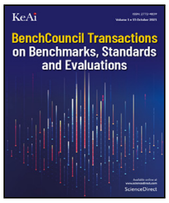
Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/



MLHarness: A scalable benchmarking system for MLCommons

Yen-Hsiang Chang [a,*], Jianhao Pu [a], Wen-mei Hwu [a], Jinjun Xiong [a,b,*]

[a] University of Illinois at Urbana-Champaign, Urbana, IL, USA
[b] University at Buffalo, Buffalo, NY, USA

ARTICLE INFO

Keywords:
Machine learning
Inference
Benchmarking

ABSTRACT

With the society's growing adoption of machine learning (ML) and deep learning (DL) for various intelligent solutions, it becomes increasingly imperative to standardize a common set of measures for ML/DL models with large scale open datasets under common development practices and resources so that people can benchmark and compare models' quality and performance on a common ground. MLCommons has emerged recently as a driving force from both industry and academia to orchestrate such an effort. Despite its wide adoption as standardized benchmarks, MLCommons Inference has only included a limited number of ML/DL models (in fact seven models in total). This significantly limits the generality of MLCommons Inference's benchmarking results because there are many more novel ML/DL models from the research community, solving a wide range of problems with different inputs and outputs modalities. To address such a limitation, we propose MLHarness, a scalable benchmarking harness system for MLCommons Inference with three distinctive features: (1) it codifies the standard benchmark process as defined by MLCommons Inference including the models, datasets, DL frameworks, and software and hardware systems; (2) it provides an easy and declarative approach for model developers to contribute their models and datasets to MLCommons Inference; and (3) it includes the support of a wide range of models with varying inputs/outputs modalities so that we can scalably benchmark these models across different datasets, frameworks, and hardware systems. This harness system is developed on top of the MLModelScope system, and will be open sourced to the community. Our experimental results demonstrate the superior flexibility and scalability of this harness system for MLCommons Inference benchmarking.

1. Introduction

With the rise of machine learning (ML) and deep learning (DL) innovations in both industry and academia, there is a clear need for standardized benchmarks and evaluation criteria to facilitate comparison and development of ML/DL innovations. MLCommons Inference [1], a standard ML/DL inference benchmark suite with properly defined metrics and benchmarking methodologies, has emerged recently to facilitate such an effort. However, MLCommons Inference only included five models when it was first introduced in 2019, and has only included seven models recently [2]. This limited and slow-growing number of ML/DL models stifles the adoption of MLCommons Inference as a general benchmarking platform because there are many more novel ML/DL models from the research community, solving a wide range of problems with different inputs and outputs modalities. To address such a limitation, we propose MLHarness, a scalable benchmarking harness system for MLCommons Inference, to make MLCommons Inference embrace new models and modalities easily. MLHarness is developed on top of MLModelScope [3] with three distinctive new features: (1) it codifies the required benchmarking environment for MLCommons Inference explicitly, including models, datasets, frameworks, software and hardware stacks; (2) it provides an easy and declarative approach for model developers to contribute their models and datasets to MLCommons Inference; and (3) it supports a wide range of models with varying inputs and outputs modalities. Our experiments show that MLHarness is capable of reporting all the required metrics as defined by MLCommons Inference for models with different input/output modalities and for models both within and beyond MLCommons Inference.

In particular, we make the following contributions: (1) we propose MLHarness, a scalable benchmarking harness system for MLCommons Inference while supporting models beyond those in MLCommons Inference; (2) we extend MLModelScope to provide user-defined pre-processing and post-processing interfaces so that MLModelScope can easily support new models and new modalities; (3) we showcase MLHarness' capabilities as a scalable benchmarking harness system for MLCommons Inference by running experiments on a range of models both within and beyond MLCommons Inference under different frameworks and systems configurations; and (4) we further demonstrate the unique value of scalable benchmarking in identifying abnormal

* Corresponding authors.
E-mail addresses: yhchang3@illinois.edu (Y. Chang), jpu3@illinois.edu (J. Pu), w-hwu@illinois.edu (W. Hwu), jinjunx@illinois.edu, jinjun@buffalo.edu (J. Xiong).

https://doi.org/10.1016/j.tbench.2021.100002
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 4 November 2021

system behaviors and how MLHarness helps to explain those seemingly abnormal behaviors resulting from complex software and hardware interactions.

2. Background

2.1. ML/DL benchmark challenges

ML and DL innovations such as applications, datasets, frameworks, models, and software and hardware systems, are being developed in a rapid pace. However, current practice of sharing ML/DL innovations is to build ad-hoc scripts and write manuals to describe the workflow. This makes it hard to reproduce the reported metrics and to port the innovations to different environments and solutions. Therefore, having a standard benchmarking platform with an exchange specification and well-defined metrics to fairly compare and benchmark the innovations is a crucial step toward the success of ML/DL community.

Previous work includes (1) ML/DL model zoos curated by framework developers [4–9], but they only aim for sharing ML/DL models as a library; (2) package managers for a specific software environment such as Spack [10], while they are just targeting on maintaining packages in different software and hardware stacks; (3) benchmarking platforms such as MLCommons [1,11] and MLModelScope [3], but the former only focuses on few specific models and the latter only focuses on models in computer vision tasks; (4) collections of reproducible MLOps components and architectures [12–14], while their main focuses are on deployment and automation; (5) plug-and-play shareable containers such as MLCube [15], but its generality makes it hard to identify and locate the crucial components for the cause of abnormal behaviors in ML/DL models; (6) simulator of ML/DL inference servers such as iBench [16], but the main focus on capturing data transfer capabilities between clients and servers provides no insights on profiling models. As the above applications either only focus on a specific software and hardware stack, or use ad-hoc approaches to handle specific ML/DL tasks, or are lack of a consistent benchmarking method, it is hard to use them individually to have a well rounded experience when developing ML/DL innovations.

To address these ML/DL benchmark challenges, we propose a new scalable benchmarking system: MLHarness by taking advantage of two open-source projects: MLModelScope [3] for its exchange specification on software and hardware stacks, and MLCommons Inference [1] for its community-adopted benchmarking scenarios and metrics. With MLHarness, we are able to benchmark and compare quality and performance of models on a common ground through a set of well-defined metrics and exchange specification.

2.2. Overview of MLCommons

MLCommons [1,11], previously known as MLPerf, is a platform aims to answer the needs of the nascent machine learning industry. MLCommons Training [11] measures how fast systems can train models to a target quality metric, while MLCommons Inference [1] measures how fast systems can process inputs and produce results using a trained model. Both of these two benchmark suites target on providing benchmarking results on different scales of computing services, ranging from tiny mobile devices to high performance computing data centers. As the main focus of this paper is on benchmarking ML/DL inferences, we only focus on MLCommons Inference in the rest of this paper.

2.2.1. Characteristics of MLCommons Inference

MLCommons Inference is a standard ML/DL inference benchmark suite with a set of properly defined metrics and benchmarking methodologies to fairly measure the inference performance of ML/DL hardware, software, and services. MLCommons Inference focuses on the following perspectives when designing its benchmarking metrics:

- **Selection of Representative Models.** MLCommons Inference selects representative models that are mature, open source, and have earned community support. This permits accessibility and reproducible measurements, which facilitates MLCommons Inference becoming a standardized benchmarking suite.
- **Scenarios.** MLCommons Inference consists of four evaluation scenarios, including single-stream, multistream, server, and offline. These four scenarios aim for simulating realistic behaviors of inference systems in many critical applications.
- **Metrics.** Apart from the commonly used model metrics such as accuracy, MLCommons Inference also includes a set of systems related metrics, such as percentile-latency and throughput. These make MLCommons Inference appealing in satisfying the demand of different use cases, such as 99% percentile-latency for a data center to respond to a user query.

2.2.2. Workflows of MLCommons Inference

Fig. 1 shows the critical components as defined in MLCommons Inference, where the numbers and arrows denote the sequence and the directions of the data flows, respectively. The description of the components follows:

- **Load Generator (LoadGen).** The LoadGen produces query traffics as defined by the four scenarios above, collects logging information, and summarizes benchmarking results. It is a stand-alone module that stays the same across different models.
- **System Under Test (SUT).** The SUT consists of the inference system under benchmarking, including ML/DL frameworks, ML/DL models, software libraries and the target hardware system. Once the SUT receives a query from the LoadGen, it completes an inference run and reports the result to the LoadGen.
- **Data Set.** Before issuing queries to the SUT, the LoadGen needs to let the SUT fetch the data needed for the queries from the dataset and pre-process the data. This is not included in the latency measurement.
- **Accuracy Script.** After all queries are issued and the results are received, the accuracy script will be invoked to validate the accuracy of the model from the logging information.

2.2.3. Limitations of MLCommons Inference

As we can observe from the characteristics and the workflows of MLCommons Inference above, MLCommons Inference involves benchmarking under different scenarios with various metrics, which provides a community acknowledged ML/DL benchmark standard. However, the focus on the seven representative models shadows its advantage because MLCommons Inference only provides ad-hoc scripts for these representative models, and it is hard to extend them to many other models beyond MLCommons Inference.

In fact, the only critical component in MLCommons Inference is the LoadGen, while the other components can be replaced with any inference systems. In this paper, we present how to replace the components other than the LoadGen by MLModelScope [3], an inference platform with a clearly defined exchange specification and an across-stack profiling and analysis tool, and extend MLModelScope so that it becomes a scalable benchmarking harness for MLCommons Inference. This greatly extends the applicability of MLCommons Inference for models well beyond it.

2.3. Overview of MLModelScope

MLModelScope [3] is a hardware and software agnostic distributed platform for benchmarking and profiling ML/DL models across datasets, frameworks and systems.

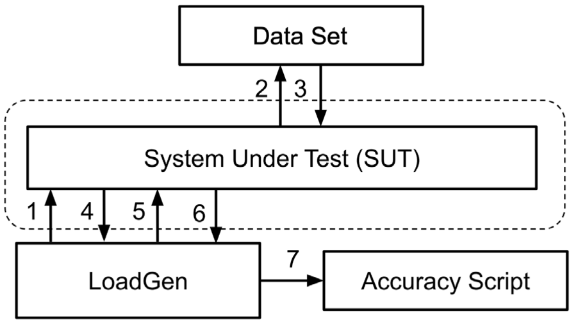


Fig. 1. Workflow of MLCommons Inference [1].

2.3.1. *Characteristics of MLModelScope*

MLModelScope consists of a specification and a runtime that enable repeatable and fair evaluation. The design aspects follow:

- **Specification.** MLModelScope utilizes the software and model manifests as proposed in DLSpec [17], which capture different aspects of an ML/DL task and ensure usability and reproducibility. The software manifest defines the software requirements, such as ML/DL frameworks to run an ML/DL task. The model manifest defines the logic to run the model for the ML/DL task, such as pre-processing and post-processing methods, and the required artifact sources. An example is shown in Listing 1.
- **Runtime.** The runtime of MLModelScope follows the manifests to set up the required environment for inference. Moreover, MLModelScope includes the across-stack profiling and analysis tool, XSP [18], which introduces a leveled and iterative measurement approach to overcome the impact of profiling overhead. As shown in Fig. 2, MLModelScope captures profiling data for different levels, which enables users to correlate the information and analyze the performance data in different levels.

```
name: Inception-v3 # model name
version: 1.0.0 # semantic version of the model
task: classification
framework: # framework information
    name: TensorFlow
    version: ^1.x # framework version constraint
model: # model sources
    graph_path: https://.../inception_v3.pb
    graph_checksum: 328f68...3a813e
steps: # pre-processing steps
    decode:
        element_type: int8
        data_layout: NHWC
        color_layout: RGB
    crop:
        method: center
        percentage: 87.5
    resize:
        dimensions: [3, 299, 299]
        method: bilinear
        keep_aspect_ratio: true
    mean: [127.5, 127.5, 127.5]
    rescale: 127.5
...
```

Listing 1: An excerpt of manifest from MLModelScope [3]

2.3.2. *Limitations of MLModelScope*

Although MLModelScope involves a clearly defined specification and is able to run several hundreds of models in different ML/DL

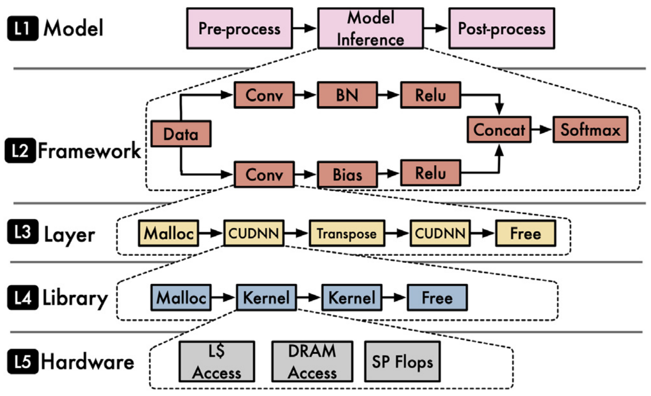


Fig. 2. Profiling levels in MLModelScope [3].

frameworks, it currently only supports models for computer vision tasks. While MLModelScope discussed the possibility of using user-defined pre-processing and post-processing inline Python scripts to serve as a universal handler for all kinds of models, MLModelScope did not implement those interfaces but only introduced built-in image manipulations to support computer vision tasks. In this paper, we have actually implemented the user-defined pre-processing and post-processing interfaces and demonstrated its usage on models with different modalities and different pre-processing and post-processing, such as question answering and medical 3D image segmentation.

3. MLHarness implementation

This section describes the crucial implementations of MLHarness, a scalable benchmarking harness system for MLCommons Inference [1] for tackling the limitations of MLCommons Inference and MLModelScope [3]. The implementations include the support of user defined pre-processing and post-processing interfaces and the encapsulation of MLModelScope for MLCommons Inference.

3.1. *Pre-processing and post-processing interfaces*

As described in DLSpec [17] and MLModelScope [3], to make the user defined pre-processing and post-processing interfaces universal, inline Python scripts are chosen to allow great flexibility and productivity, as Python functions can download and run Bash scripts and some C++ code. On the other hand, MLModelScope is implemented in Go; therefore, it is necessary to build a bridge between the runtime of MLModelScope and the embedded Python scripts in the model manifest so that, within the MLModelScope runtime, we can invoke the Python runtime to execute user defined pre-processing and post-processing functions.

A naive solution is to save the input data and the functions as files and execute pre-processing and post-processing functions apart from MLModelScope. However, this approach is impractical since it introduces high serialization and process initialization overhead, and it also makes MLModelScope incapable of supporting streaming data [17].

In order to avoid using intermediate files, we instead use Python/C APIs [19] to embed a Python interpreter into MLModelScope to execute Python functions, as suggested by DLSpec. To use these APIs, we need to implement wrappers in Go to call them. Instead of building these wrappers from scratch, we use the open source Go-Python3 bindings [20]. In this fashion, the Python functions can be executed within MLHarness directly to avoid the problems mentioned in the naive solution.

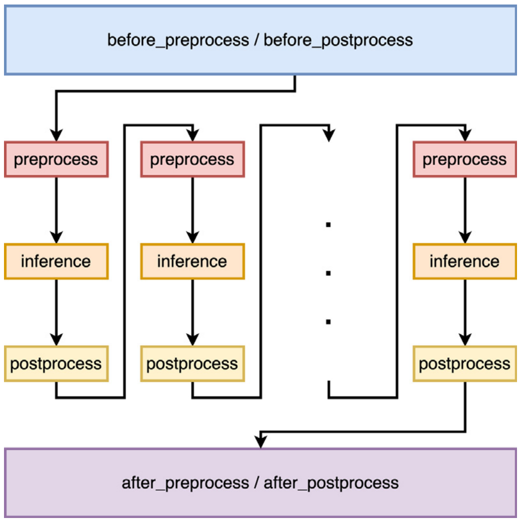


Fig. 3. Workflow of MLHarness.

3.1.1. *Implementation details*

Fig. 3 shows the invocation sequence of user defined pre-processing and post-processing functions by MLHarness. The `before_preprocess` and the `before_postprocess` functions are invoked only once at the startup stage; the `after_preprocess` function and the `after_postprocess` functions are invoked only once after all inferences are done. These four functions are for the sake of loading datasets, writing logging information to files, and specifying configurations during runtime if necessary. The `preprocess` function and the `postprocess` function are invoked right before and after every model inference, respectively, to pre-process and post-process the inputs and outputs of the model.

```go
func Processing(tensor interface{}, functionName string) interface{} {
    pyData := MoveDataToPythonInterpreter(tensor)
    pyFunc := FindTheProcessingFunctionByItsName(functionName)
    pyResult := ExecuteProcessingFunction(pyFunc, pyData)
    result := GetResultFromPythonInterpreter(pyResult)
    return result
}
```

Listing 2: Pre-processing and post-processing implementation in Go

To embed a Python interpreter, we need to initialize it through Python/C APIs at the beginning of MLHarness. Then, as Listing 2 shows, the function handling the embedded Python pre-processing and post-processing scripts consists of four parts, utilizing the Go-Python3 bindings:

- `MoveDataToPythonInterpreter`. Moving data from Go to Python is not easy since the data being processed are large, for example, a tensor representing an image. One solution is to serialize the data at one end, transfer the data as a string, and deserialize at the other end. However, it introduces a high overhead due to the high cost of encoding and decoding. To overcome this problem and to make data transfer efficient, we propose to copy the data in-memory, i.e., we only send the shape of the tensor and the address of its underlying flattened array, and reconstruct the tensor by copying data from the address and by its shape. Note that to guarantee the validity of data transfer, we need to make sure that the underlying flattened array represents the tensor contiguously, particularly in case lazy operations were done on the tensor, such as transposition.

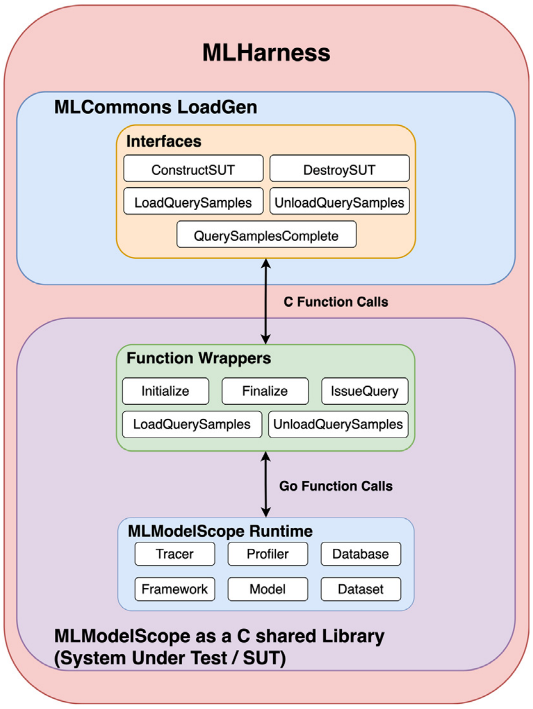


Fig. 4. Structure of MLHarness.

- `FindTheProcessingFunctionByItsName`. The processing functions in the model manifest are registered in the `__main__` module of the Python interpreter during its initialization. To get the corresponding `PyObject` of these functions, we query the `__main__` module by the names of functions, which are the six processing functions as listed in Fig. 3.
- `ExecuteProcessingFunction`. The signatures of the processing functions in the model manifest are in the form of `process(ctx, data)`, where `ctx` is a dictionary capturing additional information in the manifest, and `data` is the tensor we got from `MoveDataToPythonInterpreter`. Therefore, in order to invoke the processing function, we need to call the Go-Python3 binding with the `PyObject` of the processing function, the dictionary of `ctx` and the `data` going to be processed. Note that the `data` is only effective for `preprocess` and `postprocess`, and it is just a `PyObject` of `None` for the other four processing functions.
- `GetResultFromPythonInterpreter`. This is similar to the first part except that it moves data from Python to Go instead of the other way around. Note that we still copy the data in-memory to avoid unnecessary overhead.

3.2. *Structure of MLHarness*

Fig. 4 shows the encapsulation of MLModelScope for MLCommons Inference. In order to utilize MLCommons Inference defined scenarios and performance metrics, we keep the LoadGen and the accuracy script the same as they were in MLCommons Inference. On the other hand, We replace the built-in SUT and data set in MLCommons Inference with MLModelScope runtime to run the models. In this way, MLModelScope is capable of acting as an easy-to-use black box to respond to the queries issued by the LoadGen in MLCommons Inference, and it also provides across-stack profiling information for further analysis, which are not available when merely using MLCommons Inference.

3.2.1. Implementation details

MLModelScope is developed in Go, but the LoadGen in MLCommons Inference is developed in C++ and used in Python through Python bindings. In order to make the communication between MLModelScope and MLCommons Inference feasible, we build MLModelScope as a C shared library [21], use the ctypes module [22] in Python to load the shared library, and call the functions in the shared library. Three notable implementations are described below:

```
name: MLPerf_BERT # model name
version: 1.0.0 # semantic version of the model
framework: # framework information
    name: PyTorch
    version: '>=1.5.0' # framework version constraint
inputs: # model inputs
    - type: text # input modality
      element_type: string
outputs: # model outputs
    - type: text # output modality
      element_type: string
model: # model sources
    graph_path: https://.../bert.pt
    graph_checksum: c3bb5a...aa1ccd
preprocess: |
  from transformers import BertTokenizer
  import numpy as np
  ...
  class SquadExample(object):
    ...
  class InputFeatures(object):
    ...
  def read_squad_examples(...):
    ...
  def convert_examples_to_features(...):
    ...
  features = []
  tokenizer = BertTokenizer(...)
  examples = read_squad_examples(...)
  convert_examples_to_features(features, examples, tokenizer, ...)
  def preprocess(ctx, data):
    cur = features[int(data)]
    return cur.input_ids, cur.input_mask, cur.segment_ids
postprocess: |
  import numpy as np
  import json
  def postprocess(ctx, data):
    res = np.stack([data[0], data[1]], axis = -1).squeeze(0).tolist()
    return [json.dumps(res)]
...
```

Listing 3: An excerpt of model manifest for BERT

- **Function wrappers.** To simplify the process of building the C shared library and leaving MLModelScope as a black box, we create function wrappers for critical applications in MLModelScope and only export them in the shared library. This includes the `Initialize` and `Finalize` wrappers to initialize and finalize the profiling tools in MLModelScope. It also includes the `LoadQuerySamples`, `IssueQuery`, and `UnloadQuerySamples` wrappers to pre-load and pre-process the data from the data set, handle queries from the LoadGen, and free the memory occupied by the pre-loaded data, respectively.
- **Data transmissions.** It is hard to directly exchange data between Go and Python, since there is no one-to-one correspondence between data types in these two languages. To solve this problem, we utilize the built-in primitive C compatible data types in ctypes [22] for Python and CGO [23] for Go, since they define how to transform data if there is no clear correspondence between data types in C and the corresponding languages. Using this method, the data conversion can be done in-memory instead of through serialization.
- **Blocking statements.** When we exchange data between Go and Python, the garbage collector at one end does not automatically know that it needs to keep the data before the data are really copied or used at the other end, which might result into undefined behaviors. To solve this problem, we need to manually create blocking statements to block garbage collection until a deep copy of the data is made at the other end. This can be done using the `KeepAlive` function [24] in Go and managing reference counts [25] in Python to prevent garbage collection being invoked until the `KeepAlive` is executed and the reference count is decreased to zero, respectively.

3.3. Example of MLHarness

With the help of user defined pre-processing and post-processing interfaces, MLHarness is able to handle various models' inputs and outputs modalities that are not supported in MLModelScope. Also, it is easy to use the model manifest to add models for MLModelScope to report MLCommons Inference defined metrics, which is hard when merely using MLCommons Inference. Listing 3 is the model manifest using the pre-processing and post-processing interfaces for BERT [26], a language representation model, to handle the question answering modality that was not supported in MLModelScope. The pre-processing step uses the tokenizer from the transformers Python third-party library [27] to parse data and prepare input features. The post-processing step reshapes the outputs into the format as defined by the accuracy script. The tedious implementation of the tokenizer is one of the reason why MLModelScope cannot support the question answering modality, since it is hard to create an equivalent built-in alternative inside MLModelScope using Go. On the contrary, through the user defined pre-processing and post-processing interfaces, MLHarness can utilize the community developed Python third-party libraries to overcome this obstacle.

4. Experimental results

We conduct two sets of experiments to demonstrate the success of MLHarness on overcoming the limitations in MLModelScope [3] and MLCommons Inference [1]. In the first set of experiments, we use MLHarness to benchmark models in MLCommons Inference and report MLCommons Inference defined metrics to show that it supports modalities that are not supported in MLModelScope, such as question answering and medical image 3D segmentation. In addition, we show that MLHarness is able to report the results for all four scenarios defined by MLCommons Inference. In the second set of experiments, we use MLHarness to benchmark models beyond MLCommons Inference and report MLCommons Inference defined metrics to show the usage of our newly extended MLModelScope as an easy-to-use black box for MLCommons Inference.

4.1. Experiment setup

Table 1 shows the systems used for experiments. The system naming convention follows the rule as the identifier of the CPU types followed by the acronym of the ML/DL framework, and then the identifier of the GPU type if a GPU is used. There are three system instance categories in total. The first category is an Intel desktop-grade CPU system, including system 9800-ORT-RTX, 9800-PT-RTX, 9800-ORT, 9800-PT, and 9800-MX-RTX. The second category is also an Intel but different desktop-grade CPU system, including system 7820-ORT-TITAN, 7820-PT-TITAN, and 7820-TF. The last category is a server-based system using AMD CPUs, including system AMD-ORT-A100 and AMD-ORT-V100. We choose different combinations of frameworks, software systems and hardware systems in order to demonstrate the flexibility and scalability of MLHarness as a harness for benchmarking.

For both sets of experiments, we report the accuracy, the throughput in the offline scenario, and the throughput and 90 percentile latency in

Table 1
Systems used for experiments.

System annotations	Framework	Processor	Accelerator
9800-ORT-RTX	ONNX Runtime	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	1x GeForce RTX 3090
9800-PT-RTX	PyTorch	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	1x GeForce RTX 3090
9800-ORT	ONNX Runtime	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	None
9800-PT	PyTorch	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	None
7820-ORT-TITAN	ONNX Runtime	1x Intel(R) Core(TM) i7-7820X CPU @ 3.60 GHz	1x TITAN V
7820-PT-TITAN	PyTorch	1x Intel(R) Core(TM) i7-7820X CPU @ 3.60 GHz	1x TITAN V
7820-TF	TensorFlow	1x Intel(R) Core(TM) i7-7820X CPU @ 3.60 GHz	None
AMD-ORT-A100	ONNX Runtime	1x AMD EPYC 7702 64-Core Processor	1x A100
AMD-ORT-V100	ONNX Runtime	1x AMD EPYC 7702 64-Core Processor	1x V100
9800-MX-RTX	MXNet	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	1x GeForce RTX 3090

Table 2
MLHarness and MLCommons reported results for all four scenarios on 9800-ORT-RTX.

Benchmark suite	Offline (sample/s)	Single-Stream		Server		Multi-Stream	
		(sample/s)	90th percentile latency (ms)	(sample/s)	99th percentile latency (ms)	(sample/query)	99th percentile latency (ms)
MLHarness	133	118	9.2	69	44	5	42
MLCommons Inference	315	308	3.2	121	14	12	44

the single-stream scenario. The accuracy is defined differently for each modalities, including the top-1 accuracy for image classification, mAP scores for object detection, F1 scores for question answering, and mean DICE scores for medical image 3D segmentation. As defined in MLCommons Inference [1], the offline scenario represents applications where all data are immediately available and latency is unconstrained, such as photo categorization; on the contrary, the single-stream scenario represents a query stream with sample size of 1, reflecting applications requiring swift responses, such as real time augmented reality. In order to facilitate the comparison between these two scenarios, we fix the batch size of the inferences to 1 in all experiments. This helps to demonstrate how scenarios can affect the throughput of models.

MLHarness is also capable of reporting results of the other two scenarios as defined by MLCommons Inference [1], which are the server and the multistream scenarios. The server scenario represents applications where query arrival is random and latency is important, such as online translation. The multistream scenario represents application with a stream of queries, where each query consists of multiple inferences, such as multi-camera driver assistance. We demonstrate that MLHarness is able to report the results of these two scenarios by running `ResNet50` on 9800-ORT-RTX.

Note that, because of the limited access to data-center scale systems, we are not able to develop and conduct experiments for the rest of the two models as provided by MLCommons Inference, which are `DLRM` for recommendation system and `RNNT` for speech recognition. But we believe our methodologies as discussed can be easily extended for the two models.

4.2. Results of models in MLCommons Inference

In this set of experiments, we demonstrate the capability of MLHarness on reporting MLCommons Inference [1] defined metrics, by benchmarking representative MLCommons Inference models with a variety of systems.

Table 2 shows the various MLCommons Inference defined experimental results of `ResNet50` produced by MLHarness on system 9800-ORT-RTX. From the results, we observe that using MLHarness, running `ResNet50` on such a target system is able to classify 133 images per second, respond to a query of one image in less than 9.2 ms in 90% of the time if the queries are received contiguously, respond to a query of one image in less than 44 ms in 99% of the time if the queries are received following the Poisson distribution with an average of 69 queries per second, and respond to a query of five images in less than 42 ms in 99% of the time if the queries are received contiguously. Note that the number of queries per second in the server scenario and the number of samples per query in the multistream scenarios are tunable parameters for the system to meet the latency requirements.

We also run the same set of experiments on 9800-ORT-RTX using the original MLCommons Inference flows, as shown in Table 2. The results show that MLCommons Inference performs two to three times better than MLHarness. In order to investigate the discrepancy that MLHarness has a worse performance than MLCommons Inference, we take the Offline scenario as an example and break down the execution time into two parts, including (1) model-inference time for the interval between the model receives pre-processed input tensors and returns output tensors and (2) post-processing time involving generating MLCommons Inference defined format. As Fig. 5 shows, while both MLHarness and MLCommons Inference spend nearly the same amount of time on model inference, the much higher latency for MLHarness to post-process data make it hard to achieve the same performance as reported by MLCommons Inference. The underlying reason of this high latency in MLHarness is due to the aggregated data transferring time between different languages, as data need to be moved several times among ML/DL frameworks, post-processing interfaces and wrappers, while it is not the case for MLCommons Inference since once the inference is done, data always reside in Python. One way to mitigate this high latency is to further optimize MLHarness for MLCommons Inference by responding directly to the LoadGen in the post-processing function instead of transferring data back to MLHarness and then reporting to MLCommons Inference suite through wrappers between different languages.

Table 3 shows the experimental results of `ResNet50` and `MobileNet` for image classification, `SSD MobileNet 300x300` and `SSD ResNet34 1200x1200` for object detection, `BERT` for question answering, and `3D-UNet` for medical image 3D segmentation. All of these models are provided by MLCommons Inference and can be found at its GitHub page [2].

An interesting observation from Table 3 is that the throughput of `ResNet50` on system AMD-ORT-V100, which is 202 samples per second, is higher than that on system AMD-ORT-A100, which is 159 samples per second. This seems to be counter-intuitive as the A100 GPU is two generations newer than the V100 GPU, hence the A100 GPU is supposed to have better performance than V100. With MLCommons' inference methodology alone, we are not able to figure out the reason of this "seemingly abnormal" behavior. This is the place for MLHarness to shine with its extended MLModelScope capabilities. Leveraging the across-stack profiling and analysis capabilities from MLModelScope, we are able to align framework-level spans from the ONNX Runtime profiler and the library-level spans from CUDA Profiling Tools Interface, and capture the detailed view of this strange

Table 3
MLHarness reported results for models in MLCommons Inference.

Model	System	Accuracy	Offline (sample/s)	Single-Stream	
				(sample/s)	90th percentile latency (ms)
MLPerf ResNet50	9800-ORT-RTX	Top1: 76.452%	133	118	9.2
	9800-ORT	Top1: 76.456%	63	62	16
	7820-ORT-TITAN	Top1: 76.456%	118	116	9.2
	7820-TF	Top1: 76.456%	20	20	57
	AMD-ORT-A100	Top1: 76.456%	159	146	6.7
	AMD-ORT-V100	Top1: 76.456%	202	154	6.4
MLPerf MobileNet	9800-ORT-RTX	Top1: 71.676%	196	160	6.5
	9800-ORT	Top1: 71.676%	61	58	23
	7820-ORT-TITAN	Top1: 71.676%	188	159	6.6
	7820-TF	Top1: 71.676%	24	24	44
	AMD-ORT-A100	Top1: 71.666%	358	270	3.7
	AMD-ORT-V100	Top1: 71.676%	382	319	3.2
MLPerf SSD MobileNet 300 × 300	9800-ORT-RTX	mAP: 23.172%	35	32	37
	9800-ORT	mAP: 23.173%	28	28	37
	7820-ORT-TITAN	mAP: 23.173%	30	28	41
	7820-TF	mAP: 23.173%	13	13	78
	AMD-ORT-A100	mAP: 23.170%	20	20	52
	AMD-ORT-V100	mAP: 23.173%	18	18	57
MLPerf SSD ResNet34 1200 × 1200	9800-ORT-RTX	mAP: 19.961%	20	19	54
	9800-ORT	mAP: 19.955%	1.4	1.7	816
	7820-ORT-TITAN	mAP: 19.955%	16	15	66
	7820-TF	mAP: 20.215%	1.4	1.4	704
	AMD-ORT-A100	mAP: 19.957%	23	21	54
	AMD-ORT-V100	mAP: 19.955%	14	13	95
MLPerf BERT	9800-ORT-RTX	F1: 90.874%	41	38	27
	9800-PT-RTX	F1: 90.881%	21	18	67
	9800-ORT	F1: 90.874%	2.2	2.5	487
	9800-PT	F1: 90.874%	0.86	0.85	1305
	7820-ORT-TITAN	F1: 90.874%	30	29	35
	7820-PT-TITAN	F1: 90.874%	27	26	39
	AMD-ORT-A100	F1: 90.879%	92	78	15
	AMD-ORT-V100	F1: 90.874%	29	29	37
MLPerf 3D-UNet	AMD-ORT-A100	mean: 0.85300	0.043	0.045	22655
	AMD-ORT-V100	mean: 0.85300	0.045	0.045	22194

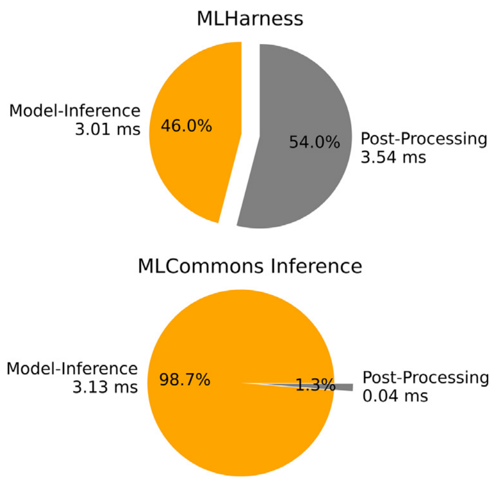


Fig. 5. Break down execution time into model-inference time and post-processing time for MLHarness and MLCommons Inference running Offline scenario with ResNet50 and a single input on 9800-ORT-RTX.

behavior by delving deeper into the results. Fig. 6 shows the performance of `ResNet50` with batch size one across the system AMD-ORT-V100 and system AMD-ORT-A100 at both the layer and the kernel (sub-layer) granularity levels, respectively. At the layer granularity, we observe that the end-to-end inference time on system AMD-ORT-V100 is indeed shorter than that on AMD-ORT-A100, and the reduced runtime mainly comes from the shortened runtime of many Conv2+ReLu layers (in orange color). For example, by focusing only on the second to the last Conv2+ReLu layer, we see that the duration on system AMD-ORT-A100 is almost twice as large as the duration on system AMD-ORT-V100. By zooming into that particular layer at the kernel level granularity, we quickly realize that the two systems have executed different GPU kernels. For system AMD-ORT-V100, there are two major kernels, i.e., cudnn::winograd::generateWinogradTilesKernel and volta_scudnn_winograd_128x128. In contrast, for system AMD-ORT-A100, there is only one major kernel, i.e., implicit_convolve_sgemm. We suspect that this discrepancy in performance is mainly due to the less optimized kernel selection algorithm offered by the newer system CUDNN library (v8.1) for A100 GPUs than for V100 GPUs. This further validates the importance of full-stack optimization for system performance.

In summary, we show that MLHarness is capable of reporting MLCommons Inference defined metrics by encapsulating MLModelScope [3] as an easy-to-use black box into MLCommons Inference, and that our harness system is able to benchmark models that are not supported in MLModelScope, including `BERT` for question answering and `3D-UNet` for medical image 3D segmentation, with the help of the new interfaces for user-defined pre-processing and post-processing functions. Moreover, as MLHarness is built on top of MLModelScope, we are able to utilize its across-stack profiling and analysis capabilities to align the information across the ML/DL framework level and the accelerating library level, and pinpoint critical distinctions between models, frameworks, and system.

4.3. Results of models beyond MLCommons Inference

Unlike the first set of experiments, which focuses on showcasing the success of MLHarness in orchestrating MLCommons Inference [1]

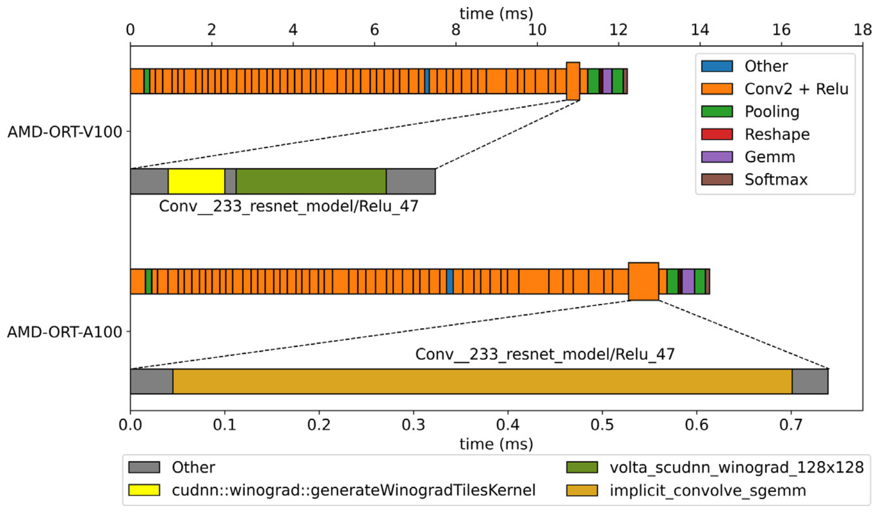


Fig. 6. Performance of ResNet50 with batch size one across systems AMD-ORT-V100 and AMD-ORT-A100 at both layer and kernel (sub-layer) granularity levels, respectively. The axis on the top is the duration to execute each layer in the model, while the axis at the bottom is the duration to execute kernels of the second to the last Conv2 + Relu layer.

Table 4
MLHarness reported results for models beyond MLCommons Inference using PyTorch and ONNX Runtime as ML/DL frameworks.

Model	System	Accuracy	Offline (sample/s)	Single-Stream	
				(sample/s)	90th percentile latency (ms)
TorchVision AlexNet	9800-ORT-RTX	Top1: 56.520%	218	171	6.1
	9800-PT-RTX	Top1: 56.516%	191	154	6.8
	9800-ORT	Top1: 56.522%	86	81	12
	9800-PT	Top1: 56.522%	12	12	90
	7820-ORT-TITAN	Top1: 56.522%	219	168	6.1
	7820-PT-TITAN	Top1: 56.522%	186	152	6.9
TorchVision ResNet18	9800-ORT-RTX	Top1: 69.758%	179	144	7.3
	9800-PT-RTX	Top1: 69.756%	122	113	9.6
	9800-ORT	Top1: 69.758%	128	118	8.8
	9800-PT	Top1: 69.758%	28	32	42
	7820-ORT-TITAN	Top1: 69.758%	175	145	7.2
	7820-PT-TITAN	Top1: 69.758%	132	119	9.2
TorchVision ResNet34	9800-ORT-RTX	Top1: 73.314%	142	124	8.7
	9800-PT-RTX	Top1: 73.306%	90	89	12
	9800-ORT	Top1: 73.314%	72	70	14
	9800-PT	Top1: 73.314%	20	19	65
	7820-ORT-TITAN	Top1: 73.314%	144	125	8.5
	7820-PT-TITAN	Top1: 73.314%	98	93	12
TorchVision ResNet50	9800-ORT-RTX	Top1: 76.130%	129	115	9.4
	9800-PT-RTX	Top1: 76.132%	76	76	15
	9800-ORT	Top1: 76.130%	63	61	16
	9800-PT	Top1: 76.130%	7.9	7.7	149
	7820-ORT-TITAN	Top1: 76.130%	128	112	9.6
	7820-PT-TITAN	Top1: 76.130%	79	78	15
TorchVision ResNet101	9800-ORT-RTX	Top1: 77.374%	100	89	12
	9800-PT-RTX	Top1: 77.376%	59	68	22
	9800-ORT	Top1: 77.374%	36	36	29
	9800-PT	Top1: 77.374%	4.7	4.8	239
	7820-ORT-TITAN	Top1: 77.374%	94	87	12
	7820-PT-TITAN	Top1: 77.374%	60	67	22
TorchVision ResNet152	9800-ORT-RTX	Top1: 78.310%	84	76	15
	9800-PT-RTX	Top1: 78.312%	46	64	26
	9800-ORT	Top1: 78.312%	26	26	41
	9800-PT	Top1: 78.312%	3.5	3.5	324
	7820-ORT-TITAN	Top1: 78.312%	74	72	15
	7820-PT-TITAN	Top1: 78.312%	52	60	26

way of benchmarking MLCommons Inference models, the second set of experiments illustrates how to make use of the exchange specification and the across-stack profiling and analysis tool in MLModelScope [3] to facilitate developments and comparisons of ML/DL innovations in the context of MLCommons Inference methodologies.

Table 4 shows a sample of six models to demonstrate how easy it is to use MLHarness to scale the MLCommons Inference way of benchmarking of various models beyond MLCommons Inference on a variety of system configurations. In this particular example, these results further show the relationships among the depth of the convolutional neural networks, the accuracy, and the throughput. The six models

Table 5
MLHarness reported results for models beyond MLCommons Inference using TensorFlow and MXNet as ML/DL frameworks.

Model	System	Accuracy	Offline (sample/s)	Single-Stream	
				(sample/s)	90th percentile latency (ms)
VGG16	7820-TF	Top1: 70.962%	9.3	9.2	115
	9800-MX-RTX	Top1: 72.852%	100	88	11
VGG19	7820-TF	Top1: 71.056%	8.7	8.6	123
	9800-MX-RTX	Top1: 73.814%	91	82	12

are AlexNet along with five models from the ResNet family. All of these models are from TorchVision [9], where the implementation details and the reference accuracy can be found at its GitHub page [28]. Again, the success of importing these models into MLHarness using the exchange specification is validated by the accuracy results, where all of them are within at least 99% of the reference accuracy as stated by TorchVision. In addition, the pre-processing and post-processing functions in the exchange specification can be regarded as a reusable component because these models share the same pre-processing and post-processing steps.

Fig. 7 compares the accuracy of the six convolutional neural networks in systems 9800-ORT-RTX to 7820-PT-TITAN as listed in Table 1. The models are placed in increasing order of depth from left to right, with ResNet152 being the deepest. As expected, there is no huge variance on the accuracy across systems, but the deeper the convolutional neural network is, the more accurate the model is.

Fig. 8 compares the throughput of the six convolutional neural networks in system 9800-ORT-RTX to 7820-PT-TITAN as listed in Table 1. From Fig. 8, we observe that there is a trend that the deeper the convolutional neural network is, the lower the throughput it has. However, for the two systems 9800-ORT and 9800-PT, which are the two system configurations without GPUs, are not following the trend when comparing AlexNet and ResNet18. As our MLHarness is built on top of MLModelScope, we then use the across-stack profiling and analysis tool, XSP [18], to identify the bottleneck. Table 6 shows the top three most time-consuming layers of AlexNet and ResNet18 identified by XSP on system 9800-PT, which has no GPU support and has PyTorch as the ML/DL framework. It clearly points out that the bottleneck of AlexNet is from matrix multiplications of fully connected layers. Although there is also a fully connected layer in ResNet18 as recorded by XSP, its size is 512 by 1000, which is much smaller than the largest one in AlexNet, whose size is 4096 by 4096.

Although in Table 4, we use the same implementations of models by converting PyTorch models to ONNX formats that can be used in ONNX Runtime, it is also valuable to compare the same structure of model with different implementations and training processes. Table 5 shows the experiments on the models from the VGG family using TensorFlow and MXNet as ML/DL frameworks, where the models for TensorFlow can be found at TensorFlow Model Graden [29] and the models for MXNet can be found at GluonCV [4]. From Table 5, we can observe that the accuracy is different between implementations of the same model, which further illustrates the difficulty of model reproducibility. This also shows how flexible MLHarness is in terms of running scalable benchmarking across different combinations of models and frameworks by utilizing the extended exchange specification as discussed in this work, and how scalable experimentation helps to identify common issues convincingly.

In summary, these exemplar experiments as discussed in this section show not only that it is easy to add models into MLHarness by utilizing the extended exchange specification and to report MLCommons Inference defined metrics for models that are beyond MLCommons Inference, but also that, with the help of MLModelScope, MLHarness can easily and scalably compare models and extract critical and detailed information, which is impossible when merely using MLCommons Inference.

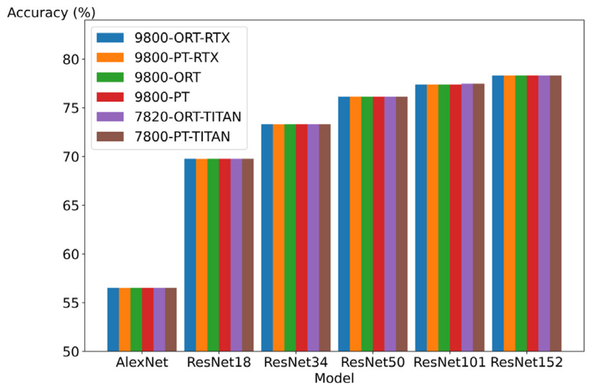


Fig. 7. Accuracy of models in different systems.

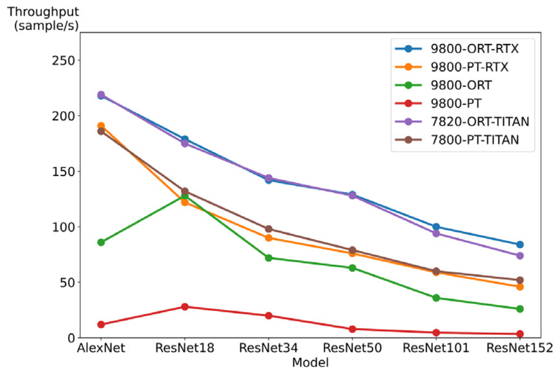


Fig. 8. Offline throughput of models in different systems.

Table 6
The top-3 most time-consuming layers of AlexNet and ResNet18 on system 9800-PT.

AlexNet		ResNet18	
Layer name	Latency (ms)	Layer name	Latency (ms)
aten::mm	47.99	aten::maxpool2d	6.31
aten::mm	17.99	aten::convolution	2.46
aten::mm	4.13	aten::convolution	2.24

4.4. Impact of MLHarness

The experimental results above demonstrate the success of MLHarness in benchmarking ML/DL model inferences by providing an extended exchange specification for researchers to easily plug in their ML/DL innovations and collect a set of well-defined metrics. One of our near future goals is to further extend MLHarness to support MLCommons training [11]. Nevertheless, the impact of MLHarness is not only restricted to ML/DL community. Benchmarking, reproducibility, portability, and scalability are important aspects in any computing-related research, such as high performance computing and computational biology. The success of MLHarness is only a starting point, from which we are aiming for extending the same techniques to other research domains that utilize heterogeneous computational resources, and providing a scalable and flexible harness system to overcome the similar set of challenges.

5. Conclusion

As ML/DL community is flourishing, it becomes increasingly imperative to standardize a common set of measures for people to benchmark and compare ML/DL models quality and performance on a common ground. In this paper, we present MLHarness, a scalable benchmarking harness system, to remedy and ease the adoption of ML/DL innovations. Our experimental results show superior flexibility and scalability of MLHarness for benchmarking and porting, by utilizing the extended exchange specification and reporting community acknowledged metrics. We also show that with the help of MLHarness, we are able to easily pinpoint critical distinctions between ML/DL innovations, by inspecting and aligning profiling information across stacks.

Acknowledgments

This work is supported in part by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network.

References

[1] V.J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J.S. Gardner, I. Hubara, S. Idgunji, T.B. Jablin, J. Jiao, T.S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A.T.R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, Y. Zhou, Mlperf inference benchmark, 2019, http://arxiv.org/abs/1911.02549.

[2] Mlcommons inference, 2021, URL https://github.com/mlcommons/inference, Accessed: 2021-07-22.

[3] A. Dakkak, C. Li, J. Xiong, W. Hwu, MlModelScope: A distributed platform for model evaluation and benchmarking at scale, 2020, CoRR https://arxiv.org/abs/2002.08295.

[4] Gluoncv, 2021, URL https://cv.gluon.ai/model_zoo/index.html, Accessed: 2021-07-21.

[5] Modelhub, 2021, URL http://modelhub.ai/, Accessed: 2021-07-22.

[6] Modelzoo, 2021, URL https://modelzoo.co/, Accessed: 2021-07-21.

[7] Onnx model zoo, 2021, URL https://github.com/onnx/models, Accessed: 2021-07-21.

[8] Tensorflow hub, 2021, URL https://www.tensorflow.org/hub, Accessed: 2021-07-21.

[9] Torchvision, 2021, URL https://pytorch.org/vision/stable/index.html, Accessed: 2021-07-21.

[10] T. Gamblin, M. LeGendre, M.R. Collette, G.L. Lee, A. Moody, B.R. de Supinski, S. Futral, The spack package manager: bringing order to HPC software chaos, in: SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–12, http://dx.doi.org/10.1145/2807591.2807623.

[11] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, A. Ike, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V.J. Reddi, T. Robie, T.S. John, T. Tabaru, C.-J. Wu, L. Xu, M. Yamazaki, C. Young, M. Zaharia, Mlperf training benchmark, 2019, http://arxiv.org/abs/1910.01500.

[12] Architecture for mlops using TFX, kubeflow pipelines, and cloud build, 2021, URL https://bit.ly/39P6JFk Accessed: 2021-07-21.

[13] G. Fursin, Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common APIs, 2020, CoRR https://arxiv.org/abs/2011.01149.

[14] Mlops: Model management, deployment, lineage and monitoring with azure machine learning, 2021, URL https://docs.microsoft.com/en-us/azure/machine-learning/concept-model-management-and-deployment, Accessed: 2021-07-21.

[15] Mlcube, 2021, URL https://github.com/mlcommons/mlcube, Accessed: 2021-07-21.

[16] W. Brewer, G. Behm, A. Scheinine, B. Parsons, W. Emeneker, R.P. Trevino, Ibench: a distributed inference simulation and benchmark suite, in: 2020 IEEE High Performance Extreme Computing Conference (HPEC), 2020, pp. 1–6, http://dx.doi.org/10.1109/HPEC43674.2020.9286169.

[17] A. Dakkak, C. Li, J. Xiong, W. Hwu, Dlspec: A deep learning task exchange specification, 2020, CoRR https://arxiv.org/abs/2002.11262.

[18] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, W. Hwu, Across-stack profiling and characterization of machine learning models on GPUs, 2019, CoRR http://arxiv.org/abs/1908.06869.

[19] Python/c API reference manual, 2021, URL https://docs.python.org/3/c-api/, Accessed: 2021-07-21.

[20] Go-Python3, 2021, URL https://github.com/DataDog/go-python3, Accessed: 2021-07-21.

[21] Go package help, 2021, URL https://pkg.go.dev/cmd/go/internal/help, Accessed: 2021-07-21.

[22] Ctypes — A foreign function library for python, 2021, URL https://docs.python.org/3/library/ctypes.html, Accessed: 2021-07-21.

[23] Command cgo, 2021, URL https://pkg.go.dev/cmd/cgo, Accessed: 2021-07-21.

[24] Go package runtime, 2021, URL https://pkg.go.dev/runtime, Accessed: 2021-07-21.

[25] Reference counting, 2021, URL https://docs.python.org/3/c-api/refcounting.html, Accessed: 2021-07-21.

[26] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, 2018, CoRR http://arxiv.org/abs/1810.04805.

[27] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T.L. Scao, S. Gugger, M. Drame, Q. Lhoest, A.M. Rush, Transformers: State-of-the-art natural language processing, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Association for Computational Linguistics, Online, 2020, pp. 38–45, URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[28] Torchvision github, 2021, URL https://github.com/pytorch/vision/tree/v0.10.0, Accessed: 2021-08-05.

[29] H. Yu, C. Chen, X. Du, Y. Li, A. Rashwan, L. Hou, P. Jin, F. Yang, F. Liu, J. Kim, J. Li, TensorFlow Model garden, 2020, https://github.com/tensorflow/models.