How PyTorch Optimizes Deep Learning Computations

Vincent Quenneville-Bélair, PhD. Facebook AI.

Overview

Compute with PyTorch

Model with Neural Networks

Ingest Data

Use Multiple GPUs and Machines

Compute with PyTorch

Example: Pairwise Distance

```
def pairwise_distance(a, b):
    p = a.shape[0]
    q = b.shape[0]
    squares = torch.zeros((p, q))
    for i in range(p):
        for j in range(q):
            diff = a[i, :] - b[j, :]
            diff_squared    = diff ** 2
            squares[i, j] = torch.sum(diff_squared)
    return squares

a = torch.randn(100, 2)
b = torch.randn(200, 2)

%timeit pairwise_distance(a, b)
# 438 ms ± 16.7 ms per loop
```

Example: Batched Pairwise Distance

```
def pairwise_distance(a, b):
    diff = a[:, None, :] - b[None, :, :]  # Broadcast
    diff_squared = diff ** 2
    return torch.sum(diff_squared, dim=2)

a = torch.randn(100, 2)
b = torch.randn(200, 2)

%timeit pairwise_distance(a, b)
# 322 μs ± 5.64 μs per loop
```

Debugging and Profiling

`%timeit`, `print`, `pdb`

`torch.utils.bottleneck`

`torch.autograd.profiler.profile`

also `pytorch.org/docs/stable/jit.html#debugging`

Script for Performance

Eager mode: PyTorch – Models are simple debuggable python programs for prototyping

Script mode: TorchScript – Models are programs transpiled and ran by lean JIT interpreter in production

From Eager to Script Mode

```
a = torch.rand(5)
def func(x):
        for i in range(10):
                x = x * x
        return x

# often used as decorator
scripted_func = torch.jit.script(func)  # also trace

%timeit func(a)
# 18.5 µs ± 229 ns per loop

%timeit scripted_func(a)
# 4.41 µs ± 26.5 ns per loop
```

JIT Intermediate Representation with Fused Operations

```
scripted_func.graph_for(a)

# graph(%x.1 : Float(*)):
#     %x.15 : Float(*) = prim::FusionGroup_0(%x.1)
#     return (%x.15)
# with prim::FusionGroup_0 = graph(%18 : Float(*)):
#     %x.4 : Float(*) = aten::mul(%18, %18) # <ipython-input-13-1ec878
#     %x.5 : Float(*) = aten::mul(%x.4, %x.4) # <ipython-input-13-1ec8
#     %x.6 : Float(*) = aten::mul(%x.5, %x.5) # <ipython-input-13-1ec8
#     %x.9 : Float(*) = aten::mul(%x.6, %x.6) # <ipython-input-13-1ec8
#     %x.10 : Float(*) = aten::mul(%x.9, %x.9) # <ipython-input-13-1ec
#     %x.11 : Float(*) = aten::mul(%x.10, %x.10) # <ipython-input-13-1
#     %x.12 : Float(*) = aten::mul(%x.11, %x.11) # <ipython-input-13-1
#     %x.13 : Float(*) = aten::mul(%x.12, %x.12) # <ipython-input-13-1
#     %x.14 : Float(*) = aten::mul(%x.13, %x.13) # <ipython-input-13-1
#     %x.15 : Float(*) = aten::mul(%x.14, %x.14) # <ipython-input-13-1
#     return (%x.15)

scripted_func.save("func.pt")
```

Performance Improvements

Algebraic rewriting – Constant folding, common subexpression elimination, dead code elimination, loop unrolling, etc.

Out-of-order execution – Re-ordering operations to reduce memory pressure and make efficient use of cache locality

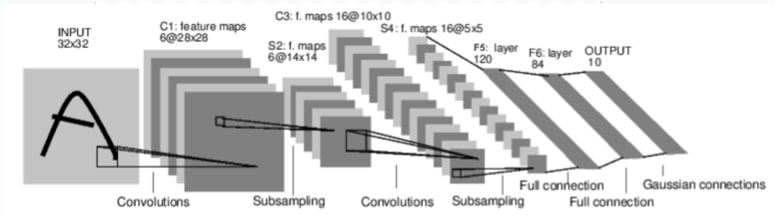
Kernel fusion – Combining several operators into a single kernel to avoid per-op overhead

Target-dependent code generation – Compiling parts of the program for specific hardware. Integration ongoing with codegen frameworks: TVM, Halide, Glow, XLA

Runtime – No python global interpreter lock. Fork and wait parallelism.

Model with Neural Networks

Application to Vision

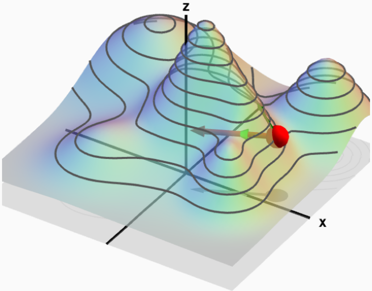


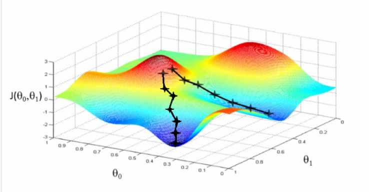
Neural Network

```python
class Net(torch.nn.Module):

    def __init__(self):
        ...

    def forward(self, x):
        ...

model = Net()
print(model)

# Net(
#     (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
#     (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
#     (fc1): Linear(in_features=576, out_features=120, bias=True)
#     (fc2): Linear(in_features=120, out_features=84, bias=True)
#     (fc3): Linear(in_features=84, out_features=10, bias=True)
# )
```

How do we choose the parameters?

Gradient Descent, $-df/dw$





GD to SGD

Minimize

$$L(w) = \frac{1}{n} \sum_i L_i(w)$$

Gradient Descent

$$w \leftarrow w - \alpha \frac{1}{n} \sum_i \frac{d}{dw} L_i(w)$$

Stochastic Gradient Descent

$$w \leftarrow w - \alpha \frac{d}{dw} L_i(w)$$

Test of time award in 2018!

Bottou Bousquet 2008

GD to SGD

Minimize

$$L(w) = \frac{1}{n} \sum_i L_i(w)$$

Gradient Descent

$$w \leftarrow w - \alpha \frac{1}{n} \sum_i \frac{d}{dw} L_i(w)$$

Stochastic Gradient Descent

$$w \leftarrow w - \alpha \frac{d}{dw} L_i(w)$$

Test of time award in 2018!

How do we compute derivatives?

Backpropagation

The derivative of

$$y = f_3(f_2(f_1(x)))$$

is

$$\frac{dy}{dx} = \frac{df_3}{df_2}\frac{df_2}{df_1}\frac{df_1}{dx}$$

by chain rule

Example

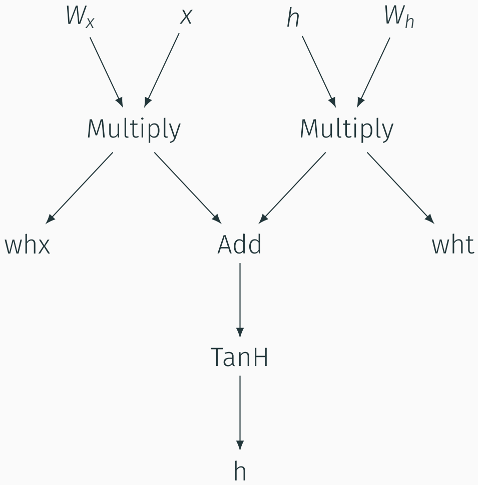
We can write

$$h_{i+1} = \tanh(W_h h_i^T + W_x x^T)$$

as

$$\begin{aligned} wht &\leftarrow W_h h^T \\ whx &\leftarrow W_x x^T \\ h &\leftarrow wht + whx \\ h &\leftarrow \tanh h \end{aligned}$$

Example



Backward pass provides derivative

Training Loop

```python
from torch.optim import SGD
from torch.optim.lr_scheduler import ExponentialLR

loader = ...
model = Net()
criterion = torch.nn.CrossEntropyLoss()  # LogSoftmax + NLLLoss

optimizer = SGD(model.parameters)
scheduler = ExponentialLR(optimizer)

for epoch in range(10):
        for batch, labels in loader:

                outputs = model(batch)
                loss = criterion(outputs, labels)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        scheduler.step()
```

Ingest Data

Datasets

```
class IterableStyleDataset(torch.utils.data.IterableDataset):

    def __iter__(self):
        # Support for streams
        ...


class MapStyleDataset(torch.utils.data.Dataset):

    def __getitem__(self, key):
        # Map from (non-int) keys
        ...

    def __len__(self):
        # Support sampling
        ...


# Preprocessing
```

DataLoader

```python
from torch.utils.data import DataLoader, RandomSampler

dataloader = DataLoader(
    dataset,                # only for map-style
    batch_size=8,           # balance speed and convergence
    num_workers=2,          # non-blocking when > 0
    sampler=RandomSampler,  # random read may saturate drive
    pin_memory=True,        # page-lock memory for data?
)
```

discuss.pytorch.org/t/how-to-prefetch-data-when-processing-with-gpu/548/19

Pinned Memory in DataLoader

Copy from host to GPU is faster from RAM directly. To prevent paging, pin tensor to page-locked RAM.

Once a tensor is pinned, use asynchronous GPU copies with `to(device, non_blocking=True)` to overlap data transfers with computation.

A single Python process can saturate multiple GPUs, even with the global interpreter lock.

pytorch.org/docs/stable/notes/cuda.html

Pinned Memory in DataLoader

Copy from host to GPU is faster from RAM directly. To prevent paging, pin tensor to page-locked RAM.

Once a tensor is pinned, use asynchronous GPU copies with `to(device, non_blocking=True)` to overlap data transfers with computation.

A single Python process can saturate multiple GPUs, even with the global interpreter lock.

Use Multiple GPUs and Machines

Data Parallel – Data distributed across devices

Model Parallel – Model distributed across devices

Single Machine Data Parallel

Single Machine Model Parallel

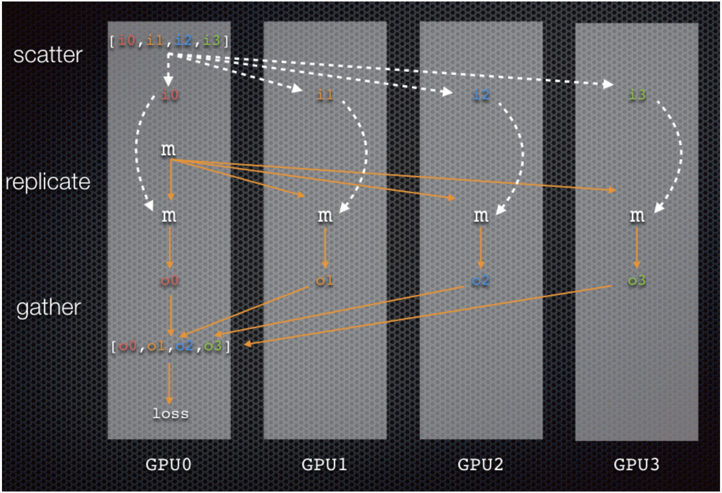
Distributed Data Parallel

Distributed Data Parallel with Model Parallel

Distributed Model Parallel

also Ben-Num Hoefler 2018

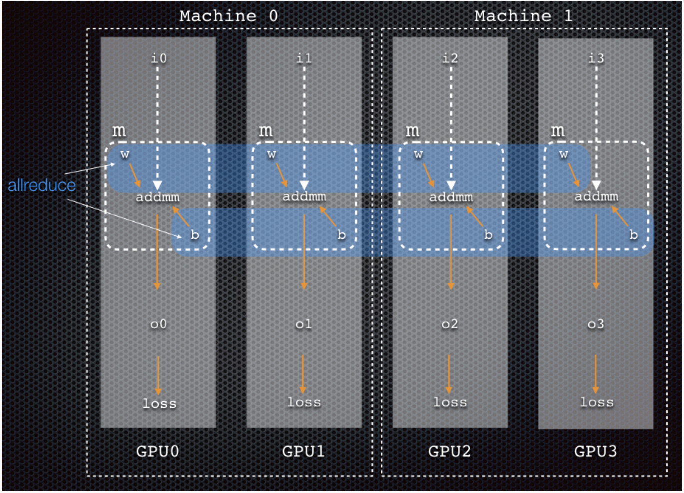
Single Machine Data Parallel



Single Machine Data Parallel

```python
model = Net().to("cuda:0")
model = torch.nn.DataParallel(model)  # also torch.multiprocessing

# training loop ...
```

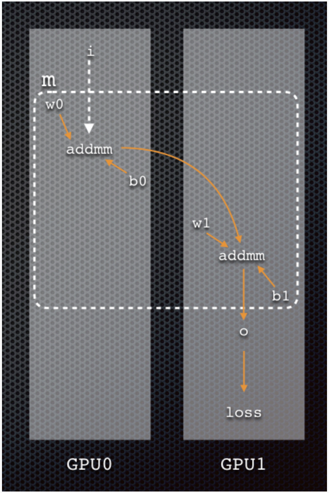
Single Machine Model Parallel



Single Machine Model Parallel

```python
class Net(torch.nn.Module):

    def __init__(self, gpus):
        super(Net).__init__(self)

        self.gpu0 = torch.device(gpus[0])
        self.gpu1 = torch.device(gpus[1])

        self.sub_net1 = torch.nn.Linear(10, 10).to(self.gpu0)
        self.sub_net2 = torch.nn.Linear(10,  5).to(self.gpu1)

    def forward(self, x):
        y = self.sub_net1(x.to(self.gpu0))
        z = self.sub_net2(y.to(self.gpu1))  # blocking
        return z

model = Net("cuda:0", "cuda:1")

# training loop ...
```

Distributed Data Parallel

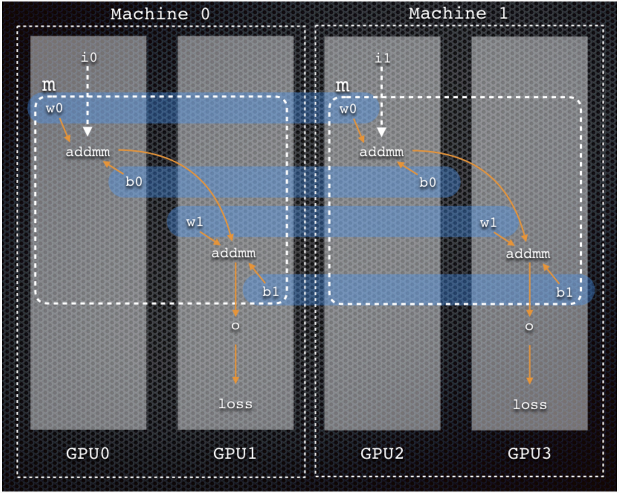


pytorch.org/tutorials/intermediate/ddp_tutorial.html

Distributed Data Parallel

```python
def one_machine(machine_rank, world_size, backend):
    torch.distributed.init_process_group(
        backend, rank=machine_rank, world_size=world_size
    )

    gpus = {
        0: [0, 1, 2, 3, 4],
        1: [5, 6, 7, 8, 9],
    }[machine_rank]
    # gpus = [machine_rank]  # or one gpu per process to avoid GIL

    model = Net().to(gpus[0])  # default to first gpu on machine
    model = torch.nn.parallel.DDP(model, device_ids=gpus)

    # training loop ...

for machine_rank in range(world_size):
    torch.multiprocessing.spawn(
        one_machine, args=(world_size, backend),
        nprocs=world_size, join=True  # blocking
    )
```

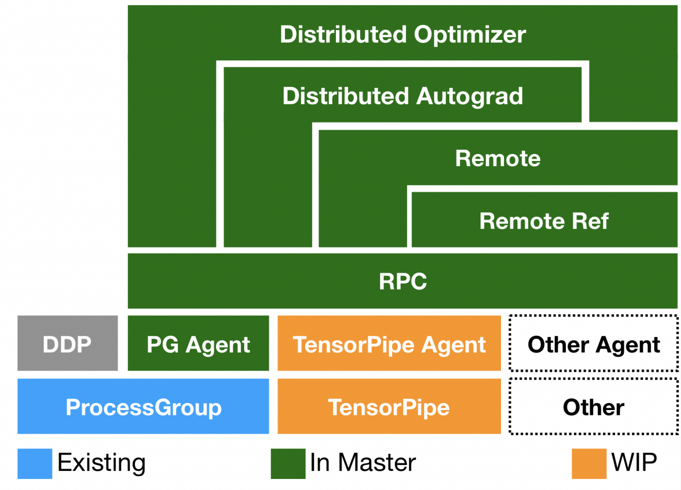
Distributed Data Parallel with Model Parallel



Distributed Data Parallel with Model Parallel

```python
def one_machine(machine_rank, world_size, backend):
    torch.distributed.init_process_group(
        backend, rank=machine_rank, world_size=world_size
    )

    gpus = {
        0: [0, 1], 1: [2, 3],
    }[machine_rank]

    model = Net(gpus)
    model = torch.nn.parallel.DDP(model)

    # training loop ...

for machine_rank in range(world_size):
    torch.multiprocessing.spawn(
        one_machine, args=(world_size, backend),
        nprocs=world_size, join=True
    )
```

Distributed Model Parallel (in development)



pytorch.org/docs/master/rpc.html

Conclusion

Conclusion

Scale from experimentation to production.

Questions?