

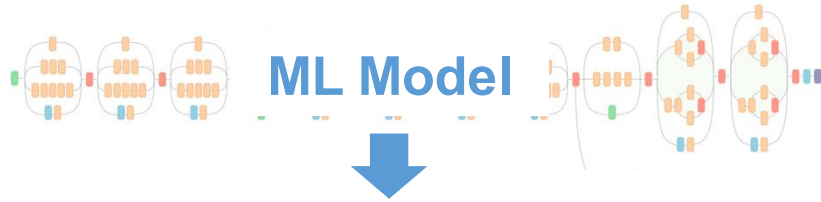
# **15-442/15-642: Machine Learning Systems**

## **Introduction to Machine Learning Systems**

Spring 2026

Tianqi Chen and Zhihao Jia  
Carnegie Mellon University

# An Overview of Machine learning Systems



Automatic Differentiation

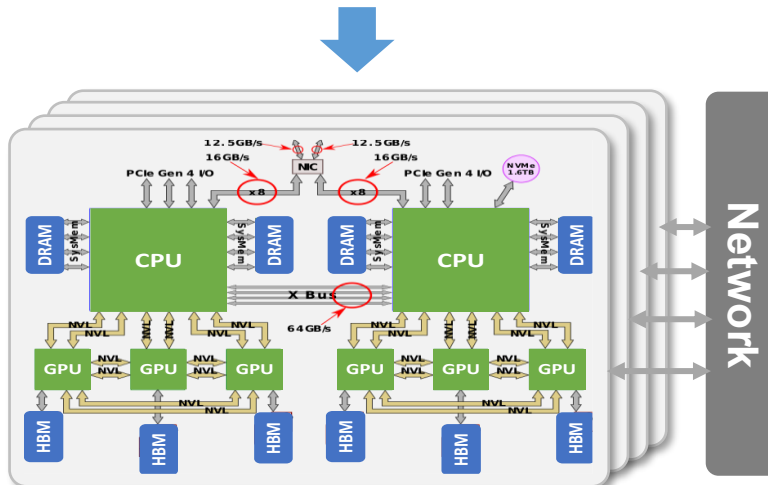
Graph-Level Optimization

Parallelization

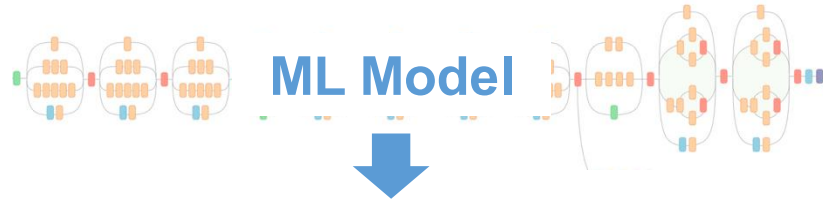
Kernel Generation

Memory Optimization

ML Systems



# Layer 1: Automatic Differentiation



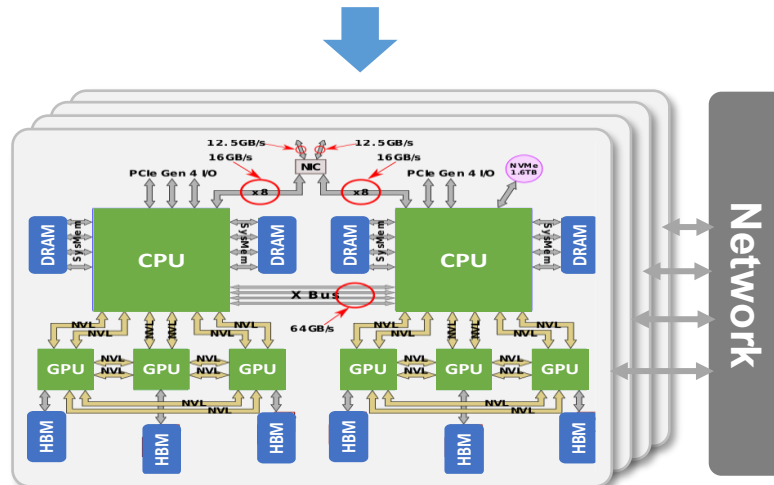
Automatic Differentiation

Graph-Level Optimization

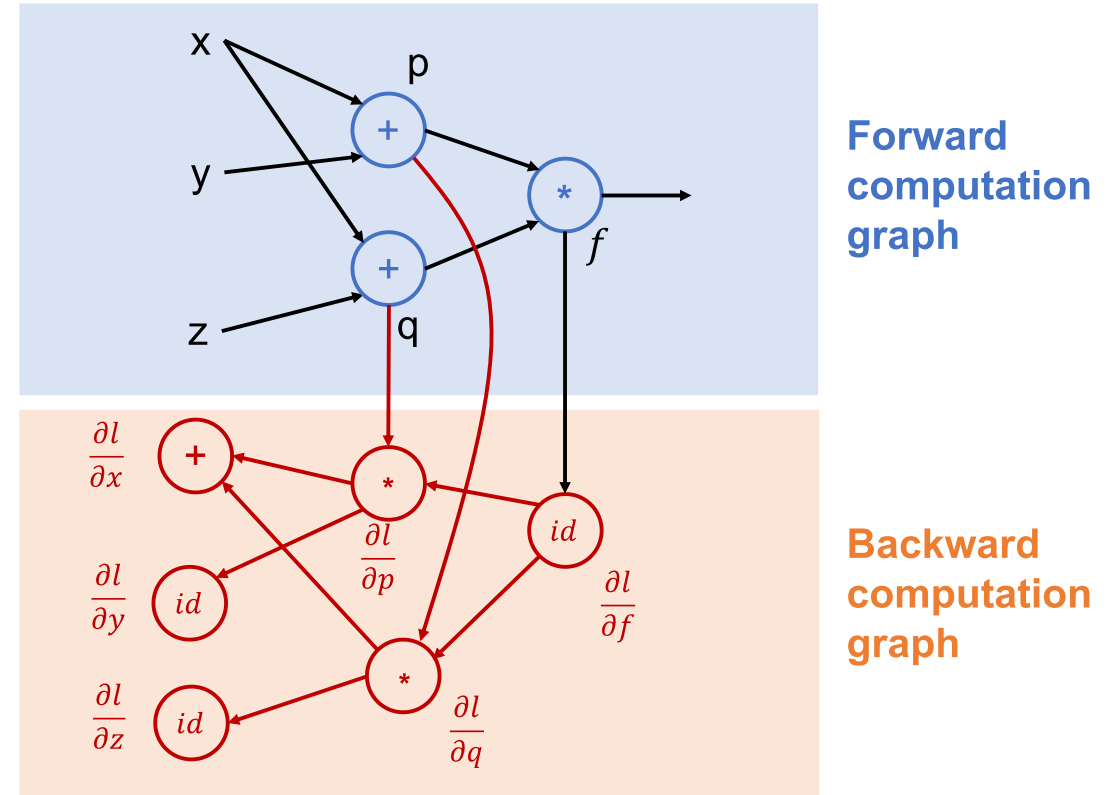
Parallelization

Kernel Generation

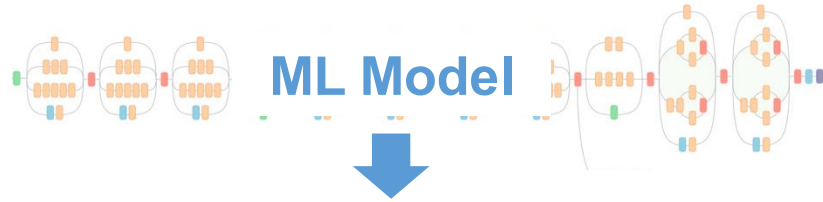
Memory Optimization



Automatically construct backward computation graph



# Layer 2: Graph-Level Optimizations



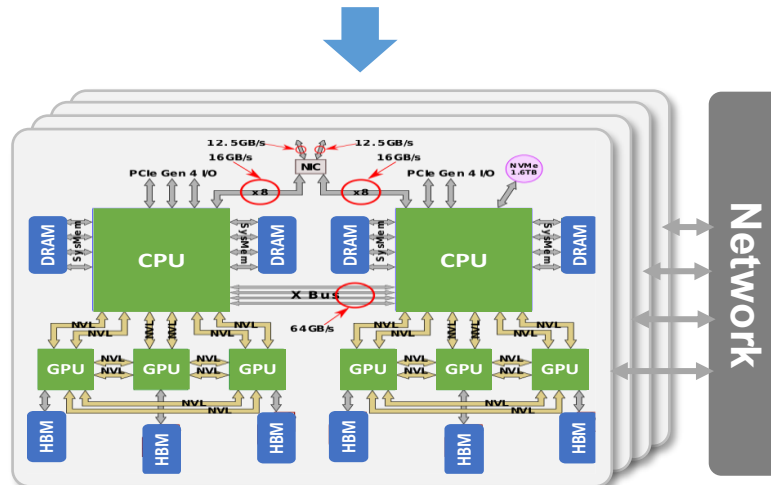
Automatic Differentiation

Graph-Level Optimization

Parallelization

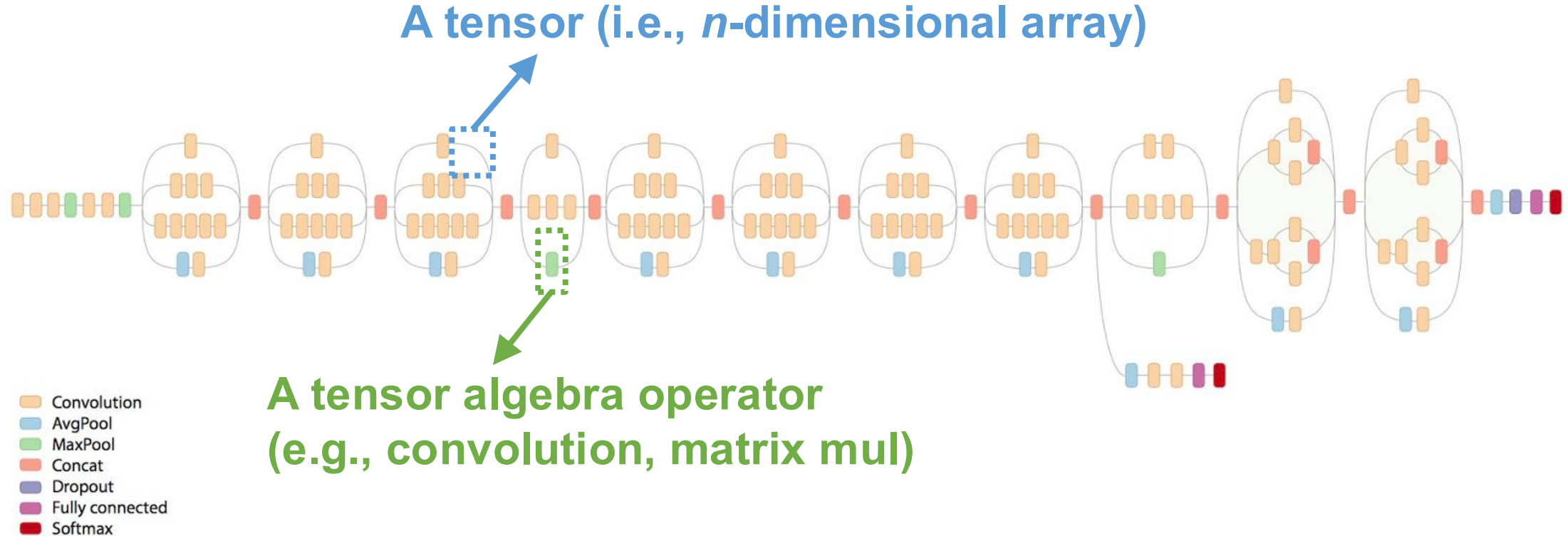
Kernel Generation

Memory Optimization

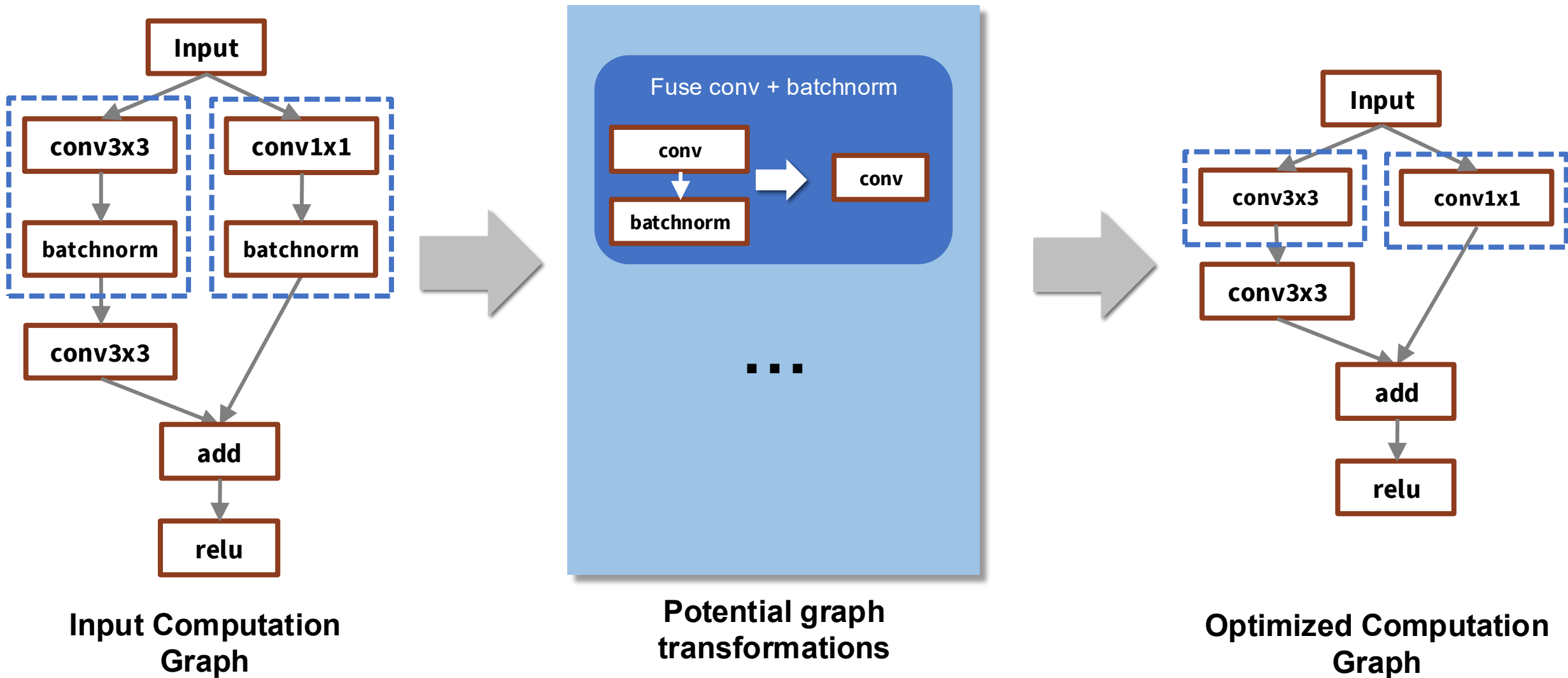


# Recap: DNNs as Computation Graphs

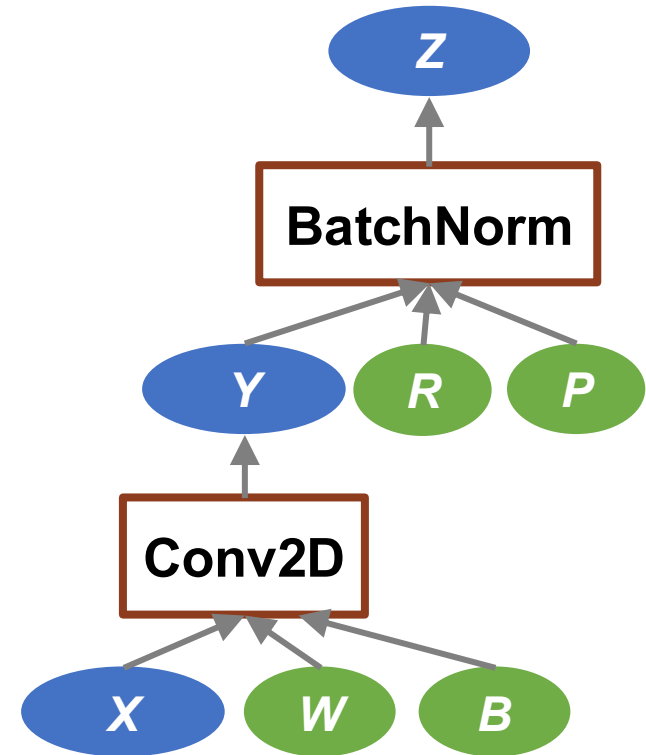
- Collection of simple trainable mathematical units that work together to solve complicated tasks



# Graph-Level Optimizations



# Example: Fusing Convolution and Batch Normalization

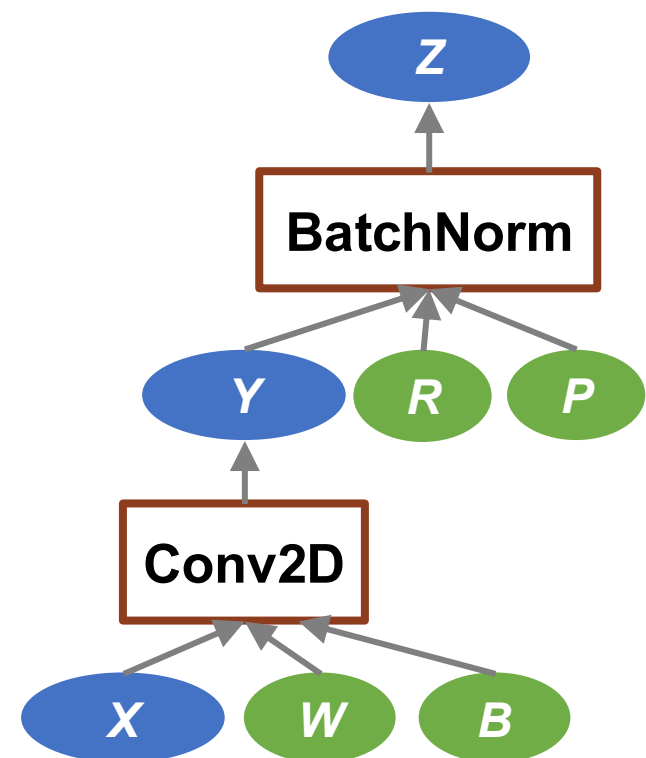


$$Z(n, c, h, w) = Y(n, c, h, w) * R(c) + P(c)$$

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

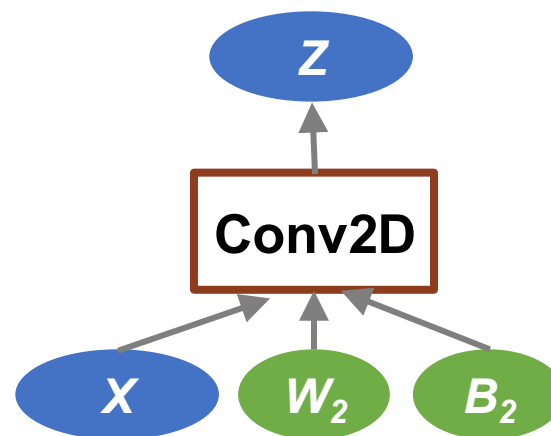
$W, B, R, P$  are constant pre-trained weights

# 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)

Fuse conv + relu

Fuse conv +  
batch normalization

Fuse multi. convs

■ ■ ■

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

# Limitations of Rule-based Optimizations

## Robustness

Experts' heuristics do not apply to all models/hardware

The screenshot shows a GitHub issue page. At the top, there are tabs for Code, Issues (250), Pull requests (11), Wiki, and Insights. The issue title is "Horovod with XLA is slower than without XLA (Tensorflow 1.12) #713" with a "New issue" button. Below the title, it says "Closed" and "LiweiPeng opened this issue on Dec 19, 2018 · 2 comments". A comment from LiweiPeng, dated Dec 19, 2018, describes a performance issue: "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?". A user named tgaddair added the "question" label on Dec 19, 2018. On the right side, there are sections for Assignees (No one assigned), Labels (question), Milestone (No milestone), and Notifications (Subscribe button).

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

The screenshot shows a Stack Overflow question page. The title is "Tensorflow XLA makes it slower?". The user asks: "I am writing a very simple tensorflow program with XLA enabled. Basically it's something like:". They provide a code snippet for a ChainSoftMax function. Below the code, they explain: "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?".

```
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})
```

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

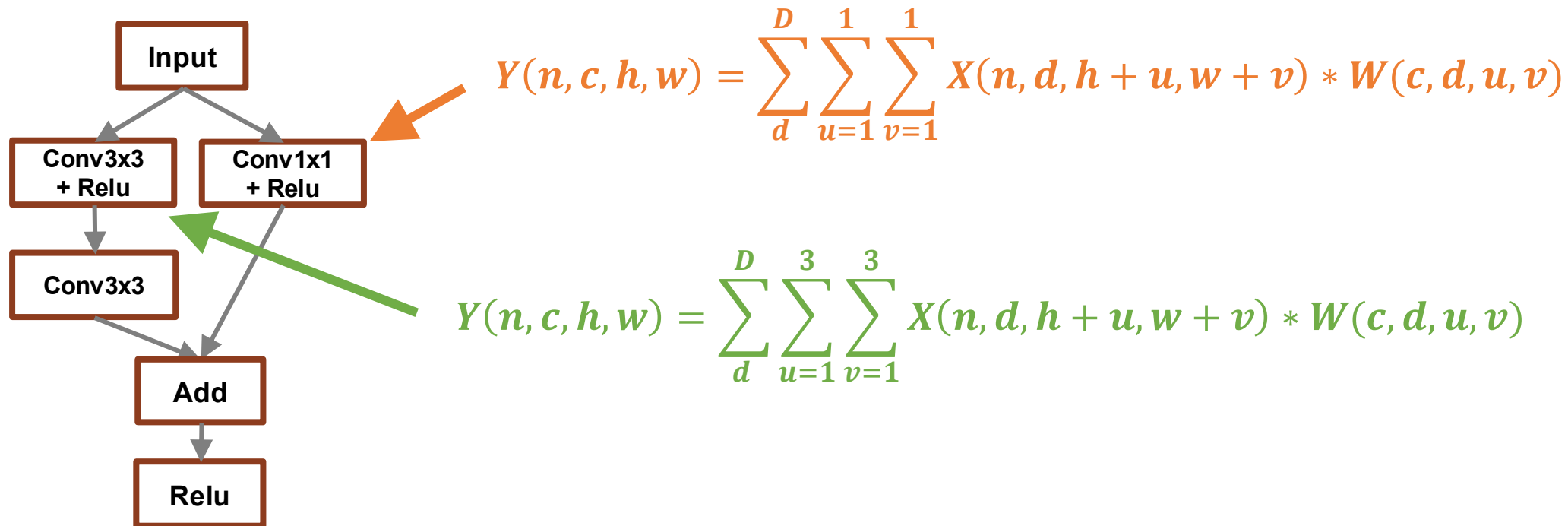
## **Scalability**

New operators and graph structures require more rules

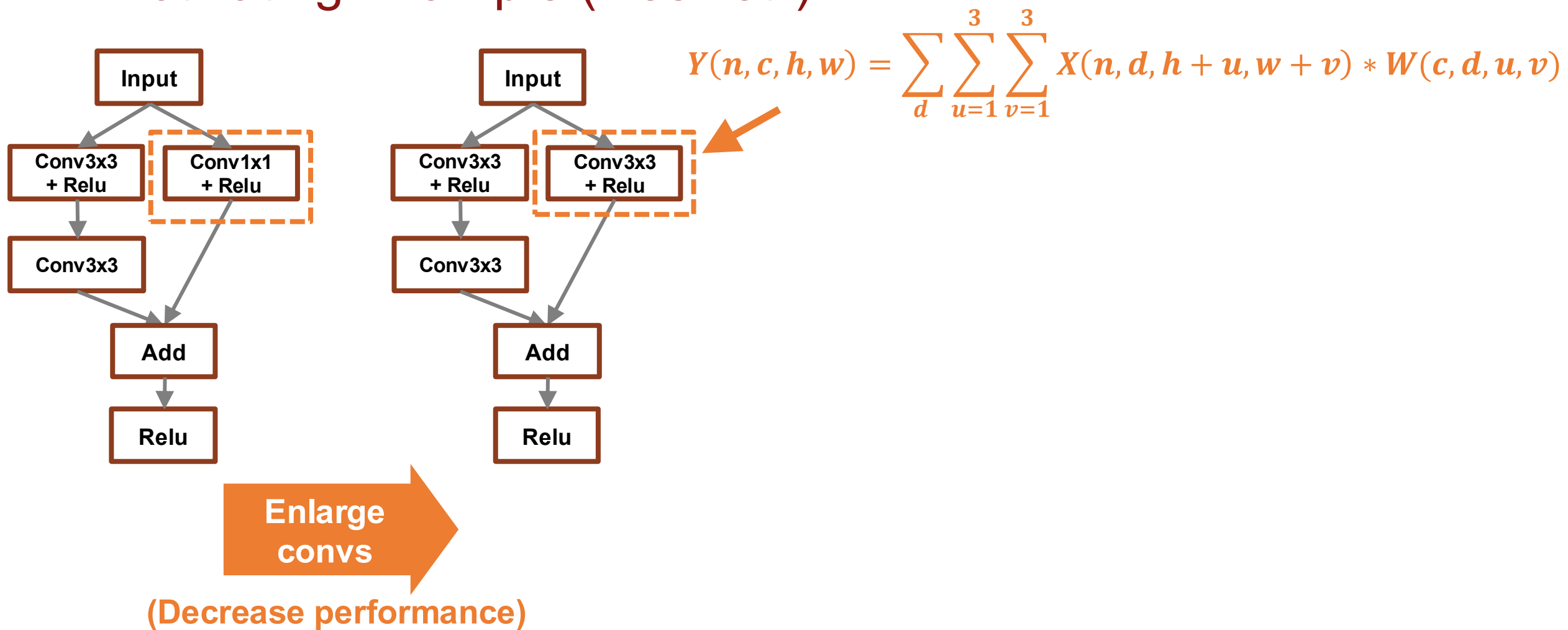
## **Performance**

Miss subtle optimizations for specific models/hardware

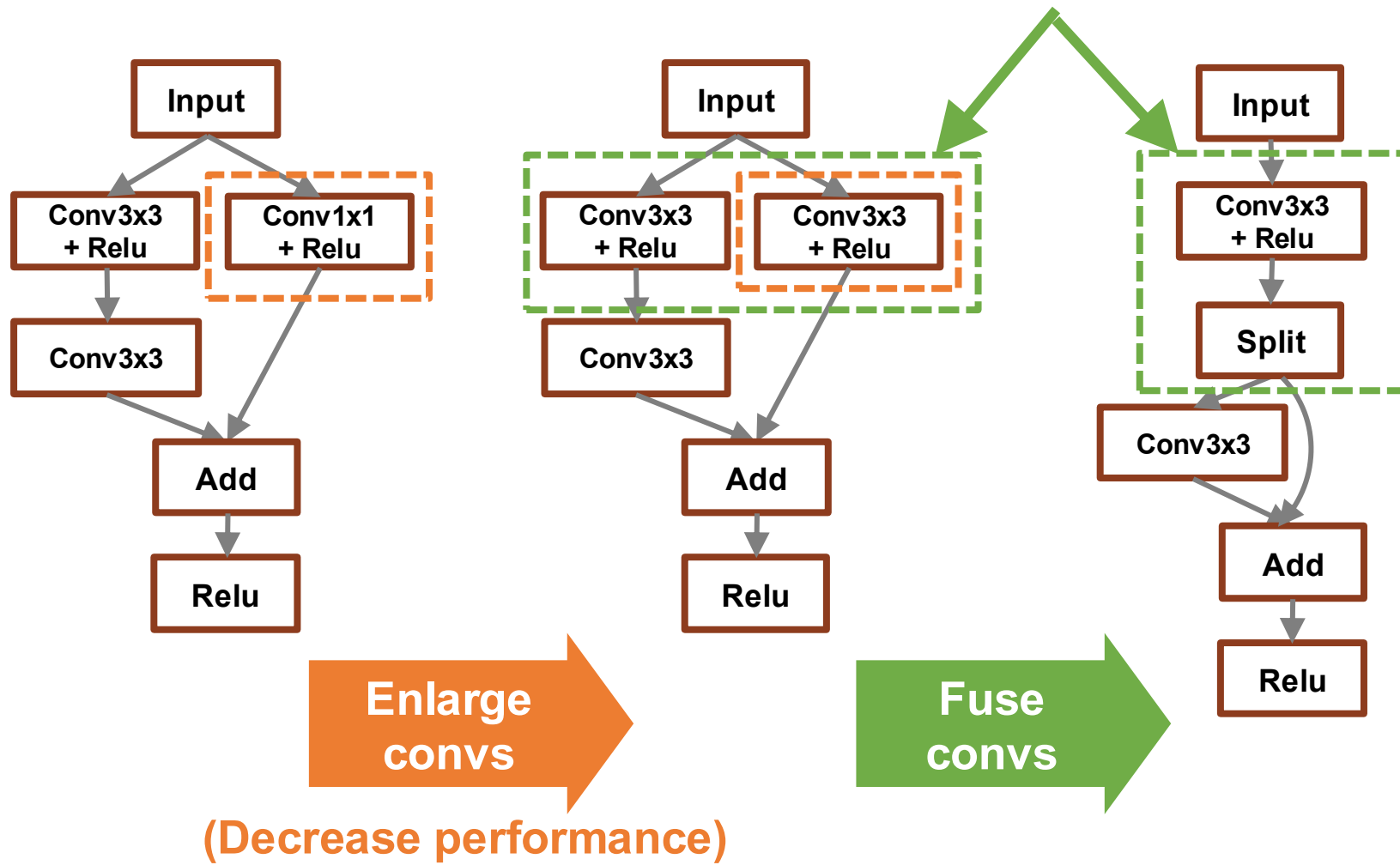
# Motivating Example (ResNet\*)



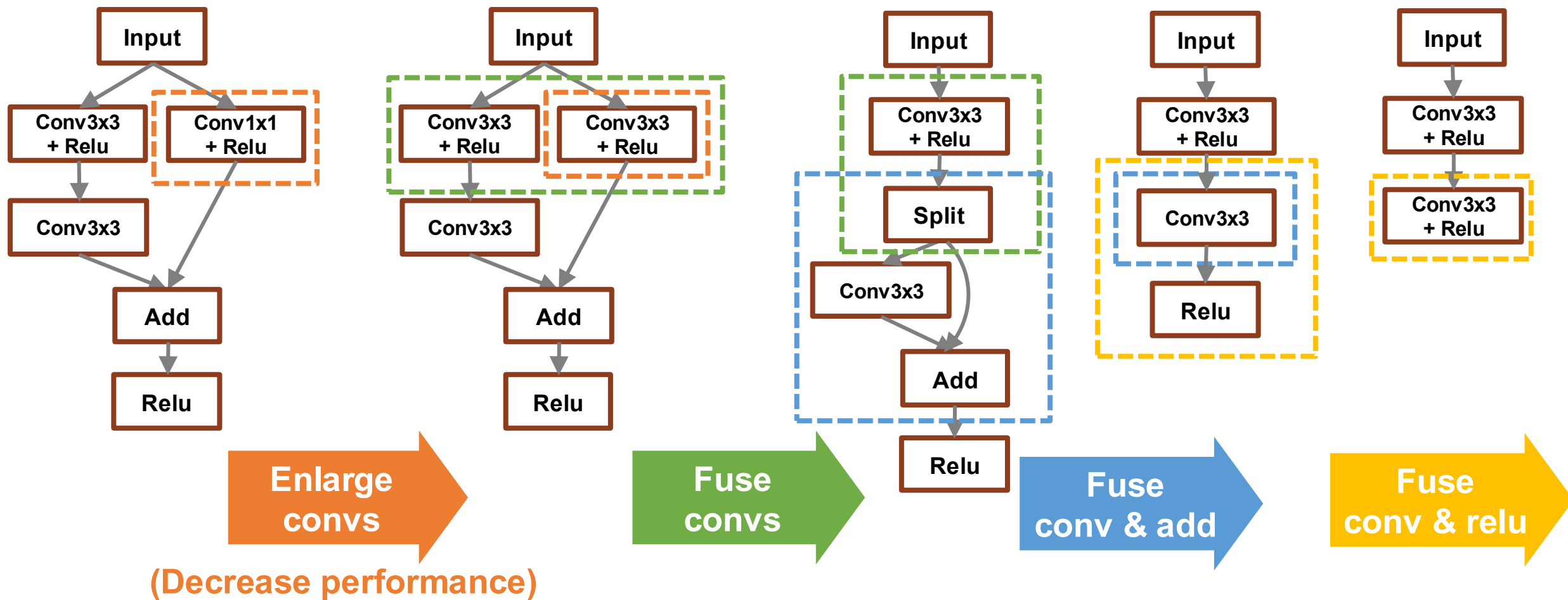
# Motivating Example (ResNet\*)



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

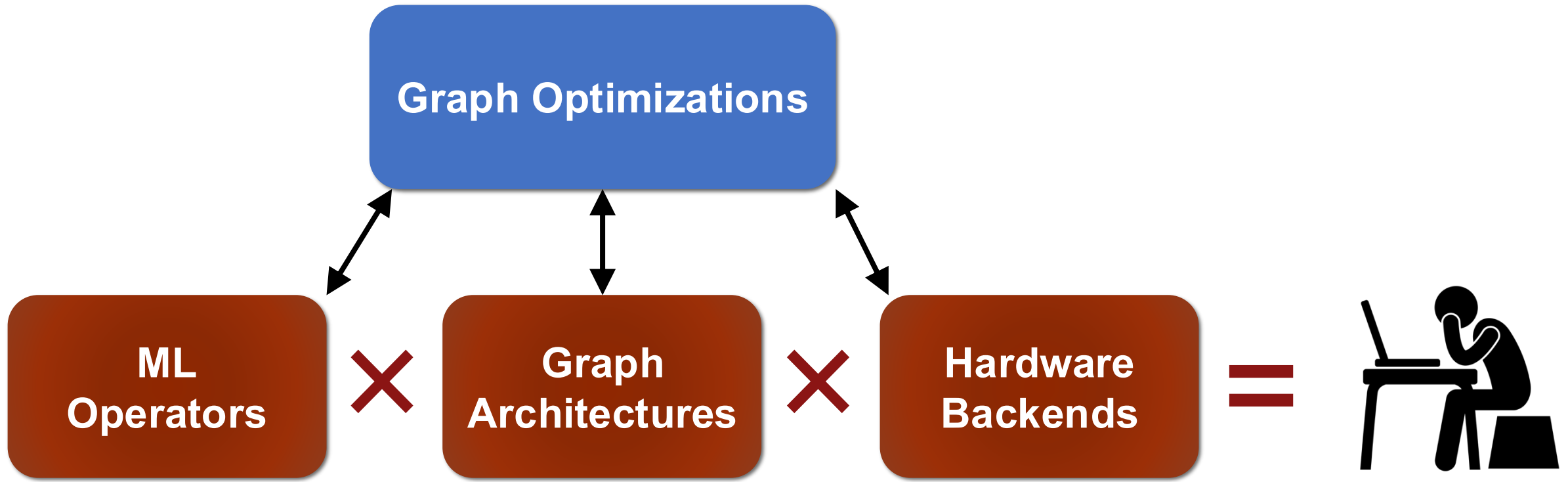


# Motivating Example (ResNet\*)



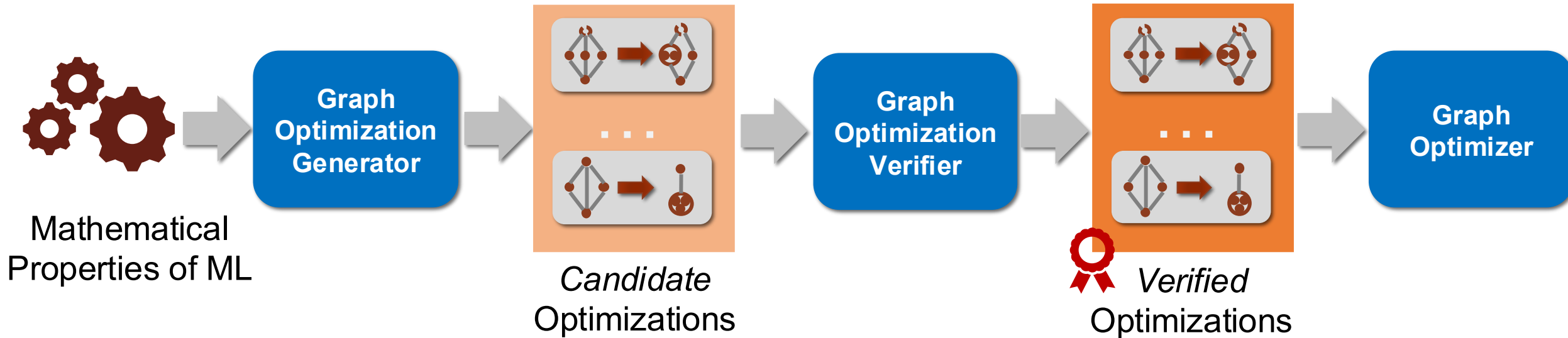
The final graph is 30% faster on V100 GPU but 10% slower on K80 GPU.



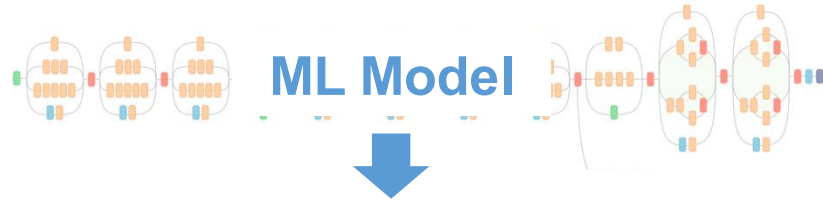


Infeasible to manually design graph optimizations for all cases

# Automated Graph Optimizations



# Layer 3: Parallelizing ML Computations



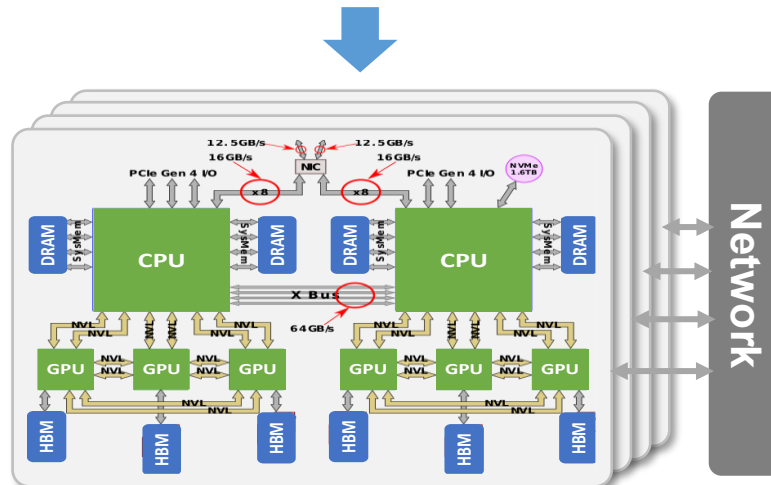
Automatic Differentiation

Graph-Level Optimization

Parallelization

Kernel Generation

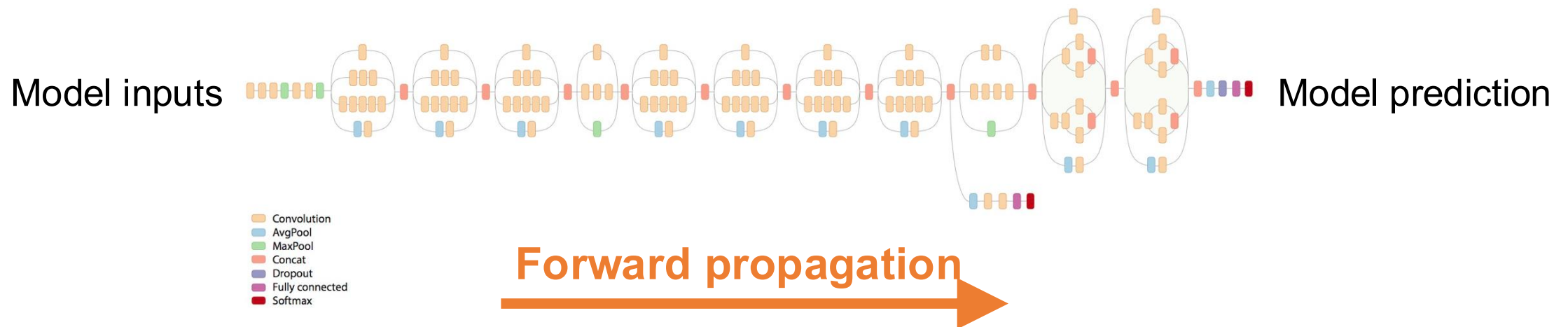
Memory Optimization



# Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

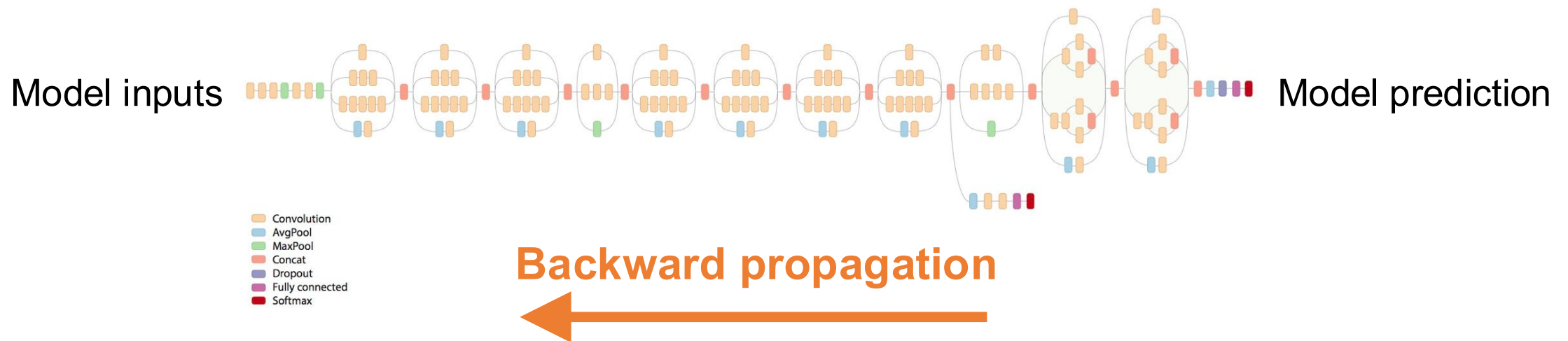
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights



# 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

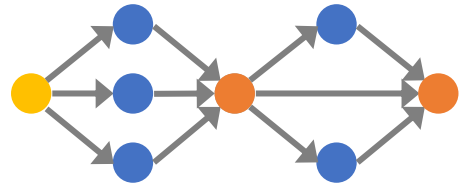
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

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

# 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



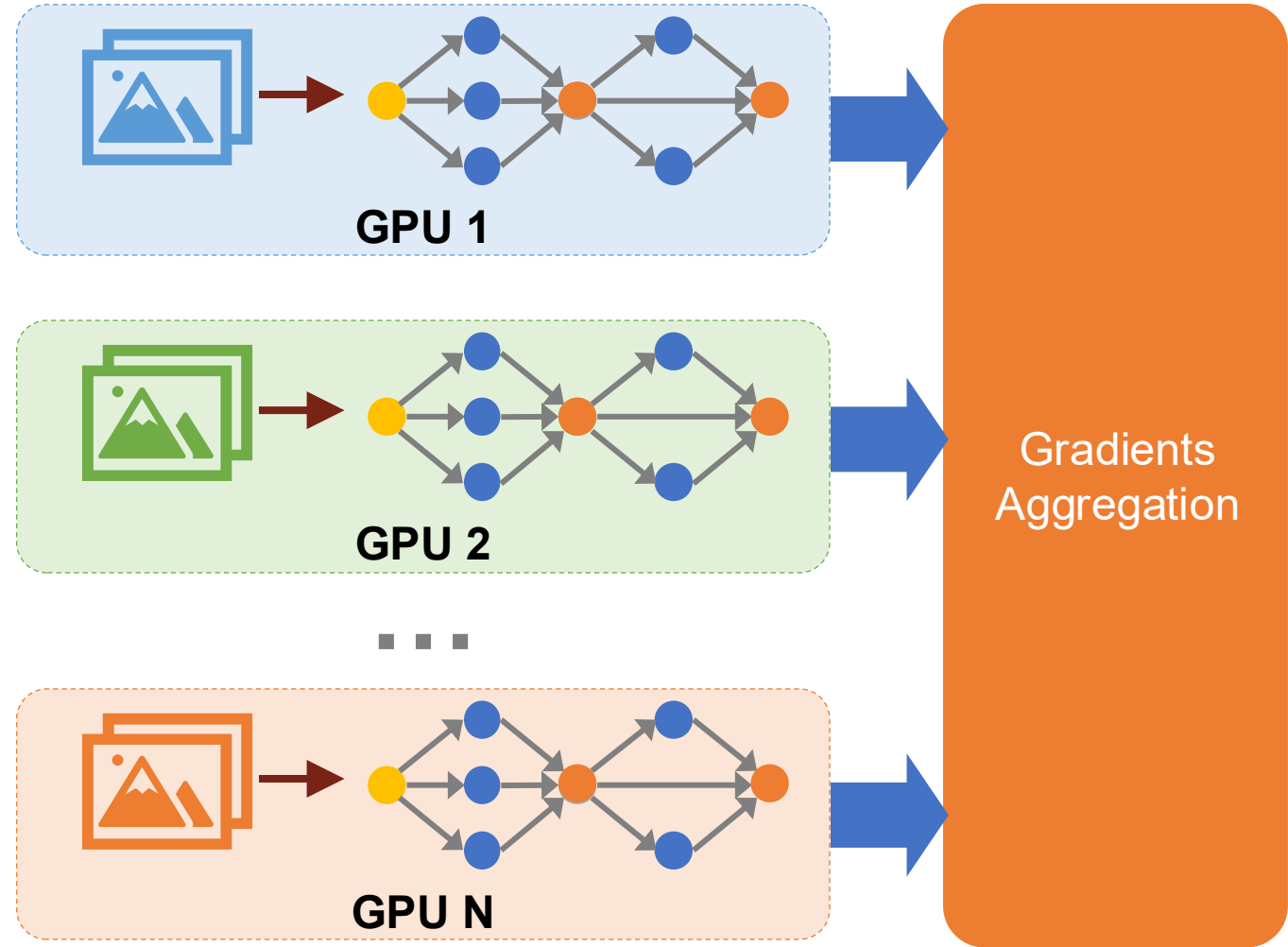
ML Model



Training Dataset

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

1. Partition training data into batches



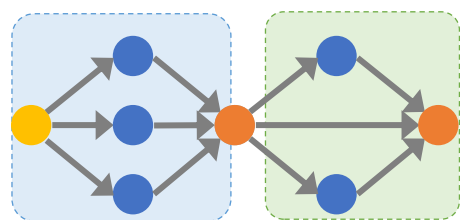
2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs



# Model Parallelism

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

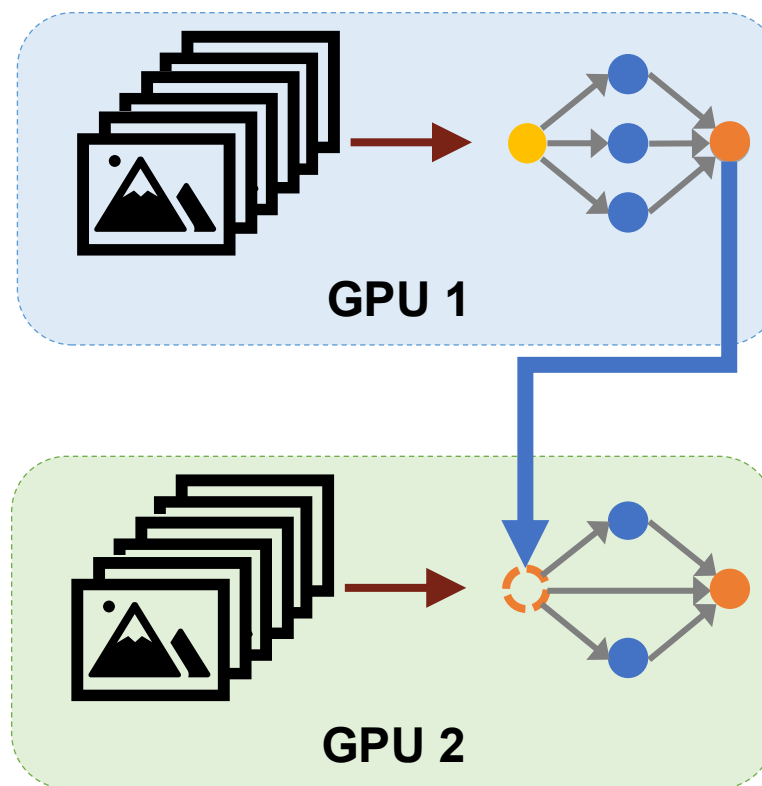


ML Model



Training Dataset

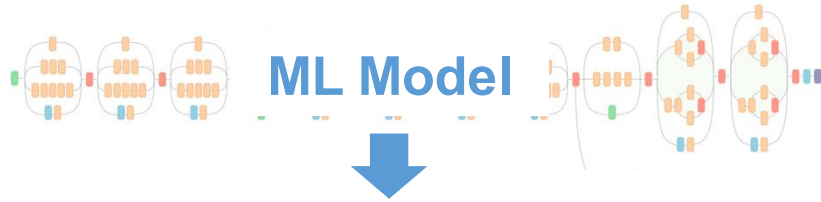
**Model  
Parallelism**



Transfer  
intermediate  
results  
between  
devices

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

# An Overview of Deep Learning Systems



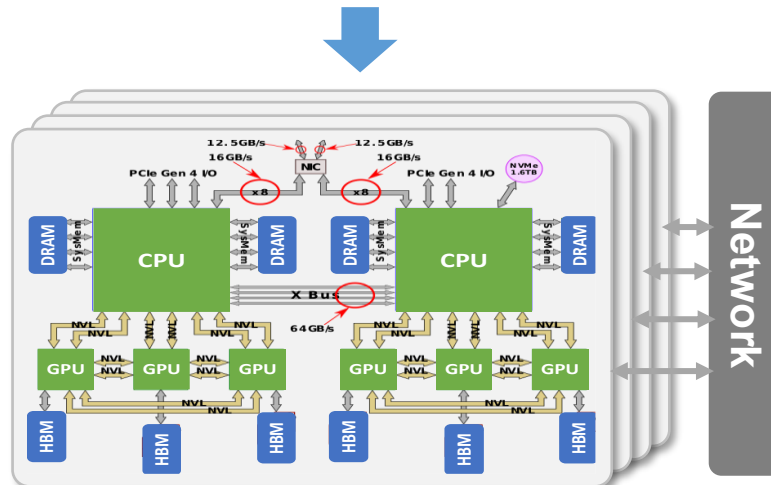
Automatic Differentiation

Graph-Level Optimization

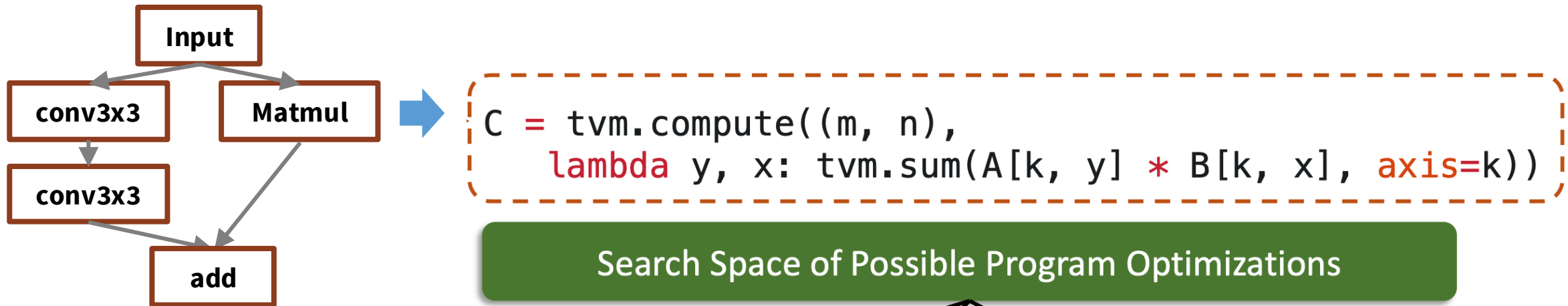
Parallelization

Kernel Generation

Memory Optimization



# Kernel Generation: 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):
        vdlc.fill_zero(CL)
        for ko in range(128):
            vdlc.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
            vdlc.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
            vdlc.fused_gemm8x8_add(CL, AL, BL)
            vdlc.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]
```

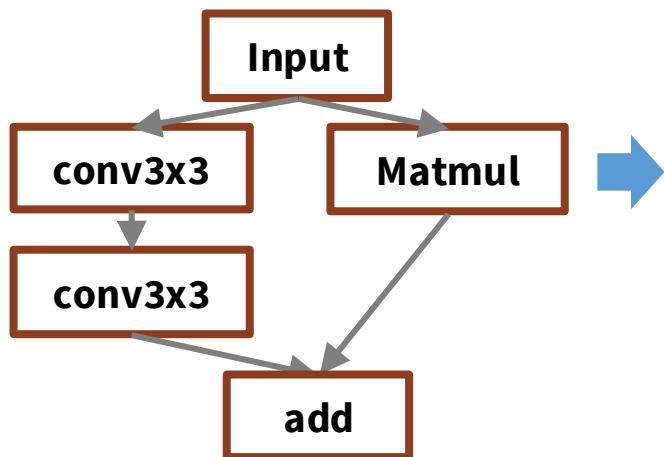
# Existing Approach: Engineer Optimized Tensor Programs

- 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



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

Loop  
Transformations

Thread  
Bindings

Cache  
Locality

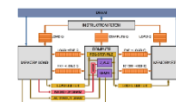
Thread  
Cooperation

Tensorization

Latency  
Hiding



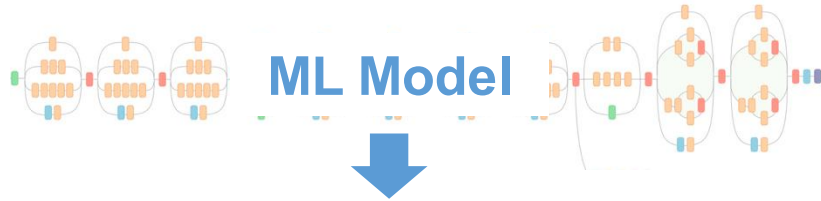
Hardware



Automated search for performant programs:

- ✓ Immediate support for new operators
- ✓ Better performance than hand-written kernels

# An Overview of Deep Learning Systems



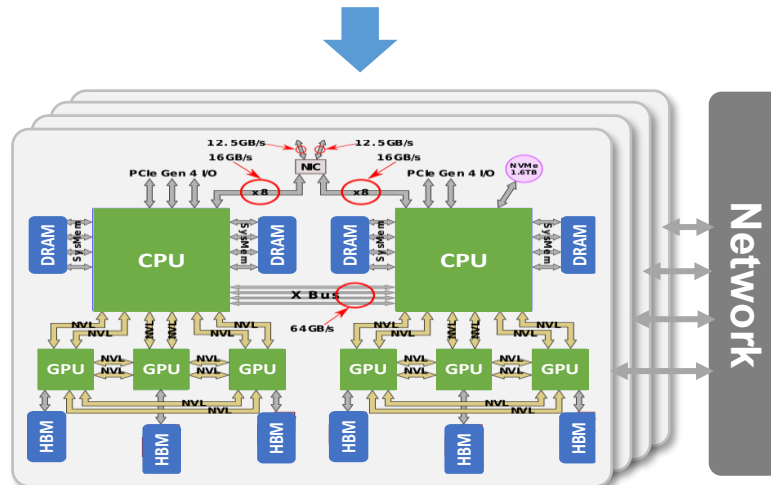
Automatic Differentiation

Graph-Level Optimization

Parallelization / Distributed Training

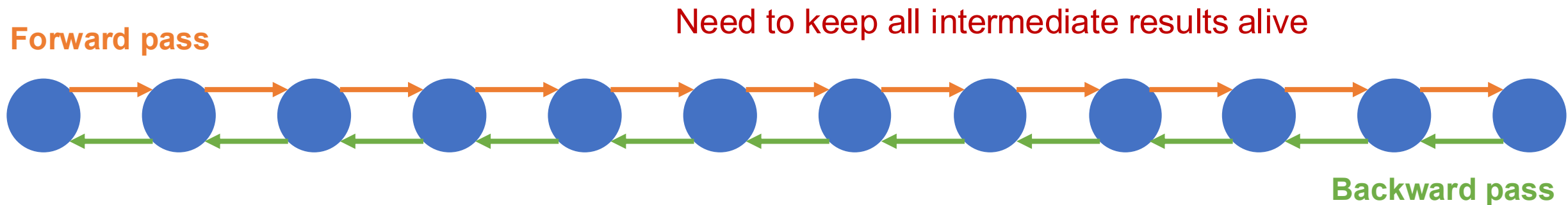
Code Optimization

Memory Optimization

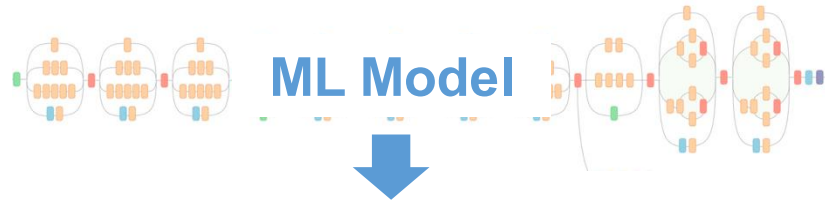


# GPU Memory is the Bottleneck in DNN Training

- 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



# Upcoming Lectures



Automatic Differentiation

Graph-Level Optimization

Parallelization / Distributed Training

Code Optimization

Memory Optimization

