PyTorch/Caffe2 Operator Micro-benchmarks

This benchmark suite provides a systemic way to measure the performance of operators for a wide range of inputs. The generated benchmark data fully characterized the performance of an operator in terms of execution time and the efficiency of the PyTorch/Caffe2 frameworks used.

Features

Key Features:

1. Language used: Python
2. Supported Frameworks: PyTorch and Caffe2
3. Supported PyTorch mode: eager and JIT
4. Input shapes: user-defined shapes, randomly generated shapes

Getting Started

Initial Setup

The instruction below installs a cpp_extension for PyTorch and it is required to run the benchmark suite.

```
$ cd pt_extension
$ python setup.py install
```

How to run the benchmarks:

Run `torch.add` benchmark:

```
$ cd pytorch/benchmarks/operator_benchmark
$ python -m pt.add_test --omp_num_threads 1 --mkl_num_threads 1
```

Note: we set the number of OpenMP and MKL threads both to 1. If you want to benchmark operators with multithreading (intra-op parallelism), use the `--omp_num_threads` and `--mkl_num_threads` flags.

List all the supported tests:

```
$ python -m pt.add_test --list_tests
```

Filter and run a test (use `add_M8_N16_K32` as an example):

```
$ python -m pt.add_test --test_name add_K32_M8_N1
--omp_num_threads 1 --mkl_num_threads 1
```

Run all the supported benchmarks:

```
$ python -m benchmark_all_test
```

Code to support `torch.add` in the benchmark

The following example shows the code to support `torch.add` with 27 different tests. In the subpages of this wiki, we'll step through the complete flow of adding PyTorch and Caffe2 operators to the benchmark suite. Existing benchmarks for operators are in `pt` and `c2` directories and we highly recommend putting your new operators in those locations.

```
add_short_configs = op_bench.cross_product_configs(
    M=[8, 64, 128],
    N=range(2, 10, 3),
    K=[2 ** x for x in range(0, 3)],
    tags=["short"]
)

class AddBenchmark(op_bench.TorchBenchmarkBase):
    def init(self, M, N, K, device):
        self.inputs = {
            "input_one": torch.rand(M, N, K, device=device, requires_grad=self.auto_set()),
            "input_two": torch.rand(M, N, K, device=device, requires_grad=self.auto_set())
        }
        self.set_module_name("add")

    def forward(self, input_one, input_two):
        return torch.add(input_one, input_two)

op_bench.generate_pt_test(add_short_configs, AddBenchmark)
```

Output and Command Line Control of the Benchmark

The output is intended to be a human readable format. Here is an example output for `torch.add`:

```
# ----------------------------------------
# PyTorch/Caffe2 Operator Micro-benchmarks
# ----------------------------------------
# Tag : short

# Benchmarking PyTorch: add
# Mode: Eager
# Name: add_M8_N16_K32
# Input: M: 8, N: 16, K: 32
Forward Execution Time (us) : 6.651

# Benchmarking PyTorch: add
# Mode: Eager
# Name: add_M16_N16_K64
# Input: M: 16, N: 16, K: 64
Forward Execution Time (us) : 11.976

# Benchmarking PyTorch: add
# Mode: Eager
# Name: add_M64_N64_K128
# Input: M: 64, N: 64, K: 128
Forward Execution Time (us) : 222.370
```

At a high level, the output includes the execution time of `torch.add` with three different inputs. Let's look at each line in detail:

1. `Tag: short` tags a group of inputs. For each operator, you could be interested in a large number of inputs, but you may not always want to run all the inputs. `Tag` allows you to only run some of the inputs. Most of the inputs to operators being supported in the benchmark are grouped using two tags. One group is tagged with `short` which stores some commonly used shapes. The other group is tagged with `long` which stores many random inputs to have better coverage compared with `short`.

2. `Benchmarking PyTorch: Add` shows name of the operator being benchmarked.

3. `Mode: Eager` shows that PyTorch eager mode is here.

4. `Name: add_M8_N16_K32` is the name of the test and it can be used to filter tests.

5. `Input: M: 8, N: 16, K: 32` shows inputs to the operator.

6. `Forward Execution Time (us) : 6.651` reports the execution time of an operator in microseconds.

Command-Line Control

You can control all the aspects of the benchmark suite through the command-line. Please find details of those arguments by running the following command or look into `benchmark_runner.py`.

```
$ python benchmark_runner.py --help
```

Run all the supported benchmarks:

```
$ python -m benchmark_all_test --omp_num_threads 1 --mkl_num_threads 1
```

List all the supported operators:

```
$ python -m benchmark_all_test --list_ops
```

List all the supported tests:

```
$ python -m benchmark_all_test --list_tests
```

Filter and run an operator (use add as an example):

```
$ python -m benchmark_all_test --operators add --omp_num_threads 1 --mkl_num_threads 1
```

Note: this filter is based on the operator name rather than the file name.

Run torch.add benchmark with tag ‘long’:

```
$ python -m pt.add_test --tag_filter long
```

Adding New Operators to the Benchmark Suite

In the previous sections, we gave several examples to show how to run the already available operators in the benchmark suite. In the following sections, we’ll step through the complete flow of adding PyTorch and Caffe2 operators to the benchmark suite. Existing benchmarks for operators are in `pt` and `c2` directories and we highly recommend putting your new operators in those directories as well.

Add a New PyTorch Operator

Let’s say you want to measure the execution time of the following operator:

```
C = torch.add(A, B) # Shape of A and B is [M, N, K]
```

The code below shows how to add it to the benchmark suite. Let’s go over the example line by line.

```
import operator_benchmark as op_bench
import torch

add_long_configs = op_bench.cross_product_configs(
    M=[8, 64, 128],
    N=range(2, 10, 3),
    K=[2 ** x for x in range(0, 3)],
    tags=["long"]
)

add_short_configs = op_bench.config_list(
    attr_names=["M", "N", "K"],
    attrs=[
        [8, 16, 32],
        [16, 16, 64],
        [64, 64, 128],
    ],
    tags=["short"],
)

class AddBenchmark(op_bench.TorchBenchmarkBase):
    def init(self, M, N, K, device):
        self.inputs = {
            "input_one": torch.rand(M, N, K, device=device, requires_grad=self.auto_set()),
            "input_two": torch.rand(M, N, K, device=device, requires_grad=self.auto_set())
        }
        self.set_module_name("add")

    def forward(self, input_one, input_two):
        return torch.add(input_one, input_two)

op_bench.generate_pt_test(add_long_configs + add_short_configs, AddBenchmark)

if __name__ == "__main__":
    op_bench.benchmark_runner.main()
```

Part 1. Specify Inputs to Operators For the `torch.add` operator, we would like to make sure it delivers good performance with input tensors which are of small, medium and large sizes. We have introduced two helper functions for users to easily generate a combination of inputs.

```
# Generate list configurations that will be used for benchmark experiments
add_long_configs = op_bench.cross_product_configs(
    M=[8, 64, 128],
    N=range(2, 10, 3),
    K=[2 ** x for x in range(0, 3)],
    tags=["long"]
)

add_short_configs = op_bench.config_list(
    attr_names=["M", "N", "K"],
    attrs=[
        [8, 16, 32],
        [16, 16, 64],
        [64, 64, 128],
    ],
    tags=["short"],
)
```

Let's look at it in detail:

1. `op_bench.config_list` is a helper function which specifies a list of inputs to operators. It takes three parameters which are `attrs_names, attrs, and tags`, all of them are python lists. `attr_names` stores the names of the inputs. `attrs` stores the real value of each input. In this example, three different inputs will be returned which are: `M=8, N=16, K=32; M=16, N=16, K=64; M=64, N=64, K=128`.

2. `op_bench.cross_product_configs` is another helper function to generate a cartesian product of the inputs. Each input is specified in a python list. In this example, the helper method will return a combination of 27 (len(M) * len(N) * len(K)) inputs.

Part 2. Create Tensors and Add Computation After inputs are provided, we now look at adding the computation of an operator. Adding a new operator requires implementing a new `TorchBenchmarkBase` subclass. Every new class is required to implement 2 methods: * `init` is used to create tensors based on the inputs we provided before. In this example, the parameters to `init` are `M, N, and K` which have been specified in the input configuration. `init` also packed all the needed inputs together into a dictionary `self.inputs` which will be provided to `forward` as arguments for running the benchmark. * `forward` includes the operator to be tested and the computation based on the created tensors in `init`. Apart from `self`, the order of the arguments must match the entries specified in `self.inputs`.

The example below shows the code for `torch.add`:

```
# Given one set of M, N, K, the init method creates input tensors based on
# that. The forward method does torch.add calculation on those input tensors.

class AddBenchmark(op_bench.TorchBenchmarkBase):
    def init(self, M, N, K, device):
        # this is the method where you need to create tensors
        # M, N, and K can be in different order, but they must match with
        # names in the configs.
        self.inputs = {
            "input_one": torch.rand(M, N, K, device=device, requires_grad=self.auto_set()),
            "input_two": torch.rand(M, N, K, device=device, requires_grad=self.auto_set())
        }
        self.set_module_name("add")

    def forward(self, input_one, input_two):
        # this is the method to have operator and do computation
        return torch.add(input_one, input_two)
```

Part 3. Register Tests With the Benchmark Suite After we have inputs and the benchmark class, it's time to register them with our benchmark suite. Here is how it looks like:

```
op_bench.generate_pt_test(add_long_configs + add_short_configs, AddBenchmark)
```

`generate_pt_test` takes two parameters which are inputs configs and the benchmark class.

Part 4. Run the Registered Tests To run the benchmark, we use the main method in `benchmark_runner` module.

```
if __name__ == "__main__":
    op_bench.benchmark_runner.main()
```

That's it. You just added a new operator to the benchmark suite!

Add a New Caffe2 Operator

The steps to add a new Caffe2 operator is the same as that for a PyTorch operator. The code below shows how to add Caffe2 `Add` operator:

```
import operator_benchmark as op_bench
from caffe2.python import core

add_long_configs = op_bench.cross_product_configs(
    M=[8, 64, 128],
    N=range(2, 10, 3),
    K=[2 ** x for x in range(0, 3)],
    tags=["long"]
)

add_short_configs = op_bench.config_list(
    attrs=[
        [8, 16, 32],
        [16, 16, 64],
        [64, 64, 128],
    ],
    attr_names=["M", "N", "K"],
    tags=["short"],
)

class AddBenchmark(op_bench.Caffe2BenchmarkBase):

    def init(self, M, N, K):
        self.input_one = self.tensor(M, N, K)
        self.input_two = self.tensor(M, N, K)
        self.output = self.tensor(M, N, K)
        self.set_module_name("add")

    def forward(self):
        op = core.CreateOperator(
            "Add", [self.input_one, self.input_two], self.output, **self.args
        )

        return op
```

```
op_bench.generate_c2_test(add_long_configs + add_short_configs, AddBenchmark)

if __name__ == "__main__":
    op_bench.benchmark_runner.main()
```

There are two things worth mentioning in this code: * `self.tensor` is a helper function which takes shapes and returns a Caffe2 blob. It is designed to make the tensor creation step easier compared to the standard Caffe2 way. * `generate_c2_test` is used to register Caffe2 tests with the benchmark.

Add a List of Operators

In the previous sections, we introduced the steps required to add a single operator to the benchmark suite. There are scenarios where you want to extend the benchmark suite with a list of operators which can share the same inputs. For example, to benchmark `abs` and `acos` operators, you can use the same set of inputs for both.

Let's say we want to benchmark the following operators separately:

```
C = torch.abs(A) # Shape of A [M, N]
C = torch.acos(A) # Shape of A [M, N]
```

The following code shows how to do that:

```
import operator_benchmark as op_bench
import torch

unary_ops_configs = op_bench.config_list(
    attrs=[
        [128, 128],
        [256, 256],
        [1024, 1024],
    ],
    attr_names=["M", "N"],
    tags=["short"]
)

unary_ops_list = op_bench.op_list(
    attr_names=["op_name", "op_func"],
    attrs=[
        ["abs", torch.abs],
        ["acos", torch.acos],
    ],
)

class UnaryOpBenchmark(op_bench.TorchBenchmarkBase):
    def init(self, M, N, device, op_func):
        self.inputs = {
            "input": torch.rand(M, N, device=device)
        }
        self.op_func = op_func

    def forward(self, input):
        return self.op_func(input)

op_bench.generate_pt_tests_from_op_list(unary_ops_list, unary_ops_configs, UnaryOpBenchmark)

if __name__ == "__main__":
    op_bench.benchmark_runner.main()
```

The inputs to those operators are specified using the same method we went over before. So we just skip it here.

Part 1. Specify the List of Operators To add a list of operators to the benchmark suite, we introduce the `op_bench.op_list` method which takes two parameters: * `attrs` stores the name of the operator and the method to do the real calculation. * `attr_names` stores the names of values in attrs.

The example below shows the code to add `torch.abs` and `torch.acos` :

```
unary_ops_list = op_bench.op_list(
    attr_names=["op_name", "op_func"],
    attrs=[
        ["abs", torch.abs],
        ["acos", torch.acos],
    ],
)
```

Part 2. Create Tensors and Add Computation In this example, both operators share the same input so we only need to implement one TorchBenchmakrBase subclass. Every new subclass is required to implement 3 methods: * `init` is used to create tensors and set the operator name and function. In this example, the parameters to `init` are `M`, `N`, and `op_func` which have been specified in the configurations. * `forward` includes the operator to be tested and the computation based on the created tensors in `init`. Apart from `self`, the order of the arguments must match the entries specified in `self.inputs`. Here is the code for `abs` and `acos`:

```
class UnaryOpBenchmark(op_bench.TorchBenchmarkBase):
    def init(self, M, N, device, op_func):
        # The M and N match with the attr_names in the input configuration
        # The op_func matches with the attr_name in the ops configuration
        self.inputs = {
            "input": torch.rand(M, N, device=device)
        }
        self.op_func = op_func

    def forward(self, input):
        return self.op_func(input)
```

Part 3. Register a List of Operators To register multiple operators, we introduced the `generate_pt_tests_from_op_list` function which takes three parameters. First, the list of operators. Second,the configs. Third, the benchmark class. Here is an example:

```
op_bench.generate_pt_tests_from_op_list(unary_ops_list, unary_ops_configs, UnaryOpBenchmark
```

Add Gradient Ops

In this section, we go over the steps to benchmark the backward path of operators. #### For PyTorch Gradient Ops To measure the performance of an operator in its backward path, there are only two changes needed in addition to the steps we covered for the forward path:

1. Specify `requires_grad=True` when creating the tensor. This is a standard PyTorch way of enabling backward path.

2. Use `generate_pt_gradient_test` to register the tests.

The example below shows the relevant code for that:

```
self.input_one = torch.rand(M, N, K, requires_grad=True)
generate_pt_gradient_test(long_configs + short_configs, TorchAddBenchmark)
```

For Caffe2 Gradient Ops To add Caffe2 gradient ops, we need to implement a new backward method in the benchmark class:

```
class AddBenchmark(op_bench.Caffe2BenchmarkBase):

    def init(self, M, N, K):
        self.input_one = self.tensor(M, N, K)
        self.input_two = self.tensor(M, N, K)
        self.input_one_grad = self.tensor(M, N, K)
        self.input_two_grad = self.tensor(M, N, K)
        self.output = self.tensor(M, N, K)
        self.set_module_name("add")

    def forward(self):
        op = core.CreateOperator(
            "Add", [self.input_one, self.input_two], self.output, **self.args
        )
        return op

    def backward(self):
        grad_op = core.CreateOperator(
            "AddGradient",
            [self.output, self.input_one, self.input_two],
            [self.input_one_grad, self.input_two_grad], **self.args
        )

        return grad_op

op_bench.generate_c2_gradient_test(long_configs + short_configs,AddBenchmark)
```

After the class is implemented, we need to register the tests with `generate_c2_gradient_test` function.

This concludes the overview of the operator benchmark suite.