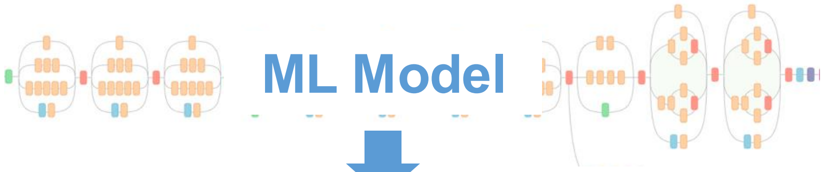
15-779 Lecture 2: ML Systems 101 (Basics + TensorFlow/PyTorch)

Zhihao Jia

Computer Science Department

Carnegie Mellon University

Recap: Machine Learning Systems



Automatic Differentiation

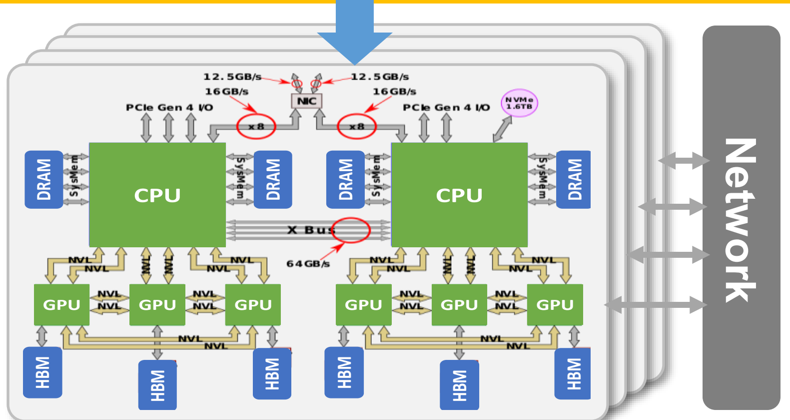
Graph-Level Optimization

Parallelization / Distributed Training

ML Compilation

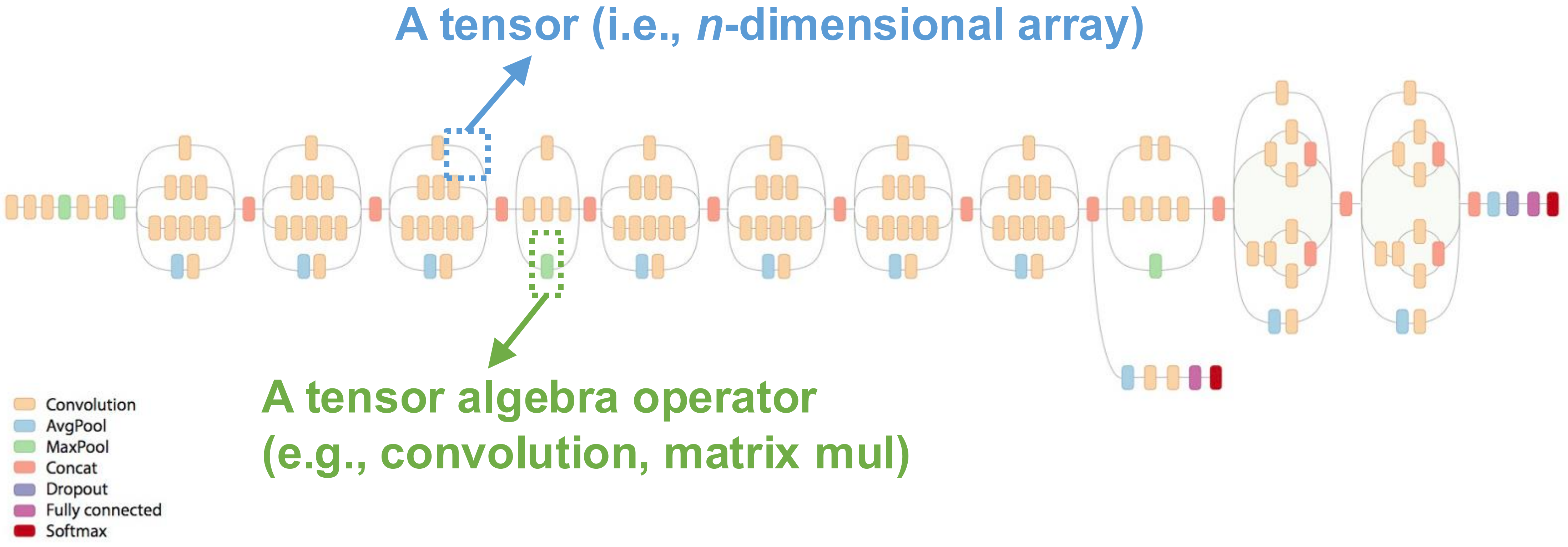
Memory Management

GPU Programming

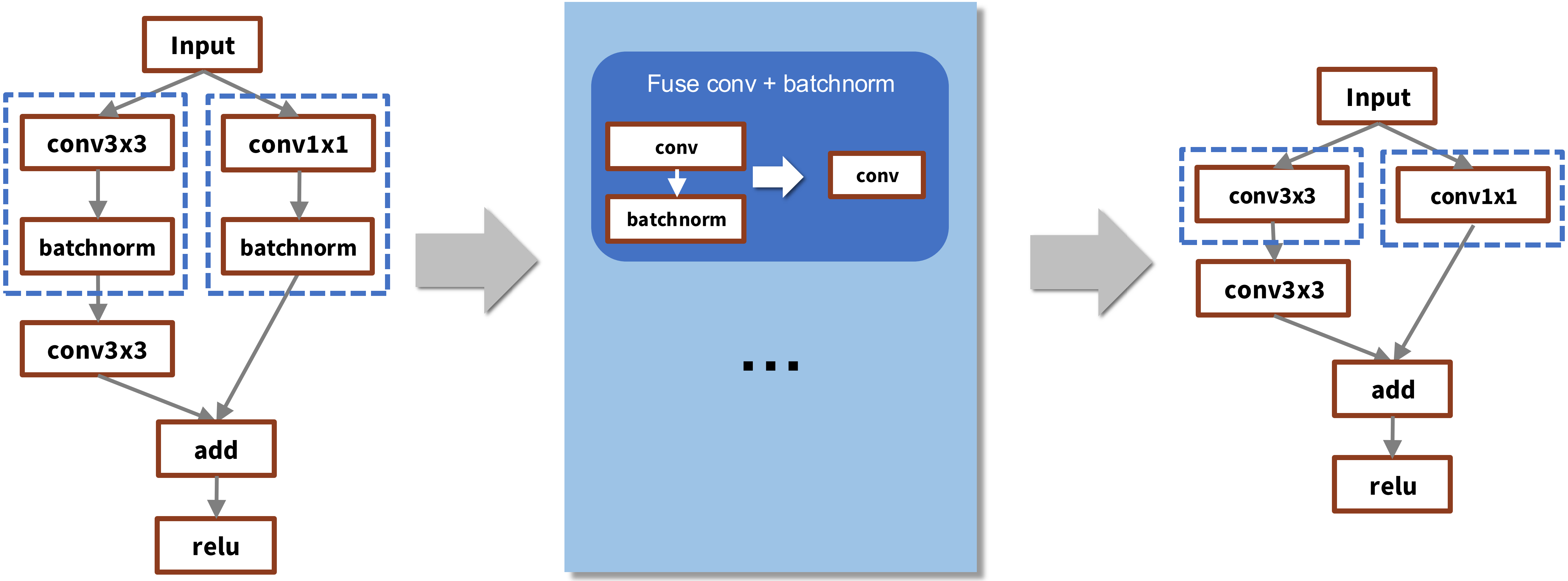


Deep Neural Network

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



Graph-Level Optimizations

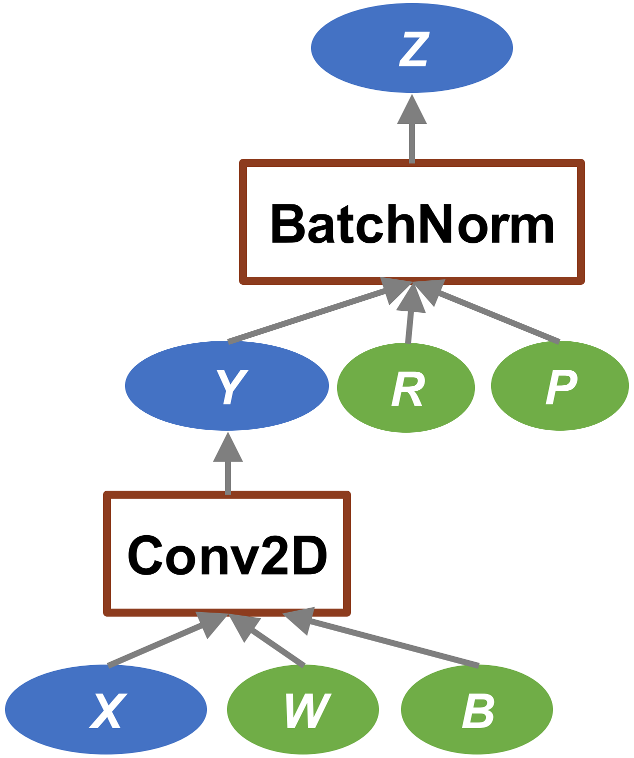


Input Computation Graph

Potential graph transformations

Optimized Computation Graph

Example: Fusing Conv and Batch Normalization

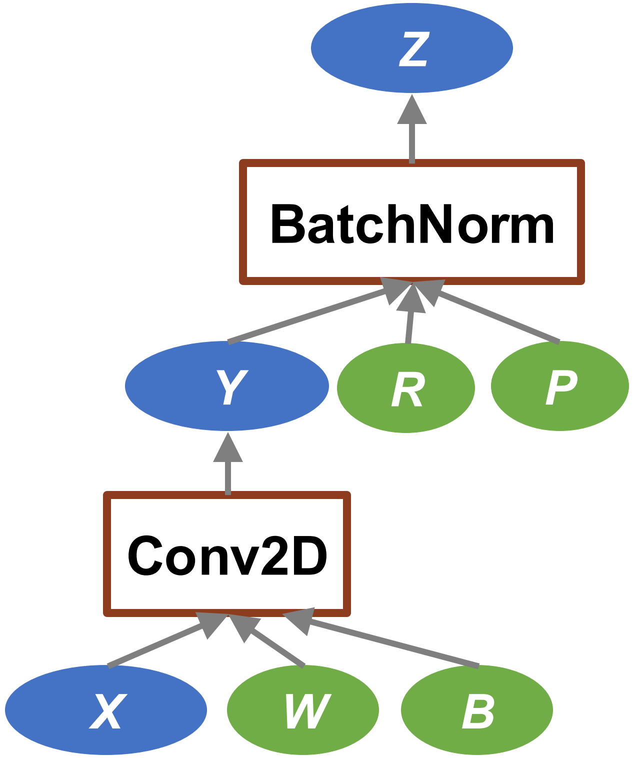


$$\boldsymbol{Z(n,c,h,w) = Y(n,c,h,w) * R(c) + P(c)}$$

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

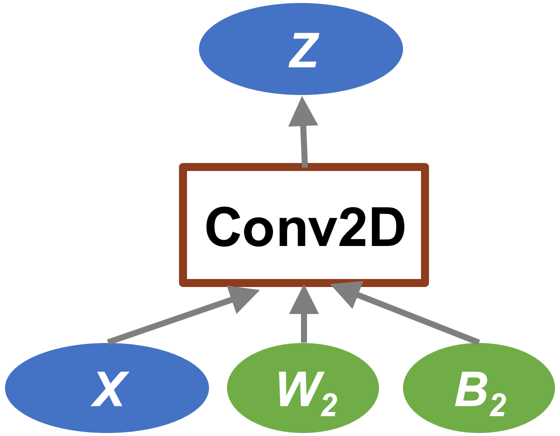
W, B, R, P are constant pre-trained weights

Fusing Conv and BatchNorm



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

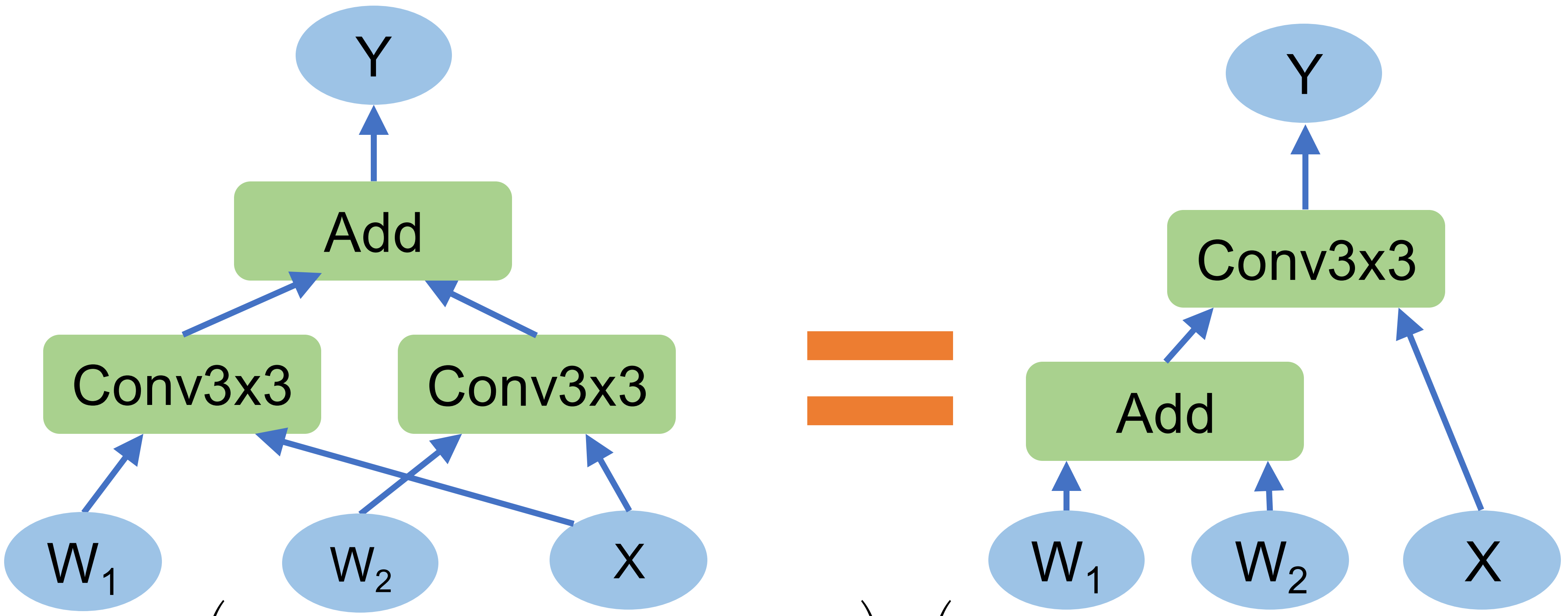
=



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

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

Fusing Two Convs

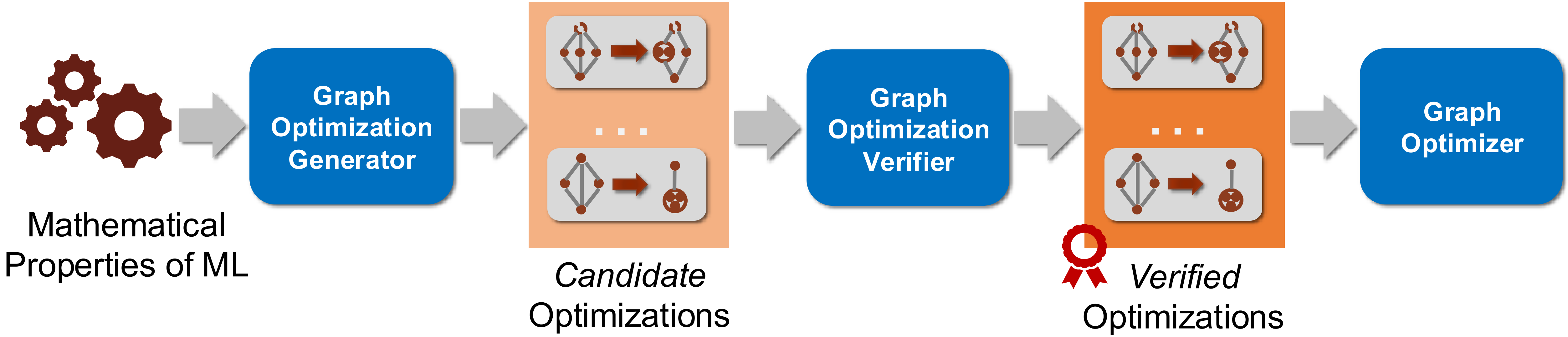


$$Y(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W_1(c,d,u,v)\right) + \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W_2(c,d,u,v)\right)$$

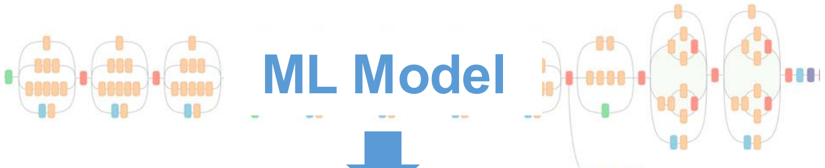
$$\Leftrightarrow Y(n,c,h,w) = \sum_{d,u,v} X(n,d,h+u,w+v) * \big((W_1(c,d,u,v) + W_2(c,d,u,v)\big)$$

Automated Discovery of Graph Optimizations

- Week 5: Automate Graph-Level Optimizations
- Week 6: Multi-Level Superoptimization



An Overview of Deep Learning Systems



Automatic Differentiation

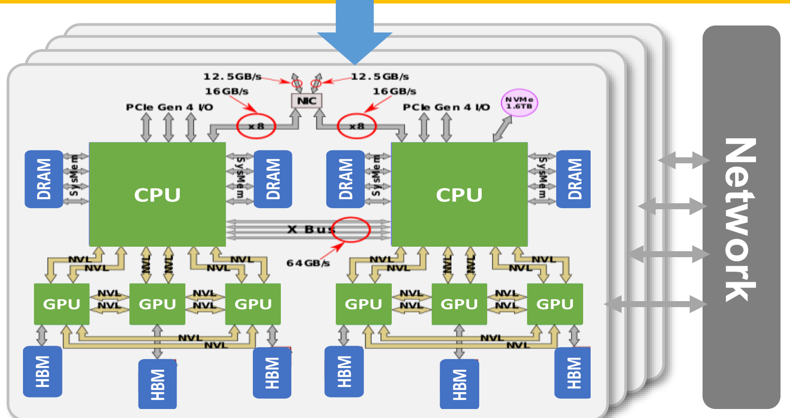
Graph-Level Optimization

Parallelization / Distributed Training

Data Layout and Placement

Kernel Optimizations

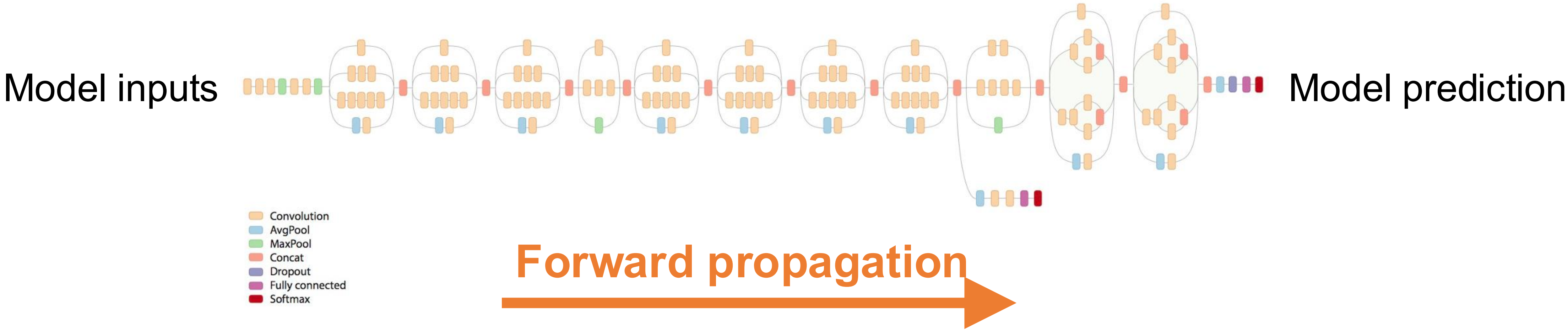
Memory Optimizations



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

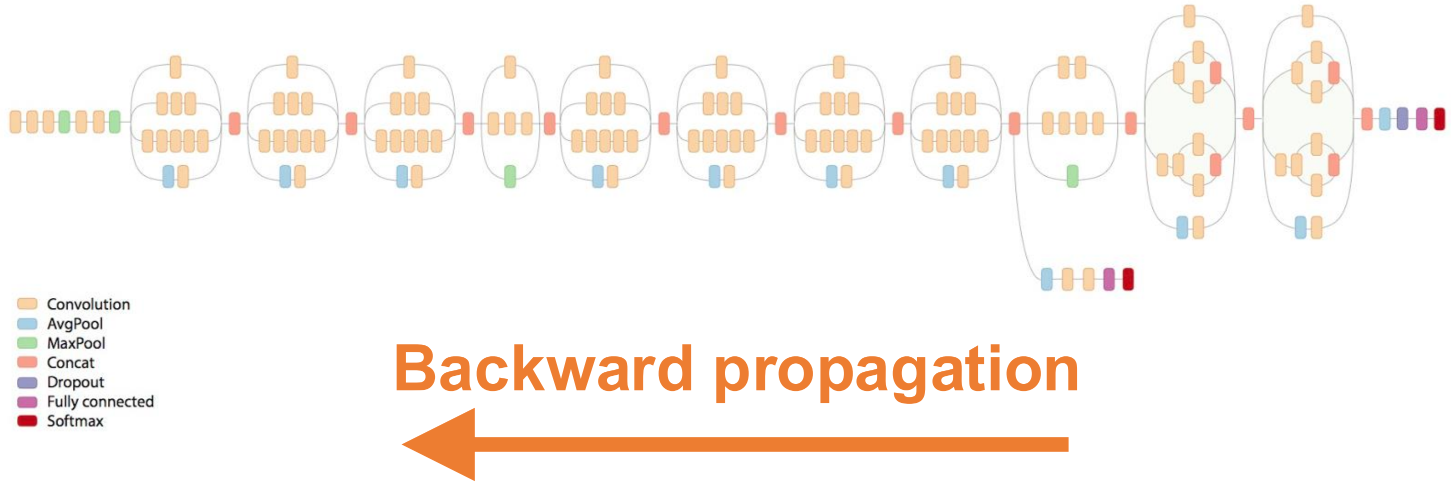


Recap: Stochastic Gradient Descent (SGD)

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

Model inputs



Model prediction

Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

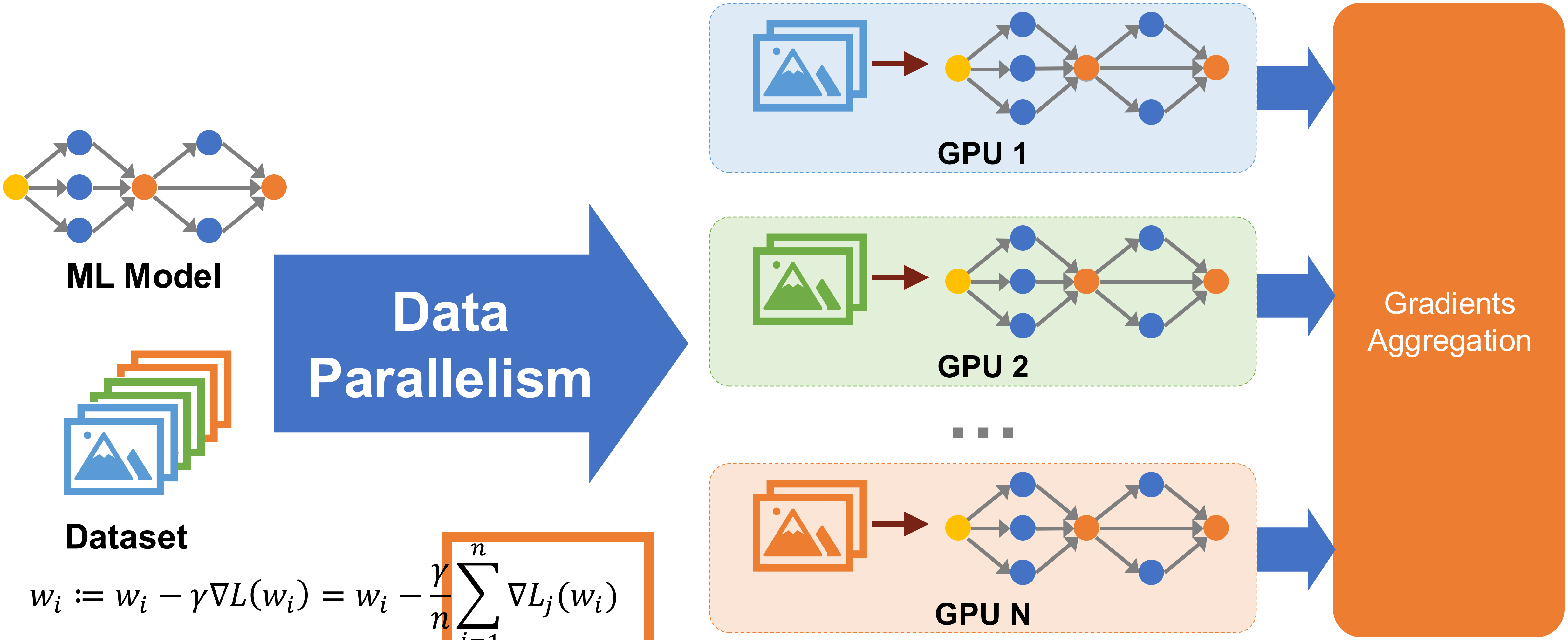
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

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

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



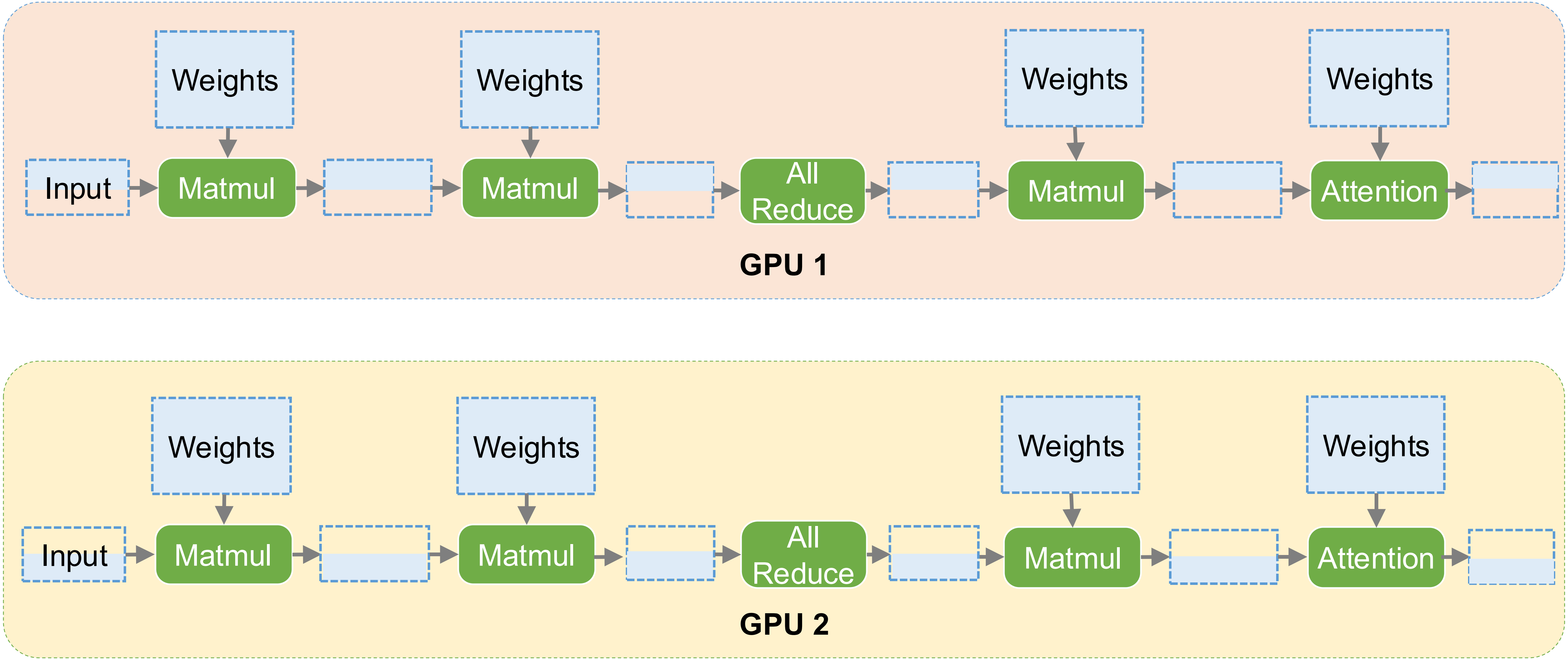
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\boxed{\sum_{j=1}^{n} \nabla L_j(w_i)}$$

1. Partition dataset into batches

2. Forward/backward of each batch on a GPU

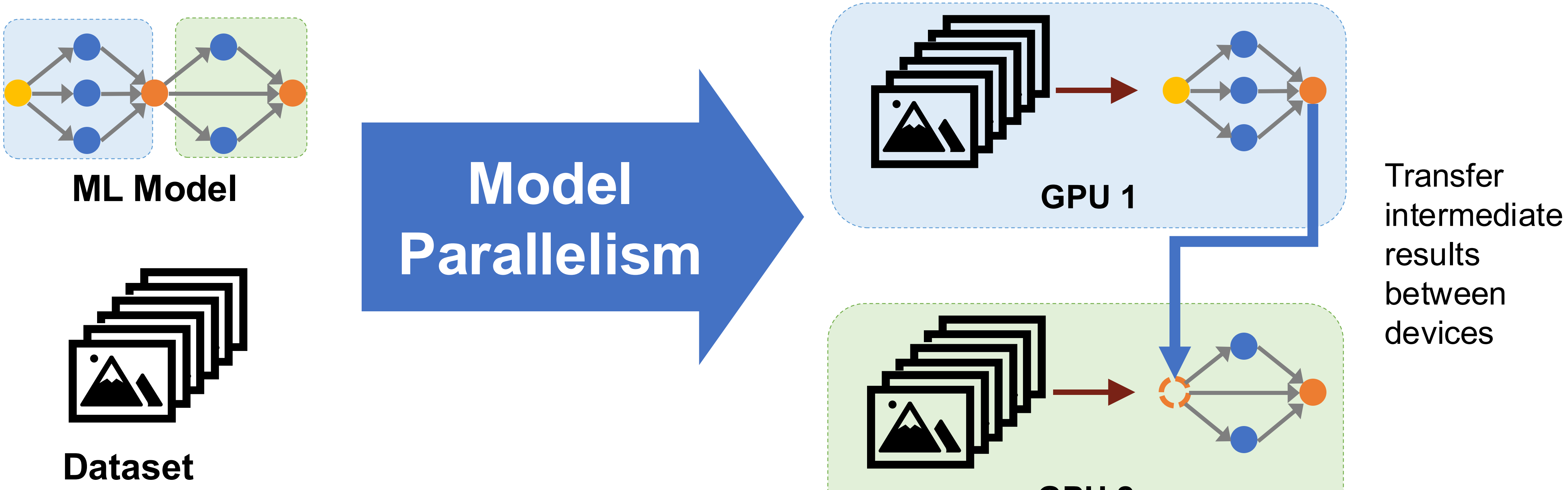
3. Aggregate gradients across GPUs

Data Parallelism for Transformer



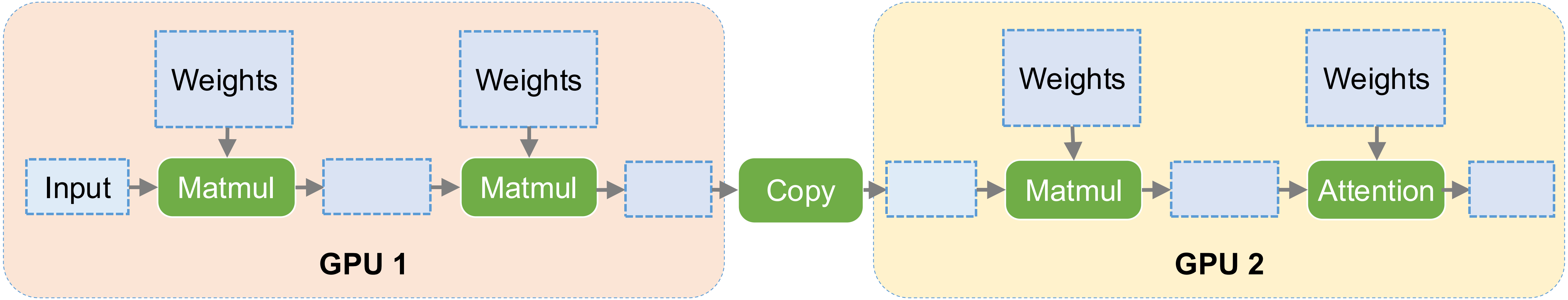
Model Parallelism

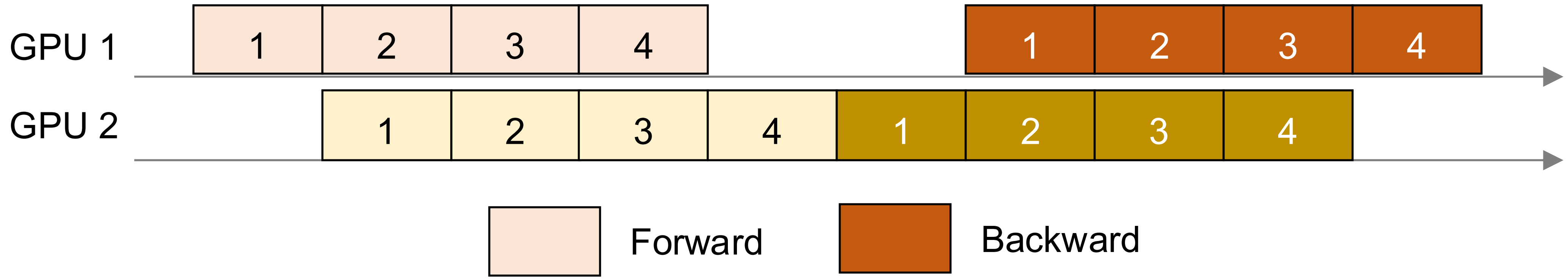
- Split a model into multiple subgraphs and assign them to different devices



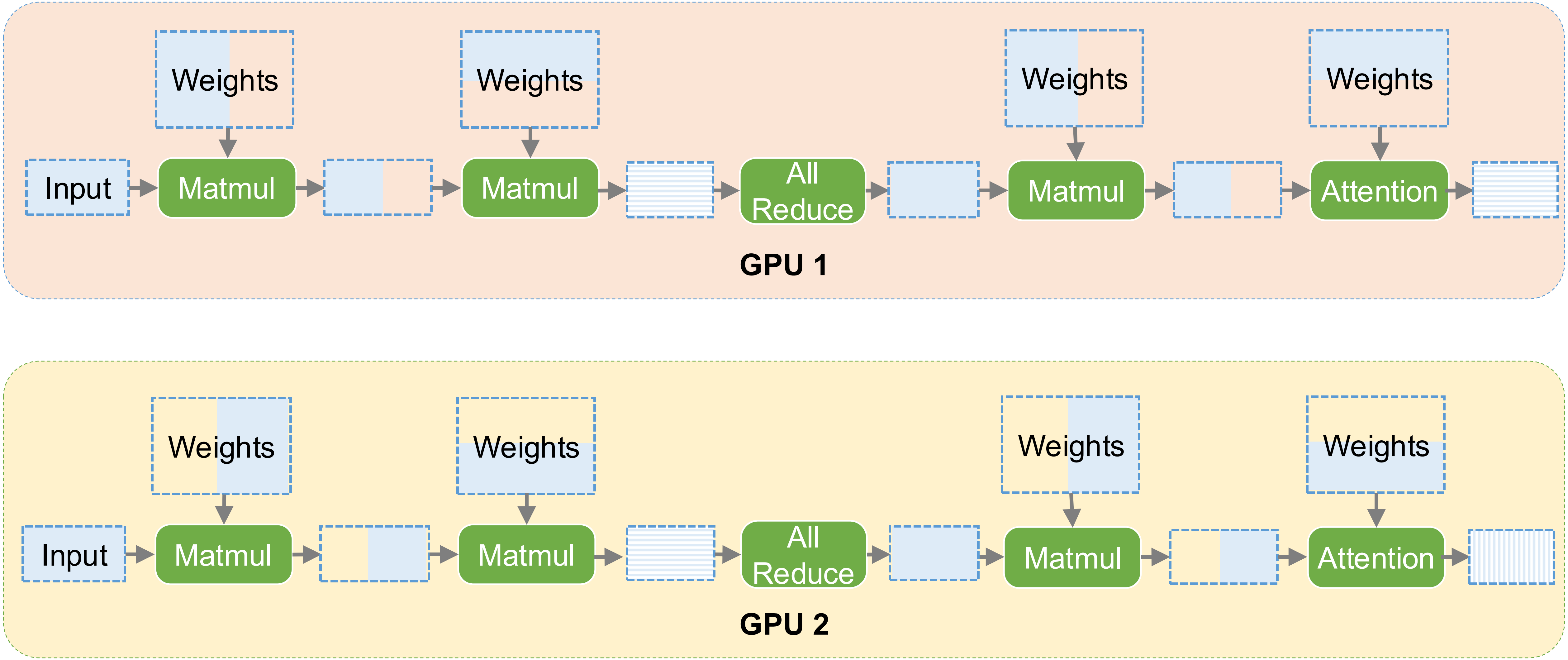
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

Pipeline Model Parallelism for Transformer

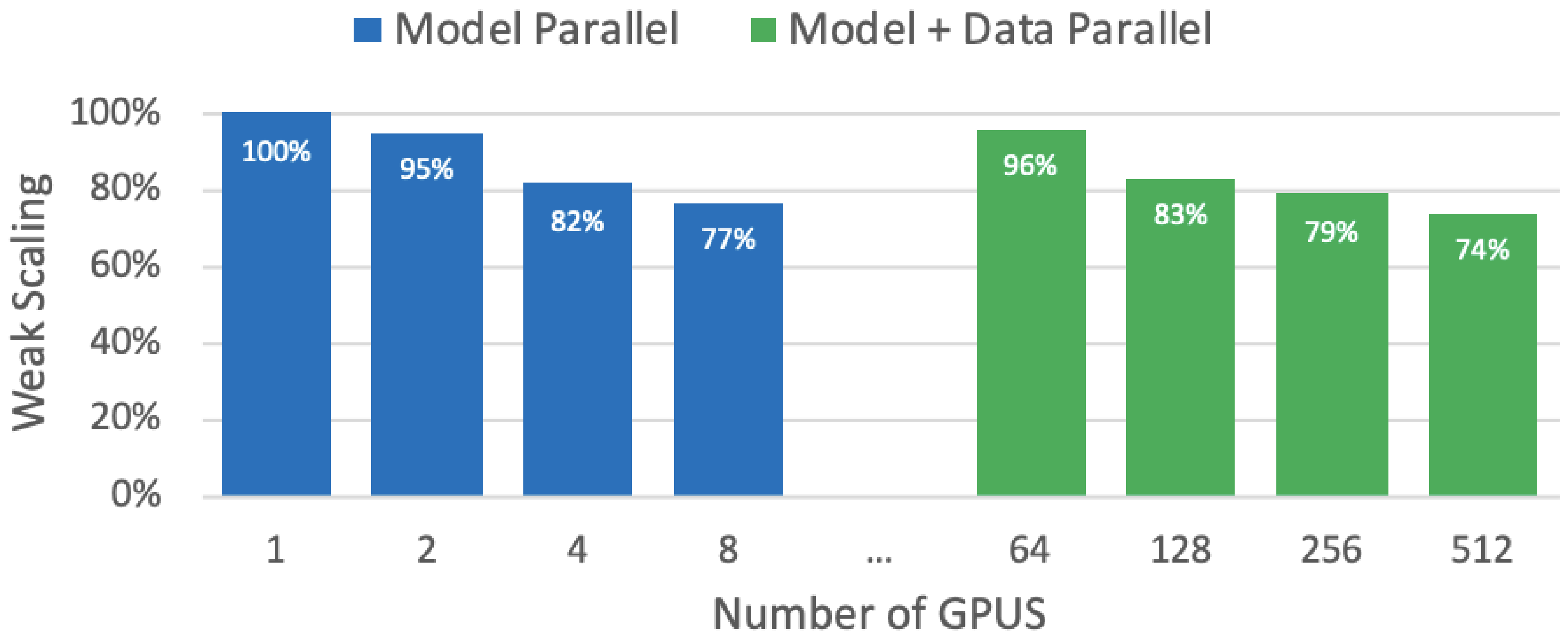




Tensor Model Parallelism for Transformer



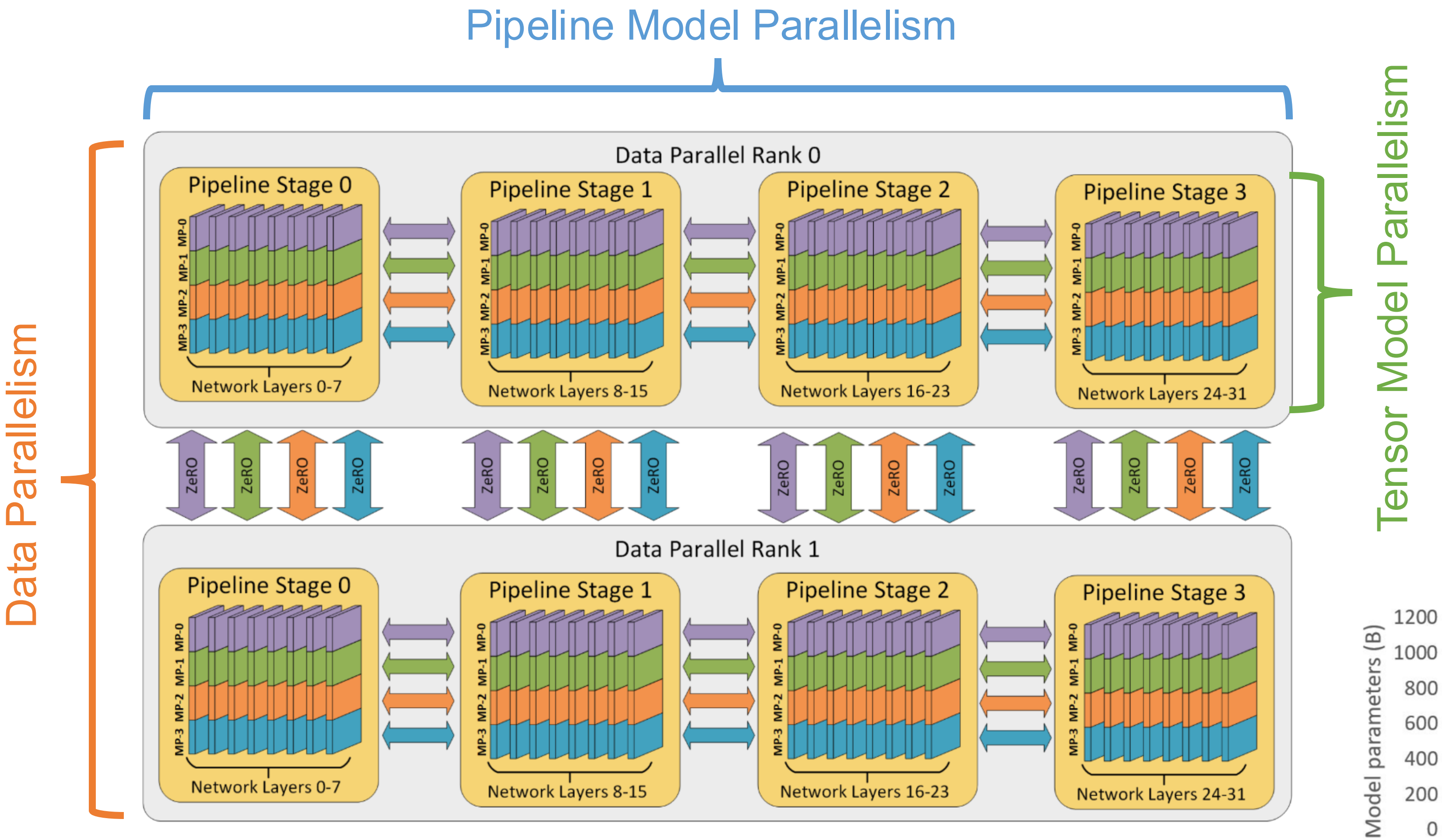
Important to Combine Different Parallelization Strategies

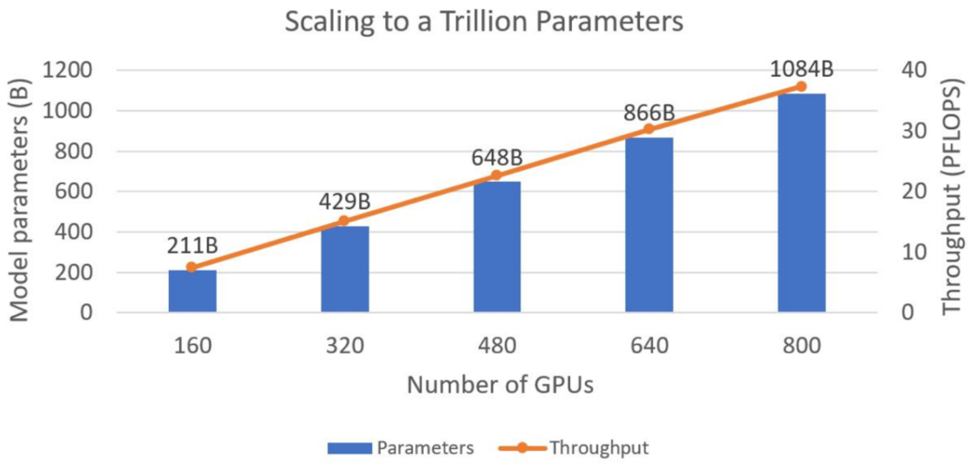


Scale to 512 GPUs by combining data and model parallelism

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.

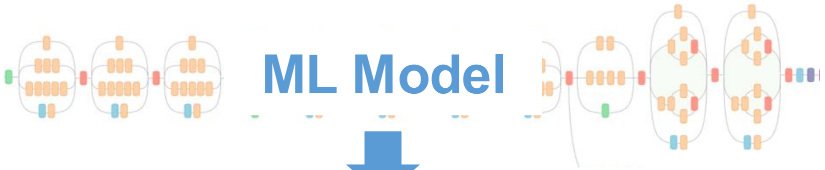
3D Parallelism for LLM Training





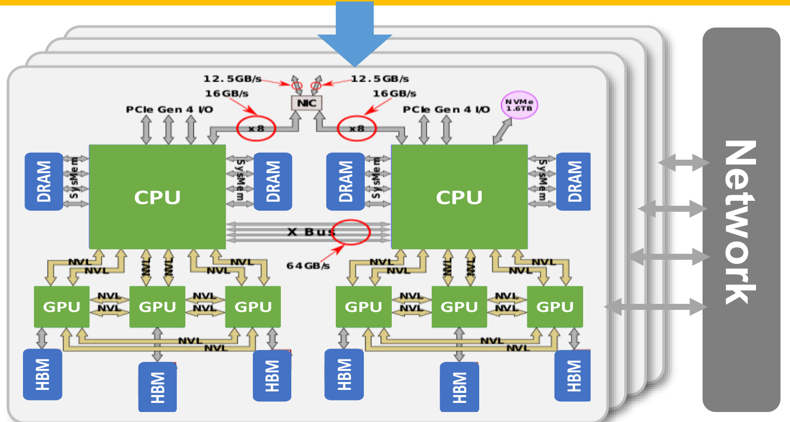
https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/

An Overview of Machine Learning Systems

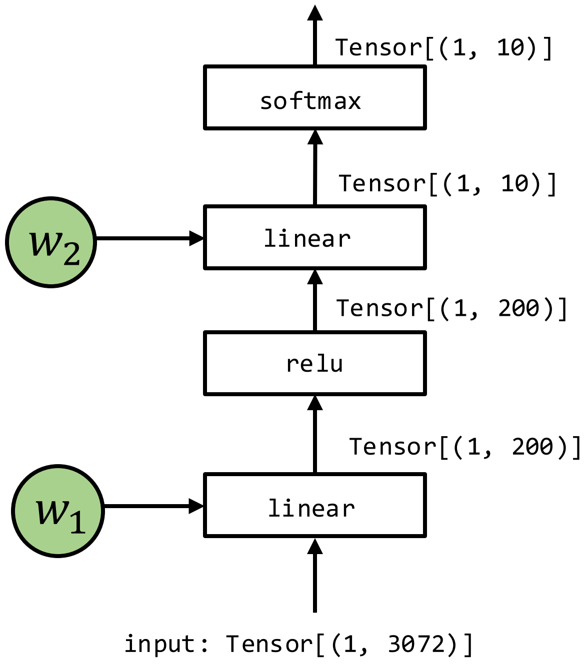


ML Model

- Algorithmic Optimization
- Graph-Level Optimization
- Parallelization / Distributed Training
- ML Compilation
- Memory Management
- GPU Programming



Key Elements in Machine Learning Compilation



Tensor multi-dimensional array that stores the input, output and intermediate results of model executions.

Tensor Functions that encodes computations among the input/output. Note that a tensor function can contain multiple operations

ML Compilation Goals

There are many equivalent ways to implement ML computation.

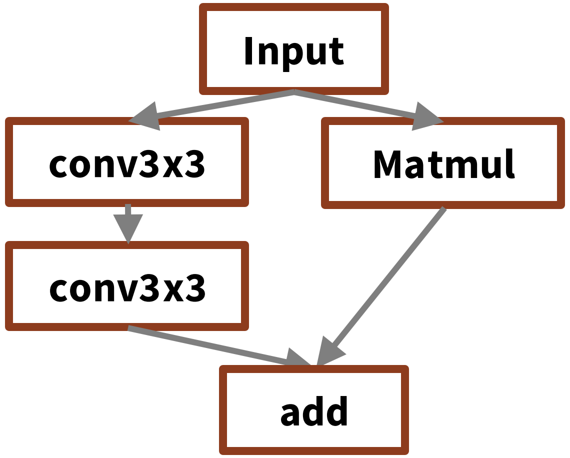
The common goals are:

Minimize memory usage.

Minimize execution time.

Maximize hardware utilization.

How to Find Fastest Program?




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

Search Space of Possible Program Optimizations

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]
```

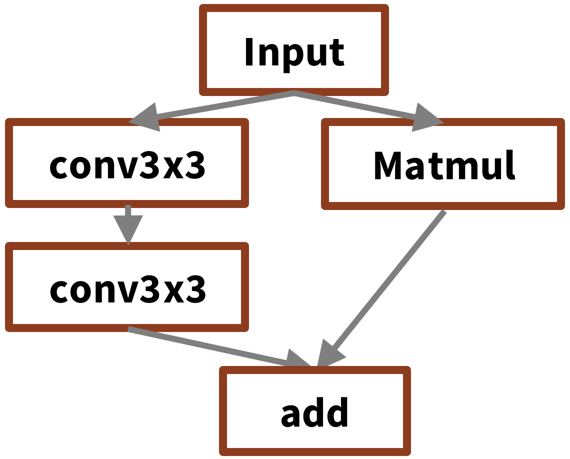
Existing Approach: Engineer Optimized Tensor Programs

- 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: TVM, Triton, Mojo, TileLang, …




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

- Loop Transformations
- Thread Bindings
- Cache Locality
- Thread Cooperation
- Tensorization
- Latency Hiding
- …

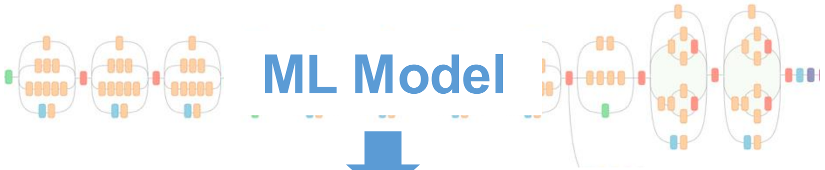
Automated search for performant programs:

✓ Immediate support for new operators

✓ Better performance than hand-written kernels

Hardware

An Overview of Machine Learning Systems



ML Model

Algorithmic Optimization

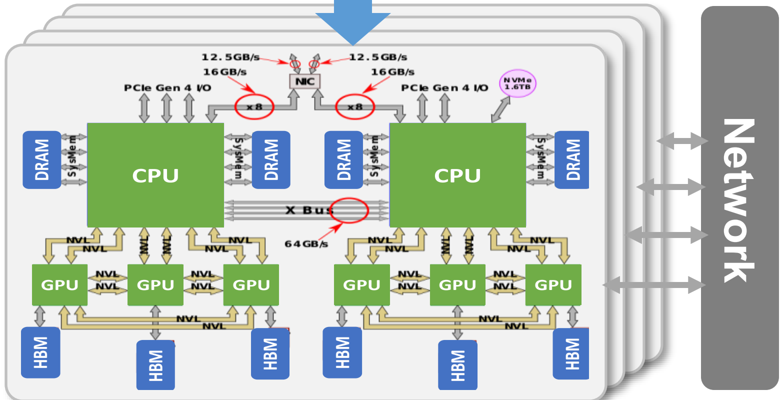
Graph-Level Optimization

Parallelization / Distributed Training

ML Compilation

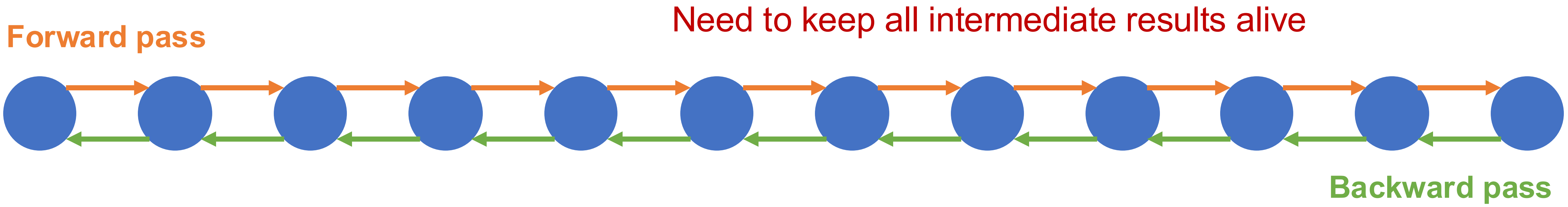
Memory Management

GPU Programming

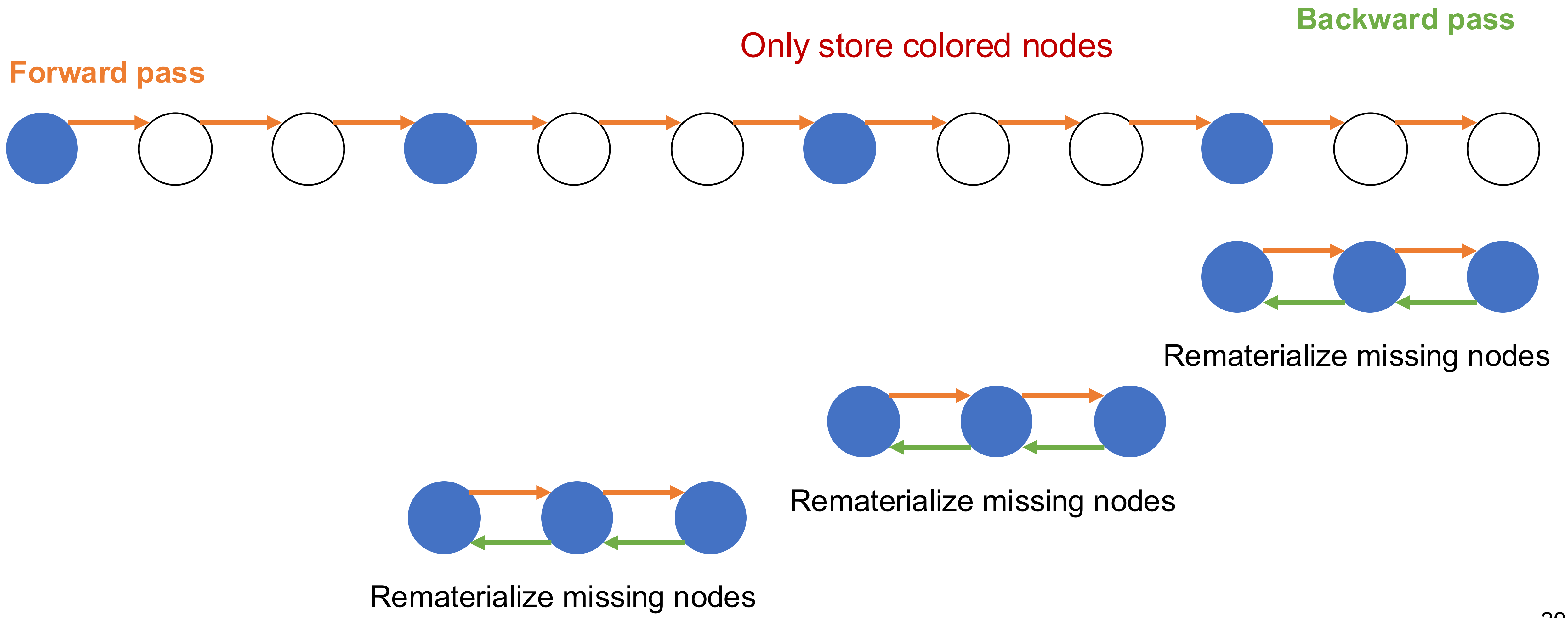


Recap: GPU Memory is the Bottleneck in DNN Training

- 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

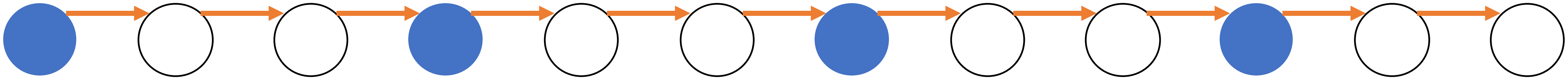


Memory Efficient Training: Tensor Rematerialization

Backward pass

Only store colored nodes

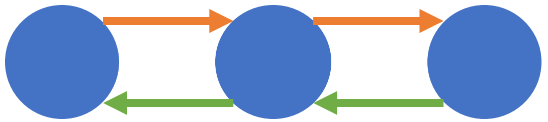
Forward pass



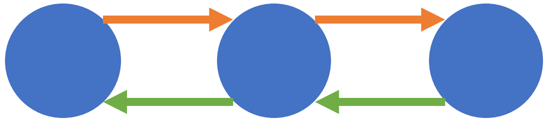
If we store a node every K steps on a N-node model.
Memory cost = O(N/K) + O(K)

Checkpointing cost   Rematerialization cost

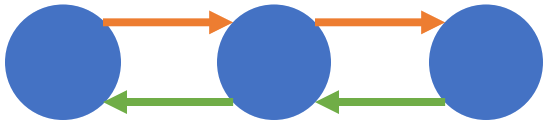
Pick K = $\sqrt{N}$



Rematerialize missing nodes



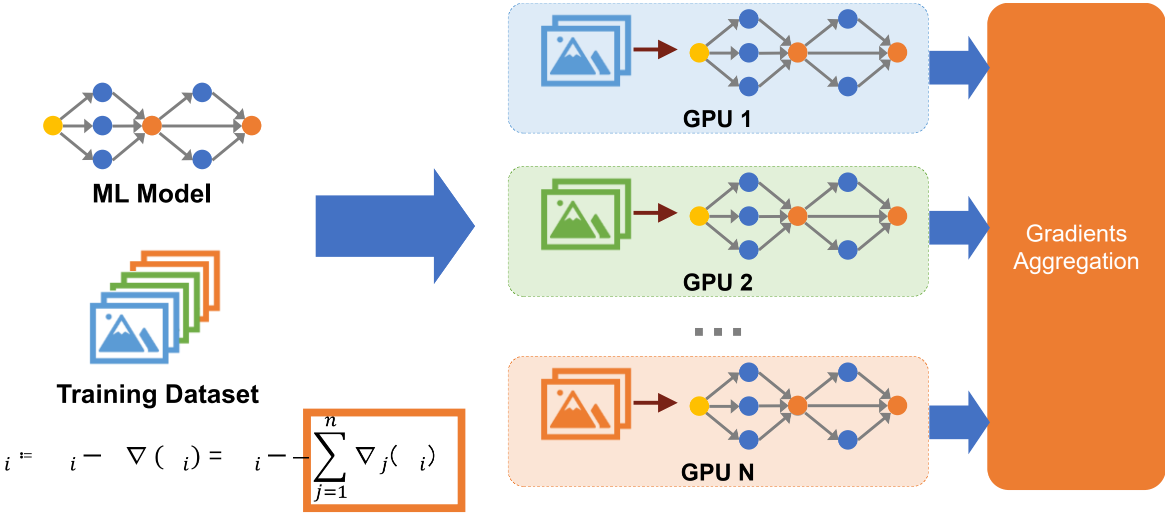
Rematerialize missing nodes



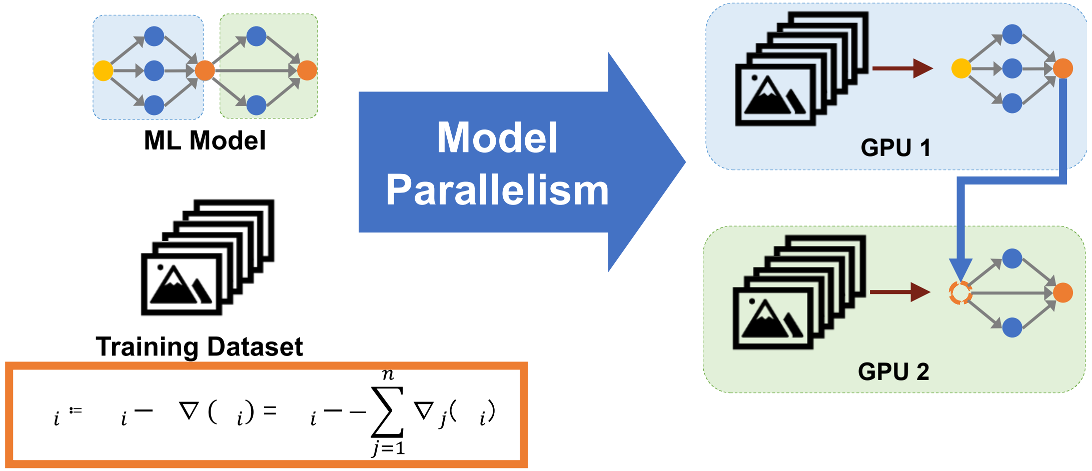
Rematerialize missing nodes

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!

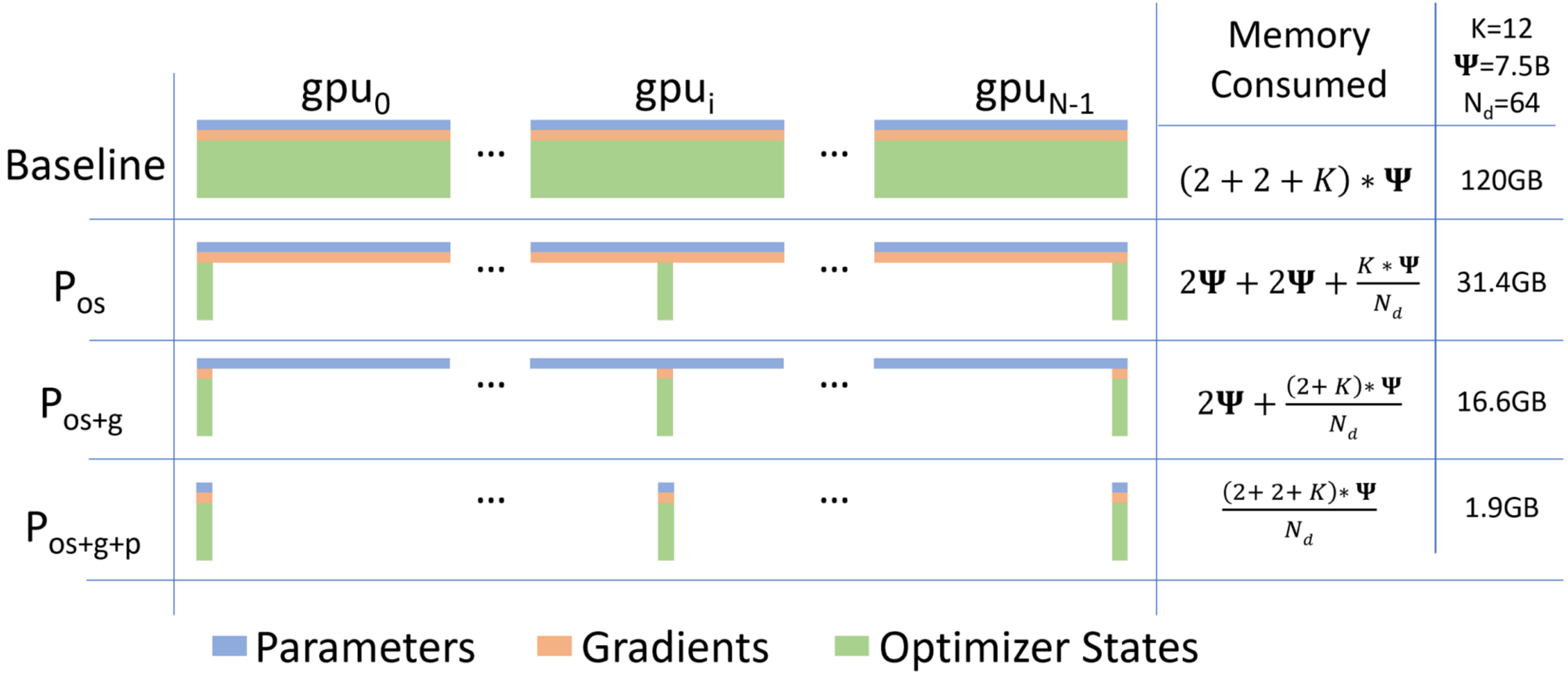
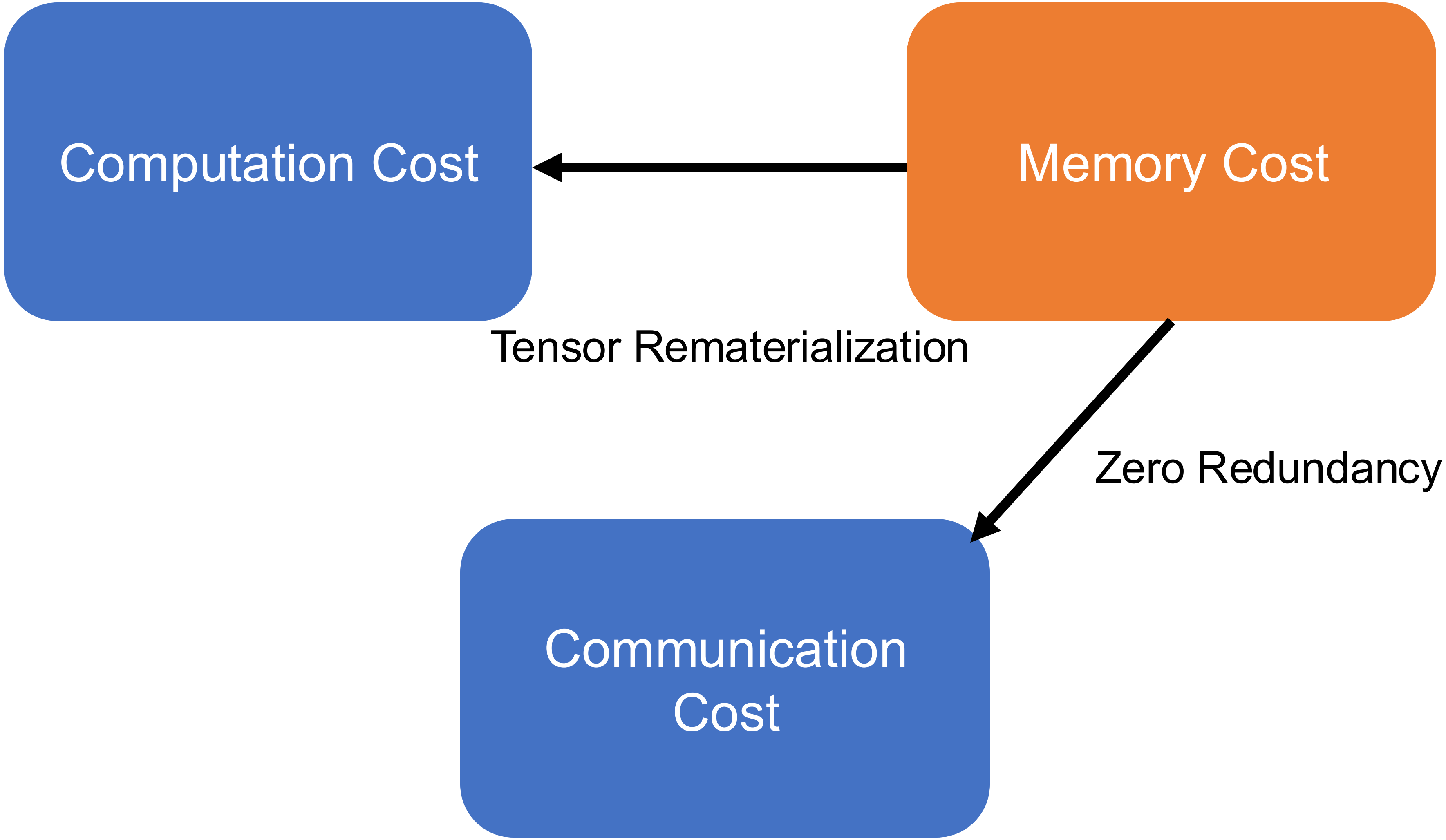


Figure from ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Balancing Computation/Memory/Communication Cost in DNN Training



Part 2. PyTorch v.s. TensorFlow

What are the key differences between them?

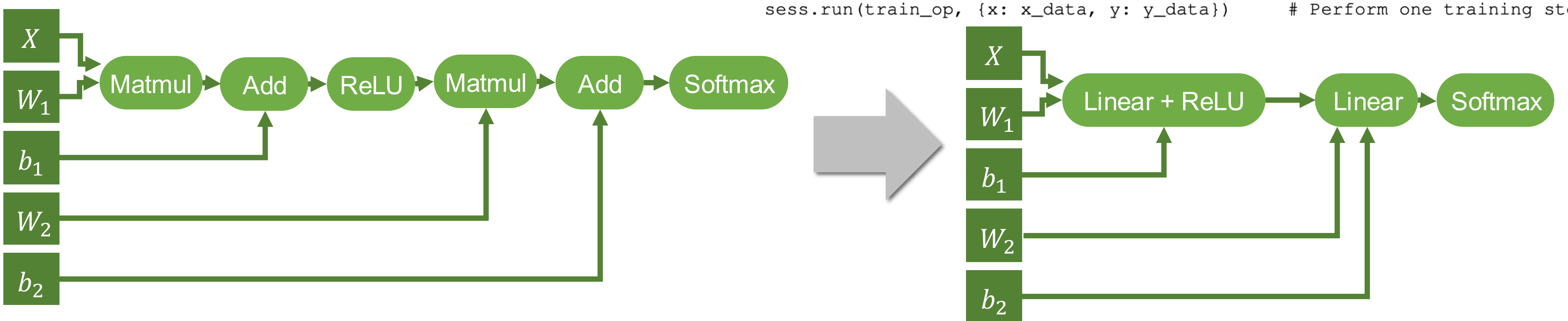
TensorFlow’s Deferred Computation Model

```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```

TensorFlow’s Deferred Computation Model

- Graph-level optimizations

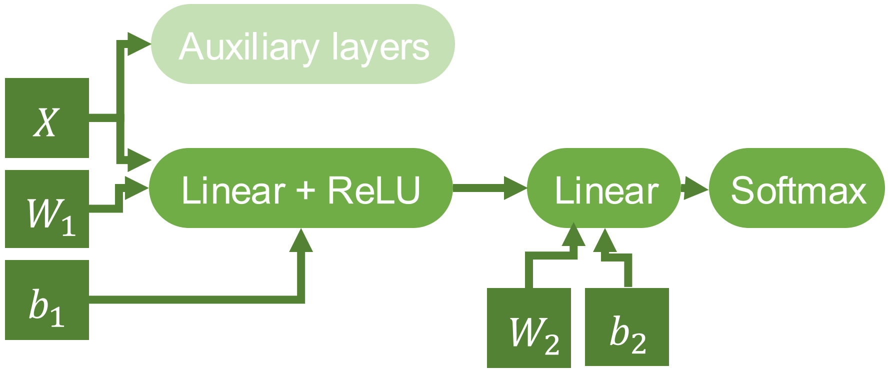
```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```



TensorFlow's Deferred Computation Model

- Graph-level optimizations
- Deferred/lazy execution

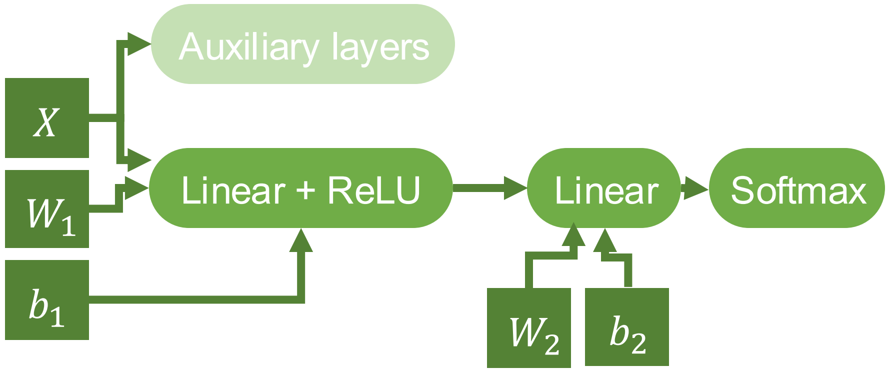
```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```



TensorFlow's Deferred Computation Model

- Graph-level optimizations
- Deferred/lazy execution
- Optimization with global information

```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```

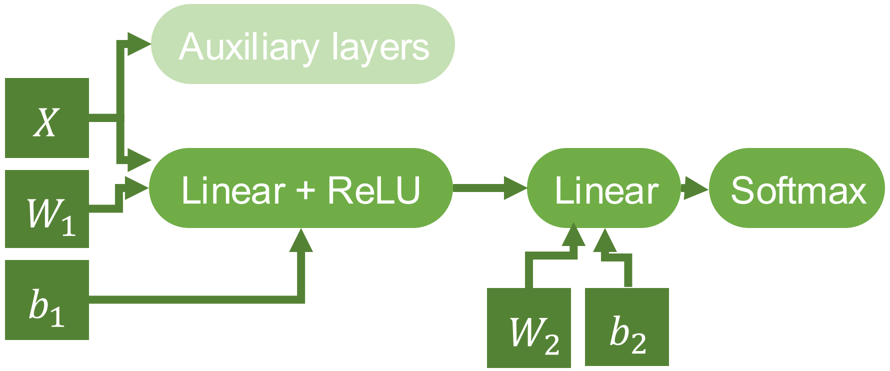


TensorFlow's Deferred Computation Model

- Graph-level optimizations
- Deferred/lazy execution
- Optimization with global information

- Hard to debug
  - Construct graphs and then run

```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```



PyTorch's Imperative, Pythonic Programming Model

```python
class LinearLayer(Module):
    def __init__(self, in_sz, out_sz):
        super().__init__()
        t1 = torch.randn(in_sz, out_sz)
        self.w = nn.Parameter(t1)
        t2 = torch.randn(out_sz)
        self.b = nn.Parameter(t2)

    def forward(self, activations):
        t = torch.mm(activations, self.w)
        return t + self.b
```

```python
class FullBasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 128, 3)
        self.fc = LinearLayer(128, 10)

    def forward(self, x):
        t1 = self.conv(x)
        t2 = nn.functional.relu(t1)
        t3 = self.fc(t1)
        return nn.functional.softmax(t3)
```

PyTorch's Imperative, Pythonic Programming Model

Be Pythonic Data scientists are familiar with the Python language, its programming model, and its tools. PyTorch should be a first-class member of that ecosystem. It follows the commonly established design goals of keeping interfaces simple and consistent, ideally with one idiomatic way of doing things. It also integrates naturally with standard plotting, debugging, and data processing tools.

Put researchers first PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible. The complexity inherent to machine learning should be handled internally by the PyTorch library and hidden behind intuitive APIs free of side-effects and unexpected performance cliffs.

Provide pragmatic performance To be useful, PyTorch needs to deliver compelling performance, although not at the expense of simplicity and ease of use. Trading 10% of speed for a significantly simpler to use model is acceptable; 100% is not. Therefore, its *implementation* accepts added complexity in order to deliver that performance. Additionally, providing tools that allow researchers to manually control the execution of their code will empower them to find their own performance improvements independent of those that the library provides automatically.

Worse is better [26] Given a fixed amount of engineering resources, and all else being equal, the time saved by keeping the internal implementation of PyTorch simple can be used to implement additional features, adapt to new situations, and keep up with the fast pace of progress in the field of AI. Therefore it is better to have a simple but slightly incomplete solution than a comprehensive but complex and hard to maintain design.

PyTorch’s Imperative, Pythonic Programming Model

- Optimized for productivity instead of performance
- Easy to prototype and debug

PyTorch's Imperative, Pythonic Programming Model

- Optimized for productivity instead of performance
- Easy to prototype and debug

- Miss optimizations due to no global information

```python
class LinearLayer(Module):
    def __init__(self, in_sz, out_sz):
        super().__init__()
        t1 = torch.randn(in_sz, out_sz)
        self.w = nn.Parameter(t1)
        t2 = torch.randn(out_sz)
        self.b = nn.Parameter(t2)

    def forward(self, activations):
        t = torch.mm(activations, self.w)
        return t + self.b
```

```python
class FullBasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 128, 3)
        self.fc = LinearLayer(128, 10)

    def forward(self, x):
        t1 = self.conv(x)
        t2 = nn.functional.relu(t1)
        t3 = self.fc(t1)
        return nn.functional.relu(t3)
```

Cannot directly fuse relu and matmul

TensorFlow v.s. PyTorch

	TensorFlow	PyTorch
Execution Model	Static graph (deferred execution)	Imperative (eager execution)
Debugging	Less direct (construct graph then run)	Native Python (easy to debug)
Optimization	Graph-level, global optimizations	Low optimizations without global information
Target Users	Production engineer, large-scale ML systems	Researchers, ML engineers, rapid prototyping

PyTorch 2.0: The Best of TensorFlow and PyTorch

```
class LinearLayer(Module):
   def __init__(self, in_sz, out_sz):
      super().__init__()
      t1 = torch.randn(in_sz, out_sz)
      self.w = nn.Parameter(t1)
      t2 = torch.randn(out_sz)
      self.b = nn.Parameter(t2)

   def forward(self, activations):
      t = torch.mm(activations, self.w)
      return t + self.b
```

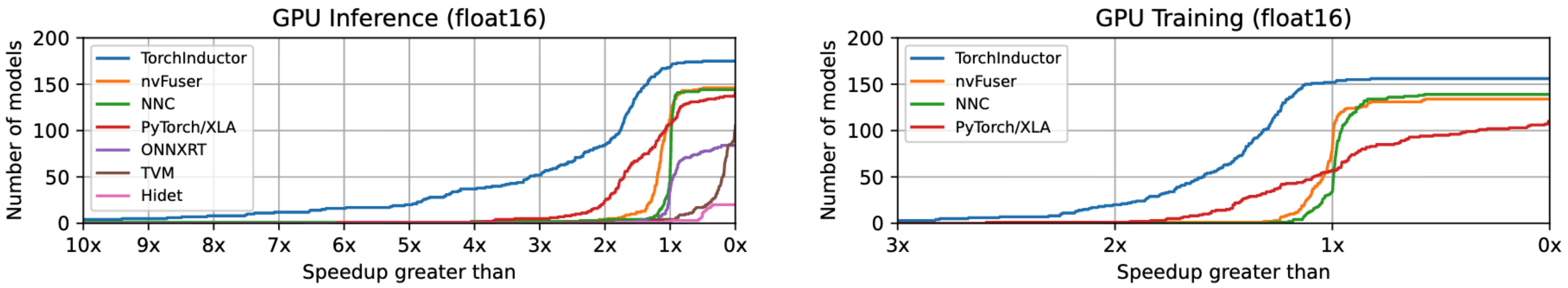
```
class FullBasicModel(nn.Module):
   def __init__(self):
      super().__init__()
      self.conv = nn.Conv2d(1, 128, 3)
      self.fc = LinearLayer(128, 10)

   def forward(self, x):
      t1 = self.conv(x)
      t2 = nn.functional.relu(t1)
      t3 = self.fc(t1)
      return nn.functional.softmax(t3)
```

`torch.compile(model)`

- Trace computation graph and compile it into optimized kernels

PyTorch 2.0: The Best of TensorFlow and PyTorch

- Capture graph while preserving imperative/eager model
- Graph-level, global optimizations
- Significant performance improvement for both training and inference



Cumulative Distribution Function (CDF) of speedups over PyTorch eager mode.