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 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.

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 reproducible evaluation along with consistent evaluation.

Aside from F1-2, the design should be frameworks and hardware agnostie to support model evaluation using diverse HW/SW stacks; be capable of performing scalable evaluation across systems to cope with the large number of evaluations due to the diverse model/HW/SW combinations; support different benchmarking scenarios.
which mimic the real-world workload exhibited in online, offline, and interactive applications; have a \textbf{benchmarking analysis and reporting} workflow which analyzes benchmarking results across evaluation runs and generates summary reports; enable \textbf{model execution inspection} to identify bottlenecks within a model-, framework-, and system-level components. Other features such as: \textbf{artifact versioning}, \textbf{efficient evaluation workflow}, and \textbf{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 \textit{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 abstraction traceability that allows users to inspect model execution at different HW/SW 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 \([\text{batch}, \text{height}, \text{width}, \text{channel}]\) \((\text{[N, H, W, C]})\), then performs 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 (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 \textit{framework} (e.g. TensorFlow, MXNet, PyTorch). The framework acts as a “runtime” for the model prediction and maps the function operators into \textit{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].
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.

3.1 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 [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.

3.2 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 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.

3.3 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.

3.4 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.

3.5 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.

**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.

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

**Model Execution Inspection** — 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.

**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.

---

**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.

```
1  name: MLPerf_ResNet50_v1.5 # model name
2  version: 1.0.0 # semantic version of the model
3  description: ...
4  framework: # framework information
5  name: Tensorflow
6  version: ’>=1.12.0 <2.0’ # framework version constraint
7  inputs: # model inputs
8    - type: image # first input modality
9      layer_name: ‘input_tensor’
10     element_type: float32
11    steps: # pre-processing steps
12      - decode:
13        data_layout: NHWC
14        color_mode: RGB
15        - resize:
16        - dimensions: [3, 224, 224]
17        method: bilinear
18        keep_aspect_ratio: true
19        - normalize:
20          mean: [123.68, 116.78, 103.94]
21          rescale: 1.0
22    outputs: # model outputs
23      - type: probability # first output modality
24        layer_name: prob
25        element_type: float32
26    steps: # post-processing steps
27      - argsort:
28      - labels_url: https://.../synset.txt
29    preprocess: [(code)]
30    postprocess: [(code)]
31  model: # model sources
32  base_url: https://zenodo.org/record/2535873/files/
33  graph_path: resnet50_v1.pb
34  checksum: 7b94a2da84d...23a46bc08886
35  attributes: # extra model attributes
36  training_dataset: # dataset used for training
37      - name: ImageNet
38      - version: 1.0.0
```

---

**4 MLModelScope Design and Implementation**

We propose MLModelScope, a DL benchmarking platform design that achieves the objectives [1-10] set out in Section 3. To achieve [7] scalable evaluation, we design MLModelScope as a distributed platform. To enable [7] real-world benchmarking scenarios, MLModelScope deploys models to be either evaluated using a cloud (as in model serving platforms) or edge (as in local model inference) scenario. To 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.

---

Cheng Li, Abdul Dakkak, et al.
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: (1) initialization, (2) evaluation, and (3) 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 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) publishes the evaluation results to the evaluation database and tracing server. A summary of the results is (6) sent to the server which (7) 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 (8) querying the evaluation database and performing analysis on the results, and (9) 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 (1) reproducible and (2) consistent evaluation. To achieve this, MLModelScope defines a benchmarking specification covering the 4 aspects of evaluation. A model in MLModelScope is specified using a **model manifest**, and a software stack is specified using a **framework manifest**. The manifests are textual specification 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 (3). As the model, software stack, system, and benchmarking scenario specification are decoupled, one can easily evaluate the different combinations, enabling (4). 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 22-21), model outputs and post-processing steps (Lines 28-30), custom pre- and post-processing functions (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-interpreter [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 in the manifest code (e.g. Lines 7–21 in Listing 1 for pre-processing). 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.

#### 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 evaluation or analysis through the MLModelScope client. To enable the client, the user can 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, and gRPC (Listing 4) to interact with the MLModelScope agents. To enforce, 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,
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, 

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 (\(\mathcal{F}\)).

In the \(\mathcal{A}\) analysis workflow, the server again \(\mathcal{A}\) takes the user input, but, rather than performing evaluation, it \(\mathcal{A}\) queries the evaluation database (Section 4.5.2), and then aggregates and analyzes the evaluation results. MLModelScope enables \(\mathcal{F}\) through an across-stack analysis pipeline. It \(\mathcal{F}\) consumes the benchmarking results and the profiling traces in the evaluation database and performs the analysis. Then the server \(\mathcal{C}\) 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 (\(\mathcal{B}\)), 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 \(\mathcal{B}\) 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 \(\mathcal{C}\) tracing server to get aggregated into a single profiling trace. \(\mathcal{C}\) 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 \(\mathcal{F}\) 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 \(\mathcal{F}\) 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).

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 – 11x 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

4.4.4 Tracing Hooks

To enable [9], 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 not 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 altogether 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...
profillers 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 workflow 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 F8 design objectives. In this section, we focus on evaluating how MLModelScope handles F7 different benchmarking scenarios, F9 result summarization, and F10 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 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 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

1Trimmed mean is computed by removing 20% of the smallest and largest elements and computing the mean of the residual; i.e. \( \text{TrimmedMean}(\text{list}) = \text{Mean}(\text{Sort}(\text{list})[[0.2*\text{len(list)}::0.8*\text{len(list)}]]) \).
The Design and Implementation of a Scalable DL Benchmarking Platform

<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 @ 1.5GHz</td>
<td>Tesla P100-SXM2</td>
<td>Pascal</td>
<td>10.0</td>
<td>732</td>
<td></td>
</tr>
</tbody>
</table>

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

<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>64</td>
</tr>
<tr>
<td>4</td>
<td>ResNet_v2_152</td>
<td>77.80</td>
<td>231</td>
<td>14.44</td>
<td>14.65</td>
<td>468.3</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>76.46</td>
<td>103</td>
<td>6.33</td>
<td>6.53</td>
<td>930.7</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td>ResNet_v1_101</td>
<td>76.40</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>75.93</td>
<td>230</td>
<td>14.58</td>
<td>14.72</td>
<td>468.0</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>ResNet_v2_50</td>
<td>75.60</td>
<td>98</td>
<td>6.17</td>
<td>6.35</td>
<td>1,119.7</td>
<td>256</td>
</tr>
<tr>
<td>11</td>
<td>ResNet_v1_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>12</td>
<td>AI_Matrix_ResNet50</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>13</td>
<td>Inception_v2</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>14</td>
<td>AI_Matrix_DenseNet121</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>15</td>
<td>MLPerf_MobileNet_v1</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>16</td>
<td>VGG16</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>17</td>
<td>VGG19</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>18</td>
<td>MobileNet_v1_1.0_224</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>19</td>
<td>AI_Matrix_GoogleNet</td>
<td>70.01</td>
<td>27</td>
<td>5.43</td>
<td>5.55</td>
<td>2,464.5</td>
<td>128</td>
</tr>
<tr>
<td>20</td>
<td>MobileNet_v1_1.0_192</td>
<td>70.00</td>
<td>16</td>
<td>2.55</td>
<td>2.67</td>
<td>3,460.8</td>
<td>128</td>
</tr>
<tr>
<td>21</td>
<td>Inception_v1</td>
<td>69.80</td>
<td>26</td>
<td>5.27</td>
<td>5.41</td>
<td>2,576.6</td>
<td>128</td>
</tr>
<tr>
<td>22</td>
<td>BVLC_GoogLeNet</td>
<td>68.70</td>
<td>27</td>
<td>6.05</td>
<td>6.17</td>
<td>951.7</td>
<td>8</td>
</tr>
<tr>
<td>23</td>
<td>MobileNet_v1_1.0_224</td>
<td>68.40</td>
<td>10</td>
<td>2.48</td>
<td>2.61</td>
<td>3,183.7</td>
<td>64</td>
</tr>
<tr>
<td>24</td>
<td>MobileNet_v1_1.0_160</td>
<td>68.00</td>
<td>16</td>
<td>2.57</td>
<td>2.74</td>
<td>4,240.5</td>
<td>64</td>
</tr>
<tr>
<td>25</td>
<td>MobileNet_v1_0.75_192</td>
<td>67.20</td>
<td>10</td>
<td>2.42</td>
<td>2.6</td>
<td>4,187.8</td>
<td>64</td>
</tr>
<tr>
<td>26</td>
<td>MobileNet_v1_0.75_160</td>
<td>65.30</td>
<td>10</td>
<td>2.48</td>
<td>2.65</td>
<td>5,569.6</td>
<td>64</td>
</tr>
<tr>
<td>27</td>
<td>MobileNet_v1_1.0_128</td>
<td>65.20</td>
<td>16</td>
<td>2.29</td>
<td>2.46</td>
<td>6,743.2</td>
<td>64</td>
</tr>
<tr>
<td>28</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>29</td>
<td>MobileNet_v1_0.75_128</td>
<td>62.10</td>
<td>10</td>
<td>2.3</td>
<td>2.47</td>
<td>8,378.4</td>
<td>64</td>
</tr>
<tr>
<td>30</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>31</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>32</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>33</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>34</td>
<td>MobileNet_v1_0.25_224</td>
<td>49.80</td>
<td>1.9</td>
<td>2.46</td>
<td>3.40</td>
<td>5,257.9</td>
<td>64</td>
</tr>
<tr>
<td>35</td>
<td>MobileNet_v1_0.25_192</td>
<td>47.70</td>
<td>1.9</td>
<td>2.44</td>
<td>2.6</td>
<td>7,135.7</td>
<td>64</td>
</tr>
<tr>
<td>36</td>
<td>MobileNet_v1_0.25_160</td>
<td>45.50</td>
<td>1.9</td>
<td>2.39</td>
<td>2.53</td>
<td>10,081.5</td>
<td>256</td>
</tr>
<tr>
<td>37</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>

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, 90th 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
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 39.44ms whereas it takes 32.4ms 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
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. K5 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 all of them. For example, while there is active work on proposing benchmark suites, reference workloads, and analysis [24, 42], they provide a set of benchmarking scenarios and a simple mechanism for 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 scalable evaluation and possibly artifact versioning but nothing else. Finally, systems such as as [15, 25, 39] track the model and data from their use in training till deployment and can achieve reproducible and 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 and has limited support of (limited to computing the nth 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 data instances 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


