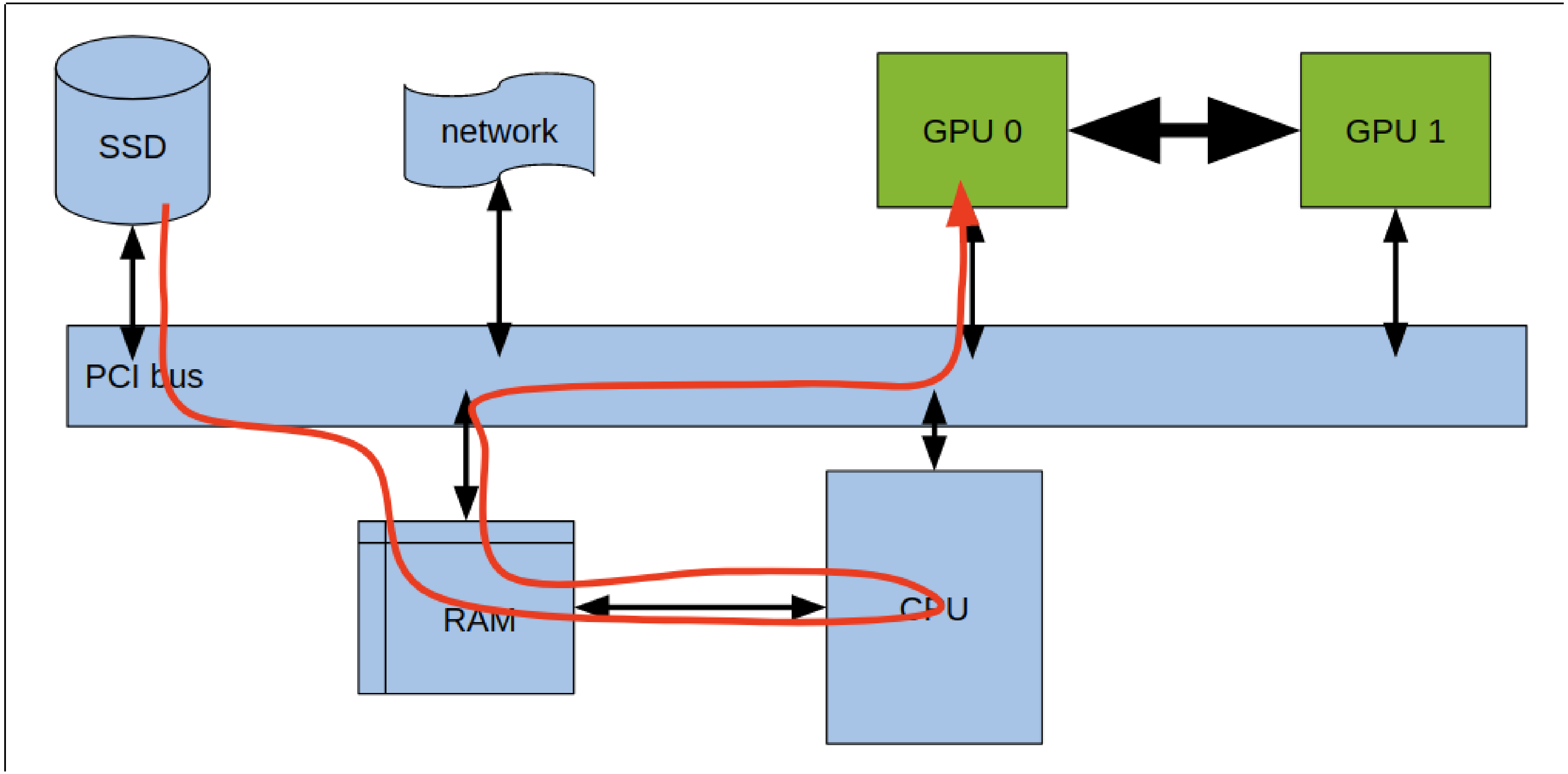
Performance and Profiling

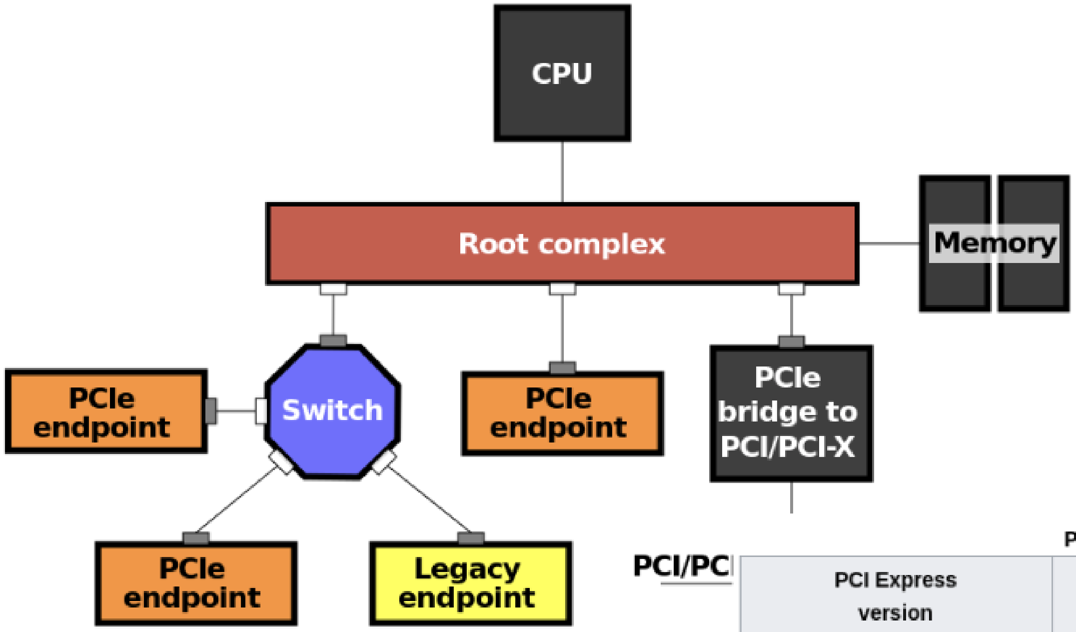
Flow of Data during Training



Common System Bandwidths

Hardware Component	speed	1 PB = ...
Intel i7 5930K (40 PCIe lanes)	30 GB/s	9 hours
NVLINK	40 GB/s	7 hours
PCIe x8 (GPU)	5 GB/s	2 days
SATA Interface	6 GB/s	2 days
56 Gbps Mellanox	6.8 GB/s	2 days
10 Gbps Ethernet	1 GB/s	2 weeks

Detailed PCI Bus Structure



PCI/PC

PCI Express link performance[29][32]

PCI Express version	Line code	Transfer rate[i]	Throughput[i]			
			×1	×4	×8	×16
1.0	8b/10b	2.5 GT/s	250 MB/s	1 GB/s	2 GB/s	4 GB/s
2.0	8b/10b	5 GT/s	500 MB/s	2 GB/s	4 GB/s	8 GB/s
3.0	128b/130b	8 GT/s	984.6 MB/s	3.94 GB/s	7.9 GB/s	15.8 GB/s
4.0	128b/130b	16 GT/s	1969 MB/s	7.9 GB/s	15.8 GB/s	31.5 GB/s
5.0[30][31] *(expected in 2019)*[33]	128b/130b	32 or 25 GT/s[ii]	3938 or 3077 MB/s	15.8 or 12.3 GB/s	31.5 or 24.6 GB/s	63.0 or 49.2 GB/s

PCI Express

Common Ways of Speeding Up Single GPU Jobs

- CPU
  - use multicore/multithreading
  - use pre-augmented data
  - avoid data copies
- GPU
  - use pinned memory for CPU/GPU transfers
  - overlay memory transfers and computation
  - switch to FP16 and use tensor cores
- speed up I/O
  - load your entire dataset into CPU/GPU memory
  - use sequential reads or NVMe

How do you find out what to do?

- system monitoring tools (CPU, GPU, disk, network)
- manual instrumentation, logging, and performance measurements
- mock loaders / trainers
- performance analysis and visualization tools

(In rough order from easy to hard.)

Manual Instrumentation

```
while True:
    t1 = time.time()
    inputs, targets = next(source)

    t2 = time.time()
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = lossfn(outputs, targets)
    loss.backward()
    optimizer.step()
    t3 = time.time()

    loading_time = moving_average(loading_time, t2-t1)
    training_time = moving_average(training_time, t3-t2)
```

I/O should overlap with training, and `t2-t1` should be much smaller than `t3-t2`

Performance Testing

To measure limit of training performance:

- mock up the loader
- store a single batch in CPU or GPU memory

To measure limit of I/O performance:

- mock up the training = measure loading performance
- just discard each batch after loading

I/O sample rate should be highr than training sample rate.

GPU Utilization

First thing to look at: **What is my GPU utilization?**

```
$ nvidi-smi
```

Check for:

- utilization of each GPU (should be close to 100%)
- GPU memory utilization (stay away from max)
- active processes and their usage (as few as possible)
- temperature (make sure you're cooled enough)

Also: NVIDIA GPU profiling tools

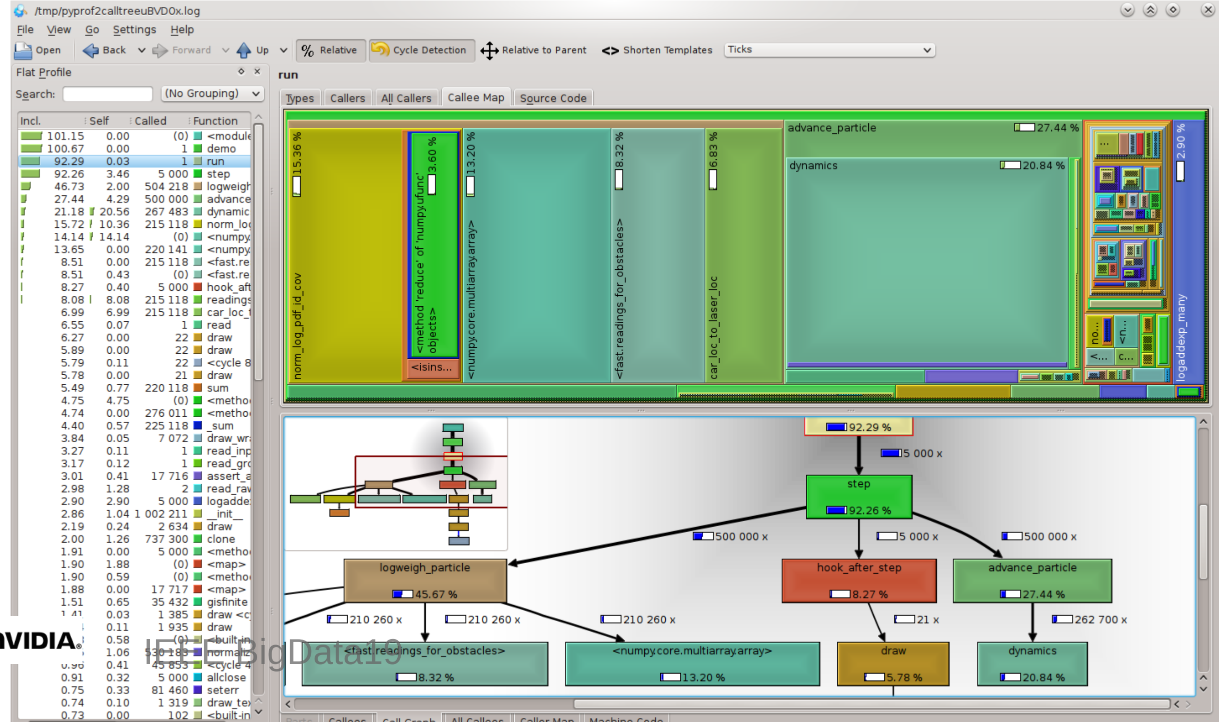
Tools: CPU Utilization

First thing to look at: **How are my CPU cores being used?**

```
$ htop
```

Many more tools, e.g.:

```
$ python -m cProfile -o myscript.cprof myscript.py
$ pyprof2calltree -k -i myscript.cprof
```



Tools: PyTorch Profiler

```
import torch
import torchvision.models as models

model = models.densenet121(pretrained=True)
x = torch.randn((16, 3, 224, 224), requires_grad=True)

with torch.autograd.profiler.profile(use_cuda=True) as prof:
    model(x)

print(prof)

prof.export_chrome_trace("mytrace") # open with chrome://tracing
```

Tools: PyTorch Profiler (Sample Output)

```
---------------------------------- --------------- --------------- --------------- --------------- ---------------
Name                                      CPU time       CUDA time           Calls       CPU total      CUDA total
---------------------------------- --------------- --------------- --------------- --------------- ---------------
conv2d                                  9976.544us      9972.736us               1      9976.544us      9972.736us
convolution                             9958.778us      9958.400us               1      9958.778us      9958.400us
_convolution                            9946.712us      9947.136us               1      9946.712us      9947.136us
contiguous                                 6.692us         6.976us               1         6.692us         6.976us
empty                                     11.927us        12.032us               1        11.927us        12.032us
mkldnn_convolution                      9880.452us      9889.792us               1      9880.452us      9889.792us
batch_norm                              1214.791us      1213.440us               1      1214.791us      1213.440us
native_batch_norm                       1190.496us      1193.056us               1      1190.496us      1193.056us
threshold_                               158.258us       159.584us               1       158.258us       159.584us
max_pool2d_with_indices                28837.682us     28836.834us               1     28837.682us     28836.834us
max_pool2d_with_indices_forward        28813.804us     28822.530us               1     28813.804us     28822.530us
batch_norm                              1780.373us      1778.690us               1      1780.373us      1778.690us
native_batch_norm                       1756.774us      1759.327us               1      1756.774us      1759.327us
threshold_                                64.665us        66.368us               1        64.665us        66.368us
conv2d                                  6103.544us      6102.142us               1      6103.544us      6102.142us
convolution                             6089.946us      6089.600us               1      6089.946us      6089.600us
_convolution                            6076.506us      6076.416us               1      6076.506us      6076.416us
contiguous                                 7.306us         7.938us               1         7.306us         7.938us
empty                                      9.037us         8.194us               1         9.037us         8.194us
mkldnn_convolution                      6015.653us      6021.408us               1      6015.653us      6021.408us
batch_norm                               700.129us       699.394us               1       700.129us       699.394us
```

Speedup: FP16 Computations

Important:

- intrinsically faster
- enables the use of TensorCores
- changes numerical results and can't be used with all computations

Speedups:

- between 1.5x and 5x for common models

Speedup: Converting to FP16

EXAMPLE

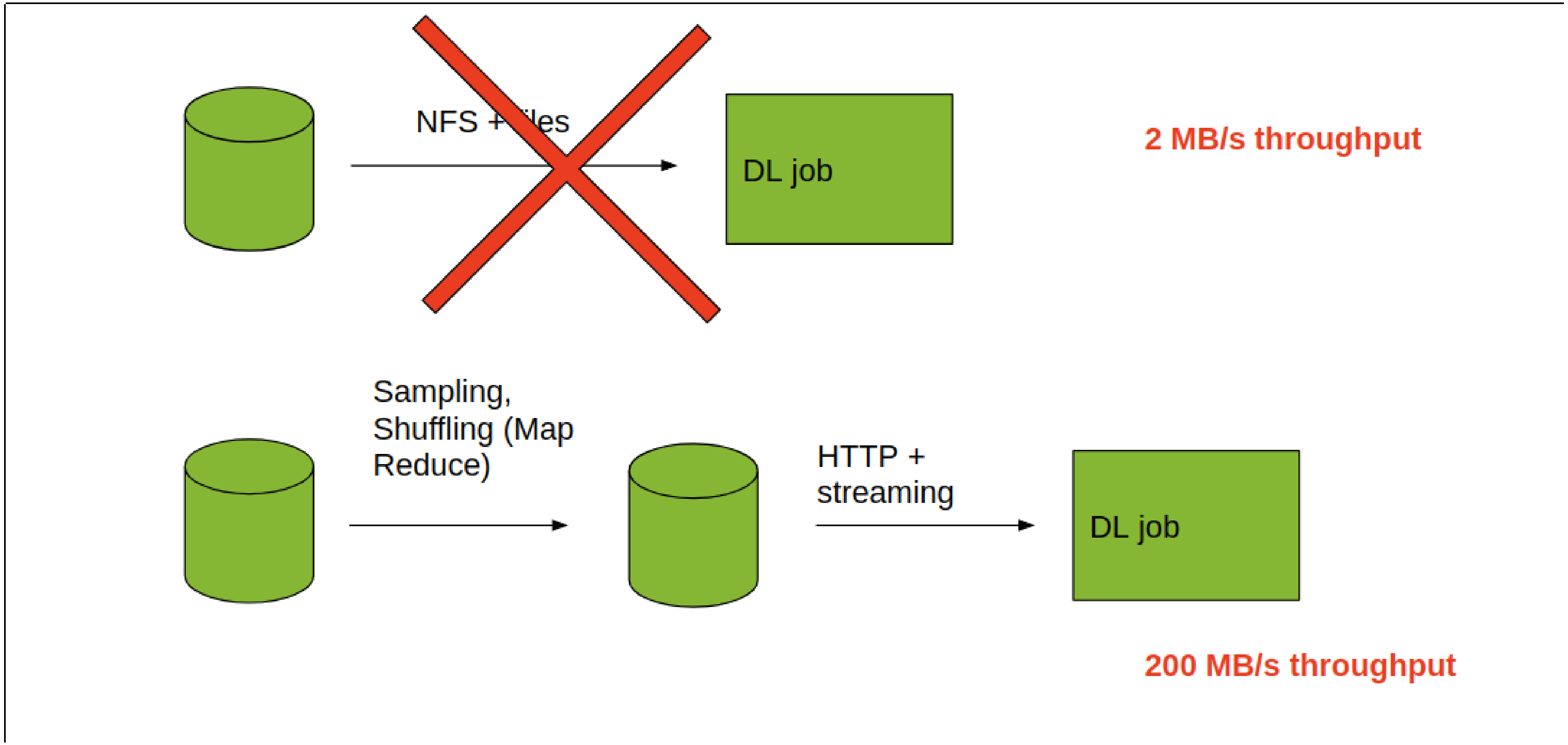
```
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device="cuda")
y = torch.randn(N, D_out, device="cuda")

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()
```

Speedup: Sequential I/O instead of Random Access



(See later.)

Linux Monitoring

Locally:

- htop, ... = watch processes
- iotop, ... = watch I/O
- nettop, ... = watch network I/O

Distributed:

- sensors in nodes/containers
- logging in log server
- visualization frontend

Recommendations

Max Out the Expensive Stuff:

- ensure that you are getting 90%+ GPU utilization for each GPU
- check your I/O bandwidth; it should be about 150 MB/s for disks, 3000 MB/s for NVMe

Avoid Maxing Out:

- GPU memory: this will kill your job
- CPU memory: limits GPU performance
- CPU utilization: limits GPU performance
- network bandwidth: limits I/O performance (for distributed I/O and SGD)

Some Options

- parallelize your model
- parallelize your model differently
- change random access I/O to sequential I/O
- use more CPU cores per GPU and use more I/O workers per GPU
- move I/O to separate node
- move data augmentation to separate node(s)
- use RDMA

(We will cover these later.)

FP16 Notebook

(notebook)