A Artifact Appendix

A.1 Abstract

This artifact appendix helps readers reproduce the main experimental results for the SOSP'19 paper: Optimizing Deep Learning Computation with Automated Generation of Graph Substitutions. The artifact evaluation includes the following experiments:

E1: Generating and pruning candidate graph substitutions. **(Section 6.1, third paragraph)**
E2: End-to-end inference performance comparison among existing DNN frameworks and XFlow on a V100 GPU. **(Figure 8)**
E3: Performance comparison among different approaches to perform multi-batch convolutions in ResNeXt-50. **(Figure 10)**
E4: Performance comparison on BERT using different strategies to optimize graph substitution and data layout. **(Figure 13)**
E5: Verification of generated graph substitutions based on user provided axioms. **(Section 6.5)**
E6: Validation of axioms using redundancy checks and symbolic tests. **(Section 6.5)**

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

- **Run-time environment:** Linux Ubuntu 16.04
- **Hardware:** Amazon AWS p3.2xlarge instance with a NVIDIA V100 GPU.
- **Metrics:** End-to-end inference time.
- **Experiments:** Described in Subsection A.1.
- **How much disk space required (approximately)?:** All source code is less than 1GB. We also provide an Amazon Machine Image (AMI) that with all dependent software pre-installed to minimize the effort to reproduce the experimental results. The AMI is approximately 150GB.
- **How much time is needed to prepare workflow (approximately)?:** Users can directly use the publicly available AMI to set up runtime environment and prepare workflow. Launching an AMI should only take a few minutes.
- **How much time is needed to complete experiments (approximately)?:** Experiments E1, E2, and E3, E4 should each take at most a few minutes to complete. Experiment E5 can take as long as 24 hours, but can also be scaled down if needed (see below).
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License, Version 2.0.
- **Workflow framework used?:** TensorFlow, TVM, ONNX, TensorRT, MetaFlow
- **Archived (provide DOI)?: https://doi.org/10.5281/zenodo.3368852**

A.3 Description

A.3.1 Hardware dependencies

This artifact depends on a NVIDIA GPU. All experiments in this paper were performed on an AWS p3x.2large instance with a V100 GPU.

A.3.2 Software dependencies

This artifact depends on the following software libraries:

- The XFlow runtime is implemented on top of cuDNN and cuBLAS libraries. Our evaluation uses cuDNN 3.7.1 and CUDA 10.0.130.
- CMake (https://cmake.org/) is also required to build the XFlow runtime. We observe that building GPU kernels in XFlow requires a minimum CMake version of 3.8.1.
- ONNX is used for saving and loading pre-built computation graphs. Instructions on installing ONNX is available at https://github.com/onnx/onnx. Our evaluation uses ONNX version 1.5.
- Cython (https://cython.org/) is used by the XFlow Python front-end. Our evaluation uses Cython 0.29.2.
- Protocol buffer (https://developers.google.com/protocol-buffers/) is used to generate and save substitutions (E2, E3, and E4). Some existing source files in XFlow were generated using protocol buffer version 3.6.1.
- TensorFlow, TensorRT, TVM, and MetaFlow are used as baseline DNN frameworks in the evaluation (E2). Detailed instructions on installing TensorFlow is available at A.5. Our evaluation on these baselines uses TensorFlow 2.0, TensorRT 5.0.2.6, TVM 0.6, and MetaFlow 0.1.
- The Z3 Theorem Prover (https://github.com/Z3Prover/z3) is used for verifcation and validation (E5 and E6). Our evaluation uses Z3 4.8.5.

A.4 Installation

We provide the following two approaches to install this the artifact.

A.4.1 Install from AMI

The following Amazon machine image has pre-installed all dependent software for this artifact. A user can directly run all experiments on a p3.2xlarge instance by using the machine image. All code for the artifact is available in the `sosp19ae` folder.

```
AMI Region = N.Virginia, AMI Name = SOSP19AE_26, AMI ID = ami-0465411574ecfb145, Owner = 491037173944
```

A.4.2 Install from source code

- Download archive from https://doi.org/10.5281/zenodo.3368852
- Install cuDNN and cuBLAS from https://developer.nvidia.com/. The experiments were performed with CUDA 10.0.130 and cuDNN 7.4.1.5.
- Install XFlow runtime

```
mkdir build; cd build; cmake ..
sudo make install -j
```

- Install XFlow python front-end

```
cd python; python3 setup.py build_ext -i
export XF_HOME=/path/to/xflow
export PYTHONPATH=$XF_HOME/python:$PYTHONPATH
```

- Install TensorFlow from https://www.tensorflow.org/install. The experiments were performed with TensorFlow version 2.0.
- Install TensorRT from https://developer.nvidia.com/tensorrt. The experiments were performed with TensorRT 5.0.2.6.
- Install Z3, using the instructions at https://github.com/Z3Prover/z3. Make sure the Z3 Python interface is installed, and that the `z3` module can be imported from Python.

A.5 Install TensorFlow

The TensorFlow runtime can be installed following the instructions at https://www.tensorflow.org/install/. The experiments in this paper were done with TensorFlow version 2.0. Note that XLA support is not linked by default in older versions of TensorFlow. If you would like to use an older version with XLA, you must compile from source. Instructions can be found at https://www.tensorflow.org/install/source.

A.6 Experiment workflow

The following experiments are included in this artifact. All DNN benchmarks use synthetic input data in GPU device memory to remove the side effects of data transfers between CPU and GPU.

A.6.1 Graph substitution generator (E1)

The source code for the generator is available at `src/generator/generator.cc`. The following command builds and executes the generator.

```
~/sosp19ae$ cd src/generator; ./compile.sh; time ./generator
```

It takes 5-10 minutes to generate all candidate substitutions (there are roughly 700 graph substitutions, which is more than in the submitted version due to new operators added). All generated substitutions are stored in a protocol buffer file available at `src/generator/graph_subst.pb`.

A.6.2 End-to-end inference performance (E2)

MetaFlow and XFlow. This experiment reproduces Figure 8 in the paper. The following command line measures the inference latency of the optimized computation graphs discovered by MetaFlow and XFlow. The five DNN benchmarks are available in the `examples` folder.

```
~/sosp19ae$ python3 examples/model.py
```

Figure 1 shows an example output of the script.

```
MetaFlow w/ cuDNN: end-to-end inference time =
1.15155971 ms (average of 100 runs)
        ...
# Analysis on the MetaFlow optimized graph
        ...
XFlow w/ cuDNN: end-to-end inference time =
0.51329023 ms (average of 100 runs)
        ...
# Analysis on the XFlow optimized graph
        ...
```

Figure 1. An example output of `python examples/bert.py`.

```
Num. Convs Per Grop = 1
    measure[Conv2D]: i(1 512 28 28) w(512 512 3 3) s(1 1) p(1 1) cost(0.2690)
Num. Convs Per Grop = 2
    measure[Conv2D]: i(1 512 28 28) w(512 256 3 3) s(1 1) p(1 1) cost(0.1496)
Num. Convs Per Grop = 4
    measure[Conv2D]: i(1 512 28 28) w(512 128 3 3) s(1 1) p(1 1) cost(0.1244)
Num. Convs Per Grop = 8
    measure[Conv2D]: i(1 512 28 28) w(512 64 3 3) s(1 1) p(1 1) cost(0.1119)
Num. Convs Per Grop = 16
    measure[Conv2D]: i(1 512 28 28) w(512 32 3 3) s(1 1) p(1 1) cost(0.1667)
Num. Convs Per Grop = 32
    measure[Conv2D]: i(1 512 28 28) w(512 16 3 3) s(1 1) p(1 1) cost(0.3039)
```

Figure 2. An example output of `python examples/eval`$_{g}roups.py$.

TensorFlow and TensorFlow XLA. The TensorFlow implementation of the five benchmarks are available in the `tensorflow_py` folder. The following command lines measure the inference latency of a DNN architecture in TensorFlow and TensorFlow XLA, respectively.

```
~/sosp19ae$ python3 tensorflow_py/model.py
~/sosp19ae$ python3 tensorflow_py/model.py --xla
```

A.6.3 Multi-branch convolutions (E3)

This experiment reproduces Figure 10 in the paper. The following command line measures the run time of a multi-branch convolution by assigning different numbers of convolutions in each group.

```
~/sosp19ae$ python3 examples/eval_groups.py
```

Figure 2 shows an example output of the script. The `cost` values indicate the average run time (in milliseconds) of the multi-branch convolution over 100 runs.

A.6.4 Different strategies for joint optimizations (E4)

This experiment reproduces Figure 13 in the paper. The following command line measures the end-to-end inference performance on BERT using different strategies to optimize graph substitution and data layout.

```
~/sosp19ae$ python3 examples/eval_joint.py
```

Figure 3 shows an example output of the script.

A.6.5 Verification of substitutions (E5)

After running the generator, the generated graph substitutions are saved in the `graph_subst.pb` file. To verify them using the axioms, run:

```
~/sosp19ae$ time python2.7 -u verify.py src/generator/graph_subst.pb
```

```
Measuring the performance of graph substitution optimizations
XFlow: end-to-end inference time = 3.6745831966400146ms

Measuring the performance of data layout optimizations
XFlow: end-to-end inference time = 3.1691980361938477ms

Measuring the performance of sequential optimizations
XFlow: end-to-end inference time = 3.156638135910034ms

Measuring the performance of joint optimizations
XFlow: end-to-end inference time = 2.7719781398773193ms
```

Figure 3. An example output of `python examples/eval_joint.py`.

This should take about two minutes to verify all graph substitutions from the axioms (there are roughly 700 graph substitutions, which is more than in the submitted version due to new operators added). While most transformations are proven in less than a second, it is normal for some transformations to take several seconds to be verified.

You may also want to try to comment out one of the axioms in `verify.py`. If you do this and run the above command, some substitution would no longer be implied by the axioms, which will cause the script to hang trying to prove it from the axioms. The script will keep trying to prove it until it is interrupted. In practice, if a substitution cannot be proved in a short time it most likely means that it is not implied by the axioms.

A.6.6 Validation of axioms (E6)

As explained in Sections 3 and 6.5, we use several techniques to validate the operator properties, or axioms, which are provided by the user. These techniques are all implemented in `validate_axioms.py`, which imports the axiom list from `verify.py` and runs the validation tests.

The two validation checks are (1) redundancy checks, and (2) symbolic verification for tensors of small size. Running `validate_axioms.py` performs both. (If needed the script can be edited to enable or disable each test by changing the `if True` to `if False` in lines 556 and 584.) The redundancy checks take about 10 minutes, and then the symbolic verification can take up to 10 hours (using 8 cores), since it must verify almost a million combinations of tensor sizes and parameter values to cover all tensors up to size $4 \times 4 \times 4 \times 4$. Note that these numbers differ from the submitted version, since the artifact version includes support for additional operators, which required a few more axioms, bringing the total number of axioms to 42. The symbolic verification test is parallel, and utilizes all cores of the machine.

To run the `validate_axioms.py`, a useful command line is:

```
~/sosp19ae$ nohup time python2.7 -u validate_axioms.py &> validate_axioms.out < /dev/null &
```

The output can then be observed in `validate_axioms.out`.

The experiment can also be scaled down to take only a few minutes by changing the `N` variable in `verify.py` (line 90). It's default value is `[1,2,3,4]`, which causes the validation tests to try all combinations of these sizes as a tensor size along each dimension, resulting in almost one million combinations. By changing the value to e.g. `[3,4]`, the number of combinations is greatly reduced, and the experiment can be run in several minutes.

To see the value of the validation tests, one can uncomment either of the wrong axioms commented out in `verify.py` (lines 442-472). These are incorrect axioms that we actually explored during the development process, and were caught by the validation framework. The first axiom, asserting that concatenation is associative, is incorrect in our system because of the split tree (i.e., different associativity results in a different split tree). If this axiom is turned on, many other axioms become redundant, since the wrong axiom, together with the correct axioms relating split and concat, actually imply that any two tensors are equal. Note that when an axiom is found redundant, the script prints the other axioms that entail it and aborts.

The wrong axiom about grouped convolution is actually correct if x and z have the same shape and so do y and w, but it is violated for some shapes that result in valid convolution applications. To see this, uncomment the axiom, set `N = [1,3]` in `verify.py`, and then run the validation tests as above.