Tutorial: modularizing and automating MLPerf

This tutorial is from https://github.com/mlcommons/ck/tree/master/docs/tutorials, as of commit 57dbe97

▶ Click here to see the table of contents.

Introduction

This tutorial was prepared for the Student Cluster Competition'22 to explain how to prepare and run a modular version of the MLPerf inference benchmark using the cross-platform automation meta-framework (MLCommons CM). It is assembled from reusable and interoperable MLOps and DevOps scripts being developed by the open MLCommons taskforce on education and reproducibility based on this roadmap.

There are 4 main goals:

- Trying the MLCommons CM meta-framework for modular benchmarking.
- Learning how to prepare and run the MLPerf inference benchmark using CM.
- Obtaining and submitting benchmarking results (accuracy and performance) for MLPerf object detection with RetinaNet in offline mode on your platform (see related slides).
- Learning how to optimize this benchmark, submit your results to the MLPerf inference v3.0 (March 2023) and get them to the scoreboard similar to v2.1.

It should take less than an hour to complete this tutorial. In the end, you should obtain a tarball (open.tar.gz) with the MLPerf-compatible results. You should submit it to the SCC'22 organizers to get extra points.

During SCC, you will attempt to run a reference (unoptimized) Python implementation of the MLPerf object detection benchmark with RetinaNet model, Open Images dataset, ONNX runtime and CPU target.

After the SCC, you are welcome to join the open MLCommons taskforce on education and reproducibility to learn how to optimize this benchmark further (trying various run-time parameters, trying different benchmark implementations (Nvidia, C++), changing ML frameworks and run-times, optimizing RetinaNet model, and trying different CPUs and GPUs) and submit Pareto-optimal results to MLPerf.

Note that both MLPerf and CM automation are evolving projects. If you encounter issues or have questions, please submit them here and feel free to join our weekly conf-calls.

System preparation

Minimal system requirements

- CPU: 1 node (x86-64 or Arm64)
- OS: we have tested this automation on Ubuntu 20.04, Ubuntu 22.04, Debian 10, Red Hat 9 and MacOS 13
- Disk space:
  - with a minimal preprocessed dataset for test runs: ~5GB
  - with a full preprocessed dataset for the official MLPerf submission: ~200GB
- Python: 3.8+
- All other dependencies (artifacts and tools) will be installed by the CM meta-framework

MLCommons CM automation meta-framework

The MLCommons is developing an open-source and technology-neutral Collective Mind meta-framework (CM) to modularize ML Systems and automate their benchmarking, optimization and design space exploration across continuously changing software, hardware and data.

CM is the second generation of the MLCommons CK workflow automation framework that was originally developed to make it easier to reproduce research papers at ML and Systems conferences. The goal is to help researchers unify and automate all the steps to prepare and run MLPerf and other benchmarks across diverse ML models, datasets, frameworks, compilers and hardware (see HPCA'22 presentation about our motivation).

CM installation

Follow this guide to install the MLCommons CM framework on your system.

After the installation, you should be able to access the CM command line as follows:

```
$ cm

cm {action} {automation} {artifact(s)} {--flags} @input.yaml @input.json
```

Note: we have prepared this tutorial using CM v1.1.1. You can enforce this version as follows:

```
python3 -m pip install cmind==1.1.1

cm --version
```

```
1.1.1
```

Pull CM repository with cross-platform MLOps and DevOps scripts

Pull stable MLCommons CM repository with cross-platform CM scripts for modular ML Systems:

```
cm pull repo mlcommons@ck
```

CM pulls all such repositories into the `$HOME/CM` directory to search for CM automations and artifacts. You can find the location of a pulled repository as follows:

```
cm find repo mlcommons@ck
```

You can also pull the stable version of this CM repository that we have used for this tutorial using checkout *f9abc76*:

```
cm pull repo mlcommons@ck --checkout=ac3fda5
```

You can now use the unified CM CLI/API of reusable and cross-platform CM scripts) to detect or install all artifacts (tools, models, datasets, libraries, etc) required for a given software project (MLPerf inference benchmark with RetinaNet, Open Images and ONNX in our case).

Conceptually, these scripts take some environment variables and files as an input, perform a cross-platform action (detect artifact, download files, install tools), prepare new environment variables and cache output if needed.

Note that CM can automatically detect or install all dependencies for a given benchmark and run it on a given platform in just one command using a simple JSON or YAML description of dependencies on all required CM scripts.

However, since the goal of this tutorial is to explain you how we modularize MLPerf and any other benchmark, we will show you all individual CM commands to prepare and run the MLPerf inference benchmark. You can reuse these commands in your own projects thus providing a common interface for research projects.

In the end, we will also show you how to run MLPerf benchmark in one command from scratch.

Optional: update CM and repository to the latest version

Note that if you already have CM and mlcommons@ck reposity installed on your system, you can update them to the latest version at any time and clean the CM cache as follows:

```
python3 -m pip install cmind -U
cm pull repo mlcommons@ck --checkout=master
cm rm cache -f
```

Install system dependencies for your platform

First, you need to install various system dependencies required by the MLPerf inference benchmark.

For this purpose, we have created a cross-platform CM script that will automatically install such dependencies based on your OS (Ubuntu, Debian, Red Hat, MacOS ...).

In this case, CM script serves simply as a wrapper with a unified and cross-platform interface for native scripts that you can find and extend here if some dependencies are missing on your machine - this is a collaborative way to make CM scripts portable and interoperable.

You can run this CM scripts as follows (note that you may be asked for a SUDO password on your platform):

```
cm run script "get sys-utils-cm" --quiet
```

If you think that you have all system dependencies installed, you can run this script without `--quiet` flag and type "skip" in the script prompt.

Use CM to detect or install Python 3.8+

Since we use Python reference implementation of the MLPerf inference benchmark (unoptimized), we need to detect or install Python 3.8+ (MLPerf requirement).

You need to detect it using the following CM script:

```
cm run script "get python" --version_min=3.8
```

Note, that all artifacts (including above scripts) in MLCommons CM are organized as a database of interconnected components. They can be found either by their user friendly tags (such as `get,python`) or aliases (`get-python3`) and unique identifiers (`5b4e0237da074764`). You can find this information in a [CM meta description of this script](#).

If required Python is already installed on your system, CM will detect it and will cache related environment variables such as PATH, PYTHONPATH, etc. to be reused by other CM scripts. You can find an associated CM cache entry for your python as follows:

```
cm show cache --tags=get,python
```

You can see the environment variables produced by this CM script in the following JSON file:

```
cat `cm find cache --tags=get,python`/cm-cached-state.json
```

If required Python is not detected, CM will automatically attempt to download and build it from sources using another [cross-platform CM script "install-python-src"](#). In the end, CM will also cache new binaries and related environment variables such as PATH, PYTHONPATH, etc:

```
cm show cache
```

You can find installed binaries and reuse them in your own project with or without CM as follows:

```
cm find cache --tags=install,python
```

Note that if you run the same script again, CM will automatically find and reuse the cached output:

```
cm run script "get python" --version_min=3.8 --out=json
```

Pull MLPerf inference sources

You should now download and cache the MLPerf inference sources using the following command:

```
cm run script "get mlperf inference src"
```

Compile MLPerf loadgen

You need to compile loadgen from the above inference sources while forcing compiler dependency to GCC:

```
cm run script "get mlperf loadgen" --adr.compiler.tags=gcc
```

The `--adr` flag stands for "Add to all Dependencies Recursively" and will find all sub-dependencies on other CM scripts in the CM loadgen script with the "compiler" name and will append "gcc" tag to enforce detection and usage of GCC to build loadgen.

CM automation for the MLPerf benchmark

MLPerf inference - Python - RetinaNet FP32 - Open Images - ONNX - CPU - Offline

Download Open Images dataset

You can now download the minimal Open Images validation datasets v6 with the first 500 images using the following CM script:

```
cm run script "get dataset object-detection open-images original _validation _500"
```

Note that `_` prefix means some variation of a script that can update environment variables and adds extra dependencies to customize the execution of a given script. See `"variation"` key in the meta description of this script here.

This script will automatically install various Python sub-dependencies and openssl to download and process this dataset. The minimal set will download 500 images and will need ~200MB of disk space.

After installing this dataset via CM, you can reuse it in your own projects or other CM scripts (including MLPerf benchmarks). You can check the CM cache as follows (the unique ID of the CM cache entry will be different on your machine):

```
cm show cache --tags=get,dataset,open-images,original
```

```

* cache::67d2c092e64744e5
    Tags: ['dataset', 'get', 'object-detection', 'open-images', 'openimages',
'original', 'script-artifact-0a9d49b644cf4142', '_500', '_validation']
    Path: /home/fursin/CM/repos/local/cache/67d2c092e64744e5
```

You can find the images and annotations in the CM cache as follows:

```
ls `cm find cache --tags=get,dataset,open-images,original`/install/validation/data
ls `cm find cache --tags=get,dataset,open-images,original`/install/annotations
```

Preprocess Open Images dataset

You now need to preprocess this dataset to convert it into a format consumable by the MLPerf inference benchmark using the following CM script (converting each jpeg image from the above dataset to binary numpy format and NCHW):

```
cm run script "get preprocessed dataset object-detection open-images _validation
_500 _NCHW"
```

You can find them in the CM cache as follows:

```
cm show cache --tags=get,preprocessed,dataset,open-images
```

```

* cache::6b13fc343a52499c
    Tags: ['dataset', 'get', 'object-detection', 'open-images', 'openimages',
'preprocessed', 'script-artifact-9842f1be8cba4c7b', '_500', '_NCHW', '_validation']
    Path: /home/fursin/CM/repos/local/cache/6b13fc343a52499c
```

Install ONNX runtime for CPU

Now detect or install ONNX Python runtime (targeting CPU) your system using a generic CM script to install python package:

```
cm run script "get generic-python-lib _onnxruntime" --version_min=1.10.0
```

Download RetinaNet model (FP32, ONNX format)

Download and cache this reference model in the ONNX format (float32) using the following CM script:

```
cm run script "get ml-model object-detection retinanet resnext50 _onnx"
```

It takes around ~150MB of disk space. You can find it in the CM cache as follows:

```
cm show cache --tags=get,ml-model,resnext50,_onnx
```

```

* cache::7e1ca80c06154f22
    Tags: ['fp32', 'get', 'ml-model', 'object-detection', 'resnext50', 'retinanet',
'script-artifact-427bc5665e4541c2', '_onnx']
    Path: /home/fursin/CM/repos/local/cache/7e1ca80c06154f22
```

```
ls `cm find cache --tags=get,ml-model,resnext50,_onnx`/*.onnx -l
```

Output:

```
148688824  resnext50_32x4d_fpn.onnx
```

Run reference MLPerf inference benchmark (offline, accuracy)

You are now ready to run the reference (unoptimized) Python implmenentaton of the MLPerf vision benchmark with ONNX backend.

Normally, you would need to go through this README.md and prepare all the dependencies and environment variables manually.

The CM "app-mlperf-inference" script allows you to run this benchmark as follows:

```
cm run script "app mlperf inference generic _python _retinanet _onnxruntime _cpu" \
     --scenario=Offline \
     --mode=accuracy \
     --test_query_count=10 \
     --rerun
```

This CM script will automatically find or install all dependencies described in its CM meta description, aggregate all environment variables, preprocess all files and assemble the MLPerf benchmark CMD.

It will take a few minutes to run it and you should see the following accuracy:

```
loading annotations into memory...
Done (t=0.02s)
creating index...
index created!
Loading and preparing results...
DONE (t=0.02s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=0.09s).
Accumulating evaluation results...
DONE (t=0.11s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.548
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.787
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.714
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.304
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.631
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.433
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.648
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.663
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.343
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.731

mAP=54.814%
```

Congratulations, you can now play with this benchmark using the unified CM commands!

Note that even if did not install all above dependencies manually, the same command will automatically install all the necessary dependencies (you just need to specify that you use GCC and 500 images).

You can check it by cleaning the CM cache and executing this command again (it will take around ~10 minutes depending on the speed of your system and the Internet connection):

```
cm rm cache -f

cm run script "app mlperf inference generic _python _retinanet _onnxruntime _cpu" \
     --adr.python.version_min=3.8 \
     --adr.compiler.tags=gcc \
     --adr.openimages-preprocessed.tags=_500 \
     --scenario=Offline \
     --mode=accuracy \
     --test_query_count=10 \
     --quiet \
     --rerun
```

Run MLPerf inference benchmark (offline, performance)

Let's run the MLPerf object detection while measuring performance:

```
cm run script "app mlperf inference generic _python _retinanet _onnxruntime _cpu" \
     --scenario=Offline \
     --mode=performance \
     --rerun
```

It will run for ~10 minutes and you should see the output similar to the following one in the end (the QPS is the performance result of this benchmark that depends on the speed of your system):

```

TestScenario.Offline qps=1.29, mean=59.7877, time=513.360, queries=660,
tiles=50.0:61.7757,80.0:63.8270,90.0:66.5430,95.0:67.6991,99.0:69.2812,99.9:70.5251



================================================
MLPerf Results Summary
================================================
...

No warnings encountered during test.

No errors encountered during test.

  - running time of script
"app,vision,language,mlcommons,mlperf,inference,reference,generic,ref": 529.25 sec.
```

Note that QPS is very low because we use an unoptimized reference implementation of this benchmark on CPU. In the 2nd part of this tutorial, we will explain how to optimize this benchmark and/or run other implementations such as the universal C++ implementation of this benchmark developed by OctoML and the MLCommons taskforce on education and reproducibility as well as optimized implementation of MLPerf object detection with quantized models from Nvidia.

You can also find the reference Python implementation of this benchmark in the CM cache as follows:

```
cd `cm show cache --
tags=get,mlperf,src,_default`/inference/vision/classification_and_detection/python
```

You can then modify it and rerun the above command to see the effects of your changes on the accuracy and performance of the MLPerf benchmark.

Customize MLPerf inference benchmark

The execution of the original MLPerf inference benchmark is customized by various flags and environment variables.

The "Collective Mind" concept is to gradually expose all optimization "knobs" via unified CM script interface to enable automated and reroducible design space exploration and optimization of the whole application/software/hardware stack (one of the goals of the MLCommons taskforce on education and reproducibility).

That is why we have provided a user-friendly mapping of the flags from the CK MLPerf script CLI to the native MLPerf variables and flags using this meta description.

For example, you can specify a number of threads used by this benchmark as follows:

```
cm run script "app mlperf inference generic _python _retinanet _onnxruntime _cpu" \
     --scenario=Offline \
     --mode=performance \
     --rerun \
     --num_threads=4
```

Prepare MLPerf submission

You are now ready to generate the submission similar to the ones appearing on the official MLPerf inference dashboard.

We have developed another script that runs the MLPerf inference benchmark in both accuracy and performance mode, runs the submission checker, unifies output for a dashboard and creates a valid MLPerf submission pack in `open.tar.gz` with all required MLPerf logs and stats.

You can run this script as follows (just substitute *OctoML* with the name of your organization):

```
cm run script --tags=run,mlperf,inference,generate-run-cmds,_submission,_short \
      --submitter="OctoML" \
      --hw_name=default \
      --lang=python \
      --model=retinanet \
      --backend=onnxruntime \
      --device=cpu \
      --scenario=Offline \
      --test_query_count=10 \
      --clean
```

It will take around 15-30 minutes to run and you should see the following output in the end:

```

[2022-11-09 16:33:45,968 log_parser.py:50 INFO] Sucessfully loaded MLPerf log from
open/OctoML/results/onnxruntime-
cpu/retinanet/offline/accuracy/mlperf_log_detail.txt.
[2022-11-09 16:33:45,969 log_parser.py:50 INFO] Sucessfully loaded MLPerf log from
open/OctoML/results/onnxruntime-
cpu/retinanet/offline/performance/run_1/mlperf_log_detail.txt.
```

```
[2022-11-09 16:33:45,971 log_parser.py:50 INFO] Sucessfully loaded MLPerf log from
open/OctoML/results/onnxruntime-
cpu/retinanet/offline/performance/run_1/mlperf_log_detail.txt.
[2022-11-09 16:33:45,971 submission_checker1.py:1516 INFO] Target latency: None,
Latency: 504767463228, Scenario: Offline
[2022-11-09 16:33:45,971 submission_checker1.py:2455 INFO] ---
[2022-11-09 16:33:45,971 submission_checker1.py:2459 INFO] Results
open/OctoML/results/onnxruntime-cpu/retinanet/offline 1.30475
[2022-11-09 16:33:45,971 submission_checker1.py:2461 INFO] ---
[2022-11-09 16:33:45,971 submission_checker1.py:2467 INFO] ---
[2022-11-09 16:33:45,971 submission_checker1.py:2468 INFO] Results=1, NoResults=0
[2022-11-09 16:33:45,971 submission_checker1.py:2474 INFO] SUMMARY: submission looks
OK
                                                                             0
Organization                                                            OctoML
Availability                                                         available
Division                                                                  open
SystemType                                                                edge
SystemName                 mlperf-tests-e2-x86-16-64-ubuntu-22 (auto dete...
Platform                                                       onnxruntime-cpu
Model                                                                retinanet
MlperfModel                                                          retinanet
Scenario                                                               Offline
Result                                                                 1.30475
Accuracy                                                                54.814
number_of_nodes                                                              1
host_processor_model_name                      Intel(R) Xeon(R) CPU @ 2.20GHz
host_processors_per_node                                                     1
host_processor_core_count                                                   16
accelerator_model_name                                                     NaN
accelerators_per_node                                                        0
Location                   open/OctoML/results/onnxruntime-cpu/retinanet/...
framework                                                  onnxruntime v1.13.1
operating_system              Ubuntu 22.04 (linux-5.15.0-1021-gcp-glibc2.35)
notes                                                                      NaN
compilance                                                                   1
errors                                                                       0
version                                                                   v2.1
inferred                                                                     0
has_power                                                                False
Units                                                                Samples/s
```

Note that `--clean` flag cleans all previous runs of MLPerf benchmark to make sure that the MLPerf submission script picks up the latest results.

You will also see the following 3 files in your current directory:

```
ls -l

open.tar.gz
```

```
summary.csv
summary.json
```

You should submit these files to the organizing committee to get extra points in the Student Cluster Competition.

Note that by default, CM-MLPerf will store the raw results in `$HOME/mlperf_submission` (with truncated accuracy logs) and in `$HOME/mlperf_submission_logs` (with complete and very large accuracy logs).

You can change this directory using the flag `--submission_dir={directory to store raw MLPerf results}` in the above script.

Push results to a live dashboard

You can attempt to push your results to public W&B dashboard for SCC'22. You just need to rerun the above command with `_dashboard` variation:

```
cm run script --tags=run,mlperf,inference,generate-run-
cmds,_submission,_short,_dashboard \
      --submitter="OctoML" \
      --hw_name=default \
      --lang=python \
      --model=retinanet \
      --backend=onnxruntime \
      --device=cpu \
      --scenario=Offline \
      --test_query_count=10
```

You should normally see your results at this live W&B dashboard or at the newer version.

Summary

Here is a compact list of CM commands to prepare and run the MLPerf object detection benchmark with RetinaNet, Open Images, ONNX runtime (CPU) on Ubuntu 22.04:

With explicit dependencies first

```

sudo apt update && sudo apt upgrade
sudo apt install python3 python3-pip python3-venv git wget

python3 -m pip install cmind
source $HOME/.profile

cm pull repo mlcommons@ck

cm run script "get sys-utils-cm" --quiet

cm run script "get python" --version_min=3.8

cm run script "get mlperf inference src"
cm run script "get mlperf loadgen" --adr.compiler.tags=gcc

cm run script "get dataset object-detection open-images original _validation _500"

cm run script "get preprocessed dataset object-detection open-images _validation
_500 _NCHW"

cm run script "get generic-python-lib _onnxruntime" --version_min=1.10.0

cm run script "get ml-model object-detection retinanet resnext50 _onnx"

cm run script "app mlperf inference generic _python _retinanet _onnxruntime _cpu" \
     --scenario=Offline --mode=accuracy --test_query_count=10 --rerun

cm run script "app mlperf inference generic _python _retinanet _onnxruntime _cpu" \
     --scenario=Offline --mode=performance --rerun

cm run script --tags=run,mlperf,inference,generate-run-cmds,_submission,_short \
      --submitter="OctoML" \
      --hw_name=default \
      --lang=python \
      --model=retinanet \
      --backend=onnxruntime \
      --device=cpu \
      --scenario=Offline \
      --test_query_count=10 \
      --clean

cm run script --tags=run,mlperf,inference,generate-run-
cmds,_submission,_short,_dashboard \
      --submitter="OctoML" \
      --hw_name=default \
      --lang=python \
      --model=retinanet \
      --backend=onnxruntime \
      --device=cpu \
      --scenario=Offline \
      --test_query_count=10 \
      --clean
```

With one CM command that will install all dependencies automatically

```
sudo apt update && sudo apt upgrade
sudo apt install python3 python3-pip python3-venv git wget

python3 -m pip install cmind
source $HOME/.profile

cm pull repo mlcommons@ck
cm run script --tags=run,mlperf,inference,generate-run-cmds,_submission,_short,_dashboard \
      --adr.python.version_min=3.8 \
      --adr.compiler.tags=gcc \
      --adr.openimages-preprocessed.tags=_500 \
      --submitter="OctoML" \
      --hw_name=default \
      --lang=python \
      --model=retinanet \
      --backend=onnxruntime \
      --device=cpu \
      --scenario=Offline \
      --test_query_count=10 \
      --clean
```

Use Python virtual environment with CM and MLPerf

If you prefer to avoid installing all above python packages to your native Python, you can install multiple virtual environments using the same CM interface.

Here are the CM instructions to run the MLPerf benchmark in the Python virtual environment called "mlperf":

```
cm pull repo mlcommons@ck

cm run script "get sys-utils-cm" --quiet

cm run script "install python-venv" --version=3.10.8 --name=mlperf

cm run script --tags=run,mlperf,inference,generate-run-cmds,_submission,_short,_dashboard \
      --adr.python.name=mlperf \
      --adr.python.version_min=3.8 \
      --adr.compiler.tags=gcc \
      --adr.openimages-preprocessed.tags=_500 \
      --submitter="OctoML" \
      --hw_name=default \
      --model=retinanet \
      --backend=onnxruntime \
      --device=cpu \
      --scenario=Offline \
      --test_query_count=10 \
      --clean
```

Note that you need to add a flag `--adr.python.name={name of a virtual environment (mlperf)`.

The next steps

Please check other parts of this tutorial to learn how to use other implementation of the MLPerf inference benchmark (C++, Nvidia, etc) with other ML engines (PyTorch, TF, TVM), other MLPerf scenarios (single stream, multiple stream, server), quantized/pruned models and GPUs:

- 2nd part: customize MLPerf inference (C++ implementation, CUDA, PyTorch)
- 3rd part: customize MLPerf inference (ResNet50 Int8, ImageNet, TVM)
- *To be continued*

You are welcome to join the open MLCommons taskforce on education and reproducibility to contribute to this project and continue optimizing this benchmark and prepare an official submission for MLPerf inference v3.0 (March 2023) with the help of the community.

See the development roadmap here.

Authors

- Grigori Fursin (OctoML, MLCommons, cTuning foundation)
- Arjun Suresh (OctoML, MLCommons)

Acknowledgments

We thank Hai Ah Nam, Steve Leak, Vijay Janappa Reddi, Tom Jablin, Ramesh N Chukka, Peter Mattson, David Kanter, Pablo Gonzalez Mesa, Thomas Zhu, Thomas Schmid and Gaurav Verma for their suggestions and contributions.