The Design and Implementation of a Scalable DL Benchmarking Platform

Cheng Li*
University of Illinois Urbana-Champaign
Urbana, Illinois
cli99@illinois.edu

Abdul Dakkak*
University of Illinois Urbana-Champaign
Urbana, Illinois
dakkak@illinois.edu

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

Wen-mei Hwu
University of Illinois Urbana-Champaign
Urbana, Illinois
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, but lacks a DL benchmarking platform to facilitate evaluation and comparison of DL innovations, be it models, frameworks, libraries, or hardware. Due to the lack of a benchmarking platform, the current practice of evaluating the benefits of proposed DL innovations is both arduous and error-prone — stifling the adoption of the innovations.

In this work, we first identify 10 design features which are desirable within a DL benchmarking platform. These features include: performing the evaluation in a consistent, reproducible, and scalable manner, being framework and hardware agnostic, supporting real-world benchmarking workloads, providing in-depth model execution inspection across the HW/SW stack levels, etc. We then propose MLModelScope, a DL benchmarking platform design that realizes the 10 objectives. MLModelScope proposes a specification to define DL model evaluations and techniques to provision the evaluation workflow using the user-specified HW/SW stack. MLModelScope defines abstractions for frameworks and supports board range of DL models and evaluation scenarios. We implement MLModelScope as an open-source project with support for all major frameworks and hardware architectures. Through MLModelScope's evaluation and automated analysis workflows, we performed case-study analyses 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 demonstrated how MLModelScope's tracing capability gives a holistic view of model execution and helps pinpoint bottlenecks.

1 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, compilers, or hardware. DL innovations are introduced at such a rapid pace [10] that being able to evaluate and compare these innovations in a timely manner is critical for their adoption. As a result, there have been concerted community efforts in developing DL benchmark suites [1, 6, 14, 24] 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 increasingly being proposed, there is an urging need for a DL benchmarking platform that consistently evaluates and compares different DL models across HW/SW stacks, while coping with the fast-paced and diverse landscape of DL.

As a fledgling field, the benchmarking platform design for DL faces new challenges and requirements. 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 [15, 23, 31]. Thus, the benchmarking platform design must guarantee a **F1 reproducible evaluation** along with **F2 consistent evaluation**.

Aside from **F1-2**, the design should: be **F3 frameworks and hardware agnostic** to support model evaluation using diverse HW/SW stacks; be capable of performing **F4 scalable evaluation** across systems to cope with the large number of evaluations due to the diverse model/HW/SW combinations; support different **F7 benchmarking scenarios**

[1] The two authors contributed equally to this paper.

which mimic the real-world workload exhibited in online, offline, and interactive applications; have a F8 **benchmarking analysis and reporting** workflow which analyzes benchmarking results across evaluation runs and generates summary reports; enable F9 **model execution inspection** to identify bottlenecks within a model-, framework-, and system-level components. Other features such as: F5 **artifact versioning**, F6 **efficient evaluation workflow**, and F10 **different user interfaces** are also desirable to increase the design's scalability and usability. We discuss the design objectives in detail in Section 3.

In this paper, we propose *MLModelScope*, a scalable DL benchmarking platform design that realizes the above 10 objectives and facilitates benchmarking, comparison, and understanding of DL model executions. MLModelScope 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 common abstraction API across frameworks; providing across-stack tracing capability that allows users to inspect model execution at different HW/SW abstraction levels; defining an automated evaluation analysis workflow for analyzing and reporting evaluation results; and, finally, exposing the capabilities through a web and command-line interface.

We implement MLModelScope and integrate it with the Caffe [21], Caffe2 [20], CNTK [34], MXNet [4], PyTorch [29], TensorFlow [40], TensorFlow Lite [37], and TensorRT [5] frameworks. MLModelScope runs on ARM, PowerPC, and x86 and supports CPU, GPU, and FPGA execution. We bootstrap MLModelScope with over 300 built-in models covering different DL tasks such as image classification, object detection, semantic segmentation, etc. MLModelScope is open-source, extensible, and customizable — coping with the fast-paced DL landscape. To the authors' knowledge, this paper is the first to describe the design and implementation of a scalable DL benchmarking platform.

We showcase MLModelScope's benchmarking, inspection, and analysis capabilities using several case studies. We use MLModelScope to evaluate 37 DL models and systematically compare their performance using 4 systems under different benchmarking scenarios. We perform comparisons to understand the correlation between a model's accuracy, its size, and its 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, users can easily digest the evaluation results produced by MLModelScope to understand model-, framework-, and system-level bottlenecks.

This paper describes the design and implementation of MLModelScope and is structured as follows. Section 2 gives a background. Section 3 describes the objectives of MLModelScope. Section 4 proposes the MLModelScope design which addresses these objectives and describes its implementation. Section 5 performs in-depth evaluations using MLModelScope. Section 6 details related work before we conclude in Section 7.

2 Background

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

2.1 DL Model Evaluation Pipeline

A DL model evaluation pipeline performs input pre-processing, followed by model prediction and output post-processing (Figure 3). 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 step decodes the input image into a tensor of dimensions $[batch, height, width, channel]$ ($[N, H, W, C]$), then performs resizing, normalization, etc. The image classification model's output is a tensor of dimensions $[batch * numClasses]$ which is sorted to get the top K predictions (label with probability). A DL *model* is defined by its graph topology and its weights. The graph topology is defined as a set of nodes where each node is a function operator with the implementation provided by a *framework* (e.g. TensorFlow, MXNet, PyTorch). The framework acts as a "runtime" for the 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.

2.2 Current DL Benchmarking

While there has been a drive to provide reference DL benchmarks [1, 14, 24], the current benchmarking effort is still scattered, lacks a standard benchmarking methodology, and revolves around a series of scripts that evaluate a model on a local system. 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 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 [15, 17, 23] benchmarking results. Due to the daunting effort to perform fair benchmarking, DL innovations proposed have outpaced researchers' ability to compare and analyze them [10].

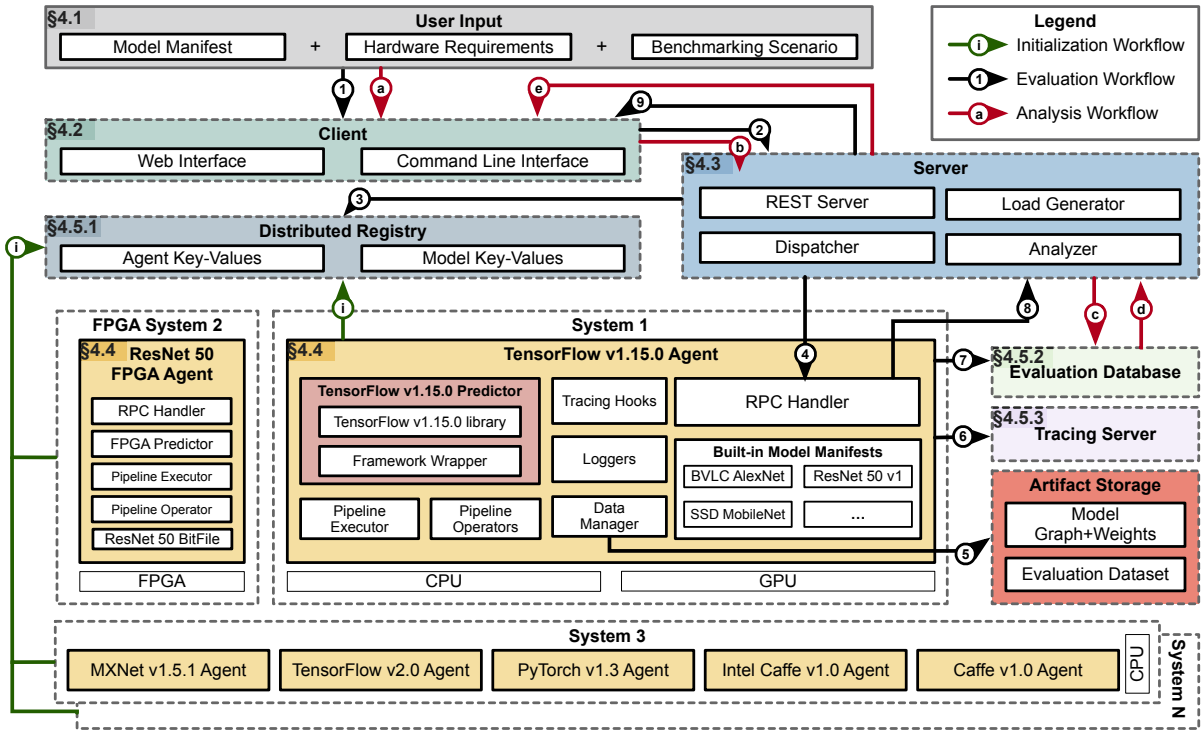

Figure 1. The MLModelScope design and workflows.

3 Design Objectives

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

F1 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 [15, 17, 19, 23, 31, 36]. Thus, all aspects of a model evaluation must be specified and provisioned by the platform design to guarantee reproducible evaluation.

F2 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 components, making it difficult to quantify or isolate the benefits of an individual component (be it model, framework, or other SW/HW components). A fair apple-to-apple comparison between model executions requires a consistent evaluation methodology rather than running ad-hoc scripts for each. Thus the design should have a well-defined benchmarking specification for all models and maximize the common code base that drives model evaluations.

F3 Framework/Hardware Agnostic — The DL landscape is diverse and there are many DL frameworks (e.g. TensorFlow, MXNet, PyTorch) and hardware (e.g. CPU, GPU, FPGA). Each has its own use scenarios, features, and performance characteristics. To have broad support of model evaluation, the design must support different frameworks and hardware. Furthermore, the design must be valid without modifications to the frameworks.

F4 Scalable Evaluation — DL innovations, such as models, frameworks, libraries, compilers, and hardware accelerators are introduced at a rapid pace [10, 18]. Being able to quickly evaluate and compare the benefits of DL innovations is critical for their adoption. Thus the ability to perform DL evaluations with different model/HW/SW setups in parallel and have a centralized management of the benchmarking results is highly desired. For example, choosing the best hardware out of N candidates for a model is ideally performed in parallel and the results should be automatically gathered for comparison.

F5 Artifact Versioning — DL frameworks are continuously updated by the DL community, e.g. the recent versions TensorFlow at the time of writing are v1.15 and v2.0. There are many unofficial variants of models, frameworks, and datasets as researchers might update or modify them to suite

their respective needs. To enable management and comparison of model evaluations using different DL artifacts (models, frameworks, and datasets), the artifacts used for evaluation within a benchmarking platform should be versioned.

F6 Efficient Evaluation Workflow — Before model inference can be performed, the input data has to be loaded into memory and the pre-processing stage transforms it into a form that the model expects. After the model prediction, the post-processing stage transforms the model's output(s) to a form that can be used to compute metrics. The input 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.

F7 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.

F8 Benchmarking Analysis and Reporting — Benchmarking produces raw data which needs to be correlated and analyzed to produce human-readable results. An automated mechanism to summarize and visualize these results within a benchmarking platform can help users quickly understand and compare the results. Therefore, the design should have a benchmarking result analysis and reporting workflow.

F9 Model Execution Inspection — Benchmarking is often followed by performance optimization. However, the complexity of DL model evaluation makes performance debugging challenging as each level within the HW/SW abstraction hierarchy can be a suspect when things go awry. Current model execution inspection methods rely on the use of a concoction of profiling tools (e.g. Nvidia's Nsight System or Intel's Vtune). Each profiling tool captures a specific aspect of the HW/SW stack and researchers manually correlate the results to get an across-stack view of the model execution profile. To ease inspecting model execution bottlenecks, the benchmarking platform design should provide tracing capability at all levels of HW/SW stack.

F10 Different User Interfaces — While the command-line is the most common interface in the current benchmarking suites, having other UIs, such as web UI, to accommodate other use cases can greatly boost productivity. While a command-line interface is often used in scripts to quickly perform combinational evaluations across models, frameworks, and systems, a web UI, on the other hand, can serve as a "push-button" solution 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.

```yaml
name: MLPerf_ResNet50_v1.5 # model name
version: 1.0.0 # semantic version of the model
description: ...
framework: # framework information
  name: TensorFlow
  version: '>=1.12.0 <2.0' # framework ver constraint
inputs: # model inputs
  - type: image  # first input modality
    layer_name: 'input_tensor'
    element_type: float32
    steps: # pre-processing steps
      - decode:
          data_layout: NHWC
          color_mode: RGB
      - resize:
          dimensions: [3, 224, 224]
          method: bilinear
          keep_aspect_ratio: true
      - normalize:
          mean: [123.68, 116.78, 103.94]
          rescale: 1.0
outputs: # model outputs
  - type: probability # first output modality
    layer_name: prob
    element_type: float32
    steps: # post-processing steps
      - argsort:
          labels_url: https://.../synset.txt
preprocess: [[code]]
postprocess: [[code]]
model: # model sources
  base_url: https://zenodo.org/record/2535873/files/
  graph_path: resnet50_v1.pb
  checksum: 7b94a2da05d...23a46bc08886
attributes: # extra model attributes
  training_dataset:  # dataset used for training
    - name: ImageNet
    - version: 1.0.0
```

Listing 1. The `MLPerf_ResNet50_v1.5`'s model manifest contains all information needed to run the model evaluation using TensorFlow on CPUs or GPUs.

4 MLModelScope Design and Implementation

We propose MLModelScope, a DL benchmarking platform design that achieves the objectives F1-10 set out in Section 3. To achieve F4 scalable evaluation, we design MLModelScope as a distributed platform. To enable F7 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 1 shows the high level components which include:

- ***User Inputs*** — are the required inputs for model evaluation including: 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 at least 32GB of RAM and an NVIDIA V100 GPU), and the benchmarking scenario to employ.
- ***Client*** — is either the web UI or command-line interface which users use to supply their inputs and initiate the model evaluation by sending a REST request to the MLModelScope server.
- ***Server*** — acts on the client requests and performs REST API handling, dispatching the model evaluation tasks to MLModelScope agents, generating benchmark workloads

based on benchmarking scenarios, and analyzing the evaluation results.

• ***Agents*** — runs on different systems of interest and perform model evaluation based on requests sent by the MLModelScope server. Each agent includes logic for downloading model assets, performing input pre-processing, using the framework predictor for inference, and performing post-processing. An agent can be run within a container or as a local process. Aside from the framework predictor, all code within 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 running agents and available models), an evaluation database (a database containing evaluation results), a tracing server (a server to publish profile events captured during an evaluation), and an artifact storage server (a data store repository containing model assets and datasets).

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

4.1 User Input

All aspects of DL evaluation — model, software stack, system, and benchmarking scenario — must be specified to MLModelScope for it to enforce F1 reproducible and F2 consistent evaluation. To achieve this, MLModelScope defines a benchmarking specification covering the 4 aspects of evaluation. A model in MLModelScope is specified using a *model manifest*, and a software stack is specified using a *framework manifest*. The manifests are textual specification in YAML [2] format. The system and benchmarking scenario are user-specified options when the user initiates an evaluation. The benchmarking specification is not tied to a certain framework or hardware, thus enabling F3. As the model, software stack, system, and benchmarking scenario specification are decoupled, one can easily evaluate the different combinations, enabling F4. 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 MLModelScope agents (Section 4.4). For these built-in models, a user can specify the model and framework's name and version in place of the manifest for ease of use. MLModelScope also provides ready-made Docker containers to be used in the framework manifests. These containers are hosted on Docker hub.

4.1.1 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 that a model can execute on. For example, an ONNX model may work across all frameworks and therefore has no constraint, but other models may only work for TensorFlow versions greater than 1.2.0 but less than 2 (e.g. Lines 4–6 in Listing 1). This allows MLModelScope to support models to target specific versions of a framework and 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-interpret-er [32] and passes the data arguments by reference. The pre- and post-processing functions are general; i.e. the

functions may import external Python modules or download and invoke external scripts. By allowing arbitrary processing functions, MLModelScope works with existing processing codes and is capable of supporting arbitrary input/output modalities.

Built-in Pre- and Post-Processing — An alternative way of specifying pre- and post-processing is by defining them as a series of built-in pre- and post-processing pipeline steps (i.e. *pipeline operators*) within the model manifest. For example, our MLModelScope implementation provides common pre-processing image operations (e.g. image decoding, resizing, and normalization) and post-processing operations (e.g. ArgSort, intersection over union, etc.) which are widely used within vision models. Users can use these 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 steps are executed in the order they are specified in the model manifest. The use of built-in processing and function processing pipelines are mutually exclusive.

Model Assets — The data required by the model are specified in the model manifest file; i.e. the graph (the `graph_path`) and weights (the `weights_path`) fields. The model assets can reside within MLModelScope's artifact repository, on the web, or the local file system of the MLModelScope agent. If the model assets are remote, then they are downloaded on demand and cached on the local file system. For frameworks (such as TensorFlow and PyTorch) which use a single file for both the model graph and weights (in deployment), the weights field is omitted from the manifest. For example, the TensorFlow `ResNet50 v1.5` model assets in Listing 1 are stored on the `Zenodo` [41] website (Lines 31-34) and are downloaded prior to evaluation.

4.1.2 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. As the core of the software stack, the framework name and version constraints are specified. To maintain the software stack, and guarantee isolation, the user can further specify docker containers using the `containers` field. Multiple containers can be specified to accommodate different systems (e.g. CPU or GPUs). At the MLModelScope initialization phase (ⓘ), MLModelScope agents (described in Section 4.4) register themselves by publishing their HW/SW stack information into the distributed registry (described in Section 4.5.1). The MLModelScope server uses this information during the agent resolution process. The server finds MLModelScope agents satisfying the user's hardware specification and model/framework requirements. Evaluations are then run on one of (or, at the user request, all of) the agents. If the user omits the framework manifest in the user input, the MLModelScope server resolves the agent constraints using the model manifest and system information. This allows MLModelScope to support evaluation on FPGA systems which do not use containers.

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

Listing 2. An example TensorFlow framework manifest, which contains the software stacks (containers) to run the model evaluation across CPUs or GPUs.

4.1.3 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 4.3).

4.2 MLModelScope Client

A user initiates a model ① evacuation or ⓐ analysis though the MLModelScope *client*. To enable F10, the client can be either a website or a command-line tool that users interact with. The client communicates with the MLModelScope server through REST API and sends user evaluation requests. The web user interface allows users to specify a model evaluation through simple clicks and is designed to help users who do not have much DL experience. For example, for users not familiar with the different models registered, MLModelScope allows users to select models based on the application area — this lowers the barrier of DL usage. The command-line interface is provided for those interested in automating the evaluation and profiling process. Users can develop other clients that use the REST API to integrate MLModelScope within their AI applications.

4.3 MLModelScope Server

The MLModelScope *server* interacts with the MLModelScope client, agent, the middleware. It uses REST API to communicate with the MLModelScope clients and middleware, and gRPC (Listing 4) to interact with the MLModelScope agents. To enforce F4, the MLModelScope server can be load balanced to avoid it being a bottleneck.

In the ①-⑨ *evaluation workflow*, the server is responsible for ② accepting tasks from the MLModelScope client, ③

```
// Opens a predictor.
ModelHandle ModelLoad(OpenRequest);
// Close an open predictor.
Error ModelUnload(ModelHandle);
// Perform model inference on user data.
PredictResponse Predict(ModelHandle, PredictRequest, ↩
      PredictOptions);
```

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

querying the distributed registry and resolving the user-specified constraints to find MLModelScope agents capable of evaluating the request, ④ dispatching the evaluation task to the resolved agent(s) and generating loads for the evaluation, ⑧ collecting the evaluation summary from the agent(s), and ⑨ returning the result summary to the client. The load generator is placed on the server to avoid other programs interfering with the evaluation being measured and to emulate real-world scenarios such as cloud serving (F7).

In the a-e *analysis workflow*, the server again a-b takes the user input, but, rather than performing evaluation, it ⓒ queries the evaluation database (Section 4.5.2), and then aggregates and analyzes the evaluation results. MLModelScope enables F8 through an across-stack analysis pipeline. It ⓓ consumes the benchmarking results and the profiling traces in the evaluation database and performs the analysis. Then the server ⓔ sends the analysis result to the client. The consistent profiling and automated analysis workflows in MLModelScope allow users to systematically compare across models, frameworks, and system offerings.

4.4 Agent and Framework Predictor

A MLModelScope *agent* is a model serving process that is run on a system of interest (within a container or on bare metal) and handles requests from the MLModelScope server. MLModelScope agents continuously listen for jobs and communicate with the MLModelScope server through gRPC [16] as shown in Listing 4. A *framework predictor* resides within a MLModelScope agent and is a wrapper around a framework and links to the framework's C library.

During the initialization phase (ⓘ), a MLModelScope agent publishes its built-in models and HW/SW information to the MLModelScope distributed registry. To perform the assigned evaluation task, the agent first ⑤ downloads the required evaluation assets using the *data manager*, it then executes the model evaluation pipeline which performs the pre-processing, calls the framework's predictor for inference and then preforms the post-processing. If profiling is enabled, the trace information is published to the ⑥ tracing server to get aggregated into a single profiling trace. ⑦ the benchmarked result and the profiling trace are published to the evaluation database. Aside from the framework predictor, all the other code — the data manager, pipeline executor, and tracing hooks — are shared across agents for different frameworks. While the default setup of MLModelScope is to run each agent on a separate system, the design does not preclude one from running agents on the same system as separate processes.

4.4.1 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 specified in the model manifest, the data manager validates the checksum of the asset before using a cached asset or after downloading the asset. Model assets are stored using the frameworks' corresponding deployment format. For datasets, MLModelScope supports the use of TensorFlow's TFRecord [38] and MXNet's RecordIO [33]. These dataset formats are optimized for static data and lays out the elements within the dataset as contiguous binary data on disk to achieve better read performance.

4.4.2 Pipeline Executor and Operators

To enable F6 efficient evaluation workflow, MLModelScope leverages a streaming data processing pipeline design to perform the model evaluation. The pipeline is composed of *pipeline operators* which are mapped onto light-weight threads to make efficient use multiple CPUs as well as to overlap I/O with compute. Each operator within the pipeline forms a producer-consumer relationship by receiving values from the upstream operator(s) (via inbound streams), applies the specified function on the incoming data and usually producing new values, and propagates values downstream (via outbound streams) to the next operator(s). The pre- and post-processing operations, as well as the model inference, form the operators within the model evaluation pipeline.

4.4.3 Framework Predictor

Frameworks provide different APIs (usually across programming languages e.g. C/C++, Python, Java) to perform inference. To enable F2 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 is designed to bind to the frameworks' C API to avoid the overhead of using scripting languages. We demonstrate this overhead by comparing model inference using Python and the C API. We used TensorFlow `1.13.0`

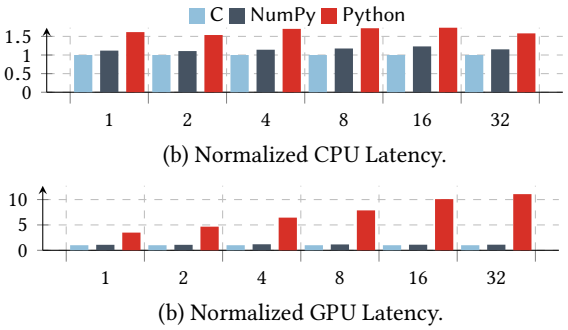

Figure 2. The `tf.Session.Run` execution time (normalized to C) across batch sizes for `Inception-v3` inference on CPU and GPU using TensorFlow with C, Python using NumPy arrays (NumPy), and Python using native lists (Python).

```
service Predict {
message PredictOptions {
  enum TraceLevel {
    NONE      = 0;
    MODEL     = 1;  // steps in the evaluation pipeline
    FRAMEWORK = 2;  // layers within the framework and above
    SYSTEM    = 3;  // the system profilers and above
    FULL      = 4;  // includes all of the above
  }
  TraceLevel trace_level = 1;
  Options    options     = 2;
}
message OpenRequest {
  string model_name                  = 1;
  string model_version               = 2;
  string framework_name              = 3;
  string framework_version           = 4;
  string model_manifest              = 5;
  BenchmarkScenario benchmark_scenario = 6;
  PredictOptions    predict_options    = 7;
}
// Opens a predictor and returns a PredictorHandle.
rpc Open(OpenRequest) returns (PredictorHandle){}
// Close a predictor and clear its memory.
rpc Close(PredictorHandle) returns (CloseResponse) {}
// Predict receives a stream of user data and runs
// the predictor on each element of the data according
// to the provided benchmark scenario.
rpc Predict(PredictorHandlePredictorHandle, UserInput) ⟷
     returns (FeaturesResponse) {}
}
```

Listing 4. MLModelScope's minimal gRPC interface in protocol buffer format.

compiled from source with cuDNN 7.4 and CUDA Runtime 10.0.1 on the `Tesla_V100` system (Amazon EC2 P3 instance) in Table 1. Figure 2 shows the normalized inference latency across language environments on GPUs and CPUs across batch sizes. On CPU, using Python is 64% and NumPy is 15% slower than C; whereas on GPU Python is $3-11\times$ and NumPy is 10% slower than C. For Python, the overhead is proportional to the input size and is due to TensorFlow internally having to unbox the Python linked list objects and create a numeric buffer that can be used by the C code. The unboxing is not needed for NumPy since TensorFlow can use NumPy's internal numeric buffer directly. By using the C API directly, MLModelScope can elide measuring overheads due to language binding or scripting language use.

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

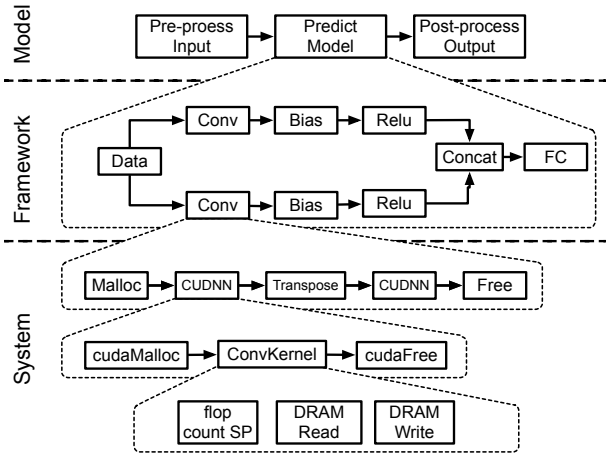

Figure 3. The model inference is defined by the pre-processing, prediction, and post-processing pipeline. A framework executes a model through a network-layer execution pipeline. Layers executed by a framework are pipelines of system library calls. The system libraries, in turn, invoke a chain of primitive kernels that impact the underlying hardware counters.

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.

4.4.4 Tracing Hooks

To enable F9, MLModelScope leverages distributed tracing [35] and captures the profiles at different levels of granularity (model-, framework-, and system-level as shown in Figure 3) using the tracing hooks. A *tracing hook* is a pair of start and end code snippets and follows the standards [28] 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 4.5.3). 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 timestamps of trace events do no need to reflect the actual wall clock time, for example, users may integrate a system simulator and publish simulated time rather than wall-clock time to the tracing server.

Model-level — Tracing hooks are automatically placed around each pipeline operator within the model evaluation pipeline. For example, the tracing hook around the model inference step measures the inference latency.

Framework-level — The tracing hooks at the framework-level leverage the DL frameworks' existing profiling capabilities and does not require modification to the framework source code. For TensorFlow, this option is controlled by the `RunOptions.TraceLevel` setting which is passed to the `TF_SessionRun` function. In MXNet, the `MXSetProfilerState` function toggles the layer profiling. Similar mechanisms exist for other frameworks such as Caffe, Caffe2, PyTorch, and TensorRT. The framework's profile representation is converted and is then published to the tracer server.

System-level — The tracing hooks at the system-level integrate with hardware and system-level profiling libraries to capture fine-grained performance information — CPU and GPU profiles, system traces, and hardware performance counters. For example, the performance counters on systems are captured through integration with PAPI [3] and Linux perf [30] while the GPU profile is captured by integrating with NVIDIA's NVML [26] and CUPTI [8]. Since overhead can be high for system-level profiling, the user can selectively enable/disable the integrated profilers.

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

4.5 Middleware

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

4.5.1 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, solve 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.

4.5.2 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. Since the models are versioned, MLModelScope allows one to track which model version produced the best result.

4.5.3 Tracing Server

The MLModelScope *tracing server* is a distributed tracing server which accepts profiling data published by the MLModelScope agent's trace hooks (Section 4.4.4). Through the innovative use of distributed tracing (originally designed to monitor distributed applications), MLModelScope joins profiling results from different profiling tools and accepts instrumentation markers within application and library code. All profiling data are incorporated into a single profiling timeline. The aggregated profiling trace is consumed by the MLModelScope analysis pipeline and also visualized separately where the user can view the entire timeline and "zoom" into a specific component as shown in Figure 3. As stated in Section 4.4.4, user-specified options control the granularity (model, framework, or system) of the trace events captured (Lines 4 – 9 in Listing 4).

4.6 Extensibility and Customization

MLModelScope is built from a set of 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 but 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 a MLModelScope predictor. The predictor interface is defined by a set of 3 functions — one to open a model, another to perform the inference, and finally, one to close the model — as shown in Listing 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 tracing hooks captures only some CUDA runtime API, GPU activities (kernels and memory copy), and GPU metrics. They can be configured to capture other GPU activities and metrics, or NVTX markers. Moreover, users can integrate other system

profilers into MLModelScope by implementing the tracing hook interface (Section 4.4.4).

4.7 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 support 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 can 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 maintain a running version of MLModelScope (omitting the web link due to the blind review process) on a representative set of systems along with the evaluation results of the built-in artifacts. It serves as a portal for the public to evaluate and measure the systems, and to contribute to MLModelScope's artifacts. Using the analysis pipeline, we automatically generated profiling reports for hundreds of models across frameworks. The analysis reports are published as web pages and a sample is available at scalable20.mlmodelscope.org for the reader's inspection.

We implemented a MLModelScope web UI using the React Javascript framework. The web UI interacts with a REST API provided by the server. A video demoing the web UI usage flow is available at https://bit.ly/2N9Z5wR. The REST API can be used by other clients that wish to integrate MLModelScope within their workflow. A MLModelScope command-line client is also available and can be used within shell scripts. The agents also expose a gRPC API which can be used to perform queries to the agents directly.

5 Evaluation

Previous sections discussed in detail how MLModelScope's design and implementation achieves the F1-6 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 1. Unless otherwise noted, all MLModelScope agents are run within a docker container built on top of NVIDIA's TensorFlow NGC v19.06 docker image with the TensorFlow v1.13.1 library. All evaluations were performed using the command-line interface and are run in parallel across the systems.

5.1 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 (shown in Table 2) and systems (shown in Table 1) under different benchmark scenarios. For each model, we measured its trimmed mean latency [1] and 90th percentile latency in online (batch size = 1) inference scenario, and the maximum throughput in batched inference scenario on the `AWS P3` system in Table 1. The model accuracy achieved using the ImageNet [11] validation dataset and model size is listed. A model deployer can use this accuracy and performance information to choose the best model on a system given the accuracy and target latency or throughput objectives.

Model Accuracy, Size, and Performance — We examined the relationship between the model accuracy and both online latency (Figure 5) and maximum throughput (Figure 4). In both figures, the area of the circles is proportional to the model's graph size. In Figure 4 we find a limited correlation between a model's online latency and its accuracy — models taking longer time to run do not necessarily achieve higher accuracies; e.g. model 15 vs 22. While large models tend to have longer online latencies, this is not always true; e.g. model 14 is smaller in size but takes longer to run compared to models 3, 5, 8, etc. Similarly, in Figure 5, we find a limited correlation between a model's accuracy and its maximum throughput — two models with comparable maximum throughputs can achieve quite different accuracies; e.g. models 2 and 17. Moreover, we see from both figures that the graph size (which roughly represents the number of weight values) is not directly correlated to either accuracy or performance. Overall, models closer to the upper left corner (low latency and high accuracy) in Figure 4 are favorable in the online inference scenarios, and models closer to the upper right corner (high throughput and high accuracy) in Figure 5 are favorable in the batched inference scenario. 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 4 and 5 respectively), we observed that models which exhibit good online inference latency do not

[1] Trimmed mean is computed by removing 20% of the smallest and largest elements and computing the mean of the residual; i.e. $\textbf{TrimmedMean}(list)) = \textbf{Mean}(\textbf{Sort}(list)\ [\lfloor 0.2 * \textbf{len}(list)\rfloor :: -\lfloor 0.2 * \textbf{len}(list)\rfloor])$.

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

Table 1. Four systems with Volta, Pascal, Maxwell, and Kepler GPUs are selected for evaluation.

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

Table 2. 37 pre-trained TensorFlow image classification models from MLPerf [24], AI-Matrix [1], and TensorFlow Slim are used for evaluation and are sorted by accuracy. The graph size is the size of the frozen graph for a model. We measured the online latency, 90^{th} percentile latency, maximum throughput in batched inference at the optimal batch size for each model.

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 network architectures can have different throughput scalability — e.g. models 4 and 6, models 5 and 8, and models 10 and 11. In general, smaller models tend to have better throughput scalability. However, there

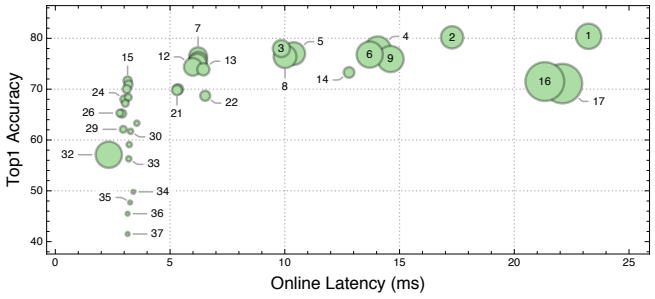

Figure 4. Accuracy vs online latency on AWS P3.

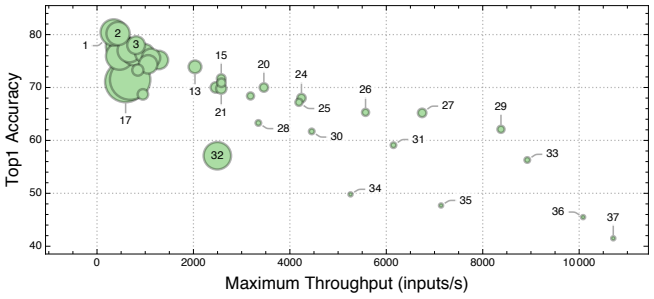

Figure 5. Accuracy vs maximum throughput on AWS P3.

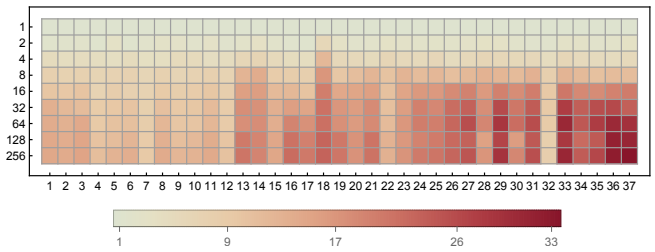

Figure 6. The throughput speedup (over batch size 1) heatmap across batch sizes on AWS P3 for the 37 models in Table 2. The $y-$axis shows the batch size, whereas the $x-$axis shows the model ID.

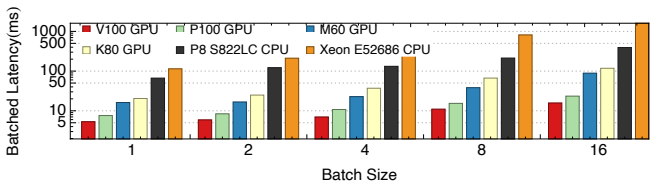

Figure 7. The batched latency of `ResetNet 50` across the GPUs and CPUs listed in Table 1.

are exceptions, for example, the VGG models (16 and 17) are large and have good throughput scalability.

Model Performance Across Systems — Overall, the `ResNet_50` class of models offer a balance between model size, accuracy, performance and are commonly used in practice. Thus we use `ResNet_50` in online inference as an example to show how to use MLModelScope to choose the best system given a model. We evaluated `ResNet_50` across all CPUs and GPUs listed in Table 1 and the results are shown in Figure 7. On the CPU side, IBM S822LC Power8 achieves between 1.7× and 4.1× speedup over Intel Xeon E5-2686. The P8 CPU is more performant than Xeon CPU [12], 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× to 1.7× 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.

5.2 Model Execution Inspection

MLModelScope's evaluation inspection capability helps users to understand the model execution and identify performance bottlenecks. We show this by performing a case study of "cold-start" inference (where the model needs to be loaded into the memory before inference) of `BVLC_AlexNet` (ID = 32). The cold-start inference is common on low-memory systems and in serving schemes that perform one-off evaluation (thus models do not persist in memory).

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

Layer Index	Layer Name	Layer Type	Layer Shape	Dominant GPU Kernel(s) Name	Latency (ms)	Alloc Mem (MB)
208	conv2d_48/Conv2D	Conv2D	⟨256, 512, 7, 7⟩	volta_cgemm_32x32_tn	7.59	25.7
221	conv2d_51/Conv2D	Conv2D	⟨256, 512, 7, 7⟩	volta_cgemm_32x32_tn	7.57	25.7
195	conv2d_45/Conv2D	Conv2D	⟨256, 512, 7, 7⟩	volta_scudnn_128x128_relu_interior_nn_v1	5.67	25.7
3	conv2d/Conv2D	Conv2D	⟨256, 64, 112, 112⟩	volta_scudnn_128x64_relu_interior_nn_v1	5.08	822.1
113	conv2d_26/Conv2D	Conv2D	⟨256, 256, 14, 14⟩	volta_scudnn_128x64_relu_interior_nn_v1	4.67	51.4

Table 3. The ResNet 50 layer information using AWS P3 (Tesla V100 GPU) with batch size 256. The top 5 most time-consuming layers are summarized from the tracing profile. In total, there are 234 layers of which 143 take less than 1ms.

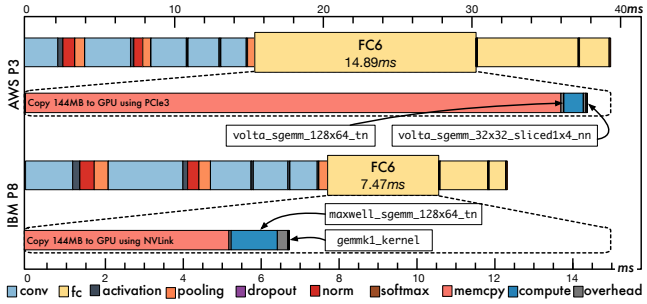

Figure 8. The MLModelScope inspection of "cold-start" `BVLC_AlexNet` inference with batch size 64 running Caffe v0.8 using GPU on AWS P3 and IBM P8 (Table 1). The color-coding of layers signify that they have the same type but does not imply that the layer parameters are the same.

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 copies.

5.3 Benchmarking Analysis and Reporting

To show MLModelScope's benchmarking analysis and reporting capability, we used MLModelScope's analysis workflow to perform an in-depth analysis of the 37 models. All results were generated automatically using MLModelScope and further results are available at scalable20.mlmodelscope.org for the reader's inspection. As an example, we highlight the model-layer-GPU kernel analysis of `ResNet_50` using batch size 256 (the optimal batch size with the maximum throughput) on AWS P3. MLModelScope can capture the layers in a model and correlate the GPU kernels calls to each layer; i.e. tell which GPU kernels are executed by a certain layer. For example, layer index 208 is the most time-consuming layer within the model and 7 GPU kernels are launched by this layer: K1 `volta_cgemm_32x32_tn` taking 6.03 ms, K2 `flip_filter` taking 0.43 ms, K3 `fft2d_r2c_16x16` taking 0.42 ms, K4 `fft2d_c2r_16x16` taking 0.25 ms, K5 `fft2d_r2c_16x16` taking 0.25 ms, K6 `ShuffleInTensor3Simple` taking 0.06 ms, and K7 `compute_gemm_pointers` taking 0.004 ms. K1-5 and K7 are launched by the cuDNN to perform convolution using the FFT algorithm [22]. K6 is launched by TensorFlow and shuffles a layer shape based on a permutation and is used by the TensorFlow convolution layer to convert from TensorFlow's filter format to the cuDNN filter format. Table 3 shows the top 5 most time-consuming layers of `ResNet_50` as well as the dominant kernel (the kernel with the highest latency) within each layer. Through the analysis and summarization workflow, users can easily digest the results and identify understand model-, framework-, and system-level bottlenecks.

6 Related Work

To the authors' knowledge, this 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 on proposing benchmark suites, reference workloads, and analysis [24, 42], they provide F7 a set of benchmarking scenarios and a simple mechanism for F8 analysis and reporting of the results. The models within these benchmarks can be consumed by MLModelScope, and we have shown analysis which uses the benchmark-provided models. Other work are purely model serving platforms [7, 27] which address F4 scalable evaluation and possibly F5 artifact versioning but nothing else. Finally, systems such as as [15, 25, 39] track the model and data from their use in training til deployment and can achieve F1 reproducible and F2 consistent evaluation.

To our knowledge, the most relevant work to MLModelScope is FAI-PEP [13]. FAI-PEP is a DL benchmarking platform targeted towards mobile devices. FAI-PEP aims to solve F1–5 and has limited support of F8 (limited to computing the n^{th} percentile latency and displaying plot of these analyzed latencies). No in-depth profiling and analysis are available within their platform.

7 Conclusion and Future Work

A big hurdle in adopting DL innovations is to evaluate, analyze, and compare their performance. 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, and a sample of the generated reports is available at scalable20.mlmodelscope.org for the reader's inspection. We are further working on maintaining an online public instance of MLModelScope where users can perform the analysis presented without instantiating MLModelScope on their system.

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

[1] AliBaba 2018. AI Matrix. https://aimatrix.ai. Accessed: 2019-10-04.

[2] Oren Ben-Kiki and Clark Evans. 2018. YAML Ain't Markup Language (YAML^{TM}) Version 1.2. http://yaml.org/spec/1.2. Accessed: 2019-10-04.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications* 14, 3 (Aug. 2000), 189–204. https://doi.org/10.1177/109434200001400303

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[5] Zhangxin Chen, Hui Liu, Song Yu, Ben Hsieh, and Lei Shao. 2013. Reservoir Simulation on NVIDIA Tesla GPUs. https://developer.nvidia.com/tensorrt. , 125-133 pages. https://doi.org/10.1090/conm/586/11670 Accessed: 2019-10-04.

[6] Cody Coleman, Matei Zaharia, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, and Chris RÃĺ. 2019. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *SIGOPS Oper. Syst. Rev.* 53, 1 (July 2019), 14–25. https://doi.org/10.1145/3352020.3352024

[7] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *NSDI*. 613–627.

[8] CUPTI 2018. The CUDA Profiling Tools Interface. https://developer.nvidia.com/cuda-profiling-tools-interface. Accessed: 2019-10-04.

[9] Abdul Dakkak, Cheng Li, Simon Garcia de Gonzalo, Jinjun Xiong, and Wen-mei Hwu. 2019. TrIMS: Transparent and Isolated Model Sharing for Low Latency Deep Learning Inference in Function-as-a-Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, IEEE, 372–382. https://doi.org/10.1109/cloud.2019.00067

[10] Jeff Dean, David Patterson, and Cliff Young. 2018. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro* 38, 2 (March 2018), 21–29. https://doi.org/10.1109/mm.2018.112130030

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, IEEE, 248–255. https://doi.org/10.1109/cvpr.2009.5206848

[12] Vadim V Elisseev, Milos Puzovic, and Eun Kyung Lee. 2018. A Study on Cross-Architectural Modelling of Power Consumption Using Neural Networks. *Supercomputing Frontiers and Innovations* 5, 4 (2018), 24–41.

[13] FAI-PEP 2019. Facebook AI Performance Evaluation Platform. https://github.com/facebook/FAI-PEP. Accessed: 2019-10-04.

[14] Wanling Gao, Fei Tang, Lei Wang, Jianfeng Zhan, Chunxin Lan, Chunjie Luo, Yunyou Huang, Chen Zheng, Jiahui Dai, Zheng Cao, Daoyi Zheng, Haoning Tang, Kunlin Zhan, Biao Wang, Defei Kong, Tong Wu, Minghe Yu, Chongkang Tan, Huan Li, Xinhui Tian, Yatao Li, Junchao Shao, Zhenyu Wang, Xiaoyu Wang, and Hainan Ye. 2019. AIBench: An Industry Standard Internet Service AI Benchmark Suite. arXiv:cs.CV/1908.08998

[15] Sindhu Ghanta, Lior Khermosh, Sriram Subramanian, Vinay Sridhar, Swaminathan Sundararaman, Dulcardo Arteaga, Qianmei Luo, Drew Roselli, Dhananjoy Das, and Nisha Talagala. 2018. A Systems Perspective to Reproducibility in Production Machine Learning Domain. (2018).

[16] gRPC 2018. gRPC. https://www.grpc.io. Accessed: 2019-10-04.

[17] Odd Erik Gundersen, Yolanda Gil, and David W. Aha. 2018. On Reproducible AI: Towards Reproducible Research, Open Science, and Digital Scholarship in AI Publications. *AIMag* 39, 3 (Sept. 2018), 56–68. https://doi.org/10.1609/aimag.v39i3.2816

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, IEEE, 620–629. https://doi.org/10.1109/hpca.2018.00059

[19] Matthew Hutson. 2018. Artificial intelligence faces reproducibility crisis.

[20] Yangqing Jia. 2018. Caffe2. https://www.caffe2.ai.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe. In *Proceedings of the ACM International Conference on Multimedia - MM '14*. ACM, ACM Press, 675–678. https://doi.org/10.1145/2647868.2654889

[22] Marc Jorda, Pedro Valero-Lara, and Antonio J Peña. 2019. Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs. *IEEE Access* (2019).

[23] Liam Li and Ameet Talwalkar. 2019. Random Search and Reproducibility for Neural Architecture Search. arXiv:cs.LG/1902.07638

[24] MLPerf 2019. MLPerf. https://mlperf.org. Accessed: 2019-10-04.

[25] Jon Ander Novella, Payam Emami Khoonsari, Stephanie Herman, Daniel Whitenack, Marco Capuccini, Joachim Burman, Kim Kultima, and Ola Spjuth. 2018. Container-based bioinformatics with Pachyderm. *Bioinformatics* 35, 5 (Aug. 2018), 839–846. https://doi.org/10.1093/bioinformatics/bty699

[26] nvml 2019. nvml. https://developer.nvidia.com/nvidia-management-library-nvml. Accessed: 2019-10-04.

[27] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, high-performance ML serving. *arXiv preprint arXiv:1712.06139* (2017).

[28] OpenTracing 2018. OpenTracing: Cloud Native Computing Foundation. http://opentracing.io. Accessed: 2019-10-04.

[29] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. 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).

[30] perf-tools 2019. perf-tools. https://github.com/brendangregg/perf-tools. Accessed: 2019-10-04.

[31] Hans E. Plesser. 2018. Reproducibility vs. Replicability: A Brief History of a Confused Terminology. *Front. Neuroinform.* 11 (Jan. 2018), 76. https://doi.org/10.3389/fninf.2017.00076

[32] Python Subinterpreter 2019. Initialization, Finalization, and Threads. https://docs.python.org/3.6/c-api/init.html#sub-interpreter-support. Accessed: 2019-10-04.

[33] RecordIO 2019. RecordIO. https://mxnet.incubator.apache.org/versions/master/architecture/note_data_loading.html. Accessed: 2019-10-04.

[34] Frank Seide and Amit Agarwal. 2016. Cntk. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. ACM, ACM Press, 2135–2135. https://doi.org/10.1145/2939672.2945397

[35] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report. Technical report, Google, Inc.

[36] Rachael Tatman, Jake VanderPlas, and Sohier Dane. 2018. A Practical Taxonomy of Reproducibility for Machine Learning Research. (2018).

[37] TensorflowLite 2018. TensorFlow Lite is for mobile and embedded devices. https://www.tensorflow.org/lite/. Accessed: 2019-10-04.

[38] TFRecord 2019. TFRecord. https://www.tensorflow.org/guide/datasets#consuming_tfrecord_data. Accessed: 2019-10-04.

[39] Jason Tsay, Todd Mummert, Norman Bobroff, Alan Braz, Peter Westerink, and Martin Hirzel. 2018. Runway: Machine learning model experiment management tool.

[40] Yuan Yu, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, Xiaoqiang Zheng, MartÃŋn Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, and Tim Harley. 2018. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*, Vol. 16. ACM Press, 265–283. https://doi.org/10.1145/3190508.3190551

[41] Zenodo 2019. Zenodo - Research. Shared. https://www.zenodo.org. Accessed: 2019-10-04.

[42] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Tbd: Benchmarking and analyzing deep neural network training. *arXiv preprint arXiv:1803.06905* (2018).