Machine Learning Systems

Lecture 12: Scalable & Distributed Machine Learning

Pooyan Jamshidi

Learning goals

- Review **gradient descent** and its **variations**
- Understand **scalability challenges** during training
- Examine **computer systems tricks** to make gradient descent scalable to handle large training sets
- Discuss **algorithms** and **architectures** to optimize gradient descent in a parallel and distributed setting
- **Case Study**: Distributed Model Training at Uber

Acknowledgement

Sebastian Ruder, “An overview of gradient descent optimization algorithms”, 2017

Seb Arnold, “An Introduction to Distributed Deep Learning”, 2016

Dean et al., “Large Scale Distributed Deep Networks”, in NIPS 2012

Li et al., “Scaling Distributed Machine Learning with the Parameter Server”, in OSDI 2014

Langford et al., “Slow learners are fast”. In NIPS 2009

CS231n Convolutional Neural Networks for Visual Recognition

Jain, “A Brief Primer: Stochastic Gradient Descent”, 2017

Blaise Barney, “Introduction to Parallel Computing”

Supervised ML

Supervised machine learning generally consists of three phases:

- **Training** (generating a model)
- **Validation** (determining values of hyper-parameters)
- **Inference** (making predictions with the trained model)

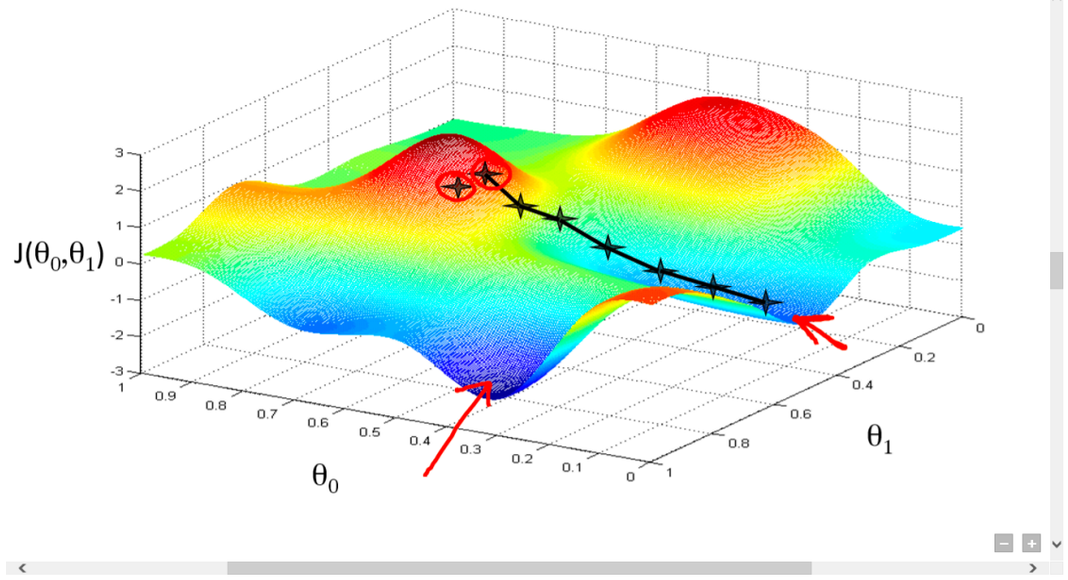
Key aim of model training

Finding values for a model's parameters, θ, such that two, often conflicting, goals are met:

- Error on the set of training examples is minimized,
- The model generalizes to new data

Gradient Descent

The most popular algorithms to perform optimization especially for optimizing neural networks



What is gradient descent?

- Gradient descent is an algorithm that iteratively tweaks a model's parameters

- With the goal of minimizing the discrepancy between the model's predictions and the "true" labels associated with a set of training examples.

Loss function

$$J(\theta) = \frac{1}{2m}\Sigma_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Batch gradient descent

repeat until convergence {

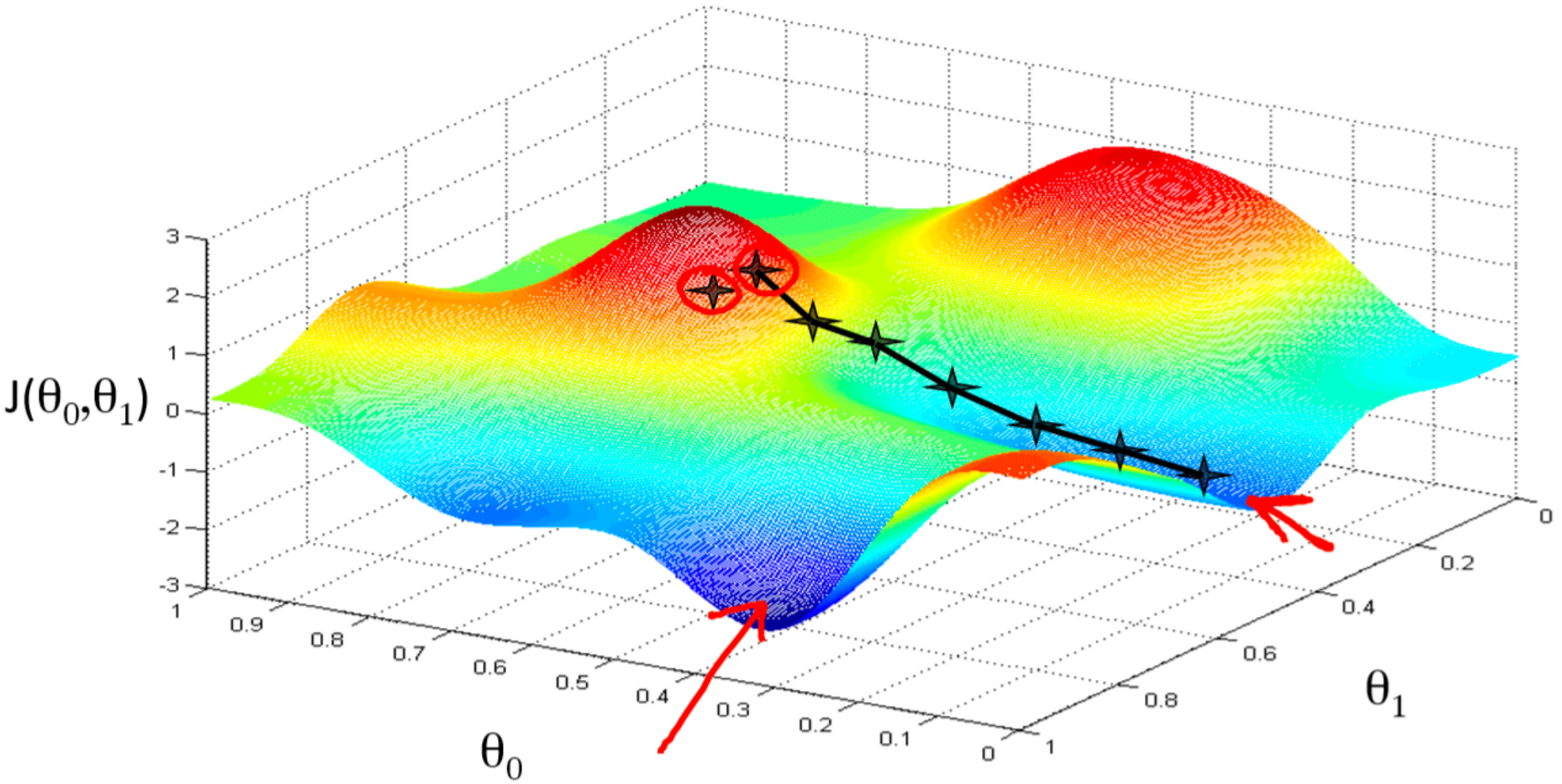
$$\theta \leftarrow \theta - \gamma \nabla_{\theta} J(\theta)$$

}

Gradient descent in code

```
for i in range(nb_epochs):
  weights_grad = evaluate_gradient(loss_function, data, params)
  weights += - learning_rate * weights_grad
```

Initial value of θ may results in two different local minima



Gradient descent is very costly

$$\nabla_\theta J(\theta) = (\frac{\partial}{\partial \theta_1} J(\theta), \frac{\partial}{\partial \theta_2} J(\theta), \ldots, \frac{\partial}{\partial \theta_n} J(\theta))$$

Each partial derivative involves computing a sum over every training example

$$\begin{aligned}\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \left(\frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \right) \\ &= \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \frac{\partial}{\partial \theta_j} h_\theta(x^{(i)})\end{aligned}$$

The key idea in stochastic gradient descent is to drop the sum

$$\frac{\partial}{\partial \theta_j} J(\theta) \approx (h_\theta(x^{(i)}) - y^{(i)}) \frac{\partial}{\partial \theta_j} h_\theta(x^{(i)})$$

Stochastic Gradient Descent (SGD)

repeat until convergence {

for $i := 1,2,...,m\{$

$$\theta \leftarrow \theta - \gamma \nabla J(\theta; x^{(i)}; y^{(i)})$$

}

}

$$\nabla J(\theta; x^{(i)}; y^{(i)}) = (h_\theta(x^{(i)}) - y^{(i)}) \nabla h_\theta(x^{(i)})$$

SGD in code

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    weights_grad = evaluate_gradient(loss_function, example, params)
    weights += - learning_rate * weights_grad
```

Mini-batch Gradient Descent

repeat until convergence {

for $i := 1,2,...,m\{$

$$\theta \leftarrow \theta - \gamma \nabla J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

}

}

Minibatch GD in code

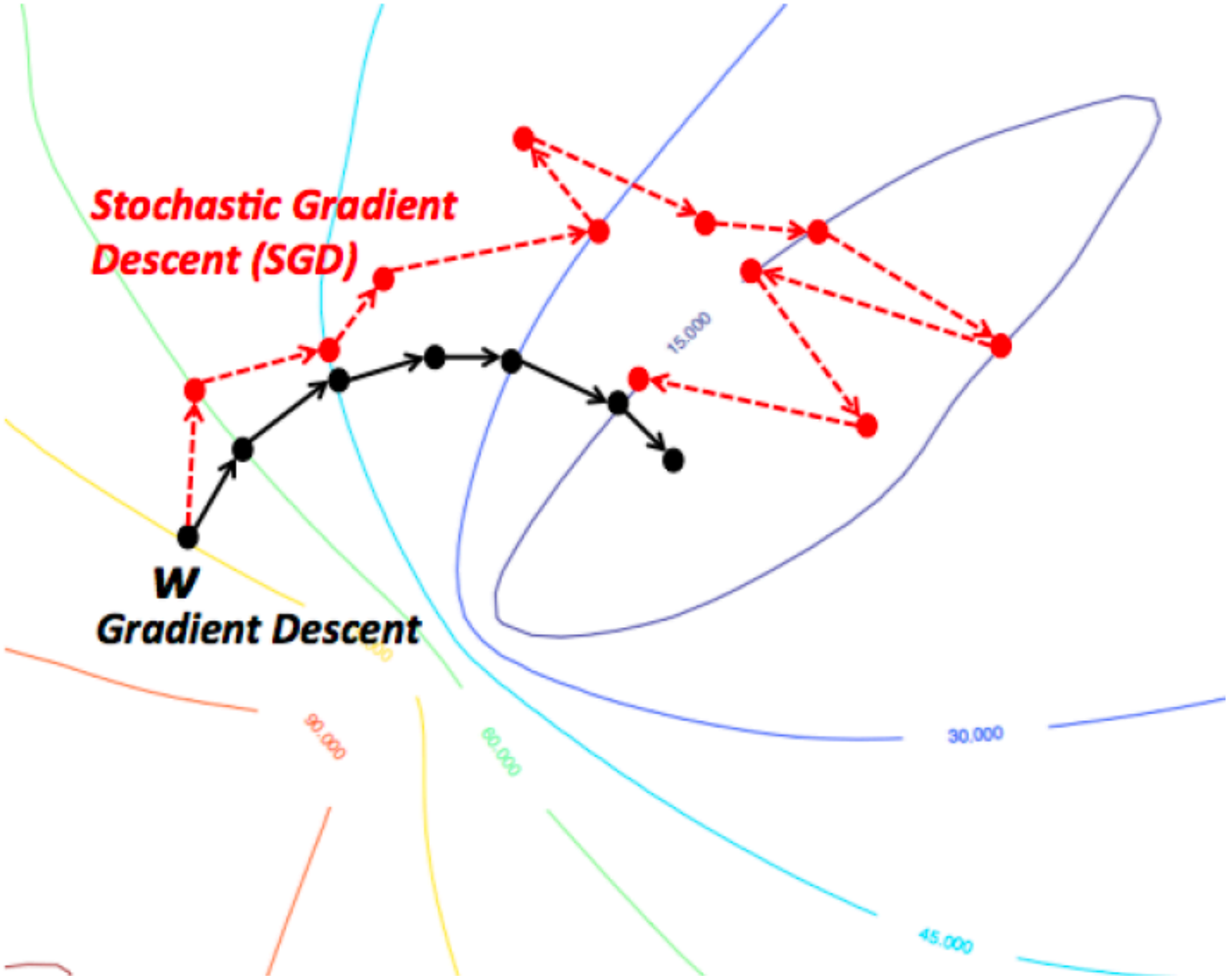
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=256):
    weights_grad = evaluate_gradient(loss_function, batch, params)
    weights += - learning_rate * weights_grad
```

What would be the ideal batch size?

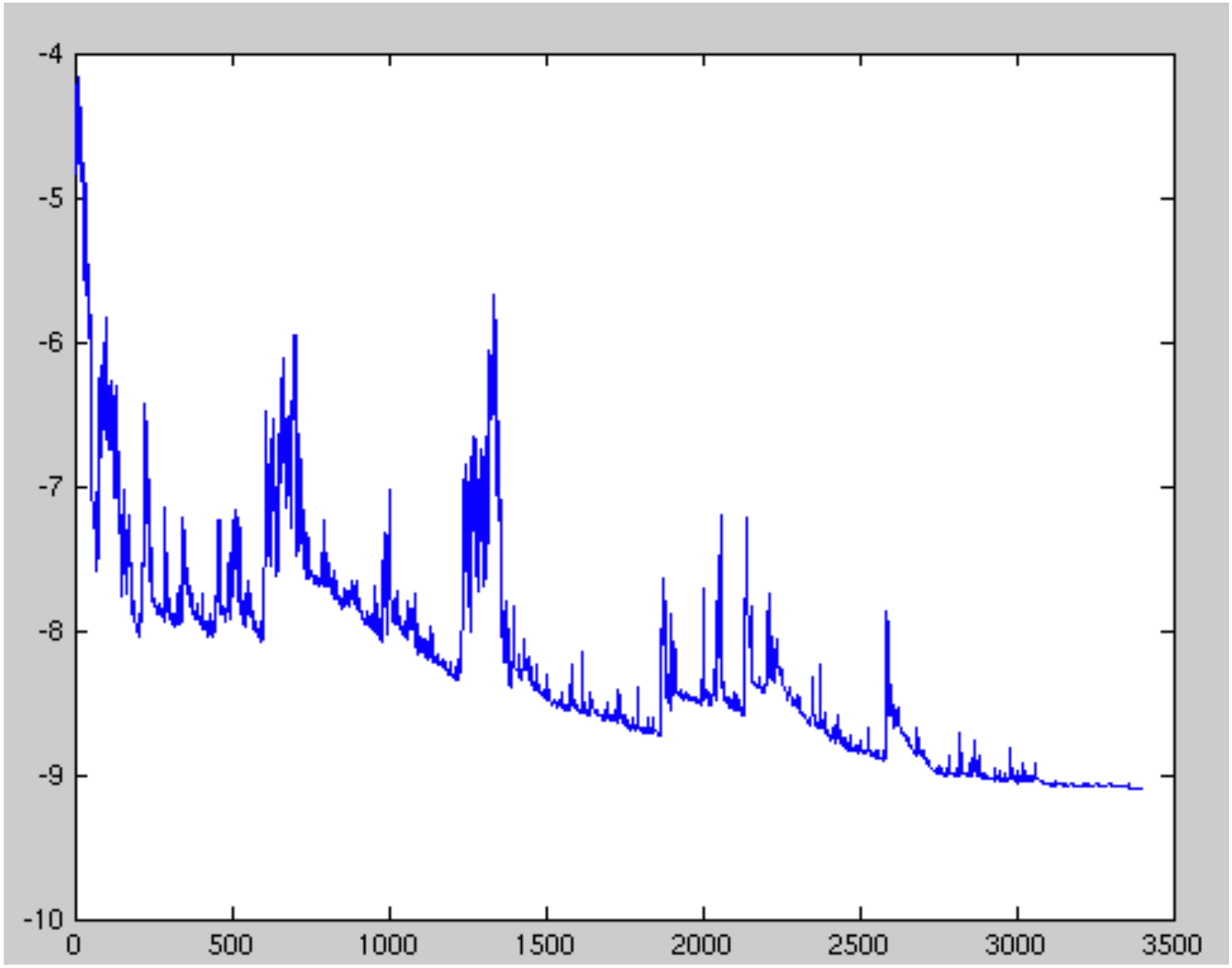
- The size of the mini-batch is a hyper-parameter
- But it is not very common to cross-validate it!
- It is usually based on memory constraints (if any)
- In practice, we use powers of 2 because operations perform faster when their inputs are sized in powers of 2
- Rule of thumb: between 32-256

Challenges

Unlike in gradient descent, the value of the cost function does not necessarily decrease



SGD performs frequent updates with a high variance that cause the objective function to fluctuate



Choosing a proper learning rate can be difficult

- A learning rate that is too small leads to painfully slow convergence,

- A learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.

Thresholds for adjusting learning rate should be defined in advance

- Adjusting the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold.

- These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics.

The same learning rate applies to all parameter updates

- If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

Minimizing highly non-convex functions are challenging

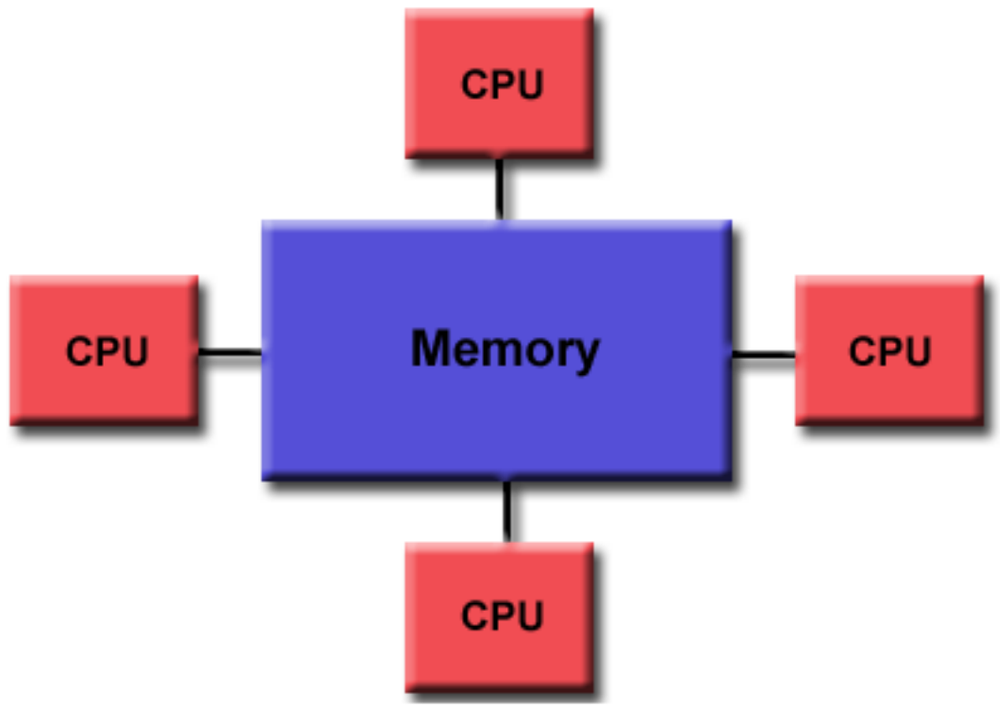
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima.

What computer systems can offer to resolve the challenges?

Parallel Computer Memory Architectures

Shared memory architectures

- Each CPU can access the same memory space
- Multiple processors can operate independently but share the same memory resources
- Changes in a memory location effected by one processor are visible to all other processors.



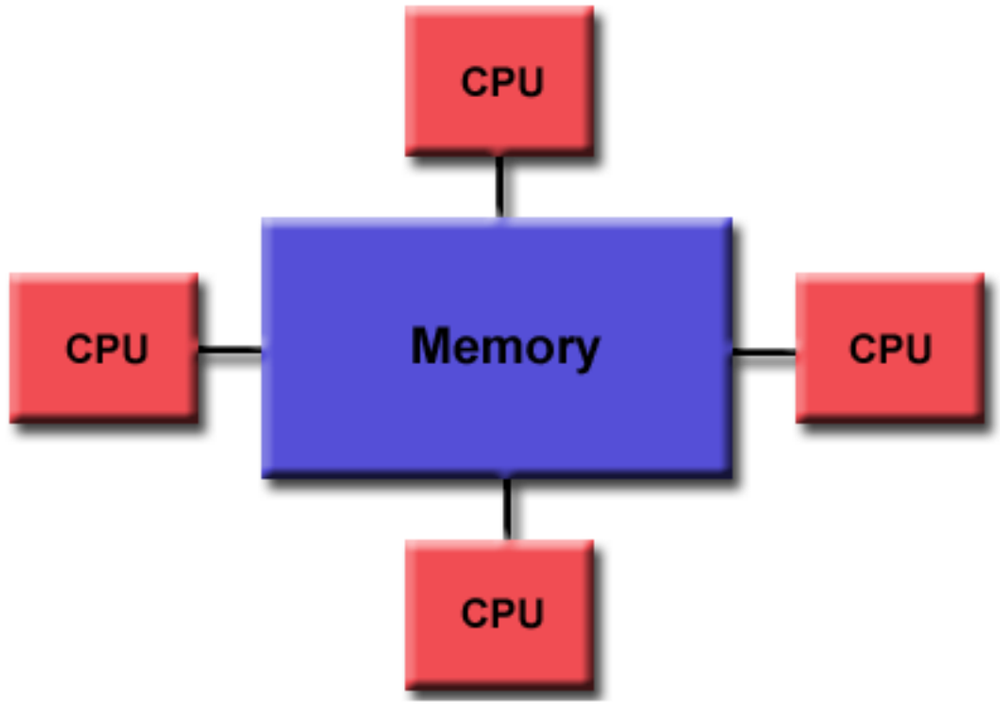
Uniform Memory Access (UMA)

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines

- Identical processors

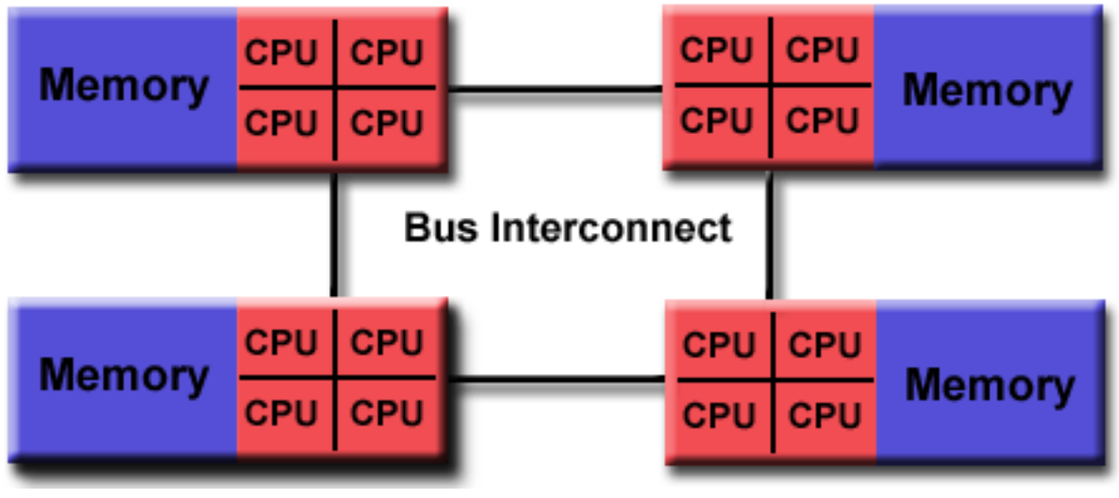
- Equal access and access times to memory

- CC-UMA - Cache Coherent UMA



Non-Uniform Memory Access (NUMA)

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- CC-NUMA - Cache Coherent NUMA



Pros and Cons

- Pros
    - Global address space provides a user-friendly programming perspective to memory
    - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Cons
    - Lack of scalability between memory and CPUs
    - Programmer responsibility for synchronization constructs that ensure "correct" access of global memory

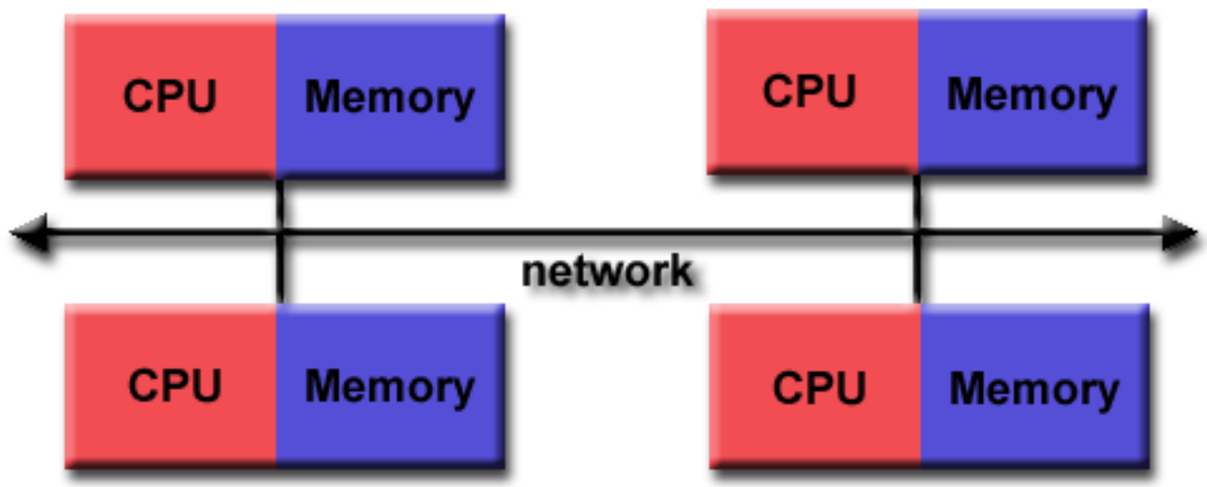
Distributed Memory

- Distributed memory systems require a communication network to connect inter-processor memory

- Processors have their own local memory

- Because each processor has its own local memory, it operates independently

- When a processor needs access to data in another processor, it is usually the task of the programmer



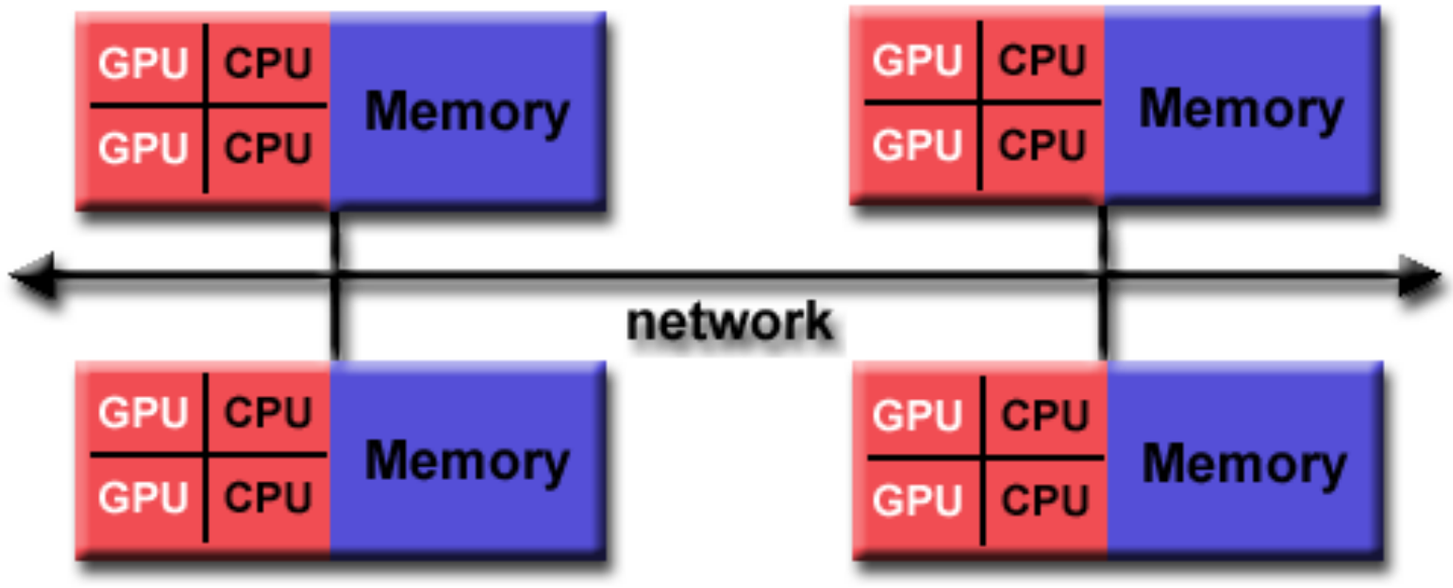
Pros and Cons

- Pros
  - Memory is scalable with the number of processors.
  - Each processor can rapidly access its own memory without interference.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Cons
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.

Hybrid Distributed-Shared Memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.

- The distributed memory component is the networking of multiple shared memory/GPU machines, they only about their own memory - not the memory on another machine.

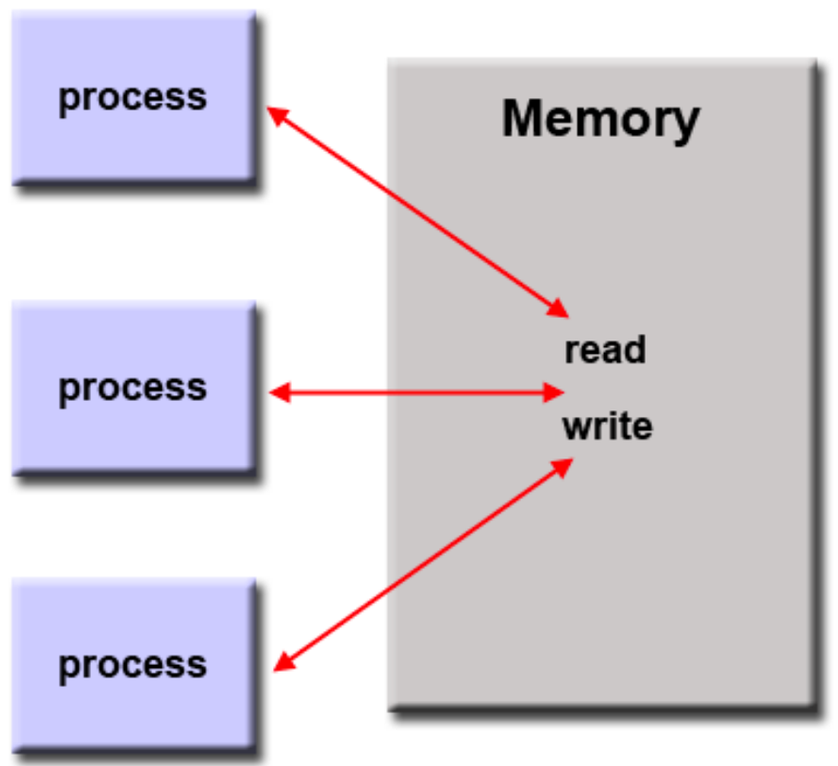


Parallel Programming Models

Shared Memory Model (without threads)

- In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.

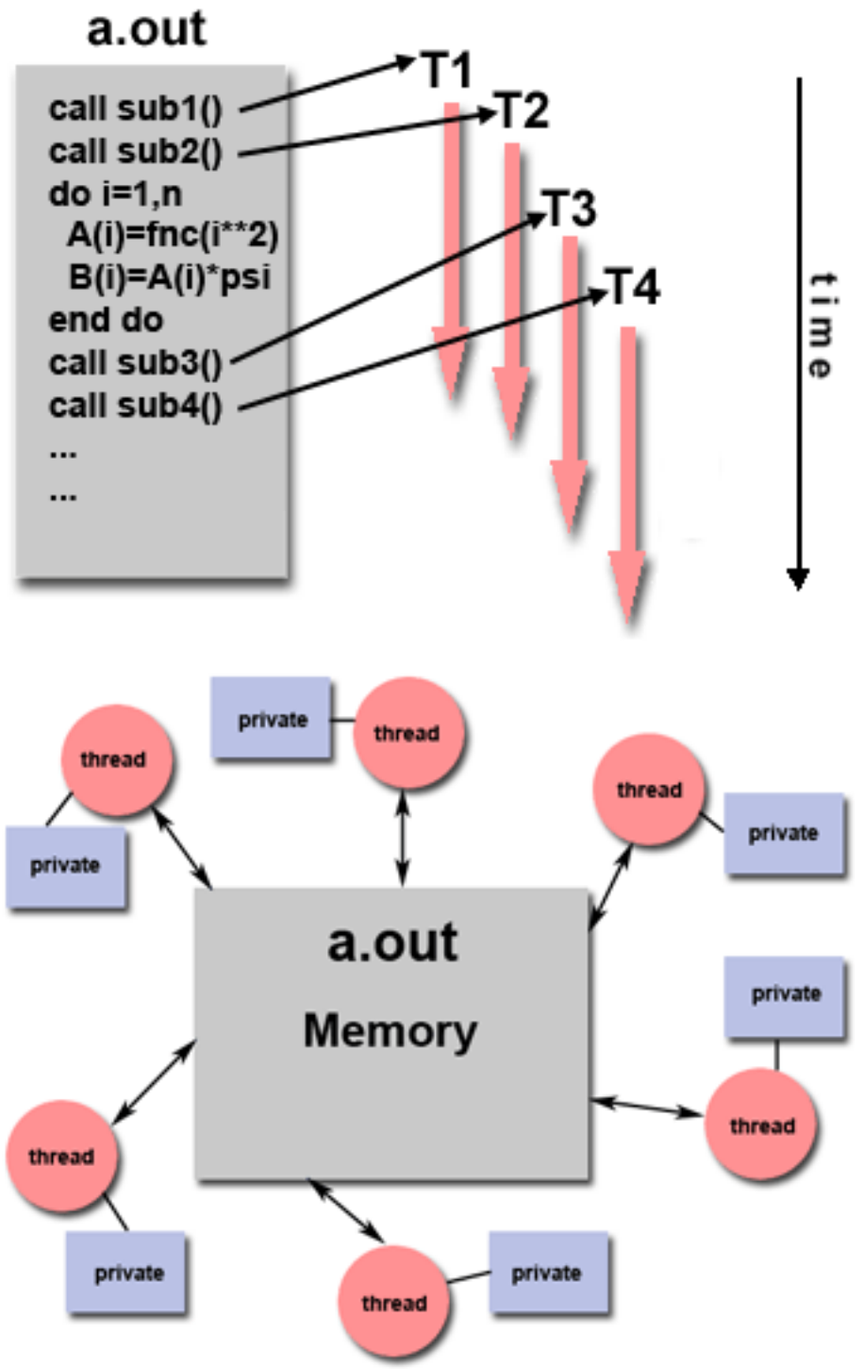
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.



Threads Model

- This programming model is a type of shared memory programming.

- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.



Parallel for loop in OpenMP

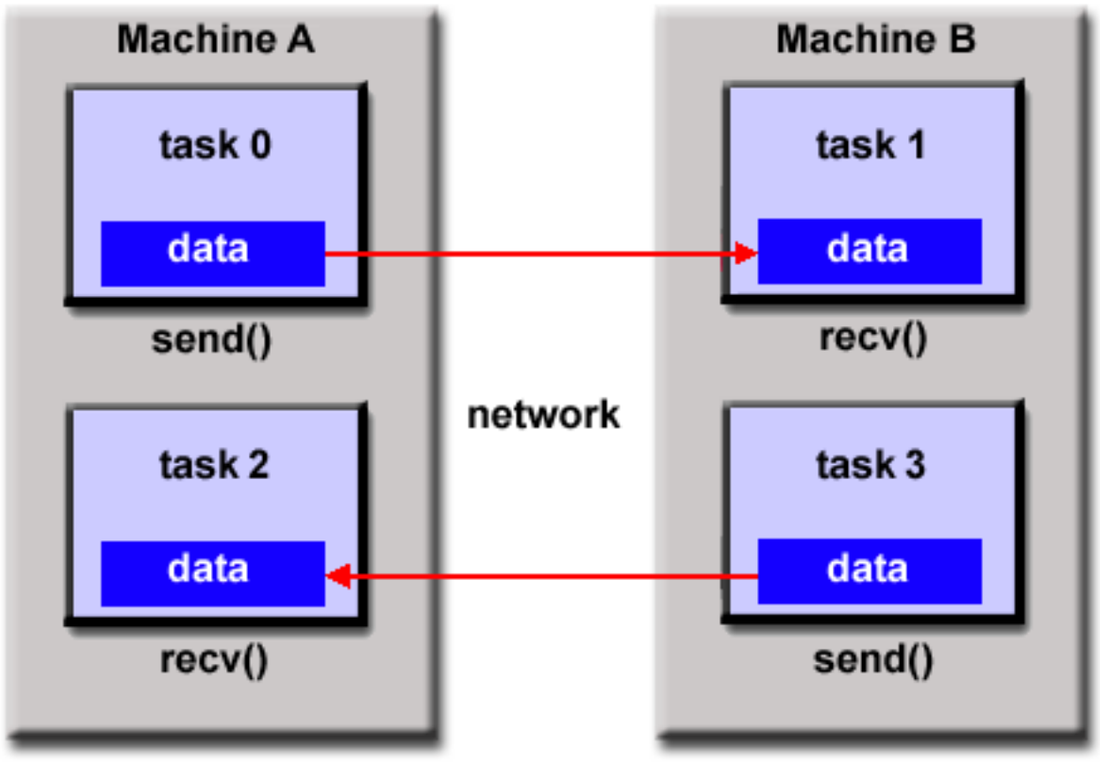
```cpp
#include <iostream>
#include <chrono>

int main() {
    int a, b=0;
#pragma omp parallel for private(a) shared(b)
    for(a=0; a<50; ++a)
    {
#pragma omp atomic
        b += a;
    }
}
```

Distributed Memory / Message Passing Model

- A set of tasks that use their own local memory during computation.

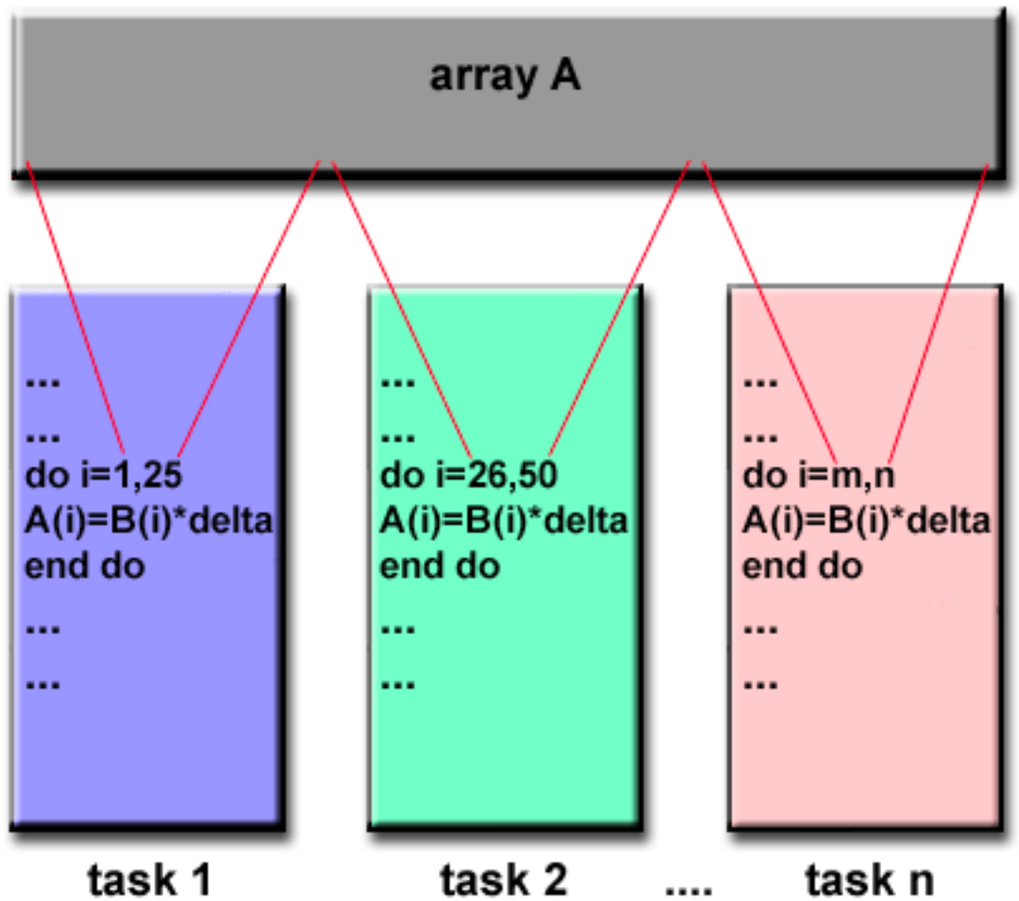
- Tasks exchange data through communications by sending and receiving messages.

- Data transfer usually requires cooperative operations to be performed by each process.



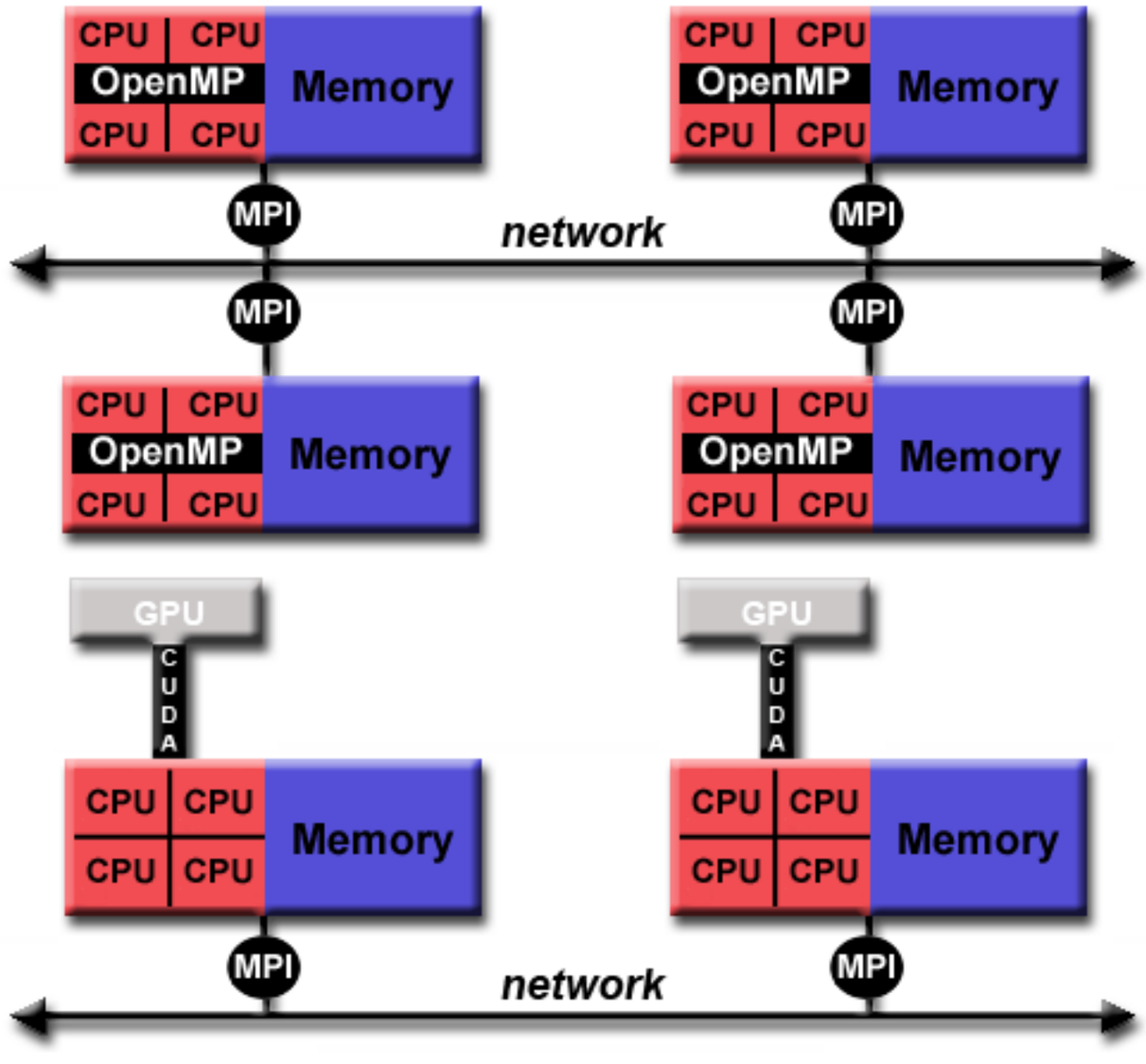
Data Parallel Model

- Address space is treated globally
- Most of the parallel work focuses on performing operations on a data set.
- A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element”.



Hybrid Model

- A hybrid model combines the previously described programming models.



Now that we learned about computer systems approaches for parallelization, how we can use them to parallelize SGD?

Lets discuss what are the core challenges?

When we need to add more parallelism to our computations?

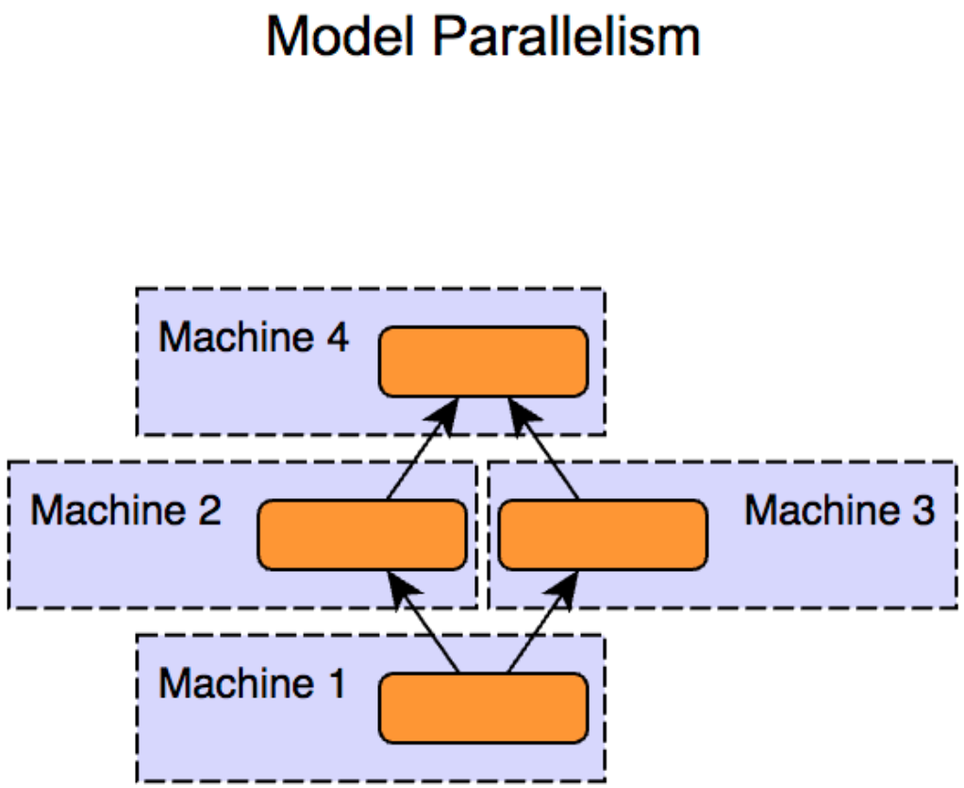
- When training really deep models (16M parameters)
- On really large datasets (17B Examples)

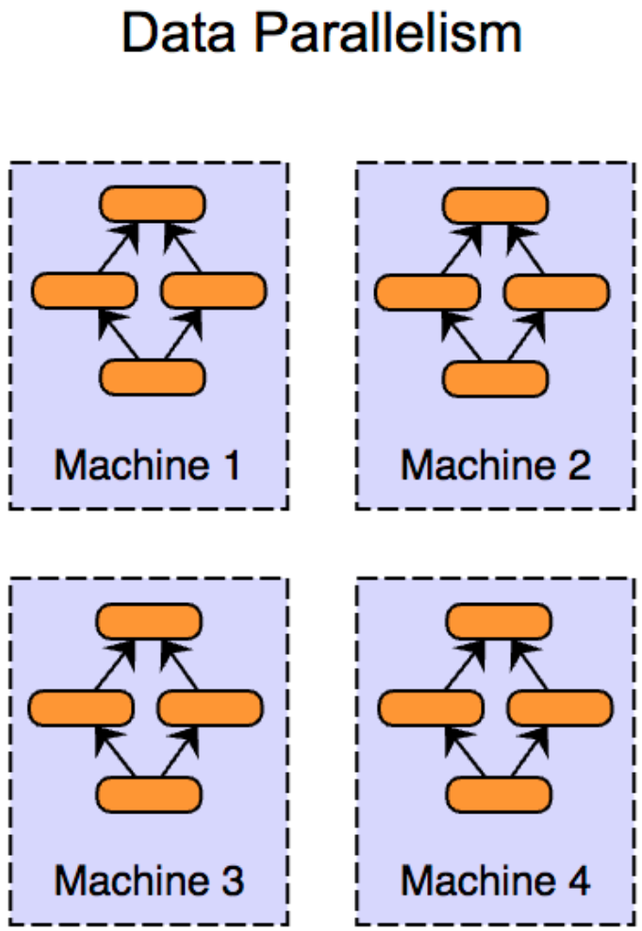
What has been explored so far and what is the current trend

- Traditionally: Distributing linear algebra operations on GPUs,

- Now: How to use multiple machines

Existing parallelizations





We will mainly focus on data parallelism in the rest of lecture

Parallel Gradient Descent

Parallel Gradient Descent

repeat until convergence {

$$\theta \leftarrow \theta - \gamma \nabla_\theta J(\theta)$$

}

- Gradient computation is usually “embarrassingly parallel”

- Why?

- Consider, for example, empirical risk minimization as a learning algorithm:

$$R_{\textbf{emp}}(h) = \frac{1}{m} \sum_{i=1}^{m} L(h(x_i), y_i) \, .$$

$$\hat{h} = \arg\min_{h \in \mathcal{H}} R_{\textbf{emp}}(h) \, .$$

Parallel Gradient Descent

- Partition the dataset into k subsets S1 , . . . , Sk
- Each machine or CPU computes $\Sigma_{i \in S_i} \nabla_\theta L(h(x_i), y_i)$
- Aggregate local gradients to get the global gradient

$$\nabla_\theta L(\,\cdot\,) = \frac{1}{k}(\Sigma_{i \in S_1} \nabla_\theta L(h(x_i), y_i) + \cdots + \Sigma_{i \in S_k} \nabla_\theta L(h(x_i), y_i))$$

Parallel Stochastic Gradient

repeat until convergence {

  for $i := 1,2,...,m\{$

    $\theta \leftarrow \theta - \gamma \nabla J(\theta; x^{(i)}; y^{(i)})$

  $\}$

$\}$

- Computation of $\nabla J(\theta; x^{(i)}; y^{(i)})$ only depends on the i-th sample—usually cannot be parallelized.

- Parallelizing SGD is a not easy.

Parallel Mini-batch SGD

repeat until convergence {

for $i := 1,2,...,m$ {

$$\theta \leftarrow \theta - \gamma \nabla J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

}

}

- Let S = S1 ∪ S2 ∪ · · · ∪ Sk

- Calculate mini-batch updates in parallel

Asynchronous SGD (shared memory)

Each thread repeatedly do these updates: {

for $i := 1,2,...,m\{$

$$\theta \leftarrow \theta - \gamma \nabla J(\theta; x^{(i)}; y^{(i)})$$

}

}

How using shared memory architecture?

- Main trick: in shared memory systems, every threads can access the latest gradient value

- Langford et al., “Slow learners are fast”. In NIPS 2009

- “Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”, NIPS 2011.

Synchronous Distributed SGD

Synchronous SGD

1: **while** $t < T$ **do**
2: Get: a minibatch $(x, y) \sim \chi$ of size M/R.
3: Compute: $\nabla \mathcal{L}(y, F(x; W_t))$ on local (x, y).
4: AllReduce: sum all $\nabla \mathcal{L}(y, F(x; W_t))$ across replicas into ΔW_t
5: Update: $W_{t+1} = W_t - \alpha \frac{\Delta W_t}{R}$
6: $t = t + 1$
7: (Optional) Synchronize: W_{t+1} to avoid numerical errors

Synchronous SGD

Replica

Replica

Replica

Start with W_t

Replica
W_t,

Replica
W_t,

Replica
W_t,

Compute $\nabla\mathcal{L}$ with local minibatch

Replica

W_t,$\nabla\mathcal{L}$,

Replica

W_t,$\nabla\mathcal{L}$,

Replica

W_t,$\nabla\mathcal{L}$,

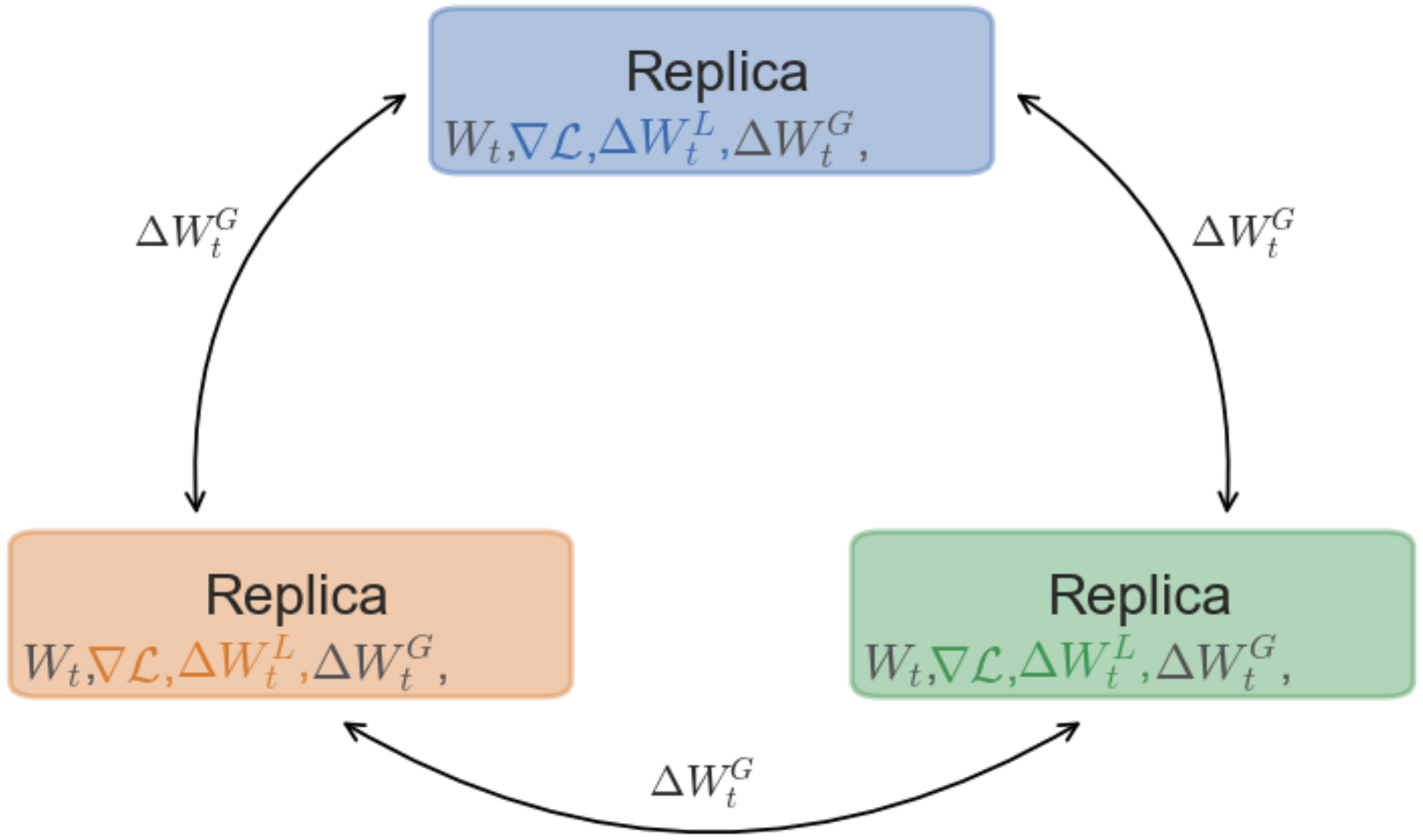
Compute ΔW_t^L with favorite optimizer

Replica
$W_t, \nabla \mathcal{L}, \Delta W_t^L,$

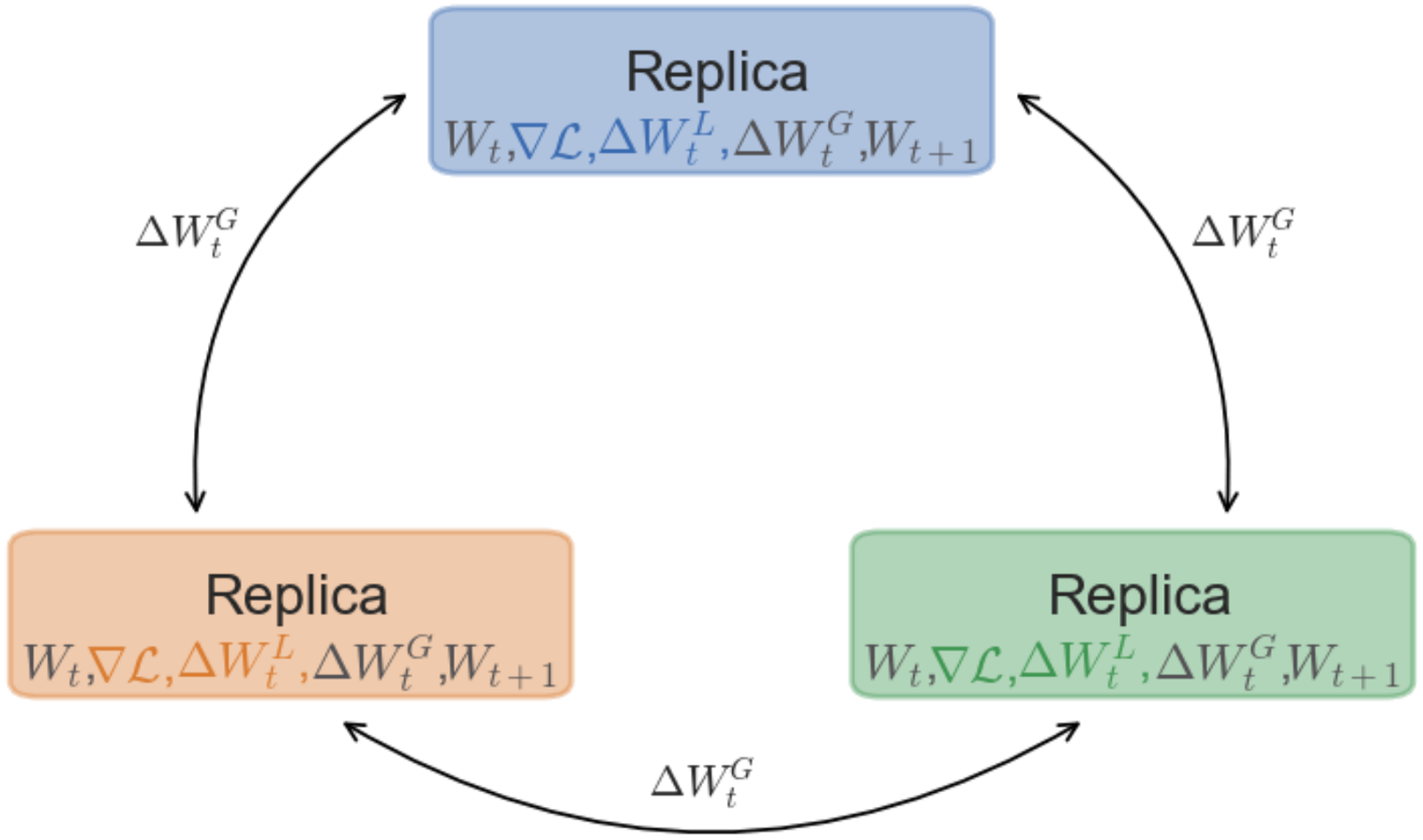
Replica
$W_t, \nabla \mathcal{L}, \Delta W_t^L,$

Replica
$W_t, \nabla \mathcal{L}, \Delta W_t^L,$

AllReduce the ΔW_t^L across replicas



Update to W_{t+1} with normalized ΔW_t



Pros

- The computation is completely deterministic.
- We can work with fairly large models and large batch sizes even on memory-limited GPUs.
- It’s very simple to implement, and easy to debug and analyze.

Cons

- This path to parallelism puts a strong emphasis on HPC, and the hardware that is in use.

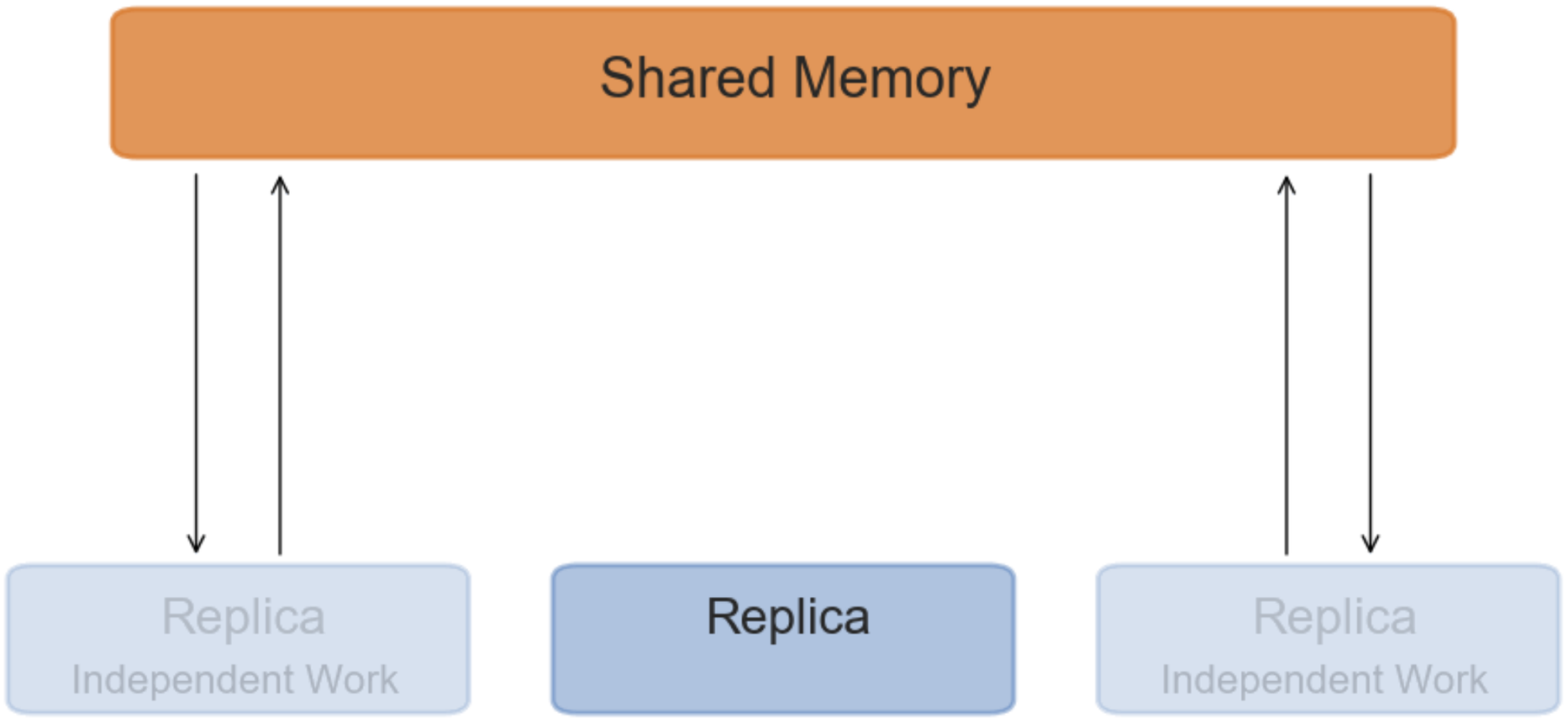
- It will be challenging to obtain a decent speedup unless you are using industrial hardware.

- And even if you were using such a hardware, the choice of communication library, reduction algorithm, and other implementation details (e.g., data loading and transformation, model size, ...) will have a strong effect on the kind of performance gain you will encounter.

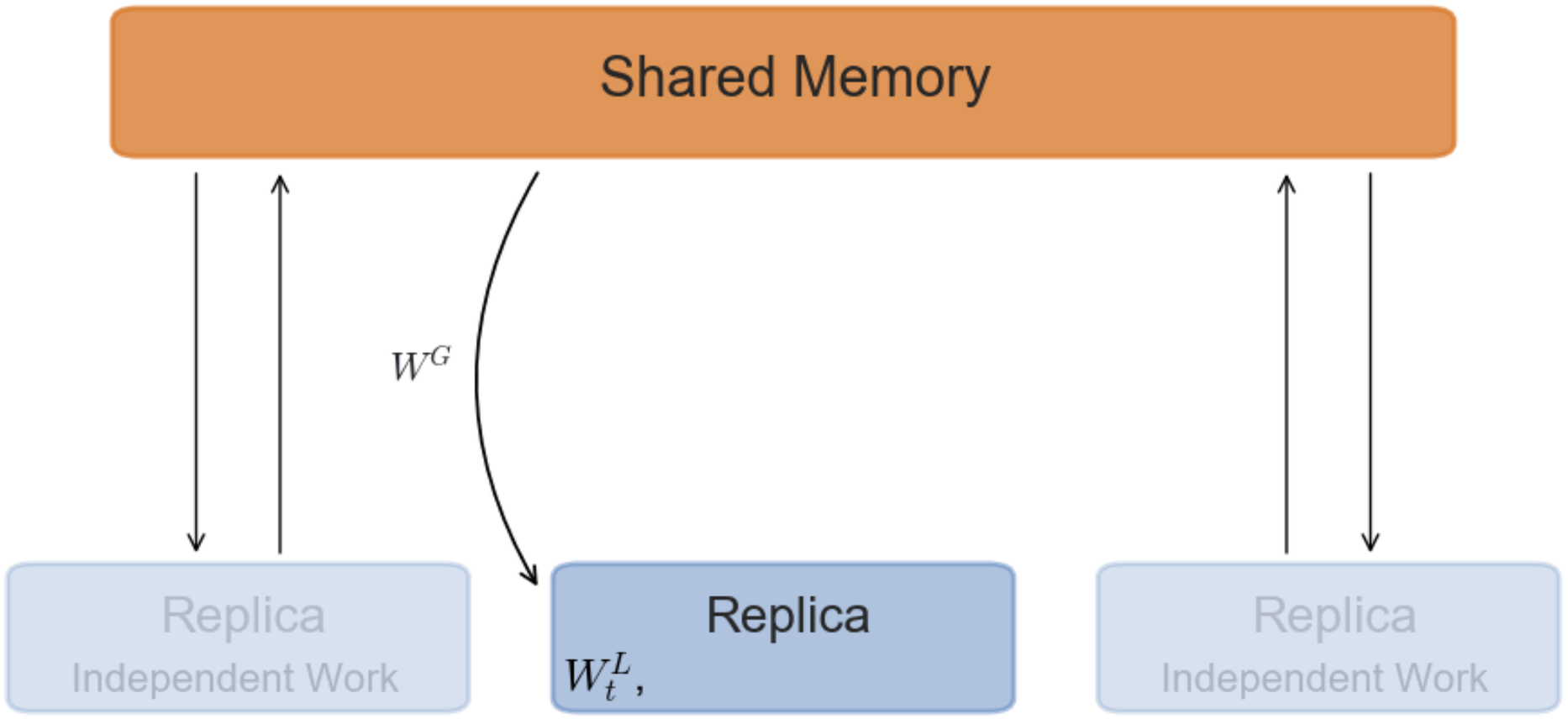
Asynchronous SGD

1: **while** $t < T$ **do**
2: Get: a minibatch $(x, y) \sim \chi$ of size M/R.
3: Copy: Global W_t^G into local W_t^L.
4: Compute: $\nabla \mathcal{L}(y, F(x; W_t^L))$ on (x, y).
5: Set: $\Delta W_t^L = \alpha \cdot \nabla \mathcal{L}(y, F(x; W_t^L))$
6: Update: $W_{t+1}^G = W_t^G - \Delta W_t^L$
7: $t = t + 1$

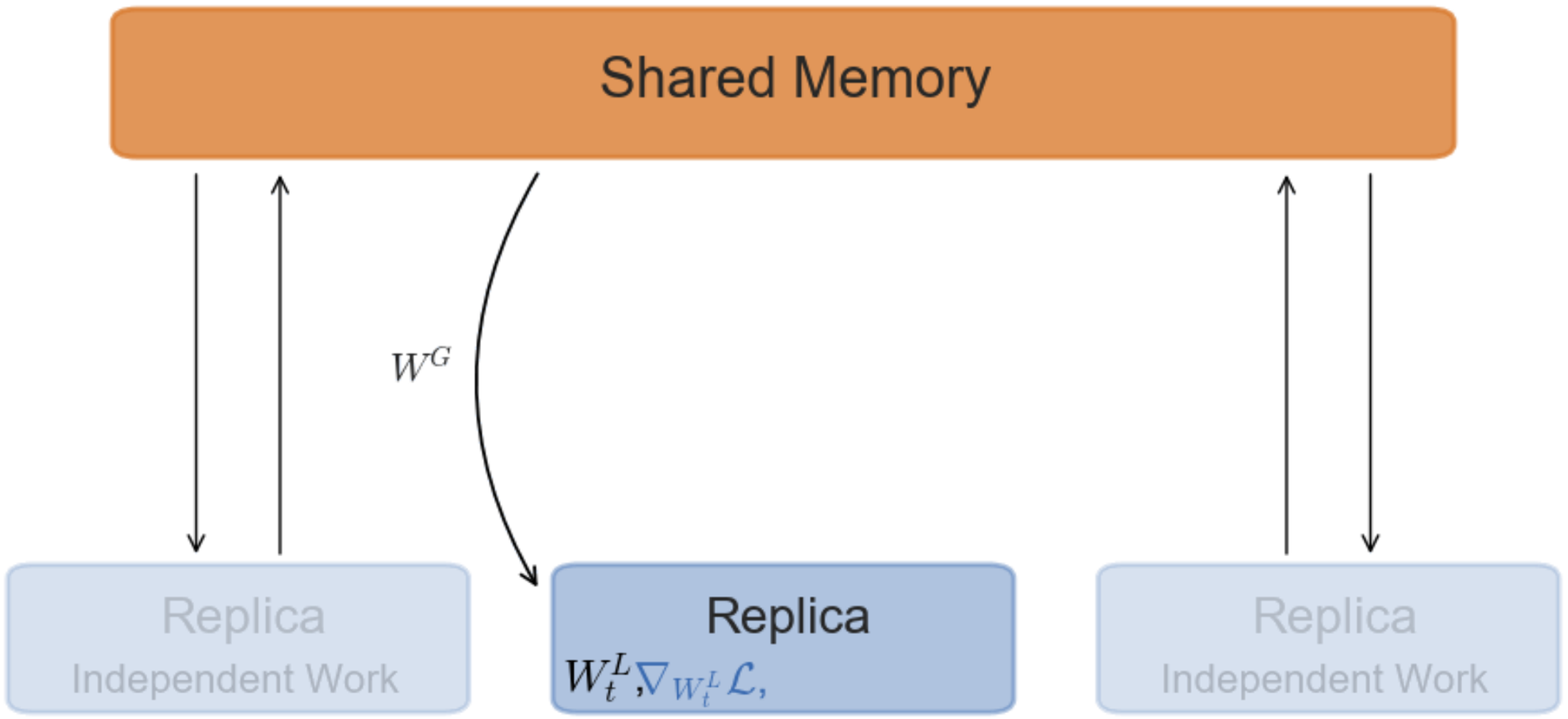
Asynchronous SGD



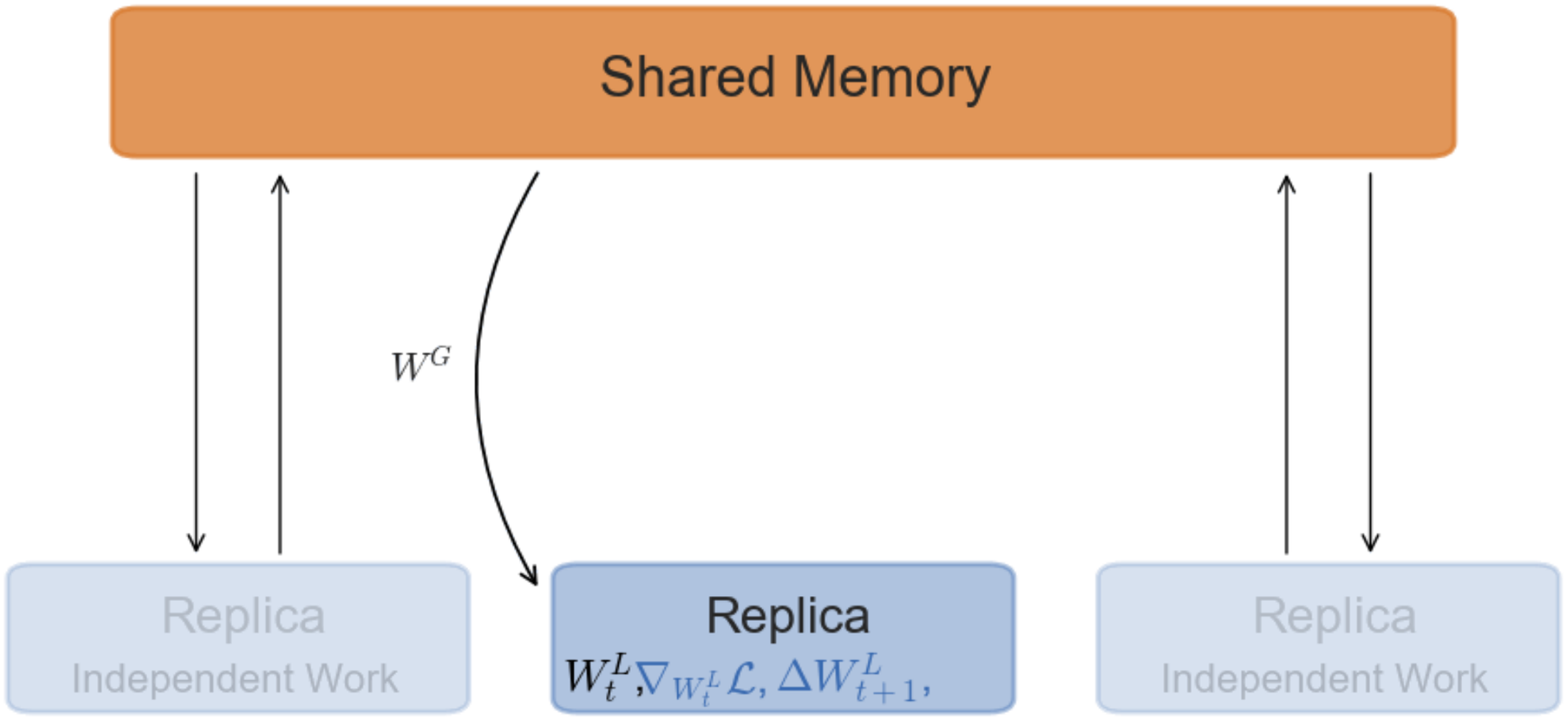
Copy W^G into W_t^L



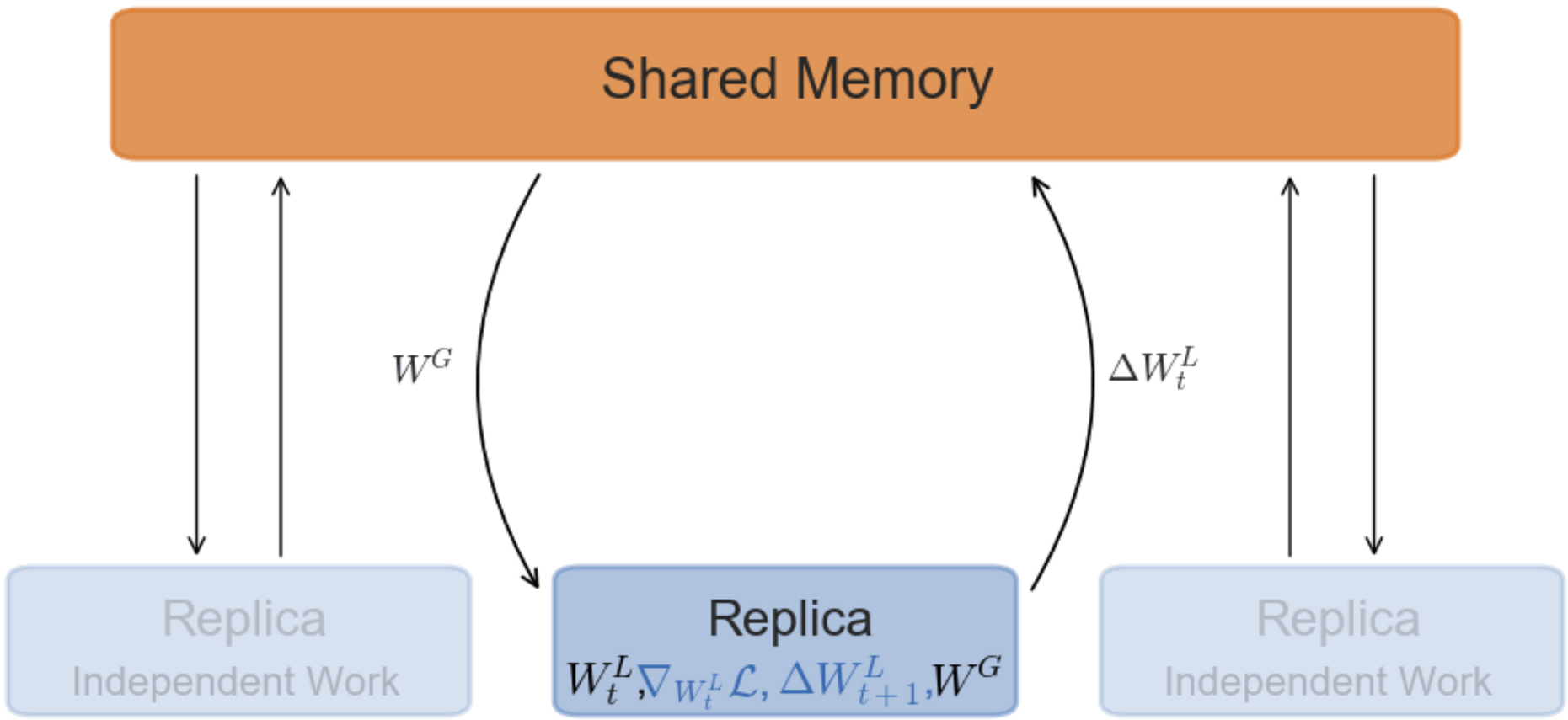
Compute gradients $\nabla_{W_t^L}\mathcal{L}$ locally



Compute update ΔW_{t+1}^{L} with favorite optimizer



Apply local update ΔW_{t+1}^L to global params W^G



Pros

- The advantage of adding asynchrony to our training is that replicas can work at their own pace,

- without waiting for others to finish computing their gradients.

Cons

- We have no guarantee that while one replica is computing the gradients with respect to a set of parameters, the global parameters will not have been updated by another one.

- If this happens, the global parameters will be updated with stale gradients - gradients computed with old versions of the parameters.

How to solve staleness?

- [Zhang & al.] suggested to divide the gradients by their staleness. By limiting the impact of very stale gradients, they are able to obtain convergence almost identical to a synchronous system.

Zhang, W., Gupta, S., Lian, X., Liu, J.: Staleness-aware async-sgd for distributed deep learning. arXiv preprint arXiv:1511.05950. (2015)

How to solve staleness?

- Each replica executes k optimization steps locally, and keeps an aggregation of the updates.

- Once those k steps are executed, all replicas synchronize their aggregated update and apply them to the parameters before the k steps.

Zhang, S., Choromanska, A.E., LeCun, Y.: Deep learning with elastic averaging sgd. In: Advances in neural information processing systems. pp. 685–693 (2015)

Implementation

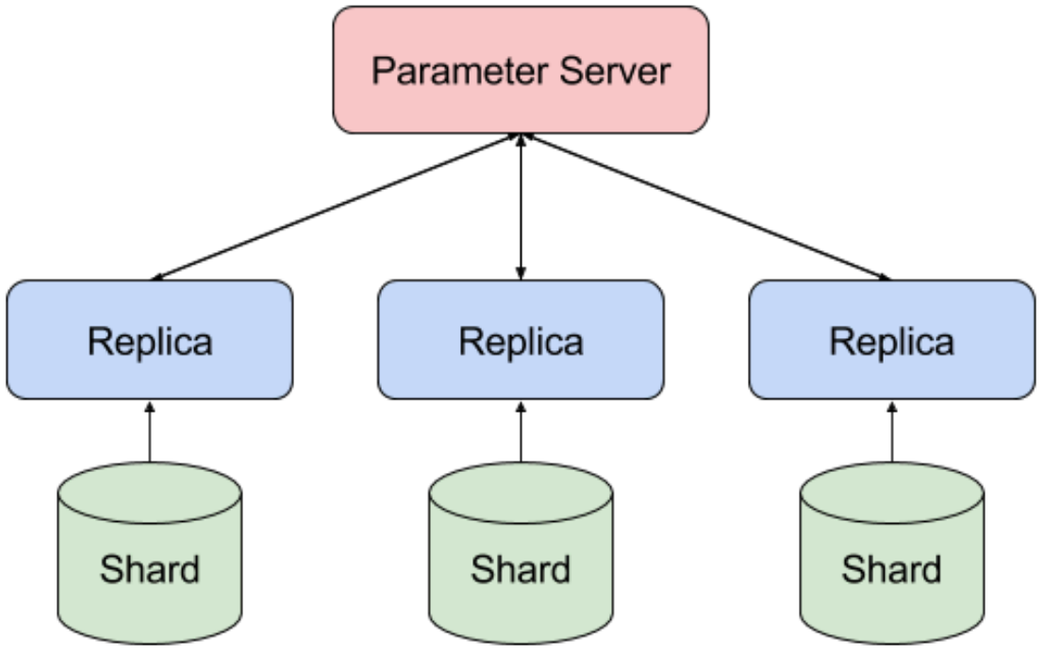
What is the first decision then?

What is the decision?

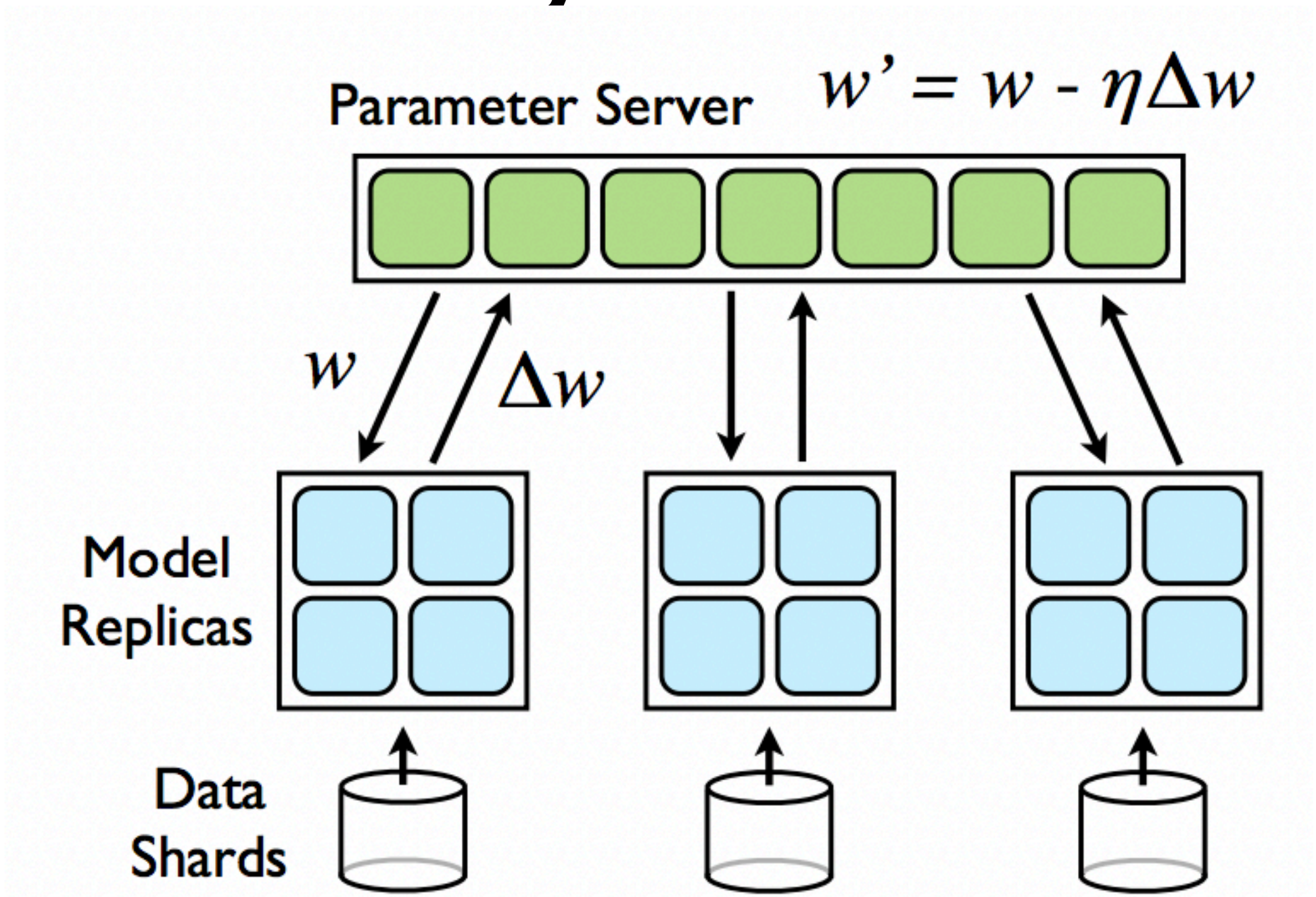
- The first decision to make is how to setup the architecture of the system

What options do we have?

Parameter server

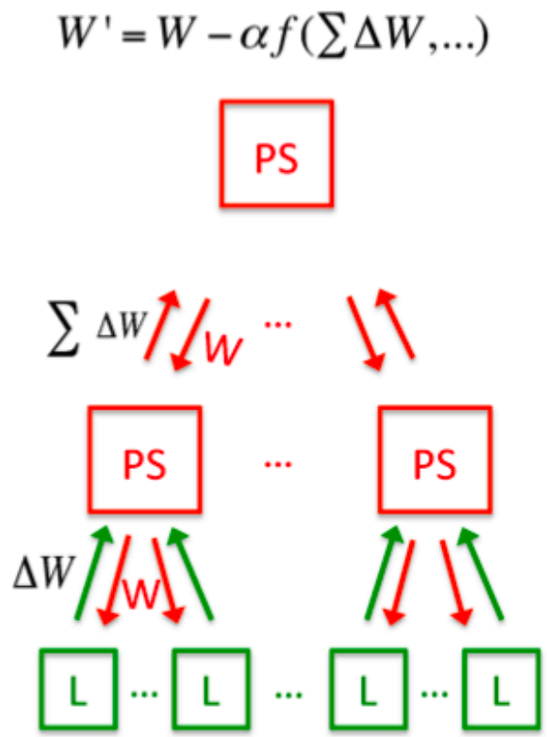


How using distributed memory architecture?

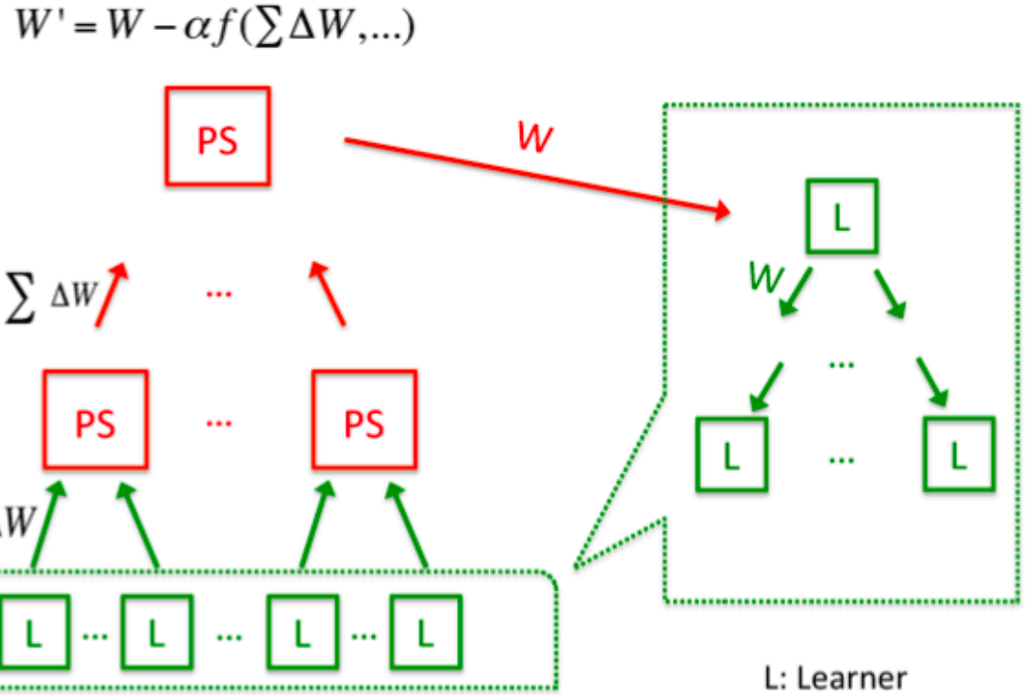


Dean et al., “Large Scale Distributed Deep Networks”, in NIPS 2012

Adding more hierarchy



L: Learner
PS: Parameter Server



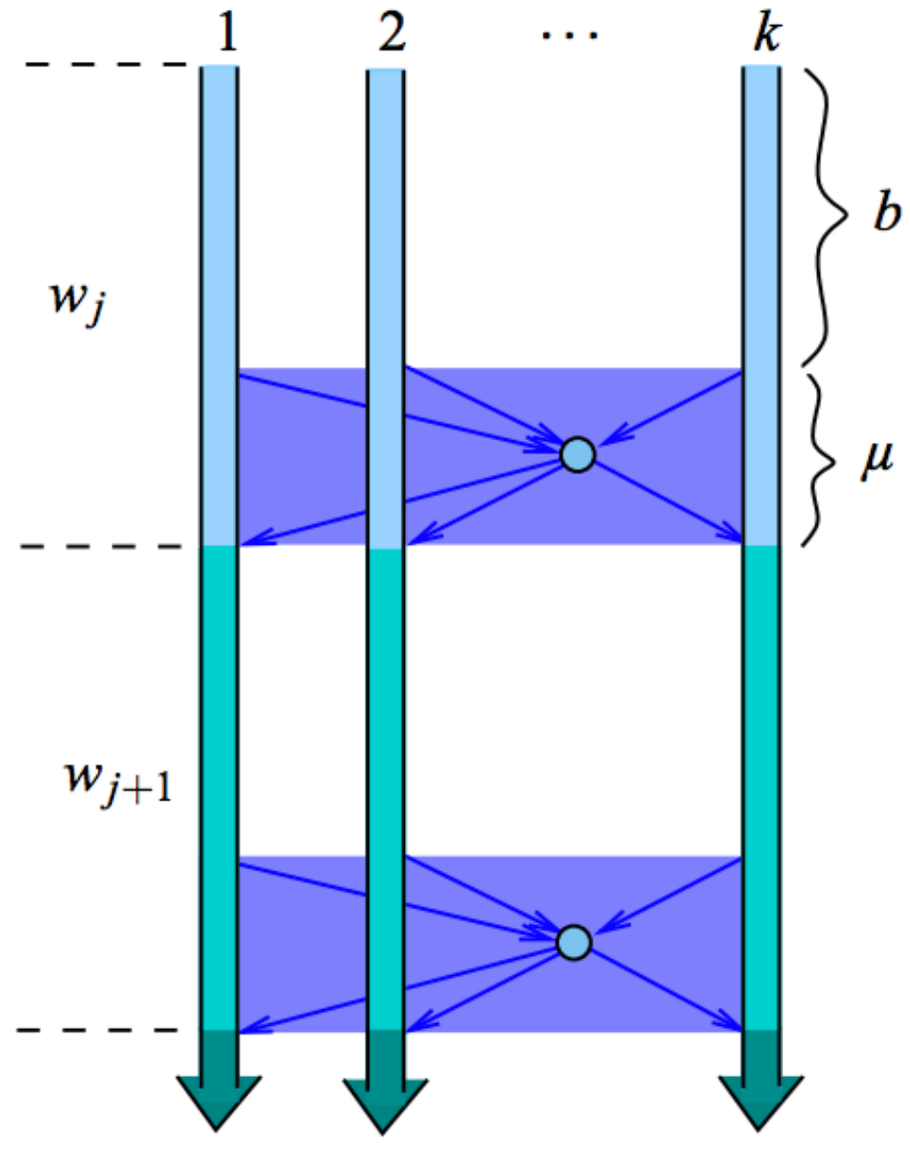
L: Learner
PS: Parameter Server

Gupta, et. al, “Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study”, 2016

Mini-batch SG on distributed systems

- Can we avoid wasting communication time?

- Use non-blocking network IO: Keep computing updates while aggregating the gradient

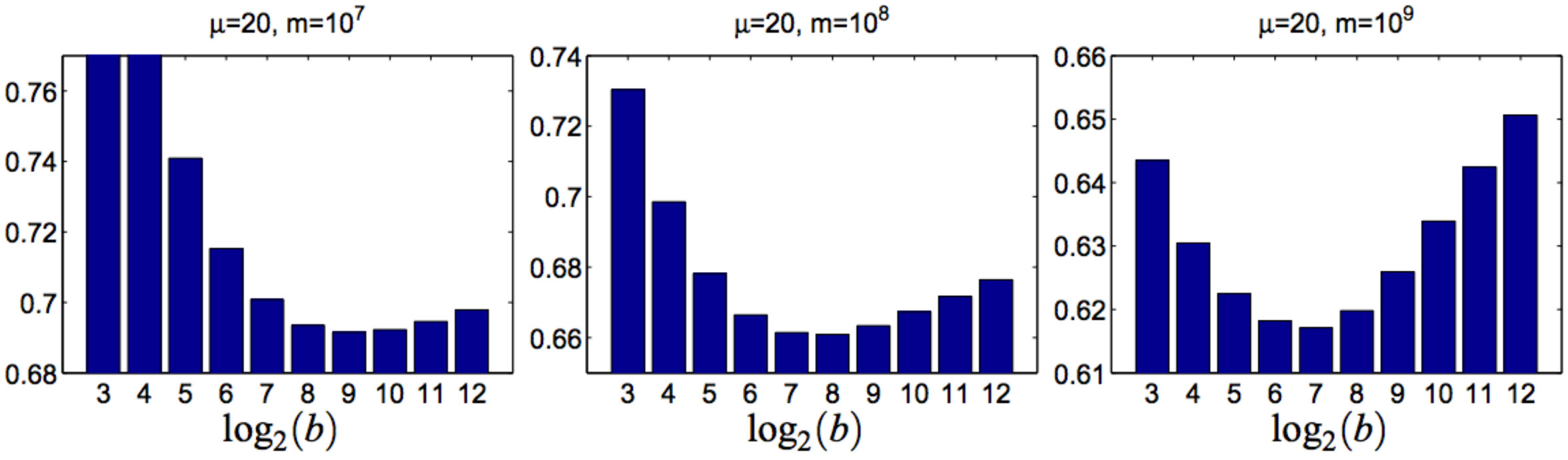


Distributed mini-batch

Algorithm 3: Distributed mini-batch (DMB) algorithm (running on each node).

for $j = 1, 2, \ldots$ **do**
  initialize $\hat{g}_j := 0$
  for $s = 1, \ldots, b/k$ **do**
    predict w_j
    receive input z sampled i.i.d. from unknown distribution
    suffer loss $f(w_j, z)$
    compute $g := \nabla_w f(w_j, z)$
    $\hat{g}_j := \hat{g}_j + g$
  end
  call the distributed vector-sum to compute the sum of $\hat{g}_j$ across all nodes
  receive μ/k additional inputs and continue predicting using w_j
  finish vector-sum and compute average gradient $\bar{g}_j$ by dividing the sum by b
  set $(w_{j+1}, a_{j+1}) = \phi(a_j, \bar{g}_j, \alpha_j)$
end

Optimal Mini-Batch Size

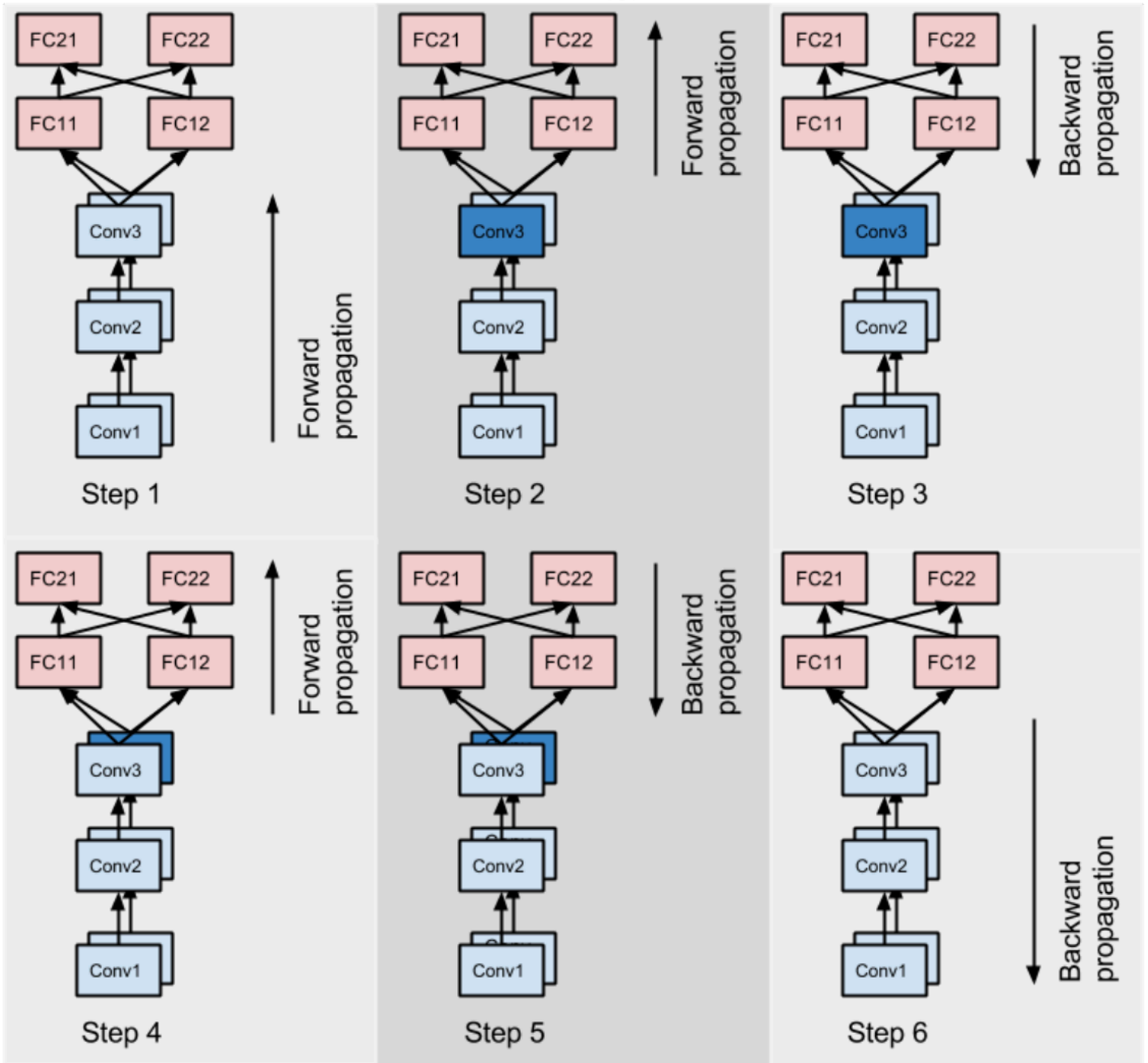


Consider type of layers when parallelizing

- Conv layers parallelize particularly well given that they are quite compute heavy with respect to the number of parameters they contain.

- This is a desirable property of the network, since you want to limit the time spent in communication

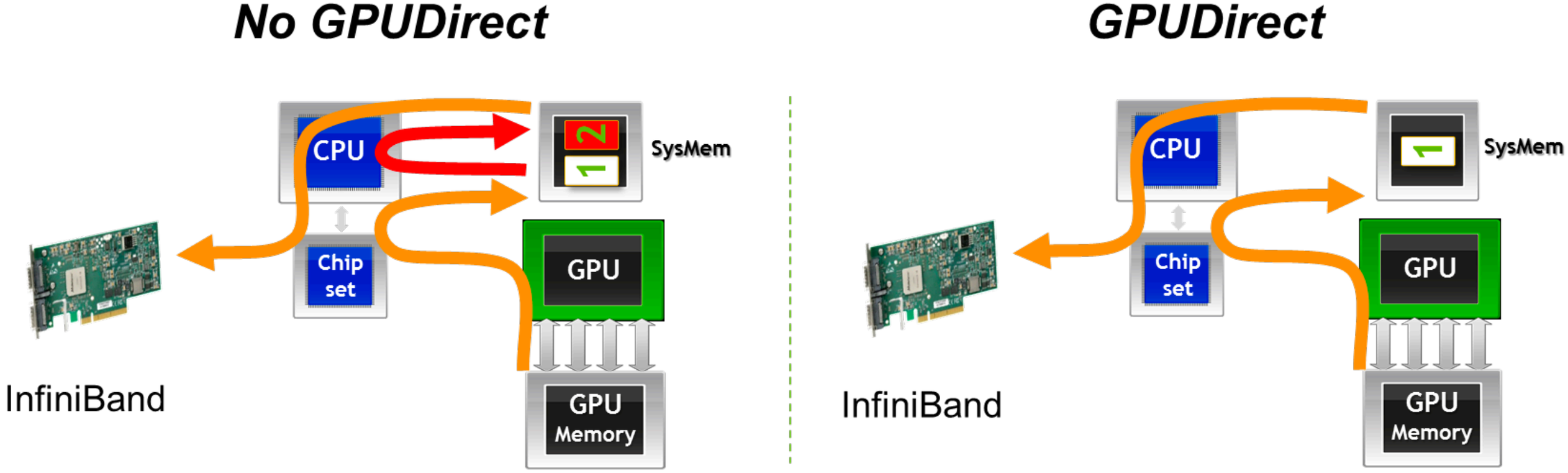
- Convolutions achieve just that since they re-multiply feature maps all over the input.



Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997. (2014)

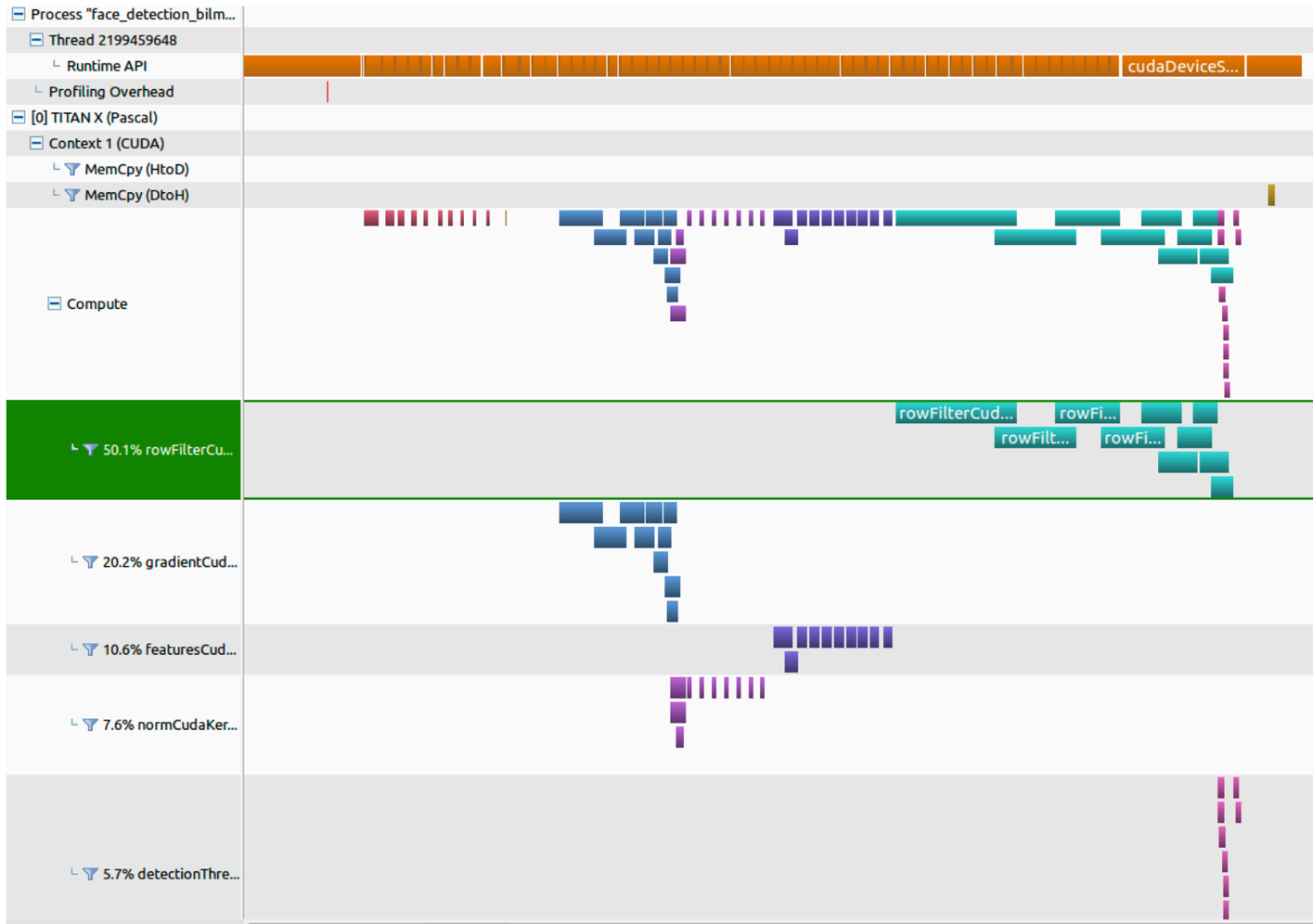
Tricks

- Device-to-Device Communication



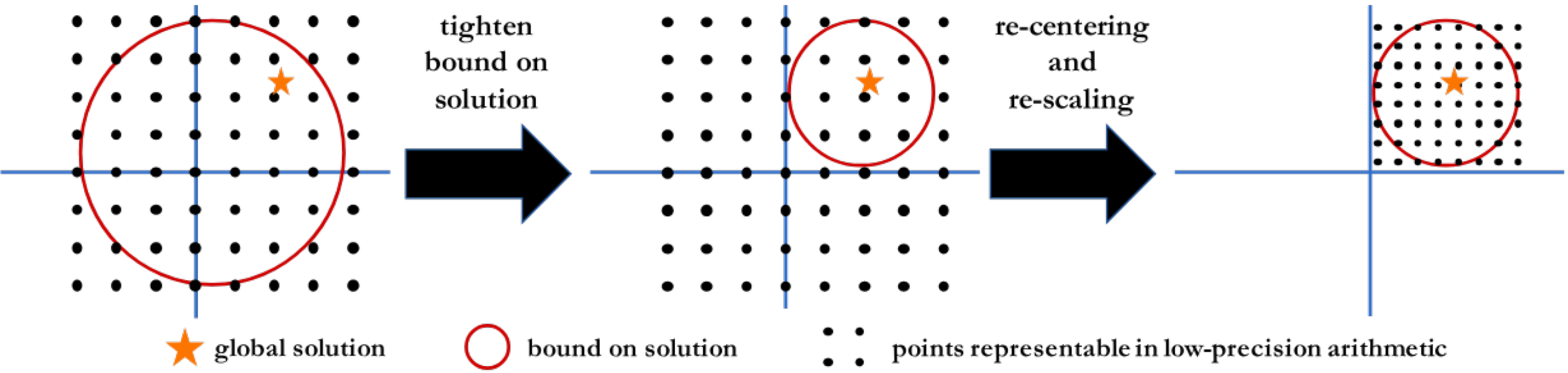
Tricks

- Overlapping Computation
- E.g., synchronizing the gradients of the current layer while computing the gradients of the next one.



Tricks

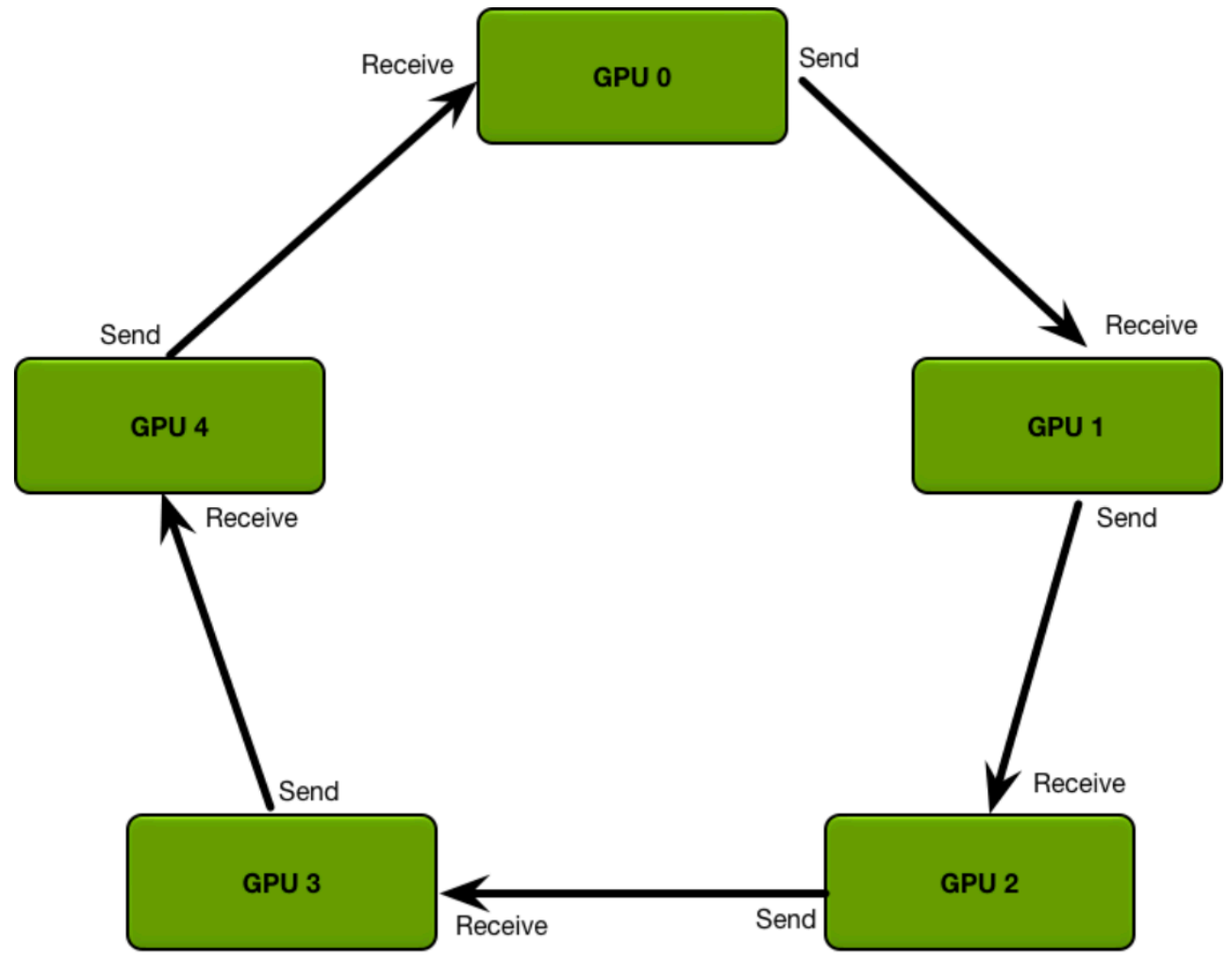
- Approximate computing



Christopher De Sa, “High-Accuracy Low-Precision Training”, 2018.

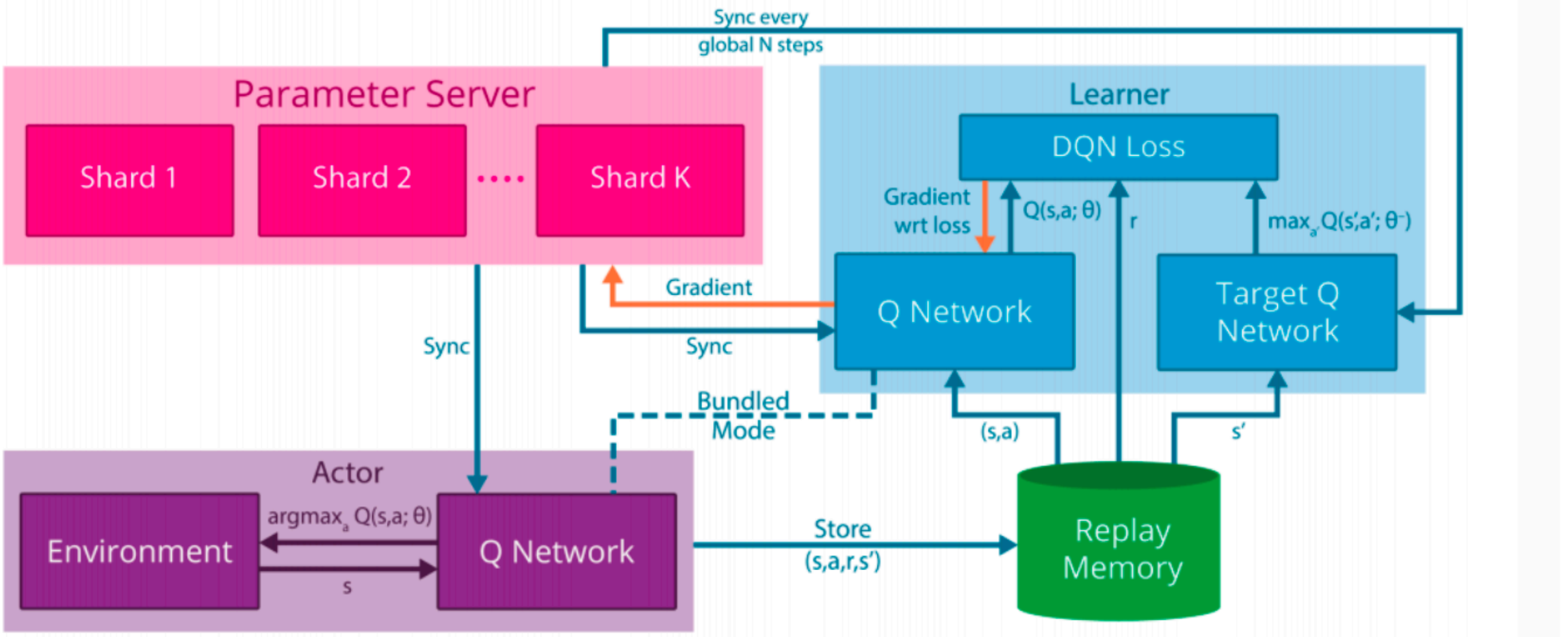
Tricks

- Reduction algorithm



John Langford, “Allreduce for Parallel Learning”, 2017

Other applications?



Summary

- We reviewed 3 variations of gradient descent
- We reviewed common challenges during training
- We examined computer system memory architectures and parallel programming models
- We discussed approaches to scale up SGD using parallel computations
- **Next**: Real-world case study about how Uber uses these ideas to make their deep learning tasks distributed over multiple machines to scale up training process

Distributed Machine Learning

Uber Case Study

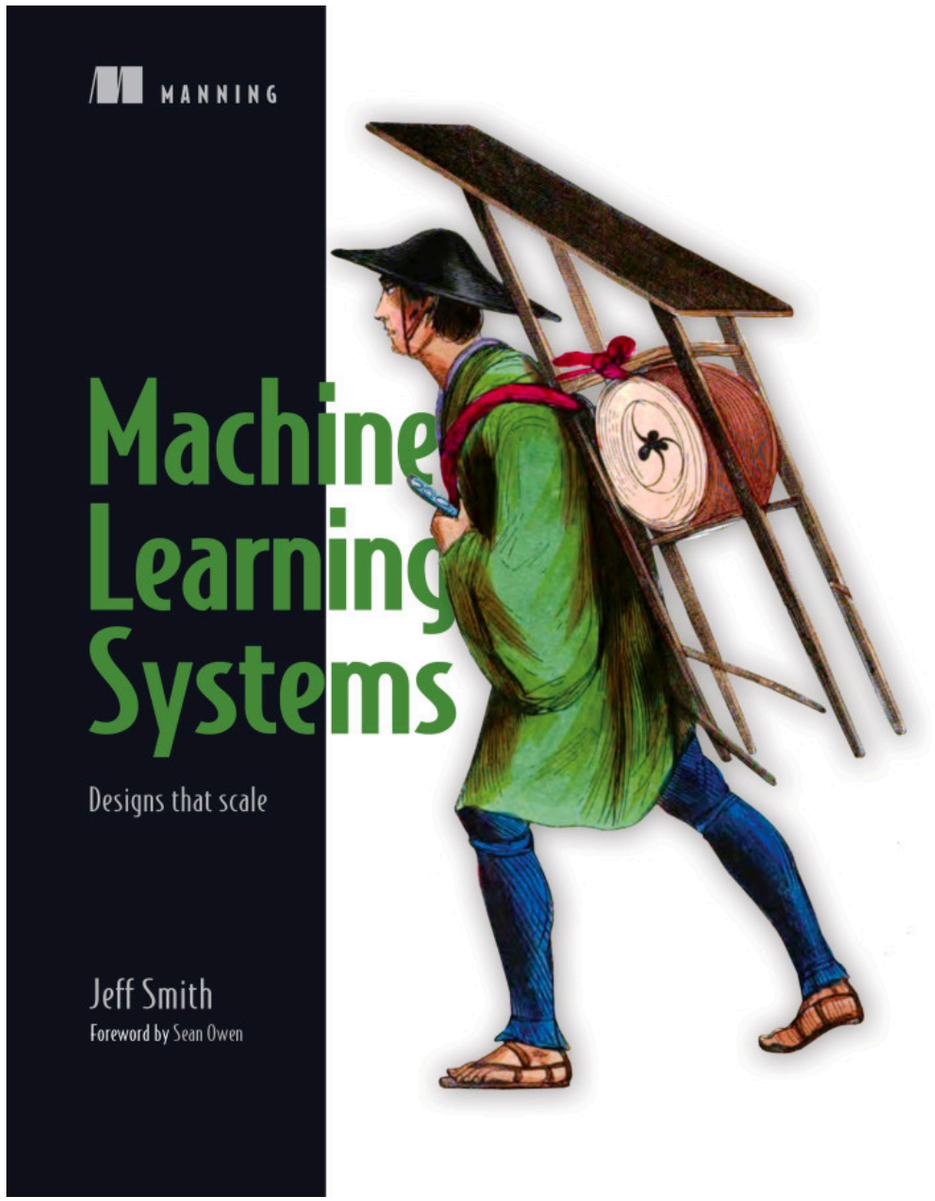
Learning goals

- Understand how to build a system that can put the power of machine learning to use.

- Understand how to incorporate ML-based components into a larger system.

- Understand the principles that govern these systems, both as software and as predictive systems.

Main Sources



UBER Engineering

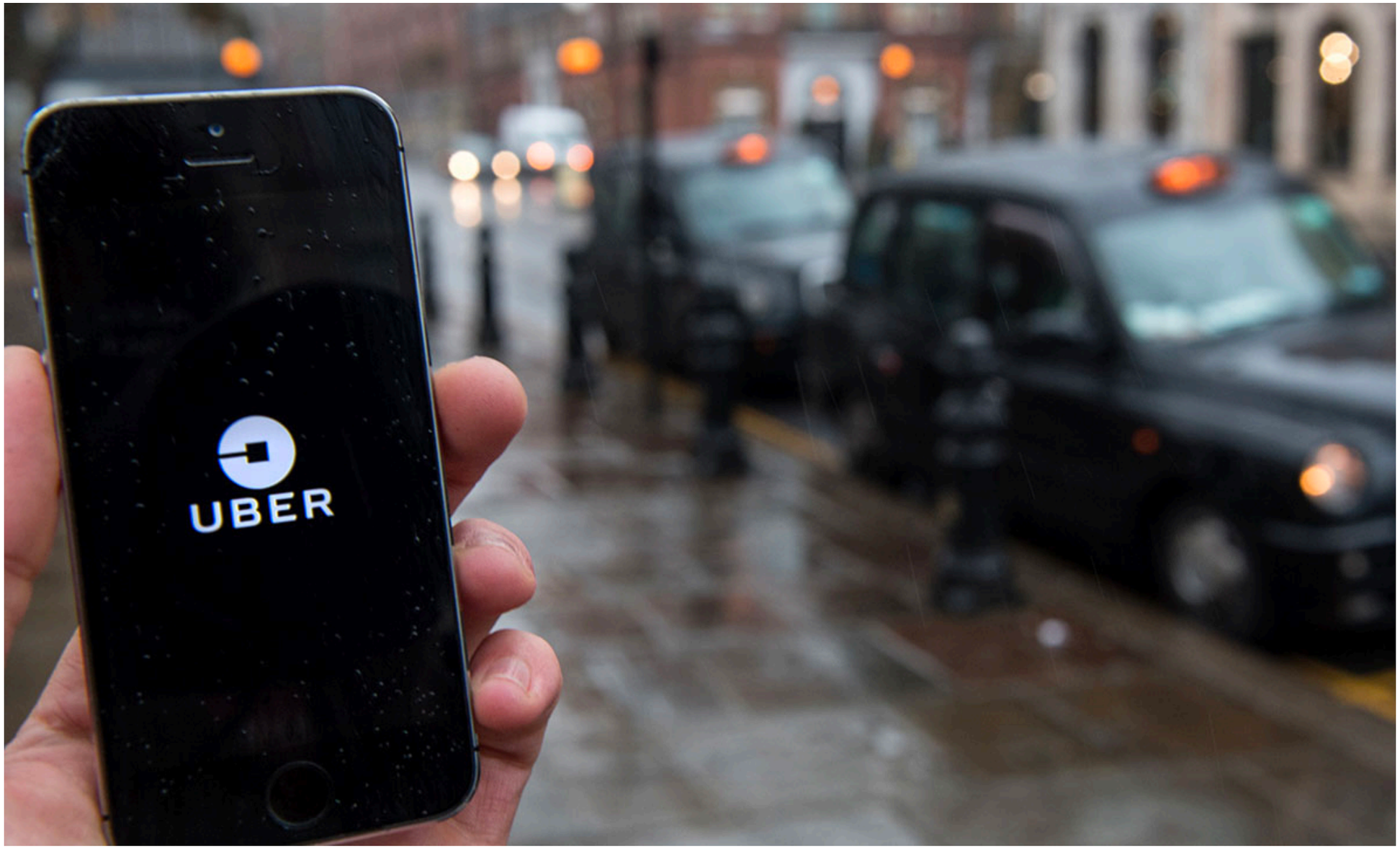
THE NETFLIX TECH BLOG

Spotify® Labs

Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow

By Alex Sergeev and Mike Del Balso

October 17, 2017

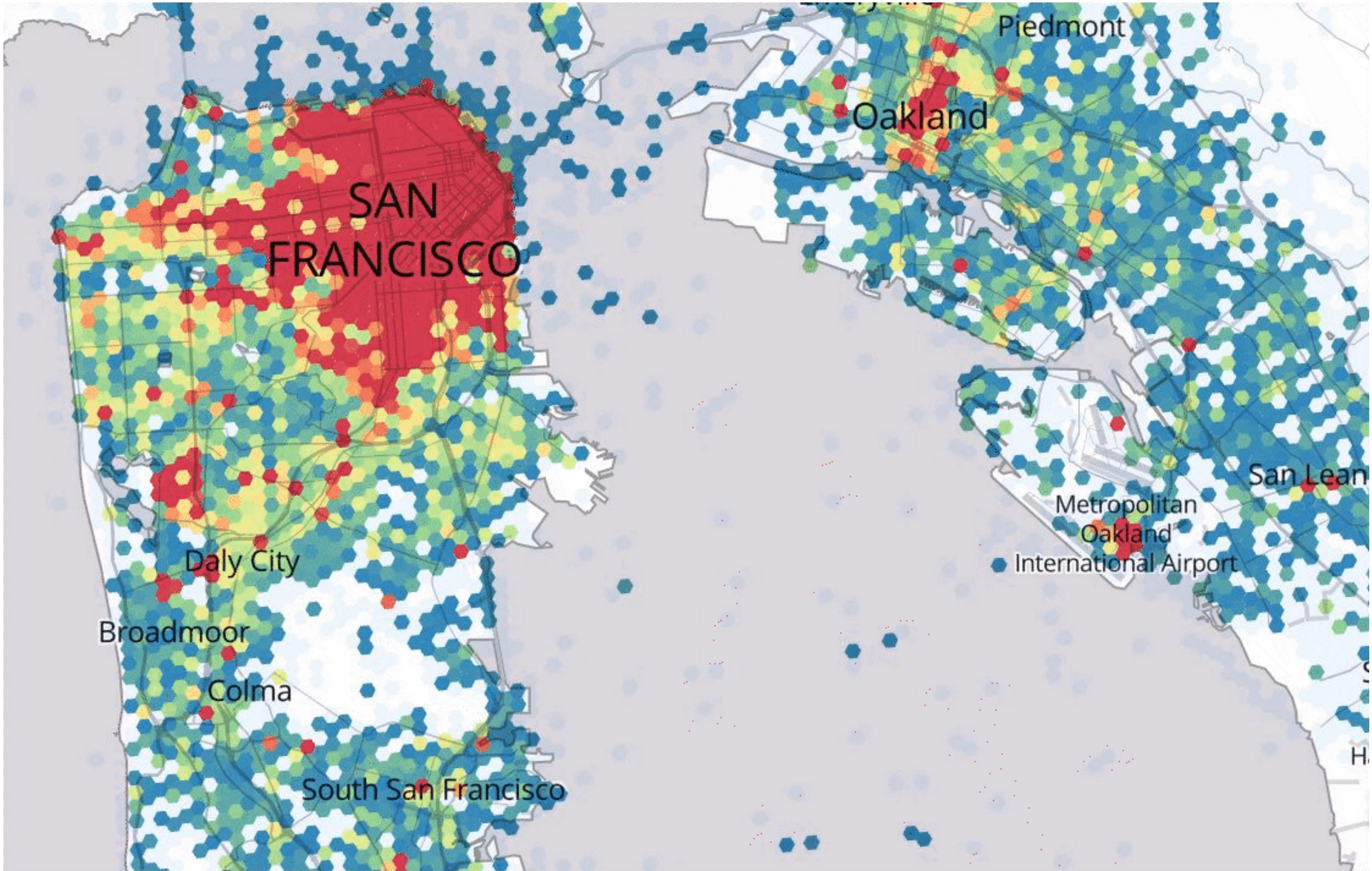


Any guess how Uber uses deep learning?

Deep learning across Uber

- Self-driving research
- Trip forecasting
- Fraud prevention

Marketplace forecasting



Let’s begin the story

Uber also uses TensorFlow

Do you know why?

Why Uber adopts TensorFlow?

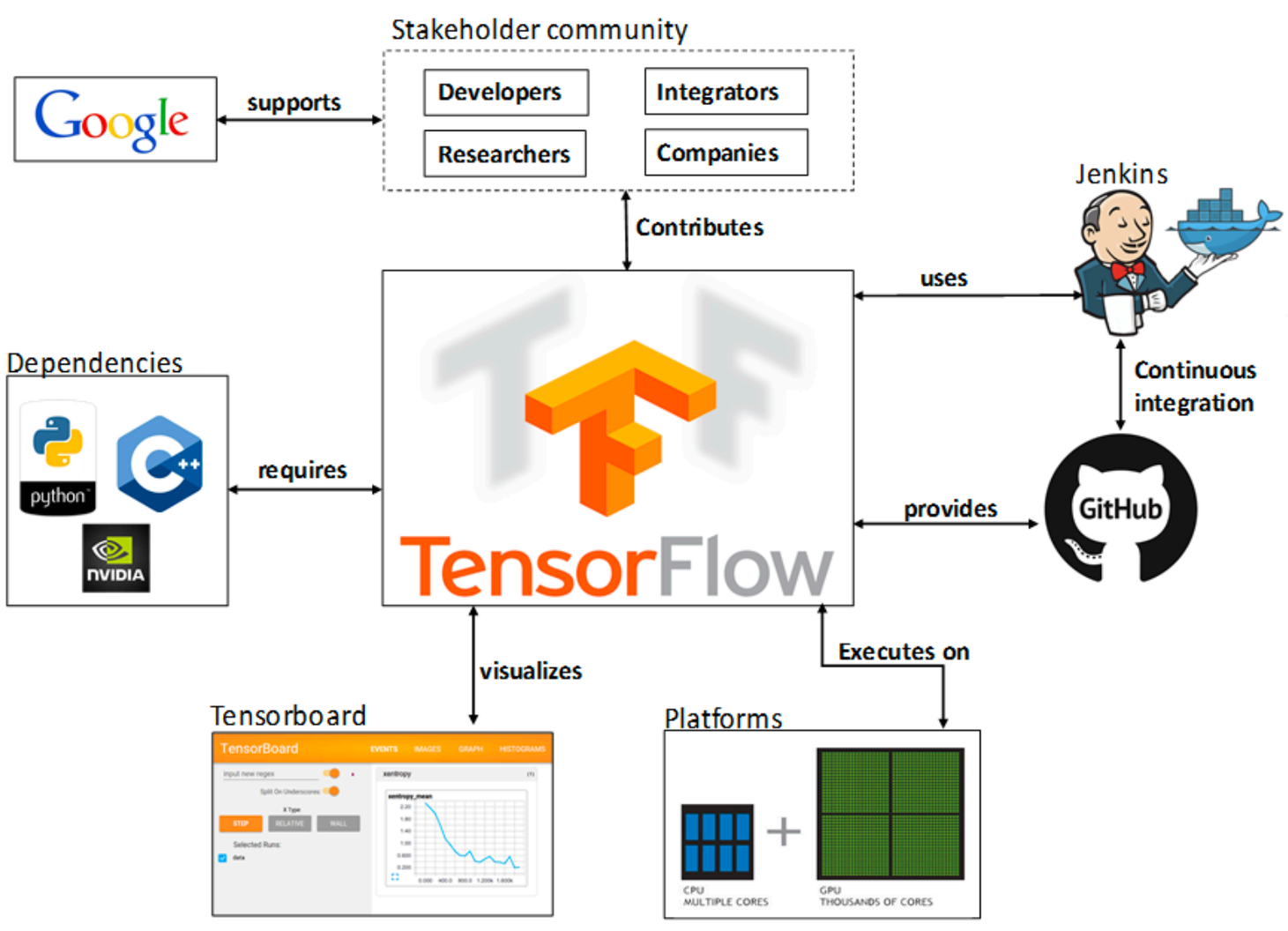
- TF is one of the most widely used open source frameworks for deep learning, which makes it easy to onboard new users.

Companies using TensorFlow



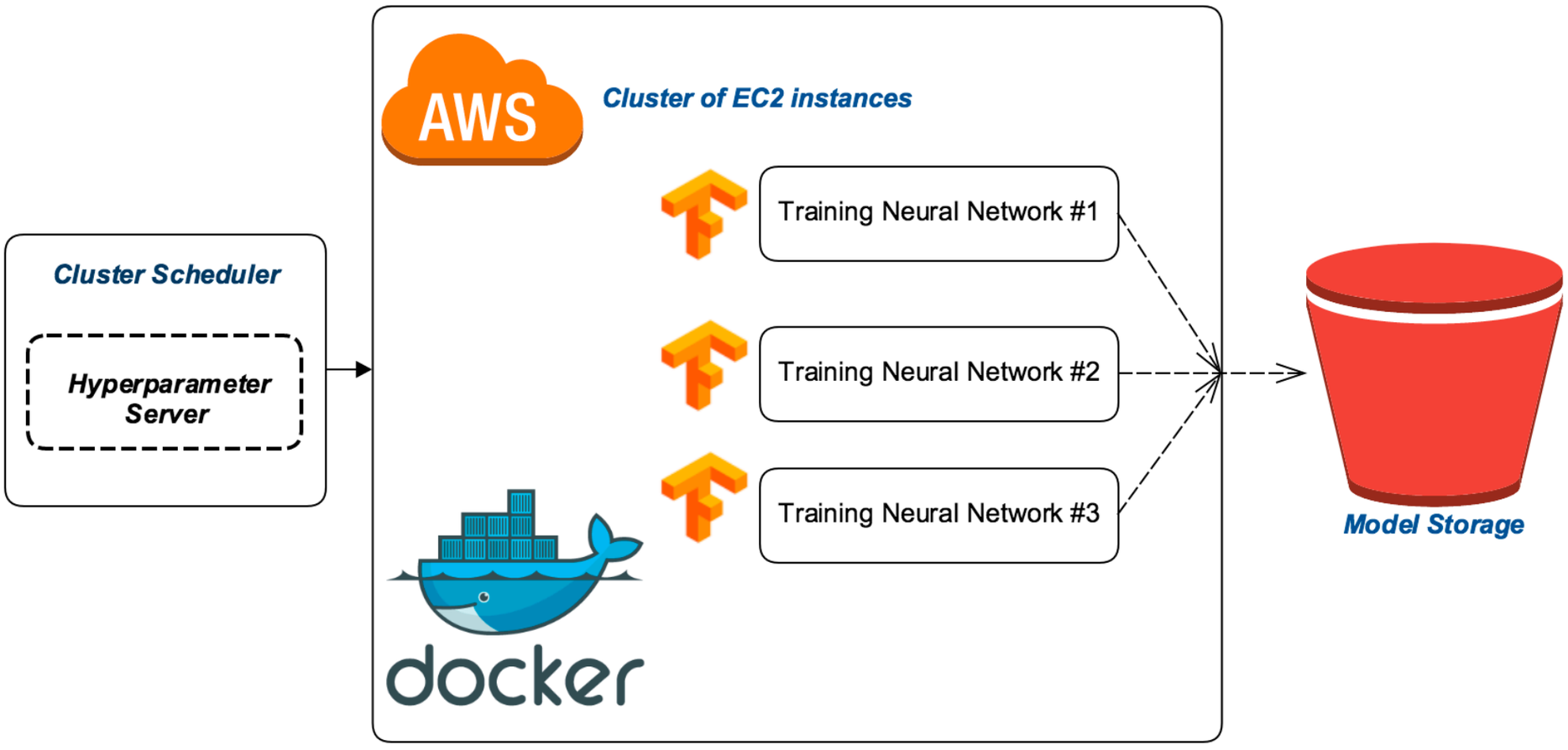
Why Uber adopts TensorFlow?

- TF combines high performance with an ability to tinker with low-level model details—for instance, we can use both high-level APIs, such as Keras, and implement our own custom operators using NVIDIA's CUDA toolkit.

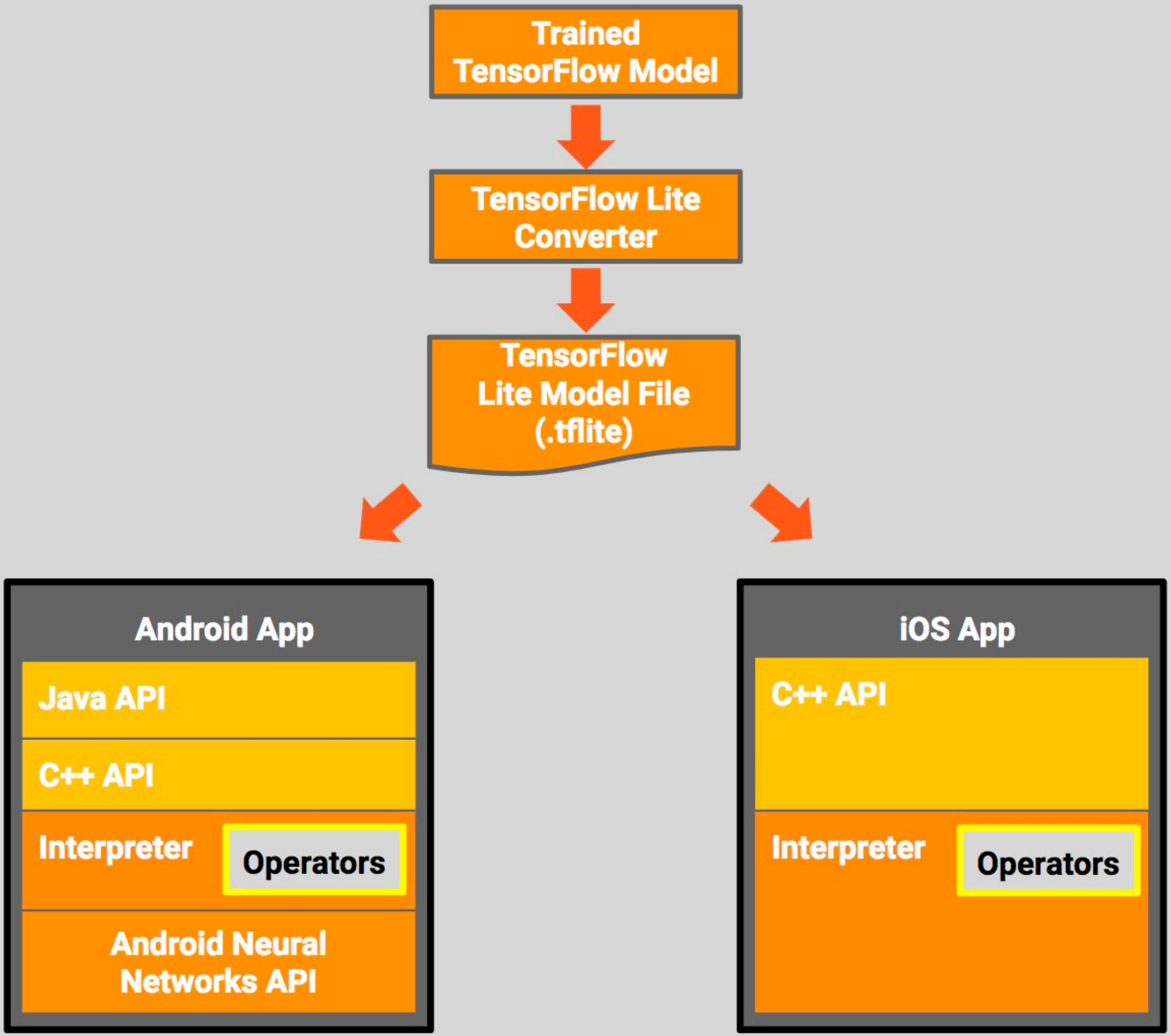


Why Uber adopts TensorFlow?

- Additionally, TF has end-to-end support for a wide variety of deep learning use cases, from conducting exploratory research to deploying models in production on cloud servers, mobile apps, and even self-driving vehicles.



Architecture



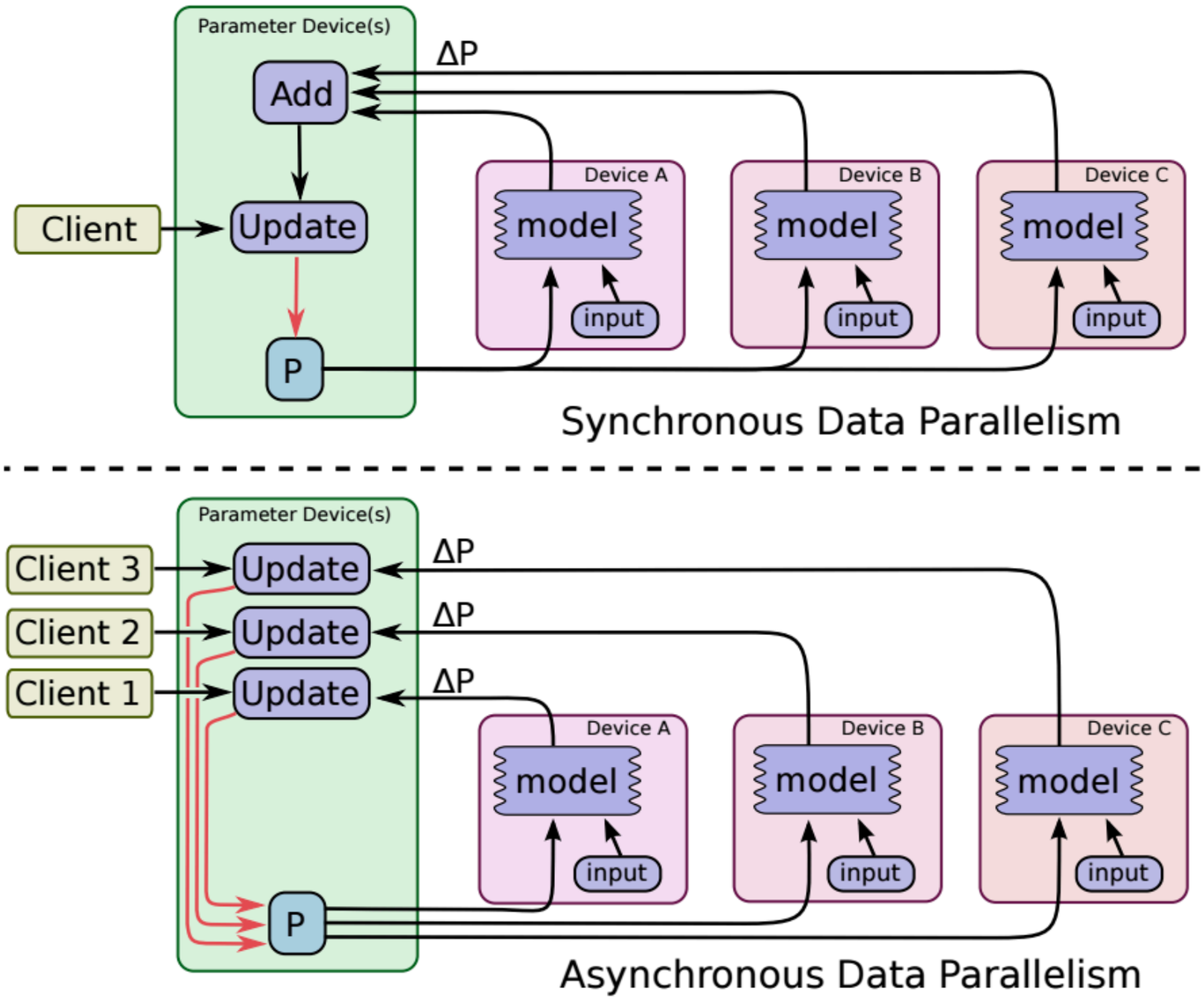
Training time increased a lot!

- Training more and more machine learning models at Uber,
- Their size and data consumption grew significantly.

- In a large portion of cases, the models were still small enough to fit on one or multiple GPUs within a server, but as datasets grew, so did the training times, which sometimes took a week—or longer!—to complete.

Going distributed

Distributed TF



Mapping job names to lists of network addresses

tf.train.ClusterSpec construction	Available tasks
`tf.train.ClusterSpec({"local": ["localhost:2222", "localhost:2223"]})`	**/job:local/task:0** **/job:local/task:1**
`tf.train.ClusterSpec({` `  "worker": [` `  "worker0.example.com:2222",` `  "worker1.example.com:2222",` `  "worker2.example.com:2222"` `  ],` `  "ps": [` `  "ps0.example.com:2222",` `  "ps1.example.com:2222"` `  ]})`	**/job:worker/task:0** **/job:worker/task:1** **/job:worker/task:2** **/job:ps/task:0** **/job:ps/task:1**

Specifying distributed devices in your model

```
with tf.device("/job:ps/task:0"):
  weights_1 = tf.Variable(...)
  biases_1 = tf.Variable(...)

with tf.device("/job:ps/task:1"):
  weights_2 = tf.Variable(...)
  biases_2 = tf.Variable(...)

with tf.device("/job:worker/task:7"):
  input, labels = ...
  layer_1 = tf.nn.relu(tf.matmul(input, weights_1) + biases_1)
  logits = tf.nn.relu(tf.matmul(layer_1, weights_2) + biases_2)
  # ...
  train_op = ...

with tf.Session("grpc://worker7.example.com:2222") as sess:
  for _ in range(10000):
    sess.run(train_op)
```

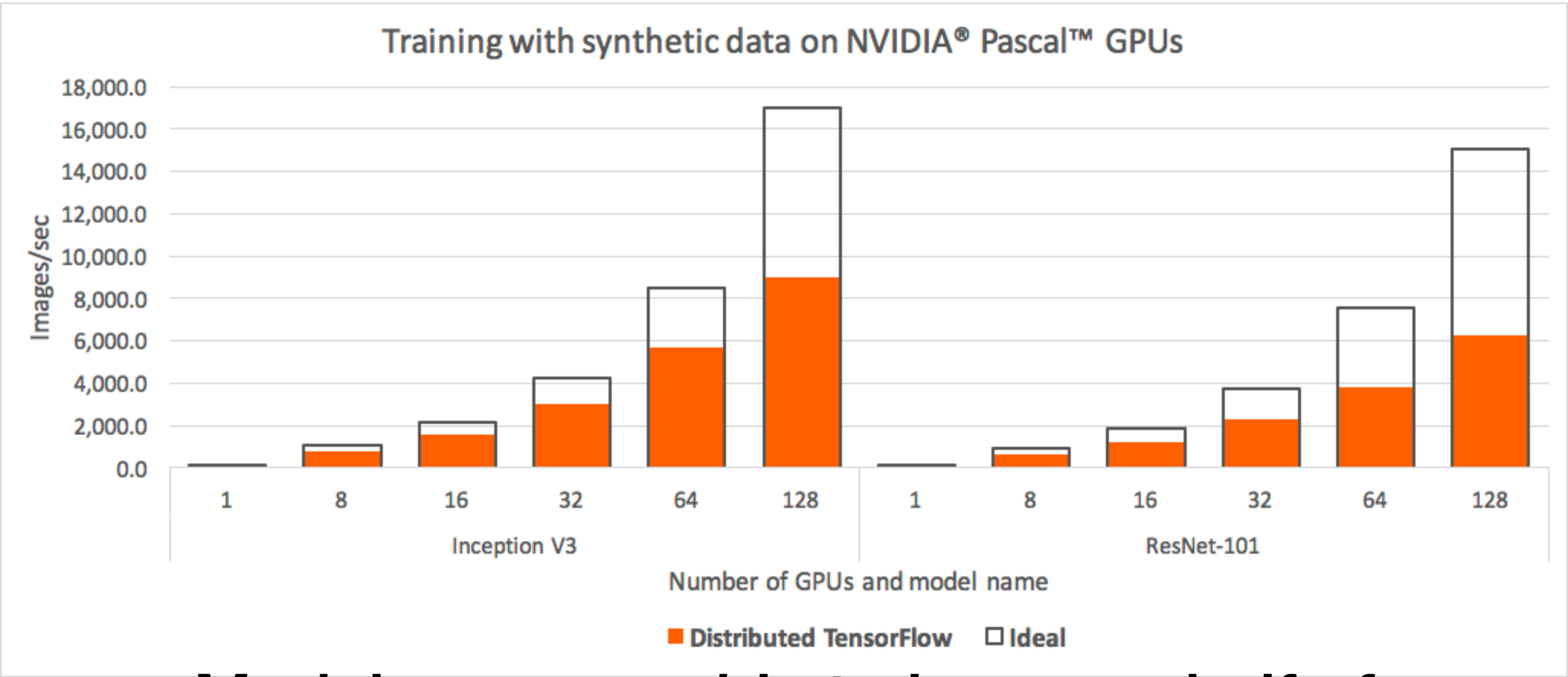
Uber first experience with Distributed TF

- It was not always clear which code modifications needed to be made to distribute their model training code.

- The standard distributed TensorFlow package introduces **many new concepts**: workers, parameter servers, `tf.Server()`

- The challenge of computing at **Uber's scale**. After running a few benchmarks, we found that we could not get the standard distributed TensorFlow to scale as well as our services required.

Distributed TF became inefficient at Uber scale



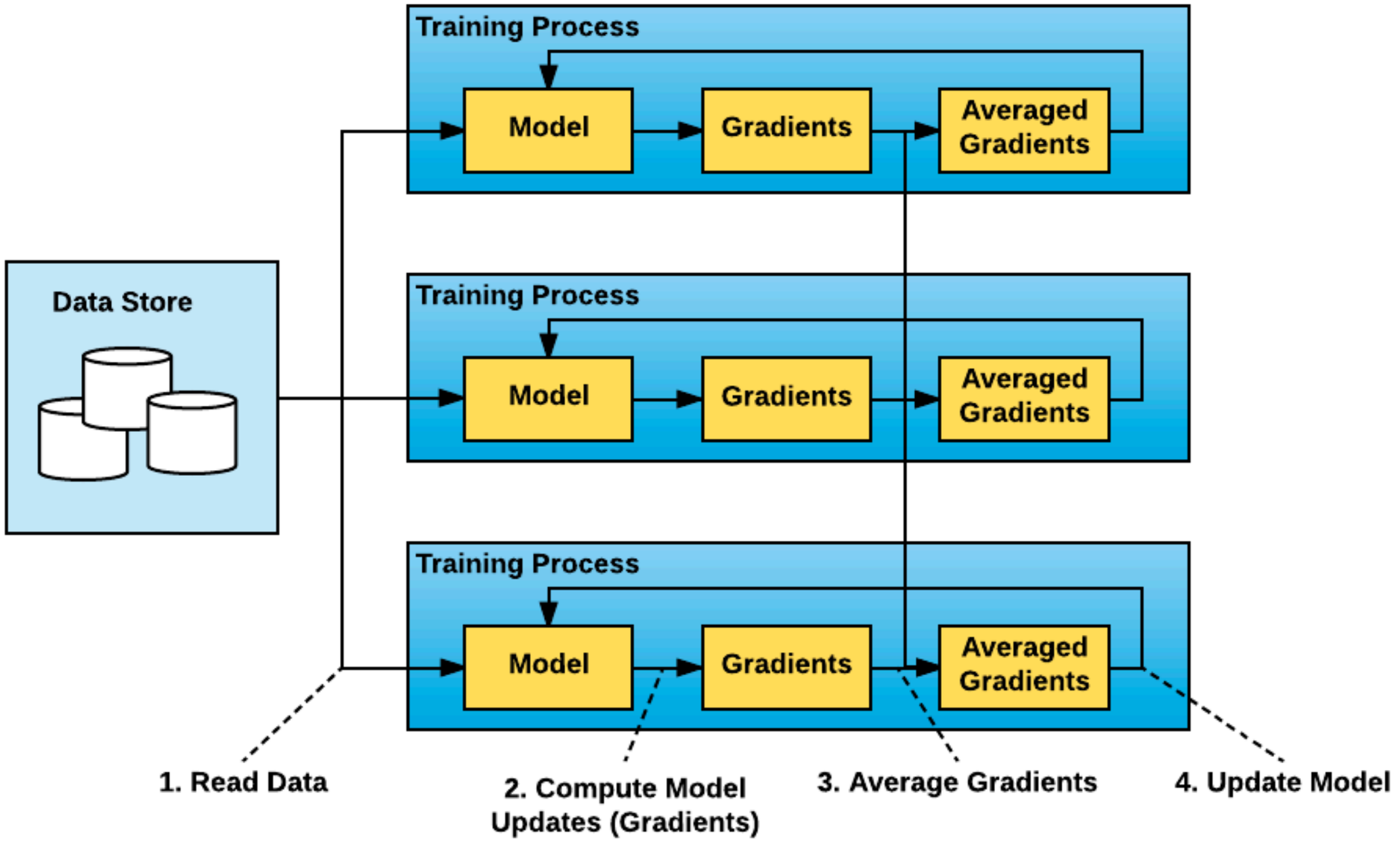
Models were unable to leverage half of the resource

They become even more motivated after observing Google training ResNet-50 in an hour!

- “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour,” demonstrating their training of a ResNet-50 network in one hour on 256 GPUs by combining principles of data parallelism with an **innovative learning rate adjustment technique**.

Leveraging a different type of algorithm

Data parallelism (Facebook)



But wait!

- What other approaches exist for distributing a (deep Learning) algorithm?
- And why Uber could possibly do Data Parallel approach? Any insight?

And here is why Uber started with Data Parallel

- Uber’s models were small enough to fit on a single GPU, or multiple GPUs in a single server

How data parallel works?

1. Run multiple copies of the training script and each copy:
    A. reads a chunk of the data
    B. runs it through the model
    C. computes model updates (gradients)
2. Average gradients among those multiple copies
3. Update the model Repeat (from Step 1a)

Data parallel vs Model parallel

- **Data Parallel** (“Between-Graph Replication”)
    - Send exact same model to each device
    - Each device operates on its partition of data § ie. Spark sends same function to many workers
    - Each worker operates on their partition of data

- **Model Parallel** (“In-Graph Replication”)
    - Send different partition of model to each device
    - Each device operates on all data

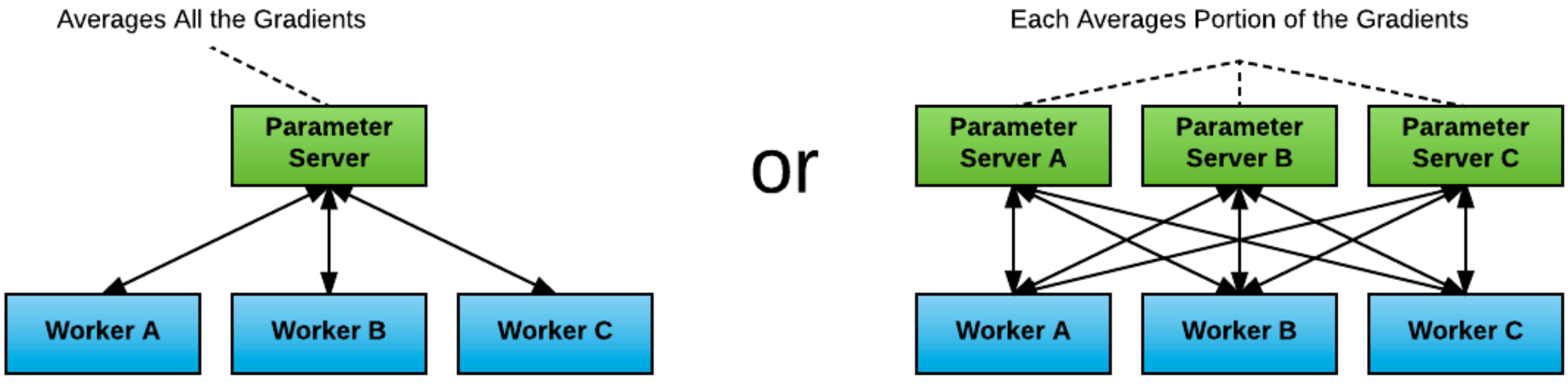
While this approach improved performance, they encountered two challenges

It was good, but they hit some challenges

- Identifying the right ratio of worker to parameter servers.
  - 1 parameter server
  - Multiple parameter server
- Handling increased TensorFlow program complexity
  - Every user of distributed TensorFlow had to explicitly start each worker and PS, pass around service discovery information.
  - Users had to ensure that all the operations were placed appropriately and code is modified to leverage multiple GPUs.

It was good, but they hit some challenges

- Identifying the right ratio of worker to parameter servers.
  - 1 parameter server
  - Multiple parameter server

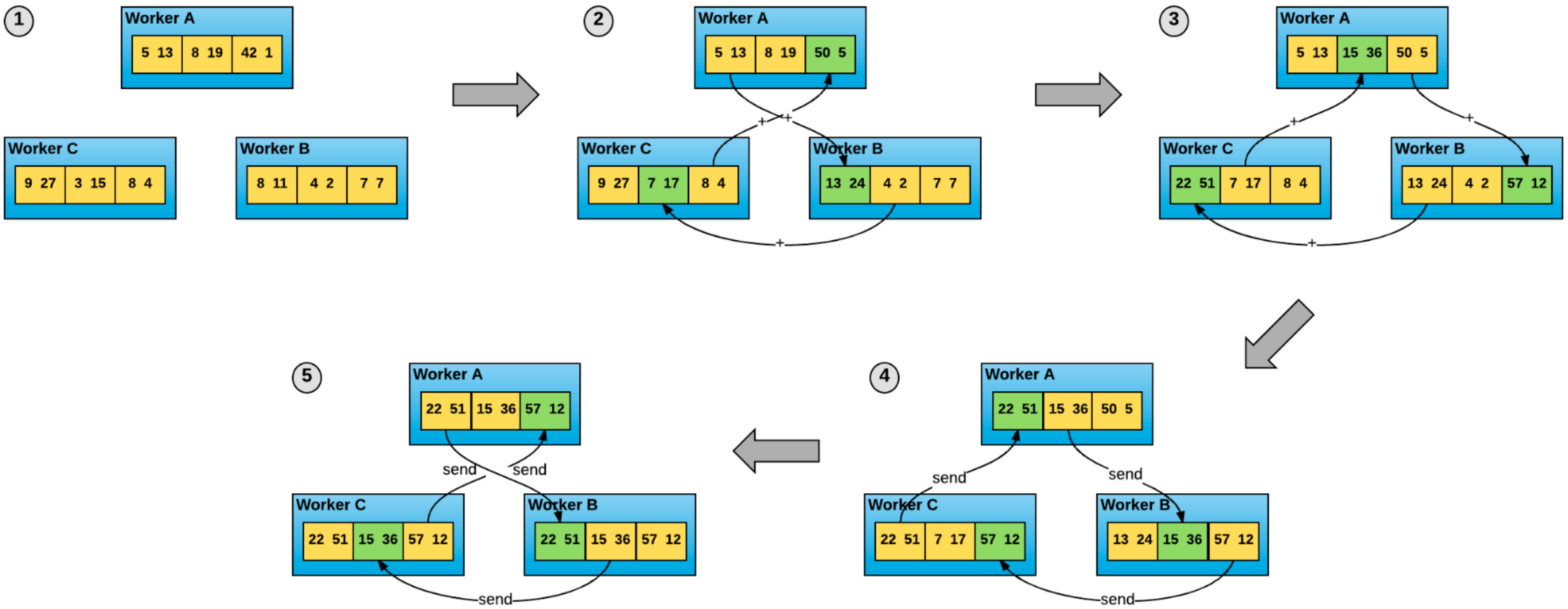


TF Complexity

- Handling increased TensorFlow program complexity
  - Every user of distributed TensorFlow had to explicitly start each worker and PS, pass around service discovery information.
  - Users had to ensure that all the operations were placed appropriately and code is modified to leverage multiple GPUs.

```
# On ps0.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=ps --task_index=0
# On ps1.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=ps --task_index=1
# On worker0.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=worker --task_index=0
# On worker1.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=worker --task_index=1
```

Baidu approach to avoid parameter server



Baidu all reduce is not only network optimal, but easier to adopt

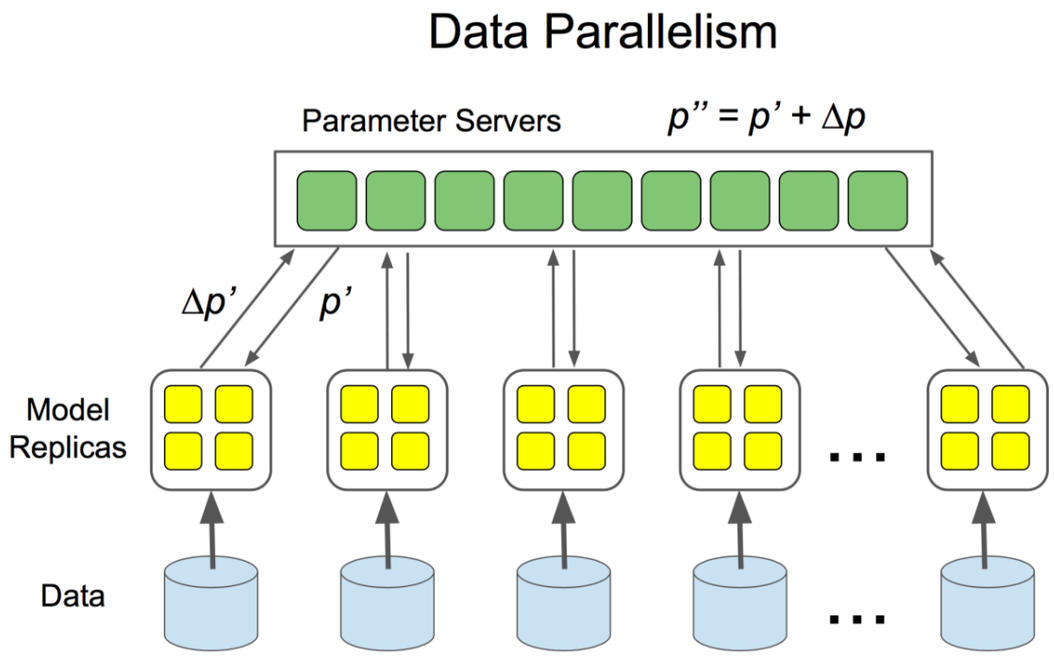
- Users utilize a Message Passing Interface (MPI) implementation such as OpenMPI to launch all copies of the TensorFlow program.

- MPI then transparently sets up the distributed infrastructure necessary for workers to communicate with each other.

- All the user needs to do is modify their program to average gradients using an allreduce() operation.

Horovod = Distributed deep learning with TensorFlow

Any similarity?



Hovord was then built upon Baidu allreduce approach

- A stand-alone Python package called Horovod.

- Distributed TensorFlow processes use Horovod to communicate with each other.

- At any point in time**, various teams at Uber may be using different releases of TensorFlow**. We wanted all teams to be able to leverage the ring-allreduce algorithm without needing to upgrade to the latest version of TensorFlow, apply patches to their versions, or even spend time building out the framework.

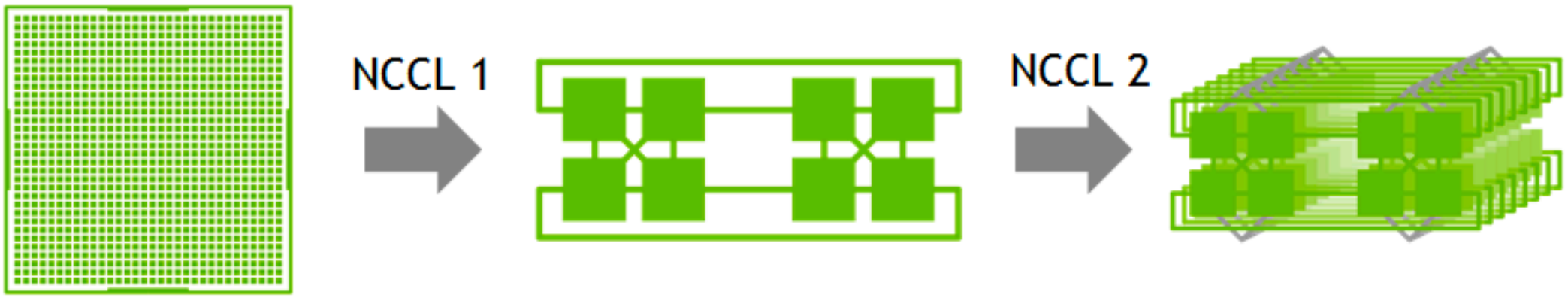
- Having a stand-alone package allowed Uber to cut the time required to **install Horovod from about an hour to a few minutes**, depending on the hardware.

From single GPU to Multi-GPU Multi-Node

- Replaced Baidu ring-allreduce with NCCL.

- NCCL is NVIDIA's library for **collective communication** that provides a highly optimized version of ring-allreduce.

- NCCL 2 introduced the ability to run ring-allreduce across **multiple machines**.



The update were included API improvements

- Several API improvements inspired by **feedback Uber** received from a number of initial users.

- A **broadcast operation** that enforces consistent initialization of the model on all workers.

- The new API allowed Uber to cut down the number of operations a user had to introduce to their single GPU program to four.

Distributing training job with Horovod

```
import tensorflow as tf
import horovod.tensorflow as hvd

# Initialize Horovod
hvd.init()

# Pin GPU to be used to process local rank (one GPU per process)
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())

# Build model...
loss = ...
opt = tf.train.AdagradOptimizer(0.01)

# Add Horovod Distributed Optimizer
opt = hvd.DistributedOptimizer(opt)
```

Distributing training job with Horovod

```
# Add hook to broadcast variables from rank 0 to all other processes during
# initialization.
hooks = [hvd.BroadcastGlobalVariablesHook(0)]

# Make training operation
train_op = opt.minimize(loss)

# The MonitoredTrainingSession takes care of session initialization,
# restoring from a checkpoint, saving to a checkpoint, and closing when done
# or an error occurs.
with
tf.train.MonitoredTrainingSession(checkpoint_dir="/tmp/train_logs",
                                  config=config,
                                  hooks=hooks) as mon_sess:
 while not mon_sess.should_stop():
   # Perform synchronous training.
    mon_sess.run(train_op)
```

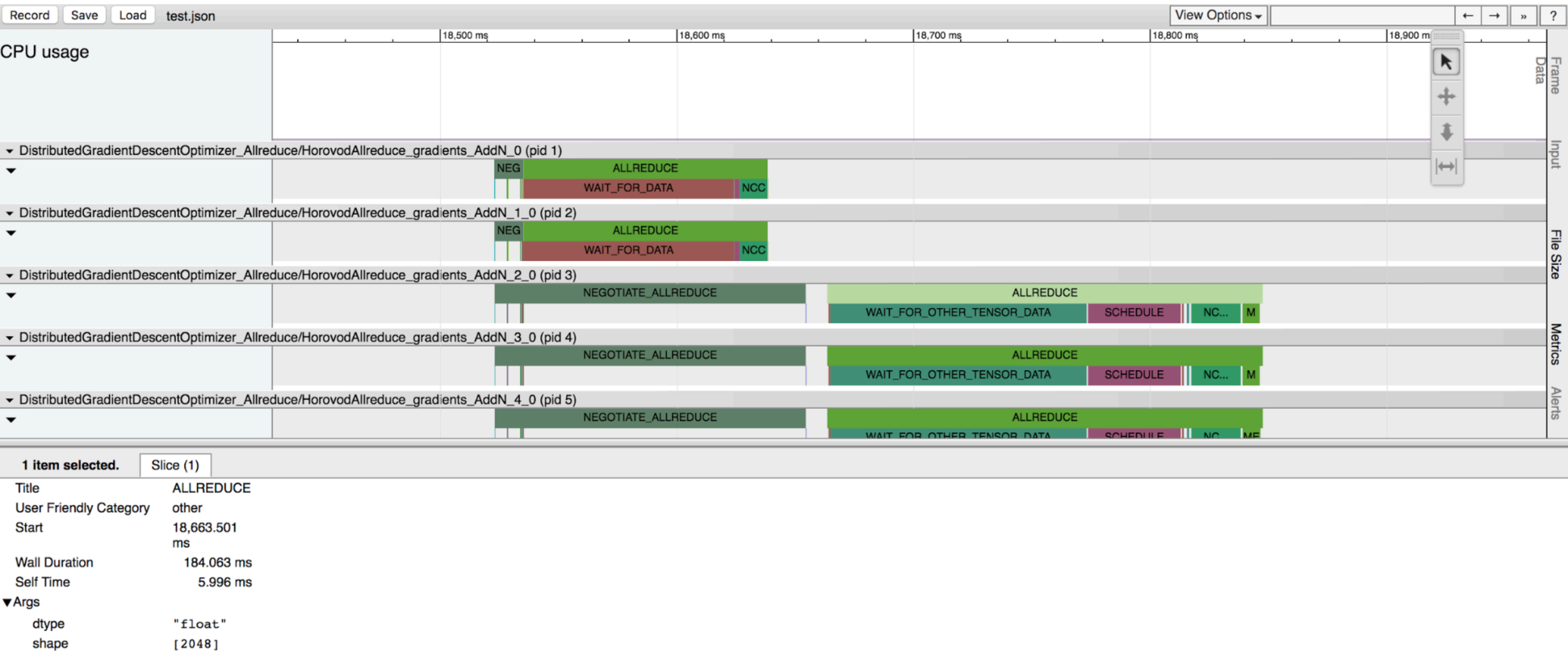
User can then run several copies of the program across multiple servers

```
$ mpirun -np 16 -x LD_LIBRARY_PATH -H
server1:4,server2:4,server3:4,server4:4 python train.py
```

Horovord also distribute Keras programs

- Horovod can also distribute Keras programs by following the same steps.

Now time come to debugging a distributed systems



Yet another challenge: Tiny allreduce

- After we analyzed the timelines of a few models, we noticed that those with a large amount of tensors, such as ResNet-101, tended to have many tiny allreduce operations.

- ring-allreduce utilizes the network in an optimal way if the tensors are large enough, but does not work as efficiently or quickly if they are very small.

Tensor Fusion

- What if multiple tiny tensors could be fused together before performing ring-allreduce on them?

Tensor Fusion

1. Determine which tensors are ready to be reduced. Select the first few tensors that fit in the buffer and have the same data type.

2. Allocate a fusion buffer if it was not previously allocated. Default fusion buffer size is 64 MB.

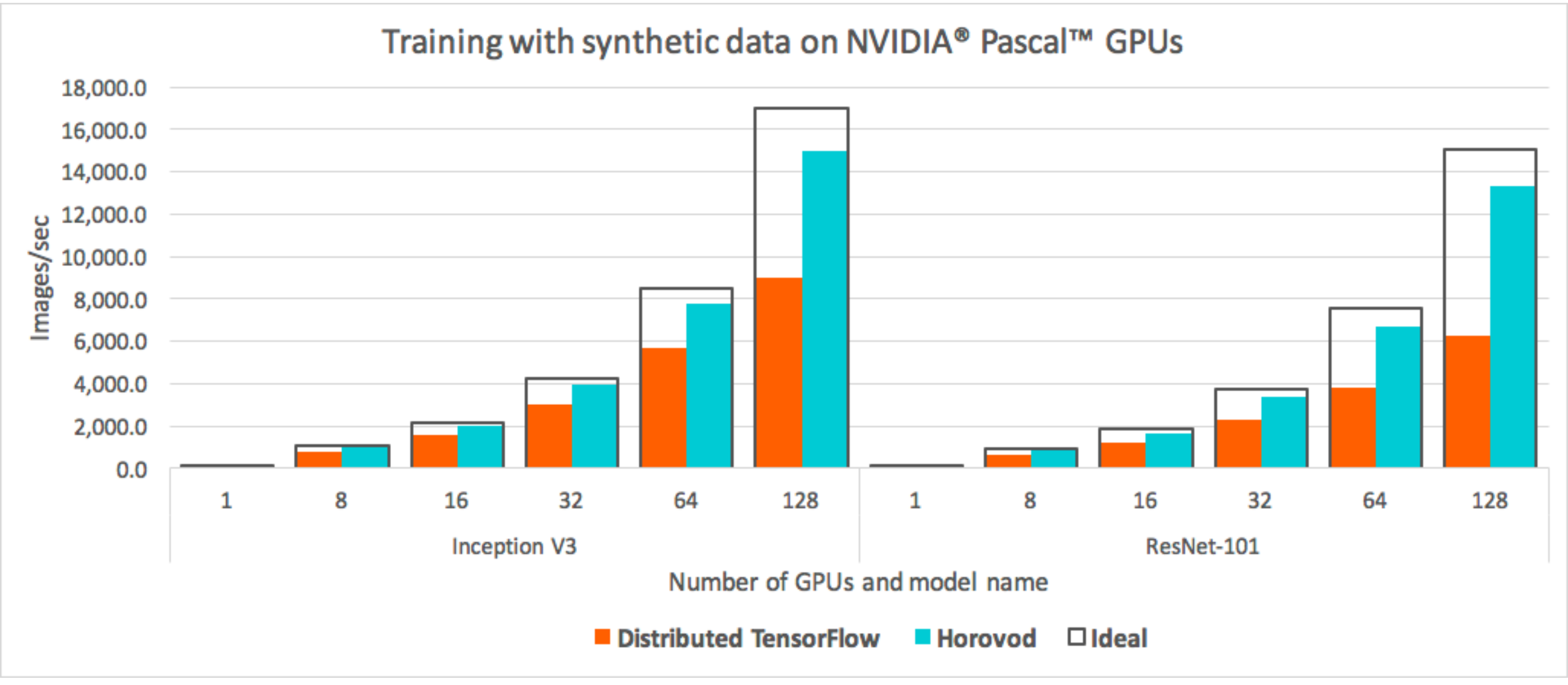
3. Copy data of selected tensors into the fusion buffer.

4. Execute the allreduce operation on the fusion buffer.

5. Copy data from the fusion buffer into the output tensors.

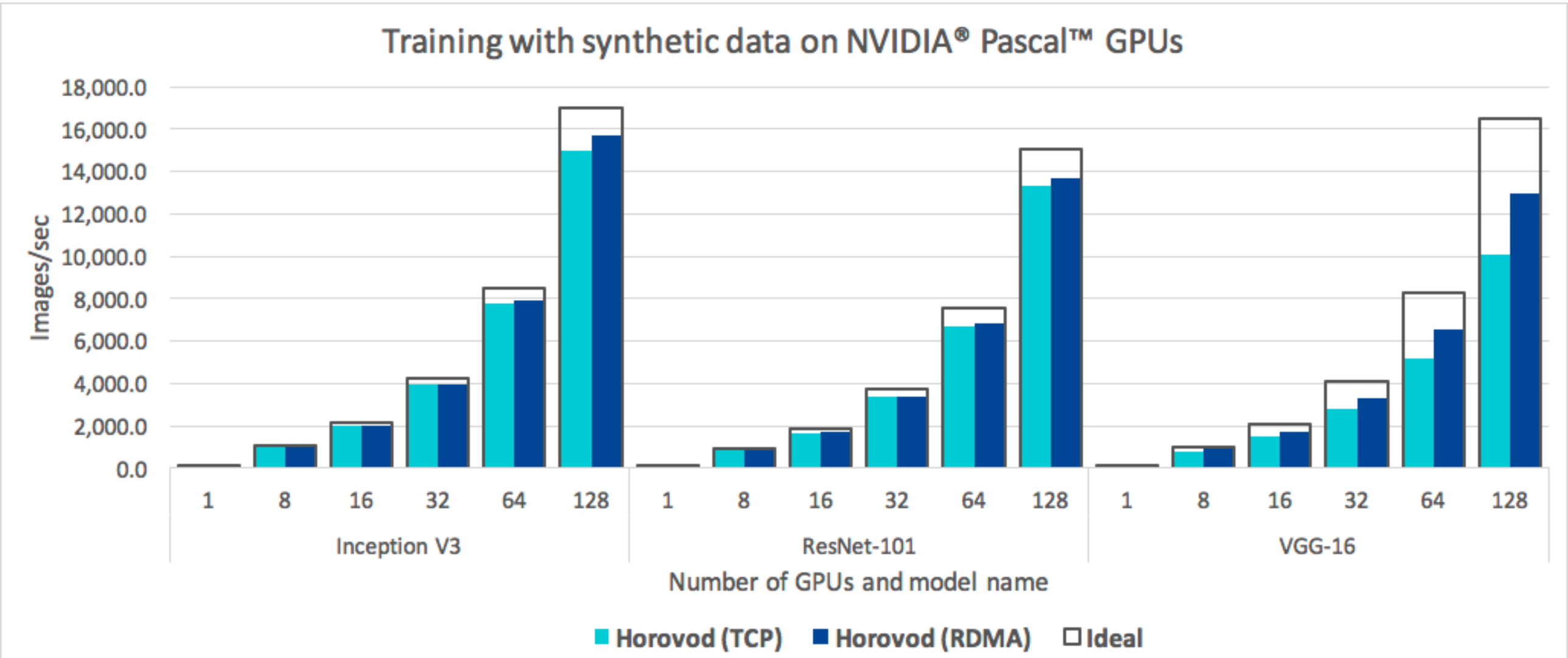
6. Repeat until there are no more tensors to reduce in the cycle.

Horovod vs TF



the training was about twice as fast as standard distributed TensorFlow.

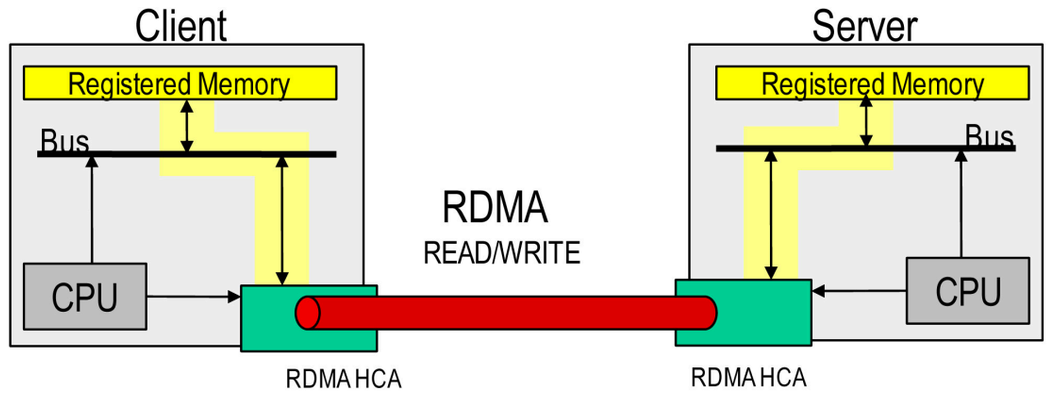
Benchmarking with RDMA network cards



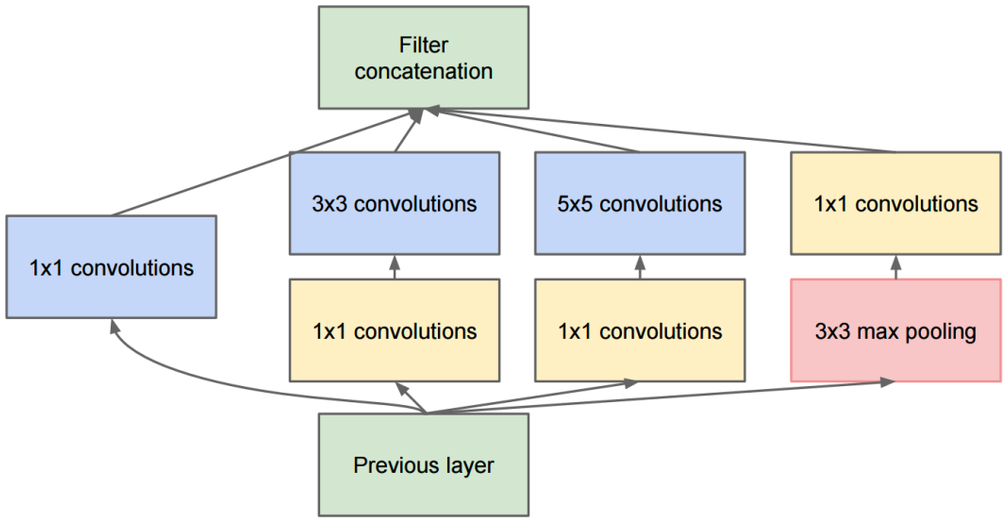
VGG-16 model experienced a significant 30 percent speedup when we leveraged RDMA networking.

Do you know why that happened?

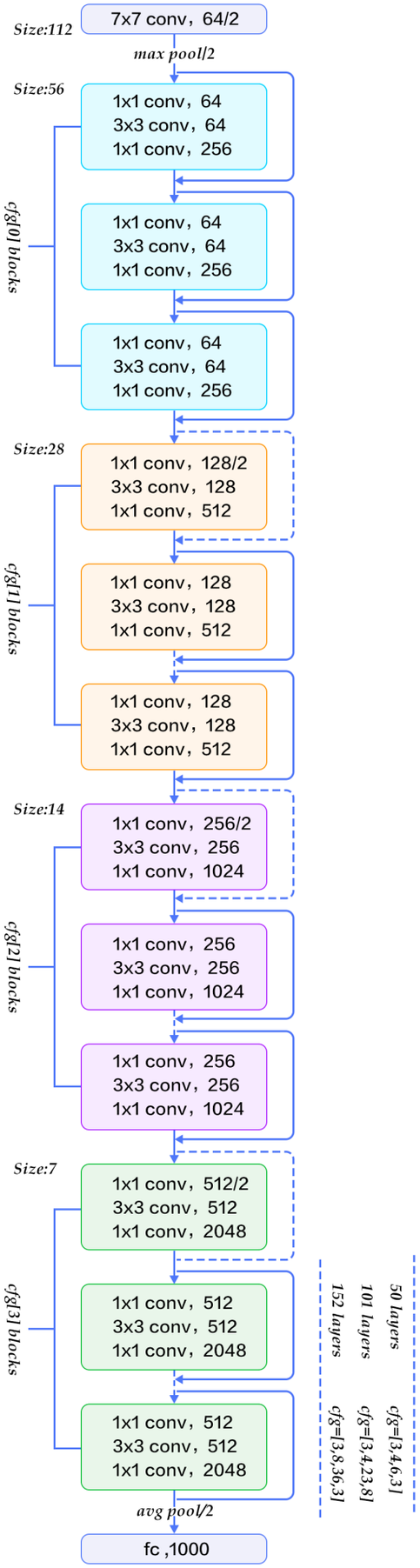
Any insight?



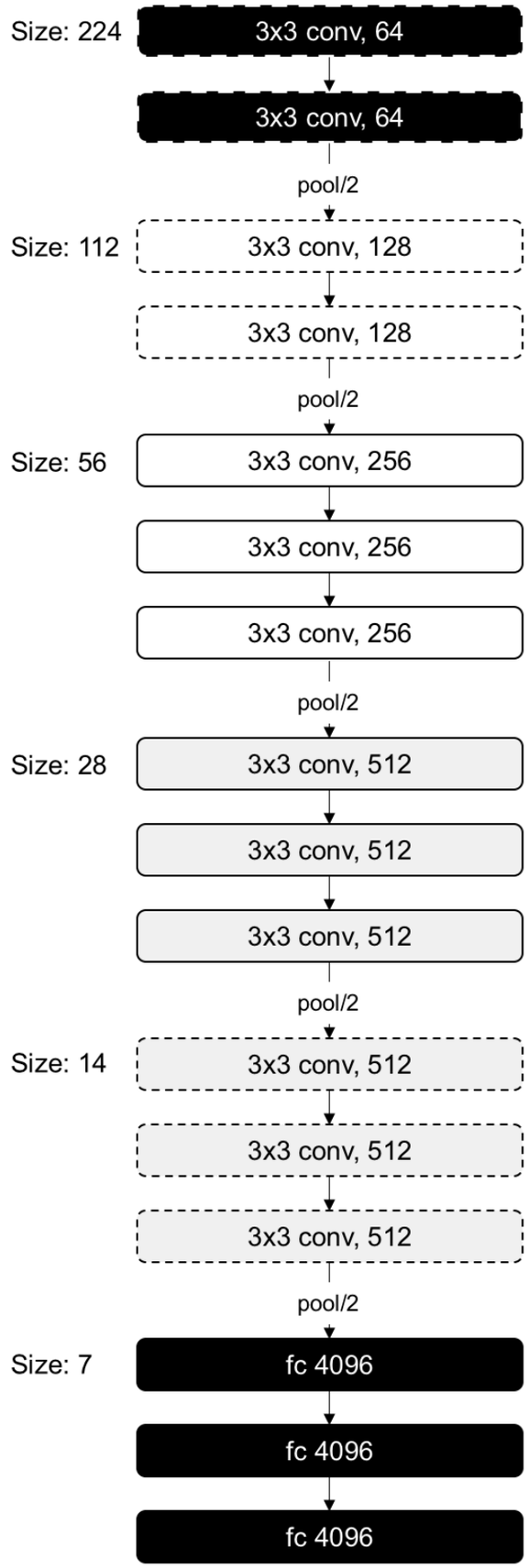
Any thought?



Parameters: 25 million



Parameters: 25 million



Parameters: 138 million

Summary

- We reviewed when ML needs to go distributed.
- We studies some alternative solutions and why Uber decided to built up their own solution
- We studied extensions that was made by Uber to accommodate their own requirements.
- We reviewed how Horovod helped Uber to scale up their training process.