 7.4.0. We first computed the `float32` “lower-bound” latency in both sequential and parallel modes. Then we used the Analyzer to uncover and explore optimization opportunities — cuDNN heuristics, framework inefficiencies, layer fusion, and usage of Tensor Cores, and show their impact on the end-to-end latency.

5.3.1 “Lower-Bound” Latency vs. Measured Latency

We measured the inference latency of the 30 models using MXNet, ONNX Runtime, and PyTorch on the `Tesla_V100` system. Figure 5.6 shows the measured latency across all models and Figure 5.10 compares the latencies using different frameworks. Due to the lack of support of some ONNX operators by ONNX Runtime [77] and PyTorch [78], not all models run within these frameworks. As MXNet is the fastest in general, subsequent sections of the paper (with the exception of Section 5.3.3) focus on informing optimizations in MXNet..

Table 5.3: We used 7 GPU systems for evaluation. The systems cover the past GPU generations (from Kepler to the latest Turing). Amazon Web Service (AWS) is used for 4 of the systems and the other 3 are local machines. The 4 Turing and Volta GPUs support Tensor Cores and their theoretical Tensor Core performance — Tensor TFLOPS(tera floating point operations per second — are listed.

Name	CPU	GPU (Release Year)	GPU Architecture	GPU Memory Capacity, Bandwidth	Theoretical FP32 TFLOPS	Theoretical Tensor TFLOPS
Tesla_K80 (AWS P2)	Intel Xeon CPU E5-2686 v4	Tesla K80 (2014)	Kepler	12 GB, 480 GB/s	5.6	✗
Tesla_M60 (AWS G3)	Intel Core i9-7900X CPU	Tesla M60 (2015)	Maxwell	7 GB, 160.4 GB/s	4.8	✗
TITAN_Xp	Intel Xeon CPU E5-2686 v4	TITAN Xp (2017)	Pascal	12 GB, 547.6 GB/s	12.2	✗
TITAN_V	Intel Core i7-7820X CPU	TITAN V (2017)	Volta	12 GB, 672 GB/s	14.9	110.0
Tesla_V100 (AWS P3)	Intel Xeon CPU E5-2686 v4	Tesla V100 SXM2 (2018)	Volta	16 GB, 900 GB/s	15.7	125.0
Quadro_RTX	Intel Xeon CPU E5-2630 v4	Quadro RTX 6000 (2019)	Turing	24 GB, 624 GB/s	16.3	130.5
Tesla_T4 (AWS G4)	Intel Xeon Platinum 8259CL CPU	Tesla T4 (2019)	Turing	15 GB, 320 GB/s	8.1	65.0

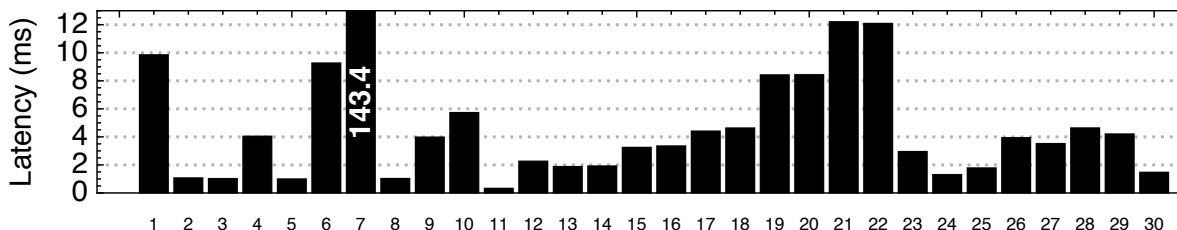


Figure 5.6: The measured latency of all ONNX models using batch size 1 with MXNet backend on Tesla_V100 in Table 5.3.

Sequential Mode vs Parallel Mode ([Question 1,2](#))

The difference between the “lower-bound” latency and the measured latency indicates the optimization opportunities in the framework and its use of the cuDNN and cuBLAS APIs. A model’s “lower-bound” latency normalized to its measured latency is referred to as its *Benanza Ratio* (BR). Figure 5.7 shows the BR in sequential ($BR_{sequential}$) and parallel mode ($BR_{parallel}$) in MXNet across all models using batch size 1 on the Tesla_V100 system.

The $BR_{sequential}$ across models has a geometric mean of 0.88, thus a potential latency speedup of $\frac{1.0}{0.88} = 1.14\times$ can be made to the measured model execution. The $BR_{parallel}$ across models has a geometric mean of 0.76, indicating a potential latency speedup of $\frac{1.0}{0.76} = 1.32\times$. The difference between a model’s parallel and sequential “lower-bound” latency depends on the existence of parallel modules within the model and how compute-intensive the data-independent paths are. Models without parallel modules have the same sequential and parallel “lower-bound” latency, thus the $BR_{sequential}$ is equal to the $BR_{parallel}$. For models with compute-intensive parallel modules, such as the Inception models (ID=4, 9, 10), the potential speedup of the latency (or $\frac{1}{BR_{parallel}}$) is $2.87\times$, $2.69\times$, and $2.45\times$ respectively. The $BR_{sequential}$ and $BR_{parallel}$ of LeNet (ID=11) are both low because LeNet is a simple model which has low latency ($0.33ms$ as shown in Figure 5.6) and the MXNet overhead and other non-compute portion is high, thus its BR is low.

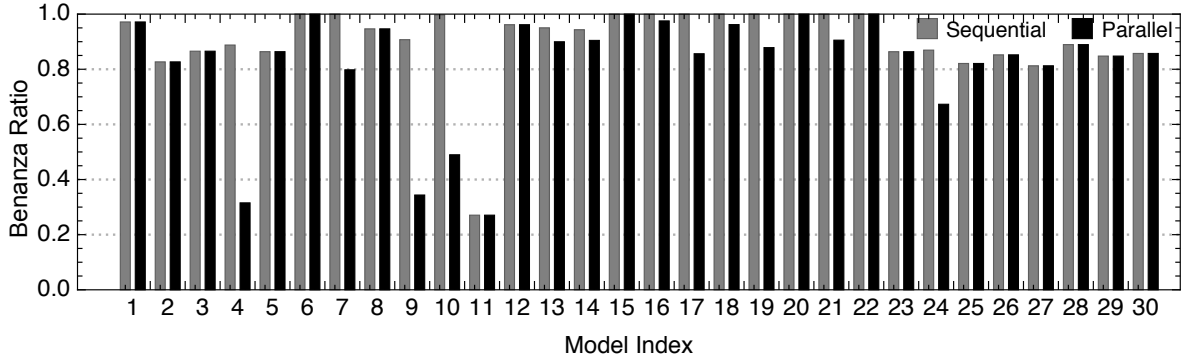


Figure 5.7: The Benanza Ratio in sequential and parallel mode of 30 models in MXNet using batch size 1 on Tesla_V100.

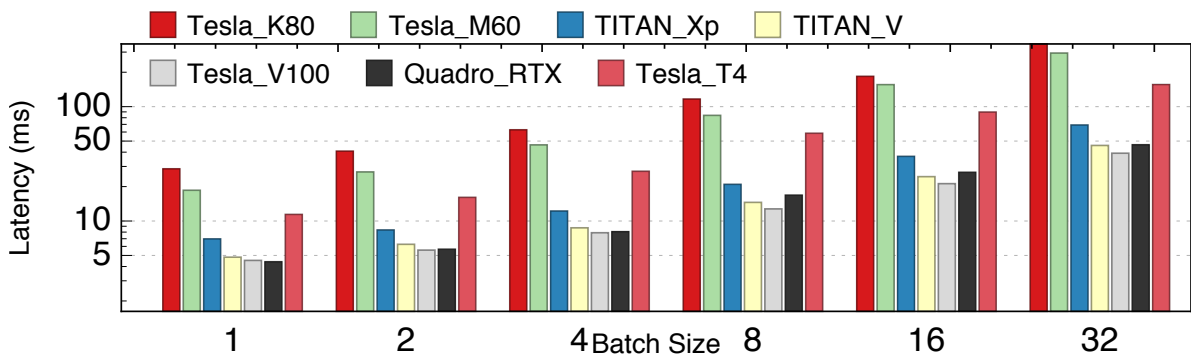


Figure 5.8: The measured latency of ResNet50_v1 in MXNet across batch sizes and systems.

The sequential “lower-bound” latency of the models with parallel modules (e.g. Inception and ResNet models) is closer to their measured latency when compared to the parallel “lower-bound” latency ($BR_{\text{parallel}} < BR_{\text{sequential}} < 1$). This suggests that parallel modules are executed sequentially in MXNet, even though the data-independent layers could be run in parallel. We verified the sequential execution behavior in MXNet by inspecting the model execution profile. Thus we evaluated the benefits of the latter optimizations in terms of the sequential “lower-bound” latency.

Batch Sizes and Systems

To demonstrate Benanza’s functions across batch sizes and systems, we evaluated the “lower-bound” latency of all models using different batch sizes from 1 to 32 on representative systems (shown in Table 5.3). We select batch size 32, since some models cannot be run using batch sizes beyond 32 due to GPU memory limitations. Figure 5.8 shows the measured latency of ResNet50-v1 on all systems in log scale. As expected, latencies are reversely correlated to the compute capability of the system (e.g. theoretical FP32

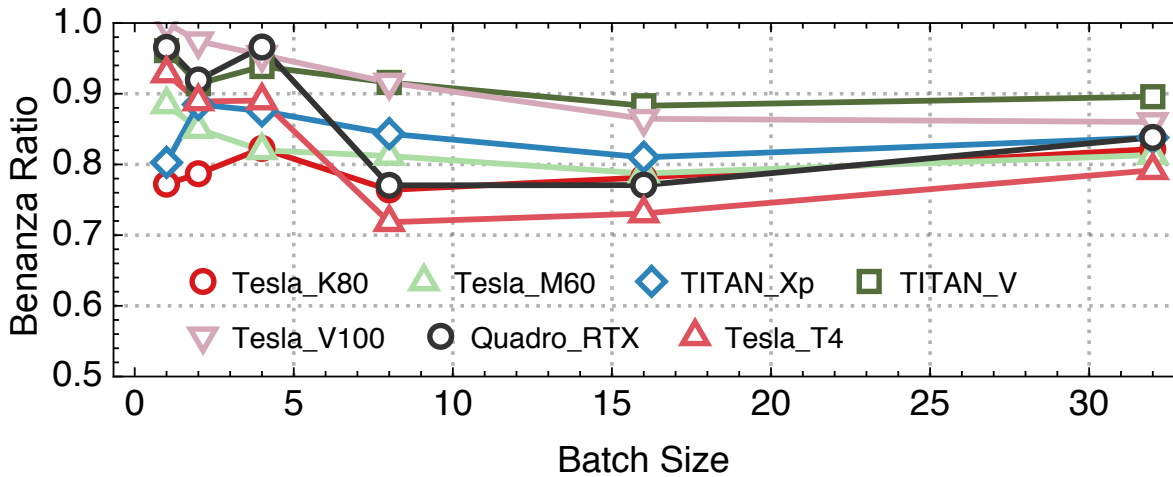


Figure 5.9: The $BR_{\text{sequential}}$ of ResNet50-v1.

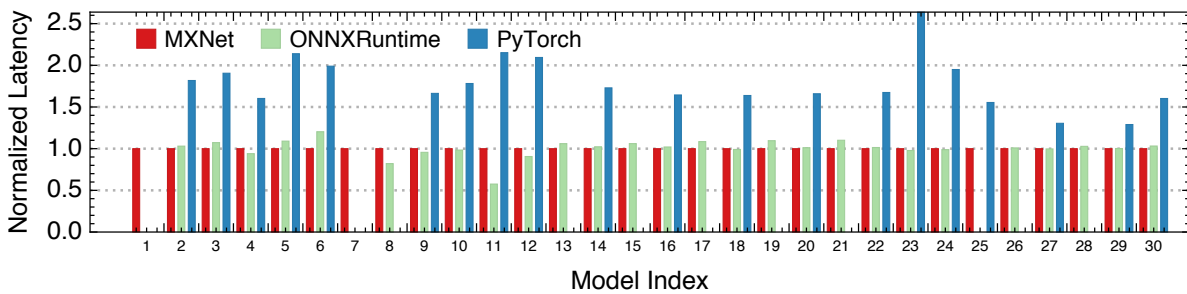


Figure 5.10: The measured latency of all ONNX models with MXNet, ONNX Runtime, and PyTorch backends (normalized to MXNet latency) using batch size 1 on Tesla_V100.

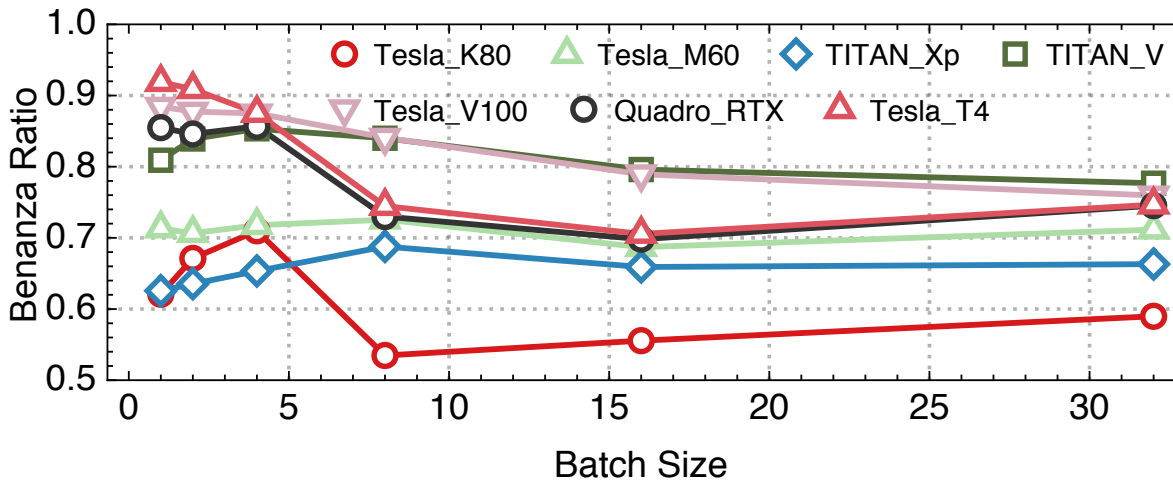


Figure 5.11: The geometric mean of the $BR_{\text{sequential}}$ of all models.

TFLOPS in Table 5.3). ResNet50-v1 has a higher latency on Quadro_RTX when compared to Tesla_V100, since Quadro_RTX has an on-chip (global) memory bandwidth of 624 GB/s whereas Tesla_V100 has an on-chip memory bandwidth of 950 GB/s.

Figure 5.9 shows the $BR_{\text{sequential}}$ of ResNet50-v1 across batch sizes and systems. The results suggest that ResNet50-v1’s optimization opportunities are system and batch size dependent. Both Tesla_V100 and TITAN_V are highly optimized to run ResNet50-v1 across batch sizes, since their BR is high — ranging from 0.86 to 1.0. The BR for Tesla_T4 and Quaro_RTX is high for batch sizes 1 to 4 but drops beyond that. ResNet50-v1 is less optimized on the other systems and has a low BR.

The geometric mean of the $BR_{\text{sequential}}$ for all the models across systems and batch sizes is shown in Figure 5.11. Both Tesla_V100 and TITAN_V still have a high BR (0.76 – 0.88). A drop was still observed for Tesla_T4 and Quaro_RTX at batch size 4. Tesla_M60 and TITAN_Xp have a BR between 0.63 and 0.72. The oldest GPU generation, Tesla_K80, has the lowest BR and is the least optimized.

Overall, the current software stack (latest MXNet, cuDNN, and CUDA libraries used in the evaluation) is more optimized for the recent GPU generations (Turing and Volta) using smaller batch sizes. Compared to Volta, the software stack is less optimized for Turing. This is possibly because Turing is newly released, and we expect optimizations that target Turing to increase. Moreover, the low BR for the older GPUs suggest that vendors prioritize optimizations for newer GPU generations over older ones.

5.3.2 cuDNN Convolution Heuristics (*Question 3*)

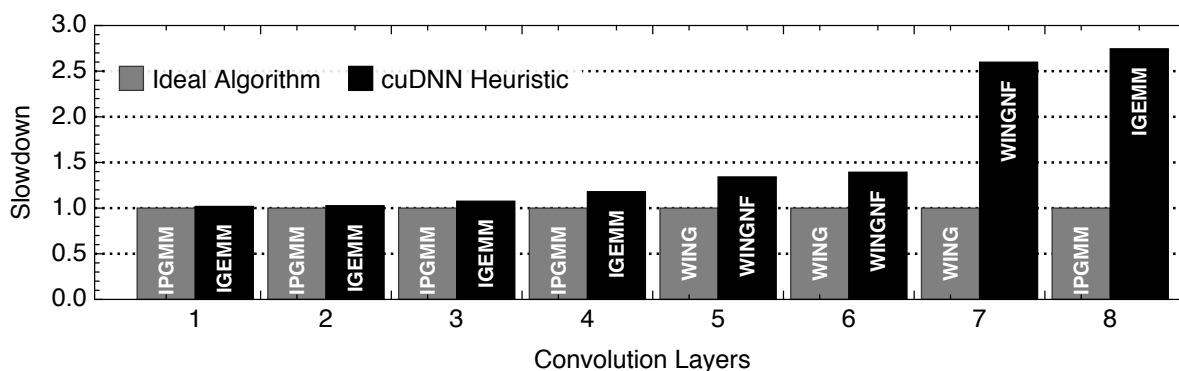


Figure 5.12: The cuDNN heuristic selects 8 non-optimal convolution layer algorithms for ResNet50_v1 using batch size 32 on Tesla_V100. Up to $2.75\times$ speedup can be achieved if selection was ideal.

Using the Benanza Analyzer, we observed that heuristics employed by cuDNN (and sub-

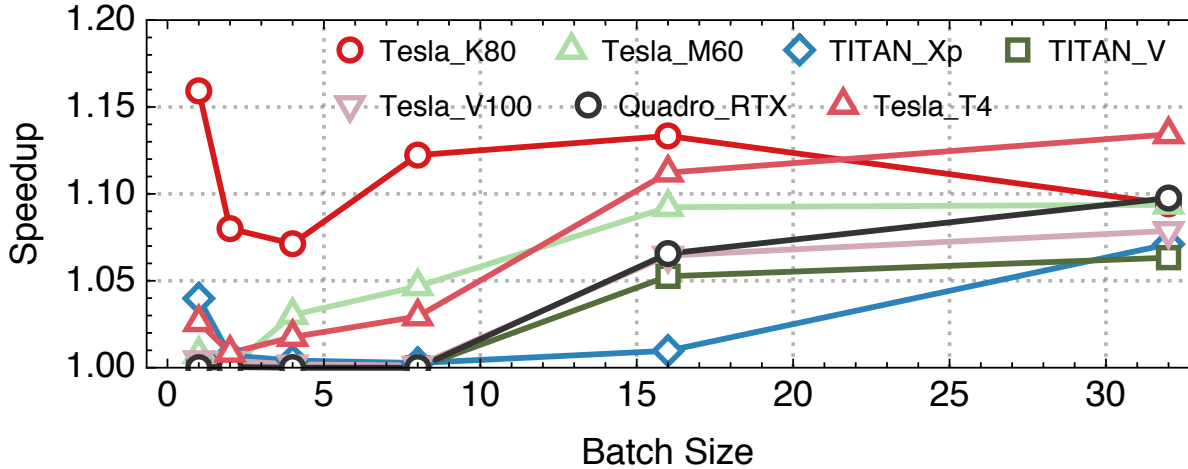


Figure 5.13: The latency speedup for ResNet50-v1 if layer fusion was performed.

sequently the frameworks) are not always optimal. For example, Figure 5.12 shows the convolution layer latencies using the algorithms informed by cuDNN heuristics (labeled as *cuDNN Heuristic*) normalized to using the optimal algorithm (labeled as *Ideal Algorithm*) for ResNet50-v1 using batch size 32 on Tesla.V100. The algorithm choices are listed in Section 5.2.2. Figure 5.13 shows the latency speedup for ResNet50-v1 across batch sizes and systems by using the optimal convolution algorithm for all convolution layers. Figure 5.14 shows the geometric mean of the latency speedup for all models by using the optimal algorithms. At batch size 32, the speedup ranges between $1.14\times$ and $1.32\times$ across GPUs. Both the latest and older GPU architectures can benefit from better algorithm heuristics.

5.3.3 Inefficiencies in Frameworks (**Question 4**)

We used Benanza to identify the inefficiencies in MXNet and PyTorch. We then implemented the optimizations informed by Benanza and show the latency speedup after the framework modifications.

MXNet ONNX Model Loader

We observed through the Analyzer that there are layers in the model execution profile where the cuDNN API arguments deviate from what is expected. An inspection of the Analyzer’s parsed Nsight profile pointed to an `image_2d_pad_constant_kernel` GPU kernel function being invoked before every convolutional layer. Non-zero padding leads to the observed deviation between the expected and actual cuDNN API calls. We inspected the MXNet source code and found that padding layers are inserted during the loading of ONNX

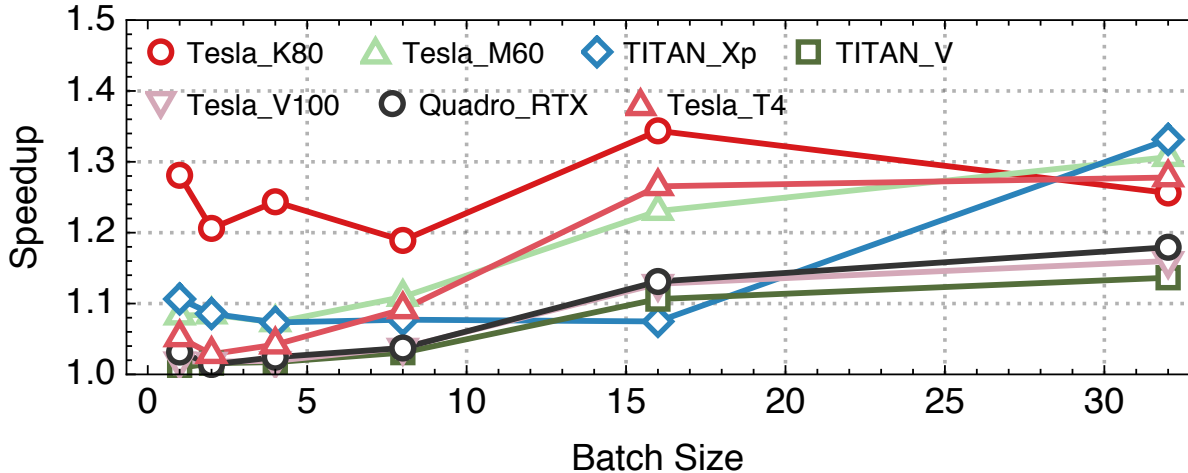


Figure 5.14: The geometric mean of the latency speedup for all models by using the optimal convolution algorithm.

models in MXNet. ONNX supports specifying asymmetric padding³ as attributes in convolution layers [79], whereas MXNet does not. Therefore, MXNet must insert padding layers before convolution layers where asymmetric padding is used when loading ONNX models. However, the MXNet ONNX model loader adds padding layers before every convolution layer (regardless of the use of asymmetric padding). A non-intrusive optimization is to only insert padding layers if asymmetric padding is used. With this simple one-line optimization, we observed up to $1.15\times$ latency speedup for `ResNet50-v1` (shown in Figure 5.15).

PyTorch cuDNN Wrapper

Using Benanza we observed that there were excessive calls to `cudaStreamWaitEvent` between cuDNN API calls. Using the Nisight’s backtrace information from the model execution profile, we identified the PyTorch source file that introduces these synchronizations. Upon further study of the source code, we found that all cuDNN functions are invoked by a cuDNN wrapper in PyTorch. The wrapper manages a pool of cuDNN handles and is designed to enable invoking cuDNN functions from different CPU threads. cuDNN functions managed by the same handle are synchronized and executed sequentially. In the current PyTorch (v1.3), however, only a single handle is used for inference, which forces synchronization before each cuDNN function call. The synchronizations cause $100\mu s$ stalls on average between cuDNN functions, thus the latency saved through this optimization is a function of the number of layers in a model. We modified PyTorch to elide the cuDNN wrapper and only synchronize

³The numbers of zero values to add before or after each spacial dimension are not guaranteed to be the same.

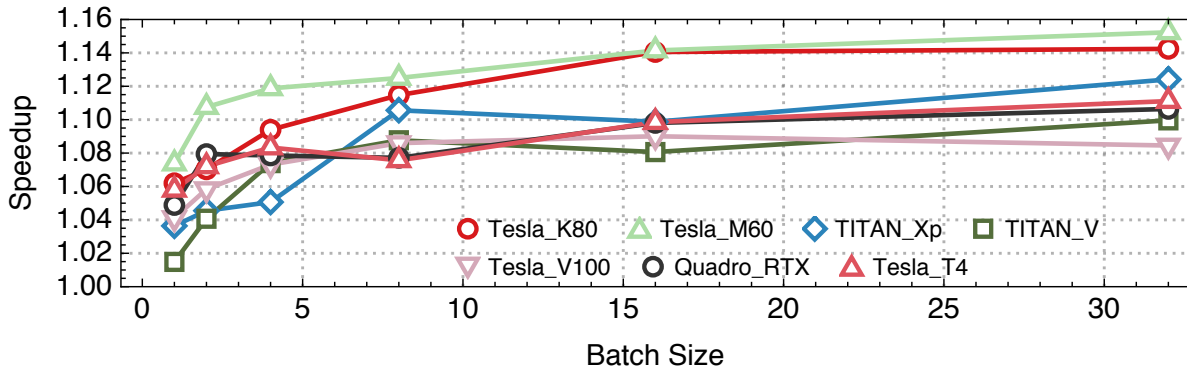


Figure 5.15: The speedup achieved for ResNet50_v1 by applying the MXNet optimization described in Section 5.3.3 across batch sizes and systems.

before and after performing inference. Figure 5.16 shows the speedup achieved by this optimization for batch size 1. MobileNet-v2 (ID=12) achieves a $2.3\times$ speedup, since it has low latency and a large number of layers.

5.3.4 Layer Fusion (*Question 5*)

We used Benanza to evaluate the potential benefits of layer fusion. Figure 5.17 shows the latency speedup from layer fusion for ResNet50-v1 across the systems and batch sizes. ResNet50-v1 has the layer sequence pattern Conv→Bias→BatchNorm→Activation. Benanza reports that the Conv→Bias sequence can be fused for better latency and performs the fusion analysis (Section 5.2.4). In all, 64 (18%) layers were fused and up to $1.09\times$ speedup was achieved over the measured latency across systems for ResNet150-v1. By inspecting the model execution profile, we found no indication that MXNet, ONNX Runtime, or PyTorch perform layer fusion using the cuDNN fused API.

5.3.5 Tensor Cores (*Question 6*)

We used Benanza to evaluate the potential benefits of using float16 and Tensor Cores available on recent GPU architectures. While the cuDNN Tensor Core API supports both NHWC and NCHW layouts, NVIDIA recommends the use of NHWC. We use Benanza to generate benchmarks targeting both the NHWC and NCHW and evaluated the “lower-bound” latency speedup, as shown in Figures 5.19 and 5.18 respectively. As expected, using the NHWC achieves higher speedup. Internally, the cuDNN API implements NCHW convolutions in terms of NHWC with an implicit transposition. As compute dominates (i.e. larger batch sizes), the relative overhead of the transposition becomes small; hence, NCHW and NHWC have similar

performance for larger batch sizes. Figure 5.20 shows the end-to-end latency speedup by using Tensor Cores(NHWC). TITAN_V achieves significant speedup (up to 1.72×). We can see that Tesla_T4 benefits most from Tensor Cores for smaller batch sizes (i.e. might be best used for low-latency inference).

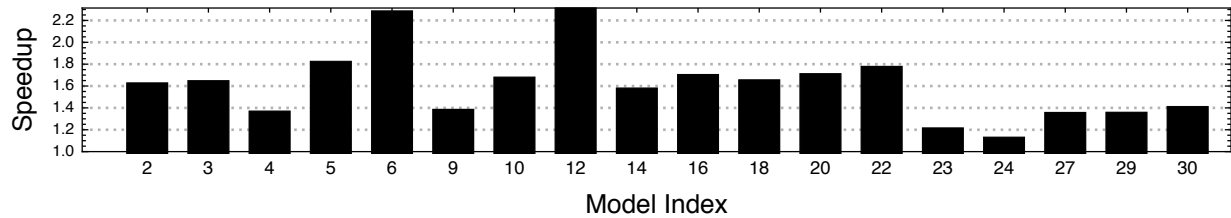


Figure 5.16: The speedup achieved by removing unnecessary cuDNN API synchronizations in PyTorch on Tesla_V100 using batch size 1.

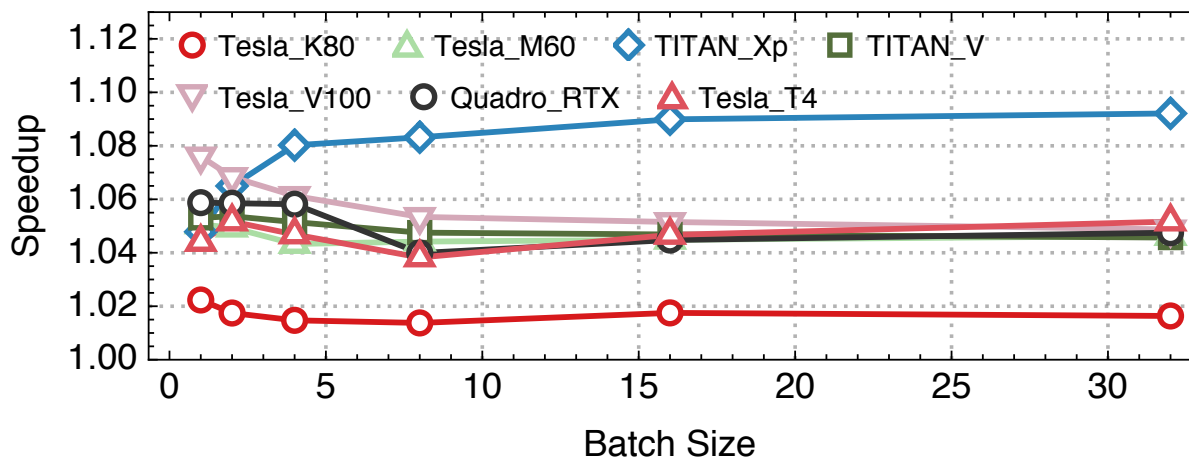


Figure 5.17: The latency speedup for ResNet50-v1 if layer fusion was performed.

5.3.6 Parallel Execution, Algorithm Selection, Layer Fusion, and Tensor Cores

(Question 1,2,3,5,6)

Benanza can be used to perform the above analysis jointly. To demonstrate this, we analyzed the latency speedup when using parallel execution of data-independent layers, optimal algorithm selection, layer fusion, and Tensor Cores (NHWC). Figure 5.21 shows the latency speedup for ResNet50-v1 across batch sizes and systems. Up to a 1.95× and 1.8× speedup can be achieved by TITAN_V and Tesla_V100 respectively. We can surmise, from the previous analysis, that most of the profit for TITAN_V is attributed to its use of Tensor Cores. Quadro_RTX and Tesla_T4 achieve marginal speedup over the Tensor Core results.

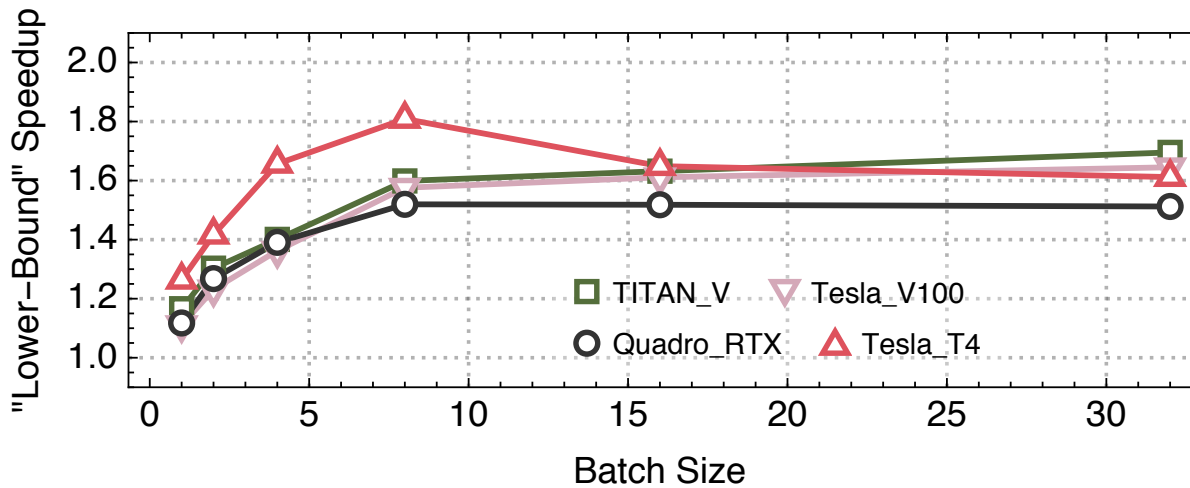


Figure 5.18: The “lower-bound” latency speedup if Tensor Cores (NCHW) were used for ResNet50-v1.

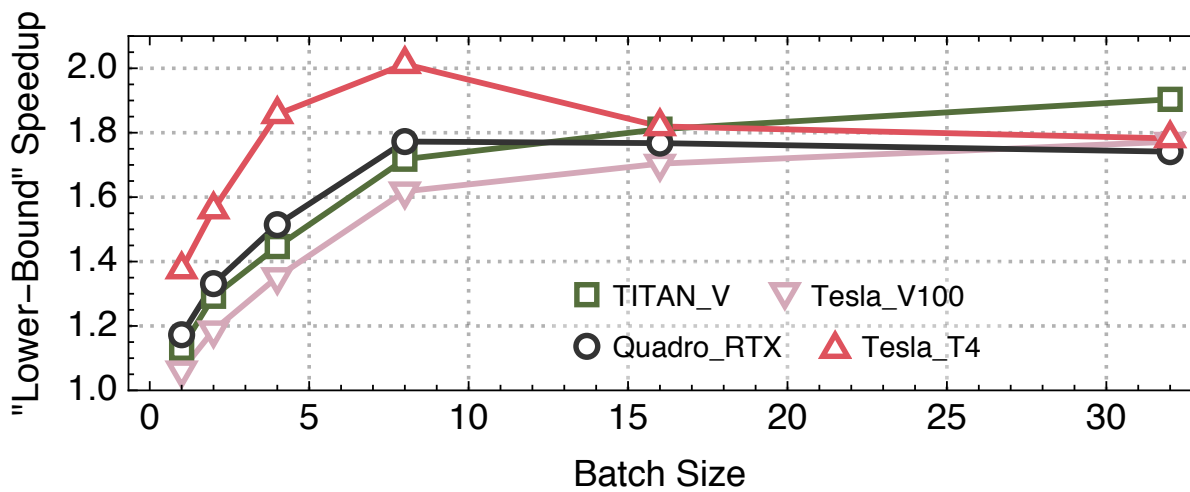


Figure 5.19: The “lower-bound” latency speedup for ResNet50-v1 if Tensor Cores (NHWC) were used.

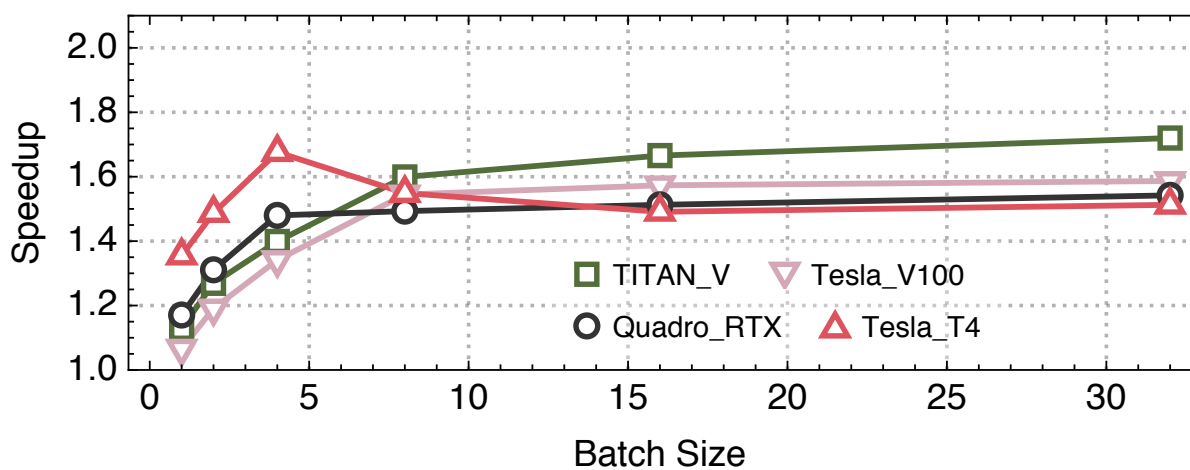


Figure 5.20: The latency speedup for ResNet50-v1 if Tensor Cores (NHWC) were used.

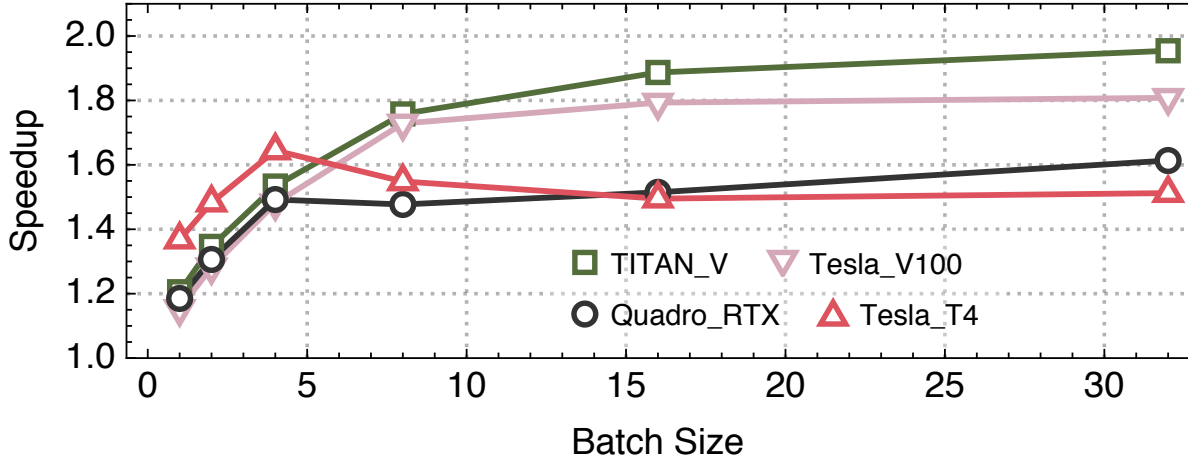


Figure 5.21: The latency speedup for ResNet50-v1 if parallel execution, optimal algorithm selections, layer fusion, and Tensor Cores (NHWC) were used.

5.4 RELATED WORK

DL Benchmarking: There has been no shortage of work on developing benchmarks to characterize DL models. These DL benchmarks either take a model as a black-box and measure the user-observable latency and throughput (end-to-end benchmarks) or delve deeper into models to characterize the layer or kernel performance (micro-benchmarks). The end-to-end benchmarks [3,4,6] provide a corpus of models that are deemed to be of value to characterize for industry and research. Micro-benchmarks [4,5,22,80] distill DL models into their layers or kernels, and are hand-curated. Micro-benchmarking enables easy measurements of layers within popular DL models and integrates easily with profiling tools. In [81], the author present a design that enables benchmarking DL models across the abstraction levels of inference pipeline and introduce a hierarchical profiling methodology (enabling framework-, model-, and hardware-profiling). In [7], the authors propose a benchmark suite to enable fair comparison of DL techniques at different levels of granularity. At the operator level, [7] takes ONNX models and generates micro-benchmarks that target the framework’s Python API to measure the latency of each operator. Benanza also takes ONNX models as input, but generates lower-level cuDNN and cuBLAS micro-benchmarks to compute the “lower-bound” latency of the model, and perform analysis. To my knowledge, there is no previous work which generates micro-benchmarks from model layers and couples it with an analysis workflow to inform optimizations.

Performance Advising: [41] introduces the roofline model to analyze inherent limitations of an application running on a system and indicate priority of optimizations. There is past work on using profiling to inform users of possible compiler-level optimizations [82] or proper

usage of APIs [83,84]. Profilers and IDEs such as NVIDIA’s Nvprof [8], Intel’s VTune [10], Oracle’s Solaris Studio [85], Microsoft’s Roslyn [86], and IBM’s XL [87], provide low-level profiling reports and some suggestions on how to address bottlenecks. To my knowledge, there has been no work on combining the microbenchmarking and profiling results to inform optimizations in the DL domain.

5.5 CONCLUSION

This chapter presents Benanza, a sustainable and extensible DL benchmarking and analysis design that automatically generates layer-wise benchmarks for DL models to compute the “lower-bound” latency and inform optimizations on GPUs. We use Benanza to evaluate a set of 30 models using different frameworks on 7 GPUs, and pinpointed the optimizations in parallel layer execution, cuDNN algorithm selection, framework inefficiency, layer fusion, and Tensor Core usage. The results show that Benanza fills a significant gap within the characterization/optimization cycle and can boost the productivity of DL model, framework, and library developers.

CHAPTER 6: MLMODELSCOPE: THE DESIGN AND IMPLEMENTATION OF A SCALABLE DL BENCHMARKING PLATFORM

In this chapter, we first identify 10 design features which are desirable within a DL benchmarking platform. These features include: performing the evaluation in a consistent, reproducible, and scalable manner, being framework and hardware agnostic, supporting real-world benchmarking workloads, providing in-depth model execution inspection across the HW/SW stack levels, etc. We then present MLModelScope, a DL benchmarking platform that realizes 10 design objectives. MLModelScope proposes a specification to define DL model evaluations and techniques to provision the evaluation workflow using the user-specified HW/SW stack. MLModelScope defines abstractions for frameworks and supports the board range of DL models and evaluation scenarios.

The emergence of Deep Learning (DL) as a popular application domain has led to many innovations. Every day, diverse DL models as well as hardware/software (HW/SW) solutions, are proposed — be it algorithms, frameworks, libraries, or hardware. DL innovations are introduced at such a rapid pace [1] that being able to evaluate and compare these innovations quickly is critical for their adoption. As a result, there have been concerted community efforts in developing DL benchmark suites [3, 4] where common models are selected and curated as benchmarks.

DL benchmark suites require significant effort to develop and maintain and thus have limited coverage of models (usually a few models are chosen to represent a DL task). Within these benchmark suites, model benchmarks are often developed independently as a set of ad-hoc scripts. To consistently evaluate two models requires one to use the same evaluation code and HW/SW environment. Since the model benchmarks are ad-hoc scripts, a fair comparison requires a non-trivial amount of effort. Furthermore, DL benchmarking often requires evaluating models across different combinations of HW/SW stacks. As HW/SW stacks are being proposed, there is an urgent need for a DL benchmarking platform that consistently evaluates and compares different DL models across HW/SW stacks, while coping with the fast-paced and diverse landscape of DL.

DL model evaluation is a complex process where the model and HW/SW stack must work in unison, and the benefit of a DL innovation is dependent on this interplay. Currently, there is no standard to specify or provision DL evaluations, and reproducibility is a significant “pain-point” within the DL community [88–90]. Thus, the benchmarking platform design must guarantee a **Feature 1 reproducible evaluation** along with **Feature 2 consistent evaluation**.

Aside from **Feature 1-2**, the design should: be **Feature 3 frameworks and hardware agnostic** to support model evaluation using diverse HW/SW stacks; be capable of performing **Feature 4 scalable evaluation** across systems to cope with the large number of evaluations due to the many model/HW/SW combinations; support different **Feature 7 benchmarking scenarios** which mimic the real-world workload exhibited in online, offline, and interactive applications; have a **Feature 8 benchmarking analysis and reporting** workflow to analyze benchmarking results across runs and generate summary reports; enable **Feature 9 model execution inspection** to identify bottlenecks within a model-, framework-, and system-level components. Other features such as **Feature 5 artifact versioning**, **Feature 6 efficient evaluation workflow**, and **Feature 10 different user interfaces** are also desirable to increase the design’s usability.

We propose *MLModelScope* [36], a scalable DL benchmarking platform design that realizes the above 10 objectives and facilitates benchmarking, comparison, and understanding of DL model execution. *MLModelScope* achieves the design objectives by proposing a specification to define DL model evaluations; introducing techniques to consume the specification and provisioning the evaluation workflow with the specified HW/SW stack; using a distributed scheme to manage, schedule, and handle model evaluation requests; supporting pluggable workload generators; defining common abstraction API across frameworks; providing across-stack tracing capability that allows users to inspect model execution at different HW/SW abstraction levels; defining an automated evaluation analysis workflow for analyzing and reporting evaluation results; and, finally, exposing the capabilities through a web and command-line interface.

We implement *MLModelScope* and integrate it with Caffe, Caffe2, CNTK, MXNet, PyTorch, TensorFlow, TFLite, and TensorRT frameworks. *MLModelScope* runs on ARM, PowerPC, and x86 and supports CPU, GPU, and FPGA execution. We bootstrap *MLModelScope* with over 300 models covering different DL tasks such as image classification, object detection, semantic segmentation, etc. *MLModelScope* is open-source, extensible, and customizable.

We showcase *MLModelScope*’s benchmarking, inspection, and analysis capabilities using several case studies. We use *MLModelScope* to evaluate 37 DL models and compare their performance on 4 systems under different benchmarking scenarios. We perform comparisons to understand the correlation between a model accuracy, size, achieved latency, and maximum throughput. We then use *MLModelScope*’s tracing capability to identify the bottlenecks of the evaluation and use its “zoom-in” feature to inspect the model execution at different HW/SW levels. We demonstrate how, using the analysis workflow, one can easily digest the evaluation results produced by *MLModelScope* to understand model-, framework-,

and system- bottlenecks. To the authors’ knowledge, we are the first to describe the design and implementation of a scalable DL benchmarking platform.

6.1 DESIGN OBJECTIVES

In this section, we detail 10 objectives for a DL benchmarking platform design to cope with the fast-evolving DL landscape. These objectives informed MLModelScope’s design choices.

- Reproducible Evaluation (**Feature 1**) — Model evaluation is a complex process where the model, dataset, evaluation method, and HW/SW stack must work in unison to maintain the accuracy and performance claims. Currently, model authors distribute their models and code (usually ad-hoc scripts) by publishing them to public repositories such as GitHub. Due to the lack of standard specification, model authors may under-specify or omit key aspects of model evaluation. As a consequence, reproducibility is a “pain-point” within the DL community [88]. Thus, all aspects of evaluation must be specified and provisioned by the platform design to guarantee reproducible evaluation.
- Consistent Evaluation (**Feature 2**) — The current practice of publishing models and code also poses challenges to consistent evaluation. The ad-hoc scripts usually have a tight coupling between model execution and the underlying HW/SW — making it difficult to quantify or isolate the benefits of an individual component (be it model, framework, or other SW/HW components). A fair apple-to-apple comparison between model executions requires a consistent evaluation methodology rather than running ad-hoc scripts for each. Thus the design should have a well-defined benchmarking specification for all models and maximize the common code base that drives model evaluations.
- Framework/Hardware Agnostic (**Feature 3**) — There are many DL frameworks (e.g. TensorFlow, MXNet) and hardware (e.g. CPU, GPU, FPGA) and each has its own use scenarios, features, and performance characteristics. To have broad support, the design must be framework and hardware agnostic. Furthermore, the design must be able to fully function without framework modifications.
- Scalable Evaluation (**Feature 4**) — DL innovations, such as models, frameworks, libraries, compilers, and hardware accelerators are introduced at a rapid pace [1, 2]. Being able to quickly evaluate and compare the benefits of DL innovations is critical for their adoption. Thus the ability to perform DL evaluations with different model/HW/SW setups in parallel and have a centralized management of the benchmarking results is highly desired. For example, choosing the best hardware out of N candidates for a model is ideally performed in parallel and the results should be automatically gathered for comparison.

- Artifact Versioning (**Feature 5**) — DL frameworks are continuously updated by the DL community, e.g. the recent versions TensorFlow at the time of writing are v1.15 and v2.0. There are many unofficial variants of models, frameworks, and datasets as researchers might update or modify them to suite their respective needs. To enable management and comparison of model evaluations using different DL artifacts (models, frameworks, and datasets), the artifacts used for evaluation within a benchmarking platform should be versioned.
- Efficient Evaluation Workflow (**Feature 6**) — Before model inference can be performed, the input data has to be loaded and transformed into a form that the model expects (pre-processing stage). After the model prediction, the post-processing stage transforms the model’s output(s) to a form that can be used to compute metrics. The data loading and pre-/post-processing can take a non-negligible amount of time, and become a limiting factor for quick evaluations [91]. Thus the design should handle and process data efficiently in the evaluation workflow.
- Benchmarking Scenarios (**Feature 7**) — DL benchmarking is performed under specific scenarios. These scenarios mimic the usage of DL in online, offline, or interactive applications on mobile, edge, or cloud systems. The design should support common inference scenarios and be flexible to support custom or emerging workloads as well.
- Benchmarking Analysis and Reporting (**Feature 8**) — Benchmarking produces raw data which needs to be correlated and analyzed to produce human-readable results. An automated mechanism to summarize and visualize these results within a benchmarking platform can help users quickly understand and compare the results. Therefore, the design should have a benchmarking result analysis and reporting workflow.
- Model Execution Inspection (**Feature 9**) — The complexity of DL model evaluation makes performance debugging challenging as each level within the HW/SW abstraction hierarchy can be a suspect when things go awry. Current model execution inspection methods rely on the use of a concoction of profiling tools (e.g. Nvidia’s Nsight System or Intel’s Vtune). Each profiling tool captures a specific aspect of the HW/SW stack and researchers manually correlate the results to get an across-stack view of the model execution profile. To ease inspecting model execution bottlenecks, the benchmarking platform design should provide a coherent, tracing capability across all levels of HW/SW stack.
- Different User Interfaces (**Feature 10**) — While the command-line is the most common interface in the current benchmarking suites, having other UIs, such as web UI, to accommodate other use cases can greatly boost productivity. While a command-line interface is often used in scripts to quickly perform combinational evaluations across models, frameworks, and systems, a web UI, on the other hand, can serve as a “push-button” solution

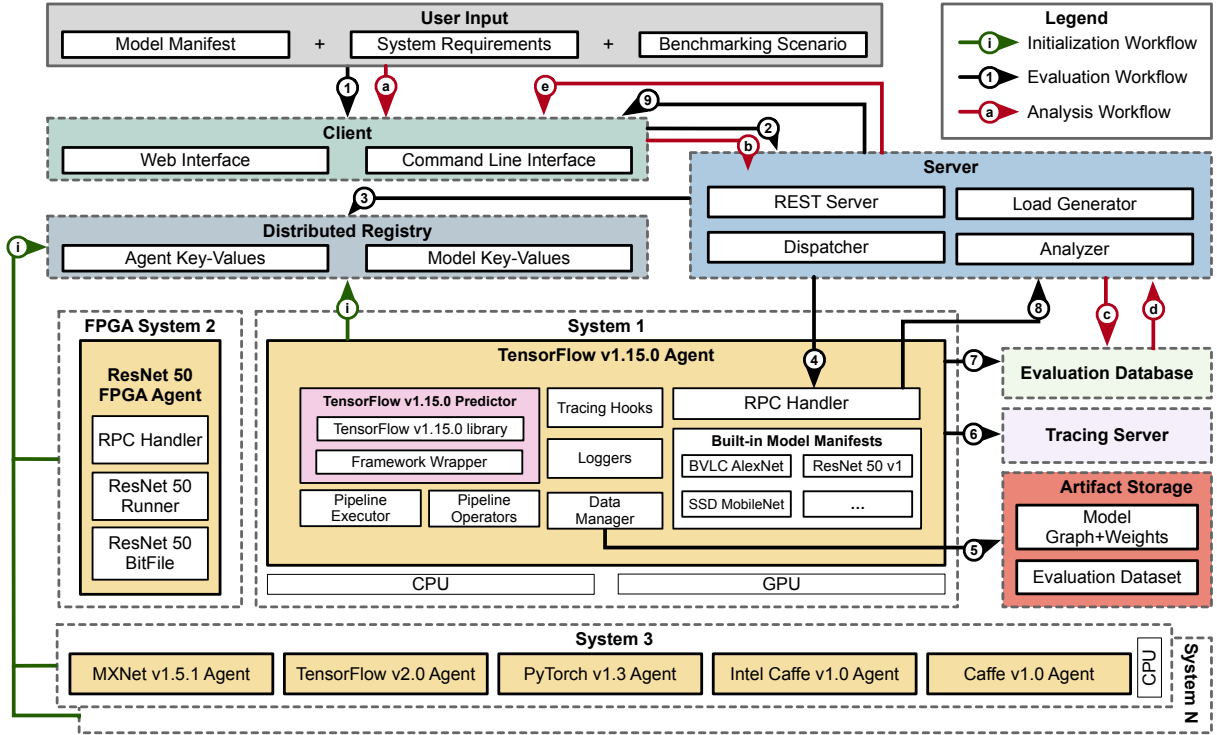


Figure 6.1: The MLModelScope design and workflows.

to benchmarking and provides an intuitive flow for specifying, managing evaluations, and visualizing benchmarking results. Thus the design should provide UIs for different use cases.

6.2 MLMODELSCOPE DESIGN AND IMPLEMENTATION

We propose MLModelScope, a DL benchmarking platform design that achieves the objectives **Feature 1-10** set out in Section 6.1. To achieve **Feature 4** scalable evaluation, we design MLModelScope as a distributed platform. To enable **Feature 7** real-world benchmarking scenarios, MLModelScope deploys models to be either evaluated using a cloud (as in model serving platforms) or edge (as in local model inference) scenario. To keep up with the fast pace of DL, MLModelScope is built as a set of extensible and customizable modular components. We briefly describe each component here and will delve into how they are used later in this section. Figure 6.1 shows the high level components which include:

- **User Inputs** are the required inputs for model evaluation and include: a model manifest (a specification describing how to evaluate a model), a framework manifest (a specification describing the software stack to use), the system requirements (e.g., an X86 system with 32GB of RAM and an NVIDIA V100 GPU), and the benchmarking scenario to employ.

- **Client** is either the web UI or command-line interface which users use to supply their inputs and initiate the model evaluation by sending a REST request to the MLModelScope server.
- **Server** acts on the client requests and performs REST API handling, dispatching the model evaluation tasks to MLModelScope agents, generating benchmark workloads based on benchmarking scenarios, and analyzing the evaluation results.
- **Agents** run on different systems of interest and perform model evaluation based on requests sent by the MLModelScope server. An agent can be run within a container or as a local process and has logic for downloading model assets, performing input pre-processing, using the framework predictor for inference, and performing post-processing. Aside from the framework predictor, all code in an agent is common across frameworks.
- **Framework Predictor** is a wrapper around a framework and provides a consistent interface across different DL frameworks. The wrapper is designed as a thin abstraction layer so that all DL frameworks can be easily integrated into MLModelScope by exposing a limited number of common APIs.
- **Middleware** consists of a set of support services for MLModelScope including: a distributed registry (a key-value store containing entries of running agents and available models), an evaluation database (a database containing evaluation results), a tracing server (a server to publish profile events captured during an evaluation), and an artifact storage server (a data store repository containing model assets and datasets).

Figure 6.1 also shows MLModelScope’s three main workflows: ① initialization, ①-⑨ evaluation, and a-e analysis. The initialization workflow is one where all agents self-register by populating the registry with their software stack, system information, and available models for evaluation. The evaluation workflow works as follows: ① a user inputs the desired model, software and hardware requirements, and benchmarking scenario through a client interface. The ② server then accepts the user request, resolves which agents are capable of handling the request by ③ querying the distributed registry, and then ④ dispatches the request to one or more of the resolved agents. The agent then ⑤ downloads the required evaluation assets from the artifact storage, performs the evaluation, and ⑥-⑦ publishes the evaluation results to the evaluation database and tracing server. A summary of the results is ⑧ sent to the server which ⑨ forwards it to the client. Finally, the analysis workflow allows a user to perform a more fine-grained and in-depth analysis of results across evaluation runs. The MLModelScope server handles this workflow by a-d querying the evaluation database and performing analysis on the results, and e generating a detailed analysis report for the user. This section describes the MLModelScope components and workflows in detail.

6.2.1 User Input

All aspects of DL evaluation — model, software stack, system, and benchmarking scenario — must be specified to MLModelScope for it to enforce **Feature 1** reproducible and **Feature 2** consistent evaluation. To achieve this, MLModelScope defines a benchmarking specification covering the 4 aspects of evaluation. A model in MLModelScope is specified using a *model manifest*, and a software stack is specified using a *framework manifest*. The manifests are textual specification and the system and benchmarking scenario are user-specified options when the user initiates an evaluation. The benchmarking specification is not tied to a certain framework or hardware, thus enabling **Feature 3**. As the model, software stack, system, and benchmarking scenario specification are decoupled, one can easily evaluate the different combinations, enabling **Feature 4**. For example, a user can use the same `MLPerf_ResNet50_v1.5` model manifest (shown in Listing 6.1) to initiate evaluations across different TensorFlow software stacks, systems, and benchmarking scenarios. To bootstrap the model evaluation process, MLModelScope provides built-in model manifests which are embedded in MLModelScope agents (Section 6.2.4). For these built-in models, a user can specify the model and framework’s name and version in place of the manifest for ease of use. MLModelScope also provides ready-made Docker containers to be used in the framework manifests. These containers are hosted on Docker hub.

Model Manifest

The model manifest is a text file that specifies information such as the model assets (graph and weights), the pre- and post-processing steps, and other metadata used for evaluation management. An example model manifest of `ResNet50 v1.5` from MLPerf is shown in Listing 6.1. The manifest describes the model name (Lines 1-2), framework name and version constraint (Lines 4-6), model inputs and pre-processing steps (Lines 7-21), model outputs and post-processing steps (Lines 22-28), custom pre- and post-processing functions (Lines 29-30), model assets (Lines 31-34), and other metadata attributes (Lines 35-38).

Framework Constraints Models are dependent on the framework and possibly the framework version. Users can specify the framework constraints required by a model. For example, an ONNX model may work across all frameworks and therefore has no constraint, but other models may only work for TensorFlow versions greater than 1.2.0 but less than 2 (e.g. Lines 4–6 in Listing 6.1). This allows MLModelScope to support models which use specific or custom frameworks.

Pre- and Post-Processing To perform pre- and post-processing for model evaluation,

```

1 name: MLPerf_ResNet50_v1.5 # model name
2 version: 1.0.0 # semantic version of the model
3 description: ...
4 framework: # framework information
5   name: TensorFlow
6   version: '>=1.12.0 <2.0' # framework ver constraint
7 inputs: # model inputs
8   - type: image # first input modality
9     layer_name: 'input_tensor'
10    element_type: float32
11    steps: # pre-processing steps
12      - decode:
13        data_layout: NHWC
14        color_mode: RGB
15      - resize:
16        dimensions: [3, 224, 224]
17        method: bilinear
18        keep_aspect_ratio: true
19      - normalize:
20        mean: [123.68, 116.78, 103.94]
21        rescale: 1.0
22 outputs: # model outputs
23   - type: probability # first output modality
24     layer_name: prob
25     element_type: float32
26     steps: # post-processing steps
27       - argsort:
28         labels_url: https://.../synset.txt
29 preprocess: [[code]]
30 postprocess: [[code]]
31 model: # model sources
32   base_url: https://zenodo.org/record/2535873/files/
33   graph_path: resnet50_v1.pb
34   checksum: 7b94a2da05d...23a46bc08886
35 attributes: # extra model attributes
36   training_dataset: # dataset used for training
37     - name: ImageNet
38     - version: 1.0.0

```

Listing 6.1: MLPerf_ResNet50_v1.5 model manifest.

arbitrary Python functions can be placed within the model manifest (Lines 29 and 30 in Listing 6.1). The pre- and post-processing functions are Python functions which have the signature `def fun(env, data)`. The `env` contains metadata of the user input and `data` is a `PyObject` representation of the user request for pre-processing or the model’s output for post-processing. Internally, `MLModelScope` executes the functions within a Python subinterpreter [92] and passes the data arguments by reference. The pre- and post-processing functions are general; i.e. the functions may import external Python modules or download and invoke external scripts. By allowing arbitrary processing functions, `MLModelScope` works with existing processing codes and is capable of supporting arbitrary input/output modalities.

Built-in Pre- and Post-Processing An alternative way of specifying pre- and post-

processing is by defining them as a series of built-in pre- and post-processing pipeline steps (i.e. *pipeline operators*) within the model manifest. For example, our MLModelScope implementation provides common pre-processing image operations (e.g. image decoding, resizing, and normalization) and post-processing operations (e.g. ArgSort, intersection over union, etc.) which are widely used within vision models. Users can use built-in operators to define the pre- and post-processing pipelines within the manifest without writing code. Users define a pipeline by listing the operations within the manifest code (e.g. Lines 7–21 in Listing 6.1 for pre-processing). The pre- and post-processing steps are executed in the order they are specified in the model manifest.

Model Assets The data required by the model are specified in the model manifest file; i.e. the `graph` (the `graph_path`) and weights (the `weights_path`) fields. The model assets can reside within MLModelScope’s artifact repository, on the web, or the local file system of the MLModelScope agent. If the model assets are remote, then they are downloaded on demand and cached on the local file system. For example, the TensorFlow ResNet50 v1.5 model assets in Listing 6.1 are stored on the Zenodo [93] website (Lines 31-34) and are downloaded prior to evaluation.

Framework Manifest & System Requirements

The framework manifest is a text file that specifies the software stack for model evaluation; an example framework manifest is shown in Listing 6.2. To maintain the software stack, and guarantee isolation, the user specifies the docker containers using the `containers` field. Multiple containers can be specified to accommodate different systems (e.g. CPU or GPUs). At the MLModelScope initialization phase (i), MLModelScope agents (described in Section 6.2.4) register themselves by publishing their HW/SW stack information into the distributed registry (described in Section 6.2.5). The MLModelScope server uses this information during the agent resolution process. The server finds agents satisfying the user’s hardware specification and model/framework requirements. Evaluations are then run on one of (or, at the user request, all of) the agents. If the user omits the framework manifest in the user input, the server uses the model and system information as constraints.

Benchmarking Scenario

MLModelScope provides a set of built-in benchmarking scenarios. The benchmarking scenarios include batched inference and online inference with a configurable distribution of time of request (e.g. Poisson distribution of requests). The MLModelScope server generates

```

1 name: TensorFlow # framework name
2 version: 1.15.0 # semantic version of the framework
3 description: ...
4 containers: # containers
5   amd64:
6     cpu: carml/tensorflow:1-15-0_amd64-cpu
7     gpu: carml/tensorflow:1-15-0_amd64-gpu
8   ppc64le:
9     cpu: carml/tensorflow:1-15-0_ppc64le-cpu
10    gpu: carml/tensorflow:1-15-0_ppc64le-gpu

```

Listing 6.2: An example TensorFlow framework manifest.

```

1 // Opens a predictor.
2 ModelHandle ModelLoad(OpenRequest);
3 // Close an open predictor.
4 Error ModelUnload(ModelHandle);
5 // Perform model inference on user data.
6 PredictResponse Predict(ModelHandle, PredictRequest, PredictOptions);

```

Listing 6.3: The predictor interface consists of 3 API functions.

an inference request load based on the benchmarking scenario option and sends it to the selected agent(s) to measure the corresponding benchmarking metrics of the model (detailed in Section 6.2.3).

6.2.2 MLModelScope Client

A user initiates a model **①** evacuation or **ⓐ** analysis through the MLModelScope *client*. To enable **Feature 10**, the client can be either a website or a command-line tool that users interact with. The client communicates with the MLModelScope server through REST API and sends user evaluation requests. The web user interface allows users to specify a model evaluation through simple clicks and is designed to help users who do not have much DL experience. For example, for users not familiar with the different models registered, MLModelScope allows users to select models based on the application area — this lowers the barrier of DL usage. The command-line interface is provided for those interested in automating the evaluation and profiling process. Users can develop other clients that use the REST API to integrate MLModelScope within their AI applications.

6.2.3 MLModelScope Server

The MLModelScope *server* interacts with the MLModelScope client, agent, the middleware. It uses REST API to communicate with the MLModelScope clients and middleware,

and gRPC (Listing 6.4) to interact with the MLModelScope agents. To enforce **Feature 4**, the MLModelScope server can be load balanced to avoid it being a bottleneck.

In the **(1-9) evaluation workflow**, the server is responsible for **(2)** accepting tasks from the MLModelScope client, **(3)** querying the distributed registry and resolving the user-specified constraints to find MLModelScope agents capable of evaluating the request, **(4)** dispatching the evaluation task to the resolved agent(s) and generating loads for the evaluation, **(8)** collecting the evaluation summary from the agent(s), and **(9)** returning the result summary to the client. The load generator is placed on the server to avoid other programs interfering with the evaluation being measured and to emulate real-world scenarios such as cloud serving (**Feature 7**).

In the **(a-e) analysis workflow**, the server again **(a-b)** takes the user input, but, rather than performing evaluation, it **(c)** queries the evaluation database (Section 5.2.3), and then aggregates and analyzes the evaluation results. MLModelScope enables **Feature 8** through an across-stack analysis pipeline. It **(d)** consumes the benchmarking results and profiling traces in the evaluation database and performs the analysis. Then the server **(e)** sends the analysis result to the client. The profiling and automated analysis workflows in MLModelScope allow users to systematically compare models, frameworks, and system offerings.

6.2.4 Agent and Framework Predictor

A MLModelScope *agent* is a model serving process that is run on a system of interest (within a container or on bare metal) and handles requests from the MLModelScope server. MLModelScope agents continuously listen for jobs and communicate with the MLModelScope server through gRPC [94] as shown in Listing 6.4. A *framework predictor* resides within a MLModelScope agent and is a wrapper around a framework and links to the framework’s C library.

During the initialization phase (i), a MLModelScope agent publishes its built-in models and HW/SW information to the MLModelScope distributed registry. To perform the assigned evaluation task, the agent first **(5)** downloads the required evaluation assets using the *data manager*, it then executes the model evaluation pipeline which performs the pre-processing, calls the framework’s predictor for inference and then performs the post-processing. If profiling is enabled, the trace information is published to the **(6)** tracing server to get aggregated into a single profiling trace. **(7)** the benchmarked result and the profiling trace are published to the evaluation database. Aside from the framework predictor, all the other code — the data manager, pipeline executor, and tracing hooks — are shared across agents for different frameworks. While the default setup of MLModelScope is to run each

agent on a separate system, the design does not preclude one from running agents on the same system as separate processes.

Data Manager

The *data manager* manages the assets (e.g. dataset or model) required by the evaluation as specified within the model manifest. Assets can be hosted within MLModelScope’s *artifact repository*, on the web, or reside in the local file system of the MLModelScope agent. Both datasets and models are downloaded by the data manager on demand if they are not available on the local system. If the checksum is specified in the model manifest, then the checksum is verified after download. Model assets are stored using the frameworks’ corresponding deployment format.

Pipeline Executor and Operators

To enable **Feature 6** efficient evaluation workflow, MLModelScope leverages a streaming data processing pipeline design to perform the model evaluation. The pipeline is composed of *pipeline operators* which are mapped onto light-weight threads to make efficient use multiple CPUs as well as to overlap I/O with compute. Each operator within the pipeline forms a producer-consumer relationship by receiving values from the upstream operator(s) (via inbound streams), applies the specified function on the incoming data and usually producing new values, and propagates values downstream (via outbound streams) to the next operator(s). The pre- and post-processing operations, as well as the model inference, form the operators within the model evaluation pipeline.

Framework Predictor

Frameworks provide different APIs (usually across programming languages e.g. C/C++, Python, Java) to perform inference. To enable **Feature 2** consistent evaluation and maximize code reuse, MLModelScope wraps each framework’s C inference API. The wrapper is minimal and provides a uniform API across frameworks for performing model loading, unloading, and inference. This wrapper is called the *predictor interface* and is shown in Listing 6.3. MLModelScope does not require modifications to a framework and thus pre-compiled binary versions of frameworks (e.g. distributed through Python’s pip) or customized versions of a framework work within MLModelScope.

```

1  service Predict {
2  message PredictOptions {
3      enum TraceLevel {
4          NONE          = 0;
5          MODEL         = 1; // steps in the evaluation pipeline
6          FRAMEWORK    = 2; // layers within the framework and above
7          SYSTEM       = 3; // the system profilers and above
8          FULL         = 4; // includes all of the above
9      }
10     TraceLevel trace_level = 1;
11     Options     options     = 2;
12 }
13 message OpenRequest {
14     string model_name           = 1;
15     string model_version        = 2;
16     string framework_name      = 3;
17     string framework_version   = 4;
18     string model_manifest      = 5;
19     BenchmarkScenario benchmark_scenario = 6;
20     PredictOptions predict_options = 7;
21 }
22 // Opens a predictor and returns a PredictorHandle.
23 rpc Open(OpenRequest) returns (PredictorHandle){}
24 // Close a predictor and clear its memory.
25 rpc Close(PredictorHandle) returns (CloseResponse) {}
26 // Predict receives a stream of user data and runs
27 // the predictor on each element of the data according
28 // to the provided benchmark scenario.
29 rpc Predict(PredictorHandlePredictorHandle, UserInput)
30     returns (FeaturesResponse) {}
31 }

```

Listing 6.4: MLModelScope’s minimal gRPC interface.

MLModelScope design supports agents on ASIC and FPGA. Any code implementing the predictor interface shown in Listing 6.3 is a valid MLModelScope predictor. This means that FPGA and ASIC hardware, which do not have a framework per se, can be exposed as a predictor. For example, for an FPGA the `Open` function call loads a bitfile into the FPGA, the `Close` unloads it, and the `Predict` runs the inference on the FPGA. Except for implementing these 3 API functions, no code needs to change for the FPGA to be exposed to MLModelScope.

Tracing Hooks

To enable **Feature 9**, MLModelScope leverages XSP (Chapter 4) to capture the profiles at different levels of granularity (model-, framework-, and system-level). A *tracing hook* in XSP is a pair of start and end code snippets and follows the standards [33] to capture an interval of time. The captured time interval along with the context and metadata is called a *trace event*. and is published to the tracing server (Section 6.2.5). Trace events are published

asynchronously to the tracing server, where they are aggregated using the timestamp and context information into a single end-to-end timeline.

The trace granularity is a user-specified option (part of the benchmarking scenario) and allows one to get a holistic and hierarchical view of the execution profile. For example, a user can enable model- and framework-level profiling by setting the trace level to `framework`, or can disable the profiling all together by setting the trace level to `none`. Through MLModelScope’s trace, a user can get a holistic view of the model evaluation to identify bottlenecks at each level of inference.

6.2.5 Middleware

The MLModelScope middleware layer is composed of services and utilities that support the MLModelScope Server in orchestrating model evaluations and the MLModelScope agents in provisioning, monitoring, and aggregating the execution of the agents.

Distributed Registry

MLModelScope leverages a distributed key-value store to store the registered model manifests and running agents, referred to as the *distributed registry*. MLModelScope uses the registry to facilitate the discovery of models, solve user-specified constraints for selecting MLModelScope agents, and load balances the requests across agents.

Evaluation Database

In the benchmarking workflow, after completing a model evaluation, the MLModelScope agent uses the user input as the key to store the benchmarking result and profiling trace in the *evaluation database*. MLModelScope summarizes and generates plots to aid in comparing the performance across experiments. Users can view historical evaluations through the website or command line using the input constraints.

Tracing Server

The *tracing server* accepts profiling data published by the MLModelScope agent’s trace hooks. As stated in Section 6.2.4, user-specified options control the granularity (model, framework, or system) of the trace events captured (Lines 4–9 in Listing 6.4).

Table 6.1: Four systems with Volta, Pascal, Maxwell, and Kepler GPUs are selected for evaluation.

Name	CPU	GPU	GPU Architecture	GPU Theoretical Flops (TFlops)	GPU Memory Bandwidth (GB/s)	Cost (\$/hr)
AWS P3 (2XLarge)	Intel Xeon E5-2686 v4 @ 2.30GHz	Tesla V100-SXM2-16GB	Volta	15.7	900	3.06
AWS G3 (XLarge)	Intel Xeon E5-2686 v4 @ 2.30GHz	Tesla M60	Maxwell	9.6	320	0.90
AWS P2 (XLarge)	Intel Xeon E5-2686 v4 @ 2.30GHz	Tesla K80	Kepler	5.6	480	0.75
IBM P8	IBM S822LC Power8 @ 3.5GHz	Tesla P100-SXM2	Pascal	10.6	732	-

6.2.6 Extensibility and Customization

MLModelScope is built from modular components and is designed to be extensible and customizable. Users can disable components, such as tracing, with a runtime option or conditional compilation, for example. Users can extend MLModelScope by adding components such as models, frameworks, or tracing hooks.

Adding Models As models are defined through the model manifest file, no coding is required to add models. Once a model is added to MLModelScope, then it can be used through its website, command line, or API interfaces. Permissions can be set to control who can use or view a model.

Adding Frameworks To use new or custom versions of a built-in framework requires no code modification but a framework manifest as shown in Listing 6.2. To add support for a new type of framework in MLModelScope, the user needs to implement the framework wrapper and expose the framework as a MLModelScope predictor. The predictor interface is defined by a set of 3 functions — one to open a model, another to perform the inference, and finally, one to close the model — as shown in Listing 6.3. The auxiliary code that forms an agent is common across frameworks and does not need to be modified.

Adding Tracing Hooks MLModelScope is configured to capture a set of default system metrics using the system-level tracing hooks, as described in Chapter 4. Users can configure these existing tracing hooks to capture other system metrics. For example, to limit profiling overhead, by default, the CUPTI tracing hooks capture only some CUDA runtime API, GPU activities (kernels and memory copy), and GPU metrics. They can be configured to capture other GPU activities and metrics, or NVTX markers. Moreover, users can integrate other system profilers into MLModelScope by implementing the XSP tracing interface.

6.3 EVALUATION

Previous sections discussed in detail how MLModelScope’s design and implementation achieves the **Feature 1-6** and **Feature 10** design objectives. In this section, we focus on evaluating how MLModelScope handles **Feature 7** different benchmarking scenarios, **Feature 8**

Table 6.2: 37 TensorFlow image classification models from MLPerf, AI-Matrix, and TensorFlow Slim are used for evaluation and are sorted by accuracy. We measured the online latency, 90th percentile latency, maximum throughput at the optimal batch size for each model.

ID	Name	Top 1 Accuracy	Graph Size (MB)	Online Trimmed Mean Latency (ms)	Online 90 th Percentile Latency (ms)	Max Throughput (Inputs/Sec)	Optimal Batch Size
1	Inception_ResNet_v2	80.40	214	23.95	24.2	346.6	128
2	Inception_v4	80.20	163	17.36	17.6	436.7	128
3	Inception_v3	78.00	91	9.2	9.48	811.0	64
4	ResNet_v2_152	77.80	231	14.44	14.65	466.8	256
5	ResNet_v2_101	77.00	170	10.31	10.55	671.7	256
6	ResNet_v1_152	76.80	230	13.67	13.9	541.3	256
7	MLPerf_ResNet50_v1.5	76.46	103	6.33	6.53	930.7	256
8	ResNet_v1_101	76.40	170	9.93	10.08	774.7	256
9	AI_Matrix_ResNet152	75.93	230	14.58	14.72	468.0	256
10	ResNet_v2_50	75.60	98	6.17	6.35	1,119.7	256
11	ResNet_v1_50	75.20	98	6.31	6.41	1,284.6	256
12	AI_Matrix_ResNet50	74.38	98	6.11	6.25	1,060.3	256
13	Inception_v2	73.90	43	6.28	6.56	2,032.0	128
14	AI_Matrix_DenseNet121	73.29	31	11.17	11.49	846.4	32
15	MLPerf_MobileNet_v1	71.68	17	2.46	2.66	2,576.4	128
16	VGG16	71.50	528	22.43	22.59	687.5	256
17	VGG19	71.10	548	23.0	23.31	593.4	256
18	MobileNet_v1_1.0_224	70.90	16	2.59	2.75	2,580.6	128
19	AI_Matrix_GoogleNet	70.01	27	5.43	5.55	2,464.5	128
20	MobileNet_v1_1.0_192	70.00	16	2.55	2.67	3,460.8	128
21	Inception_v1	69.80	26	5.27	5.41	2,576.6	128
22	BVLC_GoogLeNet	68.70	27	6.05	6.17	951.7	8
23	MobileNet_v1_0.75_224	68.40	10	2.48	2.61	3,183.7	64
24	MobileNet_v1_1.0_160	68.00	16	2.57	2.74	4,240.5	64
25	MobileNet_v1_0.75_192	67.20	10	2.42	2.6	4,187.8	64
26	MobileNet_v1_0.75_160	65.30	10	2.48	2.65	5,569.6	64
27	MobileNet_v1_1.0_128	65.20	16	2.29	2.46	6,743.2	64
28	MobileNet_v1_0.5_224	63.30	5.2	2.39	2.58	3,346.5	64
29	MobileNet_v1_0.75_128	62.10	10	2.3	2.47	8,378.4	64
30	MobileNet_v1_0.5_192	61.70	5.2	2.48	2.67	4,453.2	64
31	MobileNet_v1_0.5_160	59.10	5.2	2.42	2.58	6,148.7	64
32	BVLC_AlexNet	57.10	233	2.33	2.5	2,495.8	64
33	MobileNet_v1_0.5_128	56.30	5.2	2.21	2.33	8,924.0	64
34	MobileNet_v1_0.25_224	49.80	1.9	2.46	3.40	5,257.9	64
35	MobileNet_v1_0.25_192	47.70	1.9	2.44	2.6	7,135.7	64
36	MobileNet_v1_0.25_160	45.50	1.9	2.39	2.53	10,081.5	256
37	MobileNet_v1_0.25_128	41.50	1.9	2.28	2.46	10,707.6	256

result summarization, and **Feature 9** inspection of model execution. We installed MLModelScope on the systems listed in Table 6.1. Unless otherwise noted, all MLModelScope agents are run within a docker container built using NVIDIA’s TensorFlow NGC v19.06 container with the TensorFlow v1.13.1 library. All evaluations were performed using the command-line interface and are run in parallel across the systems.

6.3.1 Benchmarking Scenarios

To show that MLModelScope allows users to choose from different models and system offerings for the same DL task, we compared the inference performance across the 37 Ten-

orFlow models (Table 6.2) and systems (Table 6.1) under different benchmark scenarios. For each model, we measured its trimmed mean latency¹ and 90th percentile latency in on-line (batch size = 1) inference scenario, and the maximum throughput in batched inference scenario on the AWS P3 system. The model accuracy achieved using the ImageNet validation dataset and the model size is listed. A model deployer can use this accuracy and performance information to choose the best model on a system given the accuracy and target latency or throughput objectives.

Model Accuracy, Size, and Performance We examined the relationship between the model accuracy and both online latency (Figure 6.3) and maximum throughput (Figure 6.2). In both figures, the area of the circles is proportional to the model’s graph size. In Figure 6.2 we find a limited correlation between a model’s online latency and its accuracy — models taking longer time to run do not necessarily achieve higher accuracies; e.g. model 15 vs 22. While large models tend to have longer online latencies, this is not always true; e.g. model 14 is smaller in size but takes longer to run compared to models 3, 5, 8, etc. Similarly, in Figure 6.3, we find a limited correlation between a model’s accuracy and its maximum throughput — two models with comparable maximum throughputs can achieve quite different accuracies; e.g. models 2 and 17. Moreover, we see both figures show that the graph size (which roughly represents the number of weight values) is not directly correlated to either accuracy or performance. Models closer to the upper left corner (low latency and high accuracy) in Figure 6.2 are favorable in the online inference scenarios, and models closer to the upper right corner (high throughput and high accuracy) in Figure 6.3 are favorable for batched inference. Users can use this information to select the best model depending on their objectives.

Model Throughput Scalability Across Batch Sizes When comparing the model on-line latency and maximum throughput (Figures 6.2 and 6.3 respectively), we observed that models which exhibit good online inference latency do not necessarily perform well in the batched inference scenario where throughput is important. We measured how the model throughput scales with batch size (referred to as *throughput scalability*) and present this model characteristic in Figure 6.5. As shown, the throughput scalability varies across models. Even models with similar architectures can have different throughput scalability (e.g., models 4 and 6, models 5 and 8, and models 10 and 11). In general, smaller models tend to have better throughput scalability. However, there are exceptions, for example, models 16 and 17 are large and have good throughput scalability.

Model Performance Across Systems Overall, the ResNet_50 class of models offer a

¹Trimmed mean is computed by removing 20% of the smallest and largest elements and computing the mean of the residual.

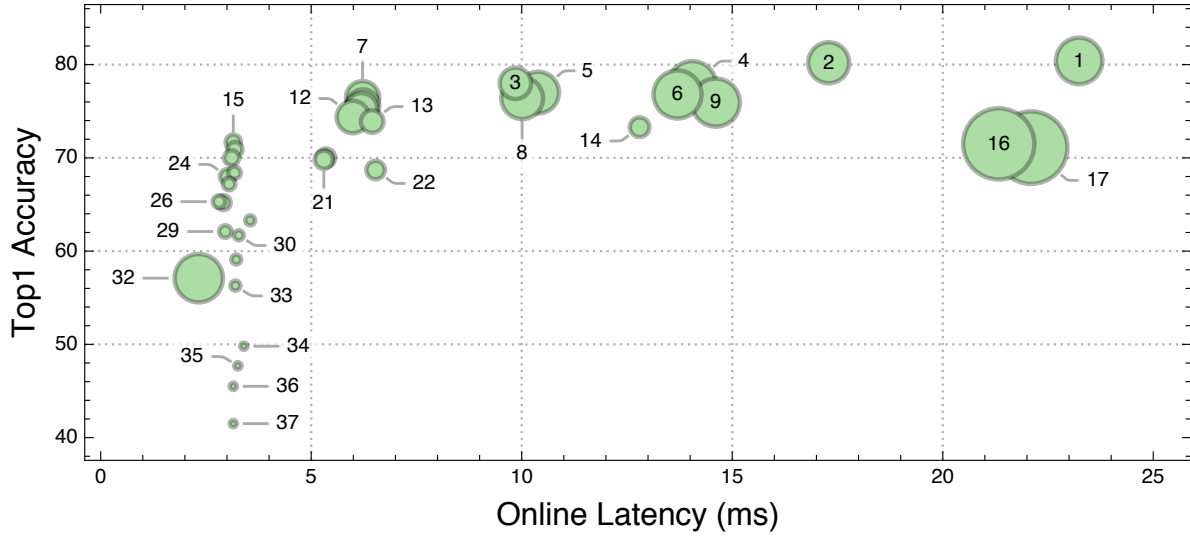


Figure 6.2: Accuracy vs online latency.

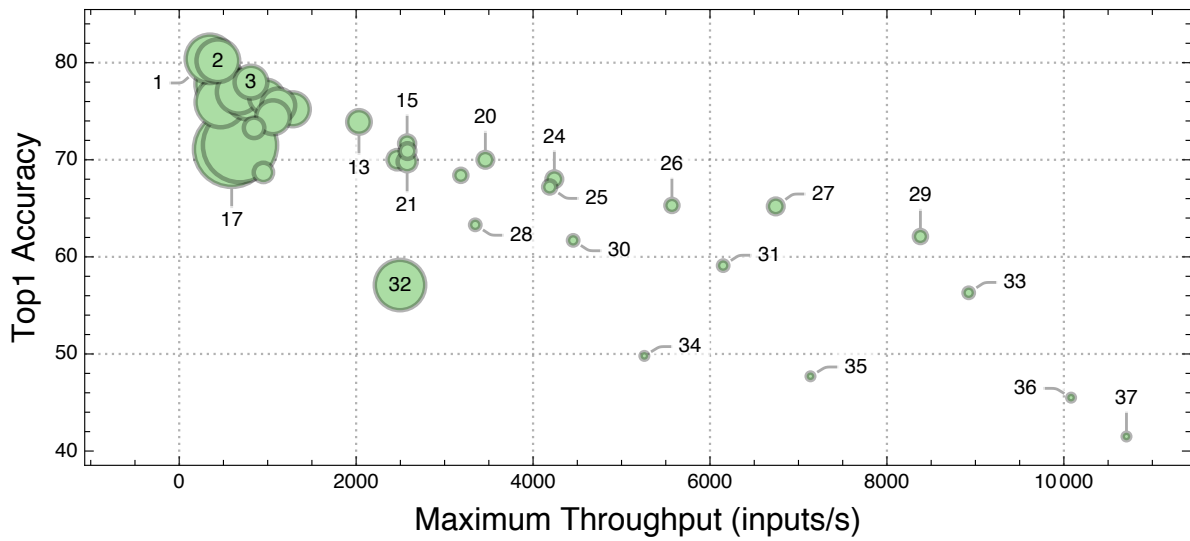


Figure 6.3: Accuracy vs maximum throughput.

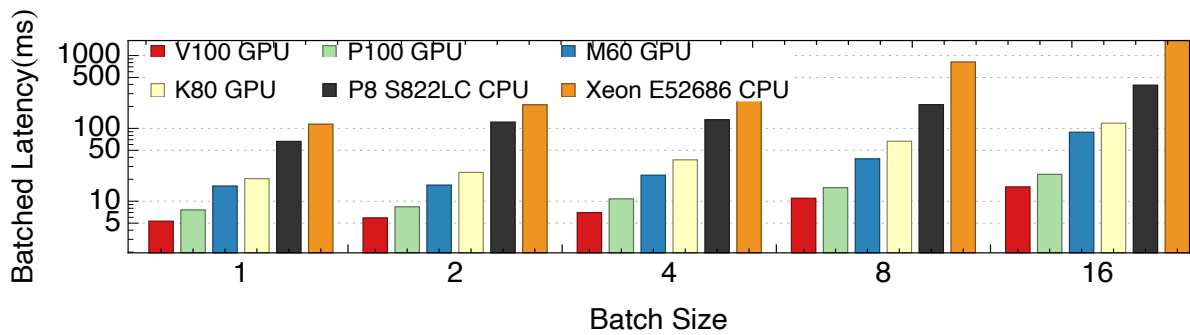


Figure 6.4: The batched latency of `ResetNet 50` across the GPUs and CPUs listed in Table 6.1.

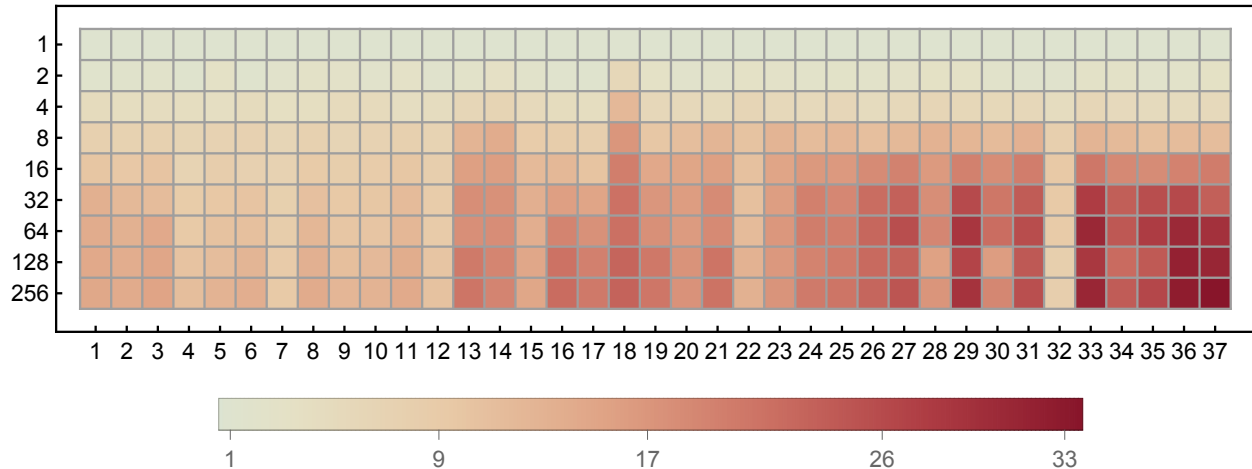


Figure 6.5: The throughput improvement (over batch size 1) heatmap across batch sizes on AWS P3 for the 37 models in Table 6.2. The y -axis shows the batch size, whereas the x -axis shows the model ID.

balance between model size, accuracy, performance and are commonly used in practice. Thus, we use `ResNet_50` in online inference as an example to show how to use `MLModelScope` to choose the best system given a model. We evaluated `ResNet_50` across all CPUs and GPUs listed in Table 6.1 and the results are shown in Figure 6.4. On the CPU side, IBM S822LC Power8 achieves between $1.7\times$ and $4.1\times$ speedup over Intel Xeon E5-2686. The P8 CPU is more performant than Xeon CPU [95], with the P8 running at 3.5 GHz and having 10 cores each capable of running 80 SMT threads. On the GPU side, as expected, V100 GPU achieves the lowest latency followed by the P100. The M60 GPU is $1.2\times$ to $1.7\times$ faster than the K80. When this information is coupled with the pricing information of the systems, one can determine which system is most cost-efficient given a latency target and benchmarking scenario. For example, given that K80 costs 0.90\$/hr and M60 costs 0.75\$/hr on AWS, we can tell that M60 is both more cost-efficient and faster than K80 — thus, M60 is overall better suited for `ResNet_50` online inference when compared to K80 on AWS.

6.3.2 Model Execution Inspection

`MLModelScope`'s evaluation inspection capability helps users to understand the model execution and identify performance bottlenecks. We show this by performing a case study of “cold-start” inference (where the model needs to be loaded into the memory before inference) of model 32. The cold-start inference is common on low-memory systems and in serving schemes that perform one-off evaluation. We choose `BVLC_AlexNet` because it is easy to see the effects of the “cold-start” inference scenario using Caffe on the AWS P3 and IBM

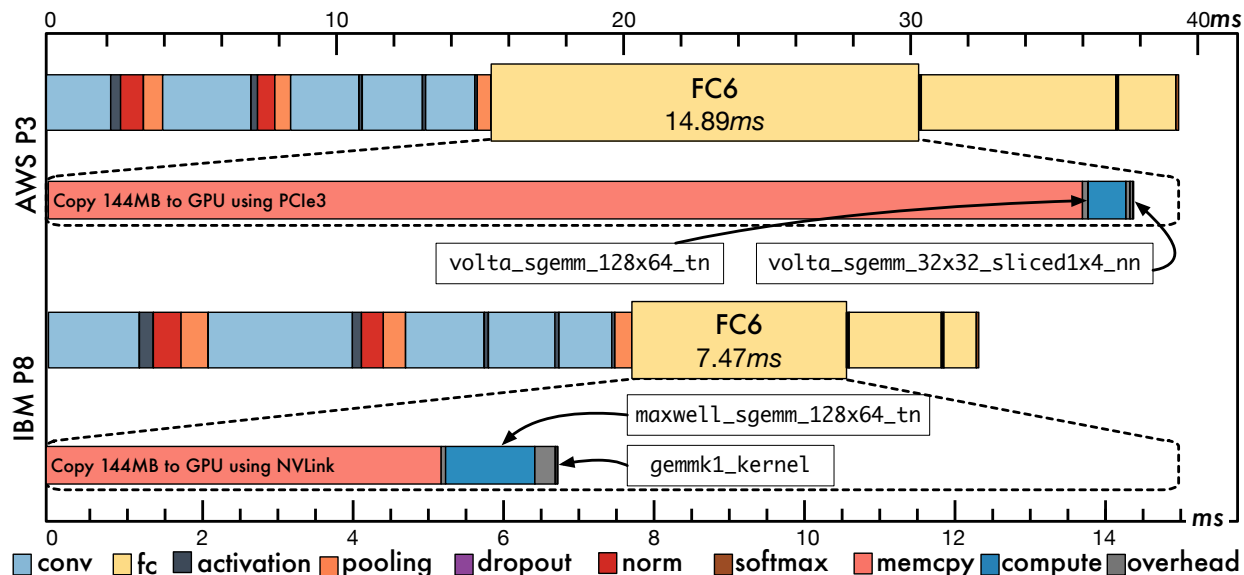


Figure 6.6: The MLModelScope inspection of “cold-start” BVLC_AlexNet inference with batch size 64 running Caffe v0.8 using GPU on AWS P3 and IBM P8. The color-coding of layers signify the layer type.

Table 6.3: The ResNet 50 layer information using AWS P3 (Tesla V100 GPU) with batch size 256. The top 5 most time-consuming layers are summarized from the tracing profile. In total, there are 234 layers of which 143 take less than 1ms.

Layer Index	Layer Name	Layer Type	Layer Shape	Dominant GPU Kernel(s) Name	Latency (ms)	Alloc Mem (MB)
208	conv2d_48/Conv2D	Conv2D	(256, 512, 7, 7)	volta_cgemm_32x32_tn	7.59	25.7
221	conv2d_51/Conv2D	Conv2D	(256, 512, 7, 7)	volta_cgemm_32x32_tn	7.57	25.7
195	conv2d_45/Conv2D	Conv2D	(256, 512, 7, 7)	volta_scudnn_128x128_relu_interior_nn.v1	5.67	25.7
3	conv2d/Conv2D	Conv2D	(256, 64, 112, 112)	volta_scudnn_128x64_relu_interior_nn.v1	5.08	822.1
113	conv2d_26/Conv2D	Conv2D	(256, 256, 14, 14)	volta_scudnn_128x64_relu_interior_nn.v1	4.67	51.4

P8 GPU systems with batch size 64. The results are shown in Figure 6.6. We see that IBM P8 with P100 GPU is more performant than AWS P3 which has V100 GPU. We used MLModelScope’s model execution inspection capability to delve deeper into the model and to reveal the reason. We “zoomed” into the longest-running layer (fc6) and find that most of the time is spent performing copies for the (fc6) layer weights. On AWS P3, the fc6 layer takes 39.44ms whereas it takes 32.4ms on P8. This is due to the P8 system having an NVLink interconnect which has a theoretical peak CPU to GPU bandwidth of 40 GB/s (33 GB/s measured) while the AWS P3 system performs the copy over PCIe-3 which has a maximum theoretical bandwidth of 16 GB/s (12 GB/s measured). Therefore, despite P3’s lower compute latency, we observed a lower overall layer and model latency on the P8 system due to the fc6 layer being memory bound.

Using MLModelScope’s model execution inspection, it is clear that the memory copy is the bottleneck for the “cold-start” inference. To verify this observation, we examined the Caffe source code. Caffe performs lazy memory copies for layer weights just before execution. This causes compute to stall while the weights are being copied — since the weights of the FC layer are the biggest. A better strategy — used by Caffe2, MXNet, TensorFlow, and TensorRT — is to eagerly copy data asynchronously and utilize CUDA streams to overlap compute with memory transfer.

6.3.3 Benchmarking Analysis and Reporting

We used MLModelScope’s analysis workflow to perform an in-depth analysis of the 37 models and to show MLModelScope’s benchmarking analysis and reporting capabilities. All results were generated automatically using MLModelScope and further results are available at [\[link\]](#) for the reader’s inspection. As an example, we highlight the model-layer-GPU kernel analysis of `ResNet_50` using batch size 256 (the optimal batch size with the maximum throughput) on AWS P3. MLModelScope can capture the layers in a model and correlate the GPU kernel calls to each layer; i.e. tell which GPU kernels are executed by a certain layer. Table 6.3 shows the top 5 most time-consuming layers of `ResNet_50` as well as the dominant kernel within each layer. Through the analysis and summarization workflow, users can easily digest the results and identify understand model-, framework-, and system-level bottlenecks.

6.4 RELATED WORK

To my knowledge, this is the first work to describe the design and implementation of a scalable DL benchmarking platform. While there have been efforts to develop certain aspects of MLModelScope, the efforts have been quite dispersed and there has not been a cohesive system that addresses **Feature 1-10**. For example, while there is active work on proposing benchmark suites, reference workloads, and analysis [3, 4], they provide **Feature 7** a set of benchmarking scenarios and a simple mechanism for **Feature 8** analysis and reporting of the results. The models within these benchmarks can be consumed by MLModelScope, and we have shown analysis which uses the benchmark-provided models. Other works are purely model serving platforms [96, 97] which address **Feature 4** scalable evaluation and possibly **Feature 5** artifact versioning but nothing else. Finally, systems such as as [89, 98, 99] track the model and data from their use in training till deployment to ensure either **Feature 1** reproducible or **Feature 2** consistent evaluation.

6.5 CONCLUSION

Evaluating, comparing, and analyzing the performance of DL innovations is critical for their adoption. This chapter first identifies 10 design objectives of a DL benchmarking platform. It then describes the design and implementation of MLModelScope — an open-source DL benchmarking platform that achieves these design objectives. MLModelScope offers a unified and holistic way to evaluate and inspect DL models, and provides an automated analysis and reporting workflow to summarize the results. We demonstrate the usability and effectiveness of MLModelScope by using it to evaluate a set of models and show how model, hardware, and framework selection affects model accuracy and performance under different benchmarking scenarios. We are actively working on curating automated analysis and reports obtained through MLModelScope.

CHAPTER 7: OTHER RELEVANT WORKS

This chapter presents other relevant works in DL performance understanding and optimization. Specifically, we propose TrIMS to mitigate the model loading overhead in DL inference, TOPS to leverage TCUs for non-GEMM operations, and CommScope to understand memory transfer behaviors across different scenarios.

7.1 TRIMS: TRANSPARENT AND ISOLATED MODEL SHARING FOR DL INFERENCE

Today, many business-logic and consumer applications rely on DL inferences as core components within their application pipelines. These pipelines tend to be deployed to the cloud through serverless computing, since they abstract away low-level details such as system setup and DevOps while providing isolation, decentralization, and scalability, all the while being more cost-effective than dedicated servers. User code which defines the pipeline (acting as glue code) is commonly deployed through Function as a Service (FaaS) [100–103] onto the cloud and is made available through HTTP endpoints. Since FaaS executes arbitrary user code, the host system **must** execute the code in isolation — through virtual machines (VMs) or containers.

While serverless ¹ is an emerging and compelling computing paradigm for event-driven cloud applications, use cases of the current FaaS offerings are limited. Currently, serverless functions run as short-lived VMs or containers, and thus are not ideal for long running jobs. FaaS functions are also unable to work efficiently with data or distributed computing resources [104, 105], thus are not ideal for functions that require large data.

Recent work has proposed extensions to the FaaS infrastructure to expand its usage within DL domains and facilitate it to leverage heterogeneous hardware. In [104], the authors advocate for code fluidity, where user functions are shipped to the data rather than the data being downloaded by the code. The advantages for this are three-fold. **Ⓐ** It avoids the overhead of copying data over slow interconnects (such as networks). **Ⓑ** Leveraging heterogeneous hardware becomes attractive if data overhead is reduced and isolation is guaranteed. Finally, **Ⓒ** since user functions that use the same data are routed to the same system, it exposes an opportunity for sharing constant data across functions.

¹Cloud provider runs the server, and dynamically manages the allocation of machine resources

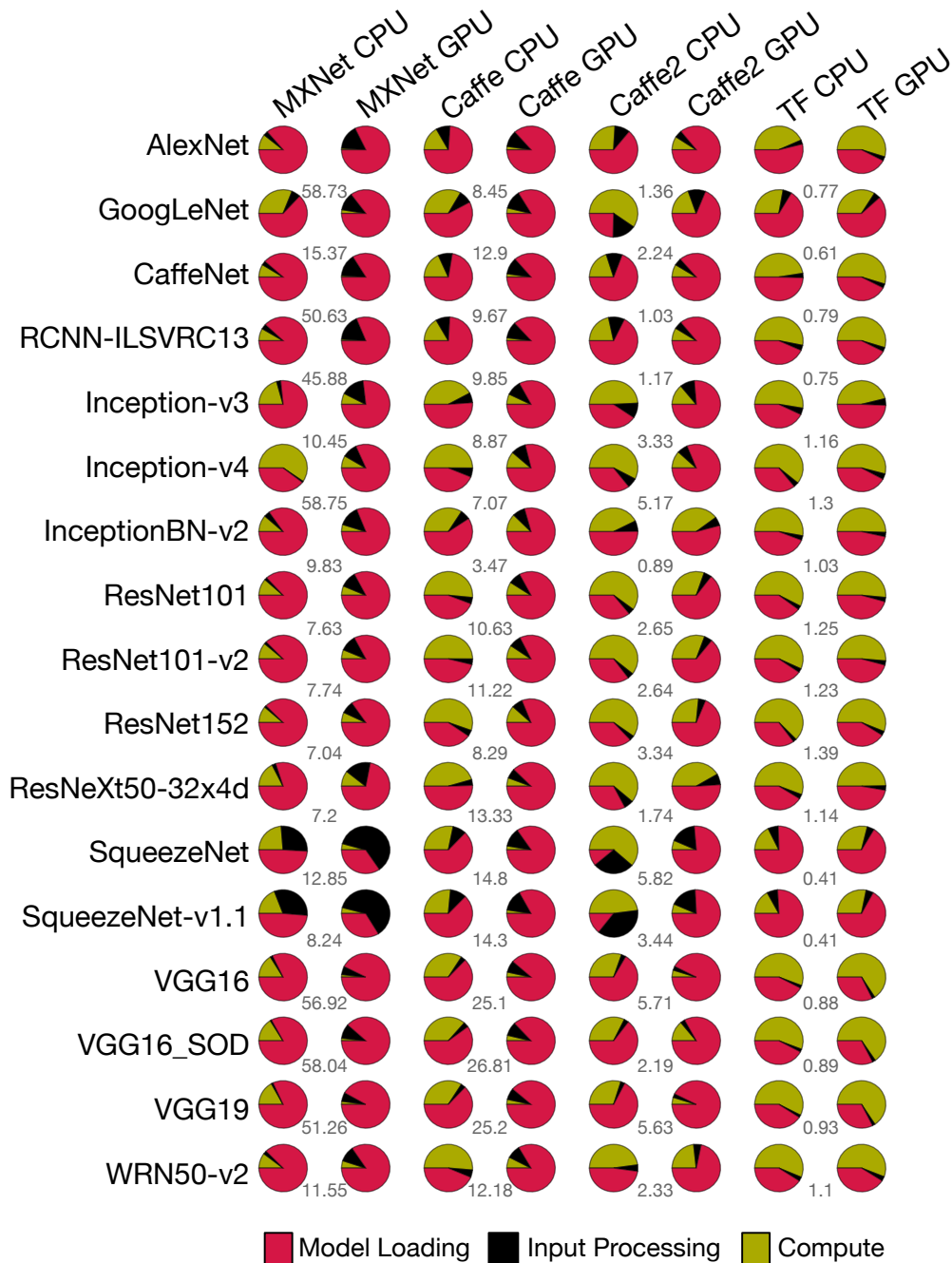


Figure 7.1: Percentage of time spent in model loading, inference computation, and image preprocessing for online DL inference ($batchsize = 1$) using CPU and GPU for MXNet, Caffe, Caffe2, and TensorFlow on an IBM S822LC with Pascal GPUs. The speedup of using GPU over CPU for inference compute is shown between the pie charts. Inference time for all frameworks is dominated by model loading except for small models. For TensorFlow, GPU initialization overhead impacts the end-to-end time and achieved speedup.

Both **a** and **b** allow users to minimize data copy overhead and accelerate the computation

using heterogeneous hardware. Yet, after removing the inter-node data copy overhead, intra-node data movement becomes a contributing factor to latency. This is even more true for heterogeneous devices, since data must be copied onto the device. This makes heterogeneous devices, such as GPUs, less attractive for accelerating latency-sensitive inference — even though they would offer a significant compute speed advantage, as shown in Figure 7.1. For **©** we observe that DL models are shared extensively across user pipelines. For example, Google reported that 41 natural language translation models can accommodate over 75% of their translation requests in [106]. Because model parameters are constant, we can use data sharing across FaaS functions to share DL models within a model catalog, hence eliminating the model loading overhead, decreasing the end-to-end latency, and reducing the memory footprint (since there is only one instance of a model in memory for many users) for DL inferences.

In [91], we propose a **T**ransparent and **I**solated **M**odel **S**haring (TrIMS) scheme to leverage the data sharing opportunity introduced by collocating user code with model catalogs within FaaS — it minimizes model loading and data movement overhead while maintaining the isolation constraints and increasing hardware resource utilization. We also introduce the TrIMS’s model resource manager (MRM) layer which offers a multi-tiered cache for DL models to be shared across user FaaS functions. By decreasing model loading and data movement overhead, TrIMS decreases latency of end-to-end model inference, making inference on GPU a viable FaaS target. TrIMS also increases memory efficiency for cloud data centers while maintaining accuracy. In [91] we focus on online prediction within latency sensitive FaaS functions. Specifically, we make the following contributions:

- We characterize the overhead for DL model inference across popular DL frameworks on both CPUs and GPUs and identify model loading as the bottleneck.
- We propose TrIMS to mitigate the model loading overhead faced by collocating user code with model catalogs within FaaS, and increase the hardware resource utilization by sharing DL models across all levels of the memory hierarchy in the cloud environment — GPU, CPU, local storage, and remote storage. To our knowledge, this work is the first to propose sharing DL models across isolated FaaS functions.
- We implement TrIMS within Apache MXNet [107] and evaluate the impact on GPU inference performance for a representative set of models and systems. We show that TrIMS provides $1.12\times - 24\times$ speedup on small (less than 600MB) models and $5\times - 210\times$ speedup on large (up to 6GB) models and is within 20% of ideal speedup (with ideal being that model loading and data movement taking no time), and gives $8\times$

system throughput improvement.

- TrIMS eliminates a substantial part of the non-compute components of the end-to-end latency, making DL model inference on GPU and other novel compute accelerators more viable.
- We architect TrIMS so that it can be easily integrated with existing FaaS systems and DL frameworks without user code changes. TrIMS is designed to be compatible with existing framework usage patterns, and requires minimal modifications for framework developers.
- While we use DL inference as the motivating application, TrIMS is not restricted to DL. TrIMS can be generalized to any application where one can share data across FaaS functions, be it a common database, knowledge base, or dataset.

7.2 TOPS: ACCELERATING REDUCTION AND SCAN USING TENSOR CORE UNITS

Deep learning’s reliance on matrix-multiplication (GEMM) for compute has driven both research and industry to develop matrix-multiplication accelerator hardware — collectively called Tensor Core Units (TCUs). TCUs are designed to accelerate Multilayer Perceptrons (MLP), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN) or Deep Neural Network (DNN) in general. TCUs come under the guise of different marketing terms, be it NVIDIA’s Tensor Cores [108], Google’s Tensor Processing Unit [109], Intel’s DLBoost [110], Apple A11’s Neural Engine [111], Tesla’s HW3, or ARM’s ML Processor [112]. They vary in the underlying hardware implementation [113–116], and are prevalent [108, 117, 118] in both cloud and edge devices.

To show the theoretical benefits of TCUs, consider the NVIDIA Volta V100 GPUs architecture. Using V100 Tensor Cores, one achieves a $8\times$ throughput increase per Streaming Multiprocessors (SM) over previous Pascal GP100 generation. This throughput increase is because each V100 SM is capable of performing 1024 half precision operations per cycle using the TCUs whereas the GP100 SM is capable of performing 128 half precision operations per cycle without the TCUs. The throughput increase is enabled by the fact that the V100 dedicates a large chip area of the SM subcore to TCUs (Figure 7.2).

Although TCUs are prevalent and promise increase in performance and/or energy efficiency and are heavily used within supercomputers [119, 120] to achieve exascale performance, they suffer from over specialization. Currently, no algorithm other than GEMM

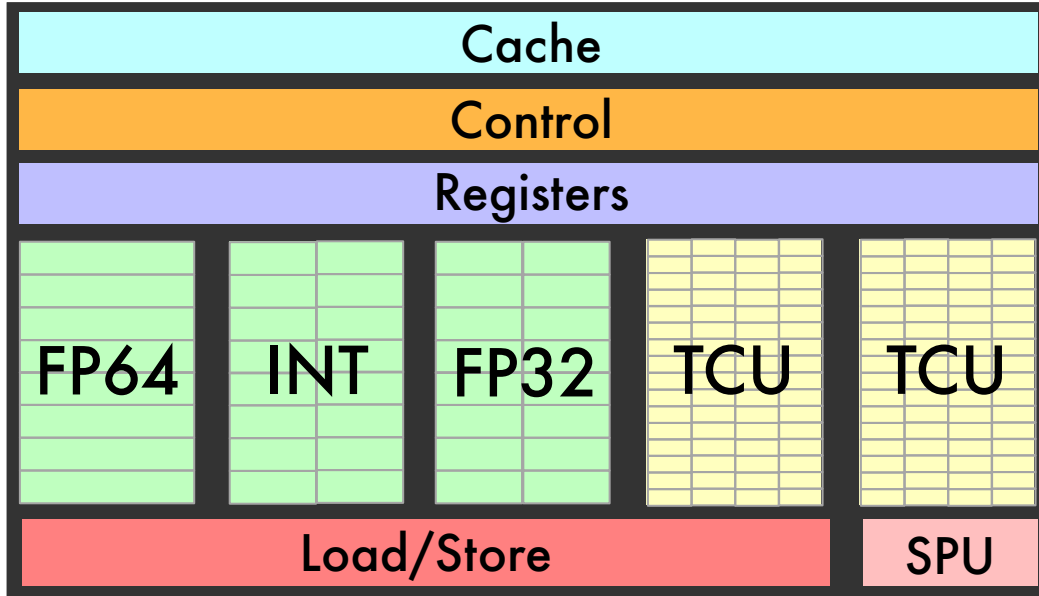


Figure 7.2: Each subcore (processing block) in the NVIDIA Tesla V100 PCI-E architecture contains 2 TCUs. In total, 640 TCUs are available — achieving a theoretical peak of 113 TFLOPS.

utilizes the NVIDIA TCUs. This results in idle TCUs, low chip utilization, and limits TCUs applicability to specialized libraries or narrow application domains.

In [121], we expand the class of algorithms that can execute on TCUs— enabling the TCUs to be used within a wider range of non-GEMM algorithms. We choose reduction and scan, since a large body of work [122–124] has shown that they are key primitives for data parallel implementations of radix sort, quicksort, lexical analysis, stream compaction, and polynomial evaluation. In this work, we formulate a mapping of reduction or scan onto TCUs. We then introduce algorithms for cache- (warp-), processing element (PE)/core- (block-), and device- (grid-) level reduction and scan and show their performance on NVIDIA TCUs. We separate our algorithm description from implementation, making the algorithms, motivation, methods, and observations generally applicable to a broader range of TCUs and numerical precision agnostic. While the formulation is the main objective of this work, we show that an implementation of our algorithms on NVIDIA V100 is either order of magnitude faster or rival the fastest GPU implementation, with much lower programming complexity. The key contributions of this work are:

1. We show how to use TCUs to compute both reduction and scan. We believe we are the first to formulate these algorithms in terms of TCU operations in a manner that is independent to the underlying TCU architecture.
2. We implement our algorithms onto NVIDIA V100 GPUs and show orders of magnitude

speedup over state-of-art algorithms for small segment sizes. Small segments are common in mathematics (e.g. evaluating polynomials), scientific applications (e.g. finite difference), and machine learning (e.g. batch norm) applications. For large segments, we are comparable to the fastest algorithms and achieve 89 – 98% of theoretical peak memory copy bandwidth.

3. We show that our implementation is up to 22% more power efficient and decreases the utilization of general purpose ALUs.
4. We describe the current usage and programmability of the NVIDIA TensorCore and evaluate GEMM on the TCUs using cuBLAS [48], CUTLASS [125] and the CUDA TCU API.

7.3 COMMSCOPE

Data-intensive applications such as machine learning and analytics have created a demand for faster interconnects to avert the memory bandwidth wall and allow GPUs to be effectively leveraged for lower compute intensity tasks. This has resulted in wide adoption of heterogeneous systems with varying underlying interconnects, and has delegated the task of understanding and copying data to the system or application developer. No longer is a malloc followed by memcpy the only or dominating modality of data transfer; application developers are faced with additional options such as unified memory and zero-copy memory. Data transfer performance on these systems is now impacted by many factors including data transfer modality, system interconnect hardware details, CPU caching state, CPU power management state, driver policies, virtual memory paging efficiency, and data placement.

CommScope [126] presents a set of microbenchmarks designed for system and application developers to understand memory transfer behavior across different data placement and exchange scenarios. CommScope comprehensively measures the latency and bandwidth of CUDA data transfer primitives, and avoids common pitfalls in ad-hoc measurements by controlling CPU caches, clock frequencies, and avoids measuring synchronization costs imposed by the measurement methodology where possible. CommScope also presents an evaluation of CommScope on systems featuring the POWER and x86 CPU architectures and PCIe 3, NVLink 1, and NVLink 2 interconnects. These systems are chosen as representative configurations of current high-performance GPU platforms. CommScope measurements can serve to update insights about the relative performance of data transfer methods on current systems. This work also reports insights into how high-level system design choices affect the

performance of these data transfers, and how developers can optimize applications on these systems.

CHAPTER 8: CONCLUSION

The performance engineering of DL workloads faces new challenges that stifle the adoption of DL innovations. This thesis addresses the challenges in (1) reducing the effort to develop, maintain, and run DL benchmarks, (2) understanding DL performance across different levels in the HW/SW stack, (3) interpreting DL benchmarking results into optimization opportunities, and (4) evaluating and comparing DL innovations in a consistent, reproducible and efficient way. First, we introduce DLBricks to address (1). DLBricks is a composable benchmark generation design that decomposes DL models into a set of unique runnable networks and constructs the original model’s performance using the performance of the generated benchmarks. Second, we present XSP to address (2). XSP is an across-stack profiling design that captures and correlates profiles from different sources to obtain a hierarchical view of DL model execution. XSP innovatively leverages distributed tracing and accurately captures the profiles at each level of the HW/SW stack in spite of profiling overhead. Third, we present Benanza to address (3). We define a “lower-bound” latency metric that estimates the ideal latency of a model given a specific GPU hardware and software stack. Benanza automatically generates micro-benchmarks given a set of models, computes their “lower-bound” latencies using the benchmark data, and informs the optimizations of their execution on GPUs. Benanza guides researchers to optimization opportunities and assesses hypothetical execution scenarios on GPUs. Finally, to address (4), we design MLModelScope, a consistent, reproducible, and scalable DL experimentation platform to facilitate the evaluation and comparison of DL innovations. This thesis also briefly discusses TrIMS, TOPS, and CommScope which solve relevant problems in DL performance domain. Overall, this thesis has provided a coherent set of works that address the challenges in the performance benchmarking, analysis and optimization of deep learning workloads.

REFERENCES

- [1] J. Dean, D. Patterson, and C. Young, “A new golden age in computer architecture: Empowering the machine-learning revolution,” *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar. 2018. [Online]. Available: <https://doi.org/10.1109/mm.2018.112130030>
- [2] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE. IEEE, Feb. 2018. [Online]. Available: <https://doi.org/10.1109/hpca.2018.00059> pp. 620–629.
- [3] “MLPerf Inference,” github.com/mlperf/inference, accessed: 2020-02-20.
- [4] W. Zhang, W. Wei, L. Xu, L. Jin, and C. Li, “AI Matrix: A Deep Learning Benchmark for Alibaba Data Centers,” 2019.
- [5] Baidu, “Deepbench,” github.com/baidu-research/DeepBench, accessed: 2020-02-20.
- [6] C. Coleman, M. Zaharia, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, and C. Ré, “Analysis of DAWNbench, a time-to-accuracy machine learning performance benchmark,” *SIGOPS Oper. Syst. Rev.*, vol. 53, no. 1, pp. 14–25, July 2019. [Online]. Available: <https://doi.org/10.1145/3352020.3352024>
- [7] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, “A modular benchmarking infrastructure for high-performance and reproducible deep learning,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2019, the 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS’19). [Online]. Available: <https://doi.org/10.1109/ipdps.2019.00018>
- [8] “NVIDIA nvprof,” docs.nvidia.com/cuda/profiler-users-guide/index.html, accessed: 2019-5-04.
- [9] “NVIDIA Nsight System,” developer.nvidia.com/nsight-systems, accessed: 2019-5-04.
- [10] “Intel VTune,” software.intel.com/en-us/vtune, accessed: 2019-5-04.
- [11] R. Adolf, S. Rama, B. Reagen, G.-y. Wei, and D. Brooks, “Fathom: Reference workloads for modern deep learning methods,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE. IEEE, Sep. 2016. [Online]. Available: <https://doi.org/10.1109/iiswc.2016.7581275> pp. 1–10.
- [12] H. Zhu, M. Akrouf, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, “Benchmarking and analyzing deep neural network training,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE. IEEE, Sep. 2018, pp. 88–100.

- [13] S. Preview, “Scopus preview,” <https://www.scopus.com/>, accessed: 2019-10-17.
- [14] “Wolfram NeuralNet Repository,” <https://resources.wolframcloud.com/NeuralNetRepository/>, 2019, accessed: 2019-10-17.
- [15] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: arxiv.org/abs/1409.1556
- [16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2016. [Online]. Available: <https://doi.org/10.1109/cvpr.2016.308> pp. 2818–2826.
- [17] K. Weiss, T. M. Khoshgoftaar, and D. Wang, “A survey of transfer learning,” *J Big Data*, vol. 3, no. 1, p. 9, May 2016.
- [18] “TensorFlow Hub is a library for reusable machine learning modules ,” <https://www.tensorflow.org/hub>, accessed: 2019-10-17.
- [19] T. Elsken, J. H. Metzen, and F. Hutter, “Neural architecture search: A survey.” *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.
- [20] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, “FBNet: Hardware-aware efficient ConvNet design via differentiable neural architecture search,” *CoRR*, vol. abs/1812.03443, 2018. [Online]. Available: arxiv.org/abs/1812.03443
- [21] “Recommended CPU Instances,” docs.aws.amazon.com/dlami/latest/devguide/cpu.html, 2019, accessed: 2019-10-04.
- [22] S. Chintala, “ConvNet Benchmarks,” github.com/soumith/convnet-benchmarks, accessed: 2020-02-20.
- [23] C. Li, A. Dakkak, J. Xiong, and W.-M. Hwu, “Benanza: Automatic μ Benchmark Generation to Compute “Lower-bound” Latency and Inform Optimizations of Deep Learning Models on GPUs.” IEEE, May 2020, the 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS’20).
- [24] M. Hutton, J. Rose, and D. Corneil, “Automatic generation of synthetic sequential benchmark circuits,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 8, pp. 928–940, Aug. 2002.
- [25] “TensorFlow Profiler,” www.tensorflow.org/api_docs/python/tf/profiler, accessed: 2020-02-20.
- [26] “MXNet Profiler,” mxnet.incubator.apache.org/api/python/profiler/profiler.html, accessed: 2020-02-20.

- [27] “NVIDIA GPU-Accelerated Containers,” www.nvidia.com/en-us/gpu-cloud/containers/, accessed: 2020-02-20.
- [28] “NVIDIA Tools Extension,” docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx, accessed: 2020-02-20.
- [29] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu, “Across-stack profiling and characterization of machine learning models on GPUs,” *arXiv preprint arXiv:1908.06869*, 2019.
- [30] “NVIDIA CUPTI,” developer.nvidia.com/cuda-profiling-tools-interface, accessed: 2020-02-20.
- [31] “NVTX Plugins for Deep Learning,” github.com/NVIDIA/nvtx-plugins, accessed: 2020-02-20.
- [32] “Trace Context,” www.w3.org/TR/trace-context, accessed: 2020-02-20.
- [33] “OpenTracing: Cloud native computing foundation,” opentracing.io, 2019, accessed: 2019-10-04.
- [34] “Open Telemetry,” opentelemetry.io, accessed: 2020-02-20.
- [35] A. Pal and M. Pal, “Interval tree and its applications,” *Advanced Modeling and Optimization*, vol. 11, no. 3, pp. 211–224, 2009.
- [36] C. Li, A. Dakkak, J. Xiong, and W.-m. Hwu, “The design and implementation of a scalable dl benchmarking platform,” *arXiv preprint arXiv:1911.08031*, 2019.
- [37] “Jaeger: open source, end-to-end distributed tracing,” www.jaegertracing.io/, 2019, accessed: 2020-05-20.
- [38] “The Cloud Native Computing Foundation,” www.cncf.io, 2019, accessed: 2020-05-20.
- [39] “Amazon EC2 P3 Instances,” aws.amazon.com/ec2/instance-types/p3/, 2019, accessed: 2019-10-04.
- [40] “NVIDIA GPU Metrics Reference,” docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference, accessed: 2020-02-20.
- [41] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for floating-point programs and multicore architectures,” Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2009.
- [42] “NVIDIA cuDNN,” developer.nvidia.com/cudnn, 2019, accessed: 2019-10-04.
- [43] G. Guennebaud, B. Jacob et al., “Eigen v3,” eigen.tuxfamily.org, 2010.
- [44] “TensorFlow-Slim Image Classification Model Library,” github.com/tensorflow/models/tree/master/research/slim, accessed: 2020-02-20.

- [45] “TensorFlow Detection Model Zoo,” github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md, accessed: 2020-02-20.
- [46] “TensorFlow DeepLab Model Zoo,” github.com/tensorflow/models/blob/master/research/deeplab/g3doc/model_zoo.md, accessed: 2020-02-20.
- [47] “MXNet Gluon Model Zoo,” gluon-cv.mxnet.io/model_zoo/index.html, 2020, accessed: 2020-02-20.
- [48] “NVIDIA cuBLAS,” developer.nvidia.com/cublas, accessed: 2020-02-20. [Online]. Available: developer.nvidia.com/cublas
- [49] “ONNX: Open Neural Network Exchange,” onnx.ai, 2019, accessed: 2019-10-04.
- [50] “Neural Network Exchange Format (NNEF),” www.khronos.org/nnef, 2019, accessed: 2019-10-04.
- [51] “ONNX Model Zoo,” github.com/onnx/models, 2019, accessed: 2019-10-04.
- [52] J. Deng, J. Guo, and S. Zafeiriou, “ArcFace: Additive angular margin loss for deep face recognition,” *CoRR*, vol. abs/1801.07698, 2018. [Online]. Available: arxiv.org/abs/1801.07698
- [53] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.
- [54] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2015. [Online]. Available: <https://doi.org/10.1109/cvpr.2015.7298594> pp. 1–9.
- [55] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *CoRR*, vol. abs/1311.2524, 2013. [Online]. Available: arxiv.org/abs/1311.2524
- [56] G. Huang, Z. Liu, and K. Q. Weinberger, “Densely connected convolutional networks,” *CoRR*, vol. abs/1608.06993, 2016. [Online]. Available: arxiv.org/abs/1608.06993
- [57] P. Wang, P. Chen, Y. Yuan, D. Liu, Z. Huang, X. Hou, and G. W. Cottrell, “Understanding convolution for semantic segmentation,” *CoRR*, vol. abs/1702.08502, 2017. [Online]. Available: arxiv.org/abs/1702.08502
- [58] E. Barsoum, C. Zhang, C. Canton-Ferrer, and Z. Zhang, “Training deep networks for facial expression recognition with crowd-sourced label distribution,” *CoRR*, vol. abs/1608.01041, 2016. [Online]. Available: arxiv.org/abs/1608.01041

- [59] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: arxiv.org/abs/1502.03167
- [60] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” *CoRR*, vol. abs/1512.00567, 2015. [Online]. Available: arxiv.org/abs/1512.00567
- [61] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. [Online]. Available: <https://doi.org/10.1109/5.726791>
- [62] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient convolutional neural networks for mobile vision applications,” *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: arxiv.org/abs/1704.04861
- [63] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: arxiv.org/abs/1512.03385
- [64] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 630–645.
- [65] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An extremely efficient convolutional neural network for mobile devices,” *CoRR*, vol. abs/1707.01083, 2017. [Online]. Available: arxiv.org/abs/1707.01083
- [66] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, “SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size,” *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: arxiv.org/abs/1602.07360
- [67] J. Redmon and A. Farhadi, “YOLO9000: better, faster, stronger,” *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: arxiv.org/abs/1612.08242
- [68] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” *CoRR*, vol. abs/1311.2901, 2013. [Online]. Available: arxiv.org/abs/1311.2901
- [69] Onnx, “ONNX shape inference,” <https://github.com/onnx/onnx/blob/master/docs/ShapeInference.md>, 2019.
- [70] Google, “Google benchmark,” github.com/google/benchmark, 2014.
- [71] A. Anderson and D. Gregg, “Optimal DNN primitive selection with partitioned boolean quadratic programming,” in *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*, ACM. ACM Press, 2018. [Online]. Available: <https://doi.org/10.1145/3179541.3168805> pp. 340–351.

- [72] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning,” *CSUR*, vol. 52, no. 4, pp. 1–43, Aug. 2019. [Online]. Available: <https://doi.org/10.1145/3320060>
- [73] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011.
- [74] “NVIDIA DLProf,” <https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/>, accessed: 2019-5-04.
- [75] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, “MIOpen: An open source library for deep learning primitives,” 2019.
- [76] “Mkl-Dnn,” github.com/intel/mkl-dnn, 2019, accessed: 2019-10-04.
- [77] Microsoft, “ONNX runtime,” github.com/microsoft/onnxruntime, 2019.
- [78] A. Paszke, S. Gross, S. Chintala, and G. Chanan, “Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration,” vol. 6, 2017.
- [79] “ONNX Operator Schemas,” <https://github.com/onnx/onnx/blob/master/docs/Operators.md#Conv>, accessed: 2019-5-04.
- [80] Intel, “benchdnn,” github.com/intel/mkl-dnn/tree/master/tests/benchdnn, 2019.
- [81] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu, “Across-stack profiling and characterization of machine learning models on GPUs,” 2019.
- [82] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, “A survey on compiler autotuning using machine learning,” *CSUR*, vol. 51, no. 5, pp. 1–42, Sep. 2018. [Online]. Available: <https://doi.org/10.1145/3197978>
- [83] H. Vandierendonck, S. Rul, and K. De Bosschere, “The paralax infrastructure,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, IEEE. ACM Press, 2010. [Online]. Available: <https://doi.org/10.1145/1854273.1854322> pp. 389–399.
- [84] A. Haj-Ali, N. K. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica, “NeuroVectorizer: End-to-end vectorization with deep reinforcement learning,” *arXiv preprint arXiv:1909.13639*, 2019.
- [85] O. Solaris, “Oracle solaris studio code analyzer,” 2019.
- [86] K. Ng, M. Warren, P. Golde, and A. Hejlsberg, “The Roslyn project, exposing the c# and VB compiler’s code analysis,” *White paper, Microsoft*, 2011.
- [87] V. Sarkar, “Automatic selection of high-order transformations in the IBM XL FORTRAN compilers,” *IBM J. Res. & Dev.*, vol. 41, no. 3, pp. 233–264, May 1997. [Online]. Available: <https://doi.org/10.1147/rd.413.0233>

- [88] H. E. Plesser, “Reproducibility vs. replicability: A brief history of a confused terminology,” *Front. Neuroinform.*, vol. 11, p. 76, Jan. 2018. [Online]. Available: <https://doi.org/10.3389/fninf.2017.00076>
- [89] S. Ghanta, L. Khhermosh, S. Subramanian, V. Sridhar, S. Sundararaman, D. Arteaga, Q. Luo, D. Roselli, D. Das, and N. Talagala, “A systems perspective to reproducibility in production machine learning domain,” 2018.
- [90] L. Li and A. Talwalkar, “Random search and reproducibility for neural architecture search,” 2019.
- [91] A. Dakkak, C. Li, S. G. De Gonzalo, J. Xiong, and W.-m. Hwu, “Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 372–382.
- [92] “Initialization, Finalization, and Threads,” docs.python.org/3.6/c-api/init.html#sub-interpreter-support, 2020, accessed: 2020-02-28.
- [93] “Zenodo - Research. Shared,” www.zenodo.org, 2020, accessed: 2020-02-28.
- [94] “gRPC,” www.grpc.io, 2018, accessed: 2019-10-04.
- [95] V. V. Elisseev, M. Puzovic, and E. K. Lee, “A study on cross-architectural modelling of power consumption using neural networks,” *Supercomputing Frontiers and Innovations*, vol. 5, no. 4, pp. 24–41, 2018.
- [96] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, “TensorFlow-serving: Flexible, high-performance ML serving,” *arXiv preprint arXiv:1712.06139*, 2017.
- [97] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, “Clipper: A low-latency online prediction serving system.” in *NSDI*, 2017, pp. 613–627.
- [98] J. Tsay, T. Mummert, N. Bobroff, A. Braz, P. Westerink, and M. Hirzel, “Runway: machine learning model experiment management tool,” 2018.
- [99] G. Fursin, H. Guillou, and N. Essayan, “Codereef: an open platform for portable mlops, reusable automation actions and reproducible benchmarking,” *ArXiv*, vol. abs/2001.07935, 2020.
- [100] “Amazon Lambda,” <http://aws.amazon.com/lambda>, accessed: 2018-8-04.
- [101] “Azure Functions,” <https://azure.microsoft.com/en-us/services/functions>, accessed: 2018-8-04.
- [102] “Google Cloud Functions,” <https://cloud.google.com/functions>, accessed: 2018-8-04.

- [103] “IBM OpenWhisk,” <http://www.ibm.com/cloud-computing/bluemix/openwhisk>, accessed: 2018-8-04.
- [104] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, “Serverless computing: One step forward, two steps back,” 2018.
- [105] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, “A mixed-method empirical study of function-as-a-service software development in industrial practice,” *PeerJ PrePrints*, vol. 6, p. e27005v1, 2018.
- [106] “Google Cloud AI,” <https://cloud.google.com/products/machine-learning>, accessed: 2018-8-04.
- [107] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv preprint arXiv:1512.01274*, 2015.
- [108] NVIDIA, “Tensor Cores,” <https://www.nvidia.com/en-us/data-center/tensorcore>, accessed: 2020-05-04.
- [109] Google, “Google Cloud TPU,” <https://cloud.google.com/tpu>, accessed: 2020-05-04.
- [110] WikiChip, “Cascade Lake - Microarchitectures - Intel,” <https://en.wikichip.org/wiki/intel/microarchitectures/cascade.lake>, accessed: 2020-05-04.
- [111] Apple, “A11 Bionic,” <https://www.apple.com/iphone-x>, accessed: 2020-05-04.
- [112] Arm, “Arm Machine Learning Processor,” <https://developer.arm.com/products/processors/machine-learning/arm-ml-processor>, accessed: 2020-05-04.
- [113] Z. Du, S. Liu, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, Q. Guo, X. Feng, Y. Chen et al., “An accelerator for high efficient vision processing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 2, pp. 227–240, 2017.
- [114] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., “In-datacenter performance analysis of a tensor processing unit,” in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 1–12.
- [115] B. Reagen, R. Adolf, P. Whatmough, G.-Y. Wei, and D. Brooks, “Deep learning for computer architects,” *Synthesis Lectures on Computer Architecture*, vol. 12, no. 4, pp. 1–123, 2017.
- [116] Y. Zhu, M. Mattina, and P. Whatmough, “Mobile machine learning hardware at arm: A systems-on-chip (soc) perspective,” *arXiv preprint arXiv:1801.06274*, 2018.
- [117] Google, “Edge TPU,” <https://cloud.google.com/edge-tpu>, accessed: 2020-5-04.

- [118] P. Pärssinen, “Modern mobile graphics processors,” *Science: Internet, Data and Things (CS-E4000)*, Spring 2018, p. 211.
- [119] Lawrence Livermore National Laboratory, “Sierra Supercomputer,” <https://computation.llnl.gov/computers/sierra>, accessed: 2020-05-20.
- [120] Oak Ridge National Laboratory, “Summit Supercomputer,” <https://www.olcf.ornl.gov/summit>, accessed: 2020-05-20.
- [121] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, “Accelerating reduction and scan using tensor core units,” in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 46–57.
- [122] G. E. Blelloch, M. A. Heroux, and M. Zagha, “Segmented operations for sparse matrix computation on vector multiprocessors,” Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993.
- [123] T. M. Chan, “More algorithms for all-pairs shortest paths in weighted graphs,” *SIAM Journal on Computing*, vol. 39, no. 5, pp. 2075–2089, 2010.
- [124] M. D. McCool, A. D. Robison, and J. Reinders, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.
- [125] NVIDIA, *CUTLASS*, <https://devblogs.nvidia.com/cutlass-linear-algebra-cuda>, accessed: 2020-05-20.
- [126] C. Pearson, A. Dakkak, S. Hashash, C. Li, I.-H. Chung, J. Xiong, and W.-M. Hwu, “Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects,” in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering - ICPE '19*, ACM. ACM Press, 2019. [Online]. Available: <https://doi.org/10.1145/3297663.3310299> pp. 209–218.