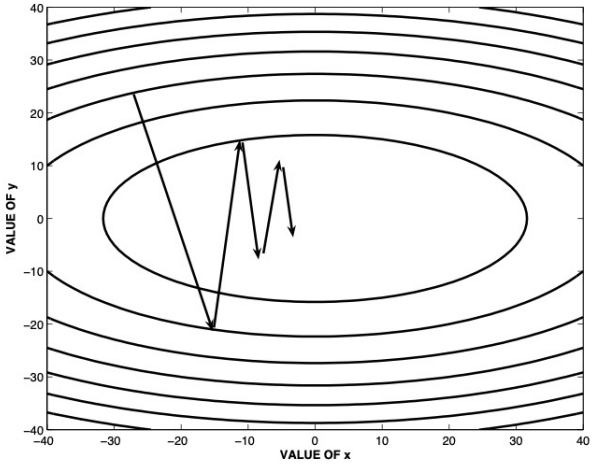
DS-UA 301
Advanced Topics in Data Science
Advanced Techniques in ML and Deep Learning

LECTURE 10

Parijat Dube

Gradient Descent Convergence



(b) Loss function is elliptical bowl

$$L = x^2 + 4y^2$$

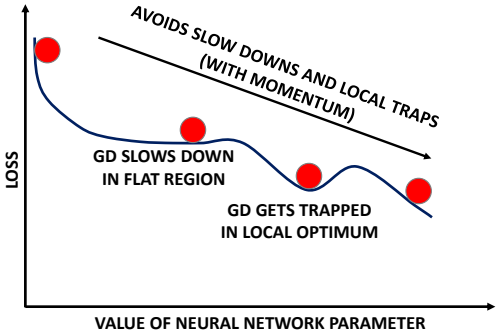
- GD convergence is poor due to difference in gradient values along different dimensions
- Effective descent direction gets away from the minima if we use finite learning rate
- Gradient descent might also get trapped at saddle points and/or local minima

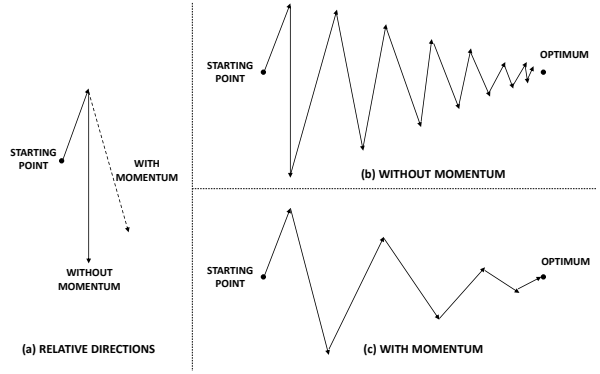
We need to :

– Move quickly in directions with small but consistent (pointing in one direction, +ve or -ve) gradients.

– Move slowly in directions with big but inconsistent (oscillating between –ve and +ve) gradients.

Gradient Descent with Momentum





- Add momentum to GD updates:

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

- Learning is accelerated as oscillations are damped and updates progress in the consistent directions of loss decrease
- Enables working with large learning rate values and hence faster convergence

Nesterov Momentum

- Simple momentum-based updates cause solution to overshoot the target minima
- Idea is to use some lookahead in computing the updates

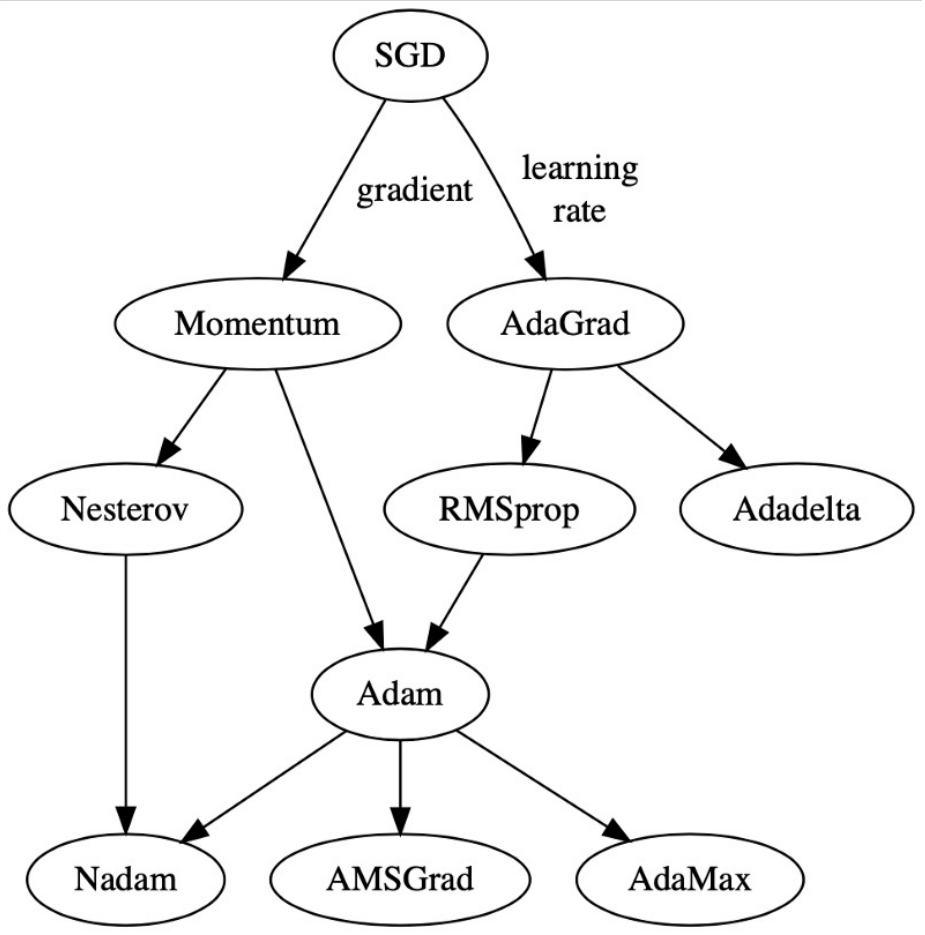
$$\overline{V} \Leftarrow \underbrace{\beta \overline{V}}_{\text{Momentum}} - \alpha \frac{\partial L(\overline{W} + \beta \overline{V})}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

- Put on the brakes as the marble reaches near bottom of hill.
- Difference from standard momentum method in terms of where the gradient is computed.

Parameter-specific learning rates

- Apply a different learning rate to each parameter at each step
- Encourage faster relative movement in gently sloping direction
- Penalize dimension with large fluctuations in gradient
- Several methods: AdaGrad, RMSProp, RMSProp+Nestrov Momentum, AdaDelta, Adam
  - Differ in the manner parameter specific learning rates are calculated

Evolutionary Map of Optimizers

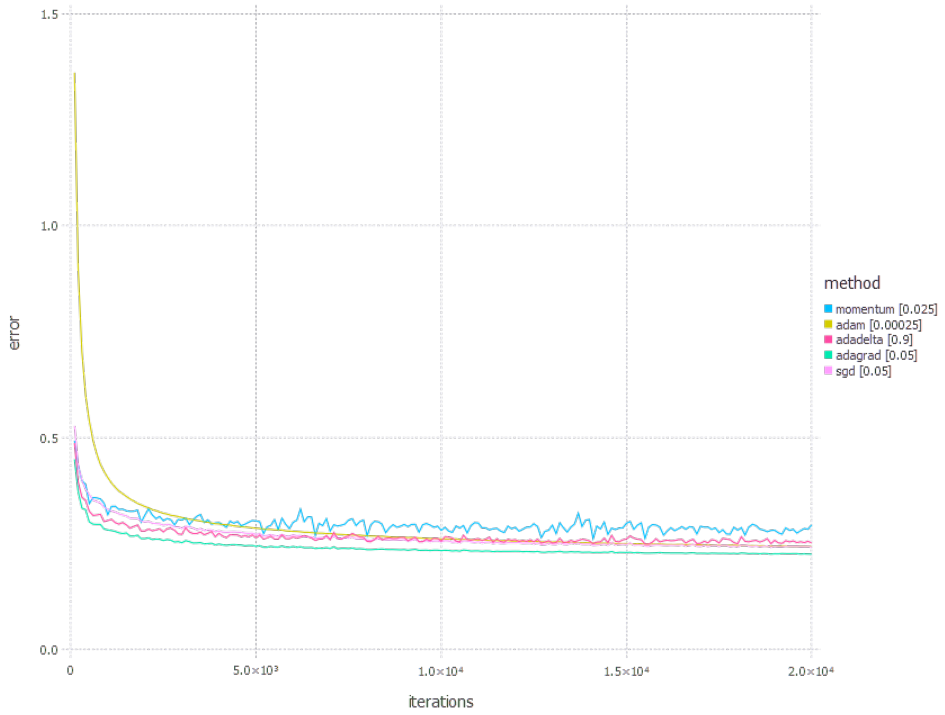


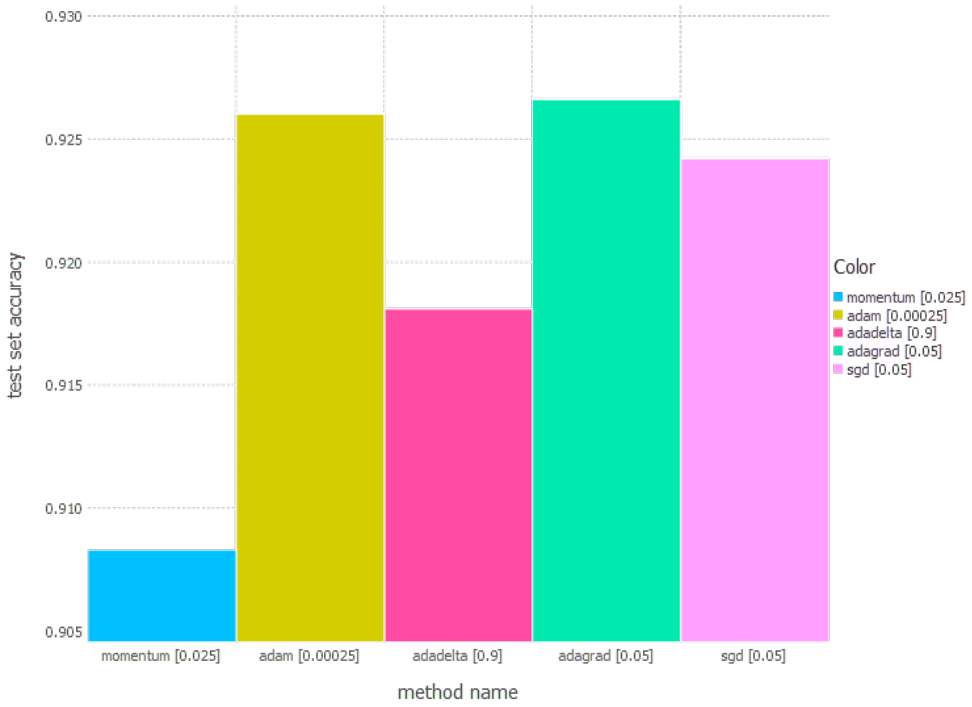
Optimization Techniques Comparison (I)

- Toy example on MNIST Classification
- Three different network architecture tested:
  1. network with linear layer and softmax output (softmax classification)
  2. network with sigmoid layer (100 neurons), linear layer and softmax output
  3. network with sigmoid layer (300 neurons), ReLU layer (100 neurons), sigmoid layer (50 neurons) again, linear layer and softmax output
- Mini-batch size of 128
- Run the algorithm for approx. 42 epochs (20000 iterations)
- http://int8.io/comparison-of-optimization-techniques-stochastic-gradient-descent-momentum-adagrad-and-adadelta/

Results on net 1

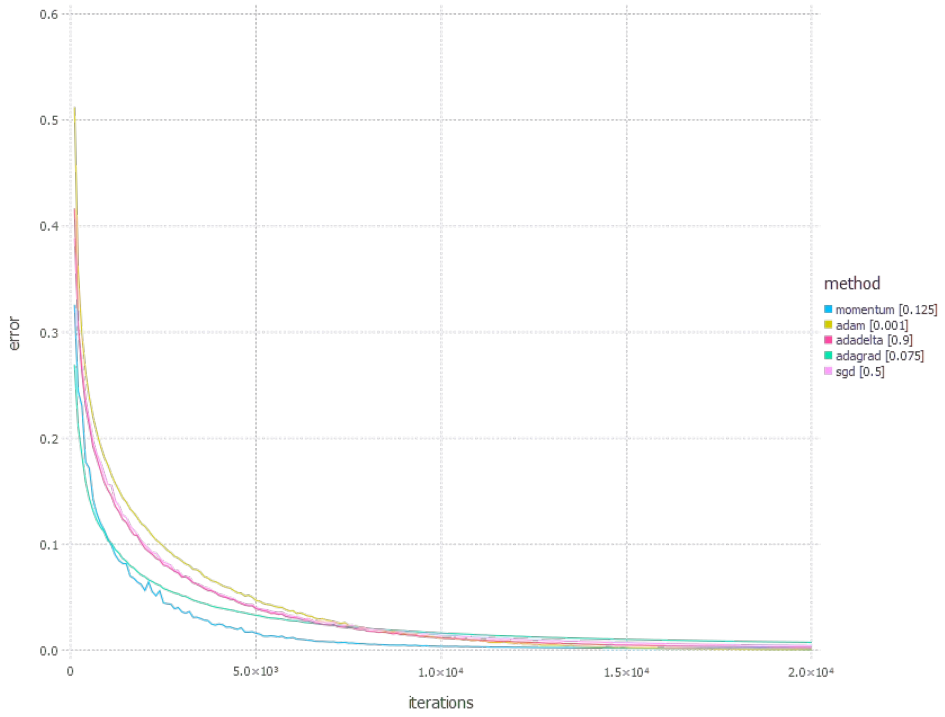
network with linear layer and softmax output (softmax classification)

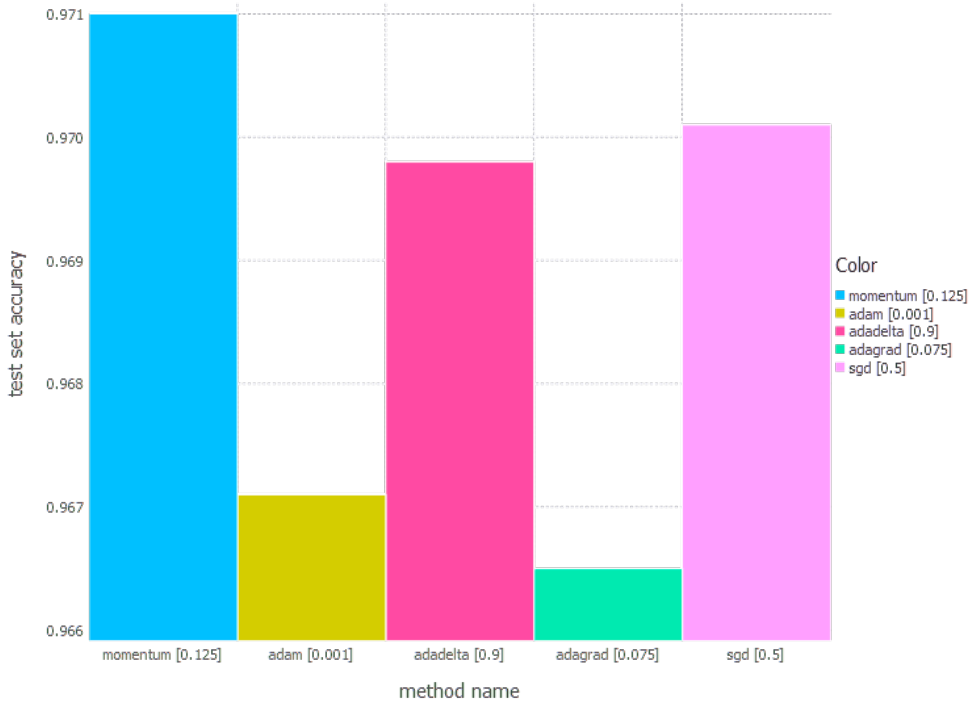




Results on net 2

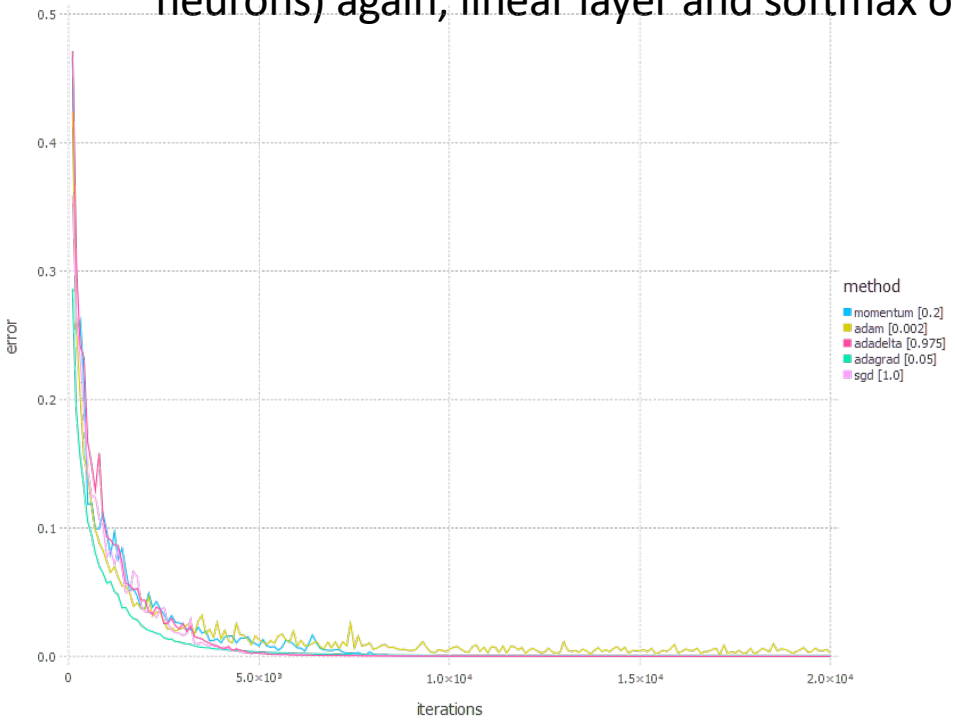
network with sigmoid layer (100 neurons), linear layer and softmax output

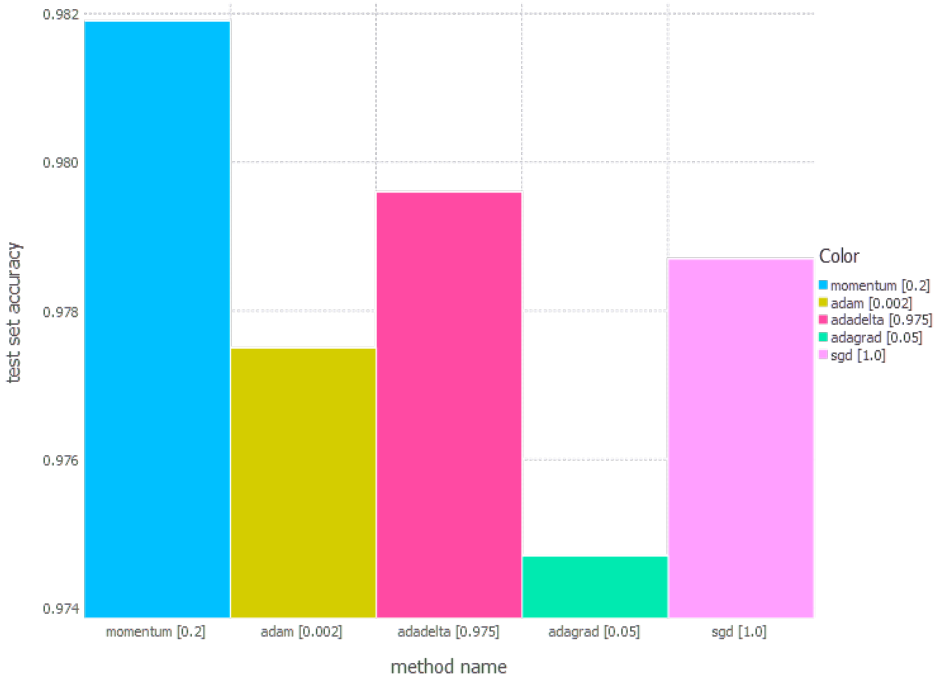




Results on net 3

network with sigmoid layer (300 neurons), ReLU layer (100 neurons), sigmoid layer (50 neurons) again, linear layer and softmax output





So….Which optimizer to use?

- Unfortunately there isn't a clear answer
- As in the toy example before, different networks have completely different behaviors
- If your input data is **sparse**…?
  - You likely achieve the best results using one of the **adaptive learning-rate** methods
  - An additional benefit is that you will not need to tune the learning rate
- Which one is the best?
  - Adagrad, Adadelta, and Adam are very similar algorithms that do well in similar circumstances
  - Insofar, **Adam** might be the best overall choice as trade off between time and accuracy
- Interestingly, many recent papers use SGD without momentum and a simple learning rate annealing schedule
  - SGD usually achieves to find a minimum but it takes much longer time than others, is much more reliant on a robust initialization and annealing schedule

  → If you care about **fast convergence** and train a deep or complex NN, you should choose one of the **adaptive learning** rate methods

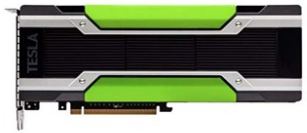
Comments on optimizer

- All the optimizers take in the parameters and gradients and manipulate over them.
- None of the optimizers are theoretically proven to have better convergence rate than SGD, except Nesterov momentum.
- Even Nesterov momentum is only proved to work on strongly convex problems.
- In practice, SGD with momentum and a per-iteration learning rate schedule often produce better final accuracy with some tuning than automatic update algorithms.

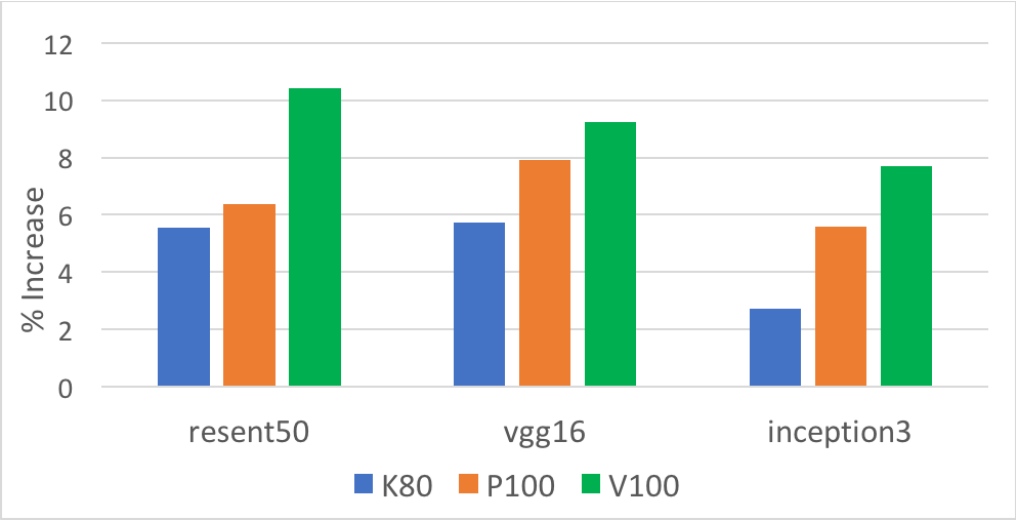
Single Node, Single GPU Training

- Training throughput depends on:
  - Neural network model (activations, parameters, compute operations)
  - Batch size
  - Compute hardware: GPU type (e.g., Nvidia M60, K80, P100, V100)
  - Floating point precision (FP32 vs FP16)
    - Using FP16 can reduce training times and enable larger batch sizes/models without significantly impacting the accuracy of the trained model
- Increasing batch size increases throughput
  - Batch size is restricted by GPU memory
- Training time with single GPU very large: 6 days with Places dataset (2.5M images) using Alexnet on a single K40.
- Small batch size => noisier approximation of the gradient => lower learning rate => slower convergence

Commonly used GPU Accelerators in Deep Learning

Nvidia M60		2 GPUs with 4.8 TFLOPS SP and 8 GB Cache each
Nvidia K80		2 GPUs with 3 TFLOPS SP and 12 GB Cache each
Nvidia P100		1 GPU with 9 TFLOPS SP and 16 GB Cache
Nvidia V100		1 GPU with 14 TFLOPS SP and 16 GB Cache

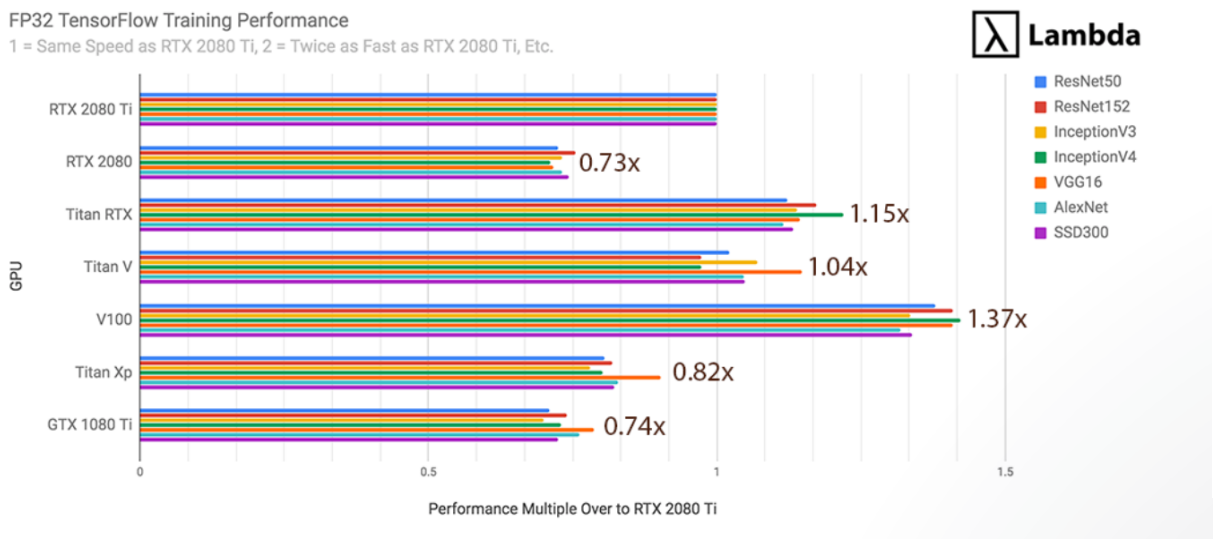
DL Scaling with Batch size on Single GPU



Training performance speed-up of DL models at batch size 64 compared to batch size 32.

- *Batch size scaling*: 10% for ResNet50 vs. 8% for InceptionV3 on a V100 GPU
- Speedup with increasing batch size is model dependent

Single GPU Training



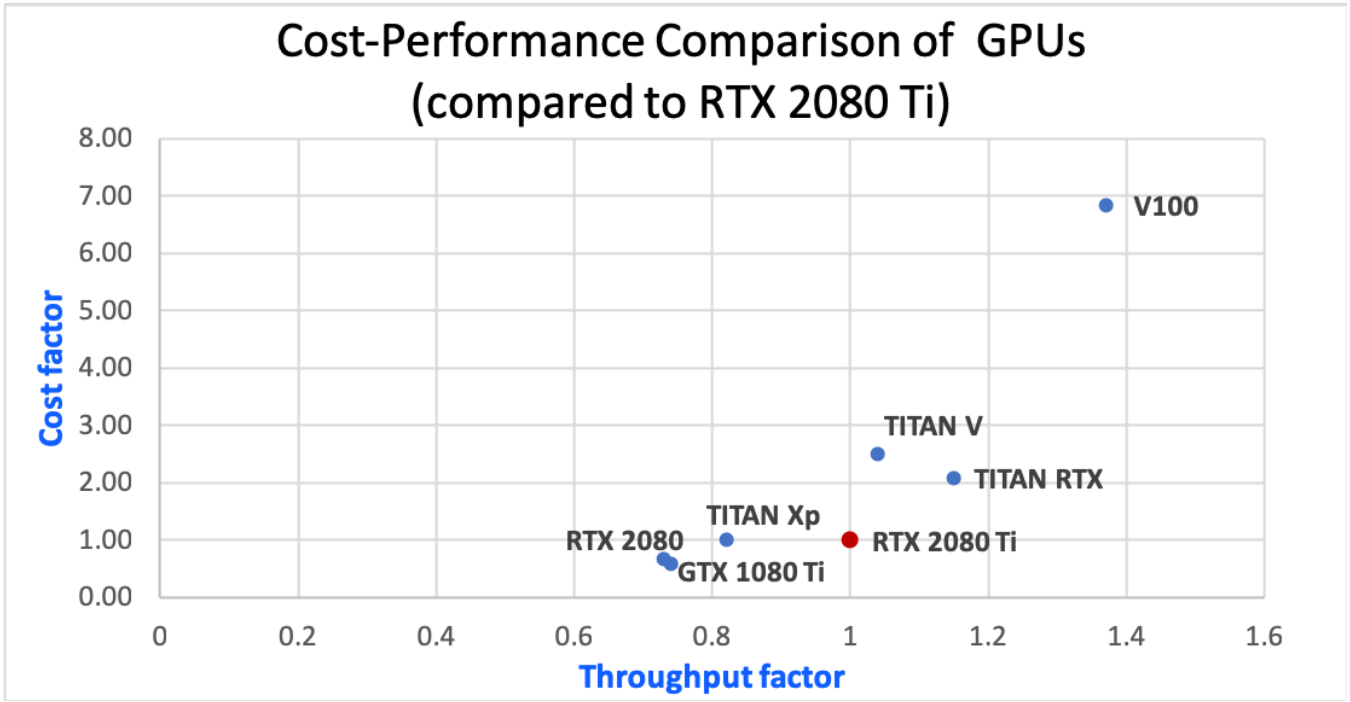
GPU Prices

- RTX 2080 Ti: $1,199.00
- RTX 2080: $799.00
- Titan RTX: $2,499.00
- Titan V: $2,999.00
- Tesla V100 (32 GB): ~$8,200.00
- GTX 1080 Ti: $699.00
- Titan Xp: $1,200.00

The scaling with GPU type is dependent on neural network architecture.

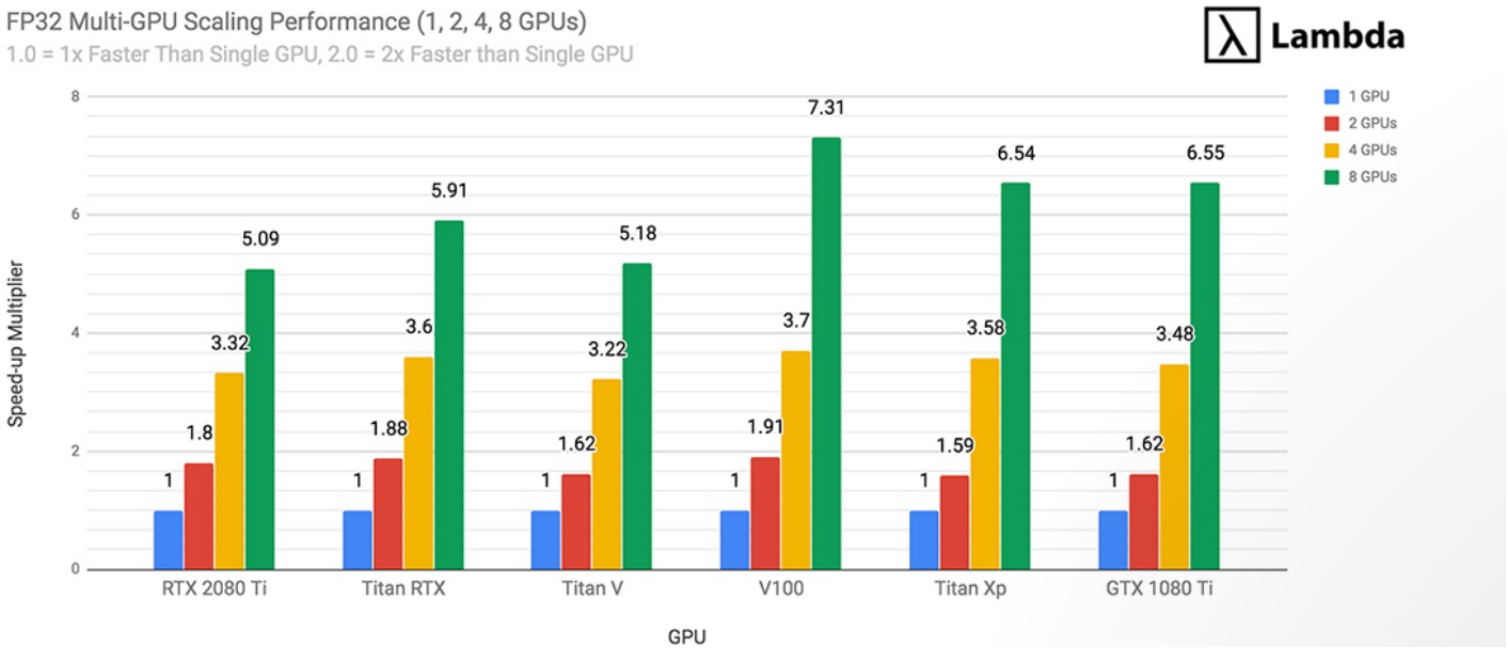
https://lambdalabs.com/blog/2080-ti-deep-learning-benchmarks/

Cost-Performance Tradeoff



Points lying above the straight line connecting RTX 2080 Ti and origin corresponds to GPUs which are costlier than RTX 2080 Ti but don't offer same proportional increase in throughput.

Single Node, Multi-GPU Training



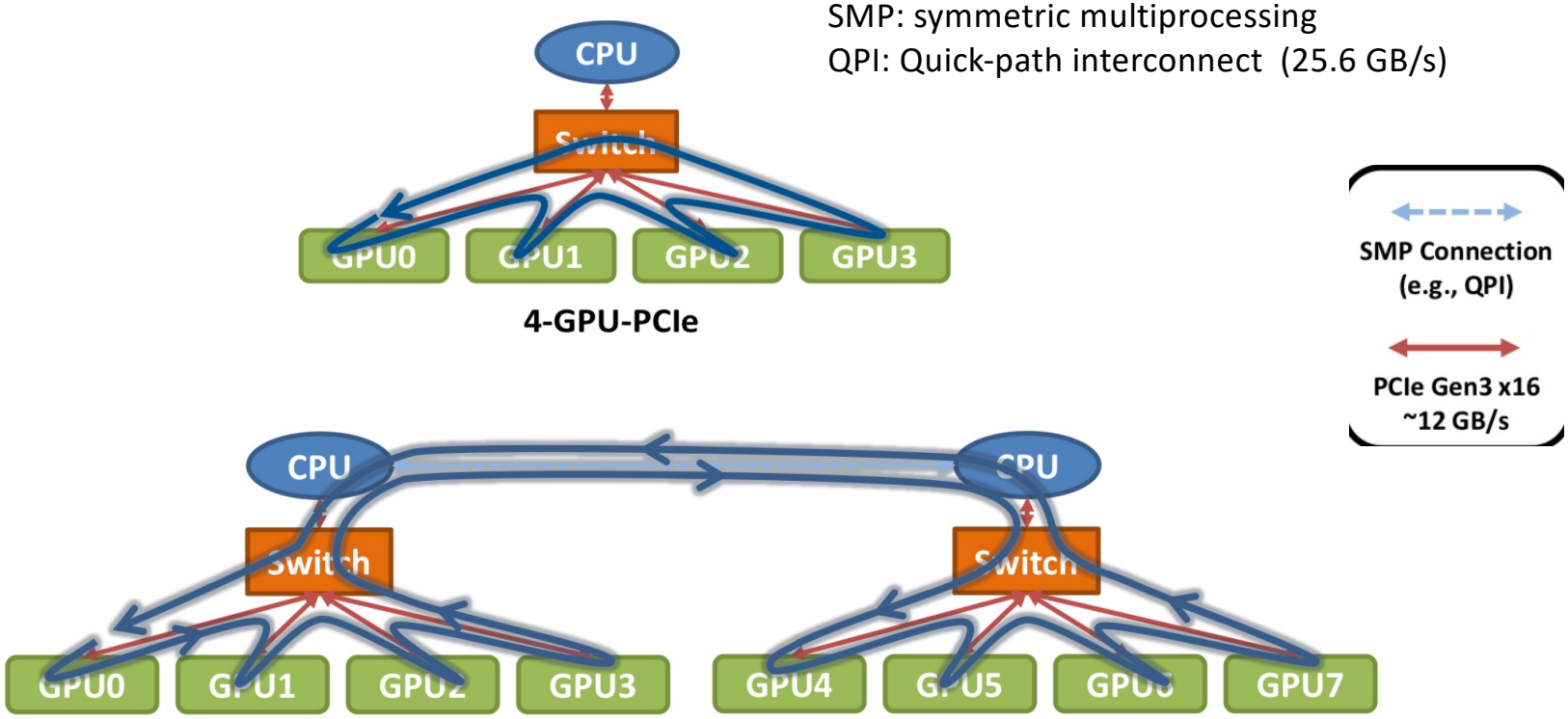
https://lambdalabs.com/blog/2080-ti-deep-learning-benchmarks/

Single Node, Multi GPU Training

- Communication libraries (e.g., NCCL) and supported communication algorithms/collectives (broadcast, all-reduce, gather)
  - NCCL (“Nickel”) is library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications
- Communication link bandwidth: PCIe/QPI or NVlink
- Communication algorithms depend on the communication topology (ring, hub-spoke, fully connected) between the GPUs.
- Most collectives amenable to bandwidth-optimal implementation on rings, and many topologies can be interpreted as one or more rings [P. Patarasuk and X. Yuan]

Ring based collectives

PCIe: Peripheral Component Interconnect express
SMP: symmetric multiprocessing
QPI: Quick-path interconnect (25.6 GB/s)



PCIe and NVLink

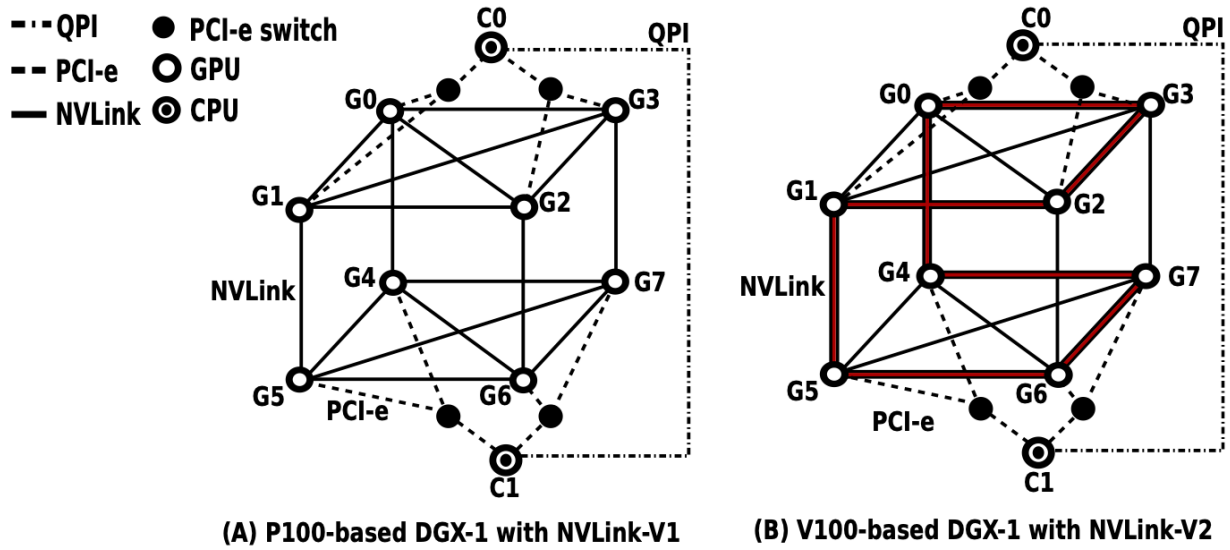


Fig. 1: PCIe and NVLink-V1/V2 topology for P100-DGX-1 and V100-DGX-1.

Distributed Training

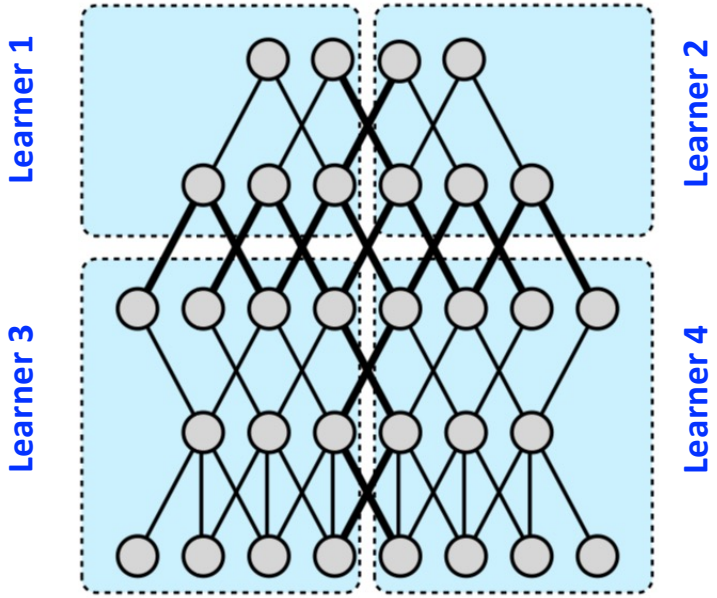
- Type of Parallelism: Model, Data, Hybrid
- Type of Aggregation: Centralized, decentralized
- Centralized aggregation: parameter server
- Decentralized aggregation: P2P, all reduce
- Performance metric: Scaling efficiency
  - Defined as the ratio between the run time of one iteration on a single GPU and the run time of one iteration when distributed over n GPUs.

Parallelism

- Parallel execution of a training job on different compute units through scale-up (single node, multiple and faster GPUs) or scale-out (multiple nodes distributed training)
- Enables working with large models by partitioning model across learners
- Enables efficient training with large datasets using large “effective” batch sizes (batch split across learners)
  - Speeds up computation
- Model, Data, Hybrid Parallelism

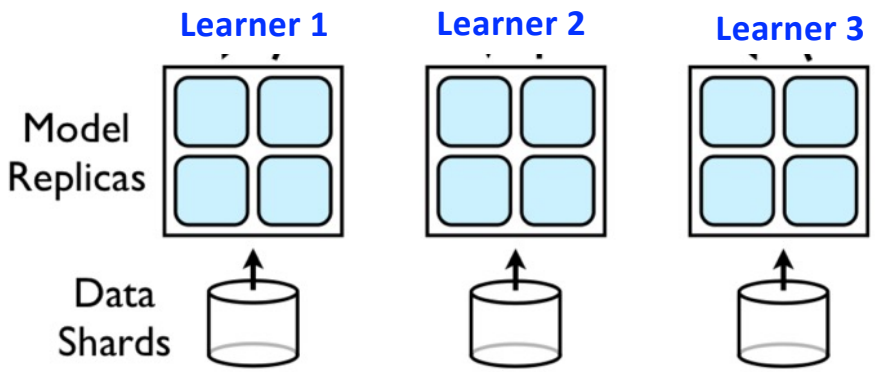
Model Parallelism

- Splitting the model across multiple learners



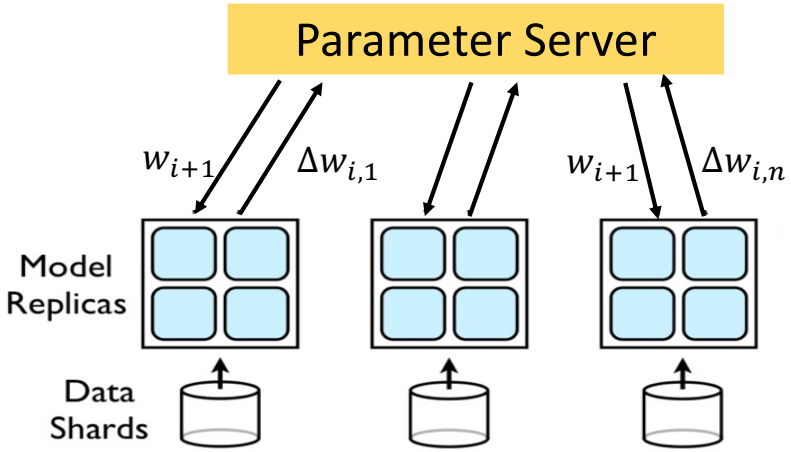
- 5 layered neural network
- Partitioned across 4 learners
- Bold edges cross learner boundaries and involve inter-learner communication
- Performance benefits depend on
    - Connectivity structure
    - Compute demand of operations
- Heavy compute and local connectivity –benefit most
    - Each machine handles a subset of computation
    - Low network traffic

Data Parallelism



- Model is replicated on different learners
- Data is sharded and each learner work on a different partition
- Helps in efficient training with large amount of data
- Parameters (weights, biases, gradients) from different replicas need to be synchronized

Parameter server (PS) based Synchronization



- Each learner executes the entire model
- After each mini-batch processing a learner calculates the gradient and sends it to the parameter server
- The parameter server calculates new value of weights and sends them to the model replicas

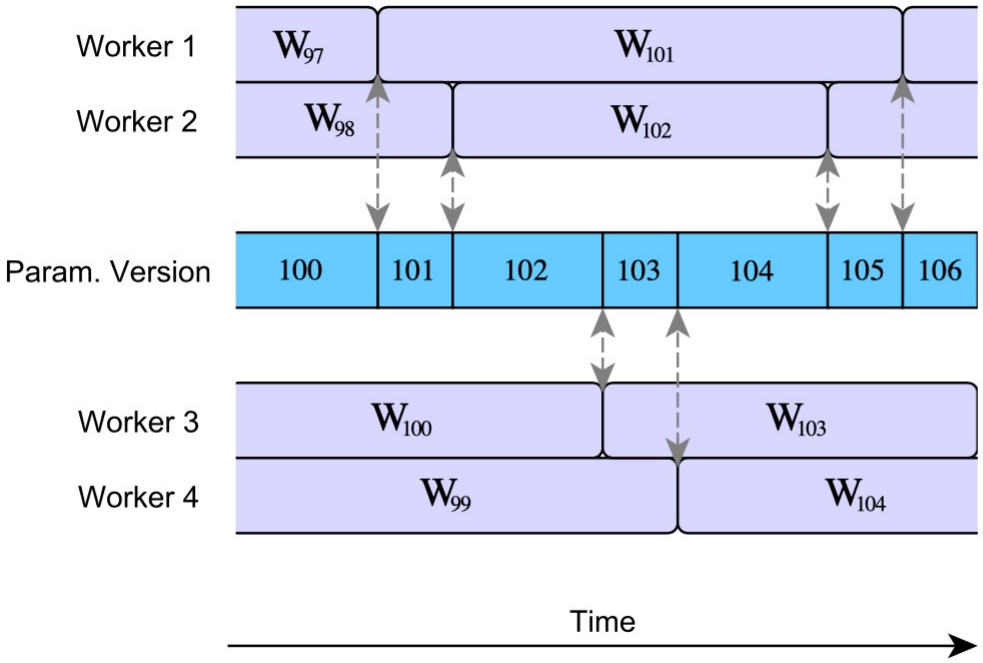
Synchronous SGD and the Straggler problem

- PS needs to wait for updated gradients from all the learners before calculating the model parameters
- Even though size of mini-batch processed by each learner is same, updates from different learners may be available at different times at the PS
  - Randomness in compute time at learners
  - Randomness in communication time between learners and PS
- Waiting for slow and straggling learners diminishes the speed-up offered by parallelizing the training

Asynchronous SGD and Stale Gradients

- PS updates happen without waiting for all learners
- Weights that a learner uses to evaluate gradients may be old values of the parameters at PS
  - Parameter server asynchronously updates weights
  - By the time learner gets back to the PS to submit gradient, the weights may have already been updated at the PS (by other learners)
  - Gradients returned by this learner are stale (i.e., were evaluated at an older version of the model)
- Stale gradients can make SGD unstable, slowdown convergence, cause sub-optimal convergence (compared to Sync-SGD)

Stale Gradient Problem in Async-SGD



$$\vdots$$

$$W_{101} = W_{100} - \lambda \Delta W_{97}$$

$$W_{102} = W_{101} - \lambda \Delta W_{98}$$

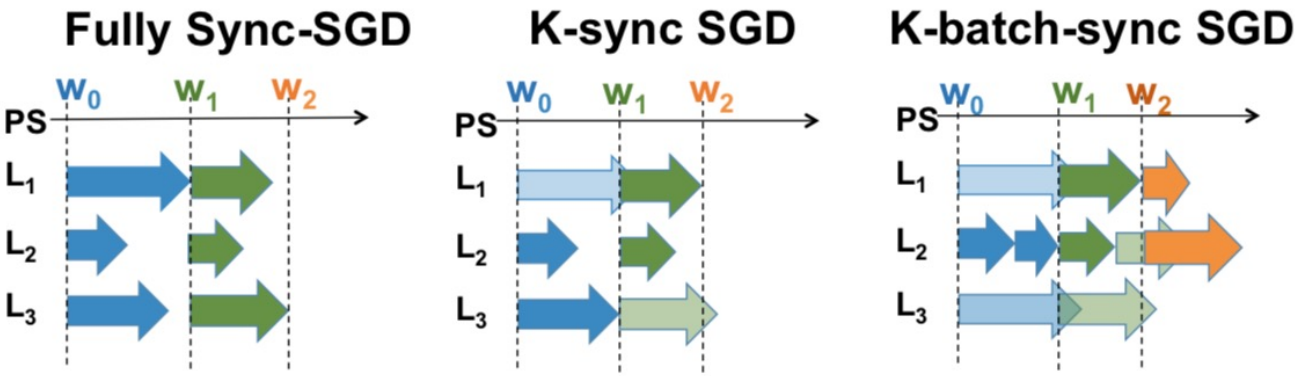
$$W_{103} = W_{102} - \lambda \Delta W_{100}$$

$$W_{104} = W_{103} - \lambda \Delta W_{99}$$

$$\vdots$$

blog.skymind.ai Distrbuted Deep Learning, Part 1: An Introduction to Distributed Training of Neural Networks

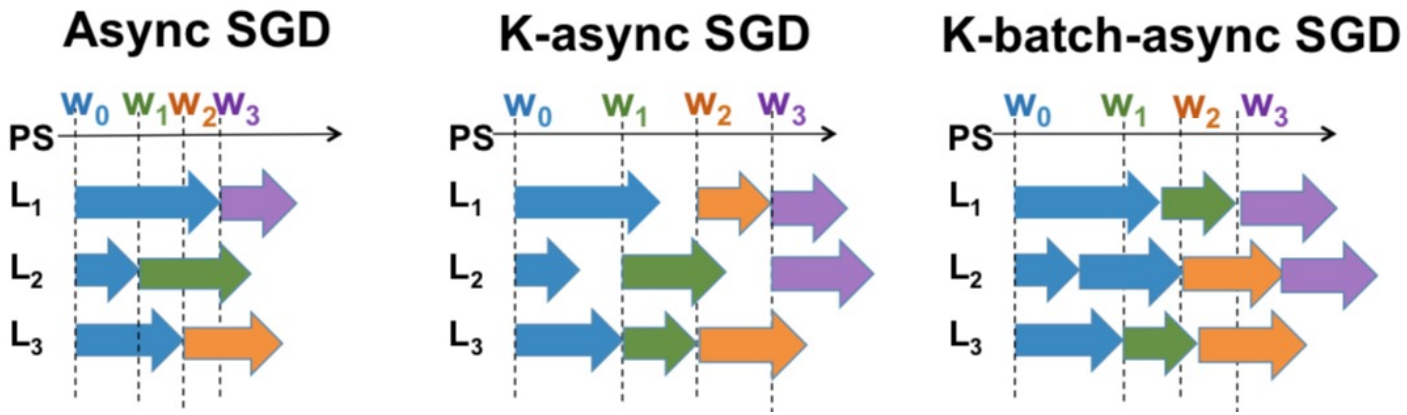
Synchronous SGD Variants



- P: total number of learners
- K: number of learners/mini-batches the PS waits for before updating parameters
- Lightly shaded arrows indicate straggling gradient computations that are canceled.

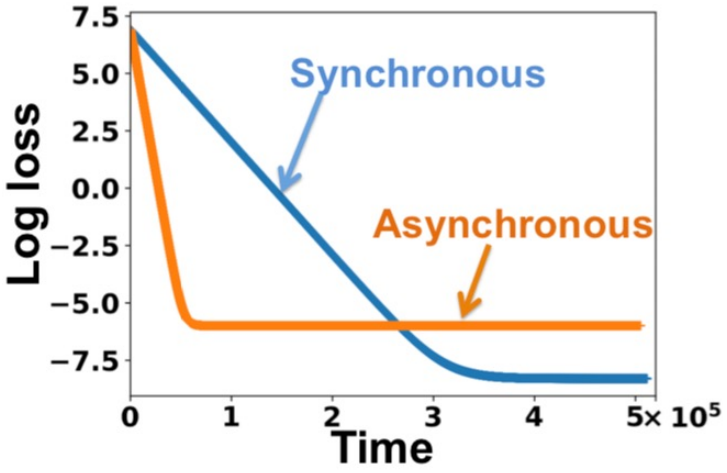
- K-sync SGD: PS waits for gradients from K *learners* before updating parameters; the remaining learners are canceled
- When K = P , K-sync SGD is same as Fully Sync-SGD
- K-batch sync: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining (unfinished) learners are canceled**
  - Irrespective of which learner the gradients come from
  - Wherever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the same local value of the parameters
- Runtime per iteration reduces with K-batch sync; error convergence is same as K-sync

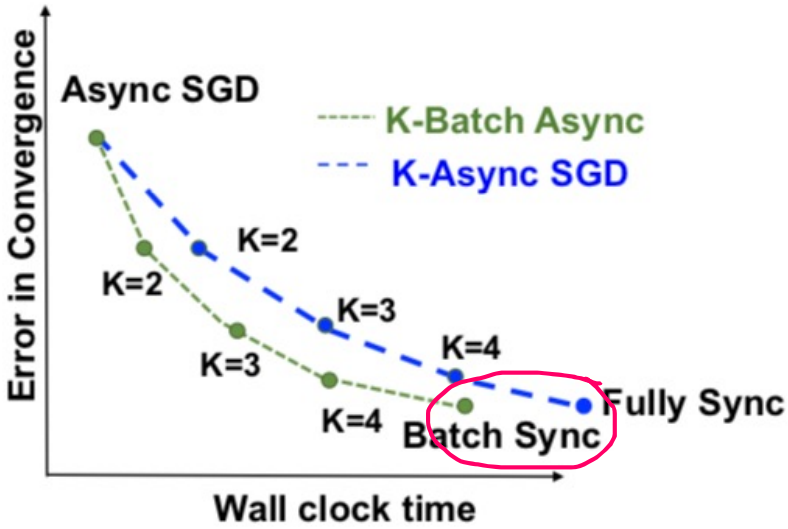
Asynchronous SGD and Variants



- K-async SGD: PS waits for gradients from K *learners* before updating parameters, but the remaining learners are not canceled; each learner may be giving a gradient calculated at stale version of the parameters
- When K = 1 , K-async SGD is same as Async-SGD
- K-batch async: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining learners are not canceled**
  - Wherever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the current value of the PS
- Runtime per iteration reduces with K-batch async; error convergence is same as K-async

Error-Runtime Tradeoff in SGD Variants

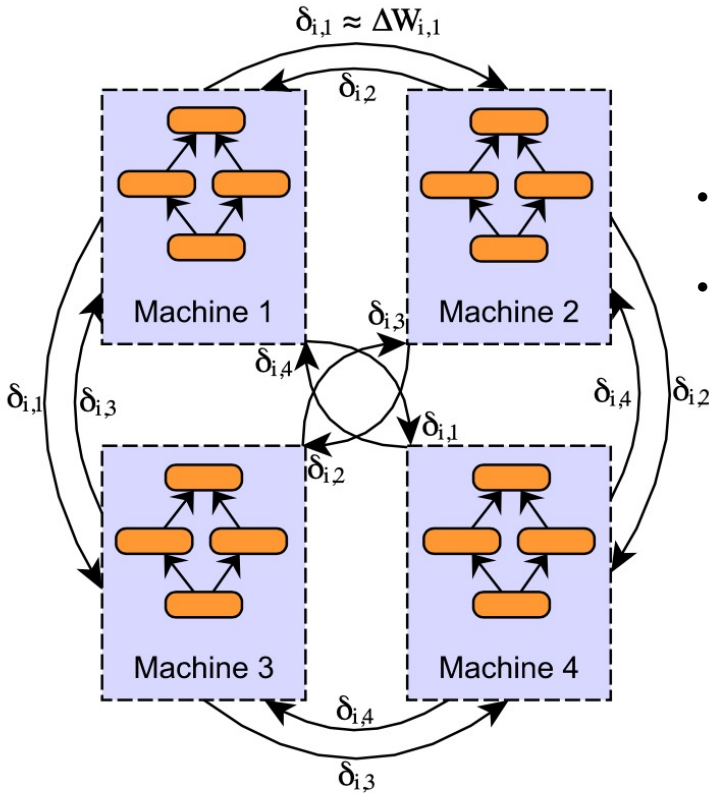




- Error-runtime trade-off for Sync and Async-SGD with same learning rate.
- Async-SGD has faster decay with time but a higher error floor.

Dutta et al. Slow and stale gradients can win the race: error-runtime tradeoffs in distributed SGD. 2018

Decentralized Aggregation: Peer-to-Peer



- Peer to peer communication is used to transmit model updates between workers).
- Updates are heavily compressed, resulting in the size of network communications being reduced by some 3 orders of magnitude.

Nikko Storm. Scalable distributed dnn training using commodity gpu cloud computing 2015

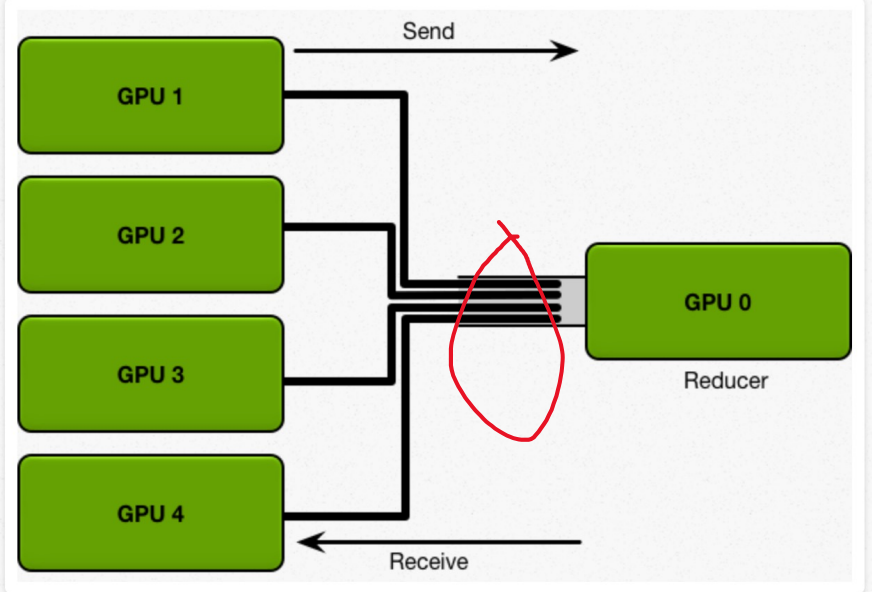
Reduction over Gradients

- To synchronize gradients of N learners, a reduction operation needs to be performed

$$\sum_{j=1}^{N} \Delta w_j$$

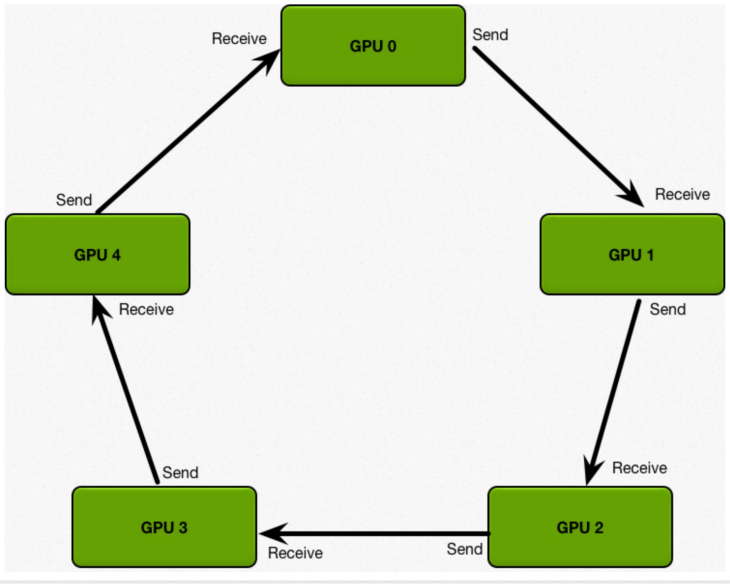
Reduction Topologies

Parameter server: single reducer



SUM (Reduce operation) performed at PS

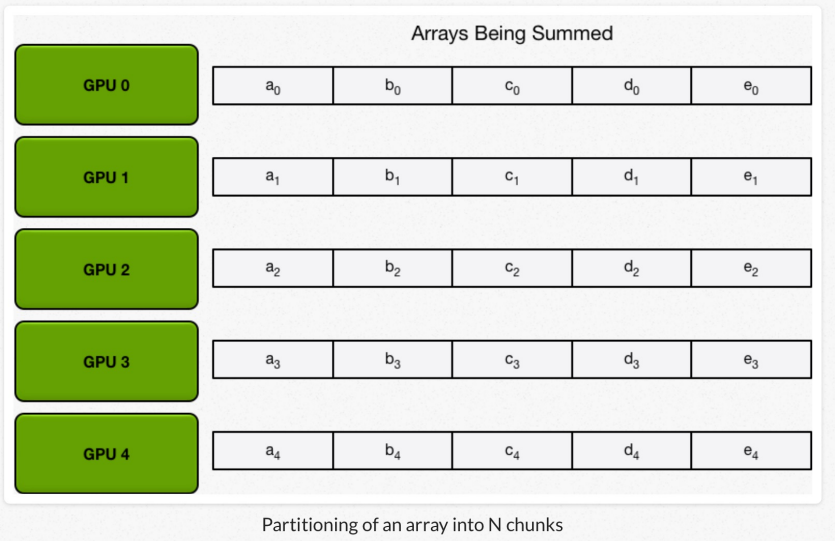
GPUs arranged in a logical Ring (aka bucket) :
all are reducers



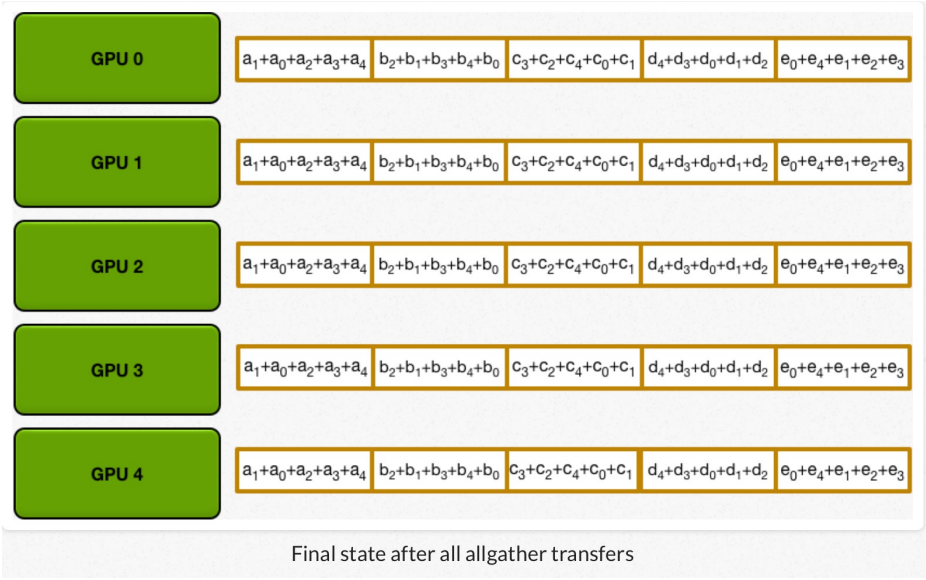
SUM (Reduce operation) performed at all nodes

- Each node has a left neighbor and a right neighbor
- Node only sends data to its right neighbor, and only receives data from its left neighbor

Example Problem



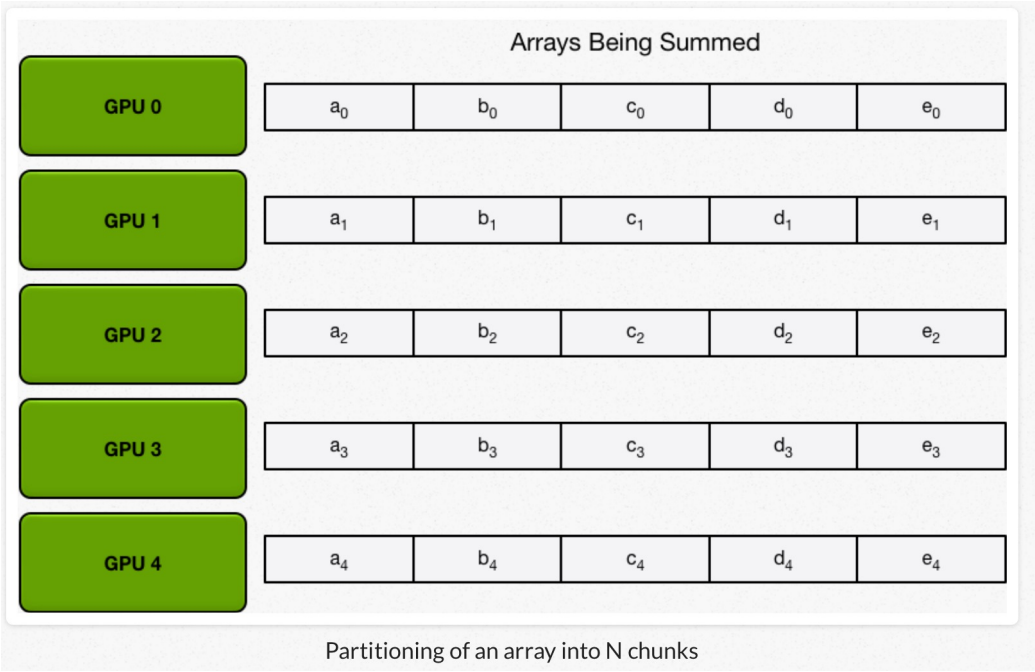
Partitioning of an array into N chunks



Final state after all allgather transfers

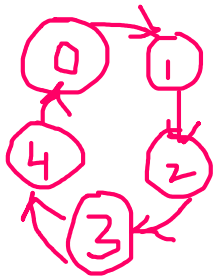
Ring All-Reduce

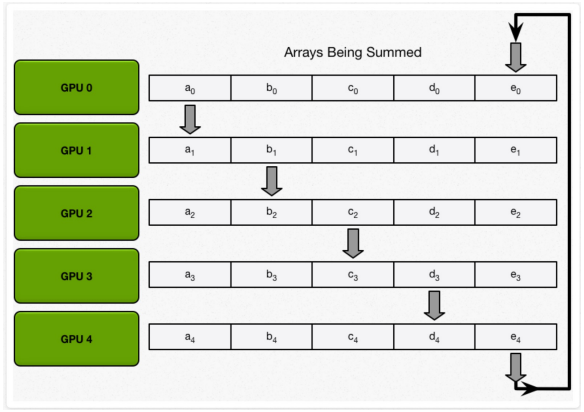
- Two step algorithm:
  - *Scatter-reduce*
    - GPUs exchange data such that every GPU ends up **with a chunk** of the final result
  - *Allgather*
    - GPUs exchange chunks from scatter-reduce such that all GPUs end up with the complete final result.

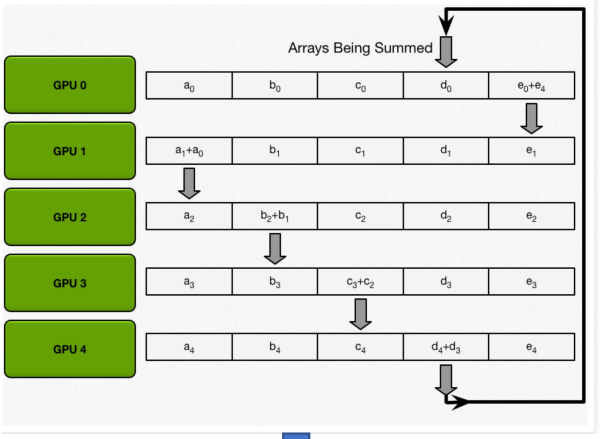


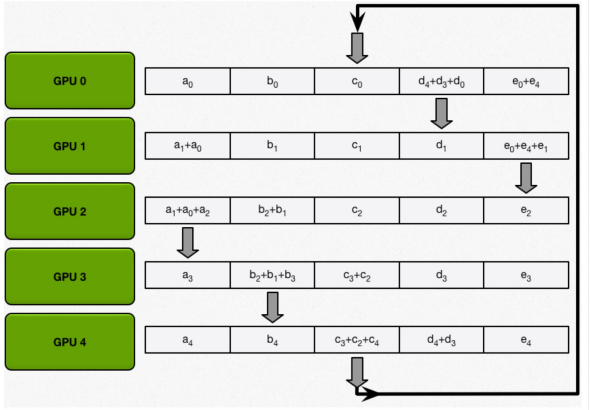
Partitioning of an array into N chunks

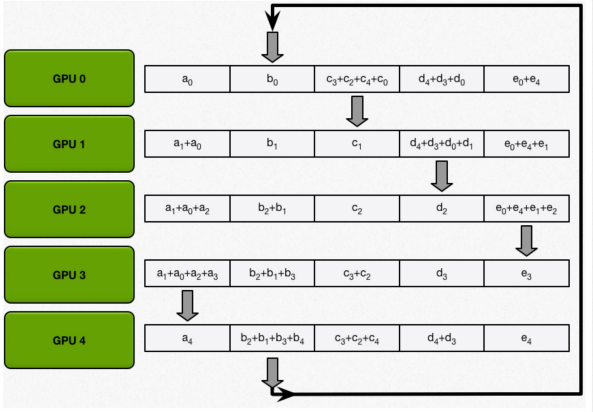
Ring All-Reduce: Scatter-Reduce Step



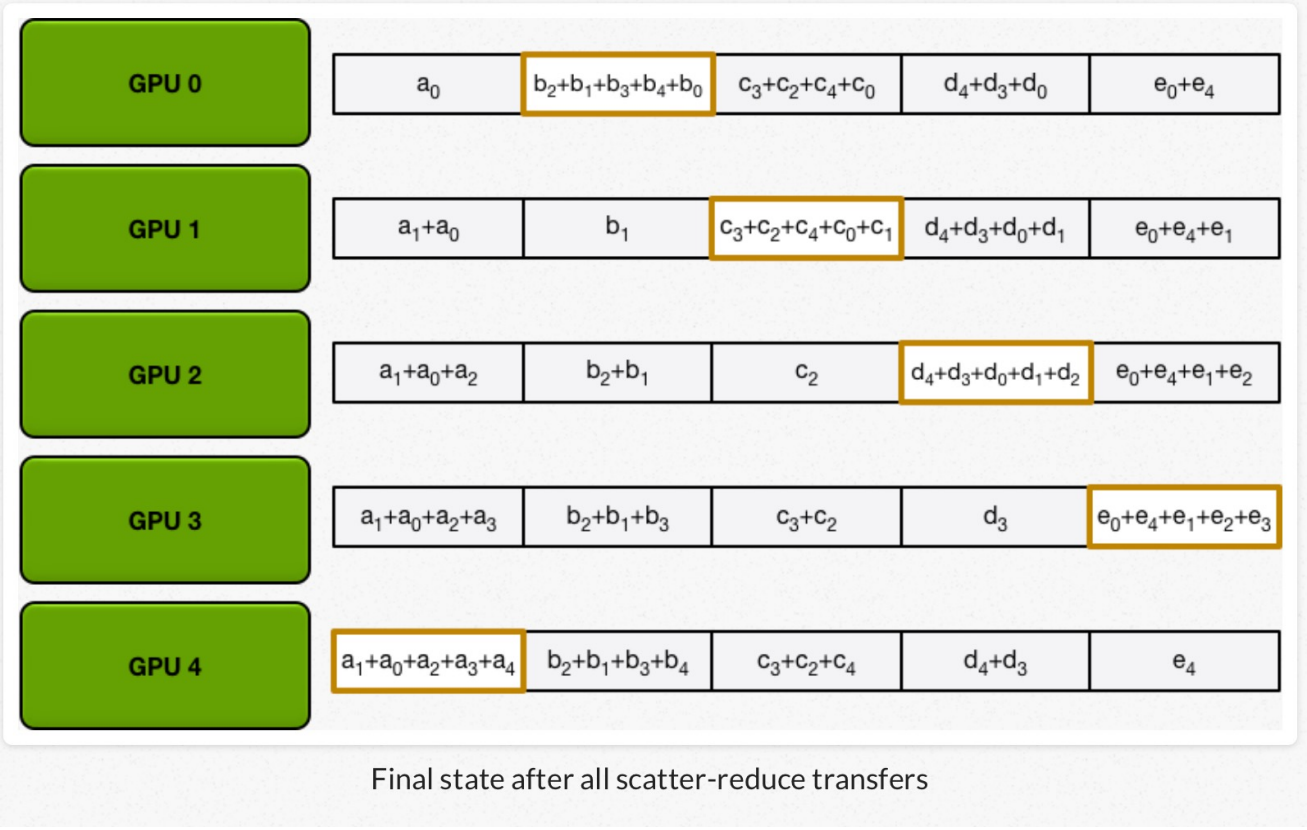








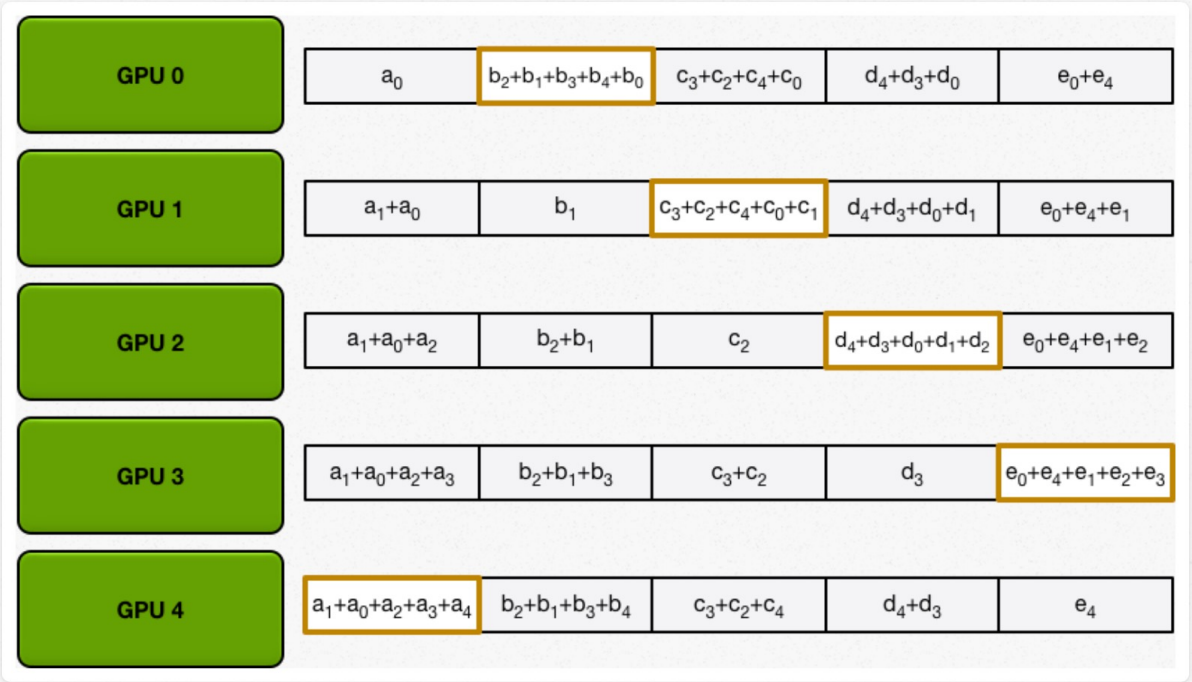
Ring All-Reduce: End of Scatter-Reduce Step



Final state after all scatter-reduce transfers

How many iterations in scatter-reduce step with N GPUs ?

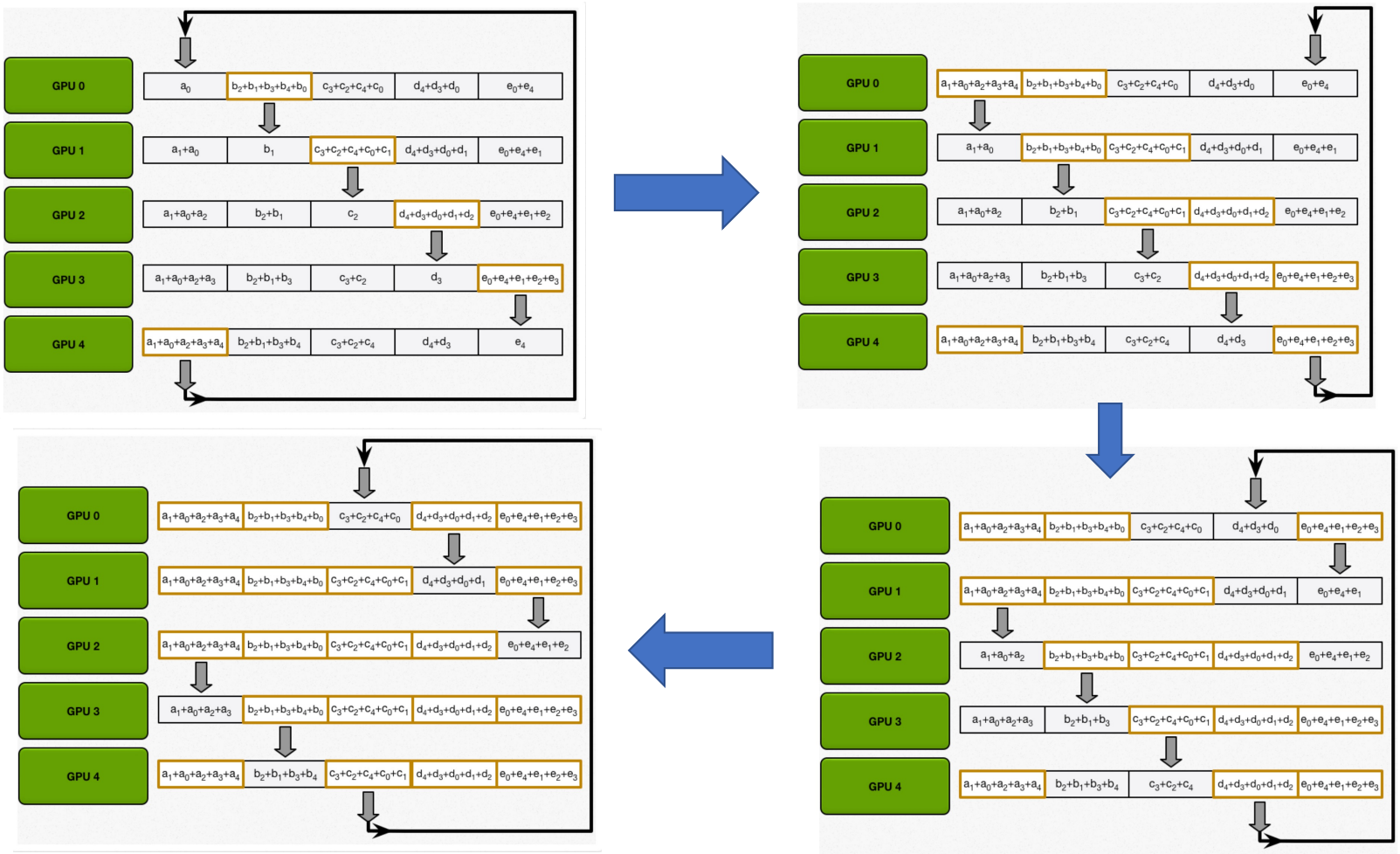
Ring All-Reduce: What to do next ?



Final state after all scatter-reduce transfers

GPU	Send	Receive
0	Chunk 1	Chunk 0
1	Chunk 2	Chunk 1
2	Chunk 3	Chunk 2
3	Chunk 4	Chunk 3
4	Chunk 0	Chunk 4

Ring All-Reduce: AllGather Step



Ring All-Reduce: End of AllGather Step

Final state after all allgather transfers

How many iterations in allgather step with N GPUs ?

Parameter Server (PS) vs Ring All-Reduce: Communication Cost

- P: number of processes N: total number of model parameters
- PS (centralized reduce)
  - Amount of data sent to PS by (P-1) learner processes: N(P-1)
  - After reduce, PS sends back updated parameters to each learner
  - Amount of data sent by PS to learners: N(P-1)
  - Total communication cost at PS process is proportional to 2N(P-1)
- Ring All-Reduce (decentralized reduce)
  - Scatter-reduce: Each process sends N/P amount of data to (P-1) learners
    - Total amount sent (per process): N(P-1)/P
  - AllGather: Each process again sends N/P amount of data to (P-1) learners
  - Total communication cost per process is 2N(P-1)/P
- PS communication cost is proportional to P whereas ring all-reduce cost is practically independent of P for large P (ratio (P-1)/P tends to 1 for large P)
- Which scheme is more bandwidth efficient ?
- Note that both PS and Ring all-reduce involve synchronous parameter updates

All-Reduce applied to Deep Learning

- Backpropagation computes gradients starting from the output layer and moving towards in the input layer
- Gradients for output layers are available earlier than inner layers
- Start all reduce on the output layer parameters while other gradients are being computed
- Overlay of communication and local compute

Scaling using compute-communication overlap in all-reduce

- https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/

PyTorch Distributed DL

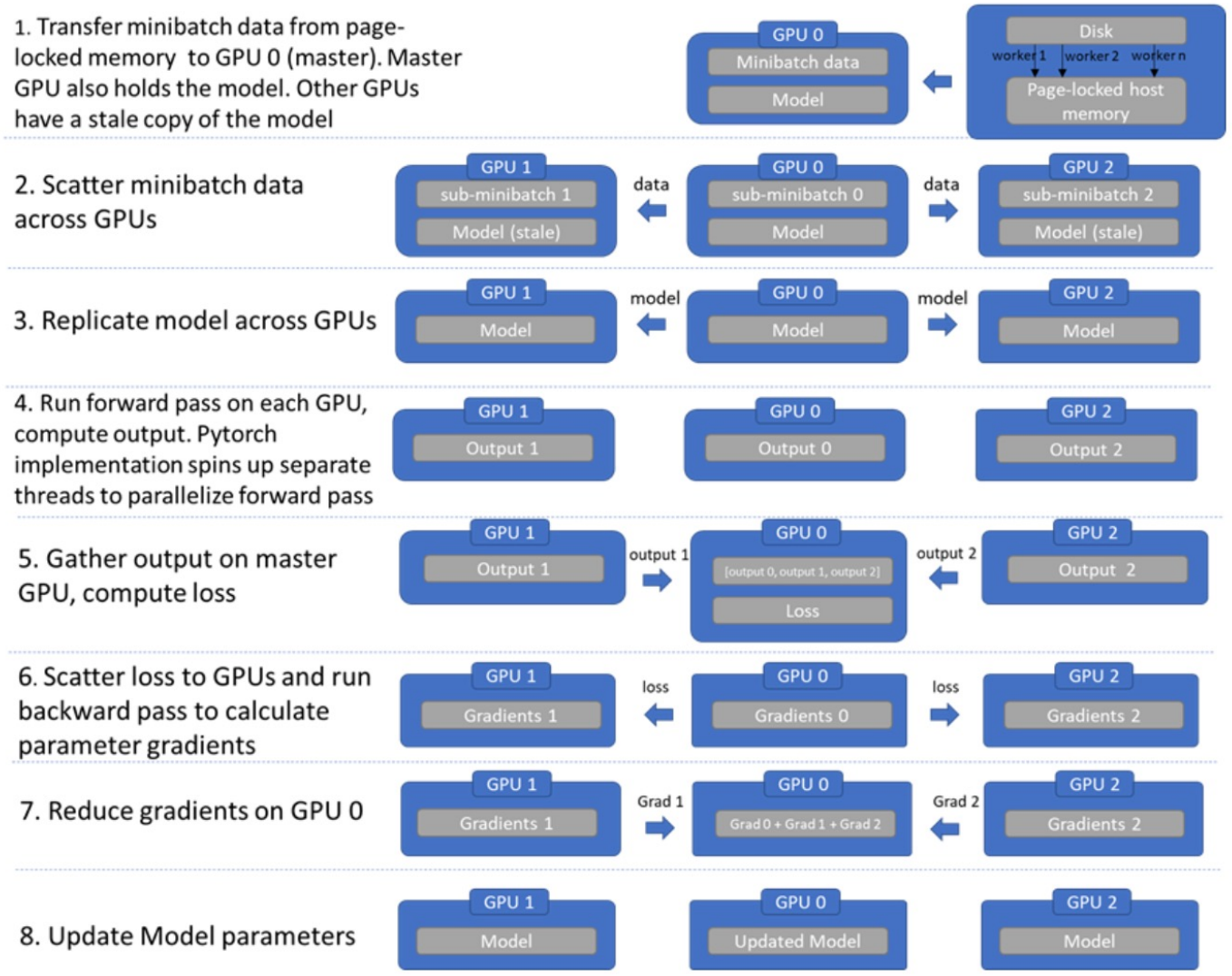
Data Parallelism: Multiple GPU configurations

- PyTorch provides several options for data-parallel training. For applications that gradually grow from simple to complex and from prototype to production, the common development trajectory would be:
  1. Use single-device training if the data and model can fit in one GPU, and training speed is not a concern.
  2. Use single-machine multi-GPU DataParallel to make use of multiple GPUs on a single machine to speed up training with minimal code changes.
  3. Use single-machine multi-GPU DistributedDataParallel, if you would like to further speed up training and are willing to write a little more code to set it up.
  4. Use multi-machine DistributedDataParallel and the launching script, if the application needs to scale across machine boundaries.

Data Parallel

One GPU (0) acts as the master GPU and coordinates data transfer.

Implemented in PyTorch data_parallel module



Inefficiencies in DataParallel

- Redundant data copies
  - Data is copied from host to master GPU and then sub-minibatches are scattered across other GPUs
- Model replication across GPUs before forward pass
  - Since model parameters are updated on the master GPU, model must be re-synced at the beginning of every forward pass
- Thread creation/destruction overhead for each batch
  - Parallel forward is implemented in multiple threads (this could just be a Pytorch issue)
- Gradient reduction pipelining opportunity left unexploited
  - In Pytorch 1.01 data-parallel implementation, gradient reduction happens at the end of backward pass. I'll discuss this in more detail in the distributed data parallel section.
- Unnecessary gather of model outputs on master GPU
- Uneven GPU utilization
  - Loss calculation performed on master GPU
  - Gradient reduction, parameter updates on master GPU

torch.nn.DistributedDataParallel

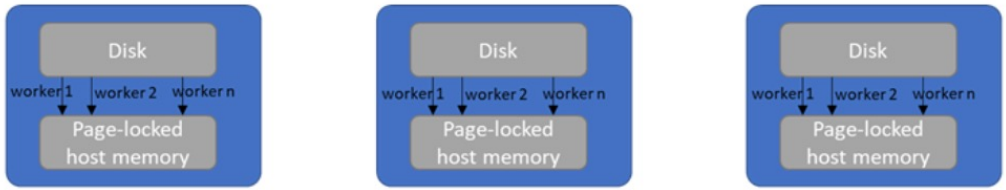
- torch.nn.parallel.DistributedDataParallel implements data parallelism at the module level
  1. Splitting the input across the specified devices by chunking in the batch dimension
  2. The module is replicated on each machine and each device, and each replica handles a portion of the input.
  3. During the backward pass, gradients from each node are **averaged**
- https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html
- DistributedDataParallel is proven to be significantly faster than torch.nn.DataParallel for single-node multi-GPU data parallel training.

Distributed Data Parallel

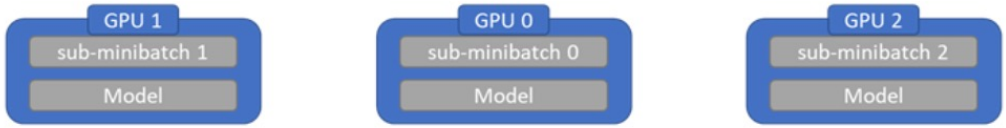
No master GPUs

Implemented in PyTorch DistributedDataParallel module

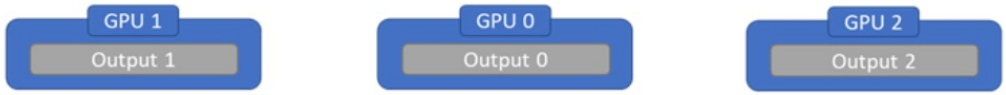
1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data



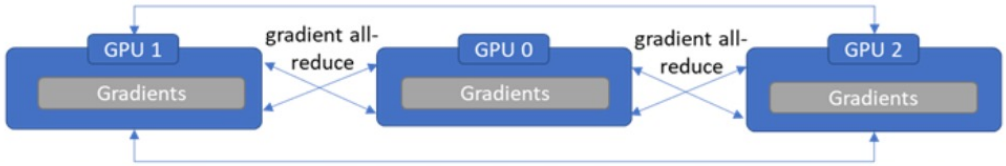
2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



3. Run forward pass on each GPU, compute output

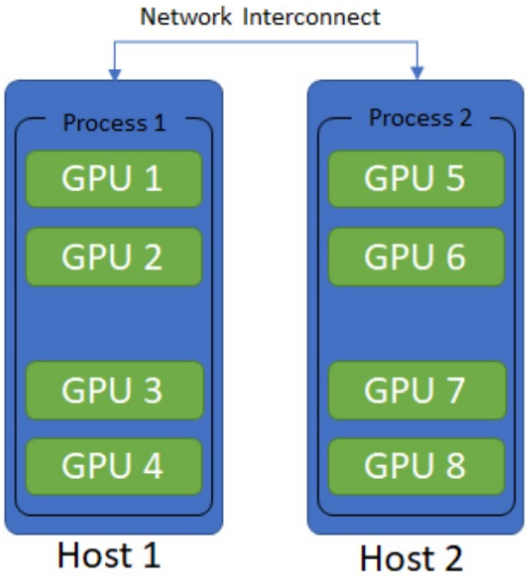


4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation

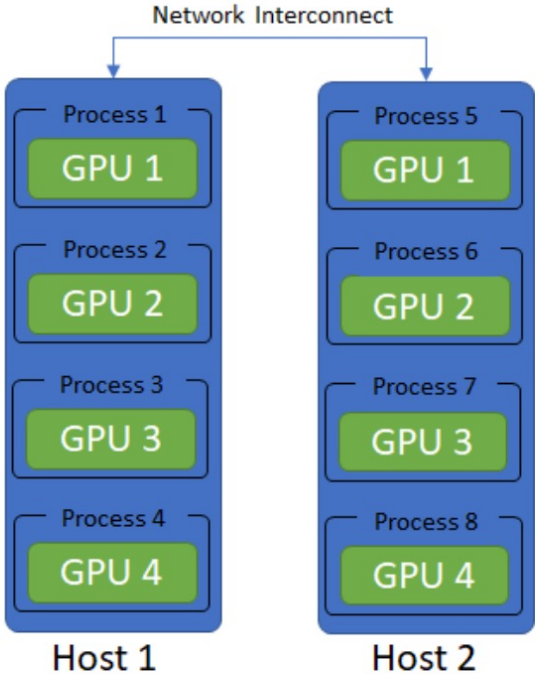


5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required

Distributed Data Parallel in Multi-host Setting



Configuration 1: Each process operates on all four GPUs in data parallel mode

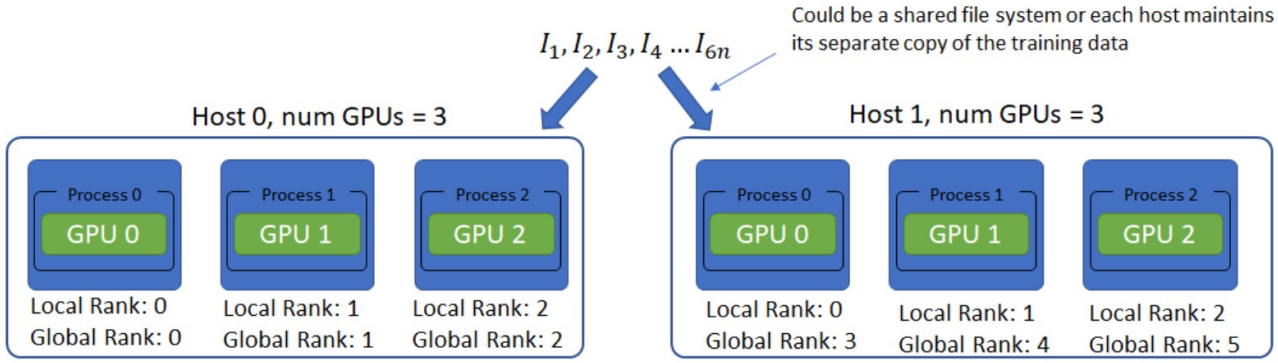


Configuration 2: Each process operates on a single GPU

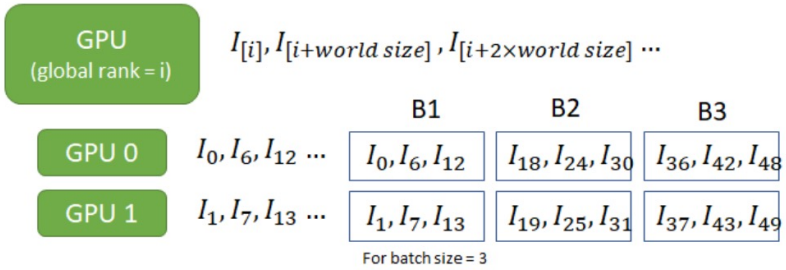
Steps in multi-host setting

- Distributed-data-parallel is launched on each host independently
- Mechanism needed to have synchronization between multiple processes running on different hosts
  - **init_process_group** function that uses a shared file system or a TCP IP address/port to sync the processes.
- Each process should load non-overlapping copies of the data
  - DistributedDataSampler

DistributedDataSampler



Worldsize: *num Hosts* × *num GPUs per host* = 2×3 = 6
Each process knows its *local rank* and *host number*. Can calculate *global rank* from this info.
Global rank = local rank + num GPUs per host × *host number*




```
def __iter__(self):
    # deterministically shuffle based on epoch
    g = torch.Generator()
    g.manual_seed(self.epoch)
    indices = torch.randperm(len(self.dataset), generator=g).tolist()

    # add extra samples to make it evenly divisible
    indices += indices[:(self.total_size - len(indices))]
    assert len(indices) == self.total_size

    # subsample
    indices = indices[self.rank:self.total_size:self.num_replicas]
    print("len indices {0}".format(len(indices)))
    return iter(indices)
```

DistributedSampler
(distributed.py)