The Design and Implementation of a Scalable Deep Learning Benchmarking Platform

Cheng Li\textsuperscript{1*}, Abdul Dakkak\textsuperscript{1*}  
University of Illinois Urbana-Champaign, Urbana, USA  
{cli99, dakkak}@illinois.edu

Jinjun Xiong  
IBM T. J. Watson Research Center, Yorktown Heights, USA  
jinjun@us.ibm.com

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

Abstract—The current Deep Learning (DL) landscape is fast-paced and is rife with non-uniform models, hardware/software (HW/SW) stacks. Currently, there is no DL benchmarking platform to facilitate the evaluation and comparison of DL innovations, be it models, frameworks, libraries, or hardware. As a result, the current practice of evaluating the benefits of proposed DL innovations is both arduous and error-prone — stifling the adoption of the innovations.

In this work, we first identify 10 design features that are desirable within a DL benchmarking platform. These features include: performing the evaluation in a consistent, reproducible, and scalable manner, supporting model and hardware agnostic evaluation, providing in-depth model execution inspection across the HW/SW stack levels, etc. We then propose MMLModelScope, a DL benchmarking platform that realizes these 10 design objectives. MMLModelScope introduces a specification to define DL model evaluations and provides a runtime to provision the evaluation workflow using the user-specified HW/SW stack. MMLModelScope defines abstraction frameworks for automation and supports the board range of DL models and evaluation scenarios. We implement MMLModelScope as an open-source project with support for major frameworks and hardware architectures. Through MMLModelScope’s evaluation and automated analysis workflows, we perform a case-study analysis of 37 models across 4 systems and show how model, hardware, and framework selection affects model accuracy and performance under different benchmarking scenarios. We further demonstrate how MMLModelScope’s tracing capability gives a holistic view of model execution and helps pinpoint bottlenecks.

I. INTRODUCTION

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 [2], [3] 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 [4], [5], [6]. Thus, the benchmarking platform design must guarantee a \textsuperscript{1}reproducible evaluation along with \textsuperscript{2}consistent evaluation.

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

We propose MMLModelScope, a scalable DL benchmarking platform design that realizes the above 10 objectives and facilitates benchmarking, comparison, and understanding of DL model execution. MMLModelScope achieves the design objectives by proposing a specification to define DL model evaluations; introducing techniques to consume the specification and provision the evaluation workflow using the specified HW/SW stack; using a distributed scheme to manage, schedule, and handle model evaluation requests; supporting pluggable workload generators; defining a common abstraction API across frameworks; providing across-stack tracing capability that allows users to inspect model execution at different HW/SW

\textsuperscript{1}The two authors contributed equally to this paper.
abstraction levels; defining an automated evaluation analysis workflow for analyzing and reporting evaluation results; and, finally, exposing the capabilities through web and command-line interfaces.

We implement MLModelScope and integrate it with the Caffe, Caffe2, CNTK, MXNet, PyTorch, TensorFlow, TensorFlow Lite, 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 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-level bottlenecks. To the authors’ knowledge, we are the first to describe the design and implementation of a scalable DL benchmarking platform.

II. BACKGROUND

This section gives a brief background of DL model evaluation and current DL benchmarking practice.

A. DL Model Evaluation Pipeline

As shown in Figure 1, a DL model evaluation pipeline consists of input pre-processing, model prediction, and output post-processing. 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 as an example, the pre-processing stage decodes the input image into a tensor of dimensions \([\text{batch}, \text{height}, \text{width}, \text{channel}]\) \([N, H, W, C]\), performing resizing, normalization, etc. The image classification model’s output is a tensor of dimensions \([\text{batch} \times \text{numClasses}]\) which is sorted to get the top \(K\) predictions (labels with probabilities). A DL model is defined by its graph topology and its weights. The model 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, or PyTorch). The framework acts as a “runtime” for model prediction and maps the function operators into system library calls. As can be observed, this pipeline is intricate and has many levels of abstraction. When a slowdown is observed, any one of these levels of abstraction can be suspect.

B. Current DL Benchmarking

While there has been a drive to provide reference DL benchmarks [2], [3], the current benchmarking effort is still scattered, lacks a standard benchmarking methodology, and revolves around a series of ad-hoc scripts that evaluate models on local systems. To consistently evaluate two models involves: instantiating the same hardware; installing the same software packages and their dependencies; and, finally, measuring and analyzing the results of both models in the same way. Because of the use of ad-hoc scripts and the lack of a standard way to evaluate models, the above process requires a lot of manual work and can be error-prone — often resulting in non-reproducible [7], [5], [6] results. Due to the daunting effort to perform fair benchmarking, DL innovations proposed have outpaced researchers’ ability to compare and analyze them [1].

III. DESIGN OBJECTIVES

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

1. Reproducible Evaluation—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 [4]. Thus, all aspects of evaluation must be specified and provisioned by the design to guarantee reproducible evaluation.

2. Consistent Evaluation—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 HW/SW components). A fair apples-to-apples 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—There are many DL frameworks (e.g., TensorFlow, MXNet, and PyTorch) and hardware (e.g., CPU, GPU, FPGA) and each has its use scenarios, features, and performance characteristics. To have broad support, the design must be framework and hardware agnostic. Furthermore, the design must be valid without requiring framework modifications.

Scalable Evaluation—DL innovations, such as models, frameworks, libraries, compilers, and hardware accelerators are introduced at a rapid pace [1], [8]. 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 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 report should be automatically gathered for comparison.

Artifact Versioning—DL frameworks are continuously updated by the DL community, e.g., the recent versions of TensorFlow at the time of writing are v1.15.3 and v2.1.1. There are many unofficial variants of models, frameworks, and datasets as researchers might update or modify them to suit 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—Before model inference can be performed, the input data has to be loaded and transformed into a form that the model expects (the pre-processing step). After the model prediction, the post-processing step 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 [9]. Thus, the design should handle and process data efficiently in the evaluation workflow.

Benchmarking Scenarios—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—Benchmarking produces raw data that 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—The complexity of DL model evaluation makes performance debugging challenging as each level within the HW/SW abstraction hierarchy can be 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 [10] or Intel’s Vtune [11]). 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 tracing capability at all levels of the HW/SW stack.

Different User Interfaces—While the command line is the most common interface in the current benchmarking suites, having other user interfaces (UIs), such as web UI, to accommodate other use cases can greatly boost productivity. For example, 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 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.

IV. MLModelScope Design and Implementation

We propose MLModelScope, a DL benchmarking platform design that achieves the objectives set out in Section III. To achieve scalable evaluation, we design MLModelScope as a distributed platform. To enable 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 adapt to 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 2 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 the 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, generating benchmark workloads based on benchmarking scenarios, dispatching the model evaluation workload to MLModelScope agents, 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 the 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**—are a set of support services for MLModelScope including: a distributed registry (a key-value store containing entries of the running agents and available models), an evaluation database (a database containing the evaluation results), a tracing server (a server to publish the profile events captured during an evaluation), and an artifact storage server (a data store repository containing model assets and datasets).

Figure 2 also shows MLModelScope’s three main workflows: ① initialization, ②-⑥ evaluation, and ⑦-⑨ 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 stack, system, 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 querying the distributed registry, and then ③ dispatches the request to one or more of the resolved agents. The agent then ④ downloads the required evaluation assets from the artifact storage, performs the evaluation, and ⑤-⑥ publishes the evaluation results to the evaluation database and tracing server. A summary of the results is ⑦ sent to the server which ⑧ forwards it to the client. Finally, the analysis workflow allows a user to perform a more fine-grained and in-depth analysis of the results across evaluation runs. The MLModelScope server handles this workflow by ⑨ querying the evaluation database and performing analysis on the results, and ⑩ generating a detailed analysis report for the user. This section describes the MLModelScope components and workflows in detail.

### A. User Input

All aspects of DL evaluation — model, software stack, system, and benchmarking scenario — must be specified to MLModelScope for it to enforce reproducible and consistent evaluation. To achieve this, MLModelScope defines a benchmarking specification covering the 4 aspects of evaluation. A model in MLModelScope is specified using a framework manifest, and a software stack is specified using a hardware requirements manifest. The manifest specifications are decoupled, one can easily evaluate the different combinations, enabling easy benchmarking.

For example, a user can use the same MLPerf_ResNet50_v1.5 model manifest (shown...
in Listing 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 the MLModelScope agents (Section IV-D). 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 Dockerhub.

1) **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 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 certain TensorFlow versions—e.g., greater than 1.2.0 but less than 2 as on Lines 4–6 in Listing 1. This allows MLModelScope to support models that use specific or custom frameworks.

**Pre- and Post-Processing**—To perform pre- and post-processing for model evaluation, arbitrary Python functions can be placed within the model manifest (Lines 29 and 30 in Listing 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 sub- interpreter [12] 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 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., top-k, intersection over union, etc.) which are widely used within models. Users can use the 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 1 for pre-processing). The pre- and post-processing operators 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 on the web or within 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 1 are stored as Zenodo [13] artifacts (Lines 31-34) and are downloaded prior to evaluation.

2) **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 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). In the MLModelScope initialization phase (4), MLModelScope agents (described in Section IV-D) register themselves by publishing their HW/SW stack information into the distributed registry (described in Section IV-E1). 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

### Listing 1: MLPerf_ResNet50_v1.5 model manifest.
```
name: MLPerf_ResNet50_v1.5 # model name
version: 1.0.0 # semantic version of the model
framework: TensorFlow # framework information
version: ‘>1.12.0 <2.0’ # framework version constraint
inputs: # model inputs
  - type: image # first input modality
    layer_name: ‘input_tensor’
    element_type: float32
outputs: # model outputs
  - type: probability # first output modality
    layer_name: prob
    element_type: float32
```

### Listing 2: Framework Manifest.
```
name: TensorFlow
version: ‘>1.12.0 <2.0’
```

### Example Model Manifest.
```
name: MLPerf_ResNet50_v1.5
version: 1.0.0
framework: TensorFlow
version: ‘>1.12.0 <2.0’
inputs: # model inputs
  - type: image # first input modality
    layer_name: ‘input_tensor’
    element_type: float32
outputs: # model outputs
  - type: probability # first output modality
    layer_name: prob
    element_type: float32
```

### Example Framework Manifest.
```
name: TensorFlow
version: ‘>1.12.0 <2.0’
```
are then run on one of (or, at 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.

3) Benchmarking Scenario: MLModelScope provides a set of built-in benchmarking scenarios. Users pick which scenario to evaluate under. 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 an inference request load based on the benchmarking scenario option and sends it to the resolved agent(s) to measure the corresponding benchmarking metrics of the model (detailed in Section IV-C).

B. MLModelScope Client

A user initiates a model evaluation or analysis through the MLModelScope client. To enable the client, it can be either be a website or a command line tool. The client communicates with the MLModelScope server through REST API and sends user evaluation requests. The web UI 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 and perform evaluations — 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.

C. 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 4) to interact with the MLModelScope agents. To enforce API functions, the MLModelScope server can be load balanced to avoid it being a bottleneck.

In the evaluation workflow, the server is responsible for accepting tasks from the MLModelScope client, querying the distributed registry and resolving the user-specified constraints to find MLModelScope agents capable of evaluating the request, dispatching the evaluation task to the resolved agent(s) and generating loads for the evaluation, collecting the evaluation summary from the agent(s), and 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 (F1).

In the analysis workflow, the server again takes the user input, but, rather than performing evaluation, it queries the evaluation database (Section IV-E2), and aggregates and analyzes the evaluation results into a report. MLModelScope enables an across-stack analysis pipeline. It consumes the benchmarking results and profiling traces in the evaluation database and performs the analysis. Then the server 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.

D. Agent and Framework Predictor

An 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 [14] as shown in Listing 4. A framework predictor resides within an MLModelScope agent and is a wrapper around a framework and links to the framework’s C or C++ library.

During the initialization phase (D1), an 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 downloads the required evaluation assets using the data manager, 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 tracing server to get aggregated into a single profiling trace. The benchmarked results 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.

1) 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

---

Listing 2: An example TensorFlow framework manifest.

```plaintext
name: TensorFlow # framework name
version: 1.15.0 # semantic version of the framework
description: ...
containers: # containers
  amd64:
    cpu: carml/tensorflow:1-15-0_amd64-cpu
    gpu: carml/tensorflow:1-15-0_amd64-gpu
  ppc64le:
    cpu: carml/tensorflow:1-15-0_ppc64le-cpu
    gpu: carml/tensorflow:1-15-0_ppc64le-gpu
```

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

```plaintext
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 4: An example TensorFlow framework manifest.

```plaintext
TensorFlow # framework name
version: 1.15.0 # semantic version of the framework
description: ...
containers:
  amd64:
    cpu: carml/tensorflow:1-15-0_amd64-cpu
    gpu: carml/tensorflow:1-15-0_amd64-gpu
  ppc64le:
    cpu: carml/tensorflow:1-15-0_ppc64le-cpu
    gpu: carml/tensorflow:1-15-0_ppc64le-gpu
```

Listing 2: An example TensorFlow framework manifest.
The pre- and post-processing operations, as well as the model evaluation workflow, MLModelScope leverages a streaming 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).

3) Framework Predictor: Frameworks provide different APIs (usually across programming languages e.g. C/C++, Python, Java) to perform inference. To enable 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 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.

MLModelScope design supports agents on ASIC and FPGA. Any code implementing the predictor interface shown in Listing 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.

4) Tracing Hooks: To enable tracing, MLModelScope leverages across-stack profiling [15] to capture the profiles at different levels of granularity (model-, framework-, and system-level) using the tracing hooks. A tracing hook is a pair of start and end code snippets and follows the standards [16] 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 MLModelScope tracer server (Section IV-E3). Trace events are published asynchronously to the MLModelScope 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 the stack.

E. 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.

1) 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, solves user-specified constraints for selecting MLModelScope agents, and load balances the requests across agents. The registry is dynamic — both model manifests and predictors can be added or deleted at runtime throughout the lifetime of MLModelScope.

2) 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.

3) Tracing Server: The MLModelScope tracing server is a distributed tracing server that accepts profiling data published by the MLModelScope agent’s trace hooks. As stated in Section IV-D4, user-specified options control the granularity.
F. 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 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. Instead, a user provides a framework manifest as shown in Listing 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 an 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 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. Users can configure these existing tracing hooks to capture other system metrics. For example, to limit profiling overhead, by default, the CUPTI [17] tracing hooks capture only some CUDA runtime API, GPU activities (kernel and memory copies), and CPU metrics. They can be configured to capture other GPU activities and metrics, or NVTX [18] markers. Moreover, users can integrate other system profilers into MLModelScope by implementing the tracing hook interface.

G. Implementation

We implemented the MLModelScope design with support for common frameworks and hardware. At the time of writing, MLModelScope has built-in support for Caffe, Caffe2, CNTK, MXNet, PyTorch, TensorFlow, TensorFlow Lite, and TensorRT. MLModelScope works with binary versions of the frameworks (version distributed through Python’s pip, for example) and supports customized versions of the frameworks with no code modification. MLModelScope has been tested on X86, PowerPC, and ARM CPUs as well as NVIDIA’s Kepler, Maxwell, Pascal, Volta, and Turing GPUs. It can also evaluate models deployed on FPGAs. During the evaluation, users specify hardware constraints such as: whether to run on CPU/GPU/FPGA, type of architecture, type of interconnect, and minimum memory requirements — which MLModelScope uses for agent resolution.

We populated MLModelScope with over 300 built-in models covering a wide array of inference tasks such as image classification, object detection, segmentation, image enhancement, recommendation, etc. We verified MLModelScope’s accuracy and performance results by evaluating the built-in models and frameworks across representative systems and comparing to those publicly reported.

We implemented MLModelScope’s web UI using the React Javascript framework. The web UI interacts with a REST API provided by the server. The REST API can be used by other clients that wish to integrate MLModelScope within their workflow. An MLModelScope command-line client is also available and can be used with shell scripts. The agents also expose a gRPC API which can be used to perform queries to the agents directly.

V. Evaluation

Previous sections discussed in detail how MLModelScope’s design and implementation achieves the F1–9 and F10 design objectives. In this section, we focus on evaluating how MLModelScope handles F7 different benchmarking scenarios, F8 result summarization, and F9 inspection of model execution. We installed MLModelScope on the systems listed in Table I. 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.

A. Benchmarking Scenarios

To show how MLModelScope helps users choose from different models and system offerings for the same DL task, we compared the inference performance across the 37 TensorFlow models (Table II) and systems (Table I) under different benchmark scenarios. For each model, we measured its trimmed mean latency\(^1\), the 90\(^{th}\) percentile latency in an online (batch size = 1) inference scenario, and the maximum throughput in a 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 for 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 4) and maximum throughput (Figure 3). In both figures, the area of the circles is proportional to the model’s graph size (or number of weight parameters). In Figure 3 we find a limited correlation between a model’s online latency and its accuracy — models taking a longer time to run do not necessarily achieve higher accuracy; 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 4, we find a limited correlation between a model’s accuracy and its maximum throughput — two models with

\[ \text{Mean} = \frac{1}{n} \sum_{i=1}^{n} x_i \]

\[ \text{Median} = \begin{cases} \text{sorted}[\lfloor \frac{n}{2} \rfloor] & \text{if } n \text{ is odd} \\ \frac{1}{2} \left( \text{sorted}[\lfloor \frac{n}{2} \rfloor] + \text{sorted}[\lfloor \frac{n}{2} \rfloor + 1] \right) & \text{if } n \text{ is even} \end{cases} \]

\[ \text{Trimmed Mean} = \text{Mean}(\text{Sort}(\text{list})[\lfloor 0.2 \times \text{len}(\text{list}) \rfloor :: \lfloor 0.2 \times \text{len}(\text{list}) \rfloor ]) \]

\[ \text{Trimmed Mean} = \text{Mean}(\text{Sort}(\text{list})[\lfloor 0.2 \times \text{len}(\text{list}) \rfloor :: \lfloor 0.2 \times \text{len}(\text{list}) \rfloor ]) \]

\[^1\text{Trimmed mean is computed by removing } 20\% \text{ of the smallest and largest elements and computing the mean of the residual; i.e. } \text{Trimmed Mean}(\text{list}) = \text{Mean}((\text{list})[\lfloor 0.2 \times \text{len}(\text{list}) \rfloor :: \lfloor 0.2 \times \text{len}(\text{list}) \rfloor ])).\]
TABLE I: Four systems with Volta, Pascal, Maxwell, and Kepler GPUs are selected for evaluation.

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

TABLE II: 37 TensorFlow image classification models from MLPerf [2], AI-Matrix [3], and TensorFlow Slim [19] are used for evaluation and are sorted by accuracy. We measured the online latency, 90th percentile latency, and maximum throughput at the optimal batch size for each model.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Top 1 Accuracy</th>
<th>Graph Size (MB)</th>
<th>Online Trimmed/Mean Latency (ms)</th>
<th>Online 90th Percentile Latency (ms)</th>
<th>Max Throughput (Inputs/Sec)</th>
<th>Optimal Batch Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inception_ResNet_v2</td>
<td>80.40</td>
<td>214</td>
<td>23.95</td>
<td>24.2</td>
<td>346.6</td>
<td>128</td>
</tr>
<tr>
<td>2</td>
<td>Inception_v4</td>
<td>80.20</td>
<td>163</td>
<td>17.36</td>
<td>17.6</td>
<td>436.7</td>
<td>128</td>
</tr>
<tr>
<td>3</td>
<td>Inception_v3</td>
<td>78.00</td>
<td>91</td>
<td>9.2</td>
<td>9.48</td>
<td>811.0</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>ResNet_v2_152</td>
<td>77.70</td>
<td>231</td>
<td>14.44</td>
<td>14.65</td>
<td>466.8</td>
<td>256</td>
</tr>
<tr>
<td>5</td>
<td>ResNet_v2_101</td>
<td>77.00</td>
<td>170</td>
<td>10.31</td>
<td>10.55</td>
<td>671.7</td>
<td>256</td>
</tr>
<tr>
<td>6</td>
<td>ResNet_v1_152</td>
<td>76.80</td>
<td>230</td>
<td>13.67</td>
<td>13.9</td>
<td>541.3</td>
<td>256</td>
</tr>
<tr>
<td>7</td>
<td>MLPerf_ResNet50_v1.5</td>
<td>75.93</td>
<td>230</td>
<td>14.58</td>
<td>14.72</td>
<td>468.0</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td>ResNet_v1_101</td>
<td>75.40</td>
<td>170</td>
<td>9.93</td>
<td>10.08</td>
<td>774.7</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>AI_Matrix_ResNet152</td>
<td>75.30</td>
<td>98</td>
<td>6.17</td>
<td>6.35</td>
<td>1,119.7</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>ResNet_v2_50</td>
<td>75.20</td>
<td>98</td>
<td>6.31</td>
<td>6.41</td>
<td>1,284.6</td>
<td>256</td>
</tr>
<tr>
<td>11</td>
<td>ResNet_v1_50</td>
<td>74.38</td>
<td>98</td>
<td>6.11</td>
<td>6.25</td>
<td>1,060.3</td>
<td>256</td>
</tr>
<tr>
<td>12</td>
<td>AI_Matrix_ResNet50</td>
<td>73.90</td>
<td>43</td>
<td>6.28</td>
<td>6.56</td>
<td>2,032.0</td>
<td>128</td>
</tr>
<tr>
<td>13</td>
<td>Inception_v2</td>
<td>73.29</td>
<td>31</td>
<td>11.17</td>
<td>11.49</td>
<td>846.4</td>
<td>32</td>
</tr>
<tr>
<td>14</td>
<td>AI_Matrix_DenseNet121</td>
<td>71.68</td>
<td>17</td>
<td>2.46</td>
<td>2.66</td>
<td>2,576.4</td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>MLPerf-MobileNet_v1</td>
<td>71.50</td>
<td>528</td>
<td>22.43</td>
<td>22.59</td>
<td>687.5</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>VGG16</td>
<td>71.10</td>
<td>548</td>
<td>23.0</td>
<td>23.31</td>
<td>593.4</td>
<td>256</td>
</tr>
<tr>
<td>17</td>
<td>VGG19</td>
<td>70.90</td>
<td>16</td>
<td>2.59</td>
<td>2.75</td>
<td>2,580.6</td>
<td>128</td>
</tr>
<tr>
<td>18</td>
<td>MobileNet_v1.0.224</td>
<td>70.01</td>
<td>16</td>
<td>5.43</td>
<td>5.55</td>
<td>2,464.5</td>
<td>128</td>
</tr>
<tr>
<td>19</td>
<td>AI_Matrix_GoogleNet</td>
<td>70.00</td>
<td>16</td>
<td>5.25</td>
<td>5.41</td>
<td>2,576.6</td>
<td>128</td>
</tr>
<tr>
<td>20</td>
<td>MobileNet_v1.1.92</td>
<td>69.80</td>
<td>26</td>
<td>5.27</td>
<td>5.41</td>
<td>951.7</td>
<td>8</td>
</tr>
<tr>
<td>21</td>
<td>Inception_v1</td>
<td>68.70</td>
<td>27</td>
<td>6.05</td>
<td>6.17</td>
<td>3,183.7</td>
<td>64</td>
</tr>
<tr>
<td>22</td>
<td>BVLC_GoogLeNet</td>
<td>68.40</td>
<td>10</td>
<td>2.48</td>
<td>2.61</td>
<td>4,240.5</td>
<td>64</td>
</tr>
<tr>
<td>23</td>
<td>MobileNet_v1.0.75_224</td>
<td>68.00</td>
<td>16</td>
<td>2.57</td>
<td>2.74</td>
<td>4,187.8</td>
<td>64</td>
</tr>
<tr>
<td>24</td>
<td>MobileNet_v1.1.0.160</td>
<td>67.20</td>
<td>10</td>
<td>2.42</td>
<td>2.6</td>
<td>5,569.6</td>
<td>64</td>
</tr>
<tr>
<td>25</td>
<td>MobileNet_v1.0.75_192</td>
<td>66.30</td>
<td>10</td>
<td>2.48</td>
<td>2.65</td>
<td>6,743.2</td>
<td>64</td>
</tr>
<tr>
<td>26</td>
<td>MobileNet_v1.1.0_128</td>
<td>65.20</td>
<td>16</td>
<td>2.29</td>
<td>2.46</td>
<td>3,460.8</td>
<td>128</td>
</tr>
<tr>
<td>27</td>
<td>MobileNet_v1.0.5_224</td>
<td>63.30</td>
<td>5.2</td>
<td>2.39</td>
<td>2.58</td>
<td>3,346.5</td>
<td>64</td>
</tr>
<tr>
<td>28</td>
<td>MobileNet_v1.0.75_128</td>
<td>62.10</td>
<td>10</td>
<td>2.48</td>
<td>2.44</td>
<td>8,378.4</td>
<td>64</td>
</tr>
<tr>
<td>29</td>
<td>MobileNet_v1.0.5_192</td>
<td>61.70</td>
<td>5.2</td>
<td>2.48</td>
<td>2.67</td>
<td>4,453.2</td>
<td>64</td>
</tr>
<tr>
<td>30</td>
<td>MobileNet_v1.0.5_160</td>
<td>59.10</td>
<td>5.2</td>
<td>2.42</td>
<td>2.58</td>
<td>6,148.7</td>
<td>64</td>
</tr>
<tr>
<td>31</td>
<td>BVLC_AlexNet</td>
<td>57.10</td>
<td>233</td>
<td>2.33</td>
<td>2.5</td>
<td>2,495.8</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>MobileNet_v1.0.5_128</td>
<td>56.30</td>
<td>5.2</td>
<td>2.21</td>
<td>2.33</td>
<td>8,924.0</td>
<td>64</td>
</tr>
<tr>
<td>33</td>
<td>MobileNet_v1.0.25_224</td>
<td>49.80</td>
<td>1.9</td>
<td>2.46</td>
<td>2.46</td>
<td>5,257.9</td>
<td>64</td>
</tr>
<tr>
<td>34</td>
<td>MobileNet_v1.0.25_192</td>
<td>47.70</td>
<td>1.9</td>
<td>2.39</td>
<td>2.44</td>
<td>7,135.7</td>
<td>64</td>
</tr>
<tr>
<td>35</td>
<td>MobileNet_v1.0.25_160</td>
<td>45.50</td>
<td>1.9</td>
<td>2.39</td>
<td>2.44</td>
<td>10,081.5</td>
<td>256</td>
</tr>
<tr>
<td>36</td>
<td>MobileNet_v1.0.25_128</td>
<td>41.50</td>
<td>1.9</td>
<td>2.28</td>
<td>2.46</td>
<td>10,707.6</td>
<td>256</td>
</tr>
</tbody>
</table>

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 3 are favorable in the online inference scenarios, and models closer to the upper right corner (high throughput and high accuracy) in Figure 4 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 online latency and maximum throughput (Figures 3 and 4 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 model throughput scales with batch size (referred to as *throughput scalability*) and present this model characteristic in Figure 6. 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...
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 I and the results are shown in Figure 5. 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 [20], 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.

### B. 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 a 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 P8 GPU systems with batch size 64. The results are shown in Figure 7. 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 found 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

---

**TABLE III: 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.**

<table>
<thead>
<tr>
<th>Layer Index</th>
<th>Layer Name</th>
<th>Layer Type</th>
<th>Layer Shape</th>
<th>Dominant GPU Kernel(s) Name</th>
<th>Latency (ms)</th>
<th>Alloc Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>208</td>
<td>conv2d_48/Conv2D</td>
<td>Conv2D</td>
<td>(256, 512, 7, 7)</td>
<td>volta_cgemm_32x32_tn</td>
<td>7.59</td>
<td>25.7</td>
</tr>
<tr>
<td>221</td>
<td>conv2d_51/Conv2D</td>
<td>Conv2D</td>
<td>(256, 512, 7, 7)</td>
<td>volta_cgemm_32x32_tn</td>
<td>7.57</td>
<td>25.7</td>
</tr>
<tr>
<td>195</td>
<td>conv2d_45/Conv2D</td>
<td>Conv2D</td>
<td>(256, 512, 7, 7)</td>
<td>volta_scudnn_128x128_relu_interior_nn_vl</td>
<td>5.67</td>
<td>25.7</td>
</tr>
<tr>
<td>3</td>
<td>conv2d/Conv2D</td>
<td>Conv2D</td>
<td>(256, 64, 112, 112)</td>
<td>volta_scudnn_128x64_relu_interior_nn_vl</td>
<td>5.08</td>
<td>82.1</td>
</tr>
<tr>
<td>113</td>
<td>conv2d_26/Conv2D</td>
<td>Conv2D</td>
<td>(256, 256, 14, 14)</td>
<td>volta_scudnn_128x64_relu_interior_nn_vl</td>
<td>4.67</td>
<td>51.4</td>
</tr>
</tbody>
</table>

---

**Fig. 3:** Model accuracy vs online latency. The numbers correspond to the model ID and the size of each circle is proportional to the number of parameters in the model.

**Fig. 4:** Model accuracy vs maximum throughput. The numbers correspond to the model ID and the size of each circle is proportional to the number of parameters in the model.

**Fig. 5:** The batched latency of ResNet_50 across the GPUs and CPUs listed in Table I.

**Fig. 6:** The throughput speedup (over batch size 1) heatmap across batch sizes on AWS P3 for the 37 models in Table II. The y-axis shows the batch size, whereas the x-axis shows the model ID.
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 for 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.

C. 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. 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 kernels calls to each layer; i.e. tell which GPU kernels are executed by a certain layer. Table III 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.

VI. RELATED WORK

To the authors’ knowledge, this is the first paper 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 F1-10. For example, while there is active work [2], [21] on proposing benchmark suites, reference workloads, and analysis. These provide F7 a set of benchmarking scenarios and a simple manual mechanism for F3 analysis and reporting. Models within these benchmarks can be consumed by MLModelScope, and we have shown analysis which use the benchmark-provided models. Other related work are purely model serving platforms [22], [23] which address F4 scalable evaluation and possibly F5 artifact versioning but nothing else. Finally, systems such as as [24], [5], [25], [26] maintain either F1 reproducible or F2 consistent evaluation by track the model and data from their use in training till deployment, or by placing all the logic within a container.

VII. CONCLUSION AND FUTURE WORK

Evaluating, comparing, and analyzing the performance of DL innovations is critical for their adoption. This paper first identified 10 design objectives of a DL benchmarking platform. It then described 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 demonstrated 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.
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.

REFERENCES