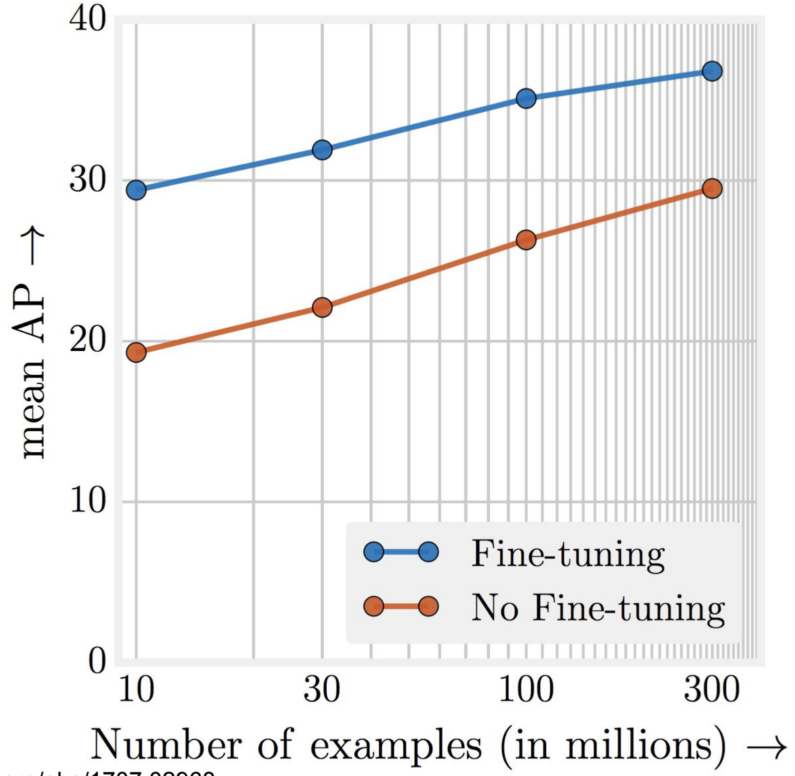
Large Scale DL for Everybody

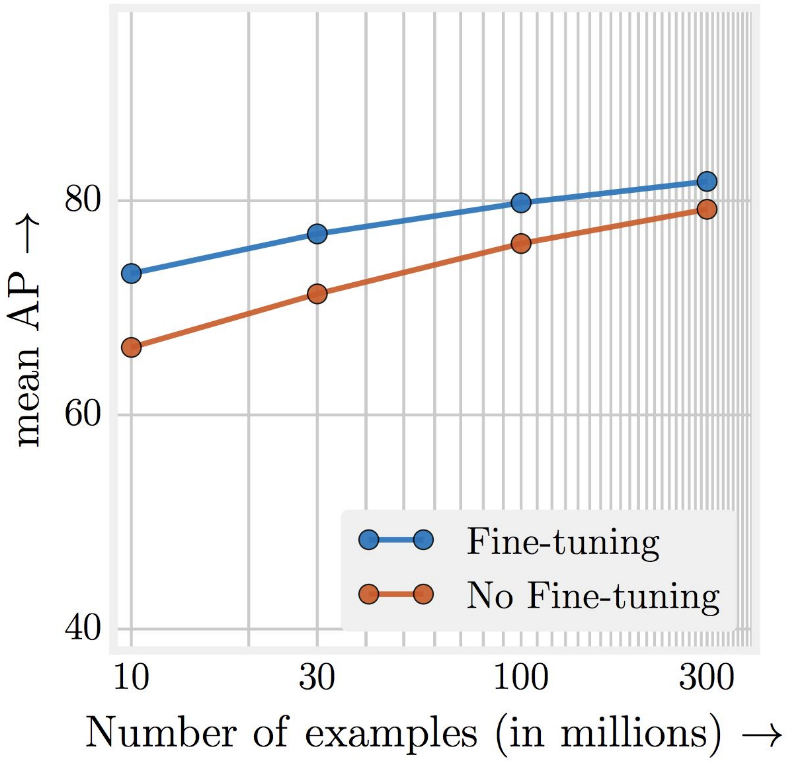
Thomas Breuel, NVIDIA Research

Why Large Scale DL?

Revisiting Unreasonable Effectiveness of Data in Deep Learning Era

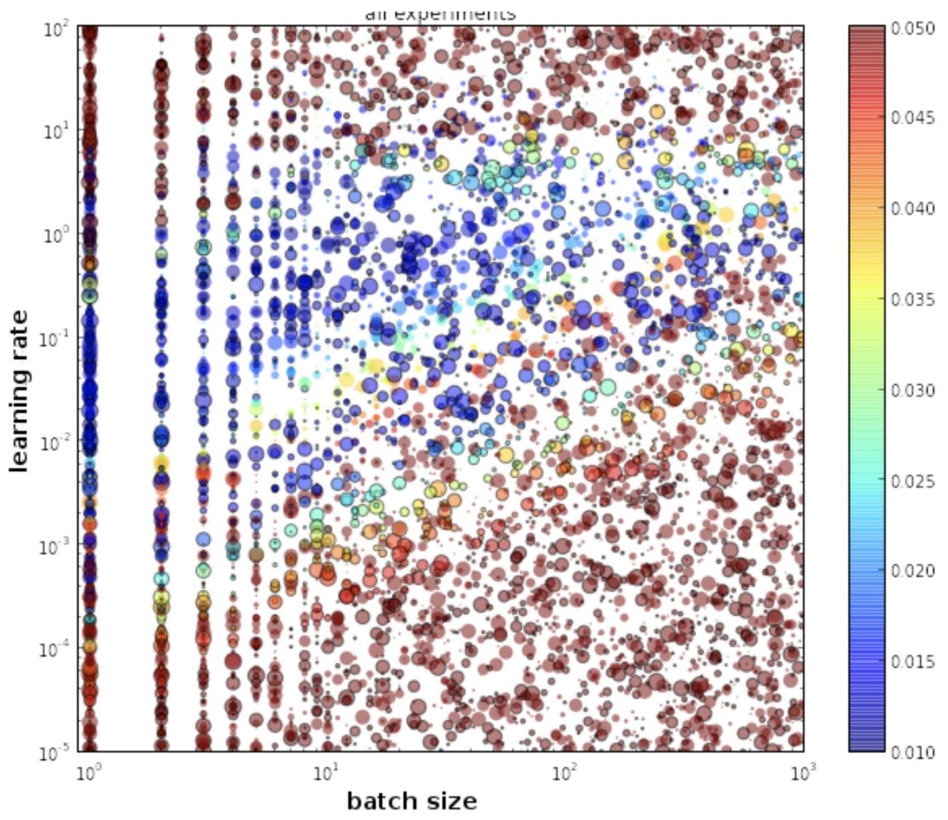
Chen Sun, Abhinav Shrivastava, Saurabh Singh , and Abhinav Gupta (Google Research)

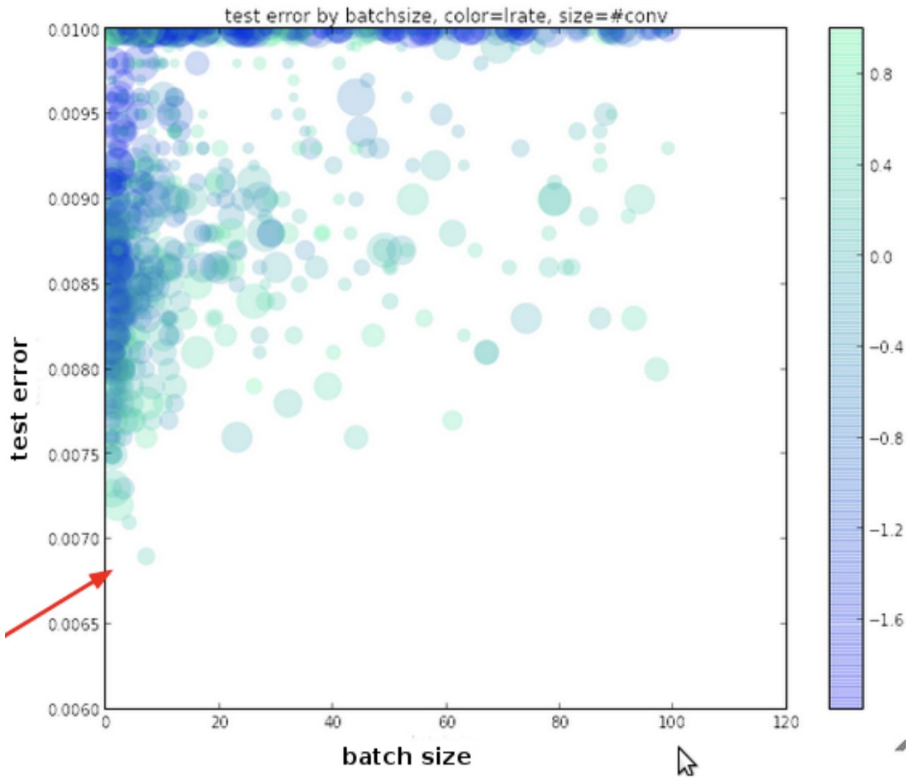




The Effects of Hyperparameters on SGD Training of Neural Networks

Thomas M. Breuel (Google Research)

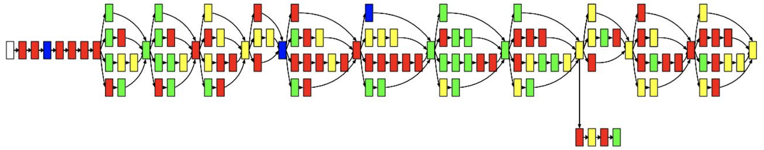


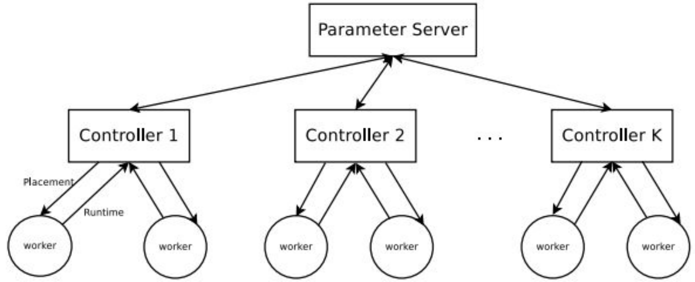


1024 GPUs for a few months

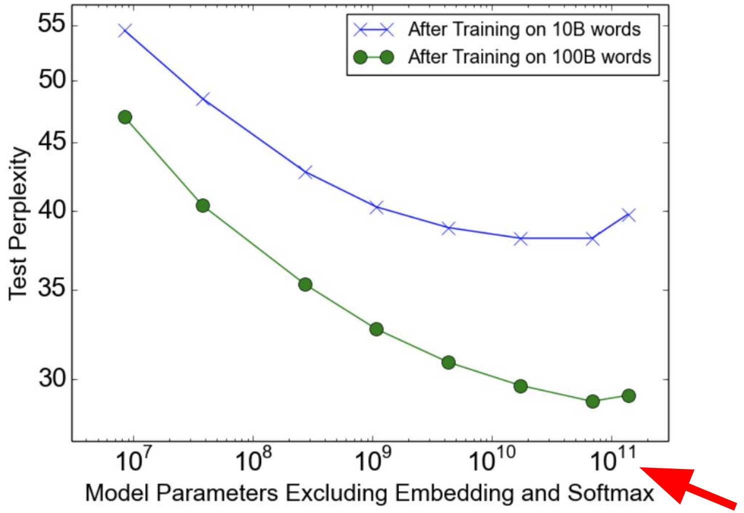
New Architectures

Mirhoseini et al.: *Device Placement with Reinforcement Learning*





Shazeer et al.: *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*



https://arxiv.org/abs/1706.04972

https://arxiv.org/abs/1701.06538

Large Scale DL

Big DL jobs at big companies:

- thousands of GPUs
- tens of thousands of cores
- thousands of nodes
- petabytes of training data

YT8m Dataset, a “Small Large Dataset”

8+ million YouTube videos, CC licensed

- 1 PB data
- 500000 h video
- 1 billion images (c.f. JFT-300m)

Labels: categories, captions, ...

YouTube 8m Data is Useful!

- static image or video training data
- source of training data for AV, speech, gesture, …
- large scale unsupervised learning
- new algorithms for large scale, distributed DL

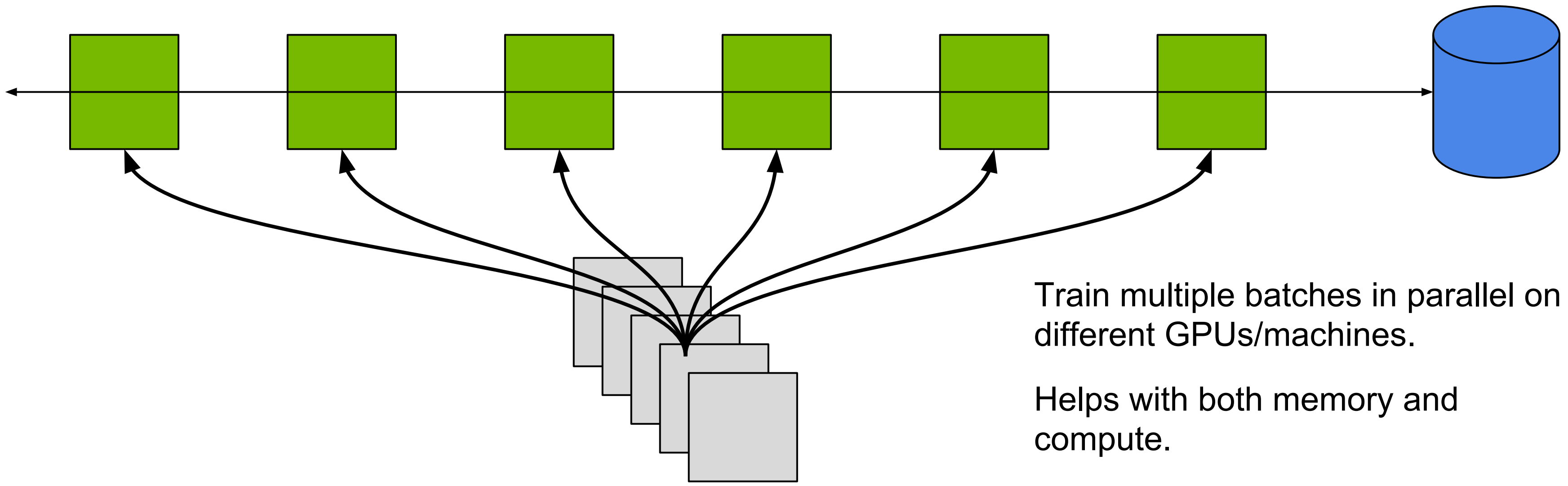
Goals for Today

What are the architectures?

Where are the bottlenecks?

What are the tools for large scale DL?

Multi-GPU and Multi-Node Training



Train multiple batches in parallel on different GPUs/machines.

Helps with both memory and compute.

Embarrassingly parallel.

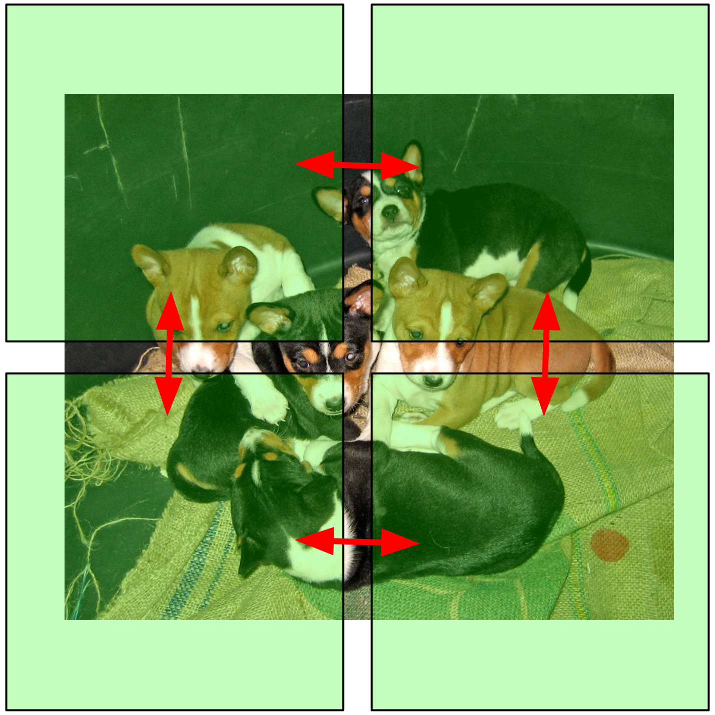
Communications costs dependent on # model parameters.

Synchronous or asynchronous updates; predictive updates.

Effective batch size = #machines * batch size

Data Parallelism

Multiple Towers / Batch Splitting across GPUs



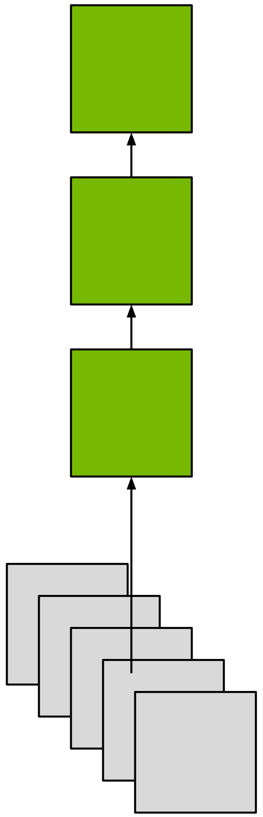
Efficient parallelism for 2D and 3D input data.

Helps with both memory and compute.

Modest bandwidth requirements across boundaries for conv layers.

Model Parallelism

Tiling with GPUs



Put first few layers on the first GPU, next few layers on the next GPU, etc.

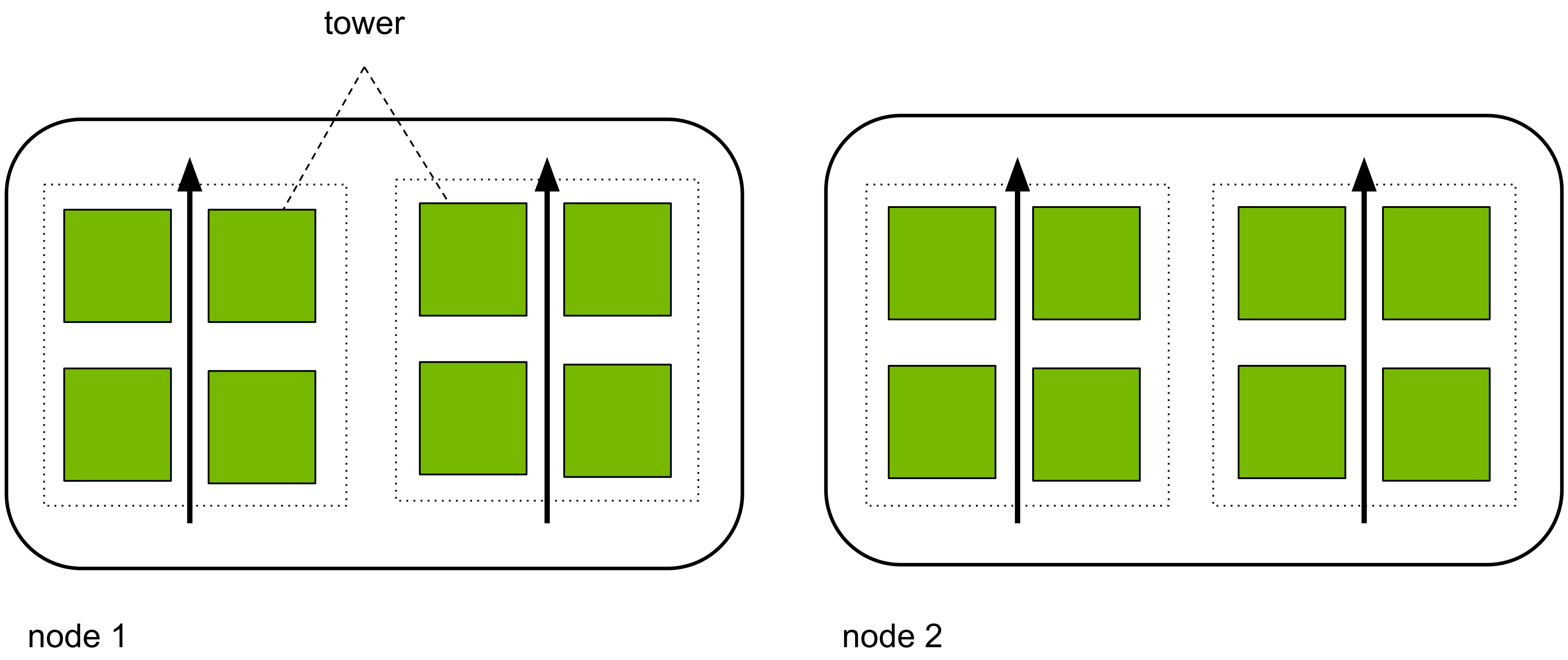
Very little parallelism for training (unless we use async updates).

Efficient pipelining for inference.

Helps with memory constraints.

Model Parallelism

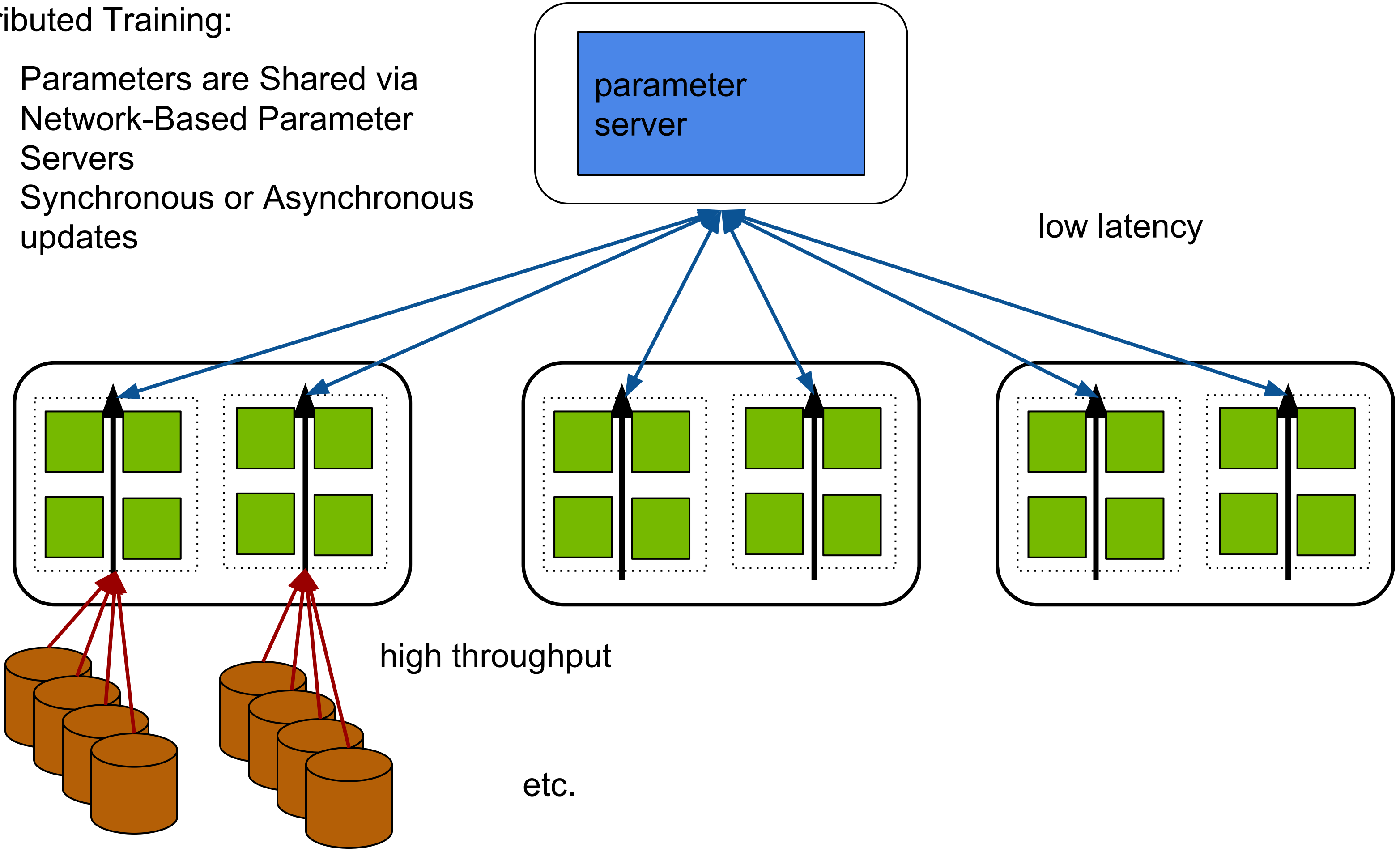
Pipelining / Vertical Splitting



Multi-GPU Patterns are Combined and Used Across Nodes

Distributed Training:

- Parameters are Shared via Network-Based Parameter Servers
- Synchronous or Asynchronous updates



Compute Graph to Nodes?

Vertical, horizontal, tiled… what is the best way of placing compute graphs on GPUs / nodes?

Approaches:

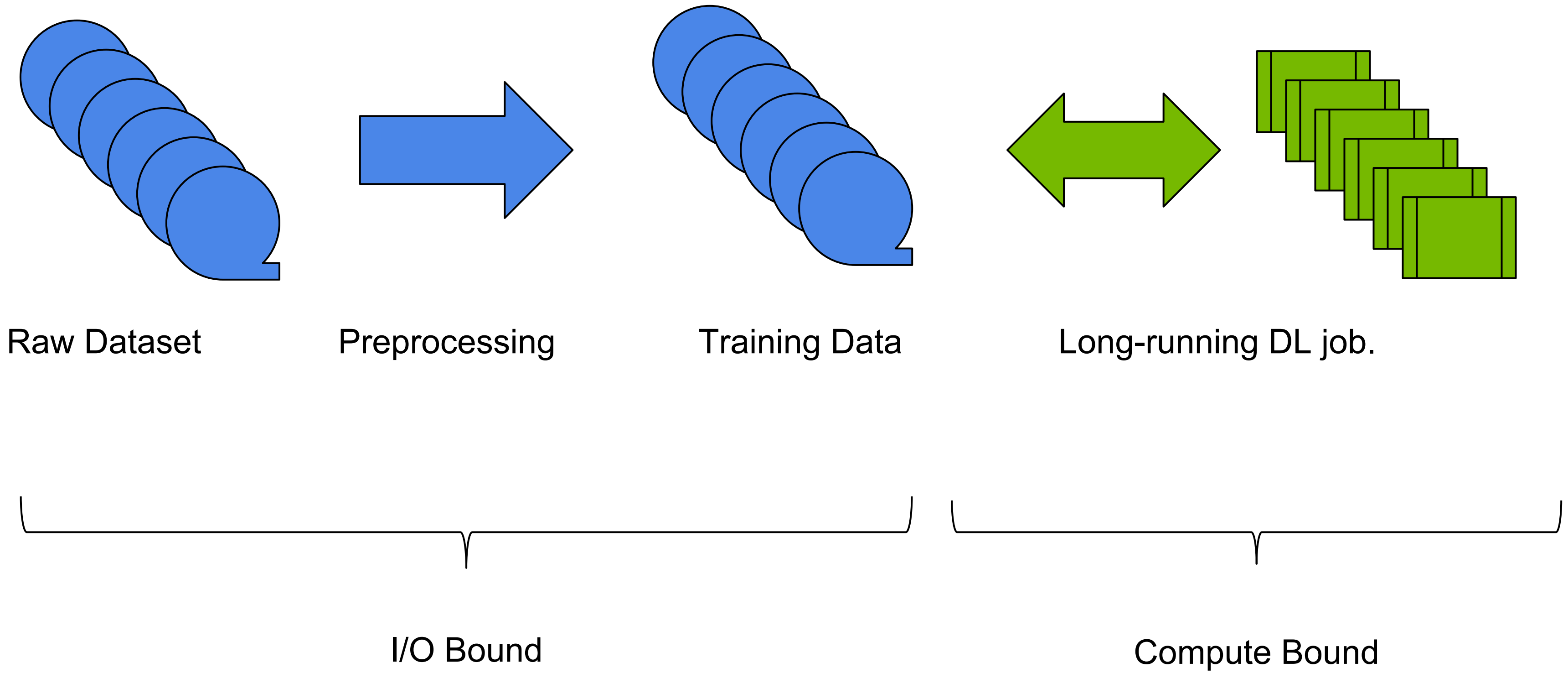
- extensive benchmarking + manual placement
- static optimization using min cut / max flow
- automatic placement via reinforcement learning or evolutionary algorithms
- dynamic utilization of resources

Other Multi-Node Jobs

Other Uses for Multi-Node Jobs

- large scale evaluation
- hyperparameter optimization
- genetic algorithms
- modular models
- local learning
- boosting
- format conversion
- shuffling
- transcoding
- inference
- data augmentation

Typical Large Scale DL Workflow



Common Tools

Commonly used tools for non-DL large scale jobs:

- Hadoop
- Spark
- Google Map Reduce
- Slurm + scripting

Training and Bottlenecks

Principle

GPU’s are the Ultimate Limit for DL Perf

$\Downarrow$

Aim for 100% GPU Utilization

CIFAR-10 / 100 Datasets

```
db = h5py.File('cifar10.h5')
images = np.array(db["images"])
classes = np.array(db["classes"])

cuimages = torch.FloatArray(images).cuda()
cuclasses = torch.LongArray(classes).cuda()

for i in range(0, len(images), 100):
    train_batch(images[i:i+100], classes[i:i+100])
```

- *< 8GB: keep on GPU*
- *<128GB: keep in RAM*
- *consider reducing precision*

100% GPU Utilization is Easy for Small Datasets

Imagenet

Size:

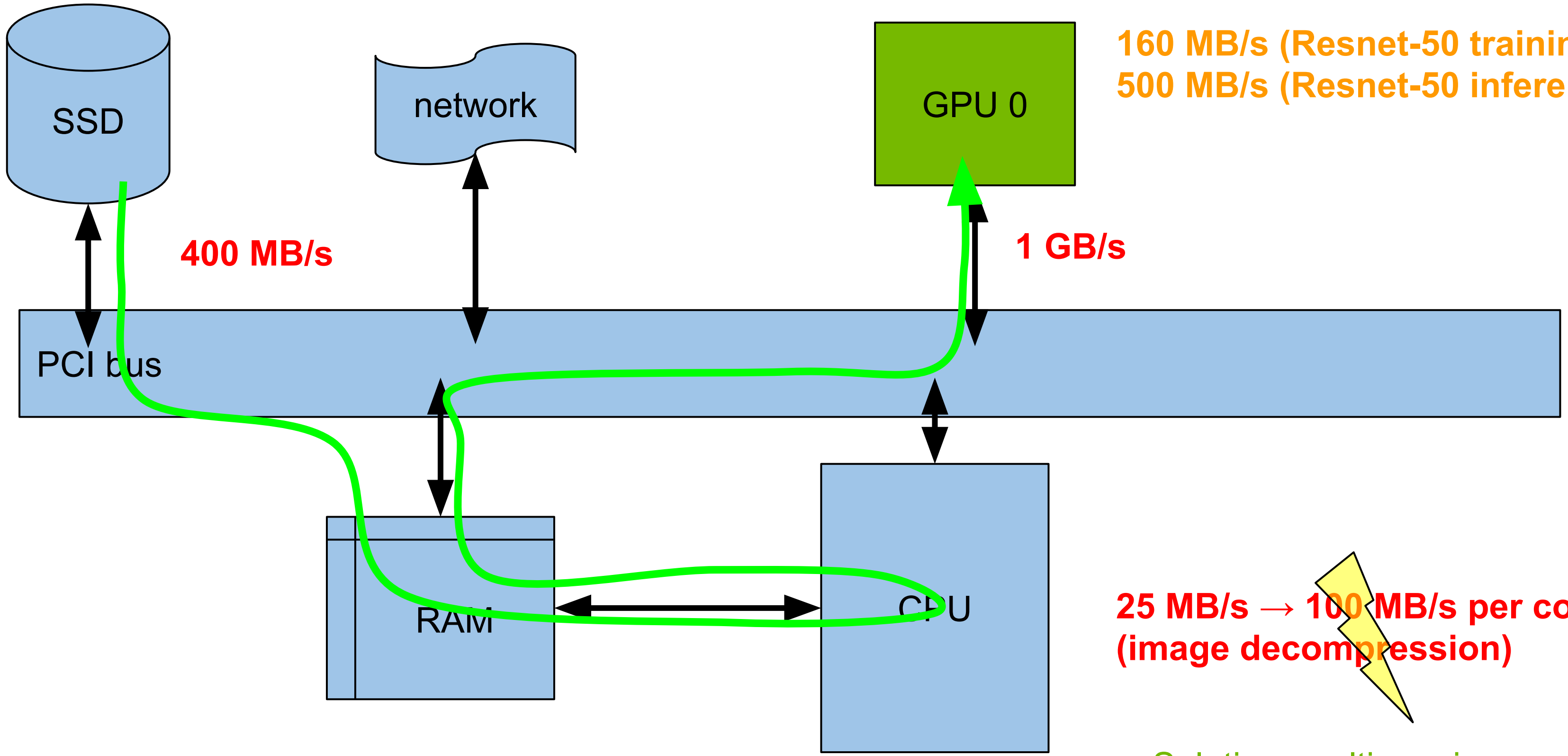
- about 150 Gbytes (slightly too large for memory)

Programming model:

- keep on local SSD
- load images from disk

Bandwidths of Different Devices

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



160 MB/s (Resnet-50 training)
500 MB/s (Resnet-50 inference

25 MB/s → 100 MB/s per core
(image decompression)

Solution: multicore image decompression

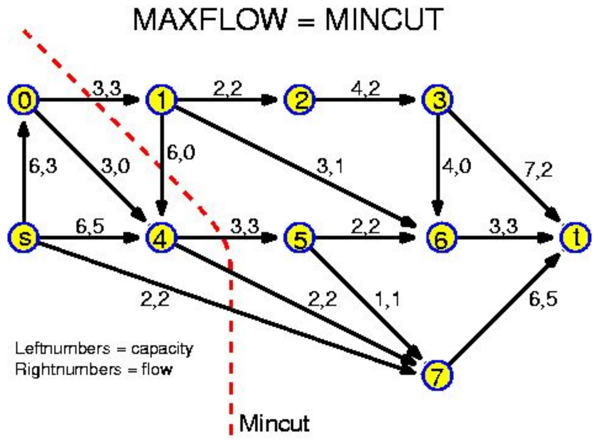
Datapath for local SSD training

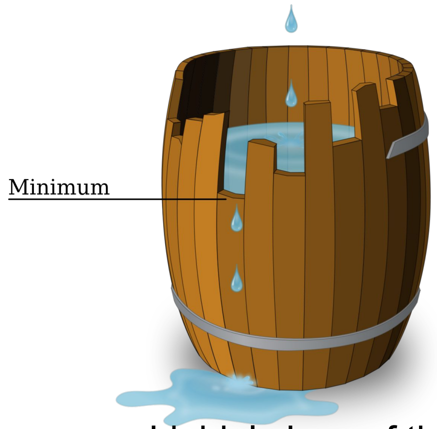
NB: All GPU-related figures are for PyTorch; they don't represent hardware limitations.

Max Flow Min Cut

A long-running deep learning network is like a big commodity graph with data flowing at it at different rates.

Max-Flow-Min-Cut Theorem: The maximum throughput is determined by the weakest link.





Liebig's Law of the Minimum

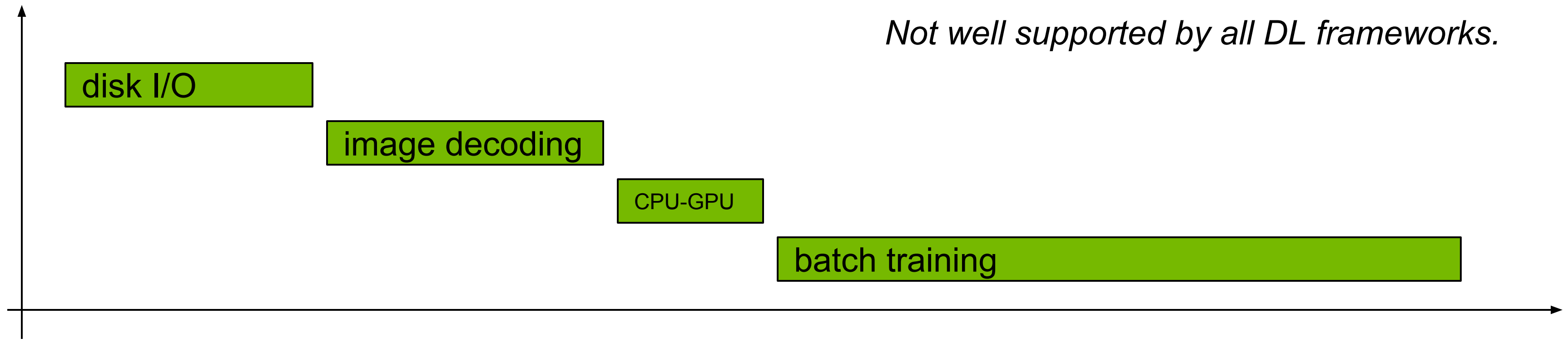
100% GPU Perf

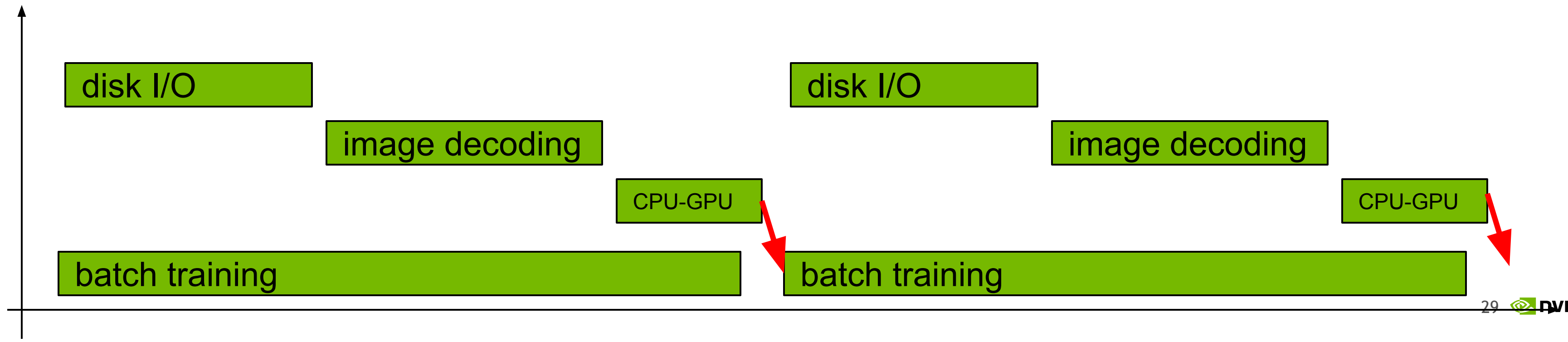
To get to 100% GPU Utilization…

… add as many CPUs, disk drives, and network cards as necessary.

Overlapping Operations

Not well supported by all DL frameworks.





Data Compression

Purpose of data compression:

- reduces amount of storage needed
- increases effective network transfer speeds
- native/archival format for many datasets

Limitations of data compression:

- often lossy, avoid multiple compression cycles
- recompression usually requires less aggressive compression
- cannot process / augment
- compute-intensive

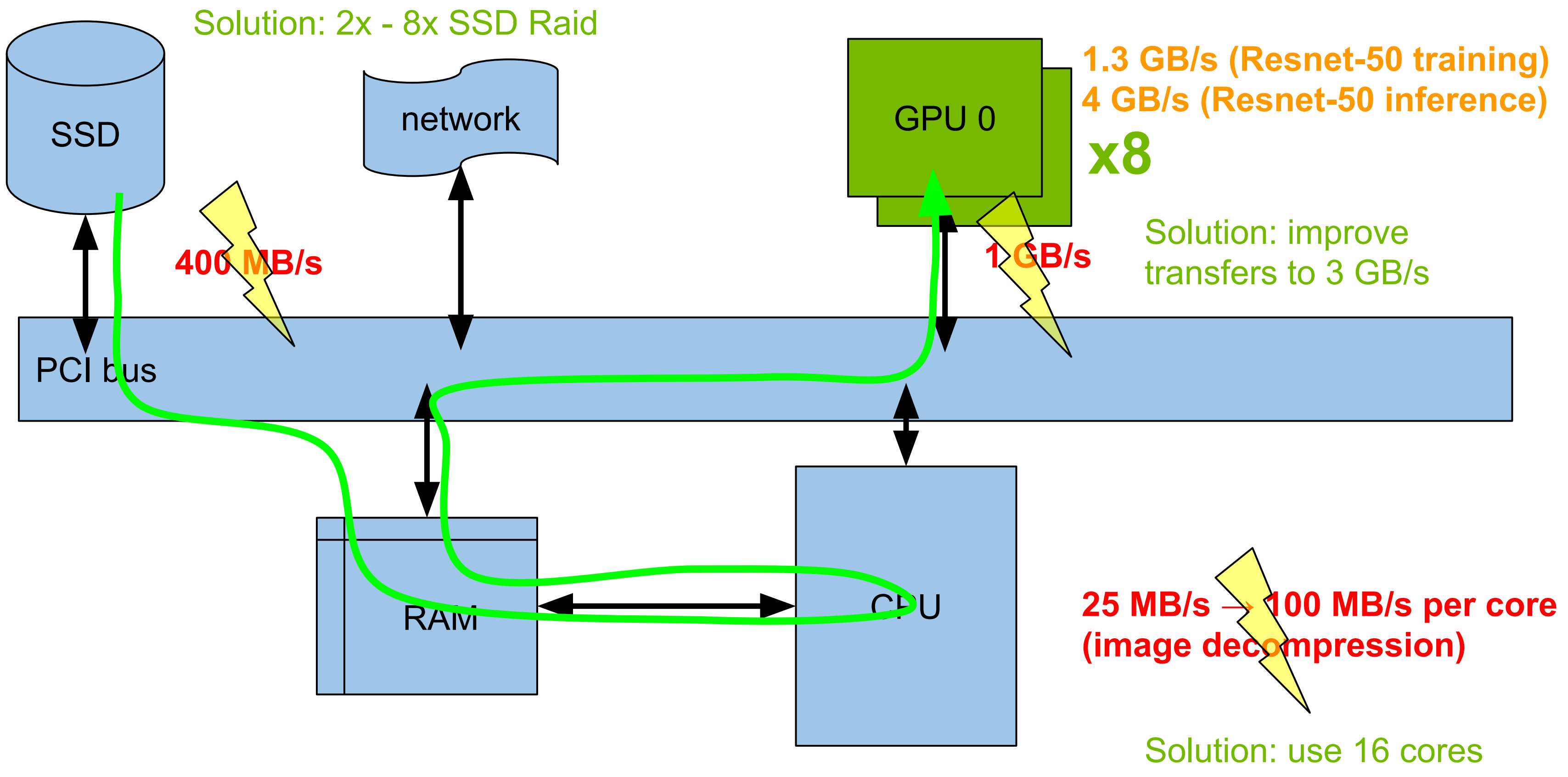
Uses of Data Compression

Use Data Compression:

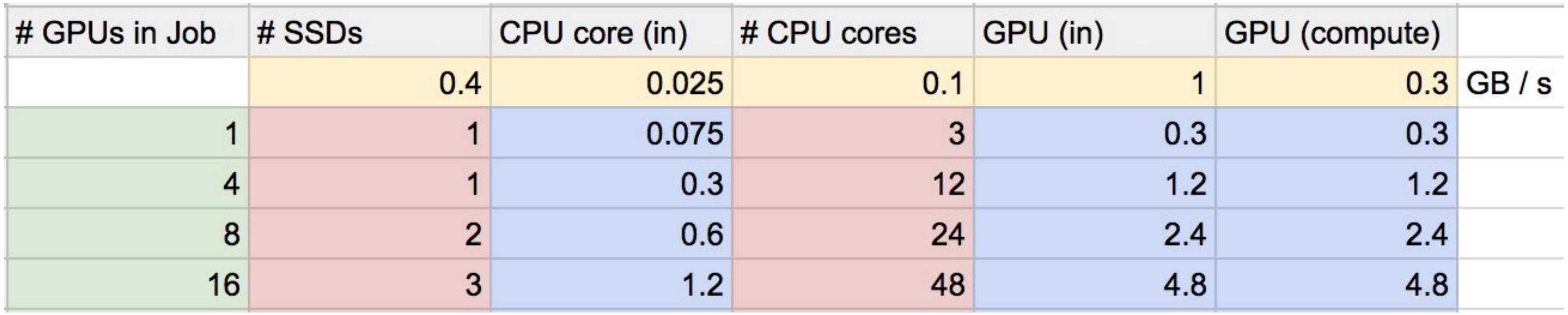
- archival storage
- transmission between racks

Use Uncompressed Tensors:

- CPU-to-GPU communications (unless decompressing on GPU)
- over NVLINK
- within-rack communications (sometimes)


Solution: 2x - 8x SSD Raid
SSD
network
GPU 0
1.3 GB/s (Resnet-50 training)
4 GB/s (Resnet-50 inference)
x8
400 MB/s
1 GB/s
Solution: improve transfers to 3 GB/s
PCI bus
RAM
CPU
25 MB/s → 100 MB/s per core (image decompression)
Solution: use 16 cores

Datapath for local SSD training, 8 GPUs



# GPUs in Job	# SSDs	CPU core (in)	# CPU cores	GPU (in)	GPU (compute)	
	0.4	0.025	0.1	1	0.3	GB / s
1	1	0.075	3	0.3	0.3	
4	1	0.3	12	1.2	1.2	
8	2	0.6	24	2.4	2.4	
16	3	1.2	48	4.8	4.8	

local storage, compressed images

Sequential vs Random Access

Rotational drives are much slower for random access than sequential access:

- read heads need to switch tracks while seeking
- within each track, they need to wait for sectors
- 200 MB/s for sequential reads, 20 MB/s for random access

Why not buy all SSD drives? They are much more expensive, both per TB and per bandwidth.

Slow vs Fast I/O

```
# 20 MB/s

files = open(“file_list”).read()...

for fname in files:
    image = imread(fname)
    ...
```

```
# 200 MB/s

records = RecordFile(fname)

for record in records:
    image = decode(record)
    ...
```

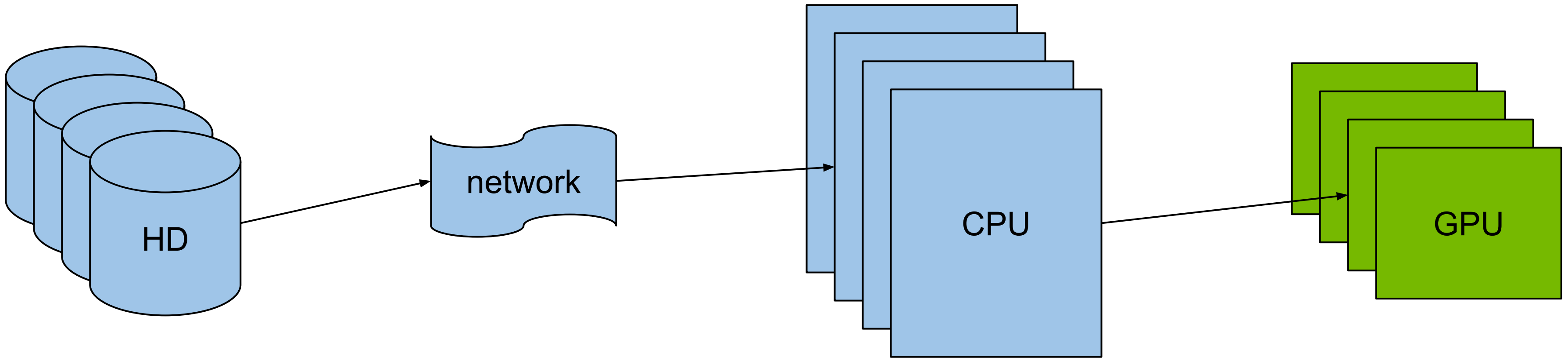
Slow vs Fast I/O

```
# 20 MB/s

open("/nfs/data/bigfile.tgz").read()
```

```
# 200 MB/s

open("/hd/data/bigfile.tgz").read()
```

Shared, multi-user NFS volumes tend to give you random access performance even for large sequential reads because the disks are used by multiple users together.

Object stores avoid this issue.



20 MB/s per HD
via NFS

1-5 GB/s per interface

25 MB/s / core (in)
100 MB/s / core (out)

1-3 GB/s per GPU

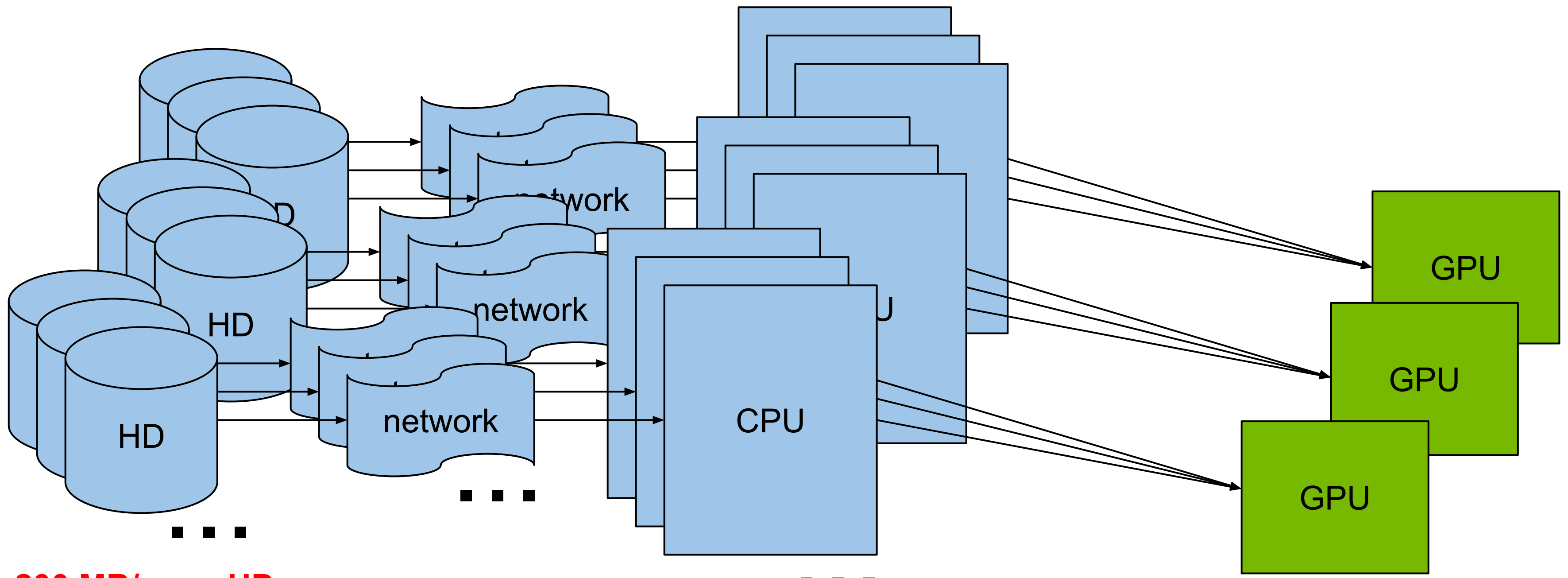
NB: 200 MB/s per HD
via object storage

0.1-1 GB/s per GPU

Distributed HD Training with NFS

# GPUs in Job	# dedicated HDs	# network interfaces	CPU core (in)	# CPU cores	GPU (in)	GPU (compute)	
	0.02	5	0.025	0.1	1	0.3	GB / s
1	4	1	0.075	3	0.3	0.3	
4	15	1	0.3	12	1.2	1.2	
8	30	1	0.6	24	2.4	2.4	
16	60	1	1.2	48	4.8	4.8	
64	240	1	4.8	192	19.2	19.2	
256	960	4	19.2	768	76.8	76.8	

NFS + compressed images



200 MB/s per HD via object storage

1-5 GB/s per interface

25 MB/s / core (in)
100 MB/s / core (out)

1 GB/s per GPU

Distributed HD Training with Object Storage:

sequential I/O; highly parallel, independent datapaths

NFS

# GPUs in Job	# dedicated HDs	# network interfaces	GPU (in)	GPU (compute)	
	0.02	5	1	0.3	GB / s
1	15	1	0.3	0.3	
4	60	1	1.2	1.2	
8	120	1	2.4	2.4	
16	240	1	4.8	4.8	
64	960	4	19.2	19.2	
256	3840	16	76.8	76.8	

Object Store

# GPUs in Job	# dedicated HDs	# network interfaces	GPU (in)	GPU (compute)	
	0.2	5	1	0.3	GB / s
1	2	1	0.3	0.3	
4	6	1	1.2	1.2	
8	12	1	2.4	2.4	
16	24	1	4.8	4.8	
64	96	4	19.2	19.2	
256	384	16	76.8	76.8	

distributed training, uncompressed data

I/O For Deep Learning

I/O For Deep Learning

Access Pattern for DL Jobs:

- repeatedly read the entire data set (100s of epochs)
- shuffle during reading
- I/O rate of each GPU exceeds the I/O rate of HDs

Implications:

- caching is useless when dataset is larger than case
- need multiple drives per GPU
- don’t care about latency, but do care about throughput

Efficient I/O Subsystem for DL

- need: parallel I/O
  - split large datasets up into shards (typically 100MB - 10GB)
  - read in parallel from several drives at once

- need: random shuffling, but also large sequential reads
  - read shards in random order
  - read each shard sequentially and shuffle in memory

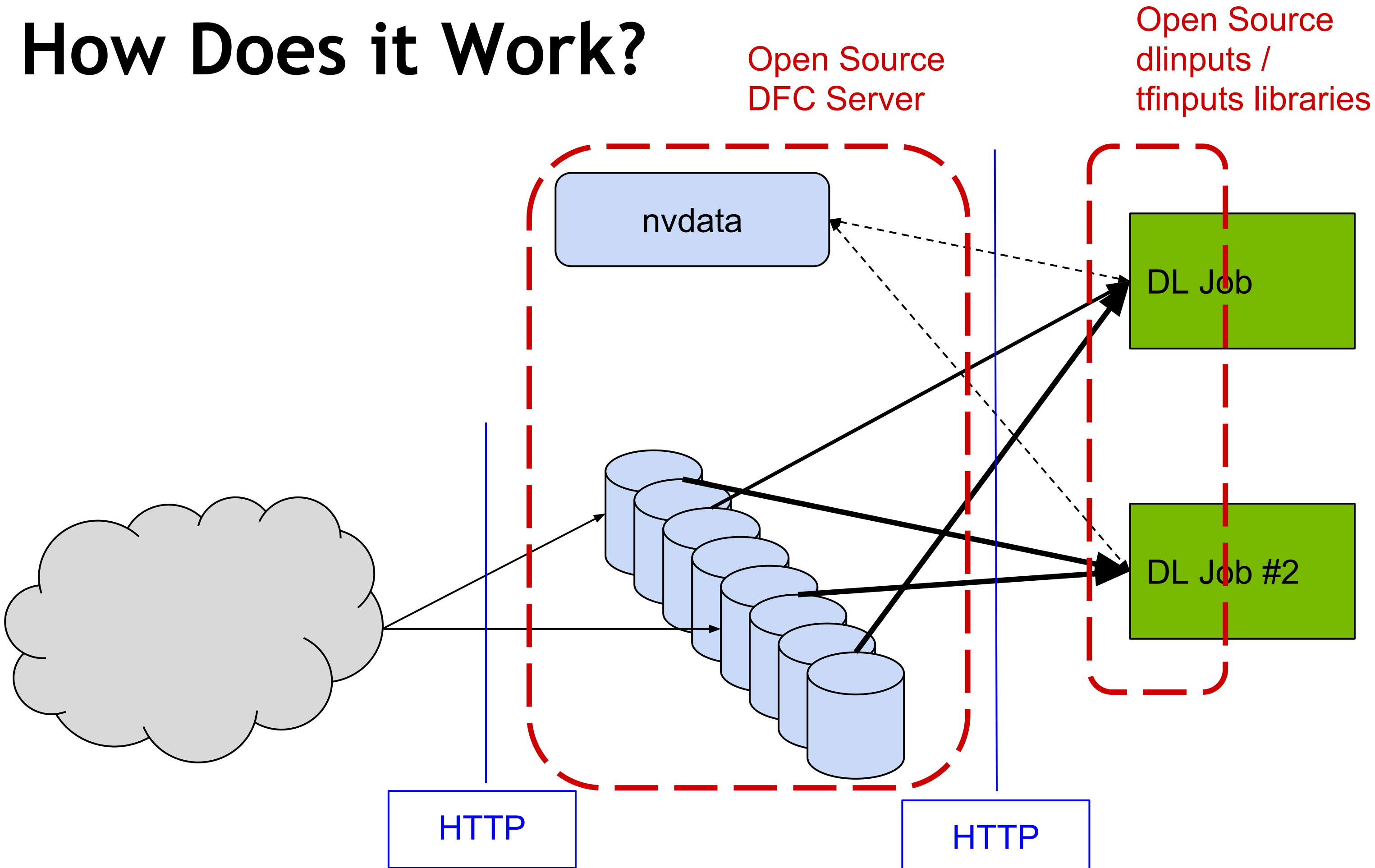
It’s Simple!

```
from dlinputs import gopen

dataset = gopen.sharditerator(“http://nvdata/imagenet-@001000.tgz”)

for sample in dataset:
    train_sample(sample[“png”], sample[“cls”])
```

It’s Simple!

```
from dlinputs import gopen
import dlinputs.filters as df

dataset = gopen.sharditerator(“http://nvdata/imagenet-@001000.tgz”)

pipeline = df.compose(df.shuffle(1000), df.batch(20))

for batch in pipeline(dataset):
    train_batch(batch[“png”], batch[“cls”])
```

How Does it Work?



Record Sequential Storage

Sharding requires a sequential record file format.

Common choices:

- tf.Record / tf.Example
- Hadoop Sequence Files
- POSIX tar files

POSIX tar files for Training

Put your dataset on a server:

```
$ tar -cf dataset.tar dataset
$ curl -T dataset.tar http://nvdata/dataset.tar
```

Sharded datasets:

```
$ tar -cf - dataset | tarshards http://nvdata/dataset-%06d.tar
```

POSIX tar files for Datasets

- standard format, lots of tools and libraries
- no need for format conversions, just tar up your data
- same format for PyTorch and TensorFlow
- data is stored/transmitted in compressed form
- your training data is also a full backup of your dataset
- key-value format of POSIX tar supported by DFC for MapReduce ops

You're probably already using them, just not for training!

Tools

Kubernetes

Kubernetes = container orchestr

deals with

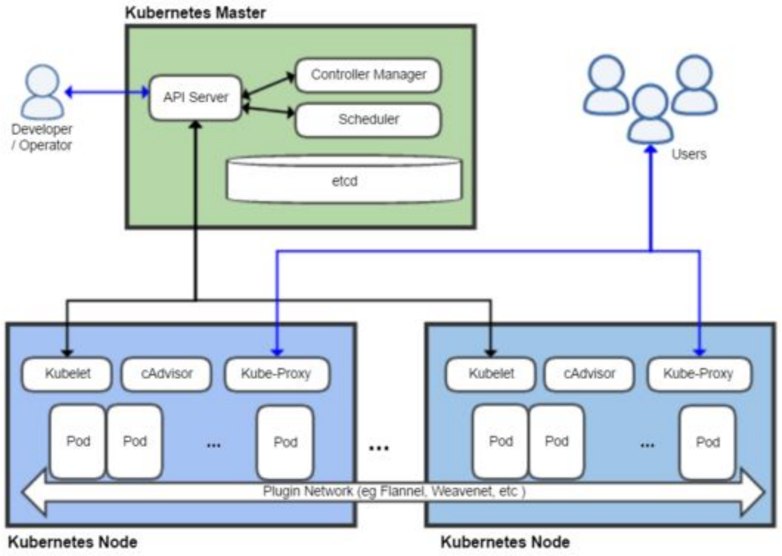
- failures / restarts
- namespaces / friendly isolatio
- virtual networking

runs on

- virtual machines in the cloud
- local bare metal (and integrat

does not provide

- bullet-proof security, resource limits

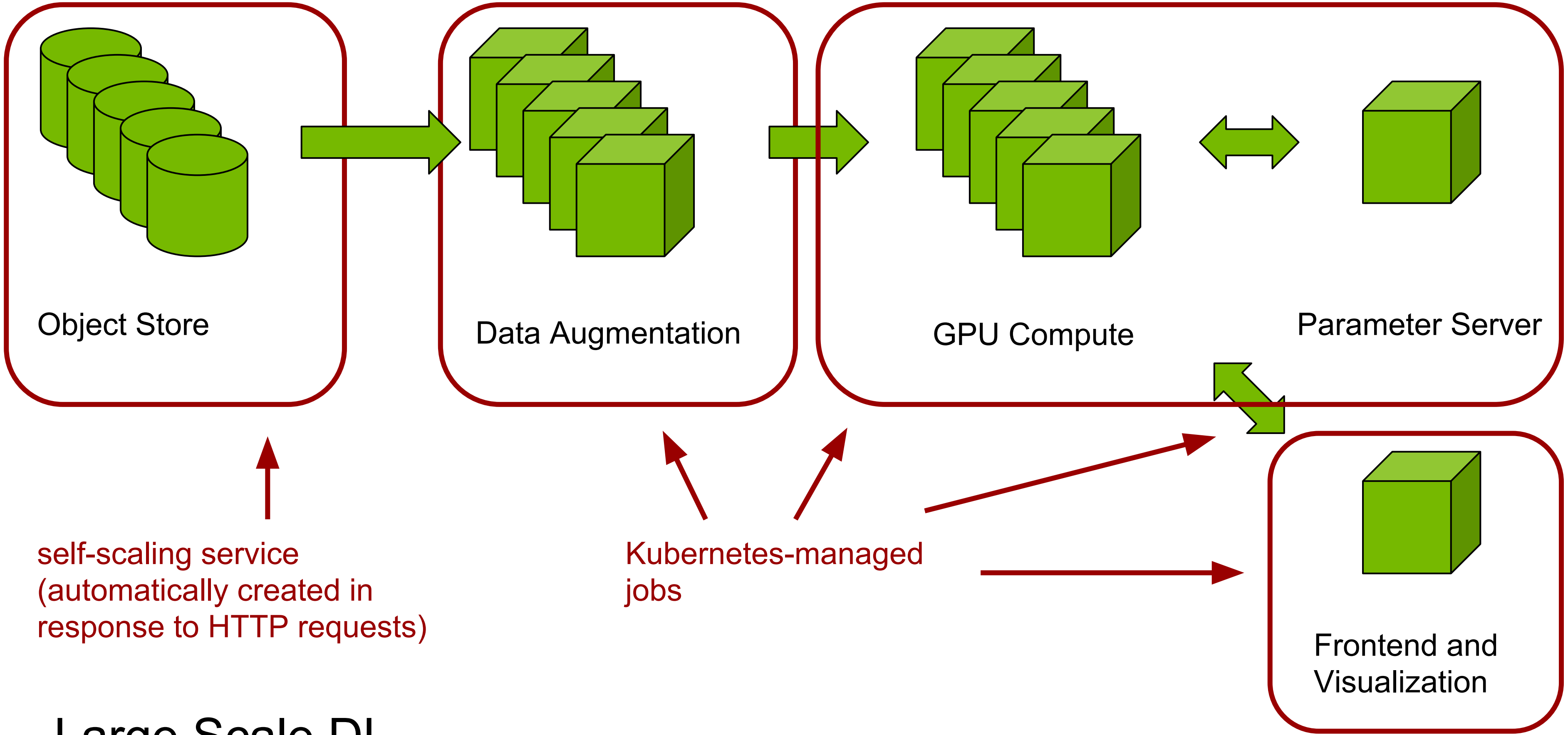


Kubernetes

Kubernetes

=

A Programming Language for Writing Distributed Applications


Object Store
Data Augmentation
GPU Compute
Parameter Server
Frontend and Visualization
self-scaling service (automatically created in response to HTTP requests)
Kubernetes-managed jobs

Large Scale DL

Components of a DL Job

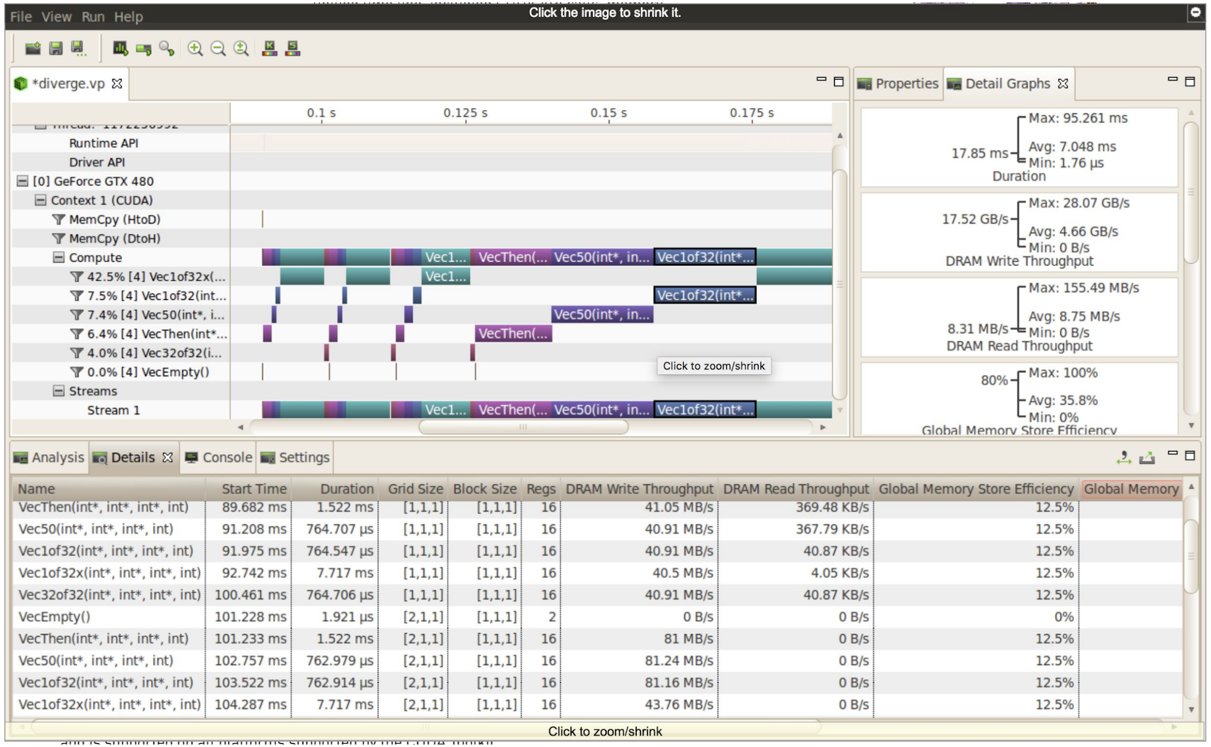
Common Components of a DL Job:

- interactive Jupyter notebook frontend
- TensorBoard data visualization
- CPU-based data loader and augmentation
- GPU-based DL training job
- parameter server for distributed SGD
- REDIS server for hyperparameter search

These are scheduled as Pod, Job, and ReplicaSet on Kubernetes.

Some are scheduled on CPU, others on GPU.

NVIDIA nSight Profiler



Name	Start Time	Duration	Grid Size	Block Size	Regs	DRAM Write Throughput	DRAM Read Throughput	Global Memory Store Efficiency	Global Memory
VecThen(int*, int*, int*, int)	89.682 ms	1.522 ms	[1,1,1]	[1,1,1]	16	41.05 MB/s	369.48 KB/s	12.5%	
Vec50(int*, int*, int*, int)	91.208 ms	764.707 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	367.79 KB/s	12.5%	
Vec1of32(int*, int*, int*, int)	91.975 ms	764.547 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
Vec1of32x(int*, int*, int*, int)	92.742 ms	7.717 ms	[1,1,1]	[1,1,1]	16	40.5 MB/s	4.05 KB/s	12.5%	
Vec32of32(int*, int*, int*, int)	100.461 ms	764.706 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
VecEmpty()	101.228 ms	1.921 µs	[2,1,1]	[1,1,1]	2	0 B/s	0 B/s	0%	
VecThen(int*, int*, int*, int)	101.233 ms	1.522 ms	[2,1,1]	[1,1,1]	16	81 MB/s	0 B/s	12.5%	
Vec50(int*, int*, int*, int)	102.757 ms	762.979 µs	[2,1,1]	[1,1,1]	16	81.24 MB/s	0 B/s	12.5%	
Vec1of32(int*, int*, int*, int)	103.522 ms	762.914 µs	[2,1,1]	[1,1,1]	16	81.16 MB/s	0 B/s	12.5%	
Vec1of32x(int*, int*, int*, int)	104.287 ms	7.717 ms	[2,1,1]	[1,1,1]	16	43.76 MB/s	0 B/s	12.5%	

NVIDIA Open Source Projects

- **dlinputs** - easy parallel I/O for PyTorch
- **tfinputs** - easy parallel I/O for TensorFlow
- **DFC** - high performance distributed web server for deep learning
- **nvimagenet** - scalable Imagenet training based on PyTorch (sample)

(On GitHub but not officially released yet.)

TODO

TODO

- Get to 100% GPU usage
- Find your bottlenecks
- Profile and benchmark
- Move to bigger datasets
- Use Docker and prepare for Kubernetes

In a few months, give **dlinputs** and **DFC** a try (or if you’re adventurous, give it a try already).