Across-Stack Profiling and Characterization of Machine Learning Models on GPUs

Cheng Li*, Abdul Dakkak*
University of Illinois, Urbana-Champaign
{cli99,dakkak}@illinois.edu

Jinjun Xiong
IBM T. J. Watson Research Center
jinjun@us.ibm.com

Wei Wei, Lingjie Xu
Alibaba Group
{w.wei,lingjie.xu}@alibaba-inc.com

Wen-mei Hwu
University of Illinois, Urbana-Champaign
w-hwu@illinois.edu

ABSTRACT

The world sees a proliferation of machine learning/deep learning (ML) models and their wide adoption in different application domains recently. This has made the profiling and characterization of ML models an increasingly pressing task for both hardware designers and system providers, as they would like to offer the best possible computing system to serve ML models with the desired latency, throughput, and energy requirements while maximizing resource utilization. Such an endeavor is challenging as the characteristics of an ML model depend on the interplay between the model, framework, system libraries, and the hardware (or the HW/SW stack). A thorough characterization requires understanding the behavior of the model execution across the HW/SW stack levels. Existing profiling tools are disjoint, however, and only focus on profiling within a particular level of the stack.

This paper proposes a leveled profiling design that leverages existing profiling tools to perform across-stack profiling. The design does so in spite of the profiling overheads incurred from the profiling providers. We coupled the profiling capability with an automatic analysis pipeline to systematically characterize 65 state-of-the-art ML models. Through this characterization, we show that our across-stack profiling solution provides insights (which are difficult to discern otherwise) on the characteristics of ML models, ML frameworks, and GPU hardware.

1 INTRODUCTION

Recently there has been numerous impressive advances from machine learning/deep learning (ML) models in solving problems from many domains such as: image classification, object detection, machine translation, etc. This has resulted in a surge of interest in deploying these models within various hardware computing platforms/devices including commodity servers, accelerators, reconfigurable hardware, mobile and edge devices, and ASICs. Therefore, there is an increasing need for hardware computing providers (such as cloud providers), computer architects, and system/chip designers to profile and understand these ML models across these computing platforms/devices, and measure their accuracy, throughput, latency, and system resource utilization (memory, bandwidth, etc.). Such an endeavor is, however, greatly hampered, if not impossible, in the current ML landscape. The reasons are multi-fold:

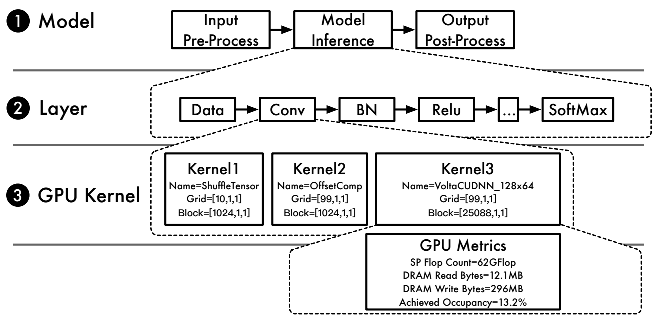


Figure 1: Model-, layer-, and GPU kernel-level profiling for the MLPerf_ResNet50_v1.5 model (Table 6) on System 2 (Table 5) with batch size 256 using NVIDIA's NGC TensorFlow v19.06 container. The model layers executed are data (Data), convolution (Conv), batch normalization (BN), relu (Relu), etc. The 3 GPU kernels from the first Conv layer are highlighted along with the GPU metrics from the third kernel.

- The number of ML models and frameworks is proliferating at an unprecedented pace because of great interest from the community, At the same time, the number of computing hardware platforms/devices interested in running these models and frameworks has also increased. The combinatorial possibilities among hardware, frameworks, and models have rendered the traditional manual process of understanding models not scalable.
- The ML models themselves are complicated and involve a stack of HW/SW components. An example is shown in Figure 1. At the model-level, there exists an evaluation pipeline. Components at the model-level include input pre-processing, model inference, and output post-processing. Stepping within the model inference, we find layer-level components, or layer nodes within the ML model, such as: convolution (Conv), batch normalization (BN), and Softmax. Within each layer are the GPU kernel-level components, a series of library calls or computation kernels that are invoked by the layers. Depending on the analysis needs, different metrics may be collected at each of these hierarchy levels.
- Existing profiling tools provide a partial view to the model's execution. For example, to characterize model latency, users insert timing code around the model inference stage of the evaluation pipeline. To capture the layer-level information, users use the ML framework's profiling capabilities [13, 25]. And, to capture GPU

*The two authors contributed equally to this paper.

kernel information (such as latency, number of flops or memory reads and writes), users have to use GPU profilers such as NVIDIA Visual Profiler [18] (NVVP), nvprof [19] or Nsight [14]. The same is true for CPU level hardware metrics or counters. It is difficult to stitch and correlate the results from these disjoint profiling tools and get a consistent across-stack timeline.

To address the above issues, in this paper, we first argue for the importance of a consistent across-stack ML model profiling and characterization methodology for both system providers and hardware designers. We then propose a consistent across-stack level profiling design along with an experimentation methodology. We show how the design copes with profiling overhead and accurately captures model-, layer-, and GPU-level profiles. Through our design, we enable a smooth hierarchical step-through of the profiling levels and require no framework modifications. We implement our design methodology within MLModelScope [35], a scalable open-source ML model evaluation platform, and show how across-stack profiling is performed. Our design requires minimal modifications to the MLModelScope runtime, and is flexible and extensible to allow further integration with other profiling tools.

We conduct extensive experimentation and demonstrate how our design and methodology enables users to easily introspect model performance at different levels of the HW/SW stack, identify bottlenecks, and systematically compare models, frameworks, and system offerings. Our experiments use 65 state-of-the-art ML models from MLPerf Inference v0.5 [11], AI-Matrix [8], and TensorFlow and MXNet model zoos [12, 23, 24, 26]. These models are run on 5 representative systems which span the past 4 GPU generations: Turing, Volta, Pascal, and Maxwell. We describe how to analyze the profile results and present 15 types of automated analysis to characterize ML models and the interplay between models and computing systems. With these analysis, we gain insights that would otherwise be difficult to discern without our methodology.

We supplement the paper with a website (aspc19.mlmodelscope.com) which contains the output of our automated analysis. The reader is encouraged to visit the website to view the raw data and get characterizations of the models that we were unable to highlight in detail in the paper due to the page limit.

The rest of the paper is organized as follows. Section 2 describes the current profiling tools and benchmarking efforts within the ML and system communities. Section 3 presents our design methodology and implementation. As a demonstration, Section 4 showcases 15 types of automated analysis that can be performed. Section 5 further evaluates 65 ML models and presents some insights that are enabled by our design. Section 6 concludes this paper.

2 BACKGROUND AND RELATED WORK

ML profiling efforts for GPUs characterize model performance at different levels of the HW/SW stack — model-, layer-, and GPU kernel-levels. Figure 1 illustrates the profiling levels by evaluating the MLPerf_ResNet50_v1.5 model (Table 6). ❶ **Model-level** profiling is used to measure the stages within a model evaluation pipeline. It captures the model inference latency or throughput. ❷ **Layer-level** profiling measures the layers executed by the ML framework. The layer index, latency, or memory allocation are all recorded as part of the layer-level profile. Finally, the ❸ **GPU Kernel-level** profiling measures GPU kernels executed during model evaluation along with the kernel names, latency, and GPU metrics. Currently, users leverage different tools [18, 19, 44, 51] and methods to profile and characterize model performance at the different levels.

There are corresponding efforts to develop benchmarks to measure ML models at these different levels. For model-level benchmarking, there is an active effort (spearheaded by both research and industry) to develop benchmark suites [11, 28, 32, 36] to characterize and compare models across systems. These benchmark suites define a representative set of models, along with the datasets and target accuracy, and provide scripts to run these models. To profile these benchmarks users need to manually insert code to time specific evaluation stages such as model loading, input pre-processing, model inference, and output post-processing. Some common analysis performed at the model-level are model end-to-end latency and throughput under specific workload scenarios. The evaluation results are then used as reference points to compare models or systems, and the results are sometimes curated to provide a scoreboard of model performance across systems. These benchmark suites are similar in spirit to SPEC [40, 43] for CPUs and GPUs and TPC [49] for databases.

Model-level characterization takes the framework as a black box, and thus does not identify inefficiencies within a framework that affect model execution performance. When one needs to examine components with a model (i.e. layer-level granularity) they turn to the framework profilers [13, 21, 25]. These framework profilers are either built-in to the framework or are community contributed framework plugins. They are used to monitor the layers executed by the framework and measure their performance. To profile framework layers, one explicitly enables the framework's profiler options during execution. The framework then generates a profile output which contains information such as layer latency and memory usage, and is used by the user to identify straggling layers.

While framework profilers can tell the most time-consuming layer, they do not provide low level details such as the system library calls being executed, the kernels invoked, and the system metrics throughout the execution. Researchers, especially library developers and system architects, who are interested in model and system interaction use system- and hardware-level profilers to capture such low level details. For GPUs, tools such as NVIDIA Visual Profiler [18] (NVVP), nvprof [19] or Nsight [14] are commonly used to capture the model performance at the GPU kernel level. These tools leverage NVIDIA's CUDA Profiling Tools Interface (CUPTI) [7] library to capture information such as GPU execution trace, CUDA API calls, and CUDA kernel metrics. Typically, researchers run the models using these profiling tools, extract the K most time-consuming kernels, and then run the tools again (using different options) to drill down for more detailed information about the kernels of interest. Because of the low level nature of these profilers, their results cannot be correlated back to layer- or model-level profiles. Instead, the low level profilers suggest users to insert NVTX [2, 3] markers within the framework source code or user code. For example, NVIDIA's TensorFlow NGC [16] containers provide a modified version of the TensorFlow framework that allows users to encapsulate layers with NVTX markers. Using the NGC container requires users to modify their inference code,

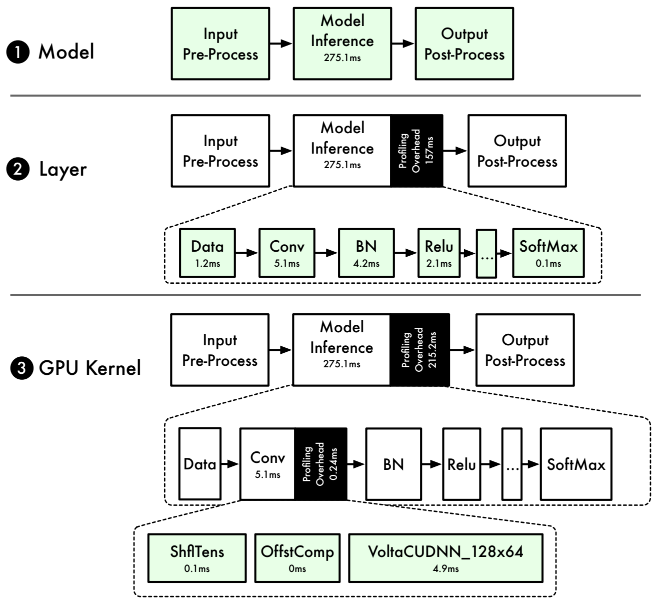


Figure 2: The model-, layer-, and GPU-level profiling gives detailed account of the execution of a model. By correlating layers or kernels across the profiling hierarchy, one can understand the added overheads. At each level, the green components correctly measure their latency whereas the rest incur added overheads.

rely on the profiler's format for analysis, and does not work with a "vanilla" TensorFlow distribution.

To demonstrate the GPU kernel-level analysis, consider the MLPerf_ResNet50_v1.5 model in Figure 1. 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`). We can even tell how many times this kernel is called (34 times through the entire evaluation as described in Section 4). However, due to lack of correlation between the GPU kernels and the layers, no other useful analysis can be performed. For example, it is hard to correlate the `volta_scudnn_128x64_relu_interior_nn_v1` kernel to a specific layer within the model. Knowing the correlation between layers and GPU kernels enables more meaningful analysis of the model performance. We describe how we remidy this in Section 4; where our analysis can tell precisely that only 3 kernel calls are due to 208th layer.

To overcome the unknown correlation between layers and GPU kernels, there have been efforts, mainly by system and framework developers, to develop fine-grained microbenchmarks of representative layers [29,31,33,54]. These microbenchmarks tend to target convolution or RNN layers and are purposely built for algorithm developers, compiler writers, and system researchers. Since these microbenchmarks do not consider how layers are scheduled within a framework (e.g. overlapping) or any optimization that a framework performs on the model graph (e.g. layer fusion), they only serve as a lower-bound approximation of a layer's real-world performance. Recent benchmark suites take a multi-tier approach [8,30,53], whereby they provide a collection of benchmarks that cover both end-to-end model and layer benchmarking.

Given the above, we believe a profiling methodology that captures ML model characteristics at model-, layer-, and GPU kernel-levels — coupled with automated analysis of the results — would boost the productivity of researchers and help understand the model/system performance and identify the bottlenecks. And, with the exception of manual work, the authors are unaware of any methodology that performs the aforementioned across-stack profiling.

3 ACROSS-STACK PROFILING

This section presents an across-stack profiling design that leverages distributed tracing to capture and aggregate profiles from different profiling providers. We also show how our design is general and extensible.

3.1 Leveled Profiling Design

The across-stack profiling leverages results from different existing profiling tools, libraries, or methods (referred to as *profiling providers*). To enable these profiling providers to publish into the same profile timeline, we use an off-the-shelf distributed tracer [27, 52]. By leveraging the distributed tracer, originally designed to monitor distributed applications, especially those built from microservices [37,46], we are able to add instrumentation code (referred to as *trace points*) within existing applications and library codes. The trace points store the time stamps, a cause/effect (or parent/child) relationship, user-defined contexts for correlation, and other user-defined metadata. The start and end trace point pair (called *span* [22]) represents an interval of time within the tracing timeline. We capture profiles from different profiling providers and convert them into spans. To construct the parent-child relationship between spans, we either assign a span to its parent or, if not possible, during the profile analysis build an interval tree [47] of all time stamps. Using the interval tree, we reconstruct the parent-child relationship by checking for interval set inclusion (if the interval span s_1 contains the interval span s_2 then s_1 is a parent of s_2). The profiles are placed within a single timeline (referred to as *trace*) with the adjusted parent-child relationship.

While leveled profiling design scheme presented above is general, this paper focuses only on how it can be used to profile ML models on GPUs. Specifically, we explain how we collect the model-, layer-, and GPU kernel-level profile information from the different profiling providers.

Model-level profiling— To profile at model granularity, we place tracing points within the evaluation pipeline. For example, to measure the time spent running the model inference for C APIs, one places tracing points around the calls to `TF_SessionRun` for TensorFlow or `MXPredForward` for MXNet. This does not require changes to the framework code, but requires adding an extra line to the model evaluation code.

Layer-level profiling— To profile at layer granularity, we leverage the ML frameworks' existing profiling capabilities. In TensorFlow, enabling layer profiling requires calling the framework's inference function with the profiling option enabled. For TensorFlow, this option is controlled by the `RunOptions.TraceLevel` setting which is passed to the `TF_SessionRun` function. In MXNet, the

`MXSetProfilerState` function enables and disables layer profiling. Similar mechanisms exist for other frameworks such as Caffe [42], Caffe2 [41], PyTorch [48], and TensorRT [17]. The framework's profile representation is converted into a sequence of spans and are then published to the tracer server. The layer spans are set to be children of the model inference span, and hence each layer can be correlated to the model inference stage. Since we leverage the existing framework's profiling capabilities, profiling at layer-level does not require modification to the framework source code and only requires minor modification to the evaluation code.

GPU Kernel-level profiling— To obtain the GPU profile, we leverage NVIDIA's CUPTI library [7]. The CUPTI library captures the CUDA runtime API (CUDA host functions), 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) that occur throughout the model evaluation process. As with existing GPU profilers, one can specify which CUDA runtime API, GPU activities, or metrics to capture. The captured GPU profile is translated into spans and published to the tracer server. The GPU kernel metrics, if specified, are added as metadata to the corresponding kernel span. The GPU-level profiling requires small modification to the MLModelScope evaluation code.

Since CUDA kernels are often launched asynchronously by the ML frameworks or libraries, we use both CUPTI's callback and activity mechanisms to capture the GPU kernel information. The CUPTI callback API captures both the CUDA runtime and driver API calls, along with the kernel launch time. The CUPTI activity API captures the effective kernel duration. Therefore, for each kernel two spans are created within our timeline and are correlated by the `correlation_id` provided by CUPTI. During post-processing, we use the `correlation_id` to aggregate information from the kernel spans. We then build an interval tree and use the kernel time stamp to correlate it with the parent layer span.

3.2 Profiling Overhead and Leveled Experimentation

We observe that using a distributed tracer adds negligible overhead per span (tiny spans are usually not of interest). Other than that, the profiling design incurs the same overheads introduced by the profiling providers. For eexample, the layer-level profiling adds overhead to the model inference depending on how many layers are executed. As with existing NVIDIA profilers, the GPU-level profiling incurs overhead, which can be substantial. Profiling GPU kernel metrics, for example, can slow down execution by over 100× (depending on the metric type with memory metrics being especially expensive to profile). This is due to the limited number of GPU hardware performance counters, which require GPU kernels to be run multiple times to capture the user-specified metrics. Since the across-stack profiling design allows one to enable or disable profiling providers depending on the characterization target, the profiling overhead can be controlled by choosing the profiling level. We refer to the evaluation practice which makes use of traces from different profiling levels as *leveled experimentation.*

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 1: The layer information for the top 5 most time consuming layers summarized from the layer profile. In total, there are 234 layers of which 143 take less than 1 ms.

To demonstrate the profiling overhead and the leveled experimentation, again consider the MLPerf_ResNet50_v1.5 model running on System 2 (in Table 5) with batch size 256. Figure 2 shows the latency of the components at different profiling levels. We can enable model-level profiling to get the baseline model latency of 275.1*ms*. To capture the latency of each layer, we enable layer-level profiling. While the layer-level profiling affects the model inference latency, it accurately captures the latency for each layer. We can quantify this overhead by subtracting the model-latency in the model-level profiling trace from the model-latency in the layer-level profiling trace. We find that the layer-level profiling introduces a 57% overhead. We can drill down further to perform the GPU kernel-level profiling. Enabling the GPU kernel-level profiling adds extra overhead to the model inference latency, making it 214.2*ms* at this profiling level. If we look at the first convolution layer, the GPU profiling of the 3 child kernels incurs a 0.24*ms* overhead. The kernel latencies recorded match what's reported by NVIDIA's nvprof tool.

3.3 Extensibility of Design

Care was taken to ensure the extensibility of the design. Our design allows other profiling tools or methods to be integrated by either using the common tracing interface or registering the other profilers as observers [50] to each span. For example, one can register an observer on spans to measure ML library calls, such as cuDNN [15] and NCCL [45], and publish their execution records to the tracer. Similarly, one can add other system and hardware profilers.

3.4 Integration within MLModelScope Runtime

We integrated the profiler within MLModelScope, an open-source framework and hardware agnostic, extensible, and customizable framework for evaluating ML models at scale. For distributed tracing, we use Jaeger [10] — a production grade [34] distributed tracing library. MLModelScope uses the frameworks' C-level API directly to avoid 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 MLModelScope to use the CUPTI library.

We also modified the user interface of MLModelScope. Users control the profiling granularity (model, layer, GPU API and activity, GPU metrics) of the model evaluation through MLModelScope's command line, library, or web interface. We also added a profile ingestion pipeline within MLModelScope which is described in detail in Section 4.

4 ACROSS-STACK CHARACTERIZATION

We design a post-processing and automated analysis pipeline which consumes the profiling traces and output across-stack characteristics of ML models. The across-stack profiling correlates model-, layer- and GPU kernel-level profiles into a single trace. Since meaningful characterization requires multiple runs, the pipeline takes traces from multiple evaluations, correlate the information, and compute the trimmed mean value (or other user-defined statistical summary) for the same performance value (e.g. latency) across runs. We implement the 15 analysis described in this section within MLModelScope and add these analysis as command-line or website options. These analysis can be performed either offline or online during MLModelScope's evaluation process.

We group the 15 analysis into three categories based on the characterization level or granularity. To illustrate the analysis, we use TensorFlow MLPerf_ResNet50_v1.5 (Model 7 in Table 6) from the MLPerf Inference's v0.5 release [11]. The MLPerf_ResNet50_v1.5 model is run within the NVIDIA's NGC [16] TensorFlow container v19.06 on an Amazon P3 [9] instance (System 2 in Table 5). 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 and 4.3, since the model achieves maximum throughput at that batch size.

With our design, we are able to perform characterizations that are either impossible or hard using existing tools or methods. To show this, we will also discuss how the analysis can be performed absent our methodology. We refer the reader to aspc19.mlmodelscope.com which contains example output of the automated characterizations.

4.1 Using Model Profile

Both model throughput and latency are of great importance to researchers and developers who want to understand a model's end-to-end behavior when it it deployed. Leveraging only model-level profiling, we automate the computation of the ❶ model's throughput and latency. Using this data, we can compute the optimal batch size given a latency target. We define the optimal batch size as the batch size where doubling it does not increase the current throughput by more than 5%. Absent our methodology, researchers and developers manually insert timing code around the model inference code. Researchers and developers then perform multiple evaluations to measure and select the optimal batch size.

To demonstrate the model-level analysis, we run MLPerf_ResNet50_v1.5 across batch sizes and compute both its throughput and latency. As can be seen in Figure 3, the throughput saturates at batch size 256 (optimal batch size) and the model achieves a maximum throughput of 930.7 images/second. The batch latency (shown in Tables 8 and 6) for the model is 275.05ms.

4.2 Using Model and Layer Profiles

Leveraging both the model- and layer-level profiles enables us to characterize the layers executed by the frameworks. These layers may be different from the ones statically defined by the programmer, since the framework may apply optimizations such as layer fusion or other graph optimizations. Using the data captured, we can generate ❷ a layer information table reporting layer index, name, shape, latency, and allocated memory. For example, Table 1 shows the top 5 most time consuming layers for MLPerf_ResNet50_v1.5, noting that they all are convolution layers.

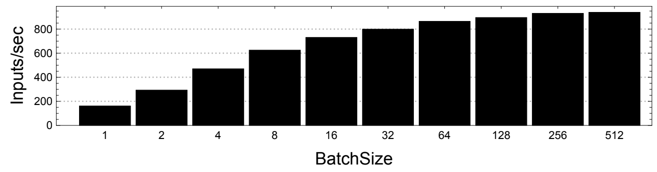


Figure 3: The MLPerf_ResNet50_v1.5 model throughput across batch sizes on System 2 (in Table 5).

We can also use the model and layer profile data to visualize both the ❸ latency per layer (Figure 4a) and ❹ allocated memory per layer (Figure 4b) in the order the layers are executed. Figures 4a and 4b show a latency and memory allocation trend exists for layer execution. For example, we observe that the latency of a model 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 for the middle and end stages.

We can also group the information by layer type (or the layer operators executed) to derive useful layer execution statistics such as: ❺ the number of times each layer type is executed (Figure 5a), ❻ the aggregated latency by layer type (Figure 5b), and ❼ the aggregated allocated memory by layer type (Figure 5c). We observe that MLPerf_ResNet50_v1.5 mostly consists of `Add`, `Conv2D`, `Mul`, and `Relu` layers. This is because of the ResNet modules [39] which have the pattern of `Convolution` → `Batch Norm` → `Relu`. The ResNet modules get executed by TensorFlow as a `Conv2D` → `Mul` → `Add` → `Relu` layer sequence. We can also see that this same group of layers dominate both latency and memory allocation, with `Conv2D` being the most time consuming layer type.

Current practice for layer-level analysis — Absent our methodology, users use the framework profiler to gather layer-level profile information. Since the framework profiler's output is framework dependent, it means that users either have to use the framework's profiling format to perform the analysis (making the analysis framework specific) or have to convert the framework profile output into a common profile format (our methodology). In some cases, a framework may not provide the ability to insert user markers, such as recording the model's latency or pre-processing time within the profile output. Therefore, users need to write scripts to merge the profile captured by the model-level profiling into the framework's profile output. Furthermore, to gather statically meaningful results, users needs to run the evaluation multiple times and perform layer-wise correlation across runs. Except for manual work, the authors are not aware of any methodology that performs the aforementioned layer-level analysis.

4.3 Using Model, Layer, and GPU Profiles

To distill fine-grained performance information, we perform the analysis using the data captured from the model-, layer- and GPU kernel-level profiles. ❽ A summary characterizing the model's GPU kernel execution is computed. Table 2 shows the top 5 most time consuming GPU kernels for the MLPerf_ResNet50_v1.5 model, which either perform matrix multiplication or convolution. The

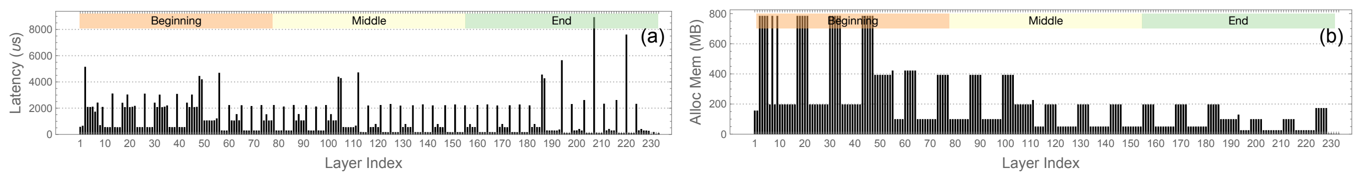
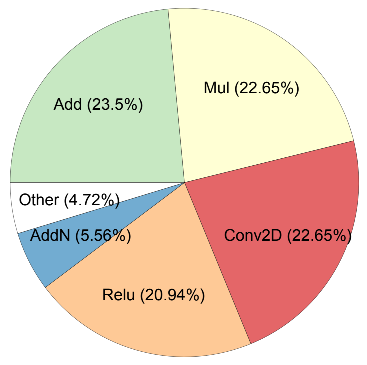
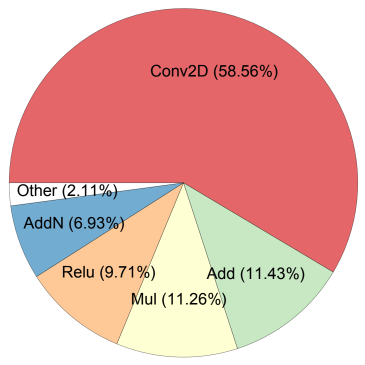


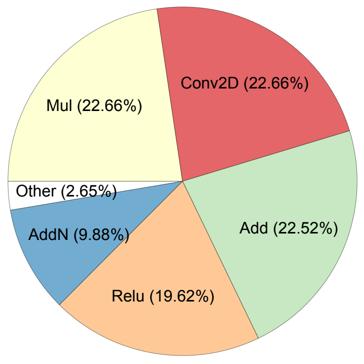
Figure 4: The (a) latency and (b) allocated memory for each layer for MLPerf_ResNet50_v1.5 with batch size 256 on System 2. To understand the performance trend throughout the model evaluation process, we subdivide the timeline into 3 intervals: beginning, middle, and end.



(a) Distribution of layers.



(b) Aggregated latency for layers.



(c) Aggregated memory allocation for layers.

Figure 5: Layer statistics for the MLPerf_ResNet50_v1.5 model with batch size 256 on System 2.

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	✗
volta_cgemm_32x32_tn	208	6.03	77.42	43.93	43.81	12.19	841.59	12.83	✗
volta_scudnn_128x128_relu_interior_nn_v1	195	5.48	59.20	27.71	8.40	15.49	1,563.30	10.80	✗
volta_scudnn_128x64_relu_interior_nn_v1	3	4.91	62.89	11.55	283.05	13.20	203.58	12.81	✗
volta_scudnn_128x128_relu_interior_nn_v1	57	4.56	59.24	34.83	37.64	15.15	779.55	12.99	✗

Table 2: The GPU kernel information for the top 5 most time consuming kernels along with their layer mapping. In total, 375 GPU kernel are invoked of which 284 take less than 1*ms*.

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	✗
Eigen::TensorCwiseBinaryOp<scalar_product_op>	52	28.43	10.33	2.85	4,181.23	6,371.12	49.72	0.26	0.10	✓
Eigen::TensorCwiseBinaryOp<scalar_sum_op>	51	26.38	9.59	2.64	4,063.49	6,052.22	49.69	0.25	0.10	✓
Eigen::TensorCwiseBinaryOp<scalar_max_op>	48	24.71	8.98	0	3,773.84	5,699.95	98.39	0	0	✓
volta_scudnn_128x128_relu_interior_nn_v1	4	23.02	8.37	276.64	671.68	335.01	15.96	262.08	12,02	✗

Table 3: The GPU kernel information aggregated by name for the top 5 most time consuming kernels. In total, 30 unique GPU kernels are invoked.

GPU metrics for each kernel can also be captured. Although all metrics supported by the NVIDIA profiling tools [1] can be captured, this paper focuses on `flop_count_sp`, `dram_read_bytes`, `dram_write_bytes`, and `achieved_occupancy`:

- `flop_count_sp` — total number of single-precision floating-point operations executed by a kernel.
- `dram_read_bytes` — total number of bytes read from the GPU's DRAM to its L2 cache.
- `dram_write_bytes` — the total number of bytes written from the GPU's L2 cache to its DRAM.
- `achieved_occupancy` — the ratio of the average active warps per active cycle to the maximum number of warps per streaming

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: GPU kernel information aggregated by layer for the top 5 most time consuming layers.

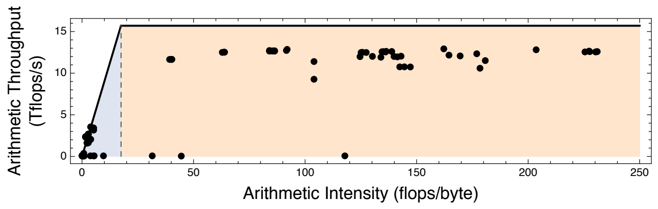


Figure 6: The roofline analysis for all the kernels in MLPerf_ResNet50_v1.5 GPU with batch size 256 on System 2, which has an ideal arithmetical intensity of $17.44 flops/byte$. Kernels within the blue region are memory bound, whereas the ones within the orange region are compute bound.

multiprocessor (SM). The `achieved_occupancy` is an indicator of level of parallelism within a kernel.

Using both the kernel flops and memory accesses metrics, we calculate the kernel arithmetic intensity and arithmetic throughput. These parameters are used for the GPU roofline [55] analysis. A kernel's arithmetic intensity is the ratio between the number of flops to the number of memory accesses. It is computed by:

$$\texttt{arithmetic\_intensity} = \frac{\texttt{flop\_count\_sp}}{\texttt{dram\_read\_bytes} + \texttt{dram\_write\_bytes}}$$

A kernel's arithmetic throughput is the ratio between the number of flops to the latency. It is computed by:

$$\texttt{arithmetic\_throughput} = \frac{\texttt{flop\_count\_sp}}{\texttt{kernel latency}}$$

Using the GPU's theoretical FLOPS and memory bandwidth, we compute the ideal arithmetic intensity using the equation:

$$\texttt{ideal\_arithmetic\_intensity} = \frac{\texttt{peak\_FLOPS}}{\texttt{memory\_bandwidth}}$$

The Tesla V100-SXM2, for example, has a peak Flops of 15.7 TFlops and a global memory bandwidth of 900 GB/s, hence an ideal arithmetic intensity of $\frac{15.7\ TFlops}{900\ GB/s} = 17.44$ flops/byte. A kernel is *memory-bound* if its arithmetic throughput is less than the GPU's ideal arithmetic intensity and is *compute-bound* otherwise.

Leveraging the GPU-level profile, we can perform a ❾ roofline analysis for all the GPU kernels executed within the MLPerf_-ResNet50_v1.5 model (shown in Figure 6). Using Figure 6, we observe that the most time consuming kernels are convolution kernels, and all the convolution kernels are within the orange region (i.e. compute-bound).

We can aggregate the information of GPU kernels by name, and create a table of ❿ aggregated GPU kernel information, as shown in Table 3. 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 [15] library) and it to be compute-bound — taking 30.87% of the overall model inference latency. The 2nd and 3rd most time consuming kernels are `scalar_product_op` and `scalar_sum_op` from the Eigen [38] library, which are memory-bound, and take 10.33% and 9.59% of the model inference latency respectively.

Since each kernel can be correlated to the layer that invokes it, we can aggregate the information of GPU kernels within each layer and build a table of ⓫ GPU kernel information aggregated by layer (shown in Table 4). A layer's kernel latency, flops, DRAM reads and writes are calculated by adding the corresponding values of the kernels invoked by that layer. The layer's achieved occupancy is calculated as the weighted sum (using kernel latency as the weight) of the achieved occupancy of all the kernels within the layer. Using this data, we visualize the ⓬ total flops, DRAM reads and writes per layer (shown in Figures 7 (a), (b) and (c) respectively). The layer kernel latency can be subtracted from the layer latency to compute ⓭ the time not spent within GPU computation (shown in Figure 8). We call this difference the layer's *CPU latency*. The layer arithmetic intensity and throughput are calculated using the layer's total lops and memory access values. A ⓮ roofline analysis for all the layers can be performed (shown in Figure 9). With the roofline analysis, we observe that the `Conv2D` layers are the most compute and memory intensive for the MLPerf_ResNet50_v1.5 model. We also observe that `Conv2D`, `MatMul`, `BiasAdd`, and `Softmax` layers are compute-bound, whereas the other layers (`Add`, `Mul`, and `Relu`) are memory-bound.

We aggregate all the kernel information within a model to compute the GPU kernel latency, flops, and memory access information (shown in Table 7 across batch sizes) for the model. 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. The model's achieved occupancy is calculated as the weighted sum of the achieved occupancy of all the kernels invoked. Model arithmetic intensity and throughput are calculated using the model's total flops and memory accesses. This information is used to ⓯ classify the entire model as either compute- or memory-bound (see Table 8 and Figure 14 for example).

Figure 11 shows the roofline analysis for the MLPerf_ResNet50_-v1.5 model across batch sizes. We observe that the model is compute-bound except for batch sizes 16 and 32 where it is memory-bound. We use the data in Tables 1, 2, and 3 to explain this phenomenon. We observe that the kernels invoked sometimes vary across batch sizes. This is due to the cuDNN library, which contains multiple algorithms to perform convolution and relies on heuristics to choose the optimal algorithm (based on the running system and input parameters). For batch sizes less than 16, the `CUDNN_CONVOLUTION_`

Name	CPU	GPU	GPU Architecture	Theoretical Flops (TFlops)	Memory Bandwidth (GB/s)	Ideal Arithmetic Intensity (flops/byte)
System 1	Intel Xeon E5-2630 v4 @ 2.20GHz	Quadro RTX 6000	Turing	16.3	624	26.12
System 2 (EC2 P3)	Intel Xeon E5-2686 v4 @ 2.30GHz	Tesla V100-SXM2-16GB	Volta	15.7	900	17.44
System 3	Intel Xeon E5-2682 v4 @ 2.50GHz	Tesla P100-PCIE-16GB	Pascal	9.3	732	12.70
System 4	Intel Xeon E5-2682 v4 @ 2.50GHz	Tesla P4	Pascal	5.5	192	28.34
System 5 (EC2 G3)	Intel Xeon E5-2686 v4 @ 2.30GHz	Tesla M60	Maxwell	4.8	160	30.12

Table 5: Five systems with Turing, Volta, Pascal, and Maxwell GPUs are selected for evaluation. All evaluations are run within NVIDIA's TensorFlow NGC v19.06 and MXNet NGC v19.06 docker containers. We calculate the ideal arithmetic intensity of each system using the theoretic Flops and memory bandwidth reported by NVIDIA.

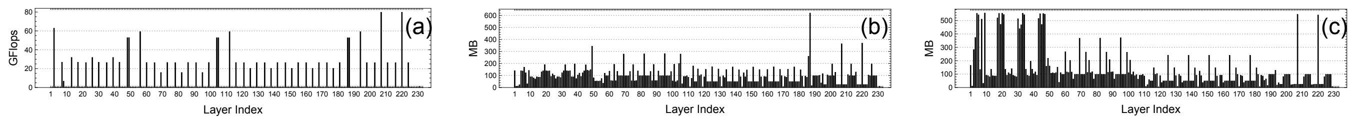


Figure 7: The (a) total GPU kernel flops, (b) GPU DRAM reads, and (c) GPU DRAM writes per layer for the MLPerf_ResNet50_-v1.5 model with batch size 256 on System 2.

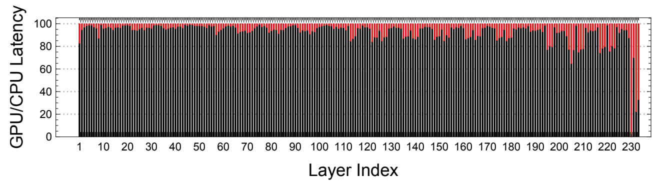


Figure 8: The MLPerf_ResNet50_v1.5 normalized GPU and CPU latency per layer for batch size 256 on System 2.

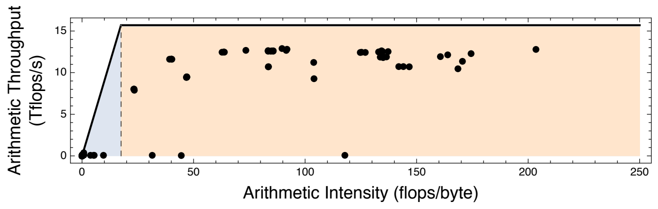


Figure 9: The roofline analysis for all the layers in MLPerf_-ResNet50_v1.5 for batch size 256 on System 2.

`FWD_ALGO_IMPLICIT_GEMM` algorithm is invoked by the cuDNN library (kernel `cudnn::detail::implicit_convolve_sgemm`). The `cudnn::detail::implicit_convolve_sgemm` kernel has high arithmetic intensity and dominates the model's latency. For batch sizes greater than 16, the cuDNN library chooses a different algorithm — `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm (kernel `volta_scudnn_128x64_relu_interior_nn_v1`). Although the `volta_scudnn_128x64_relu_interior_nn_v1` 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 the `volta_scudnn_128x64_relu_interior_nn_v1` kernel's arithmetic intensity is not high enough to compensate the effects of the other memory-bound kernels. The results 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.

Current practice for GPU kernel-level analysis — The 10 analysis — which aggregates the GPU information by name — is currently the most common type of analysis performed by researchers to report model performance. A less common, but still possible analysis is the model-level 15 roofline analysis. Analysis 15 can be performed using GPU profilers (such as nvprof) along with scripts to parse the profiler's output and aggregate results for all kernels across multiple model evaluations. With the exception of these two analysis, all the other analysis listed in this section require the correlation between layer- and GPU kernel-level profiles.

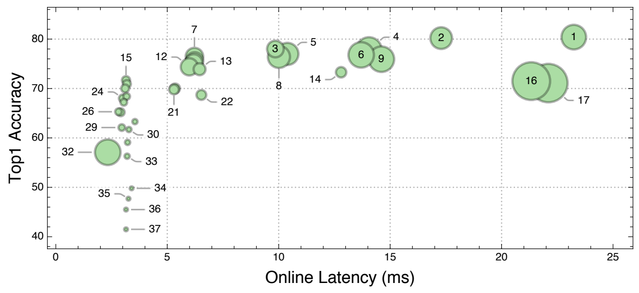
As stated in Section 2, currently there is no tool or method that correlate GPU profiles to layer profiles other than modifying framework source code or manually inserting profiling code. For example, to be able to correlate GPU kernels to a certain layer. Users are asked to manually inserts NVTX markers within their source code or to insert fake NVTX TensorFlow layers to capture the layer-level information [20]. Any GPU kernel profile captured within the NVTX markers' range can then correlated to the layer. Users then use the nvprof or Nsight to capture and view the captured annotations by the markers. NVIDIA NGC also provides modified versions of TensorFlow as Docker containers where NVTX markers are already added to the TensorFlow runtime. Unlike NVIDIA's solution, our methodology does not require modifications to the user's source code or the framework. Our methodology is applicable to (and has been used by) other frameworks such as MXNet, PyTorch, Caffe, Caffe2, and TensorRT.

5 EVALUATION

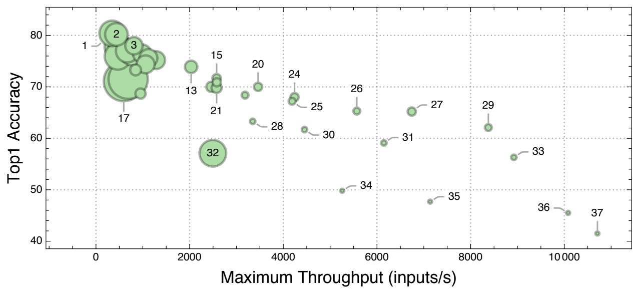
We profile and characterize 55 state-of-the-art TensorFlow ML models (shown in Table 6) which are from the MLPerf Inference [11], AI-Matrix [8], and TensorFlow model zoo [23, 24, 26]. The models chosen solve different computer vision tasks: Image Classification, Object Detection, Instance Segmentation, Semantic Segmentation, and Super Resolution. To compare TensorFlow against MXNet, we select an additional 10 MXNet model zoo models [12] (show in Table 9) that are comparable to the TensorFlow models. We evaluated the models using NVIDIA's NGC [16] TensorFlow container v19.06,

ID	Name	Task	Top 1 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 6: For evaluation, we use 55 pretrained TensorFlow models from MLPerf [11], AI-Matrix [8], and TensorFlow Slim, Detection Zoo, DeepLab [23,24,26] 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 System 2 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.



(a) Accuracy vs online latency (batch size = 1).



(b) Accuracy vs maximum throughput.

Figure 10: Comparing the accuracy, latency, and throughput for all 37 models on System 2 using their optimal batch size. The size of each circle is proportional to the model's graph size.

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	✗

Table 7: MLPerf_ResNet50_v1.5's GPU kernel information aggregated within the model across batch sizes on System 2.

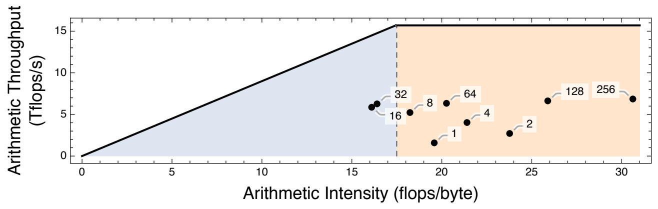


Figure 11: The roofline analysis of the MLPerf_ResNet50_-v1.5 model across batch sizes on System 2.

and NGC MXNet container v19.06 on 5 representative systems (listed in Table 5) with Turing, Volta, Pascal, and Maxwell GPUs.

This section presents insights obtained through the models, frameworks, and GPU systems analysis described in Section 4. Due to the page limit, and with the exception of the framework comparison (in Section 5.2), we only characterize the TensorFlow models. We refer the reader to aspc19.mlmodelscope.com which contains further characterizations for all the models and frameworks. The website has been automatically generated using our across-stack profiling and analysis tools.

5.1 Model Evaluation

Leveraging the model- and layer-level profiling data, we look at all 55 TensorFlow models as shown in Table 6. Models solving the same task are clustered together and are then sorted by their reported accuracy. The table, which has been computed through our analysis pipeline, shows: each model's performance on System 2, accuracy, model graph size, online latency (batch size is 1), maximum throughput, optimal batch size (see Section 4.1), and latency percentage attributed to the convolutional layers. The convolutional layer percentage analysis shows that it is generally the case that image classification models are dominated by the convolutional layers, whereas this statement is less true for object detection and segmentation tasks. The accuracy and performance information is used to help model deployers differentiate between ML models and choose the best model given the accuracy and target latency objectives.

Model accuracy, performance, and graph size — To compare performance across models, we look at models that solve the same task. We choose the 37 image classification TensorFlow models to examine the effects of model accuracy on online latency (Figure 10b), and maximum throughput (Figure 10a). In both figures, the area of the circle is proportional to the model's graph size. In Figure 10a, where models within the upper left quadrant are better, shows limited correlation between the model accuracy and online latency. I.e. models with comparable online latency targets can vary considerably in accuracy. Similarly, in Figure 10b, where models in the upper right quadrant models are favorable, shows no direct correlation between model accuracy and maximum throughput. I.e. models with comparable accuracy can achieve quite different maximum throughput. Overall, the graph size (which roughly represents the number of weight values) is not directly correlated to either accuracy or performance.

Model throughput scalability across batch sizes — When comparing the online latency and throughput (Figures 10b and 10a) we observe that models that exhibit good throughput-oriented behavior do not necessarily perform well in latency-oriented scenarios (where online latency matters). This can be partly explained by how well the throughput scales with batch size. Figure 12 shows the throughput speedup of the 37 image classification models as the batch size varies. The throughput scalability varies across models, and even models with similar network architectures can have different throughput scalability — e.g., models 4 and 6, models 5 and 8, and models 10 and 11. Overall, we observe that the ResNet_50 class of models offer a balance between model size, accuracy, online latency, and maximum throughput.

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

Table 8: In-depth characterization of the 37 image classification models listed in Table 6 at the optimal batch sizes on System 2. The model execution is partitioned into beginning (*B*), middle (*M*) , and end (*E*) intervals based on layer index. The most intensive stages for latency, allocated memory, flops and memory access are shown for each model.

Model latency percentage of convolutional layers — With the model and layer profile data, we calculate the percentage of latency attributed to the convolutional layers (Tensorflow's `Conv2D` [4] and `DepthwiseConv2dNative` [5] layers) with each model's optimal batch size on System 2. This is shown in the last column of Table 6. We observe that: ① the convolutional layer latency percentage ranges between 36.3% and 80.2% for image classification models. This suggests that convolutional 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 (with the exception of Faster_RCNN_NAS) attribute only 0.6% to 14.9% of latency to the convolutional layer. For these models, the dominating layer is the `Where` [6] layer, which reshapes a tensor with respect to a user-defined operator. For ③ instance segmentation models, convolutional layers dominate latency; except for Mask_RCNN_Inception_v2, which is again dominated by `Where` layer. For ④ semantic segmentation models, latency is affected by both the convolutional layers and the memory-bound layers (such as `Transpose`, `Add`, and `Mul`). Finally, ⑤ the super resolution model SRGAN is dominated by the convolution layers.

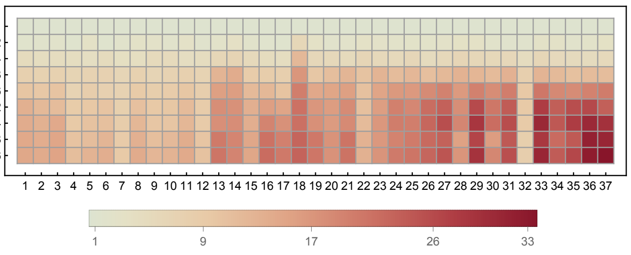


Figure 12: The throughput speedup (over batch size 1) heatmap across batch sizes on System 2 for the 37 image classification models in Table 6. The $y-$axis shows the batch size, whereas the $x-$axis shows the model ID.

GPU latency, flops and memory accesses — With model-, layer-, and GPU kernel-level profiling we perform in-depth analysis on the 37 image classification models using their optimal batch sizes on System 2, as shown in Table 8. The table shows the: model latency at the optimal batch size, GPU latency percentage, GPU metrics,

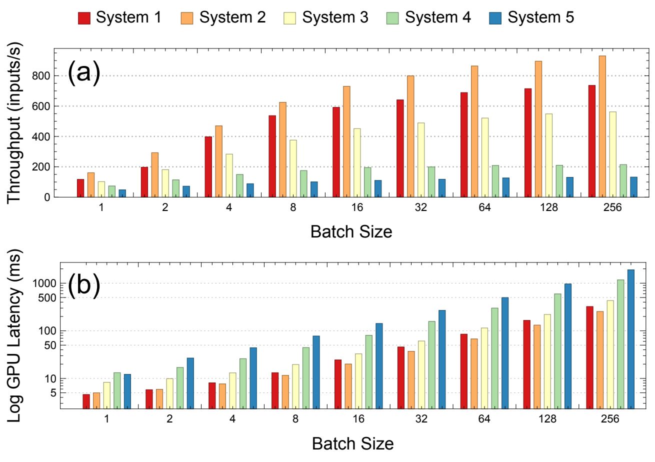


Figure 13: The throughput and latency (log scale) of MLPerf_ResNet50_v1.5 across batch sizes and systems.

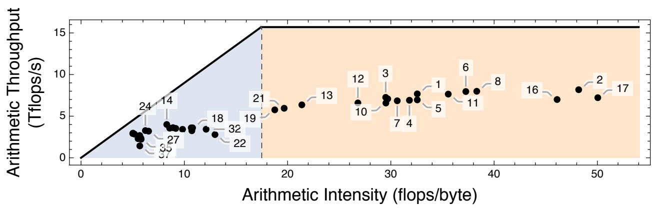


Figure 14: The roofline analysis for all 37 image classification models running the optimal batch sizes for System 2.

arithmetic intensity and throughput, and the most intensive stage for latency, memory allocation, GPU flops, and memory accesses throughout the model execution. We find that across the models, the GPU latency percentage (i.e. latency due to GPU kernel execution and ignoring CPU code) 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). The table shows that models with high batch latency tend to have 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 shows that time spent within CPU code (framework overhead or GPU stalls due to synchronization) is still high for some models.

Batch size vs GPU achieved occupancy — The GPU achieved occupancy is a partial indicator of the utilization of a GPU. Table 7 shows a trend that as a model's batch size approaches its optimal batch size, the overall GPU achieved occupancy increases.

Roofline analysis — Figure 14 shows the roofline analysis of all 37 image classification models using their optimal batch sizes on System 2. Out of 37 models, 20 are bounded by memory. Models with low compute and memory requirements tend to be memory-bound and have lower accuracy, e.g. some variants of MobileNet which are optimized for edge devices. All models achieve at most 52% of the theoretical peak throughput, suggesting that there is room for further framework- or model-level optimizations.

Latency, memory allocation, flops, and memory access trend — To understand the performance trend within model execution, we divide the model execution timeline into 3 intervals: beginning, middle, and end based on layer index. We then compute the total latency, flops, and memory accesses within each interval and identify which interval dominates. The last 4 columns in Table 8 show the results on System 2 for the 37 image classification models. The demanding intervals vary across models and suggests that one can potentially interleave model executions (for applications that use multiple models) to maximize GPU utilization.

5.2 ML Framework Evaluation

To compare ML frameworks, 10 MXNet models were selected from the MXNet model zoo [12]. We chose 6 variants of the ResNet network architecture which are compute intensive and are compute-bound at the optimal batch size, and 4 variants of the MobileNet network architecture which are less compute intensive and are memory-bound. The models (shown in Table 9) are comparable to the TensorFlow models and we perform the TensorFlow/MXNet comparison on System 2. The online latency and maximum throughput in the table are normalized to the corresponding values using TensorFlow. We find the optimal batch size for each of model. With the exception of model 18, all models match TensorFlow's optimal batch size.

Compute-bound models — Table 9 shows that the online latencies of MXNet's ResNets are larger than the TensorFlow ones. After looking into the analysis results, we find that while the total GPU kernel latencies of the TensorFlow's and MXNet's ResNets are about the same, MXNet has a much larger non-GPU latency compared to TensorFlow for batch size 1. `ResNet_v1_50`, for example, has a non-GPU latency of 4.44*ms* (55.1% of the total online latency) for MXNet whereas it is only 2.18*ms* for TensorFlow (35.3% of the total). We find that as the batch size increases and the ResNets become compute-bound, the percentage of the non-GPU latency decreases and the MXNet ResNet models (models 4 − 11) achieve about the same maximum throughput as the TensorFlow models. At the ideal batch size, ResNets' GPU latency percentage, flops and memory accesses, achieved occupancy, and the roofline analysis are comparable for both TensorFlow and MXNet. This suggests that MXNet incurs a fixed overhead for model execution which is pronounced for small batch sizes.

Memory-bound models — For the less compute intensive MobileNet models, we observe that the MXNet achieves the same online latency as the TensorFlow. However, as the batch size increases (i.e. the MobileNet models become memory-bound) we find that MXNet has less memory accesses and therefore a higher achieved GPU occupancy compared to TensorFlow. As a result, the MXNet MobileNet models achieve between 35% and 74% more throughput at their corresponding optimal batch sizes (Shown in Table 9). 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 operations. We observe that Eigen introduces and makes excessive DRAM reads and writes. This becomes a performance limiting factor as models become memory-bound.

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	✓

Table 9: Characterization of 10 MXNet models, which are comparable to the TensorFlow ones listed in Table 6 (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 System 2. The online latency and maximum throughput are normalized to TensorFlow's.

5.3 System/GPU Evaluation

We evaluate MLPerf_ResNet50_v1.5 on all 5 systems in Table 5 using the NGC TensorFlow container. Although System 1's GPU has a higher peak FLOPS compared to System 2, it has a lower memory bandwidth. Hence, we expected System 1 to straggle on memory-bound layer and perform slightly worse than System 2. Figure 13a shows the throughput across systems and batch sizes. We observe that even when fixing the software stack (such as TensorFlow, cuDNN, and cuBLAS, CUDA driver versions) performance differs across systems as the batch size varies. Throughput also scales differently across systems with respect to the batch size. This is mainly attributed to the GPUs having different peak performance and cuDNN's heuristics which dispatch to different algorithms depending on the GPU and batch size (as discussed in Section 4.3).

Figure 13b shows the GPU latency (the total latency of all the GPU kernel calls) in log scale for the systems across different batch sizes. Looking at the GPU kernel-level profile for each system, we find that Systems 3, 4, and 5 call the same set of GPU kernels, while Systems 1 and 2 call a different set of GPU kernels. As discussed in Section 4.3, this is expected due to the behavior of cuDNN's `cuDNNConvolutionForward` function, which uses the layer's input parameters along with the GPU architecture to perform heuristics for kernel and algorithm selection. For example, the convolution layers for batch size 256 on systems 3, 4, and 5 invoke the `maxwell_scudnn_*` kernels, whereas on systems 1 and 2 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, System 2 calls the `volta_scudnn_128x64_relu_interior_nn_v1` kernel 34 times while System 1 only calls it 18 times (the other 16 being dispatched to the `volta_scudnn_128x128_relu_interior_nn_v1` kernel).

6 CONCLUSION

A big hurdle in deploying and optimizing ML workloads within systems is understanding their performance characteristics across the HW/SW stack. We observe that the characterization and analysis currently performed on ML models and systems is largely limited by the use of ad-hoc profiling techniques and the lack of profiling correlation across the stack levels. This paper proposes an across-stack profiling design that leverage information from different profiling providers and correlates them. The design is enabled by the innovative use of distributed tracing and copes with the inherent overheads caused by the profiling through the leveled experimentation methodology. While the across-stack profiling design is general, this paper focuses on how it enables an in-depth automated profiling, analysis and characterization of ML models on GPUs.

We show that the across-stack profiling design helps researchers gain insights to understand the current sources of inefficiency of ML models, frameworks, and systems. To ease usage, we extend MLModelScope— a scalable open-source ML model evaluation platform — with our across-stack profiling design. We then develop 15 analysis that are automatically performed to characterize 65 state-of-the-art TensorFlow and MXNet models on 5 representative GPU systems. Through this characterization, we derive meaningful insights about the models, frameworks, and systems. Absent our methodology, these insights would be difficult to discern. The underlying data and further results are published to aspc19.mlmodelscope.com for the reader to reference and inspect. We are currently extending our across-stack profiler to capture other metrics such as power to characterize energy-efficiency of models and frameworks on mobile and edge devices.

ACKNOWLEDGMENTS

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM Cognitive Horizon Network. The authors would like to thank Tao Zhang for the valuable discussions on performing roofline analysis and Yongbin Gu for validating the initial profiling results against NVIDIA's nvprof.

REFERENCES

[1] NVIDIA GPU Metrics Reference. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference. Accessed: 2019-7-24.

[2] NVIDIA Tools Extension. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx. Accessed: 2019-7-24.

[3] NVTX Plugins for Deep Learning. https://github.com/NVIDIA/nvtx-plugins. Accessed: 2019-7-24.

[4] tensorflow::ops::Conv2D . https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/conv2-d. Accessed: 2019-7-24.

[5] tensorflow::ops::DepthwiseConv2dNative . https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/depthwise-conv2d-native. Accessed: 2019-7-24.

[6] tensorflow::ops::Where . https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/where. Accessed: 2019-7-24.

[7] The CUDA Profiling Tools Interface. https://developer.nvidia.com/cuda-profiling-tools-interface, 2018. Accessed: 2019-08-04.

[8] AI-Matrix. https://github.com/alibaba/ai-matrix, 2019. Accessed: 2019-08-04.

[9] Amazon EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3/, 2019. Accessed: 2019-08-04.

[10] Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/, 2019. Accessed: 2019-08-04.

[11] MLPerf Inference. https://github.com/mlperf/inference, 2019. Accessed: 2019-08-04.

[12] MXNet Gluon Model Zoo. https://gluon-cv.mxnet.io/model_zoo/index.html, 2019. Accessed: 2019-08-04.

[13] MXNet Profiler. https://mxnet.incubator.apache.org/api/python/profiler/profiler.html, 2019. Accessed: 2019-08-04.

[14] Nsight. https://developer.nvidia.com/tools-overview, 2019. Accessed: 2019-08-04.

[15] NVIDIA cuDNN. https://developer.nvidia.com/cudnn, 2019. Accessed: 2019-7-04.

[16] NVIDIA GPU-Accelerated Containers. https://www.nvidia.com/en-us/gpu-cloud/containers/, 2019. Accessed: 2019-08-04.

[17] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2019. Accessed: 2019-7-04.

[18] Nvidia Visual Profiler. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual, 2019. Accessed: 2019-08-04.

[19] nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2019. Accessed: 2019-08-04.

[20] NVTX Plugins for Deep Learning. https://github.com/NVIDIA/nvtx-plugins, 2019. Accessed: 2019-08-04.

[21] PyTorch Profiler. https://pytorch.org/docs/stable/_modules/torch/autograd/profiler.html, 2019. Accessed: 2019-08-04.

[22] Spans. https://opentracing.io/docs/overview/spans/, 2019. Accessed: 2019-08-04.

[23] TensorFlow DeepLab Model Zoo. https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/model_zoo.md, 2019. Accessed: 2019-08-04.

[24] TensorFlow Detection Model Zoo. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md, 2019. Accessed: 2019-08-04.

[25] TensorFlow Profiler. https://www.tensorflow.org/api_docs/python/tf/profiler, 2019. Accessed: 2019-08-04.

[26] TensorFlow-Slim Image Classification Model Library. https://github.com/tensorflow/models/tree/master/research/slim, 2019. Accessed: 2019-08-04.

[27] What is Distributed Tracing. https://opentracing.io/docs/overview/what-is-tracing/, 2019. Accessed: 2019-08-04.

[28] Adolf, R., Rama, S., Reagen, B., Wei, G.-Y., and Brooks, D. Fathom: Reference workloads for modern deep learning methods. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on* (2016), IEEE, pp. 1–10.

[29] Baidu. Deepbench. https://github.com/baidu-research/DeepBench, 2018.

[30] Ben-Nun, T., Besta, M., Huber, S., Ziogas, A. N., Peter, D., and Hoefler, T. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. IEEE. The 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'19).

[31] Braun, S. Lstm benchmarks for deep learning frameworks. *arXiv preprint arXiv:1806.01818* (2018).

[32] Cheng, Y., Chai, Z., and Anwar, A. Characterizing co-located datacenter workloads: An alibaba case study. *arXiv preprint arXiv:1808.02919* (2018).

[33] Chintala, S. ConvNet Benchmarks. https://github.com/soumith/convnet-benchmarks, 2018.

[34] The Cloud Native Computing Foundation. https://www.cncf.io, 2019. Accessed: 2019-08-04.

[35] Dakkak, A., Li, C., Xiong, J., and Hwu, W.-M. Frustrated with replicating claims of a shared model? a solution. *arXiv preprint arXiv:1811.09737* (2019).

[36] HPE Deep Learning Performance Guide. https://dlpg.labs.hpe.com, 2019. Accessed: 2019-08-04.

[37] Eriksen, M. Your server as a function. *ACM SIGOPS Operating Systems Review 48*, 1 (2014), 51–57.

[38] Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[39] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[40] Henning, J. L. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[41] Jia, Y. Caffe2. https://www.caffe2.ai, 2018.

[42] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (2014), ACM, pp. 675–678.

[43] Juckeland, G., Brantley, W., Chandrasekaran, S., Chapman, B., Che, S., Colgrove, M., Feng, H., Grund, A., Henschel, R., Hwu, W.-M. W., et al. Spec accel: A standard application suite for measuring hardware accelerator performance. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems* (2014), Springer, pp. 46–67.

[44] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[45] The NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl, 2019. Accessed: 2019-08-04.

[46] Newman, S. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[47] Pal, A., and Pal, M. Interval tree and its applications. *Advanced Modeling and Optimization 11*, 3 (2009), 211–224.

[48] Paszke, A., Gross, S., Chintala, S., and Chanan, G. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration 6* (2017).

[49] Poess, M., and Floyd, C. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record 29*, 4 (2000), 64–71.

[50] Pree, W., and Gamma, E. *Design patterns for object-oriented software development*, vol. 183. Addison-wesley Reading, MA, 1995.

[51] Shende, S. S., and Malony, A. D. The tau parallel performance system. *The International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311.

[52] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[53] Viebke, A., Pllana, S., Memeti, S., and Kolodziej, J. Performance modelling of deep learning on intel many integrated core architectures.

[54] Wang, Y., Wei, G., and Brooks, D. Benchmarking tpu, gpu, and CPU platforms for deep learning. *CoRR abs/1907.10701* (2019).

[55] Williams, S., Waterman, A., and Patterson, D. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.