PYTORCH

Adam Paszke, Sam Gross, Soumith Chintala, **Francisco Massa**, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Alban Desmaison, Andreas Kopf, Edward Yang, Zach Devito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy & Team

Performance optimizations with PyTorch

Performance optimizations for PyTorch

- Mastering Tensors
  - Batching operations
  - Advanced indexing

Performance optimizations for PyTorch

- Mastering Tensors
  - Batching operations
  - Advanced indexing
- PyTorch hybrid-frontend
  - Compile Python code in C++ / CUDA
  - Fuse operations together

Performance optimizations for PyTorch

- Mastering Tensors
  - Batching operations
  - Advanced indexing
- PyTorch hybrid-frontend
  - Compile Python code in C++ / CUDA
  - Fuse operations together
- Delving deeper with C++/CUDA extensions
  - Use PyTorch C++ API and handwritten kernels for maximum speed

Mastering Tensors

- Tensors are:
  - multi-dimensional arrays
    - `torch.rand(3, 2, 4, 5, 2)`
    - `torch.rand(10, 4).unsqueeze(1)`

Mastering Tensors

- Tensors are:
  - multi-dimensional arrays
    - `torch.rand(3, 2, 4, 5, 2)`
    - `torch.rand(10, 4).unsqueeze(1)`
  - support non-contiguous memory access
    - `tensor.t()` shares memory with `tensor`

Mastering Tensors

- Tensors are:
  - multi-dimensional arrays
    - `torch.rand(3, 2, 4, 5, 2)`
    - `torch.rand(10, 4).unsqueeze(1)`
  - support non-contiguous memory access
    - `tensor.t()` shares memory with `tensor`
  - support efficient broadcast
    - `torch.rand(10, 1) + torch.rand(1, 10)` -> tensor of shape [10, 10]
    - equivalent to `tensor1.expand(10, 10) + tensor2.expand(10, 10)`

Batching operations

- How can batching be useful?
  - Performing operations on larger tensors is more efficient than in smaller ones
    - Less Python overhead
    - Better exploit parallelism / vectorization at the C++ / CUDA level

Batching operations

- How can batching be useful?
  - Performing operations on larger tensors is more efficient than in smaller ones
    - Less Python overhead
    - Better exploit parallelism / vectorization at the C++ / CUDA level

```
def add_1d(x, y):
    out = torch.empty_like(x)
    N = x.shape[0]
    for i in range(N):
        out[i] = x[i] + y[i]
    return out
```

Batching operations

- How can batching be useful?
  - Performing operations on larger tensors is more efficient than in smaller ones
    - Less Python overhead
    - Better exploit parallelism / vectorization at the C++ / CUDA level

```
def add_1
    out = t
    N = x.sh
    for i
        o
    retu
```

```
out = x + y
```

Batching operations

- With the proper tensor preparation, batching can be used more often than what is generally thought.

Batching operations

- With the proper tensor preparation, batching can be used more often than what is generally thought.
  - adding dimensions with unsqueeze (= indexing with a None)
  - expanding tensors
    - = broadcast

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
def pairwise_sum_prod(x):
    N = x.shape[0]
    s = 0
    for i in range(N):
        for j in range(N):
            s += x[i] * x[j]
    return s
```

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
def pairwise_sum_prod(x):
    N = x.shape[0]
    s = 0
    for i in range(N):
        for j in range(N):
            s += x[i] * x[j]
    return s
```

x is a tensor
of size [100]

%timeit pairwise_sum_prod(x)
82.3 ms ± 1.38 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
x = tensor([0.3148, 0.0059, 0.9338])
```

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
x = tensor([0.3148, 0.0059, 0.9338])
```

```
x[None, :] -> tensor([[0.3148, 0.0059, 0.9338]])

x[:, None] -> tensor([[0.3148],
                      [0.0059],
                      [0.9338]])
```

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
x = tensor([0.3148, 0.0059, 0.9338])
```

`x[:, None].expand(3, 3)`

```
tensor([[0.3148, 0.3148, 0.3148],
        [0.0059, 0.0059, 0.0059],
        [0.9338, 0.9338, 0.9338]])
```

`x[None, :].expand(3, 3)`

```
tensor([[0.3148, 0.0059, 0.9338],
        [0.3148, 0.0059, 0.9338],
        [0.3148, 0.0059, 0.9338]])
```

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
x = tensor([0.3148, 0.0059, 0.9338])
```

```
x[:, None].expand(3, 3)  *  x[None, :].expand(3, 3)
```

```
tensor([[9.9121e-02, 1.8654e-03, 2.9399e-01],
        [1.8654e-03, 3.5106e-05, 5.5328e-03],
        [2.9399e-01, 5.5328e-03, 8.7197e-01]])
```

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
def pairwise_sum_prod_opt(x):
    s = x[:, None] * x[None, :]
    return torch.sum(s)
```

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
def pairwise_sum_prod_opt(x):
    s = x[:, None] * x[None, :]
    return torch.sum(s)
```

Automatic broadcasting

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
def pairwise_sum_prod_opt(x):
    s = x[:, None] * x[None, :]
    return torch.sum(s)
```

x is a tensor of size [100]

%timeit pairwise_sum_prod_opt(x)
24.9 µs ± 193 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Example of batching

Let `x` be a 1d tensor of size `N`. Compute `sum(x_i * x_j)`

```
def pairwise_sum_prod_opt(x):
    s = x[:, None] * x[None, :]
    return torch.sum(s)
```

3300x faster

x is a tensor of size [100]

%timeit pairwise_sum_prod_opt(x)
24.9 µs ± 193 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]

```
def pairwise_distance(a, b):
    squares = torch.zeros((a.shape[0], b.shape[0]))
    for i in range(squares.shape[0]):
        for j in range(squares.shape[1]):
            diff = a[i, :] - b[j, :]
            sqr = diff ** 2.0
            squares[i, j] = torch.sum(sqr)
    return squares
```

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]

```
def pairwise_distance(a, b):
    squares = torch.zeros((a.shape[0], b.shape[0]))
    for i in range(squares.shape[0]):
        for j in range(squares.shape[1]):
            diff = a[i, :] - b[j, :]
            sqr = diff ** 2.0
            squares[i, j] = torch.sum(sqr)
    return squares
```

a -> [100, 10], b -> [200, 10]

```
%timeit pairwise_distance(a, b)
527 ms ± 11.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]

Can we do better?

```
def pairwise_distance(a, b):
    squares = torch.zeros((a.shape[0], b.shape[0]))
    for i in range(squares.shape[0]):
        for j in range(squares.shape[1]):
            diff = a[i, :] - b[j, :]
            sqr = diff ** 2.0
            squares[i, j] = torch.sum(sqr)
    return squares
```

a -> [100, 10], b -> [200, 10]

```
%timeit pairwise_distance(a, b)
527 ms ± 11.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]
  - Take all combinations on N and M

```
a = tensor([[0.4611, 0.9329],
            [0.1024, 0.8120],
            [0.9358, 0.2192]])
```

```
b = tensor([[0.8645, 0.3465],
            [0.0489, 0.0582],
            [0.8607, 0.1976],
            [0.6976, 0.2638]])
```

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]
  - Take all combinations on N and M

```
a[:, None, :].expand(3, 4, 2)
```

```
tensor([[[0.4611, 0.9329],
         [0.4611, 0.9329],
         [0.4611, 0.9329],
         [0.4611, 0.9329]],

        [[0.1024, 0.8120],
         [0.1024, 0.8120],
         [0.1024, 0.8120],
         [0.1024, 0.8120]],

        [[0.9358, 0.2192],
         [0.9358, 0.2192],
         [0.9358, 0.2192],
         [0.9358, 0.2192]]])
```

```
b[None, :, :].expand(3, 4, 2)
```

```
tensor([[[0.8645, 0.3465],
         [0.0489, 0.0582],
         [0.8607, 0.1976],
         [0.6976, 0.2638]],

        [[0.8645, 0.3465],
         [0.0489, 0.0582],
         [0.8607, 0.1976],
         [0.6976, 0.2638]],

        [[0.8645, 0.3465],
         [0.0489, 0.0582],
         [0.8607, 0.1976],
         [0.6976, 0.2638]]])
```

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]
  - Take all combinations on N and M

```
a[:, None, :].expand(3, 4, 2)
```

```
tensor([[[0.4611, 0.9329],
         [0.4611, 0.9329],
         [0.4611, 0.9329],
         [0.4611, 0.9329]],

        [[0.1024,
         [0.1024,
         [0.1024,
         [0.1024,

        [[0.9358, 0.2192],
         [0.9358, 0.2192],
         [0.9358, 0.2192],
         [0.9358, 0.2192]]])
```

```
b[None, :, :].expand(3, 4, 2)
```

```
tensor([[[0.8645, 0.3465],
         [0.0489, 0.0582],
         [0.8607, 0.1976],
         [0.6976, 0.2638]],

                    465],
                    582],
                    976],
                    2638]],

        [[0.8645, 0.3465],
         [0.0489, 0.0582],
         [0.8607, 0.1976],
         [0.6976, 0.2638]]])
```

Can now subtract, square and sum the two tensors

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]

```
def pairwise_distance_opt(a, b):
    diff = a[:, None, :] - b[None, :, :]
    sqr = diff ** 2.0
    squares = torch.sum(sqr, dim=2)
    return squares
```

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]

```
def pairwise_distance_opt(a, b):
    diff = a[:, None, :] - b[None, :, :]
    sqr = diff ** 2.0
    squares = torch.sum(sqr, dim=2)
    return squares
```

```
%timeit pairwise_distance_opt(a, b)
199 µs ± 16.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Example of batching

- Square distance between two sets of points
  - a [N, k], b [M, k]

```
def pairwise_distance_opt(a, b):
    diff = a[:, None, :] - b[None, :, :]
    sqr = diff ** 2.0
    squares = torch.sum(sqr, dim=2)
    return squares
```

2648x faster!

```
%timeit pairwise_distance_opt(a, b)
199 µs ± 16.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Advanced indexing

- Very powerful functionality
- Enables iterating over arbitrary elements of a tensor in a vectorized way
- Follows numpy semantics

Advanced indexing

- Very powerful functionality
- Enables iterating over arbitrary elements of a tensor in a vectorized way
- Follows numpy semantics

If you need to retrieve / put elements in a tensor in specific locations, advanced indexing can most probably do it for you

Advanced indexing

```
x = tensor([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

Advanced indexing

```
x = tensor([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
x[[0, 2], [1, 0]]
```

Advanced indexing

```
x = tensor([[0, 1  2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
x[[0  2], [1  0]]
```

Advanced indexing

```
x = tensor([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
x[[0, 2], [1, 0]]
```

Advanced indexing

```
x = tensor([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
x[[0, 2], [1, 0]]
```

```
tensor([1, 6])
```

Advanced indexing

```
x = tensor([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
x[[0, 2], [[1], [0]]]
```

Advanced indexing

```
x = tensor([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
x[[0, 2], [[1], [0]]]
```

Indices are broadcast together

```
tensor([[0, 2],
        [0, 2]])
```

```
tensor([[1, 1],
        [0, 0]])
```

Advanced indexing

```
x = tensor([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
tensor([[1, 7],
        [0, 6]])
```

```
x[[0, 2], [[1], [0]]]
```

Indices are broadcast together

```
tensor([[0, 2],
        [0, 2]])
```

```
tensor([[1, 1],
        [0, 0]])
```

PyTorch Hybrid Front-End

What is PyTorch Hybrid Front-End

- Run and optimize PyTorch programs separate from the Python interpreter
  - Great for
    - speed via just in time compilation (JIT)
    - deployment in C++ only environments

What is PyTorch Hybrid Front-End

- Two ways to specify TorchScript programs

tracing:

- Executes python code and records the operations that were executed
- Control flow gets executed (and thus only one path will ever get executed)

```
def my_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
traced_fn = torch.jit.trace(my_fn, a)
traced_fn(a)
```

scripting:

- Parses the Python Abstract Syntax Tree
- Allow for a subset of Python code
  - including for loops, conditionals, etc

```
@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
```

What is PyTorch Hybrid Front-End

- Two ways to specify TorchScript programs

tracing:

```
def my_fn(x):
    for i in range(5):
        x = x * x
    return x


a = torch.rand(5)
traced_fn = torch.jit.trace(my_fn, a)
print(traced_fn.code)
```

scripting:

```
@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.code)
```

What is PyTorch Hybrid Front-End

- Two ways to specify TorchScript programs

tracing:

```
def my_fn(x):
    for i in range(5):
        x = x * x
    return x


a = torch.rand(5)
traced_fn = torch.jit.trace(my_fn, a)
print(traced_fn.code)


# outputs
def forward(self,
    x: Tensor) -> Tensor:
  x0 = torch.mul(x, x)
  x1 = torch.mul(x0, x0)
  x2 = torch.mul(x1, x1)
  x3 = torch.mul(x2, x2)
  return torch.mul(x3, x3)
```

scripting:

```
@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.code)


# outputs
def forward(self,
    x: Tensor) -> Tensor:
  x0 = x
  for i in range(5):
    x0 = torch.mul(x0, x0)
  return x0
```

Speeding up PyTorch with JIT

Speeding-up PyTorch with JIT

- What optimizations do we want to do?
  - Algebraic rewriting
    - Constant folding, common subexpression elimination, dead code elimination, etc
  - Out-of-order execution
    - Re-ordering operations to reduce memory pressure and make efficient use of cache locality
  - Kernel fusion
    - Combining operations into a single kernel to avoid per-op overhead
  - Target-dependent code generation
    - Taking parts of the program and compiling them for specific hardware
    - Integration ongoing with several codegen frameworks: TVM, Halide, Glow, XLA

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Single kernel!

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

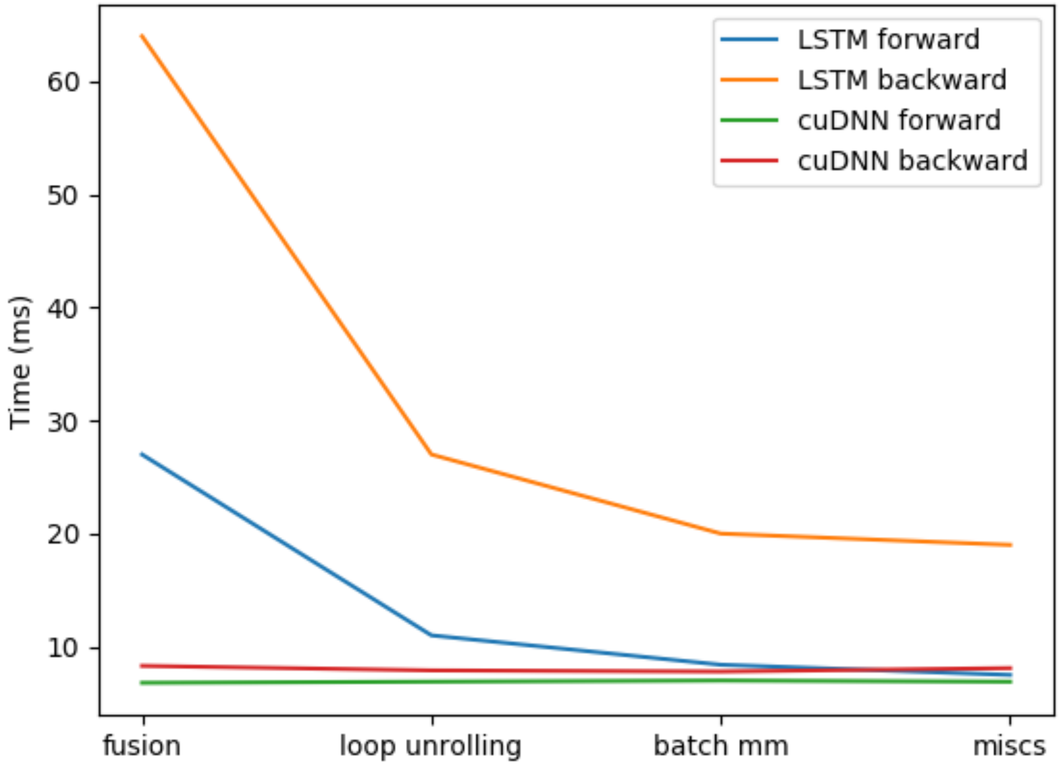
```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

```
%timeit script_fn(a)
4.02 ms ± 62.5 µs per loop (mean ± std.
dev. of 7 runs, 100 loops each)
```

```
%timeit my_fn(a)
7.29 ms ± 50.1 µs per loop (mean ± std.
dev. of 7 runs, 100 loops each)
```

Speeding-up PyTorch with JIT

- Do models actually get faster?



C++ / CUDA Extensions

C++ / CUDA Extensions

- Easily wrap C++ / CUDA kernels inside PyTorch
- Supports on-the-fly and ahead-of-time compilation
  - Your C++ code can be a string in Python
    - Gets compiled when executed
- Custom ops for torch script interoperability

C++ / CUDA Extensions

Registering an arbitrary custom op to TorchScript and PyTorch eager:

```
#include <opencv2/opencv.hpp>
#include <torch/script.h>
// taken from https://pytorch.org/tutorials/advanced/torch_script_custom_ops.html
torch::Tensor warp_perspective(torch::Tensor image, torch::Tensor warp) {
  cv::Mat image_mat(image.size(0), image.size(1), CV_32FC1, image.data<float>());
  cv::Mat warp_mat(warp.size(0), warp.size(1), CV_32FC1, warp.data<float>());
  cv::Mat output_mat;
  cv::warpPerspective(image_mat, output_mat, warp_mat, /*dsize=*/{8, 8});
  torch::Tensor output = torch::from_blob(output_mat.ptr<float>(), /*sizes=*/{8, 8});
  return output.clone();
}

// Register the op with the JIT
torch::jit::RegisterOperators("cv::warp_perspective", &warp_perspective);
```

C++ / CUDA Extensions

Using it from Python in PyTorch eager:

```
import torch
torch.ops.load_library("libwarp_perspective.so")
# use module binding magic to make it "look" like any other torch op.
torch.ops.cv.warp_perspective(torch.randn(32, 32), torch.rand(3, 3))
```

Using it in TorchScript is identical.

```
@torch.jit.script
def warp(image, warp):
    return torch.ops.cv.warp_perspective(image, warp)

print(warp.code)
> def warp(image: Tensor,
>     warp: Tensor) -> Tensor:
>   return ops.cv.warp_perspective(image, warp)
```

C++ / CUDA Extensions

Using it from Python in PyTorch eager:

```
import torch
torch.ops.load_library('
# use module binding
torch.ops.cv.warp_pe
```

```
ch op.
, 3))
```

Using it in TorchScript is identical.

```
@torch.jit.script
def warp(image, warp):
    return torch.ops.cv.
```

```
print(warp.code)
> def warp(image: Tensor,
>     warp: Tensor) -> Tensor:
>   return ops.cv.warp_perspective(image, warp)
```

This operator registration mechanism is the same one the JIT uses to register internal operators, so to the compiler your custom operator is indistinguishable from a built-in one.

Tips and tricks

Identifying bottlenecks

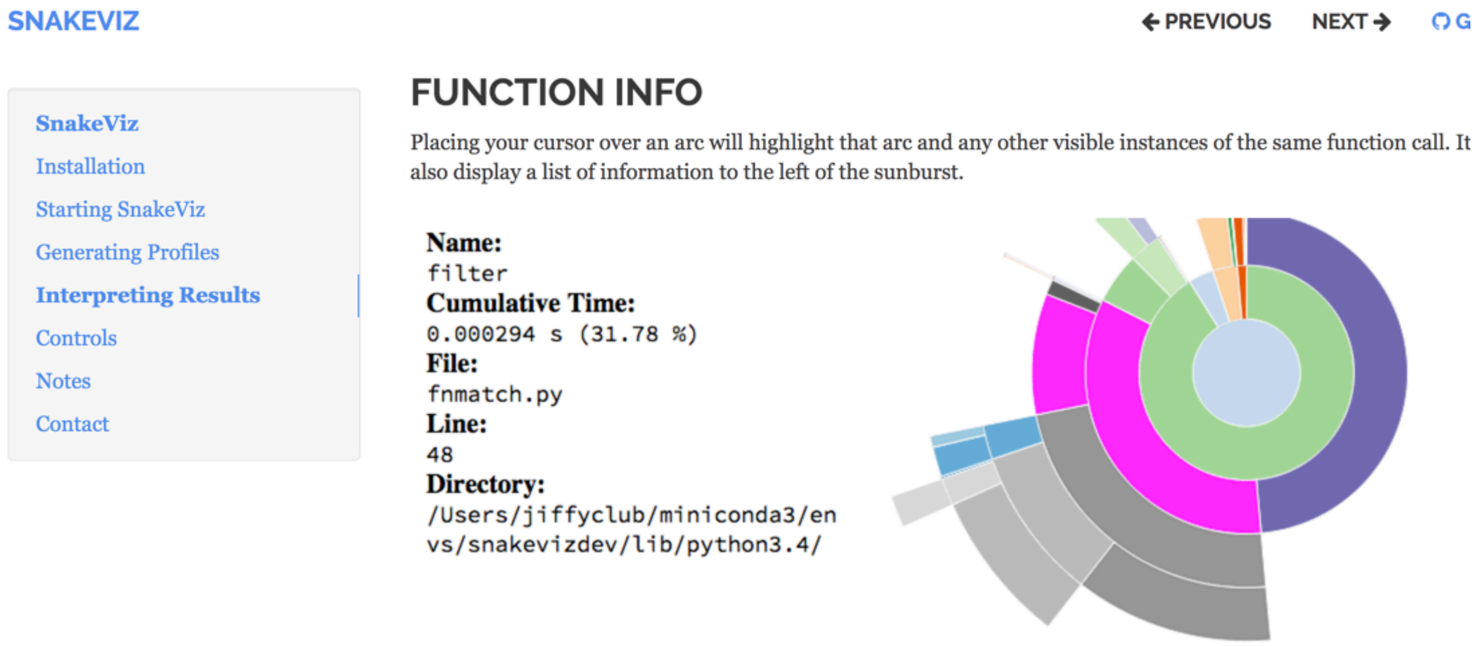
- Some possibilities
  - Python profiler (for CPU-only code)
  - torch.utils.bottleneck
  - torch.autograd.profiler

Identifying bottlenecks

- Some possibilities
  - Python profiler (for CPU-only code)
    - %timeit

Identifying bottlenecks

- Some possibilities
  - Python profiler (for CPU-only code)



Identifying bottlenecks

- Some possibilities
  - torch.utils.bottleneck

TORCH.UTILS.BOTTLENECK

torch.utils.bottleneck is a tool that can be used as an initial step for debugging bottlenecks in your program. It summarizes runs of your script with the Python profiler and PyTorch's autograd profiler.

Run it on the command line with

```
python -m torch.utils.bottleneck /path/to/source/script.py [args]
```

where [args] are any number of arguments to *script.py*, or run `python -m torch.utils.bottleneck -h` for more usage instructions.

Identifying bottlenecks

- Some possibilities
  - torch.autograd.profiler

Profiler

Autograd includes a profiler that lets you inspect the cost of different operators inside your model - both on the CPU and GPU. There are two modes implemented at the moment - CPU-only using `profile`. and nvprof based (registers both CPU and GPU activity) using `emit_nvtx`.

CLASS `torch.autograd.profiler.profile(enabled=True, use_cuda=False)` [SOURCE]

Context manager that manages autograd profiler state and holds a summary of results.

Parameters

- **enabled** (*bool, optional*) – Setting this to False makes this context manager a no-op. Default: `True`.
- **use_cuda** (*bool, optional*) – Enables timing of CUDA events as well using the cudaEvent API. Adds approximately 4us of overhead to each tensor operation. Default: `False`

Identifying bottlenecks

- Some possibilities
  - torch.autograd.profiler

```
>>> x = torch.randn((1, 1), requires_grad=True)
>>> with torch.autograd.profiler.profile() as prof:
...     y = x ** 2
...     y.backward()
>>> # NOTE: some columns were removed for brevity
... print(prof)
---------------------------------  --------------  --------------
Name                                    CPU time        CUDA time
---------------------------------  --------------  --------------
PowConstant                           142.036us         0.000us
N5torch8autograd9GraphRootE            63.524us         0.000us
PowConstantBackward                   184.228us         0.000us
MulConstant                            50.288us         0.000us
PowConstant                            28.439us         0.000us
Mul                                    20.154us         0.000us
N5torch8autograd14AccumulateGradE      13.790us         0.000us
N5torch8autograd5CloneE                 4.088us         0.000us
```

Questions?