MXNet Operator Benchmarks

WIP staging repo - https://github.com/sandeep-krishnamurthy/dl-operator-benchmark

Link to dev list discussion

https://lists.apache.org/thread.html/a7a2b27ff2bd3069eecac1b84d6a11e5a1a7845c3d6e90e9b84a1654@%3Cdev.mxnet.apache.org%3E

Feature Shepherd

Lin Yuan (https://github.com/apeforest)

Problem Statement

A deep learning framework like MXNet supports 100s of operators (~250). Benchmarking and profiling a standard neural network and use-case, like ResNet-50 based image classification, is not fully sufficient and does not guarantee to maintain the health and performance of all the supported operators under different settings (Hardware, Accelerator, Data etc...). We need to have an easy to use utility to run benchmarks and profile each operator individually. Having an operator level benchmarks will help us in - fine grained understanding of performance of operators under different settings (Hardware, Accelerator, Data etc...), automated CI/CD performance tests, plan performance optimization tasks and more. In this document, we present an utility for MXNet operator benchmark and profiling.

Motivation

A deep learning framework like MXNet supports 100s of operators (~250). Some operators are used as a layer in the neural network (ex: Conv2D), some operators work in combination to form a layer in the neural network (ex: dot, sum => Dense), and many more just used independently outside a neural network (ex: tensor creation/shape change/indexing, logical operators) mostly for data processing and tensor manipulation.

An operator is highly heterogeneous w.r.t supported precision (fp32, fp64, Int64 etc.), accelerators (mkldnn, cuda, cudnn, mxnet native only), different behaviors based on data (ex: broadcast sum behavior on a large square (1024, 1024) tensor is different than on a skewed tensor (10, 10000) and more). Below, we see few areas why we believe operator benchmarks are useful:

1. Users use a lot more operators that are not part of a standard network like ResNet. Example: Tensor manipulation operators like mean, max, topk, argmax, sort etc.
2. A standard Network Architecture like ResNet-50 is made up of many operators Ex: Convolution2D, Softmax, Dense, Pooling etc... Observing only the end to end performance can hide individual operator regressions for long time.
3. We need to know on different hardware infrastructure (Ex: CPU with MKLDNN, GPU with NVIDIA CUDA and cuDNN) how different operators perform. With these details, we can plan the optimization work at operator level, which could exponentially boost the end to end performance.
4. Operator behavior varies based on different data load:
    a. For example, MXNet's reduction operations work seamlessly with balanced tensor like (1024, 1024), however, performance behavior changes when the input tensor is skewed (1024, 10). Similar observations can be made when comparing Int32 v/s Int64 indexing of Tensor.
    b. See this issue - #14725 which talks about performance regression in FC layer backward pass with CUDA 10 based on input tensor shape - https://github.com/apache/incubator-mxnet/issues/14725#issuecomment-486016229
5. You want to have nightly performance tests across all operators in a deep learning framework to catch regressions early.
6. We can integrate this framework with a CI/CD system to run per operator performance tests for PRs. Ex: When a PR modifies the kernel of TransposeConv2D, we can run benchmarks of TransposeConv2D operator to verify performance.
7. Useful insights can be derived to plan for operator performance improvements. Example - Argmax is much slower compared to max operator on a GPU. This is an area that we can work on to improve the performance of Argmax operator. https://github.com/apache/incubator-mxnet/issues/11337
8. Benchmarking operator performance in MXNet comparing with other Deep Learning frameworks such as PyTorch. (Not in current scope)

Hence, in this utility, we will build the functionality to allow users and developers of deep learning frameworks to easily run benchmarks for individual operators across varying settings.

Requirements

1. Benchmarks for Apache MXNet operators.
2. Individual operator benchmarks to capture - time for operator execution (speed), memory usage.
3. Fine grained individual operator benchmarks to capture - time for forward pass, time for backward pass and both.
4. Ability to run operator benchmarks with default inputs, randomly generated inputs or customize with user specific inputs.
5. Ability to run operator benchmarks on CPU/GPU with different flavors of MXNet (mxnet-mkl, mxnet-cu90mkl etc.)
6. Benchmarks for operators with varying inputs to uncover any performance issues due to skewed input data. Ex: Measuring operator performance on small input tensors, large input tensors along with average normally used tensor sizes.
7. Ability to run one, group or all operator benchmarks.
8. Ability to extract results in multiple usable format - Python Dictionary, JSON, CSV, MD
9. Statistics:
    a. Mean, Median, P50, P90, P99
10. Reproducible tests
11. Common combination (Fused) of operators. Conv + Relu, Conv + BatchNorm. (Not in current scope)

Design Tenets

1. Defaults => Common use cases should be extremely easy, customized complex use cases should be possible.
    a. Example: I should be able to run Add operator benchmarks without specifying any inputs and library should provide benchmarks on valid default inputs. At the same time, as a power user, I should be able to provide my own inputs such as Tensor Shapes and context to run the benchmarks.
2. Minimum Learning Curve => Keep APIs same or close to native NDArray / Gluon Operators being benchmarked.
    a. Example: If I am doing benchmarks on nd.add(lhs, rhs) operator, interface in the benchmark utility should be similar with zero learning curve.
3. Modular and Reusable
4. For a programmer or an automated system
    a. Example: Developer using the library or integration with CI/CD

Proposed Approach

1. Provide a generic utility for executing an operator benchmarks and performance tests.
    a. This is responsible to creating input tensors of required shape on a given dtype, context.
    b. Execute the provided operator - forward or forward + backward.
    c. This generic utility will be integrated with MXNet profiler.
    d. Captures the profile output from MXNet profiler - time, memory.
    e. Return a dictionary of results.
2. Input for the performance tests will be a key/value config.

Below is an example of performance runs for operators. It uses a base utility `run_performance_test`.

```
"""
MXNet operator performance benchmarks.

NOTE:
1. You can pass list of input dictionary to run benchmarks for an operator with different input configuration.
2. Results are dictionary of time, memory for the benchmark runs.
"""

# Run performance test for Add operator
results = run_performance_test(F=mx.nd.add, ctx=mx.cpu(), warmup=10, runs=50, inputs=[{"lhs": (1024, 1024),

"rhs": (1024, 1024),

"initializer": nd.normal,

"run_backward": True,

"dtype": "float32"}])

# Run performance test for Conv2D operator
results += run_performance_test(F=nn.gluon.Conv2D, ctx=mx.cpu(), warmup=10, runs=50, inputs = [{"data": (32, 3,
256, 256),

"data_initializer": nd.normal,

"channels": 64,

"kernel_size": (3, 3),

"strides": (1, 1),

"padding": (0, 0),

"dilation": (1, 1),

"layout": "NCHW",

"activation": None,

"run_backward": True,

"dtype": "float32"]}
```

How does the backend profiling utility code looks like?

Below we take an example of profiling Add operator.

```
# Configurations
warmup = 25
runs = 50
run_backward = True

# Operator to benchmark
F = mx.nd.add

# Prepare data for the operator
lhs = mx.nd.ones(shape=(1024, 1024))
rhs = mx.nd.ones(shape=(1024, 1024))
lhs.attach_grad()
rhs.attach_grad()
mx.nd.waitall()

# Warmup
print("Warming up....")
for _ in range(warmup):
    with mx.autograd.record():
        res = mx.nd.add(lhs, rhs)
    res.backward()
    mx.nd.waitall()
print("Done warming up....")

# Run Performance Runs
print("Running performance runs....")
profiler.set_config(profile_all=True, aggregate_stats=True)
# Start Profiler
profiler.set_state('run')
for _ in range(runs):
    with mx.autograd.record():
        res = mx.nd.add(lhs1, rhs1)
    res.backward()
    mx.nd.waitall()

# Stop Profiler
profiler.set_state('stop')

# Fetch Results from Profiler
# We will add a new API in Profiler - profiler.get_summary(reset=True)
# profiler.get_summary() => Return a JSON string representing the output as shown below.
#                        => Resets all the counter in the current profiler.

print("Done Running performance runs....")
print(profiler.dumps(reset=True))
```

Pros

1. No need to write 1 class per operator to set up a performance test. Whenever a new operator is created, developer needs to add a `run_performance_test(..)` line with a list of inputs to run performance tests. A generic utility will handle the execution.
2. Less code, easy to maintain.
3. More control for users - default inputs, random inputs, specific user defined inputs.
4. Deterministic and better suited for performance benchmarks, reproducibility and CI integration.
5. More accurate benchmark results - Time and Memory because we use MXNet profiler.
6. With Python interface:
    a. Easy to maintain and develop.
    b. Reflects the performance as seen by the users. (Majority users using Python interface)
    c. Fastest way to get performance tests in place. We do not have any tests in place as of today.

Cons

1. Different operator will have different input names. For example, see above, add operator requires tensors with name lhs, rhs. However, Conv2D operator requires a tensor with data. The base performance executor utility will need to understand it and create tensors appropriately i.e., If it is one single executor, generalization across operator performance may make logic complex to manage.
2. Not easily extensible:
    a. Hard to integrated with property based testing libraries like Hypothesis, to randomly generate test cases with different tensor shapes.

Addition of new Module

We propose to add this utility as a new module (opperf) under incubator-mxnet/benchmark as "incubator-mxnet/benchmark/opperf". Note that, this does not generate any user facing APIs, this is a utility under incubator-mxnet/benchmark folder for general use by community.

Addition of new API

We propose to add a new API to MXNet Profiler for easily fetching operator profile for processing programmatically.

1) mxnet.profiler.get_summary(reset=False)

Current Behavior:

Users can either use `mxnet.profiler.dump()` to output the profiler as a JSON file. Or, use `**mxnet.profiler.dumps(reset=False)**` API to print the summary on console.

Suggested Addition:

In order to enable easy programmatic usage of MXNet profiler output, we propose to introduce a new API to return the summary as JSON string. This enables users to run profiler, get summary output, perform analysis programmatically.

```
mxnet.profiler.get_summary(reset=False)
        """Gets current profiler summary as a JSON string. If reset is True, resets all the aggregate
statistics collected up to this point i.e., it clears all    the profiler counters.

        Parameters:
    -----------
    reset: boolean, If True, resets all profiler statistics collected up to this point.
    """
```

Output:

We can visualize the output of this API as a JSON representation of the output from `**mxnet.profiler.dumps(reset=False)**` API as shown below.

However, please note that, below, Memory profile output is not the total bytes allocated. Current output from dumps is providing number of memory allocation calls made.

In the new suggested API, we will be adding additional Summary - Memory => Total Bytes Allocated (Per Device).

```
Profile Statistics.
        Note that counter items are counter values and not time units.
Device Storage
=================
Name                          Total Count        Time (ms)    Min Time (ms)    Max Time (ms)    Avg Time (ms)
----                          -----------        ---------    -------------    -------------    -------------
Memory: cpu/0                         200     1258291.2500      423624.7188     1258291.2500      417333.2500

MXNET_C_API
=================
Name                          Total Count        Time (ms)    Min Time (ms)    Max Time (ms)    Avg Time (ms)
----                          -----------        ---------    -------------    -------------    -------------
MXAutogradBackwardEx                  100          24.0700           0.1940           0.7670           0.2407
MXNDArrayFree                         100           2.4180           0.0170           0.1150           0.0242
MXNet C API Calls                     800           1.2000           0.4010           1.2000           0.3995
MXImperativeInvokeEx                  100           9.3600           0.0620           0.2930           0.0936
MXNDArrayWaitAll                      100        1520.0610          13.9160          30.4470          15.2006
MXAutogradSetIsTraining               200           0.0450           0.0000           0.0010           0.0002
MXAutogradSetIsRecording              200           0.1100           0.0000           0.0130           0.0005
MXNet C API Concurrency              1600           0.0000           0.0000           0.0010           0.0005

operator
=================
Name                          Total Count        Time (ms)    Min Time (ms)    Max Time (ms)    Avg Time (ms)
----                          -----------        ---------    -------------    -------------    -------------
DeleteVariable                        391           3.5560           0.0040           0.0520           0.0091
_backward_broadcast_add               200        1055.1880           4.7720          11.9610           5.2759
SetValueOp                            200        1263.6689           5.8420          11.1990           6.3183
broadcast_add                         200         771.6760           3.5140           7.0960           3.8584
```

API / User Experience

We can define 2 types of users of the library and describe API interface for each of these users.

1. General User, Automated Nightly tests
    a. Run benchmarks on all the operators or on specific categories of operators. Use default inputs provided by the library.
2. Power User, PR validation tests
    a. Run benchmark with customized Inputs

Use Case 1 - Run benchmarks for all the operators

A driver to run all the MXNet operators (NDArray and Gluon) benchmarks with default inputs and saves the final result as JSON in the provided file.

```
python incubator-mxnet/benchmark/opperf/run_all_mxnet_operator_benchmarks.py --output-format json --output-file
mxnet_operator_benchmark_results.json
```

Other Driver Script CLI Options:

1. **output-format** : `json` or `md` for markdown file output or `csv`.
2. **ctx** : By default, `cpu` on CPU machine, `gpu(0)` on GPU machine. You can override and set the global context for all operator benchmarks. Example: `--ctx gpu(2)`.
3. **dtype** : By default, `float32`. You can override and set the global dtype for all operator benchmarks. Example: `--dtype float64`.

Output for the above benchmark run, on a CPU machine, would look something like below:

```
{
    "MX_Multiply_Forward_Backward_Time": 0.025911798477172853,
    "MX_Gluon_Imperative_RNN_Forward_Backward_Time": 0.011011338233947754,
    "MX_Gluon_Imperative_MaxPool2D_Forward_Backward_Time": 0.1580966854095459,
    "MX_Gluon_Imperative_Conv1D_Forward_Backward_Time": 0.03413449287414551,
    "MX_Ones_Forward_Time": 0.002405076026916504,
    "MX_Modulo_Forward_Backward_Time": 0.049943366050720216,
    "MX_Subtract_Forward_Backward_Time": 0.01635995864868164,
    "MX_ArgMin_Forward_Backward_Time": 0.01545732021331787,
    "MX_Logical_Xor_Forward_Backward_Time": 0.018084139823913575,
    "MX_Zeros_Like_Forward_Time": 0.0027973604202270507,
    "MX_Inplace_Multiply_Forward_Time": 0.005555639266967774,
    "MX_ArgSort_Forward_Time": 0.13972537994384765,
    "MX_Arange_Forward_Time": 0.00010946273803710938,
........
........
}
```

Use Case 2 - Power user - Run benchmarks for specific operator

As a power user, let us assume, you want to run benchmarks on Add operator with on a `float64` tensor instead of a default `float32`.
NOTE: Similarly, you could also specify the input tensors to use for benchmarking.

Use Case 2.1 - Customize Inputs for Operators

```
results = run_performance_test(F=mx.nd.add, ctx=mx.cpu(), warmup=10, runs=50, inputs=[{"lhs": (1024, 1024),

"rhs": (1024, 1024),

"initializer": nd.normal,

"run_backward": True,

"dtype": "float64"}])
```

Output for the above benchmark run, on a CPU machine, would look something like below:

```
MX_Add_Forward_Backward_Time - 0.025401 seconds
```

Use Case 3 - Nightly CI Tests

1. We will maintain a JSON file of expected performance for each operator under "incubator-mxnet/benchmark/opperf".
2. These expected results are captured on different configuration such as - FP32/64/16, MKL, No MKL, CUDA10, instances (c5.16x, p3.8x).
3. Runs all the operator performance runs and gets the results JSON.
4. Compares with the expected results +/- % threshold.

Development Plan / Milestones

Phase 1

1. ~150 most commonly used operators will be tested on CPU(with and without MKL), GPU, FP32, FP64. See Appendix 1 for list of operators.
2. Operators will be tested with NDArray and Gluon interface only i.e., symbol interface is not used for testing owing to plans of deprecation.
3. Python interface is used along with MXNet profiler.
4. Time and Memory usage are measured to start with.
5. Statistics - Mean of the metric.

Phase 2

1. Cover remaining operators left out from Phase 1.
2. Add more statistics - p50, p90, p99, min, max.

Phase 3

1. Explore and have CPP performance tests for most commonly used operators. This will give the true measurements compared to using Python Interface.
2. Integrate with property based testing libraries like Hypothesis, to randomly generate test cases with different tensor shapes and inputs.

Current Status

See this repo for more details - https://github.com/sandeep-krishnamurthy/dl-operator-benchmark

1. 134 operators are supported:
    a. All Gluon Layers - Activation, Loss, Normalization, Basic like Dense, Convolutions, Recurrent (RNN, LSTM, GRU)
    b. NDArray operators like creation, random sampling, arithmetic, logical, comparison etc...
2. Able to run individual operator benchmarks or use high level drivers to run all tests.
3. Able to generate results as JSON.
4. Timing metric - forward only, forward+backward operation.

Alternate Solutions

Alternate Solution 1 - Use Python Classes for each Operator instead of Config

Approach

1. This benchmark utility will be built on top of MXNet's ND and Gluon interface.
2. For each operator in ND and Gluon Block, there will be a corresponding Benchmarking operator in the library with a list of default inputs, functionality to process results. See below example for Add operator benchmarks.
3. High-level drivers are provided to run operator benchmarks in bulk. Example: run_all_mxnet_operator_benchmarks(), run_all_arithmetic_operations_benchmarks() etc.
4. Results can be generated as a python dictionary/JSON/CSV for upstream system (Ex: CI, Automated Performance Monitoring System) consumption.

```
class Add(MXNetOperatorBenchmarkBase):
    """Helps to Benchmark Tensor Add operation.

    By default benchmark both forward and backward element_wise tensor addition
    of 1024*1024 tensor of precision - 'float32'.

    """

    def __init__(self, ctx=mx.cpu(), warmup=10, runs=50, inputs=None):
        # Set the default Inputs
        default_parameters = {"lhs": (1024, 1024),
                              "rhs": (1024, 1024),
                              "initializer": nd.normal,
                              "run_backward": True,
                              "dtype": "float32"}

        super().__init__(ctx=ctx, warmup=warmup, runs=runs, default_parameters=default_parameters,
                         custom_parameters=inputs)

        self.lhs = get_mx_ndarray(ctx=self.ctx, in_tensor=self.inputs["lhs"],
                                  dtype=self.inputs["dtype"],
                                  initializer=self.inputs["initializer"],
                                  attach_grad=self.inputs["run_backward"])
        self.rhs = get_mx_ndarray(ctx=self.ctx, in_tensor=self.inputs["rhs"],
                                  dtype=self.inputs["dtype"],
                                  initializer=self.inputs["initializer"],
                                  attach_grad=self.inputs["run_backward"])

    def run_benchmark(self):
        # Warm up, ignore execution time value
        _, _ = nd_forward_backward_and_time(F=nd.add, runs=self.warmup, lhs=self.lhs, rhs=self.rhs)
        # Run Benchmarks
        exe_time, _ = nd_forward_backward_and_time(F=nd.add, runs=self.runs, lhs=self.lhs, rhs=self.rhs)

        self.results["MX_Add_Forward_Backward_Time"] = exe_time / self.runs
```

API / User Experience

We can define 2 types of users of the library and describe API interface for each of these users.

1. General User, Automated Nightly tests
    a. Run benchmarks on all the operators or on specific categories of operators. Use default inputs provided by the library.
2. Power User, PR validation tests
    a. Run benchmark

USE CASE 1 - Run benchmarks for all the operators

A driver to run all the MXNet operators (NDArray and Gluon) benchmarks with default inputs and saves the final result as JSON in the provided file.

```
python dl-operator-benchmark/run_all_mxnet_operator_benchmarks.py --output-format json --output-file
mxnet_operator_benchmark_results.json
```

Other Driver Script CLI Options:

1. **output-format** : `json` or `md` for markdown file output or `csv`.
2. **ctx** : By default, `cpu` on CPU machine, `gpu(0)` on GPU machine. You can override and set the global context for all operator benchmarks. Example: `--ctx gpu(2)`.
3. **dtype** : By default, `float32`. You can override and set the global dtype for all operator benchmarks. Example: `--dtype float64`.

USE CASE 2 - Run benchmarks for all the operators in a specific category

For example, you want to run benchmarks for all `NDArray Arithmetic Operators`, the library will be providing drivers to easily run benchmarks on operators of specific categories.

```
from mxnet_benchmarks.nd import run_all_arithmetic_operations_benchmarks
# Run all Arithmetic operations benchmarks with default input values
run_all_arithmetic_operations_benchmarks()
```

Output for the above benchmark run, on a CPU machine, would look something like below:

```
MX_Add_Forward_Backward_Time - 0.015201 seconds
MX_Multiply_Forward_Backward_Time - 0.021678 seconds
MX_Subtract_Forward_Backward_Time - 0.016154 seconds
MX_Divide_Forward_Backward_Time - 0.024327 seconds
MX_Modulo_Forward_Backward_Time - 0.045726 seconds
MX_Power_Forward_Backward_Time - 0.077152 seconds
MX_Negative_Forward_Backward_Time - 0.014472 seconds
MX_Inplace_Add_Forward_Time - 0.003824 seconds
MX_Inplace_Subtract_Forward_Time - 0.004137 seconds
MX_Inplace_Multiply_Forward_Time - 0.006589 seconds
MX_Inplace_Division_Forward_Time - 0.003869 seconds
MX_Inplace_Modulo_Forward_Time - 0.018180 seconds
```

Use Case 3 - Power user - Run benchmarks for specific operator

As a power user, you want to run benchmarks for `nd.add` operator in MXNet, you just run the following python script.
Note that, we maintain same name and spec as the underlying MXNet operator. For example - to benchmark nd.add, we can use mxnet_benchmarks.nd.Add().

Use CASE 3.1 - Default Inputs for Operators

```
from mxnet_benchmarks.nd import Add
# Run all Arithmetic operations benchmarks with default input values
add_benchmark = Add()
add_benchmark.run_benchmark()
add_benchmark.print_benchmark_results()
```

Output for the above benchmark run, on a CPU machine, would look something like below:

```
MX_Add_Forward_Backward_Time - 0.015201 seconds
```

USE CASE 3.2 - Customize Inputs for Operators

As a power user, let us assume, you want to run benchmarks on a `float64` tensor instead of a default `float32`.
NOTE: Similarly, you could also specify the input tensors to use for benchmarking.

```
from mxnet_benchmarks.nd import Add
# Run all Arithmetic operations benchmarks with default input values
add_benchmark = Add(inputs={"dtype": "float64"})
add_benchmark.run_benchmark()
add_benchmark.print_benchmark_results()
```

Output for the above benchmark run, on a CPU machine, would look something like below:

```
MX_Add_Forward_Backward_Time - 0.025405 seconds
```

NOTE: You can print the input parameters used for a benchmark as shown below.

```
from mxnet_benchmarks.nd import Add
# Run all Arithmetic operations benchmarks with default input values
add_benchmark = Add(inputs={"dtype": "float64"})print(add_benchmark.inputs)
```

Output

```
{'lhs': (1024, 1024), 'rhs': (1024, 1024), 'initializer': <function normal at 0x117b607b8>, 'run_backward':
True, 'dtype': 'float64'}
```

Pros

1. More control for users - default inputs, random inputs, specific user defined inputs.
2. Deterministic and better suited for performance benchmarks, reproducibility and CI integration.
3. With Python interface:
    a. Easy to maintain and develop.
    b. Reflects the performance as seen by the users. (Majority users using Python interface)
    c. Fastest way to get performance tests in place. We do not have any tests in place as of today.
    d. Ability to run and compare benchmarks from other deep learning frameworks.
4. Extensible:
    a. Can be integrated with property based testing libraries like Hypothesis, to randomly generate test cases with different tensor shapes.

Cons

1. Need to write base tests for every new operator. If a new operator is added to MXNet, then a new performance test class for the operator needs to be added in this library with default inputs for that new operator to run performance tests.
2. It is ideal to capture performance close to Kernel. Call from Python operator APIs may hide performance regression when operator computation is small.

Alternate Solution 2 - Autogenerate test with Property Based Testing Technique

(Credits - Thanks to Pedro Larroy for this suggestion)

Approach

1. Automatically query all operators registered with MXNet engine.
2. Infer the inputs and outputs for the operators.
3. Use property based testing technique and library such as Hypothesis to generate random inputs and run the tests.

Pros

1. Any new operator added to MXNet, will be automatically queried. Hence, no need to write tests explicitly for every operator.
2. Inputs are randomly generated. Hence, better suited to capture performance regression on corner cases.

Cons

1. Non deterministic inputs. Hence, better suitable for functionality testing. It will be hard to use this technique for performance tests.
2. Still requires us to write many custom strategies or conditional property files. Example:
    a. For testing Add operator, we need to set conditions on input to generate same shapes or broadcastable shapes for lhs and rhs.
    b. For Convolution operator, we need to match Kernel, Padding and other parameter shapes appropriately.
3. Querying operators and inferring the input conditions may be hard and complex logic.
    a. Example: add is an operator, that takes 2 input tensors - lhs, rhs. Now we need to infer that lhs and rhs tensor should of same size or broadcastable. Logic to handle such conditions may soon become complex enough to not give us advantage of auto generated operator benchmarks.
    b. MXNet currently do no support a standard way of querying the registered operators. It would be ideal if MXNet can expose NNVM APIs for querying registered operators and expected inputs, outputs, types and more.
4. Complex and time consuming. We do not have any operator performance tests for MXNet. It would be ideal to revisit this approach for future enhancement.

Alternate Solution 3 - Extend existing unit tests to cover performance parameters

<To add more details> In summary, it is hard and complex to modify all unit tests to measure performance along with currently designed way of writing tests which is designed towards - consistency across context, correctness, gradient checks.

Appendix

Phase 1

Functionality supported:

1. Able to run benchmarks for critical MXNet operators (see below for the list) with:
    a. Default inputs - Varying input shapes. Ex: Small Tensors, Large Tensors, Skewed Tensor, common tensor shapes.
    b. Custom inputs - Users should be able to specify input tensors to run benchmarks
    c. Random inputs - Randomly generate input tensors for operator benchmarks
2. Able to save results as JSON or get results as Python Dictionary
3. Able to run benchmarks on CPU/GPU.

Below are the operators covered in Phase 1

1. Hardware
    a. CPU - C5.18X
    b. GPU - P3.2X (Single GPU)
2. NDArray Operations
    a. Copy, CopyTo, as_in_context, asnumpy, asscalar, astype
    a. zeros, zeros_like, ones, ones_like, full, arange
    a. Transpose (T), shape_array, size_array, reshape, reshape, reshape_like, flatten, expand_dims, split, diag
    b. tile, pad
    a. sum, nansum, prod, nanprod, mean, max, min, norm
    a. sort, argsort, topk, argmax, argmin, argmax_channel
    a. add, sub, neg, mul, div, mod, pow
    a. iadd (+=), isub (-=), imul (*=), idiv (/=), imod (%=)
    a. lesser, lesser_equal, greater, greater_equal, equal, not_equal
    a. get_item (x[i]), set_item (x[i]=)
    b. slice, slice_axis, take, batch_take, pick
    c. one_hot
    a. exp, log
    a. sqrt, square
    a. concat, split, stack
    a. dot, batch_dot
    a. normal, poisson, uniform, random, randn, randint
    b. shuffle
    a. clip
    b. where
    c. abs
    a. Precision - FP32
    b. Conversion Operations
    c. Creation Operations
    d. Shape (view) change Operations
    e. Reduction Operations
    f. Sorting and Searching Operations
    g. Arithmetic Operations
    h. Inplace Arithmetic Operations
    i. Comparison Operations
    j. Indexing Operations
    k. Exponents and Logarithms
    l. Powers Operations
    m. Join and Split Operations
    n. GEMM
    o. Random Sampling
    p. Others
    q. Neural Network Operations
3. Gluon Layers (Neural Network Operations)
    a. Dense, Lambda, Flatten, Embedding
    b. Dropout, BatchNorm
    a. Conv1D, Conv2D
    b. Conv1DTranspose, Conv2DTranspose
    a. MaxPool1D, MaxPool2D, AvgPool1D, AvgPool2D, GlobalMaxPool1D, GlobalMaxPool2D, GlobalAvgPool1D, GlobalAvgPool2D
    a. LeakyRelu, PRelu, Sigmoid, Softmax, Log_Softmax, Activation
    a. RNNCell, LSTMCell, GRUCell, RecurrentCell, SequentialRNNCell, BiDirectionalCell
    a. L1Loss, L2Loss, SigmoidBinaryCrossEntropyLoss, SoftmaxCrossEntropyLoss, KLDivLoss, HuberLoss, HingeLoss, SquaredHingeLoss, LogisticLoss, TripletLoss, CTCLoss
    a. Modes: Imperative (Hybrid will be added in Next Phase)
    b. Basic
    c. Convolutions
    d. Pooling
    e. Activations
    f. Recurrent Cells
    g. Loss
4. Custom Operator Benchmark

Phase 2

1. Other DataTypes
    a. INT64, INT8, FP64, FP16
2. NDArray Operations
    a. tostype
    a. swapaxes, flip, depth_to_space, space_to_depth
    a. round, rint, fix, floor, ceil, trunc
    a. sin, cos, tan, arcsin, arccos, arctan, degrees, radians
    b. sinh, cosh, tahnh, arcsinh, arccosh, arctanh
    a. expm1, log10, log2, log1p
    a. logical_and, logical_or, logical_xor, logical_not
    a. rsqrt, cbrt, rcbrt, reciprocal
    a. exponential, gamma, generalized_negative_binomial, multinomial, negative_binomial,
    a. SequenceLast, SequenceMask, SequenceReverse

    a. unravel_index, ravel_multi_index
    a. Sparse
    b. View Operations
    c. Rounding Operations
    d. Trigonometric Operations
    e. Exponent and Logarithmic Operations
    f. Logical Operations
    g. Powers Operations
    h. Random Operations
    i. Sequence Operations
    j. Others
3. Gluon
    a. Conv + Relu, Conv + BatchNorm (More to be added when we start this work)
    a. HybridLambda
    b. InstanceNorm, LayerNorm
    a. Conv3D
    b. Conv3DTranspose
    a. MaxPool3D, AvgPool3D, GlobalMaxPool3D, GlobalAvgPool3D
    a. , Elu, Selu, Swish
    a. ZoneOutCell, ResidualCell, DropoutCell
    a. CosineEmbeddingLoss, PoissonNLLoss
    a. Mode: Hybrid Mode for all layers covered in Phase 1. Additional coverage of layers as below.
    b. Fused Operators
    c. Basic
    d. Convolutions
    e. Pooling
    f. Activations
    g. Recurrent
    h. Loss
    i. Important Contrib Layers
4. Other Items to be explored
    a. Image APIs
    b. Data APIs
    c. Metric APIs
    d. Initializers and Optimizers

Phase 3 (To be discussed/scoped)

Benchmark PyTorch Operators to have neutral baseline for comparing MXNet operator performance. This is yet to be discussed and finalized.

FAQs

Q1) Why not use check_speed(..) utility in MXNet test_util?
A) Supports Symbol APIs only. Do not support benchmarking NDArray, Gluon Blocks. It is a lightweight simple symbol executor and expects users to create symbol graph to execute along with inputs. The proposed library in this document is more sophisticated by supporting benchmarking operators with NDArray, Gluon Block, provide various default inputs, provide high-level drivers, provide interface for users to specify different inputs, prepare results in different formats - Python dictionary, CSV, JSON.

Q2) Why not Symbol execution? Why only NDArray and Gluon?
A) MXNet users are encouraged and are mainly using NDArray APIs or Gluon APIs. MXNet community is moving towards deprecating symbol APIs with works on numpy compatible operators. To measure what our users use and observe, we propose to use NDArray and Gluon blocks for benchmarking operators in this library.

Q3) Why not extend/repurpose current MXNet unit tests?

Q4) Why Python? Shouldn't we benchmark operators as close to Kernel as possible i.e., C++ benchmarks?