Introduction to Deep Learning Systems

Zhihao Jia

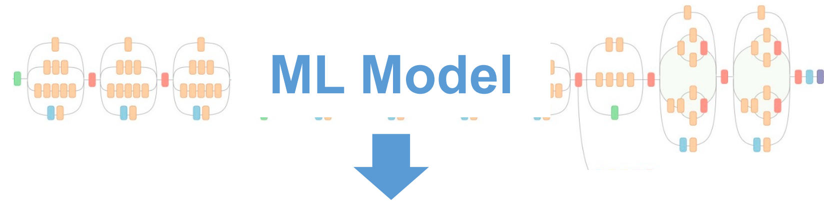
Computer Science Department
Carnegie Mellon University

Administrative

- Paper presentation assignments available on the website
  - Discuss with your partner on how you would like to deliver the presentation

- First reading assignments **due next Monday before lecture**

Recap: Deep Learning Systems



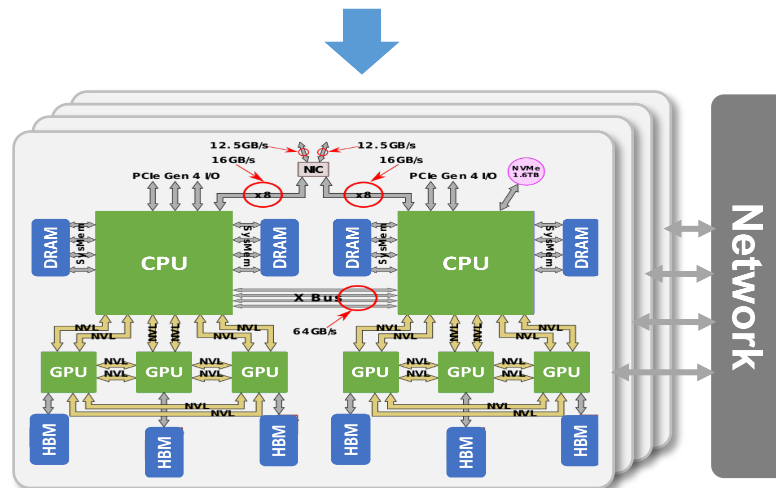
Automatic Differentiation

Graph-Level Optimization

Parallelization / Distributed Training

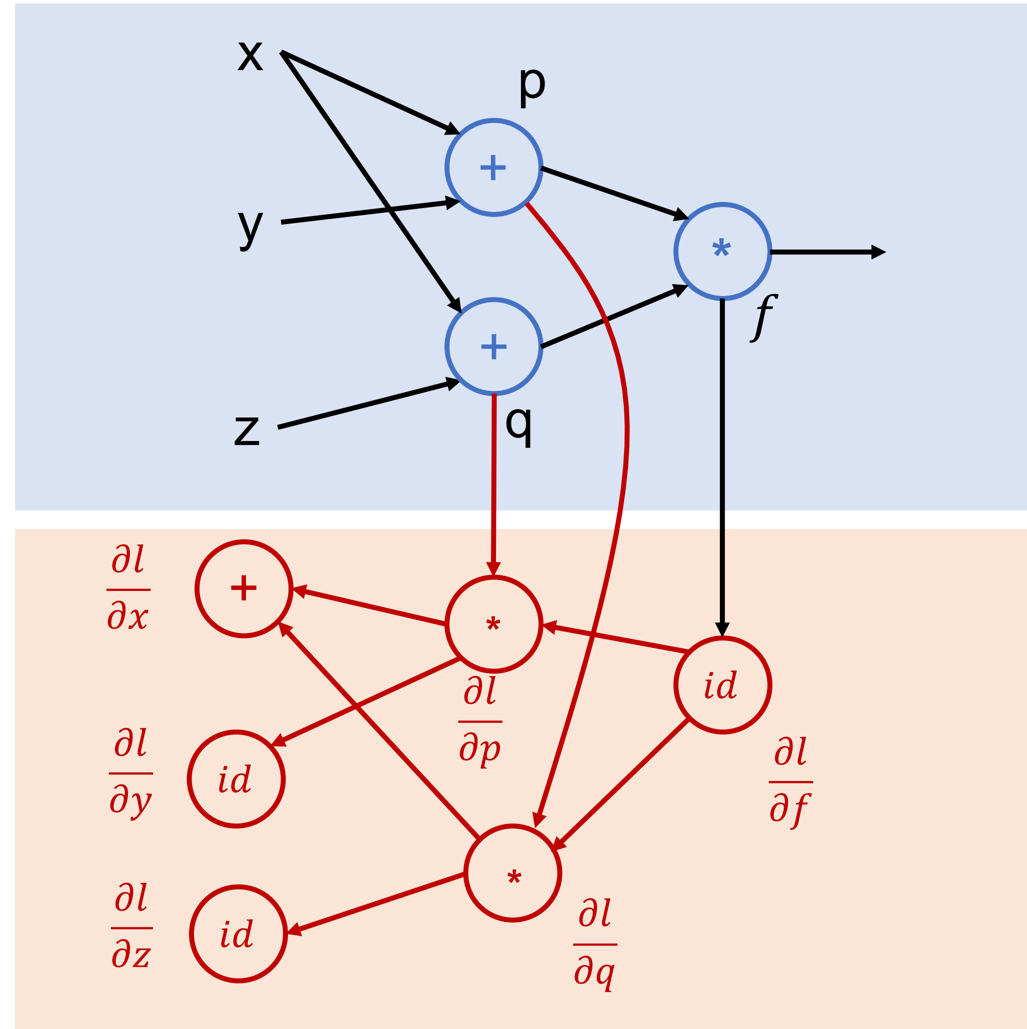
Code Optimization

Memory Optimization



Recap: Automatic Differentiation

Automatically construct backward computation graph

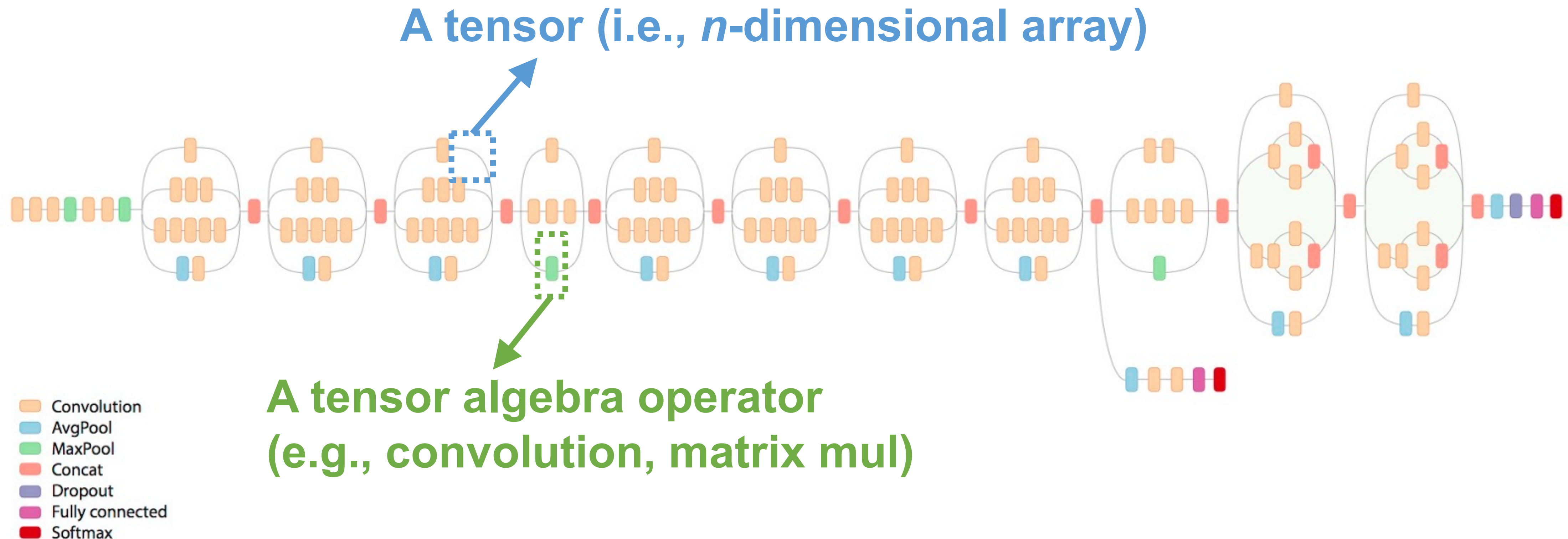


Forward computation graph

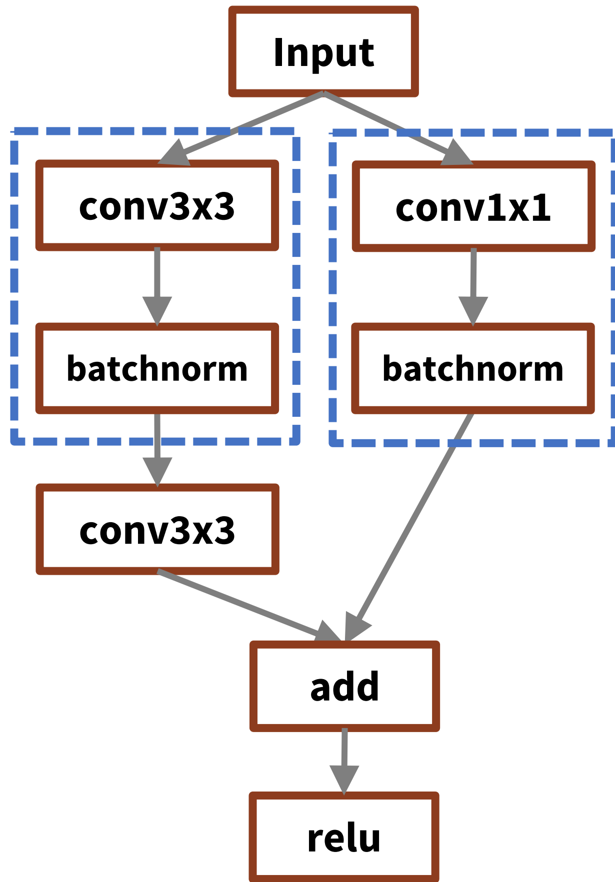
Backward computation graph

Recap: Deep Neural Network

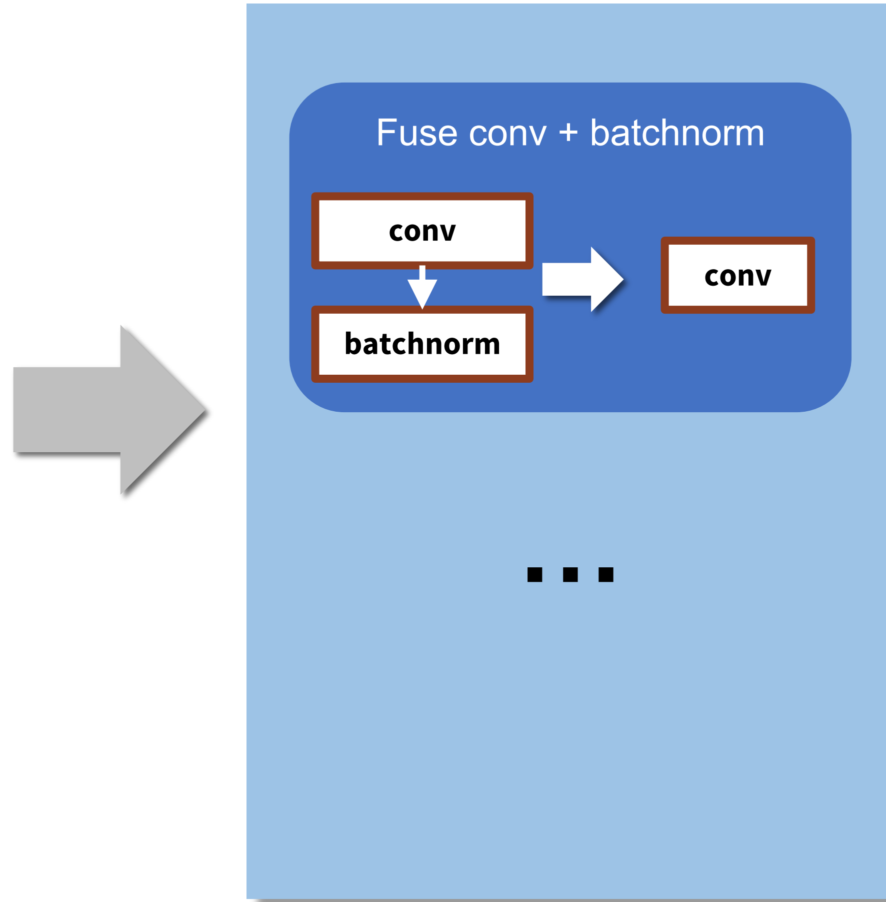
- Collection of simple trainable mathematical units that work together to solve complicated tasks



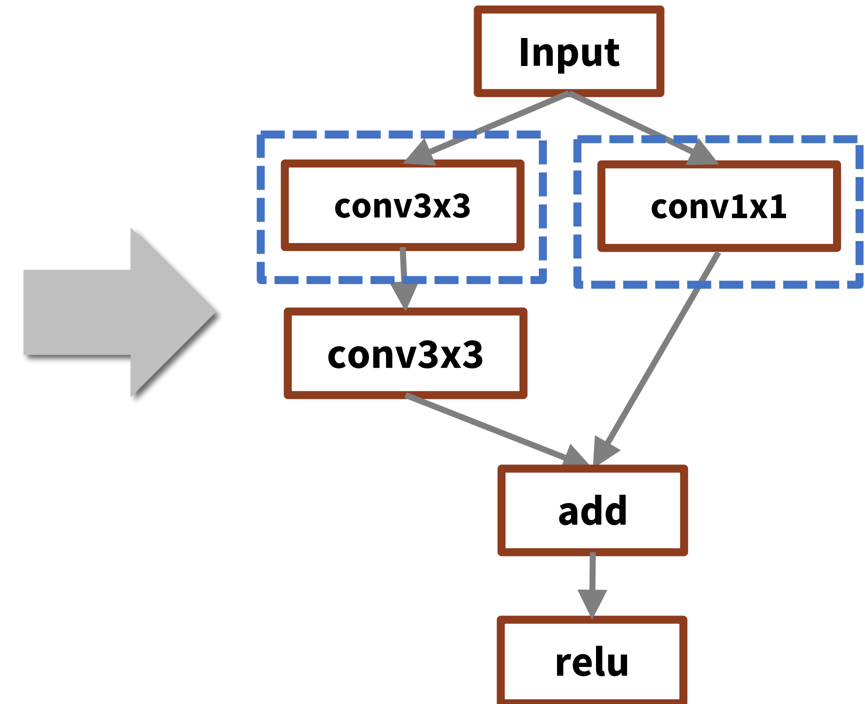
Graph-Level Optimizations



Input Computation Graph

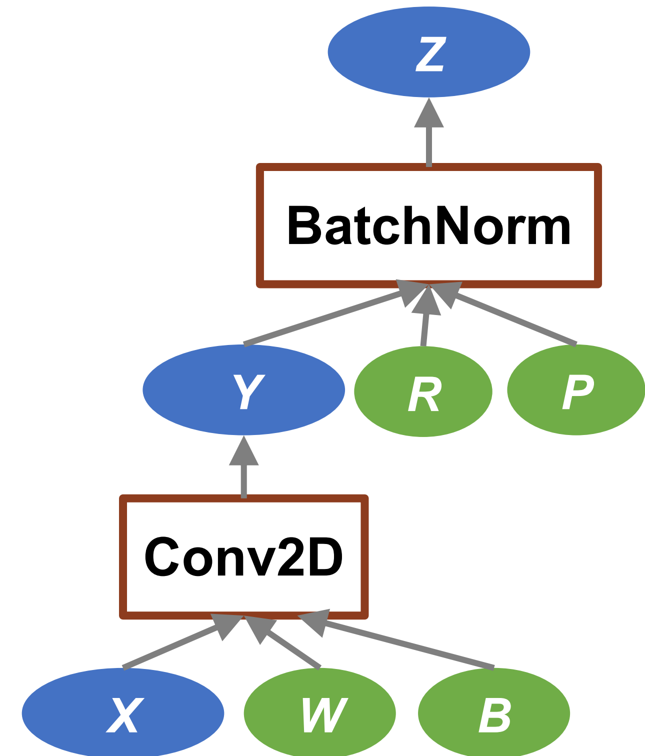


Potential graph transformations



Optimized Computation Graph

Example: Fusing Conv and Batch Normalization

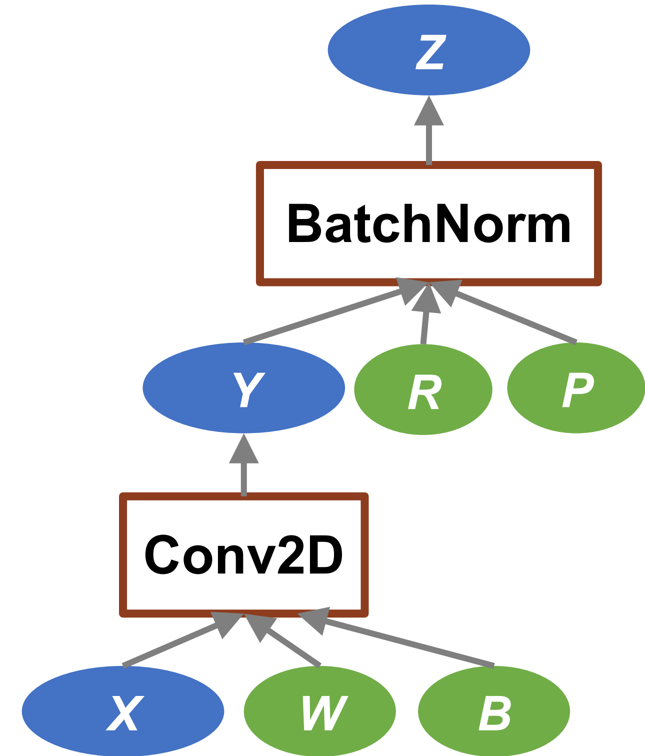


$$\boldsymbol{Z(n,c,h,w) = Y(n,c,h,w) * R(c) + P(c)}$$

$$\boldsymbol{Y(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W(c,d,u,v)\right) + B(n,c,h,w)}$$

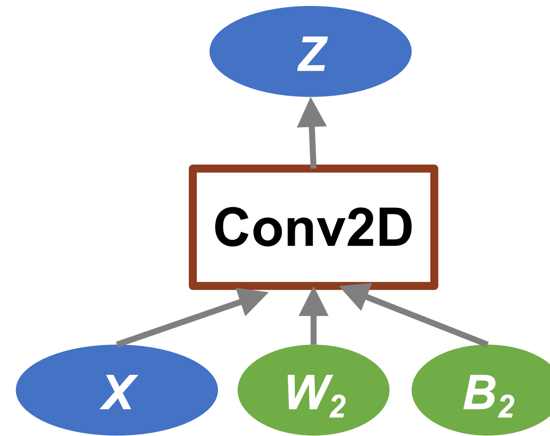
W, B, R, P are constant pre-trained weights

Fusing Conv and BatchNorm



$$Z(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W_2\,(c,d,u,v) \right) + B_2\,(n,c,h,w)$$

=

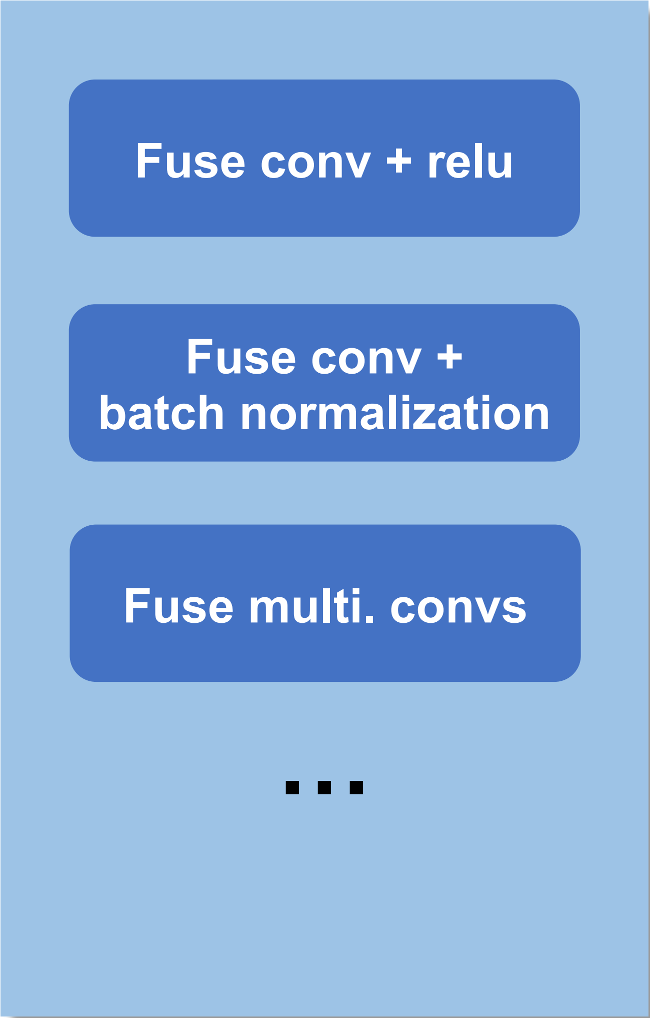


$$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$$

$$B_2(n,c,h,w) = B(n,c,h,w) * R(c) + P(c)$$

Current Rule-based Graph Optimizations

TensorFlow currently includes ~200 rules (~53,000 LOC)



Rule-based Optimizer

```
26  namespace tensorflow {
27  namespace graph_transforms {
28
29  // Converts Conv2D or MatMul ops followed by column-wise Muls into equivalent
30  // ops with the Mul baked into the convolution weights, to save computation
31  // during inference.
32  Status FoldBatchNorms(const GraphDef& input_graph_def,
33                        const TransformFuncContext& context,
34                        GraphDef* output_graph_def) {
35    GraphDef replaced_graph_def;
36    TF_RETURN_IF_ERROR(ReplaceMatchingOpTypes(
37        input_graph_def,  // clang-format off
38        {"Mul",                // mul_node
39          {
40            {"Conv2D|MatMul|DepthwiseConv2dNative",  // conv_node
41              {
42                {"*"},          // input_node
43                {"Const"},      // weights_node
44              }
45            },
46            {"Const"},          // mul_values_node
47          }
48        },  // clang-format on
49        [](const NodeMatch& match, const std::set<string>& input_nodes,
50           const std::set<string>& output_nodes,
51           std::vector<NodeDef>* new_nodes) {
52          // Find all the nodes we expect in the subgraph.
53          const NodeDef& mul_node = match.node;
54          const NodeDef& conv_node = match.inputs[0].node;
55          const NodeDef& input_node = match.inputs[0].inputs[0].node;
56          const NodeDef& weights_node = match.inputs[0].inputs[1].node;
57          const NodeDef& mul_values_node = match.inputs[1].node;
58
59          // Check that nodes that we use are not used somewhere else.
60          for (const auto& node : {conv_node, weights_node, mul_values_node}) {
61            if (output_nodes.count(node.name())) {
62              // Return original nodes.
63              new_nodes->insert(new_nodes->end(),
64                                {mul_node, conv_node, input_node, weights_node,
65                                 mul_values_node});
66              return Status::OK();
67            }
68          }
69
70          Tensor weights = GetNodeTensorAttr(weights_node, "value");
71          Tensor mul_values = GetNodeTensorAttr(mul_values_node, "value");
72
73          // Make sure all the inputs really are vectors, with as many entries as
74          // there are columns in the weights.
75          int64 weights_cols;
76          if (conv_node.op() == "Conv2D") {
77            weights_cols = weights.shape().dim_size(3);
78          } else if (conv_node.op() == "DepthwiseConv2dNative") {
79            weights_cols =
80                weights.shape().dim_size(2) * weights.shape().dim_size(3);
81          } else {
82            weights_cols = weights.shape().dim_size(1);
83          }
84          if ((mul_values.shape().dims() != 1) ||
85              (mul_values.shape().dim_size(0) != weights_cols)) {
86            return errors::InvalidArgument(
87                "Mul constant input to batch norm has bad shape: ",
88                mul_values.shape().DebugString());
89          }
90
91          // Multiply the original weights by the scale vector.
92          auto weights_vector = weights.flat<float>();
93          Tensor scaled_weights(DT_FLOAT, weights.shape());
94          auto scaled_weights_vector = scaled_weights.flat<float>();
95          for (int64 row = 0; row < weights_vector.dimension(0); ++row) {
96            scaled_weights_vector(row) =
97                weights_vector(row) *
98                mul_values.flat<float>()(row % weights_cols);
99          }
100
101         // Construct the new nodes.
102         NodeDef scaled_weights_node;
103         scaled_weights_node.set_op("Const");
104         scaled_weights_node.set_name(weights_node.name());
105         SetNodeAttr("dtype", DT_FLOAT, &scaled_weights_node);
106         SetNodeTensorAttr<float>("value", scaled_weights, &scaled_weights_node);
107         new_nodes->push_back(scaled_weights_node);
108
109         new_nodes->push_back(input_node);
110
111         NodeDef new_conv_node;
112         new_conv_node = conv_node;
113         new_conv_node.set_name(mul_node.name());
114         new_nodes->push_back(new_conv_node);
115
116         return Status::OK();
117       },
118       {}, &replaced_graph_def));
119   *output_graph_def = replaced_graph_def;
120   return Status::OK();
121 }
122
123 REGISTER_GRAPH_TRANSFORM("fold_batch_norms", FoldBatchNorms);
124
125 }  // namespace graph_transforms
126 }  // namespace tensorflow
```

Limitations of Rule-based Optimizations

Robustness
Experts' heuristics do not apply to all models/hardware

Code | Issues 250 | Pull requests 11 | Wiki | Insights

Horovod with XLA is slower than without XLA (Tensorflow 1.12) #713

New issue

Closed · LiweiPeng opened this issue on Dec 19, 2018 · 2 comments

LiweiPeng commented on Dec 19, 2018

I have a distributed nmt model (Transformer-based, AdamOptimizer) using Horovod (0.15.1). When I turned on XLA under tensorflow 1.12, the training speed is about 20% slower instead of faster.

This result is sampled after training 1.5-hours and 4000 steps. I am using 4 V100 GPUs for the training.

Because my current software is tightly coupled with Horovod, I couldn't test whether this is Horovod related or not.

Does anyone have experience on whether this is expected?

tgaddair added the question label on Dec 19, 2018

Assignees: No one assigned
Labels: question
Milestone: No milestone
Notifications: Subscribe

When I turned on XLA (TensorFlow's graph optimizer), the training speed is **about 20% slower**

stackoverflow | Search...

Home · PUBLIC · Stack Overflow · Tags · Users · Jobs · Teams Q&A for work · Learn More

Tensorflow XLA makes it slower?

1

I am writing a very simple tensorflow program with `XLA` enabled. Basically it's something like:

```
import tensorflow as tf

def ChainSoftMax(x, n)
  tensor = tf.nn.softmax(x)
  for i in range(n-1):
    tensor = tf.nn.softmax(tensor)
  return tensor

config = tf.ConfigProto()
config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1

input = tf.placeholder(tf.float32, [1000])
feed = np.random.rand(1000).astype('float32')

with tf.Session(config=config) as sess:
  res = sess.run(ChainSoftMax(input, 2000), feed_dict={input: feed})
```

Basically the idea is to see whether XLA can fuse the chain of `softmax` together to avoid multiple kernel launches. With XLA on, the above program is almost 2x slower than that without XLA on a machine with a GPU card. In my gpu profile, I saw XLA produces lots of kernels named as "`reduce_xxx`" and "`fusion_xxx`" which seem to overwhelm the overall runtime. Any one know what happened here?

With XLA, my program is **almost 2x slower than** without XLA

Limitations of Rule-based Optimizations

Robustness
Experts' heuristics do not apply to all models/hardware

Scalability
New operators and graph structures require more rules

TensorFlow currently uses ~4K LOC to optimize convolution

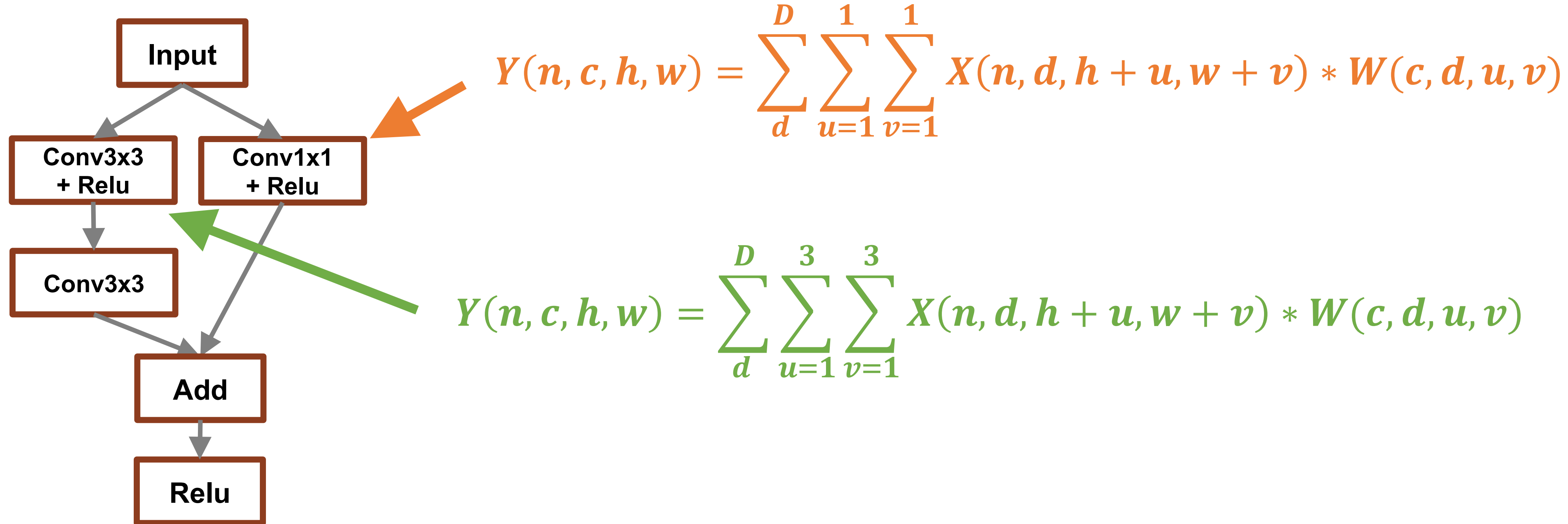
Limitations of Rule-based Optimizations

Robustness
Experts' heuristics do not apply to all models/hardware

Scalability
New operators and graph structures require more rules

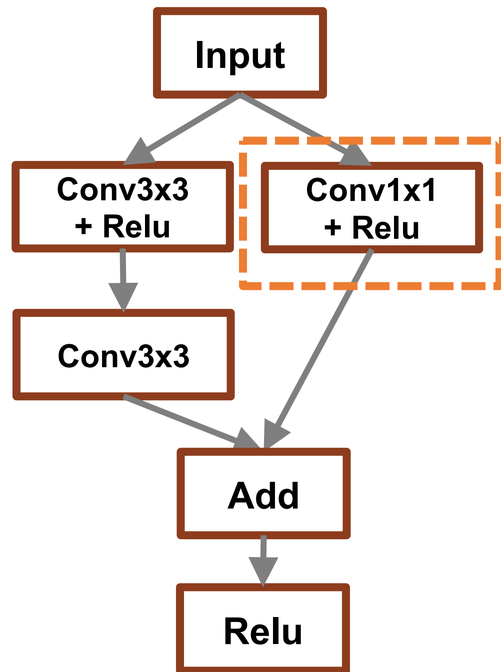
Performance
Miss subtle optimizations for specific models/hardware

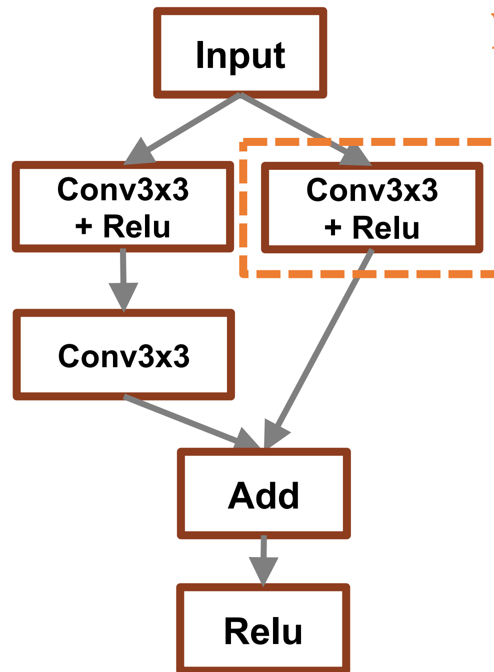
Motivating Example (ResNet*)



* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015

Motivating Example (ResNet*)





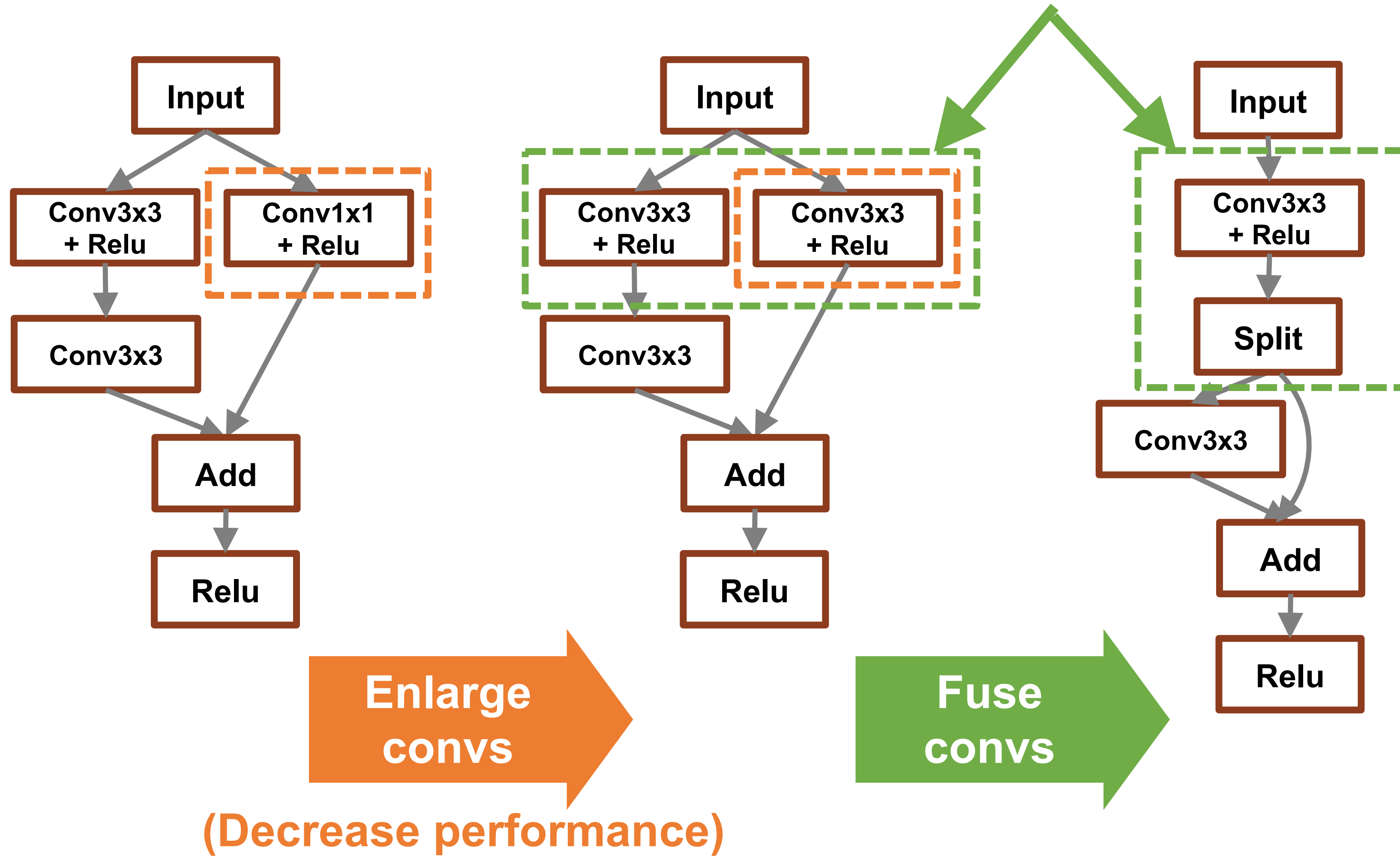
$$Y(n,c,h,w) = \sum_{d} \sum_{u=1}^{3} \sum_{v=1}^{3} X(n,d,h+u,w+v) * W(c,d,u,v)$$

Enlarge convs

(Decrease performance)

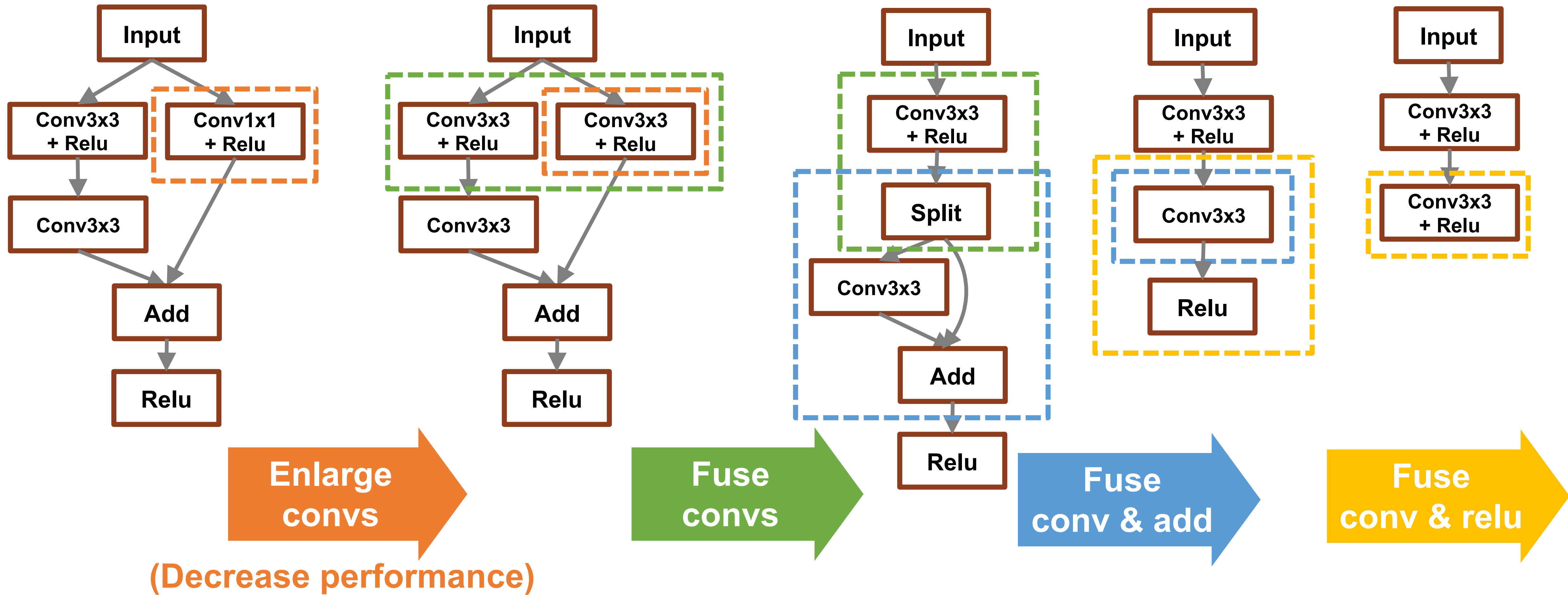
* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015

$$Y(n,c,h,w) = \sum_{d}^{D} \sum_{u=1}^{3} \sum_{v=1}^{3} X(n,d,h+u,w+v) * W'(c,d,u,v)$$



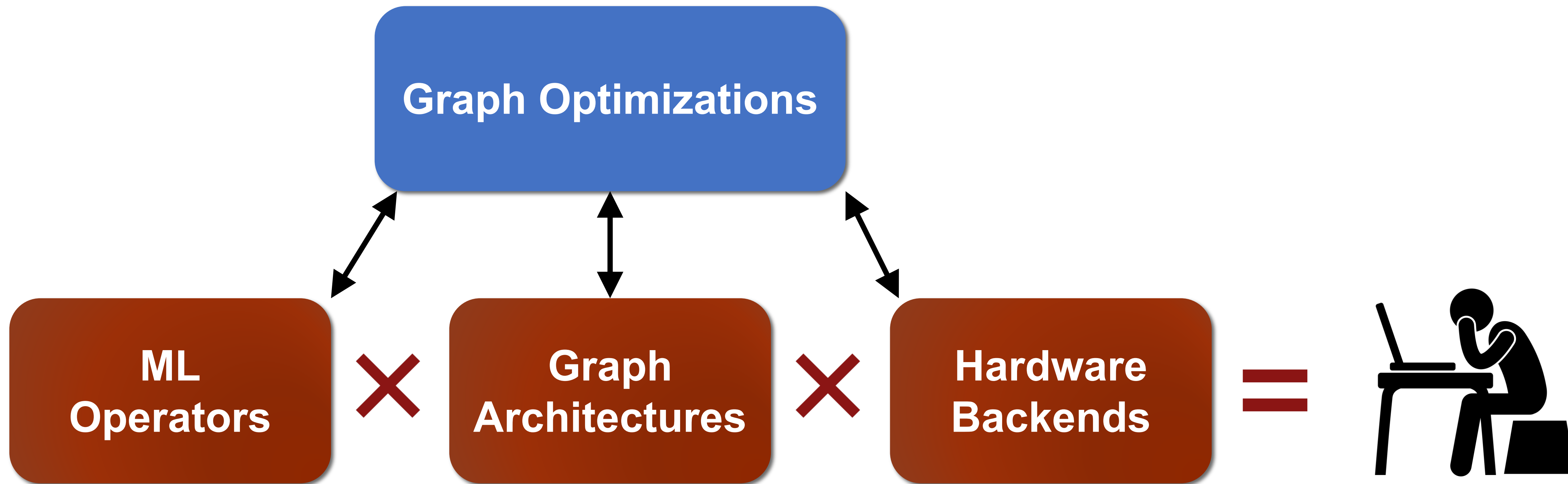
* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015

Motivating Example (ResNet*)



The final graph is 30% faster on V100 GPU but 10% slower on K80 GPU.

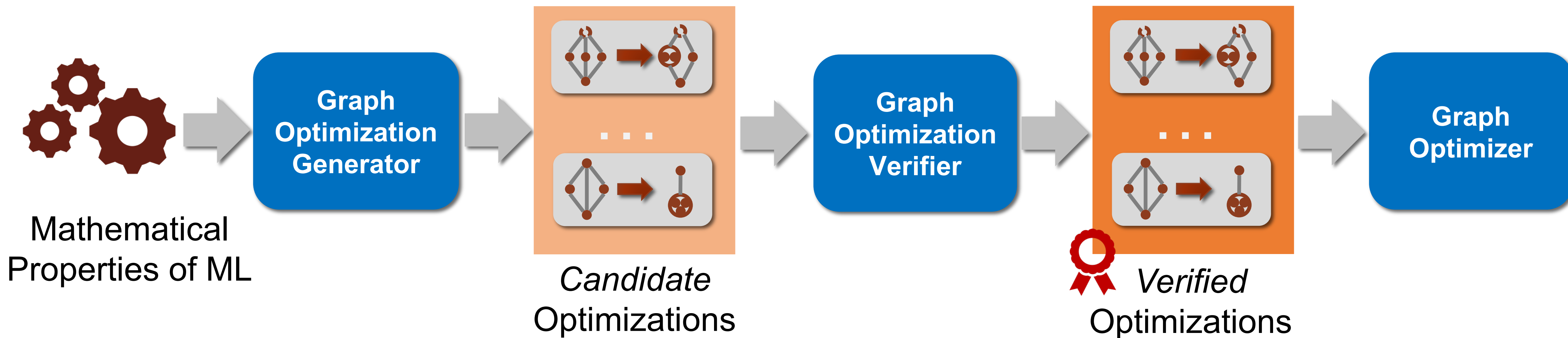
* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2016



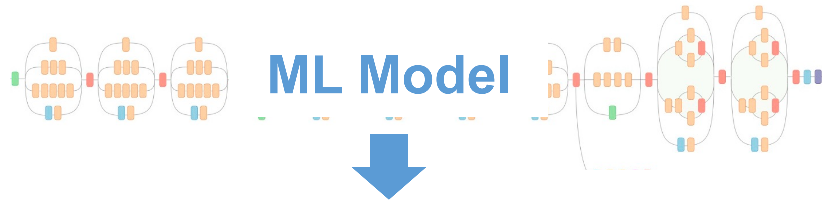
Infeasible to manually design graph optimizations for all cases

Automated Generation and Verification of Graph Optimizations

- Week 5: Graph-Level Optimizations
- Week 5: RL for Device Placement and Graph Optimizations



An Overview of Deep Learning Systems



Automatic Differentiation

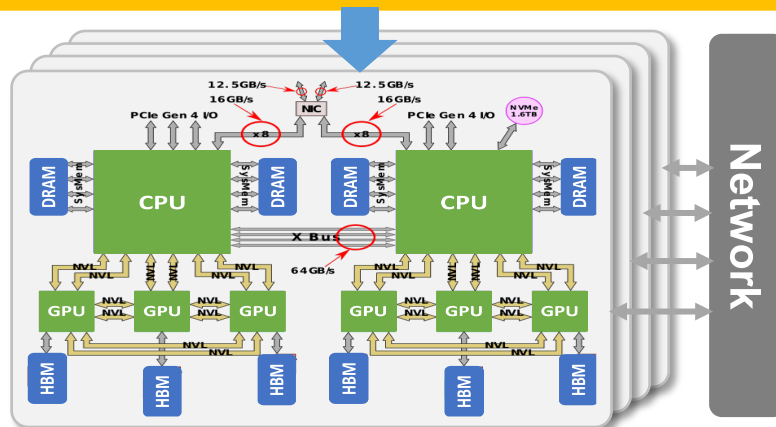
Graph-Level Optimization

Parallelization / Distributed Training

Data Layout and Placement

Kernel Optimizations

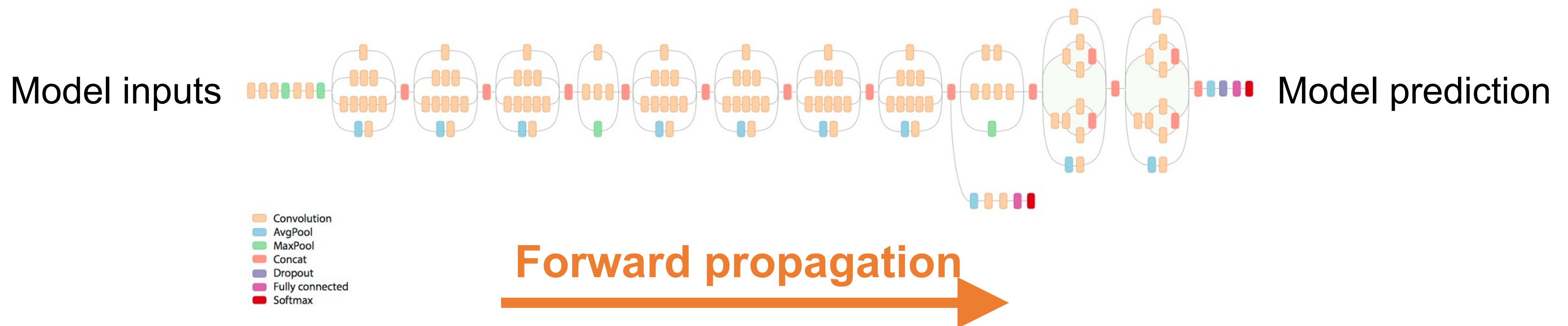
Memory Optimizations



Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

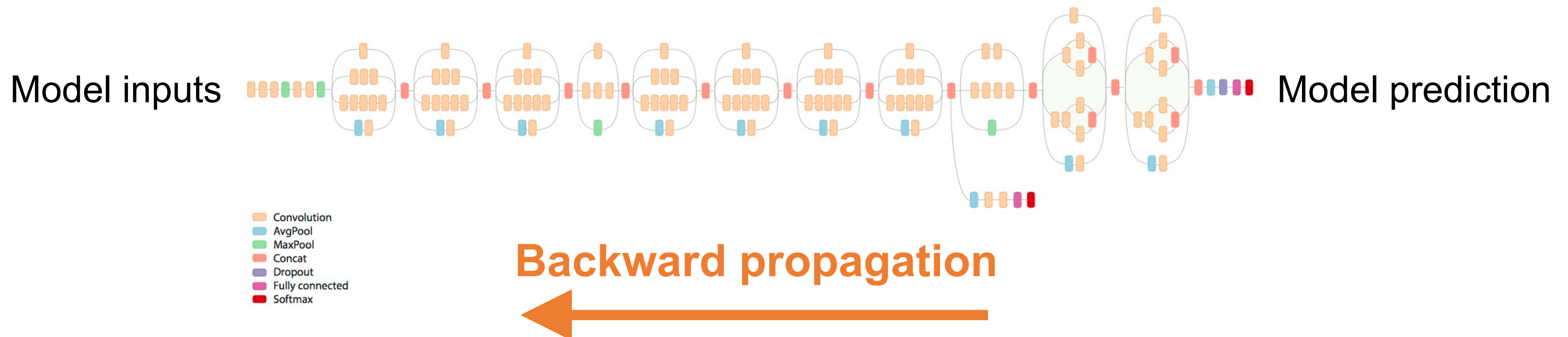
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights



Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights



Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

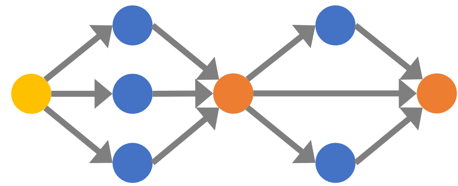
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

How can we parallelize ML training?

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

Data Parallelism



ML Model

Training Dataset

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

GPU 1

GPU 2

...

GPU N

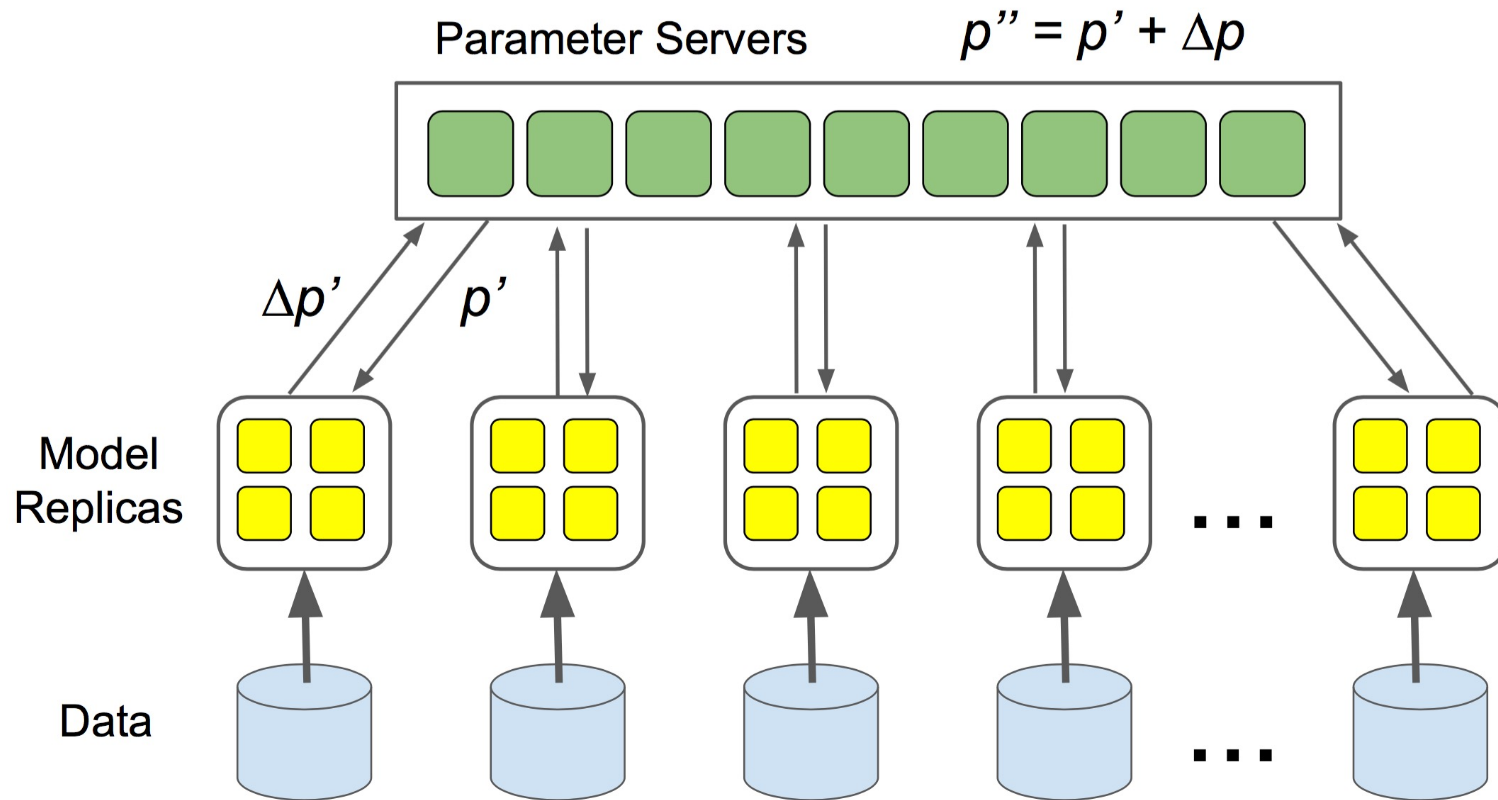
Gradients Aggregation

1. Partition training data into batches

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

Data Parallelism: Parameter Server

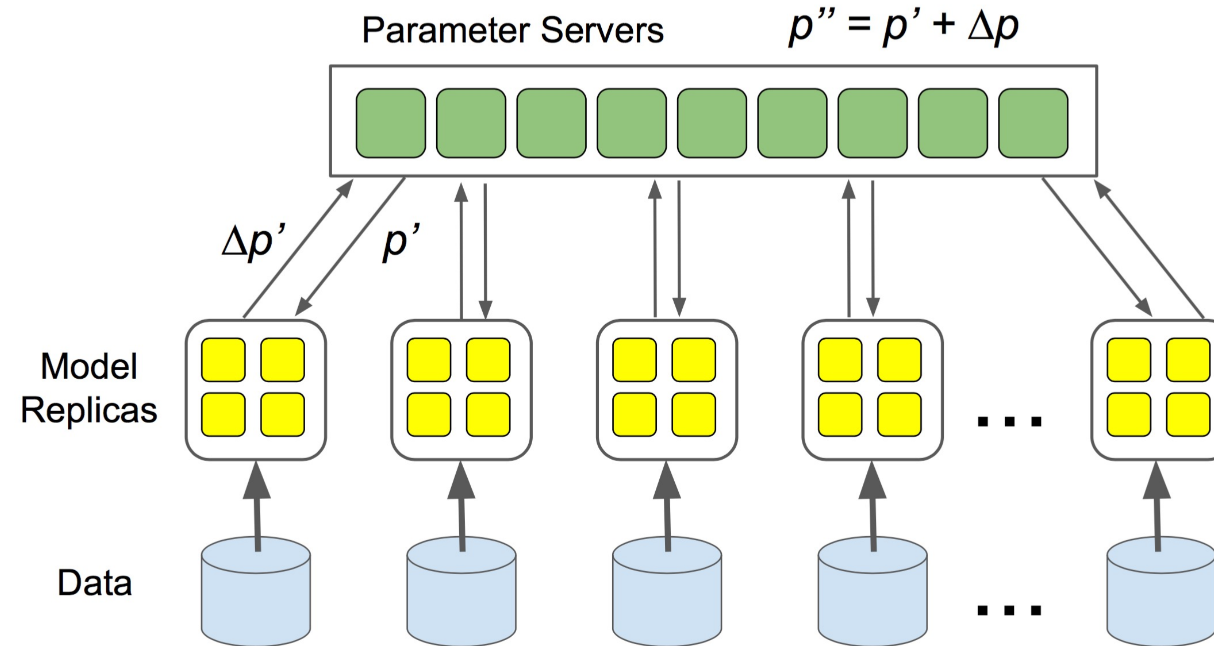


Workers push gradients to parameter servers and pull updated parameters back

Inefficiency of Parameter Server

- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers

- How can we decentralize communication in DNN training?

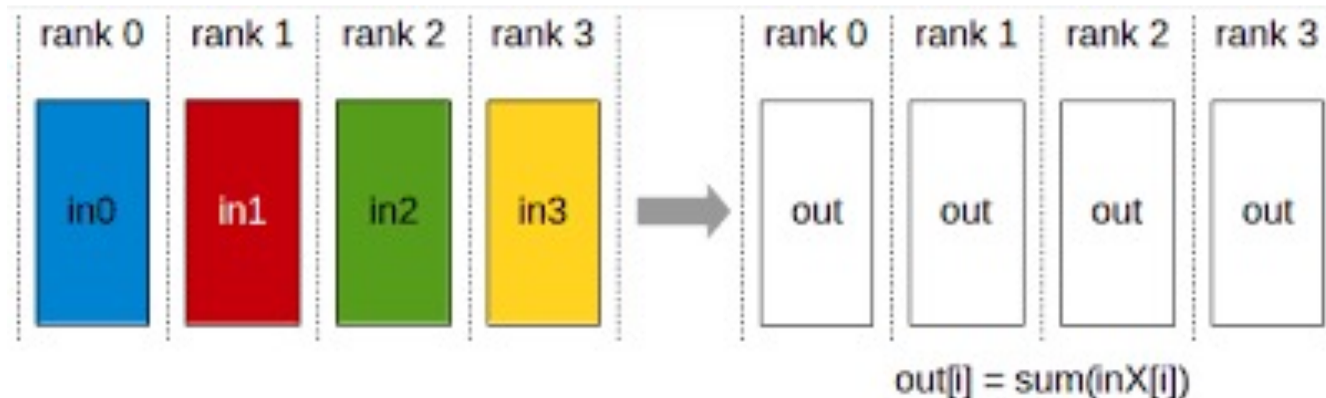


Inefficiency of Parameter Server

- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers

- How can we decentralize communication in DNN training?

- **AllReduce**: perform element-wise reduction across multiple devices

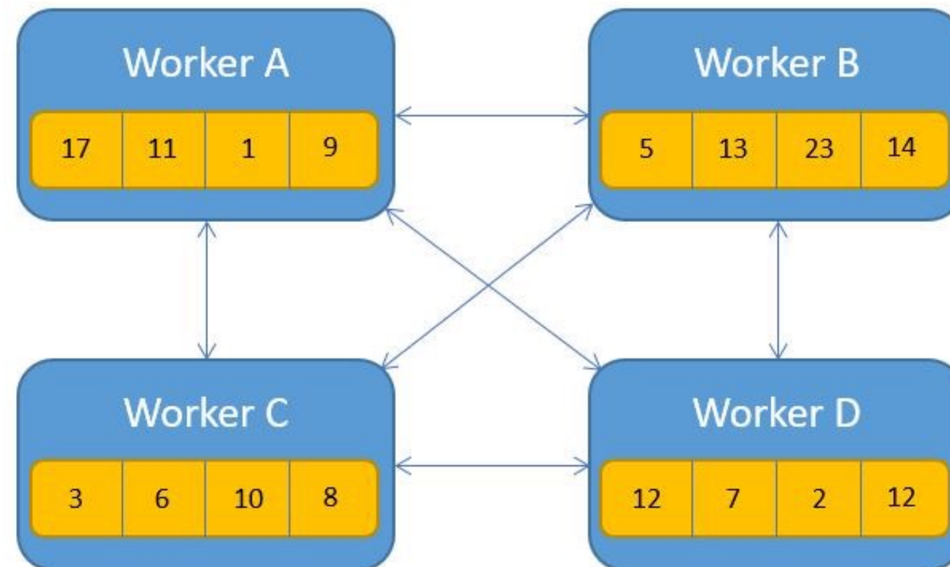


Different Ways to Perform AllReduce

- Naïve AllReduce
- Ring AllReduce
- Tree AllReduce
- Butterfly AllReduce

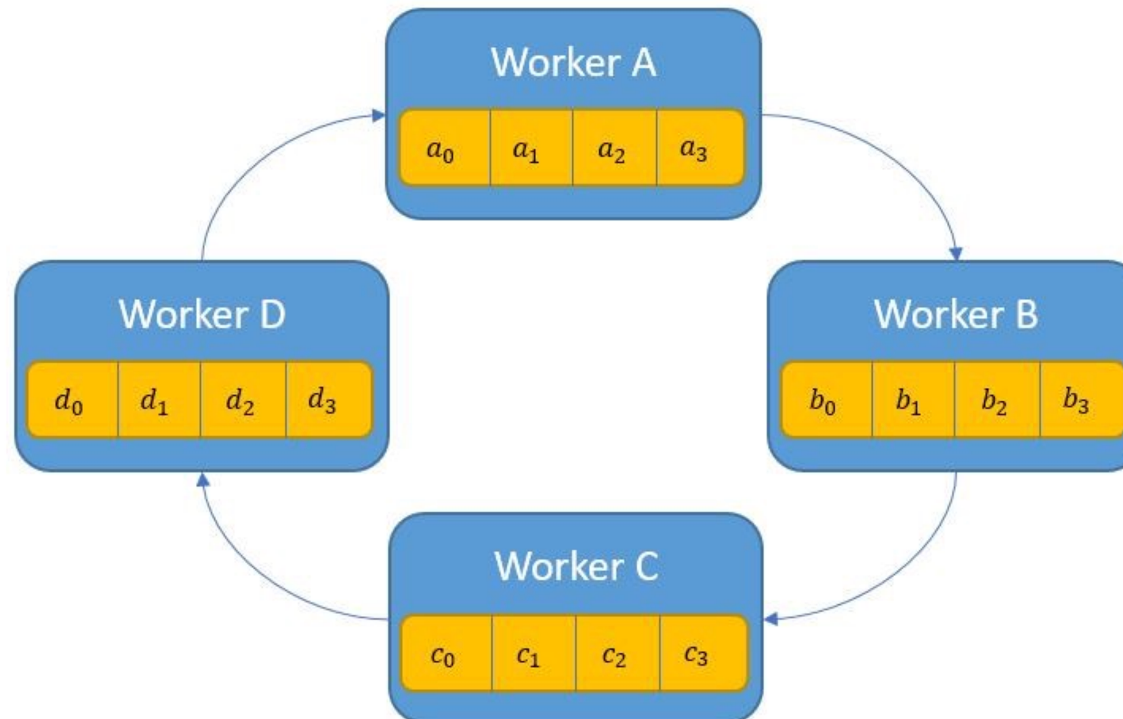
Naïve AllReduce

- Each worker can send its local gradients to all other workers
- If we have N workers and each worker contains M parameters
- Overall communication: N * (N-1) * M parameters
- **Issue**: each worker communicates with all other workers; have the same scalability issue as parameter server



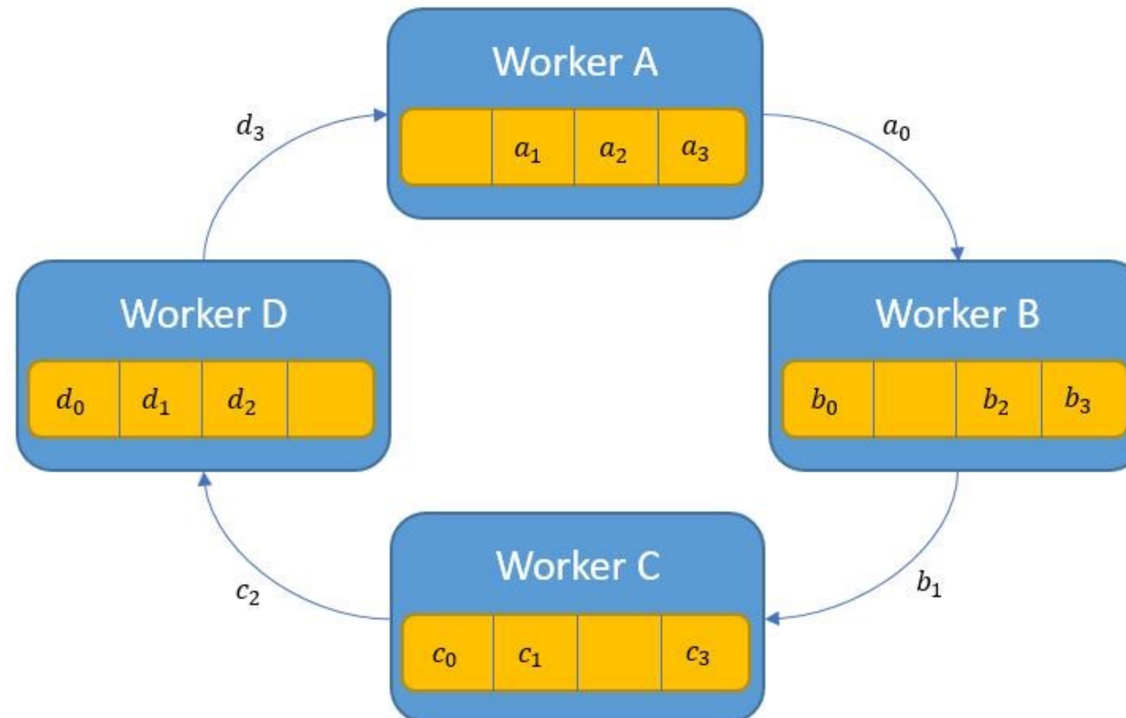
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



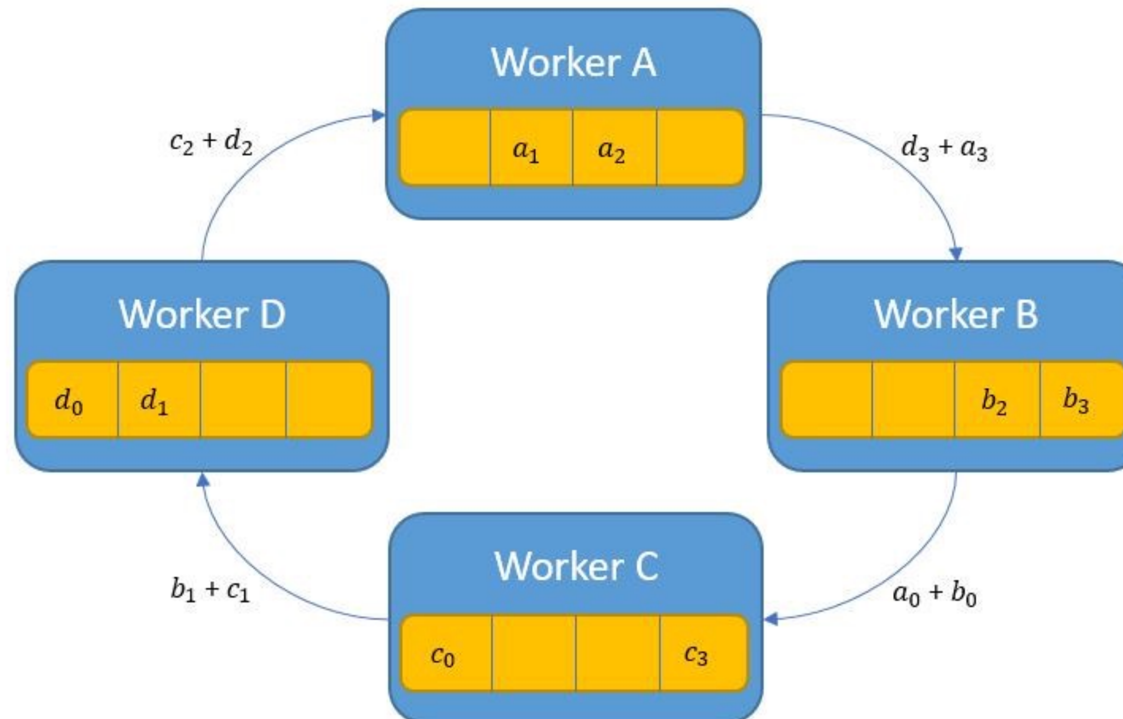
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



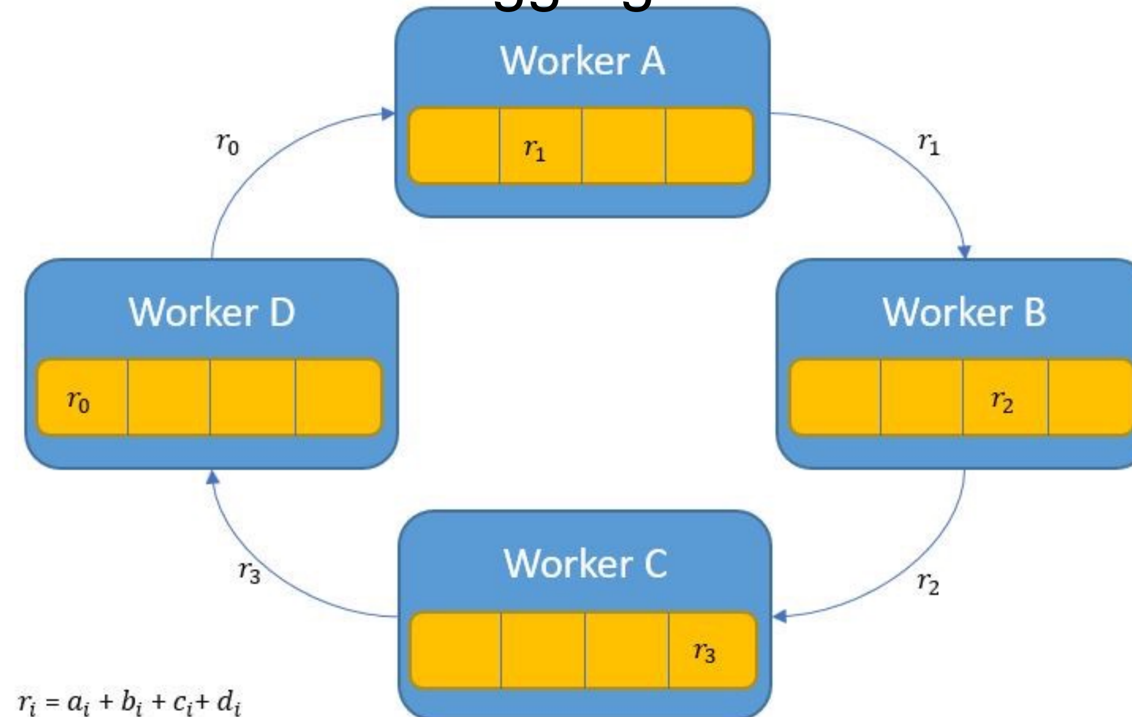
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



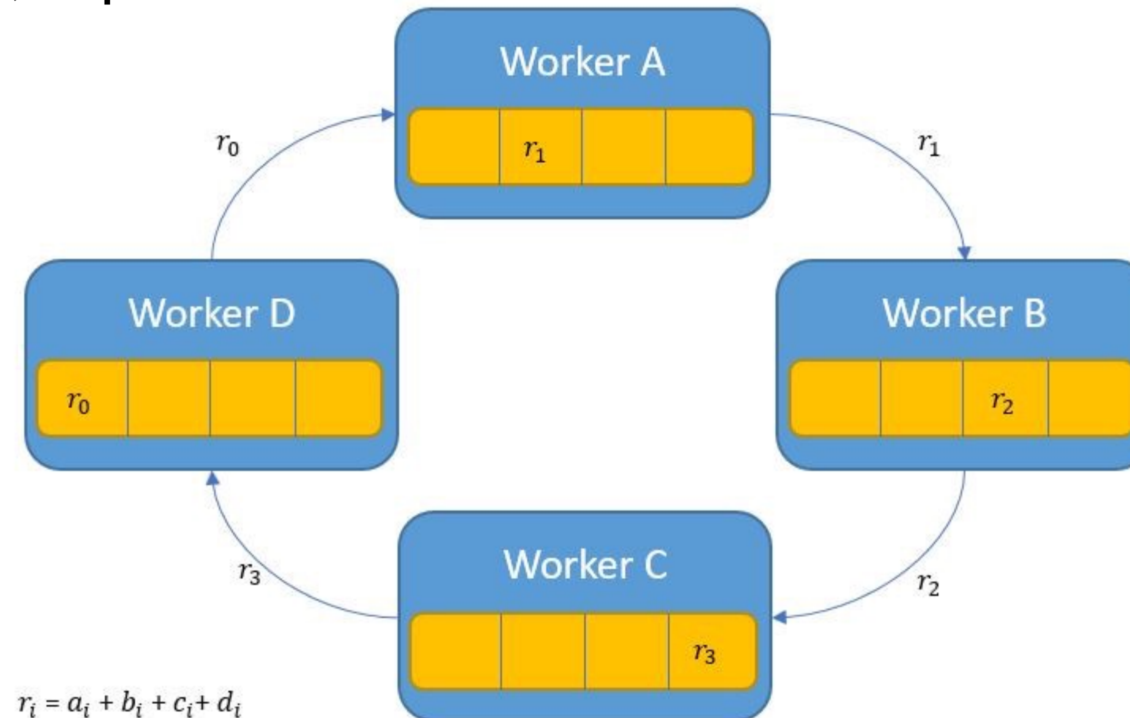
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- After step 1, each worker has the aggregated version of M/N parameters



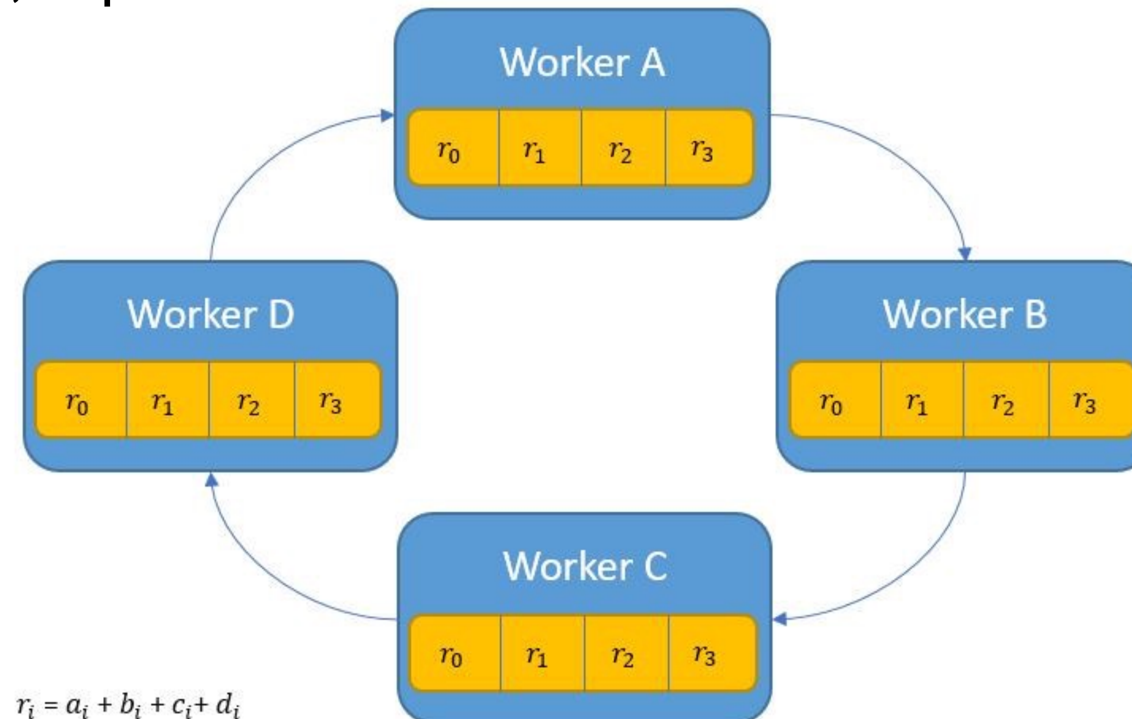
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times



Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times

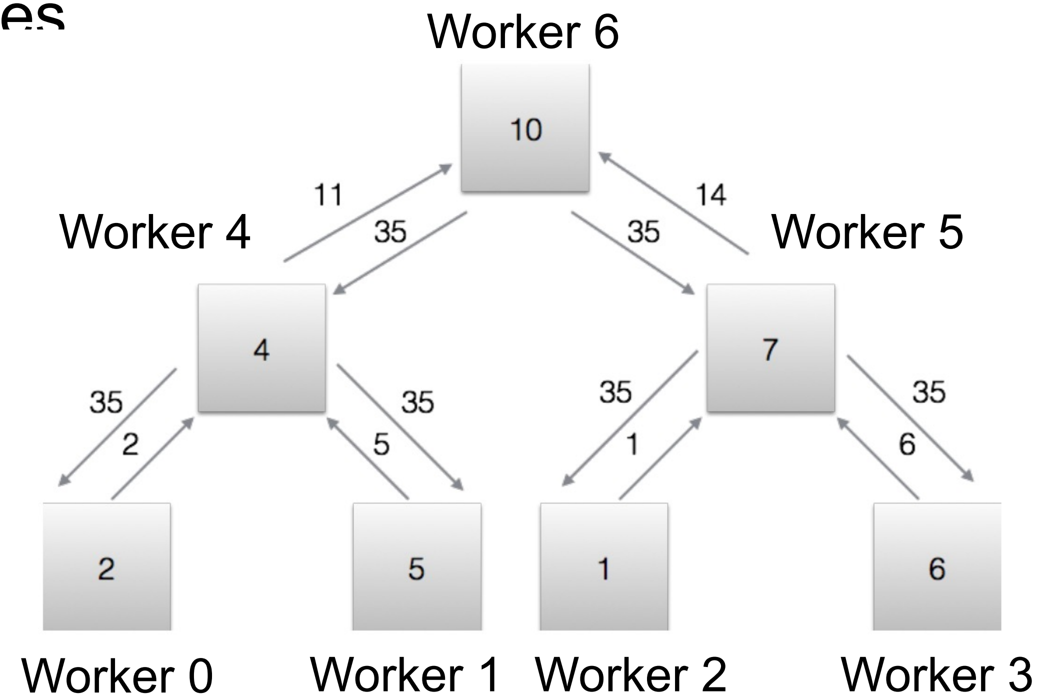


Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times
- Overall communication: 2 * M * N parameters
  - Aggregation: M * N parameters
  - Broadcast: M * N parameters

Tree AllReduce

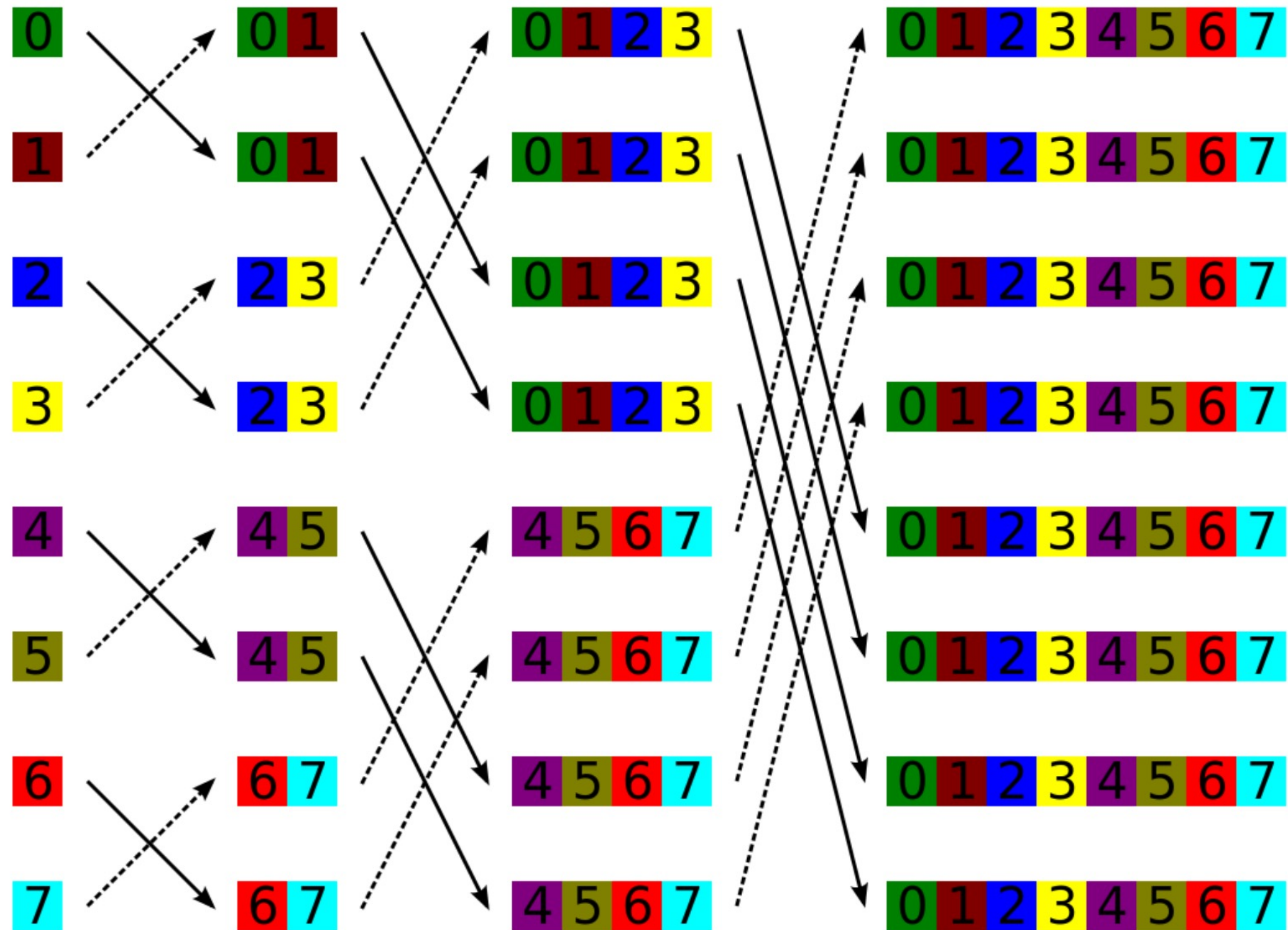
- Construct a tree of N workers;
- Step 1 (Aggregation): each worker sends M parameters to its parent; repeat log(N) times
- Step 2 (Broadcast): each worker sends M parameters to its children; repeat log(N) times



Tree AllReduce

- Construct a tree of N workers;
- Step 1 (Aggregation): each worker sends M parameters to its parent; repeat log(N) times
- Step 2 (Broadcast): each worker sends M parameters to its children; repeat log(N) times
- Overall communication: 2 * N * M parameters
  - Aggregation: M * N parameters
  - Broadcast: M * N parameters

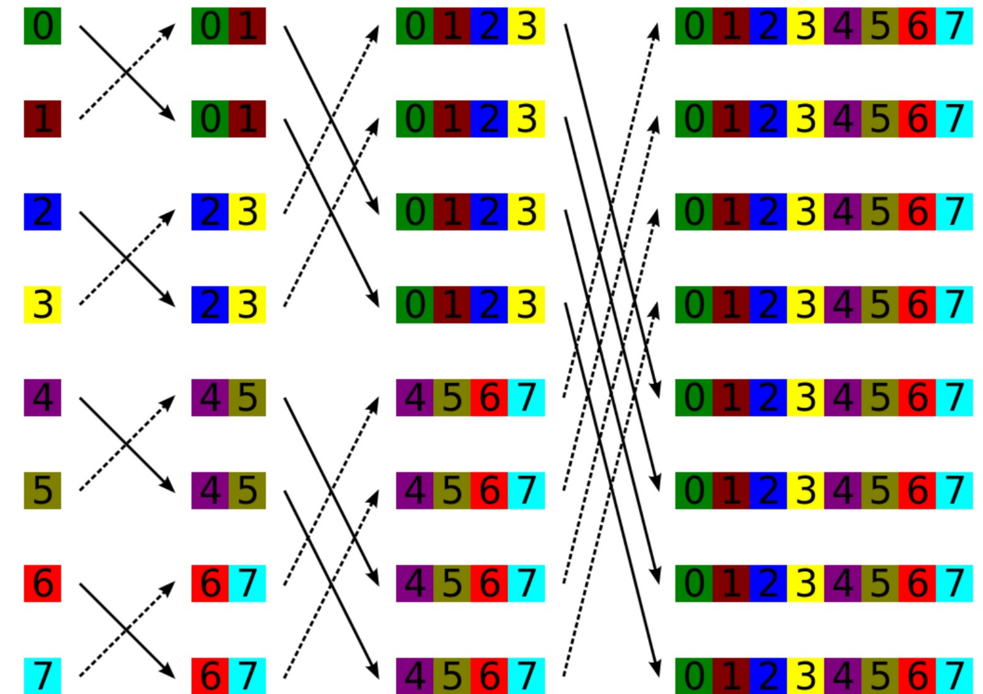
Butterfly Network



Butterfly AllReduce

- Repeat log(N) times:
  1. Each worker sends M parameters to its target node in the butterfly network
  2. Each worker aggregates gradients locally

- Overall communication: N * M * log(N) parameters



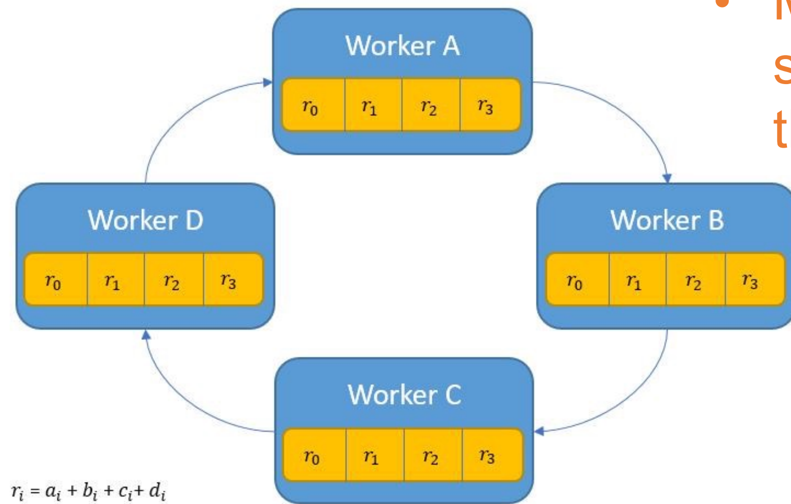
Comparing different AllReduce Methods

	Parameter Server	Naïve AllReduce	Ring AllReduce	Tree AllReduce	Butterfly AllReduce
Overall communication	$2 \times N \times M$	$N^2 \times M$	$2 \times N \times M$	$2 \times N \times M$	$N \times M \times \log N$

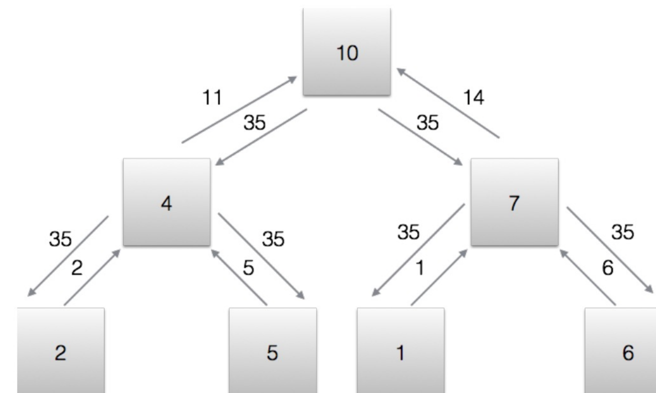
Question: Ring AllReduce is more efficient and scalable then Tree AllReduce and Parameter Server, why?

Ring AllReduce v.s. Tree AllReduce v.s. Parameter Server

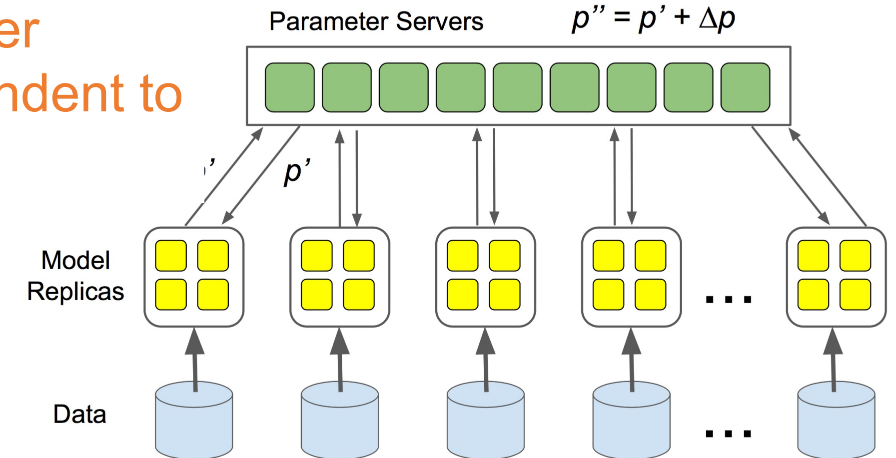
Ring AllReduce:
- Best latency
- Balanced workload across workers
- More scalable since each worker sends 2*M parameters (independent to the number of workers)



Each worker sends **M/N** parameters per iteration; repeat for **2*N** iterations
Latency: M/N * (2*N) / bandwidth



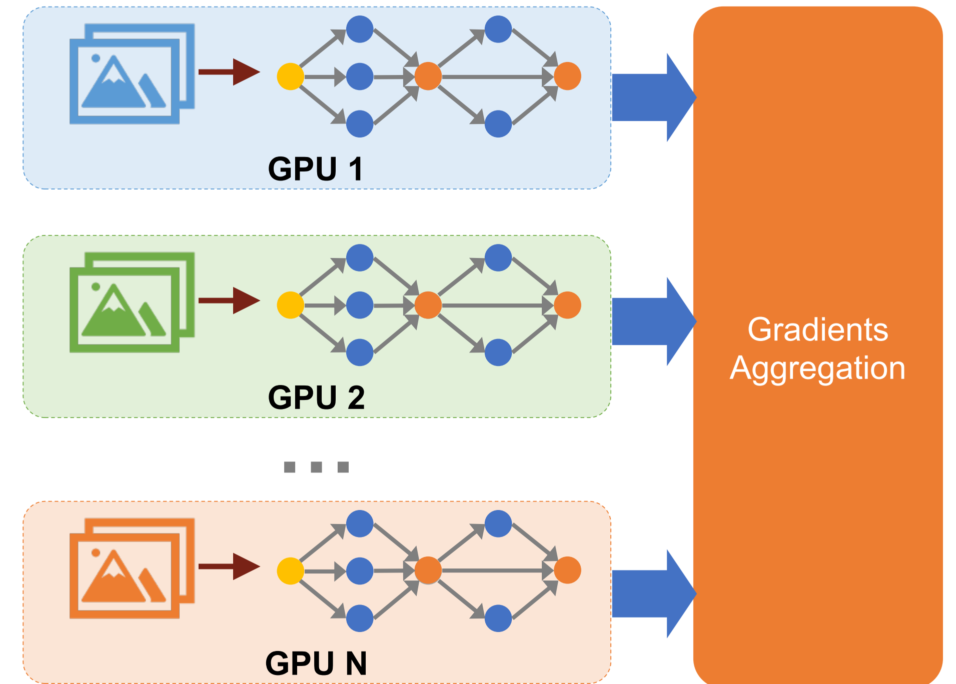
Each worker sends **M** parameters per iteration; repeat for **2*log(N)** iterations
Latency: M * 2 * log(N) / bandwidth



All workers send **M** parameters to parameter servers and receive **M** parameters from servers
Latency: M * N / bandwidth

Recap: Data Parallelism

Each worker keeps a replica of the entire model and communicates with other workers to synchronize weights updates

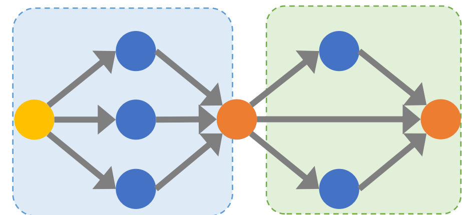


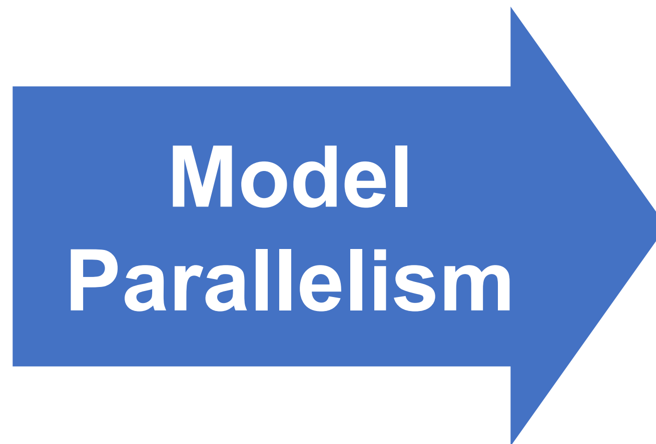
Gradients aggregation methods:

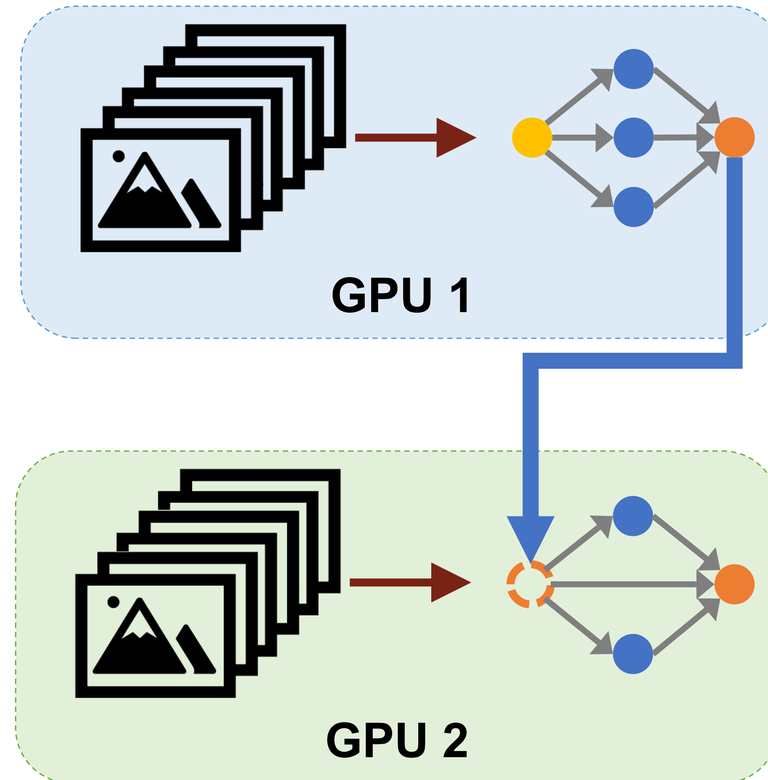
- Parameter Server
- Ring AllReduce
- Tree AllReduce
- Butterfly AllReduce
- Etc.

Model Parallelism

- Split a model into multiple subgraphs and assign them to different devices



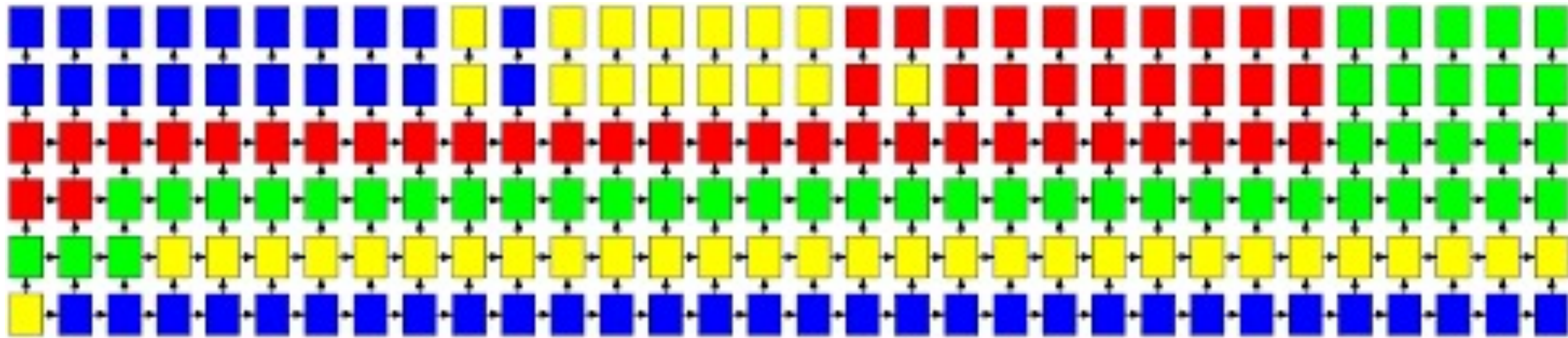




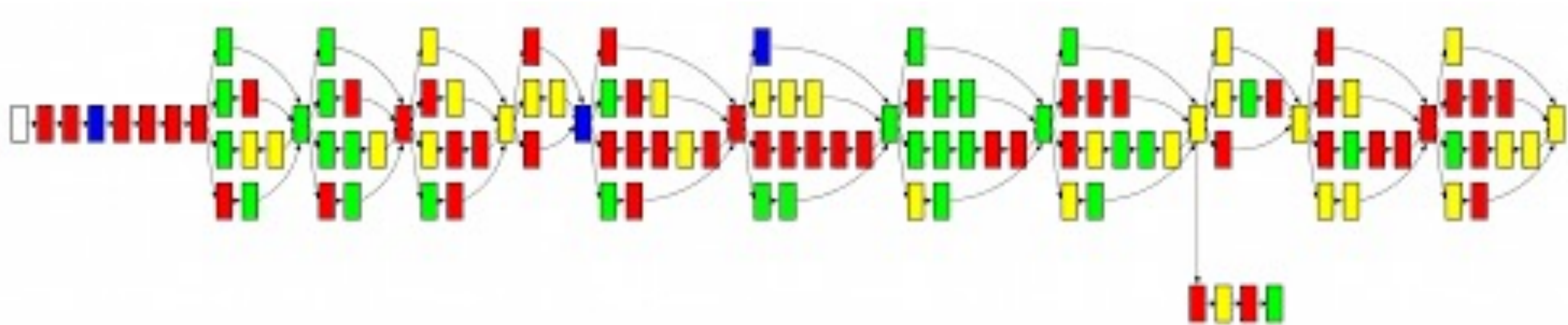
Transfer intermediate results between devices

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

Model Parallelism



Model parallelism: training a RNN on 4 GPUs



Model parallelism: training Inception-v3 on 4 GPUs

Device placement optimization with reinforcement learning. A Mirhoseini et al.

Pipeline Parallelism

- Divide a mini-batch into multiple micro-batches
- Pipeline the forward/backward computations across micro-batches
- Generally combined with model parallelism

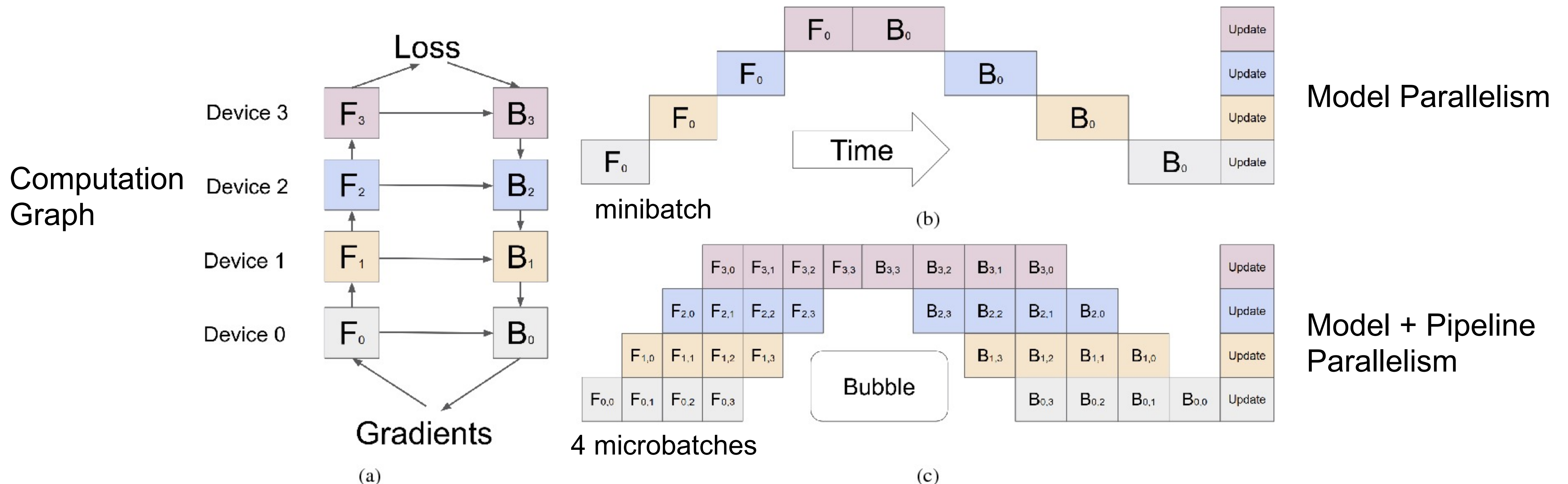
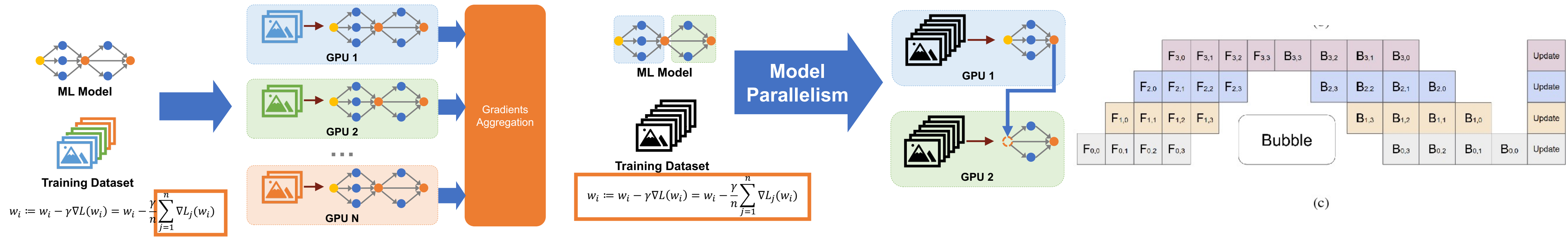


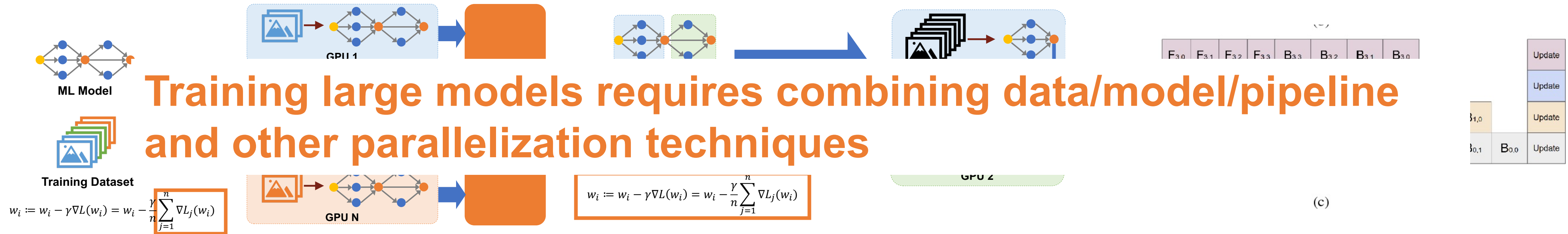
Figure from GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding

Comparing Data/Model/Pipeline Parallelism



	Data Parallelism	Model Parallelism	Pipeline Parallelism
Pros	✓ Massively parallelizable ✓ Require no communication during forward/backward	✓ Support training large models ✓ Efficient for models with large numbers of parameters	✓ Support large-batch training
Cons	❖ Do not work for models that cannot fit on a GPU ❖ Do not scale for models with large numbers of parameters	❖ Limited parallelizability; cannot scale to large numbers of GPUs ❖ Need to transfer intermediate results in forward/backward	❖ Limited utilization: bubbles in forward/backward

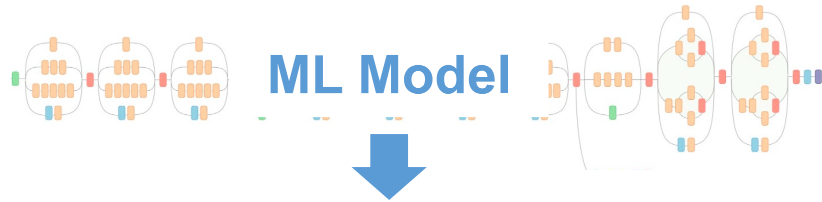
Comparing Data/Model/Pipeline Parallelism



Training large models requires combining data/model/pipeline and other parallelization techniques

	Data Parallelism	Model Parallelism	Pipeline Parallelism
Pros	✓ Massively parallelizable ✓ Require no communication during forward/backward	✓ Support training large models ✓ Efficient for models with large numbers of parameters	✓ Support large-batch training
Cons	❖ Do not work for models that cannot fit on a GPU ❖ Do not scale for models with large numbers of parameters	❖ Limited parallelizability; cannot scale to large numbers of GPUs ❖ Need to transfer intermediate results in forward/backward	❖ Limited utilization: bubbles in forward/backward

An Overview of Deep Learning Systems



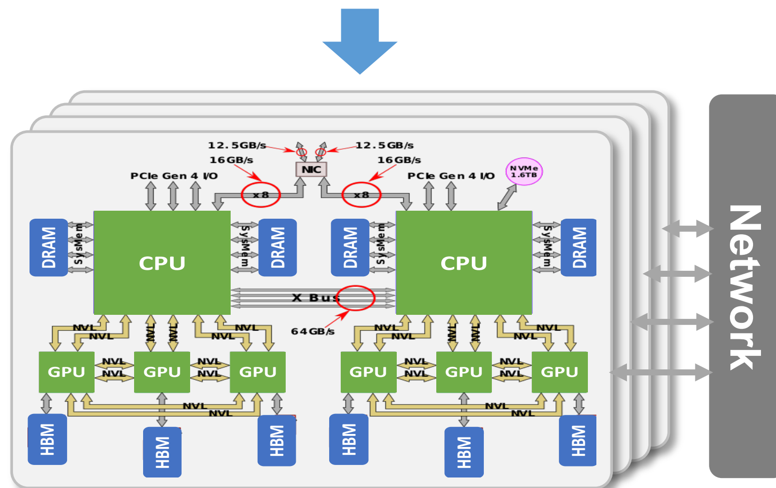
Automatic Differentiation

Graph-Level Optimization

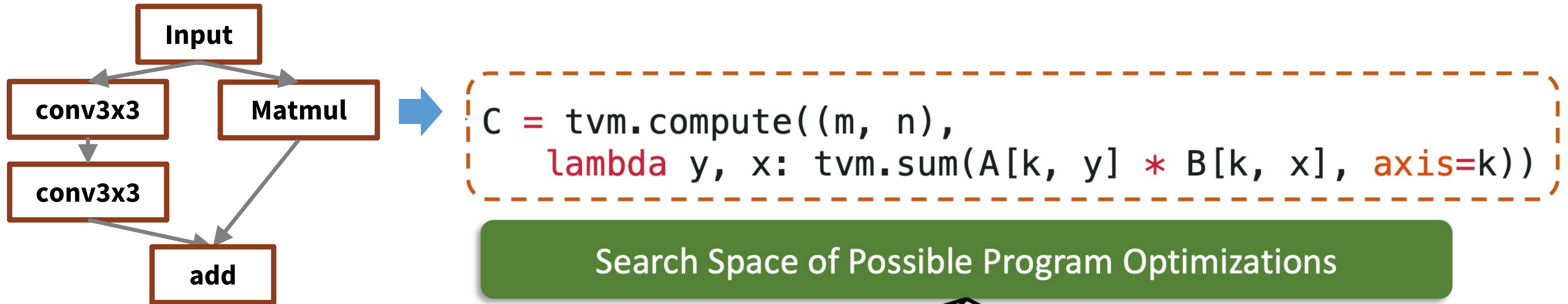
Parallelization / Distributed Training

Code Optimization

Memory Optimization



Code Optimization: How to find performant programs for each operator?



Low-level Program Variants

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
                A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

* Slides from Tianqi Chen

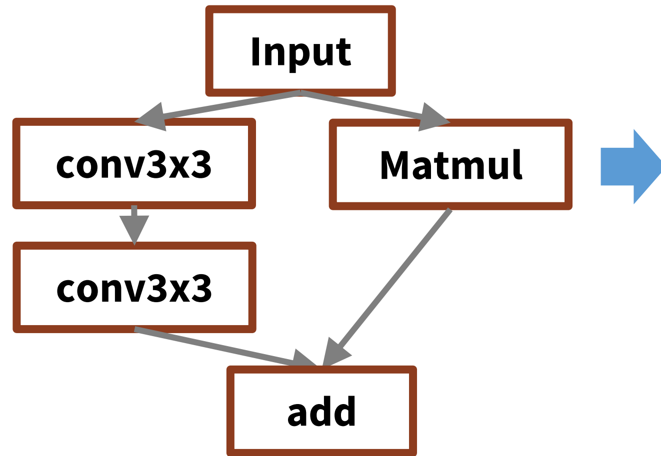
Existing Approach: Engineer Optimized Tensor Programs

- Hardware vendors provide operator libraries manually developed by software/hardware engineers
- cuDNN, cuBLAS, cuRAND, cuSPARSE for GPUs
  - cudnnConvolutionForward() for convolution
  - cublasSgemm() for matrix multiplication

Issues:

- Cannot provide immediate support for new operators
- Increasing complexity of hardware -> hand-written kernels are suboptimal

Automated Code Generation




```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Automated search for performant programs:

✓ Immediate support for new operators

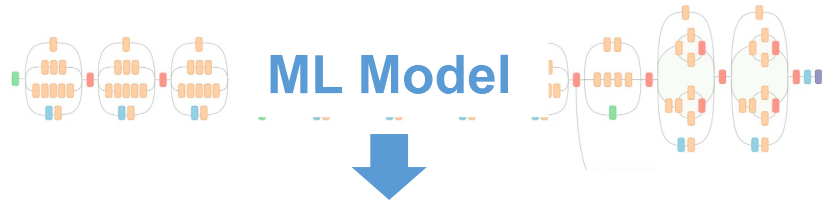
✓ Better performance than hand-written kernels

Loop Transformations | Thread Bindings | Cache Locality

Thread Cooperation | Tensorization | Latency Hiding | ...

* Slides from Tianqi Chen

An Overview of Deep Learning Systems



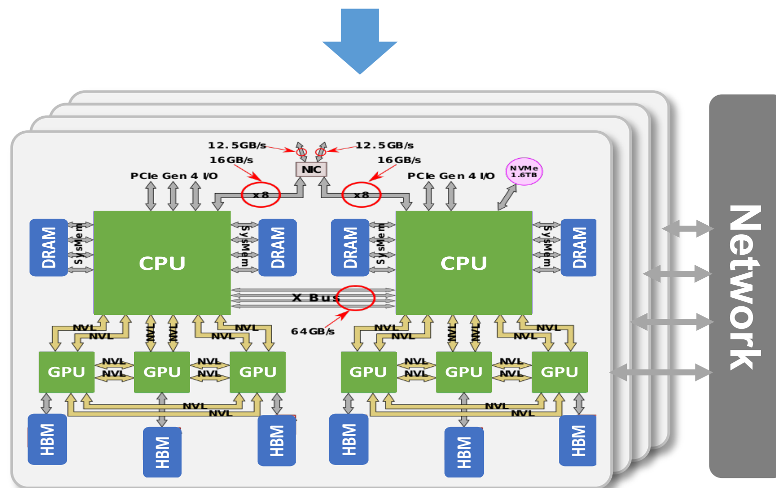
Automatic Differentiation

Graph-Level Optimization

Parallelization / Distributed Training

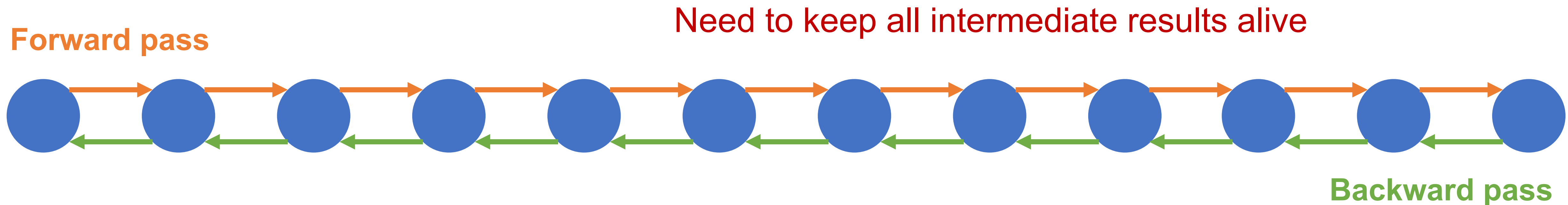
Code Optimization

Memory Optimization

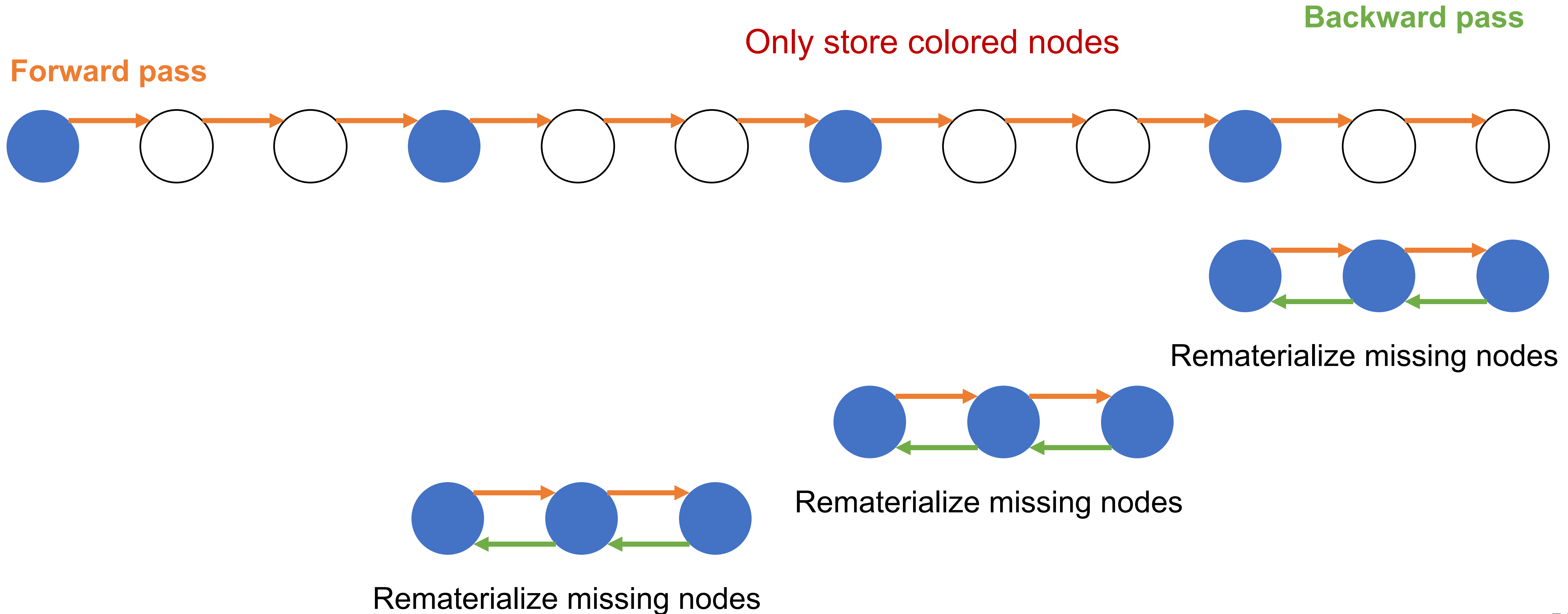


Recap: GPU Memory is the Bottleneck in DNN Training

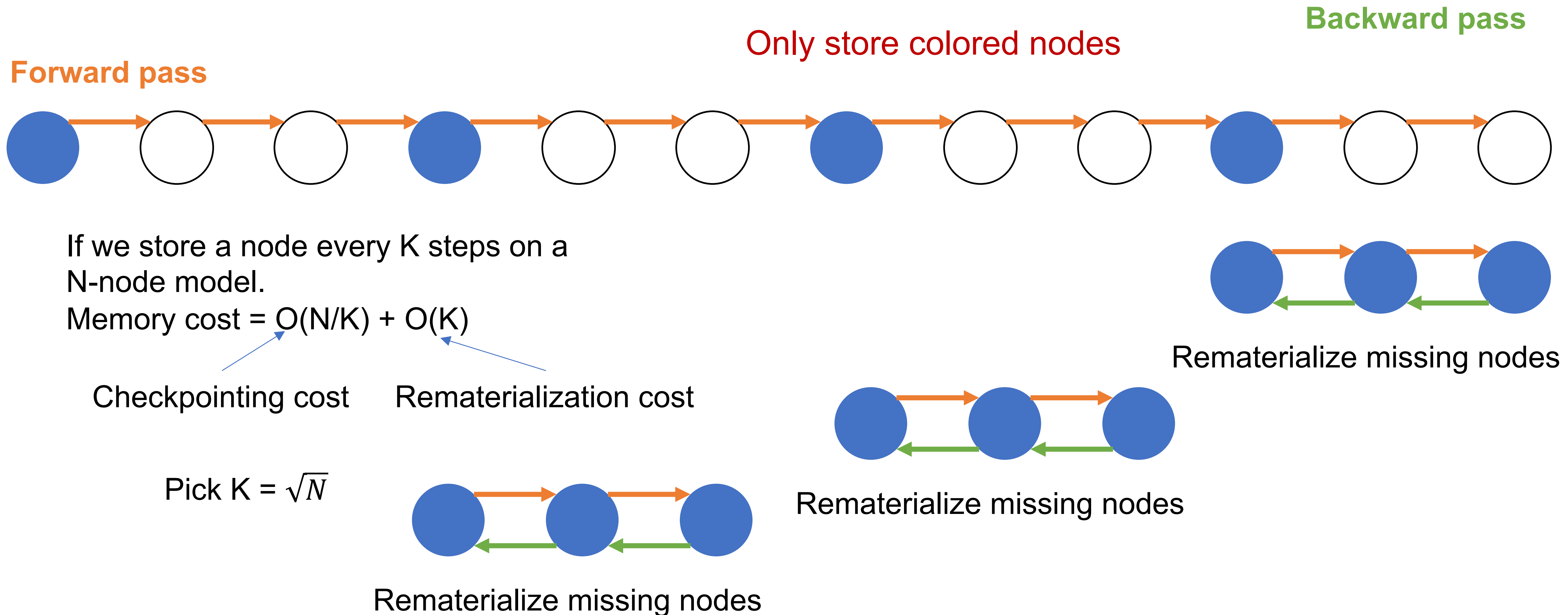
- The biggest model we can train is bounded by GPU memory
- Larger models often achieve better predictive performance
- Extremely critical for modern accelerators with limited on-chip memory



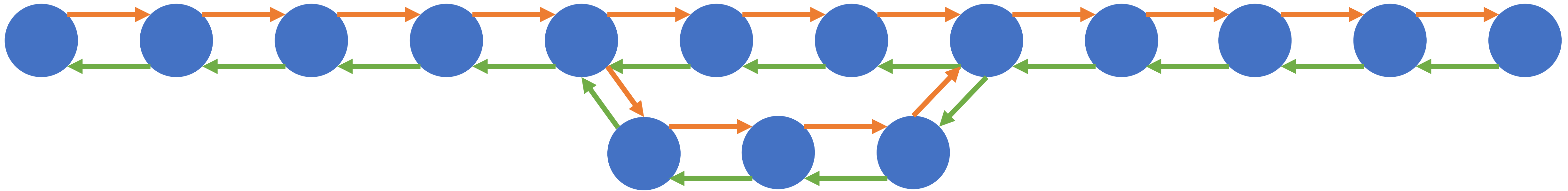
Memory Efficient Training: Tensor Rematerialization



Memory Efficient Training : Tensor Rematerialization



Memory Efficient Training : Tensor Rematerialization



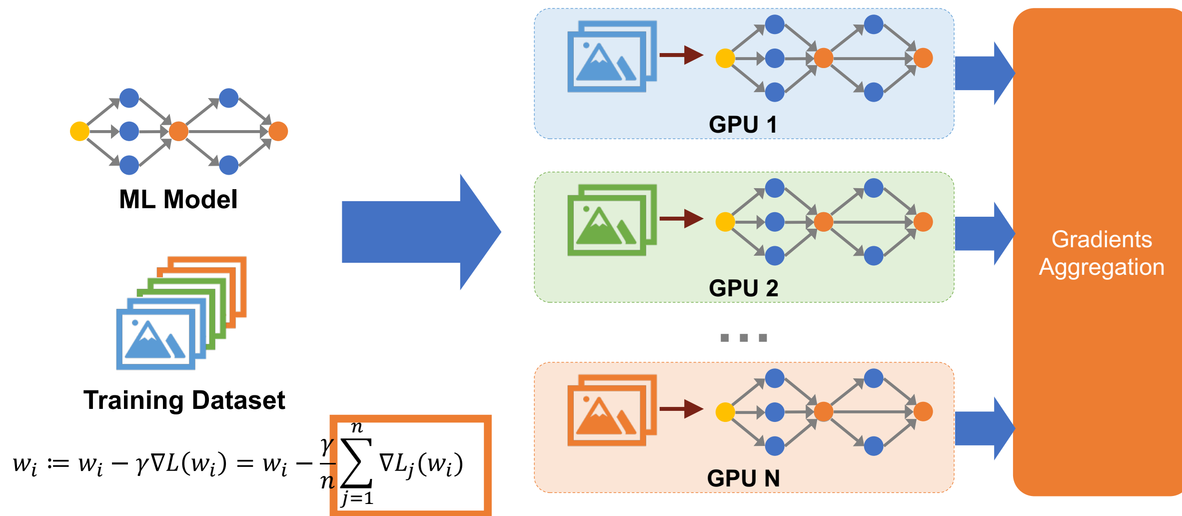
Nodes may have non-linear topology and non-uniform memory costs

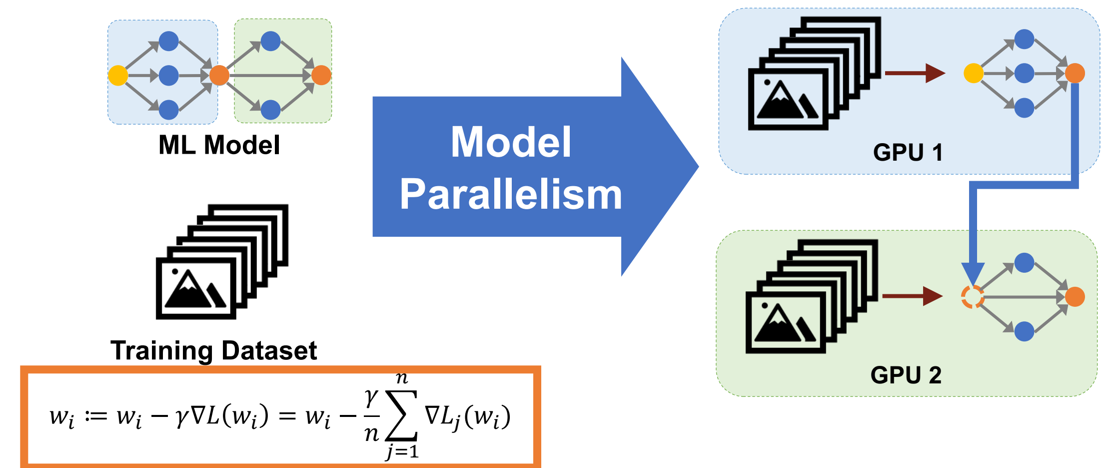
Formalize this as a mixed integer linear programming (MILP) problem and use an existing MILP solver.

We will learn this on week 7.

Memory Efficiency: Zero Redundancy

- In distributed training, data/model/pipeline parallelism all involve redundancy





Data parallelism replicates model parameters

Model/pipeline parallelism replicate intermediate tensors

Memory Efficient Training : Zero Redundancy

- Key idea: partition replicated parameters, gradients, and optimizer states across GPUs
- When needed, each GPU broadcast its local parameters/gradients to all other GPUs

Zero redundancy for data parallelism

This is achieved at the cost of extra communications!

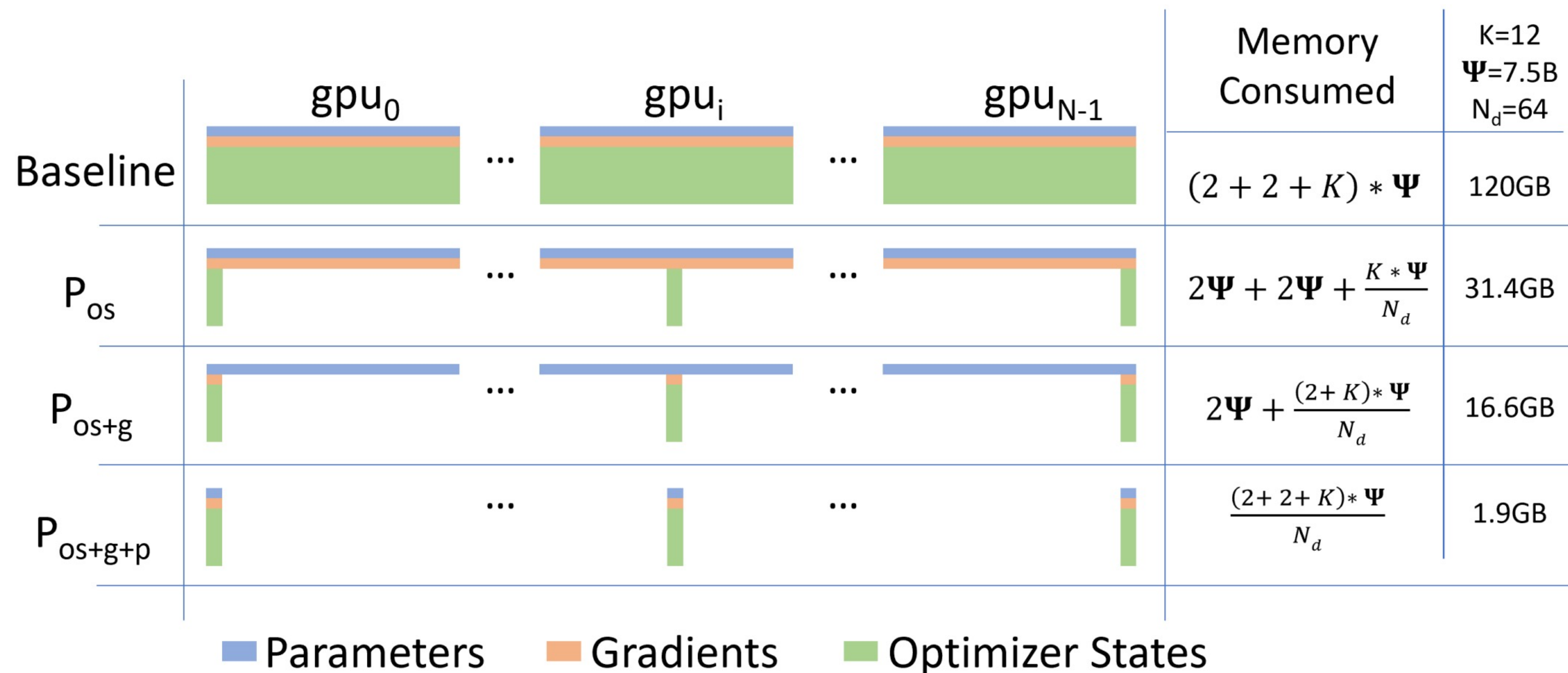


Figure from ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Balancing Computation/Memory/Communication Cost in DNN Training