A. Artifact Appendix

A.1 Abstract

Our artifact provides all the scripts and source code to reproduce the performance data described in the paper. It contains the detailed reproduce workflow on TVM (includes end-to-end evaluation and per-layer analysis), oneDNN Primitives & oneDNN Graph Compiler (includes model-level and subgraph-level evaluation, and ablation experiments on coarse-grain fusion). The artifact is supposed to work on Ubuntu20.04 x86-64 system.

A.2 Artifact check-list (meta-information)

- **Algorithm:** oneDNN Graph Compiler.
- **Program:** Podman image (4.4G), IPEX v1.13, Intel AI Model Zoo v1.13, oneDNN, and TLCBench (all sources are available on public repository).
- **Compilation:** GCC-11. CMake 3.19 or above.
- **Transformations:** LLVM 13.
- **Data set:** Benchmark Bert Large and DLRM requires BERT Large SQuAD1.1 and Criteo Terabyte, respectively. The detailed information can be found in Bert Large and DLRM.
- **Run-time environment:** Ubuntu 20.04 on x86-64 system.
- **Hardware:** We recommend to reproduce on an Intel® Xeon® Platinum 8358 processor with 32 cores to get comparable data claimed in our paper. You can also test on other Intel® Xeon® Platinum processors, but the performance data may vary.
- **Output:** The output is the execution time (ms) for the running workload.
- **Metrics:** We use execution time (ms) per iteration for evaluation.
- **Experiment workflow:** Manual Linux shell scripts.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How delivered

We provide a podman image, which includes the pre-requisite cmake, gcc, conda and llvm. Our benchmarks, source code, and scripts are available on Github. For TVM: TLCBench, TVM. For oneDNN Primitives and oneDNN Graph Compiler: WorkloadBench, oneDNN-reproduce. For end-to-end workloads: ModelBench, model-zoo-r1.13, IPEX-1.13.

A.3.2 Hardware dependencies

We recommend to reproduce on an Intel® Xeon® Platinum 8358 processor with 32 cores, which should give comparable results. Other Intel® Xeon® Platinum processors is supposed to support this artifact as well, but may generate different performance data.

A.3.3 Software dependencies

We collect performance data with Ubuntu20.04 on x86-64 system. Similar Linux distribution is supposed to work as well. Besides, GCC $\geq$ 11, CMake $\geq$ 3.19, LLVM $\geq$ 13, and Python $\geq$ 3.8 are required. MiniConda is recommended to manage TVM or IPEX environment. The corresponding podman image is provided, which contains the above mentioned dependencies.

A.4 Installation

A.5 Podman image

Podman is similar to Docker, which is a containerization tool used for creating, managing, and deploying applications within isolated environments. Note that, Docker command is applicable to this podman image. You can first download the podman image, then follow the below instructions to install podman, so that you can setup the software dependency to reproduce our described data. We also provide the Dockerfile in the image folder for your inference.

```
# install podman
$ apt get install podman
# load the reproduce image
$ podman load -i reproduce-anonymous-compiler-image.tar
# you can also use Docker to load the image, like
# £ docker load -i
↪  reproduce-anonymous-compiler-image.tar
# check your local image
$ podman images
# run the Podman container
$ podman run -it --rm
↪  localhost/reproduce-anonymous-compiler-image
# llvm path: /llvm-project/install/bin/llvm-config
```

A.5.1 TVM

Install TVM and get subgraph benching workloads:

```
$ git clone --branch bench
↪  https://github.com/crazydemo/TLCBench.git
$ source setup_env.sh
```

A.5.2 oneDNN Primitives & oneDNN Graph Compiler

Install oneDNN and prepare subgraph benching workloads:

```
$ git clone
↪  https://github.com/yifeizh2/WorkloadBench.git
$ source setup_env.sh
```

A.5.3 End-to-End model

Prepare IPEX for AI model benchmark:

```
$ git clone https://github.com/crazydemo/ModelBench.git
$ source setup_env.sh
```

A.6 Experiment workflow

A.6.1 TVM

We provide the Auto-scheduler tuned logs in **folder: tmp_logs_layers**. You can bench them easily with following commands:

```
$ conda activate tvm && cd TLCBench
$ bash bench.sh
```

You can specify the **network**, **batch-size** and **dtype** args to switch different workload (*e.g.* MLP1, MHA1) with various shapes and data types. If you would like to do the per-layer performance analysis, set **profiling=True**. And you may need to set **repeat** $\geq$ **20**, as the profiling mode need more iterations for warm up.

A.6.2 oneDNN Primitives & oneDNN Graph Compiler

We provide several scripts for reproducing oneDNN Primitives & oneDNN Graph Compiler's performance on individual matmul as well as MLP and MHA subgraphs.

As for individual matmul, to better emulate end-to-end DL workloads, instead of running each MLP layer separately, we run the entire MLP subgraph and collect the per-layer performance breakdown.

oneDNN Primitives individual matmul performance:

oneDNN Primitives' individual matmul performance can be benched and collected by running the `bench_primitive_single_matmul.sh` script.

```
$ conda activate benchdnn_graph && cd WorkloadBench
$ bash bench_primitive_single_matmul.sh
```

oneDNN Graph Compiler individual matmul kernel performance: oneDNN Graph Compiler's individual matmul performance is collected with its runtime tracing utility. By running the commands below, trace files in chrome tracing format will be dumped to the working directory. The provided script will further parse the trace files, extract per-layer performance, and summarize them into `compiler_single_matmul.csv`.

```
$ conda activate benchdnn_graph && cd WorkloadBench
$ bash bench_compiler_matmul.sh
```

oneDNN Primitives & oneDNN Graph Compiler subgraph performance:

The MLP and MHA subgraph performance for oneDNN Primitives, oneDNN Graph Compiler without coarse-grain fusion, and oneDNN Graph Compiler with coarse-grain fusion can be collected with the commands below.

```
$ conda activate benchdnn_graph && cd WorkloadBench
$ bash bench_subgraph_workloads.sh
```

A.6.3 End-to-end model

By default, only **int8** mode of IPEX will run into oneDNN Graph. If you want to benchmark the performance under **fp32** mode with oneDNN Graph and Compiler, you need to manually do some modification on IPEX and Model Zoo source code.

For IPEX:

```
// navigate to
↪  csrc/cpu/jit/codegen/onednn/interface.cpp change
↪  the value of llga_fp32_bf16_enabled from false to
↪  true
bool llga_fp32_bf16_enabled = true;
```

Then,

```
$ export DNNL_GRAPH_BUILD_COMPILER_BACKEND=1
# re-build IPEX
$ python setup.py install
```

For Model Zoo Scripts, you need to search for the model's main script and **comment out** the **ipex.optimize** in the corresponding dtype's condition branch. For instance, as for DLRM model under fp32 mode, comment out the following code:

```
# navigate to dlrm_s_pytorch.py
$ # dlrm = ipex.optimize(dlrm, dtype=torch.float, ...)
```

Note that, the above changes need to be reverted when you are running in int8 mode. As for benchmark AI models, you can follow the instructions on Bert Large and DLRM to prepare the model specified dependency and datasets. For bench **DLRM** model, you need to modify the below two args to switch to throughput mode (32 cores per instance).

```
# OMP_NUM_THREADS=1 => OMP_NUM_THREADS=32
# -share-weight-instance=£Cores =>
↪  -share-weight-instance=0
```

By default, oneDNN Graph Compiler is enabled on IPEX. To get oneDNN Primitives' performance data on end-to-end model, you need to set the below environment variable:

```
$ export _DNNL_DISABLE_COMPILER_BACKEND=1
```

A.7 Evaluation and expected result

To check whether the oneDNN Graph Compiler is enabled or not, you can check the graph verbose (via export ONEDNN_GRAPH_VERBOSE=1). You will see the below outputs from verbose once oneDNN Graph Compiler backend is enabled.

```
onednn_graph_verbose,info,backend,0:compiler_backend
onednn_graph_verbose,info,backend,1:dnnl_backend
```