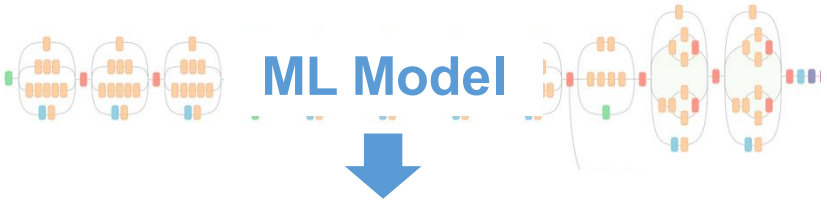
15-442/15-642: Machine Learning Systems

Introduction to Machine Learning Systems

Spring 2026

Tianqi Chen and Zhihao Jia
Carnegie Mellon University

An Overview of Machine learning Systems



Automatic Differentiation

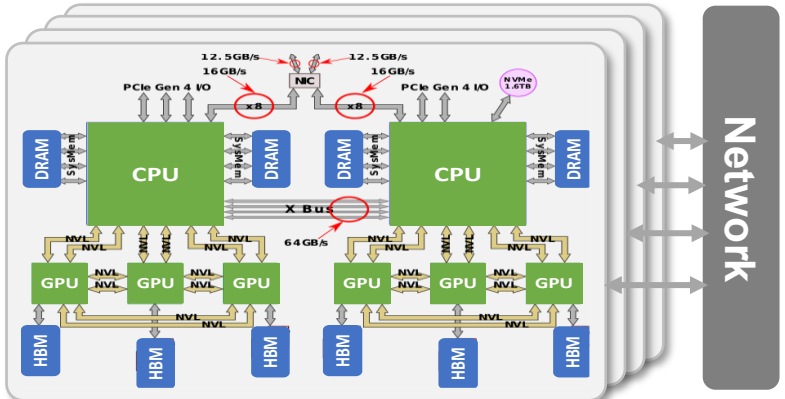
Graph-Level Optimization

Parallelization

Kernel Generation

Memory Optimization

ML Systems



Layer 1: Automatic Differentiation

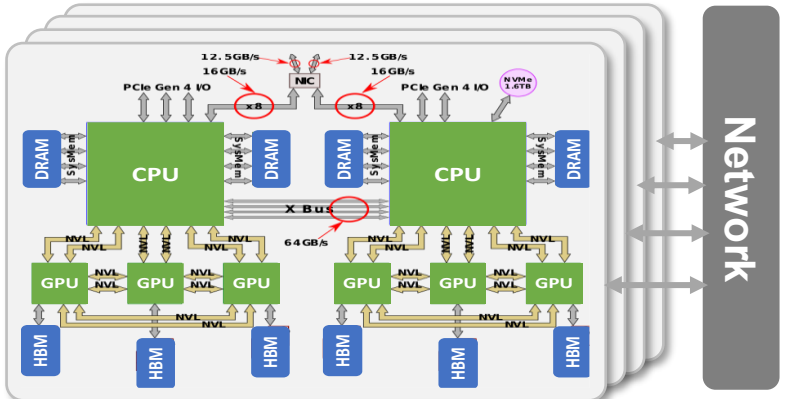
Automatic Differentiation

Graph-Level Optimization

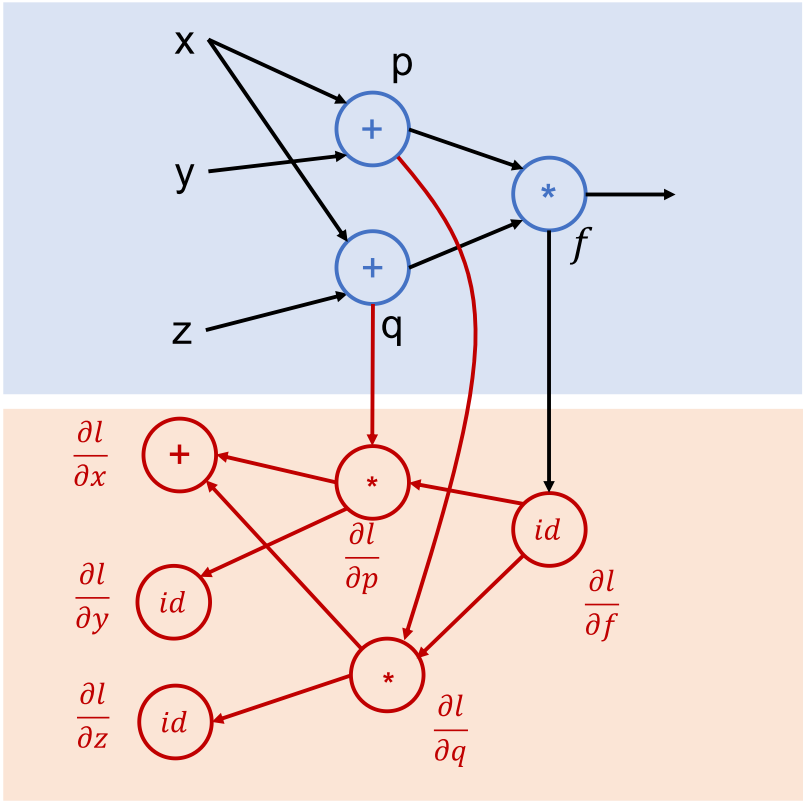
Parallelization

Kernel Generation

Memory Optimization



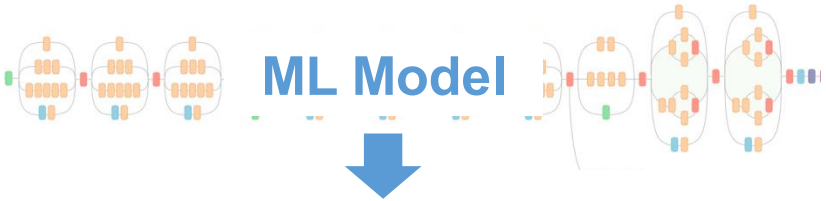
Automatically construct backward computation graph



Forward computation graph

Backward computation graph

Layer 2: Graph-Level Optimizations



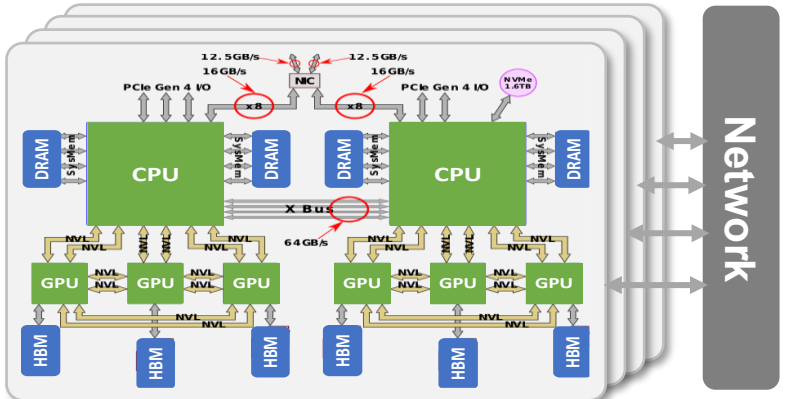
Automatic Differentiation

Graph-Level Optimization

Parallelization

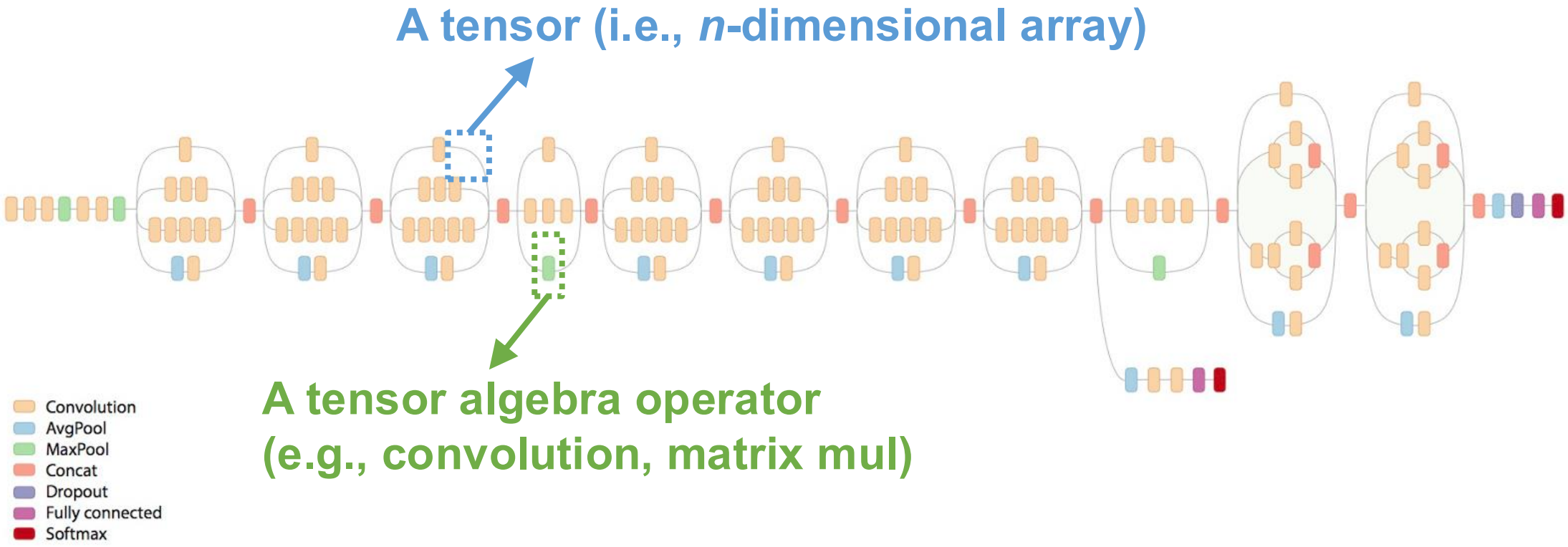
Kernel Generation

Memory Optimization

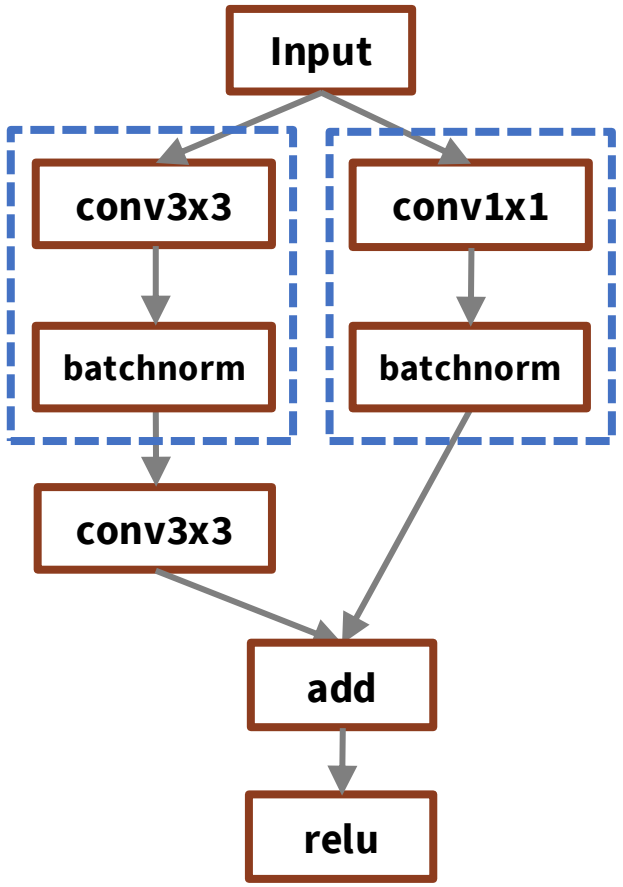


Recap: DNNs as Computation Graphs

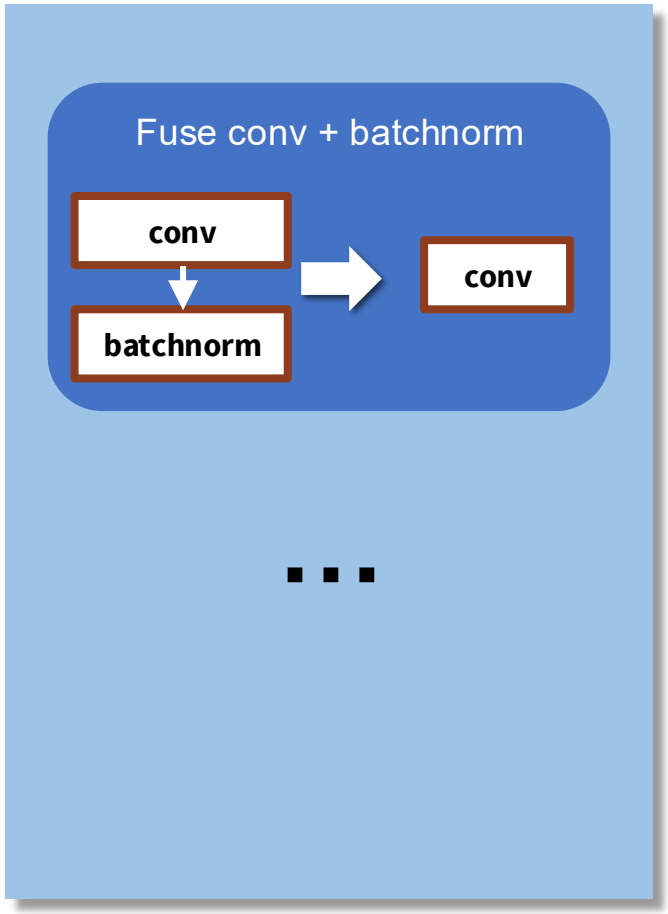
- Collection of simple trainable mathematical units that work together to solve complicated tasks



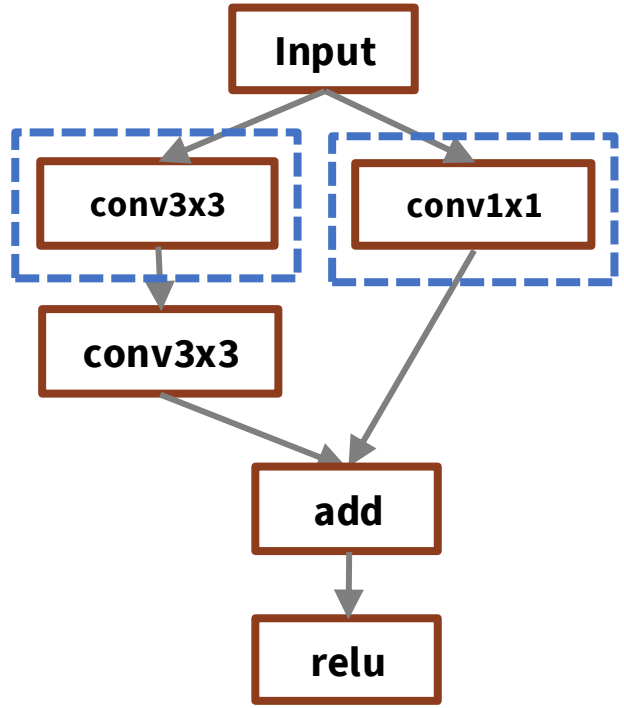
Graph-Level Optimizations



Input Computation Graph

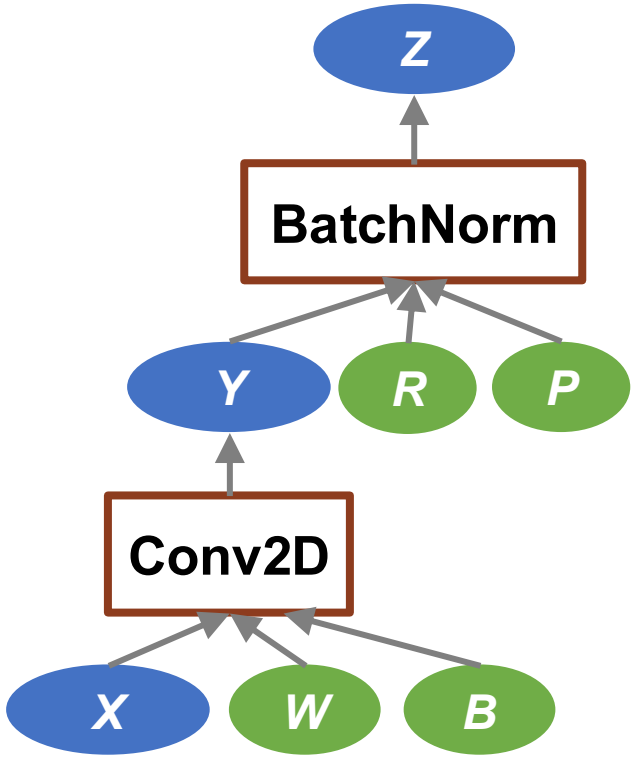


Potential graph transformations



Optimized Computation Graph

Example: Fusing Convolution and Batch Normalization

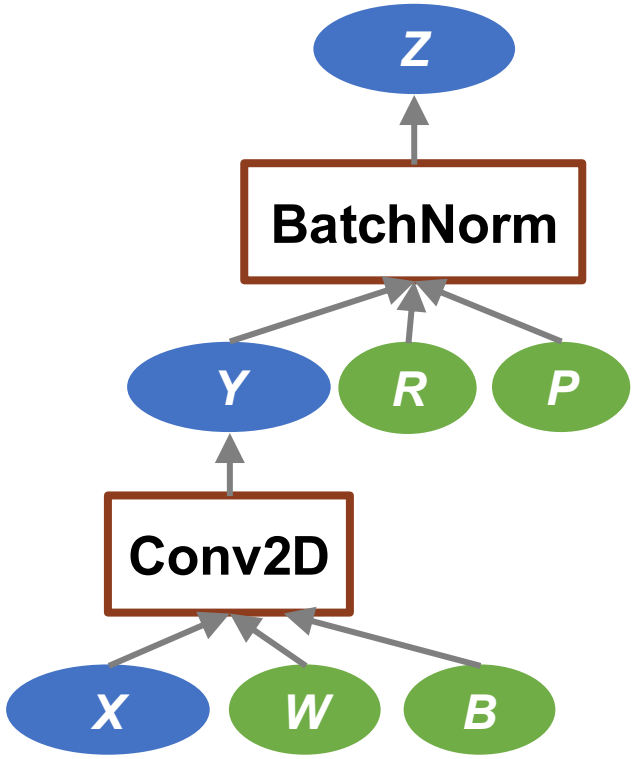


$$Z(n, c, h, w) = Y(n, c, h, w) * R(c) + P(c)$$

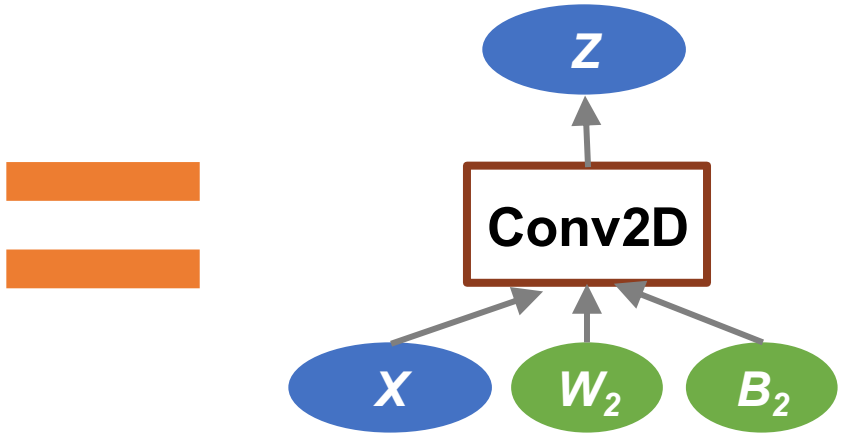
$$Y(n, c, h, w) = \left(\sum_{d,u,v} X(n, d, h + u, w + v) * W(c, d, u, v) \right) + B(n, c, h, w)$$

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

Fusing Conv and BatchNorm



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

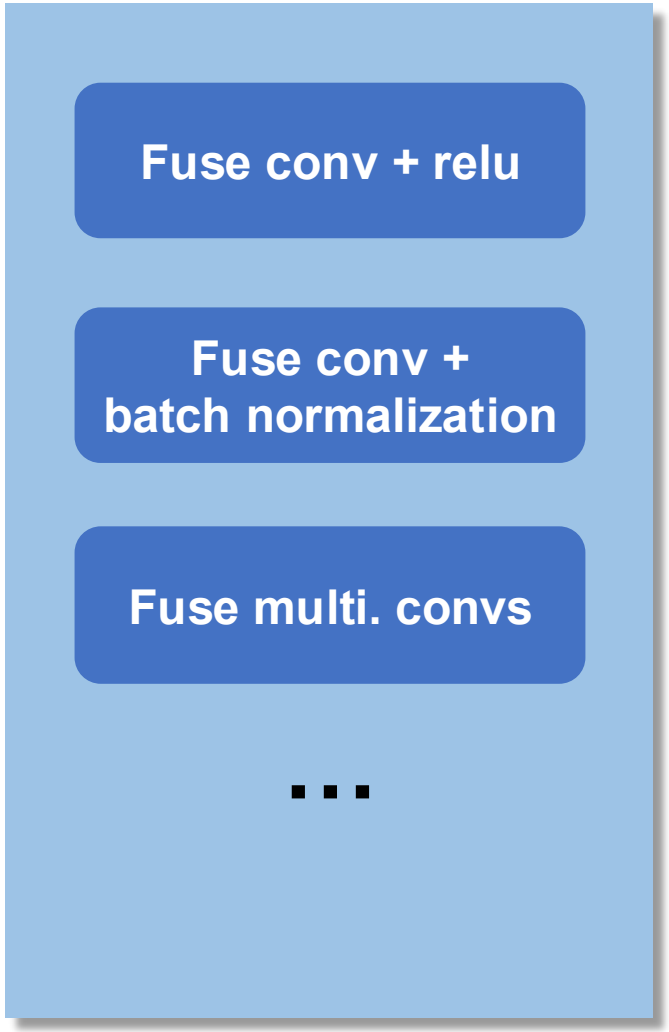


$$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$$

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

Current Rule-based Graph Optimizations

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

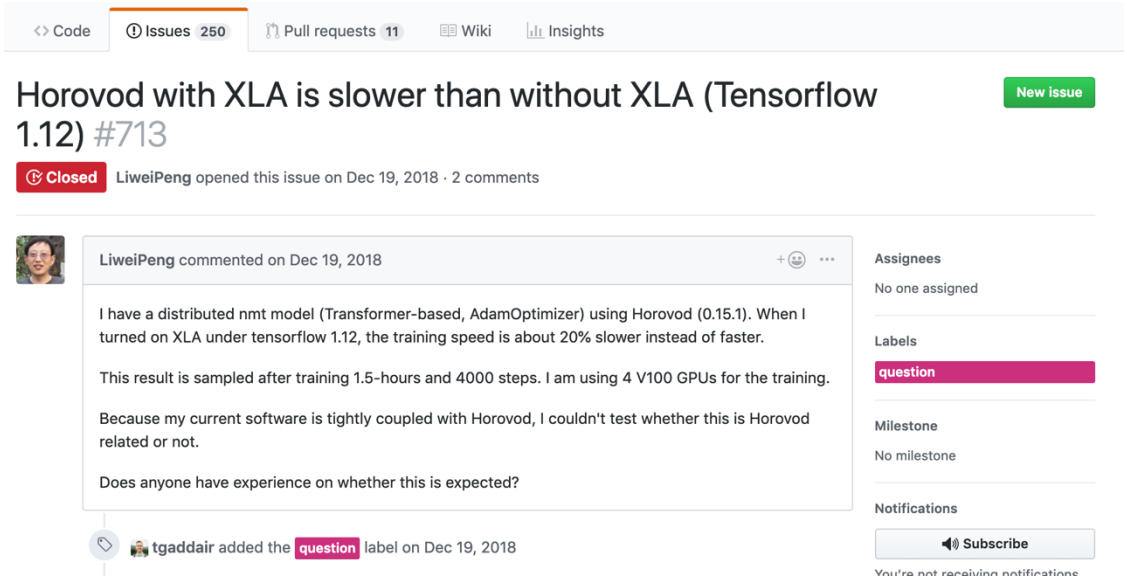


Rule-based Optimizer

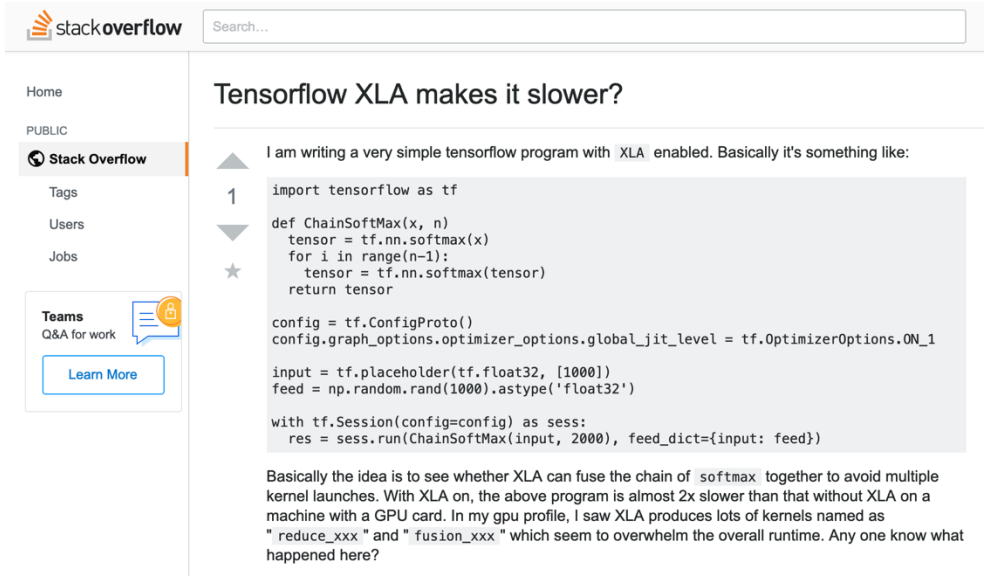
```
namespace tensorflow {
namespace graph_transforms {

// Converts Conv2D or MatMul ops followed by column-wise Muls into equivalent
// ops with the Mul baked into the convolution weights, to save computation
// during inference.
Status FoldBatchNorms(const GraphDef& input_graph_def,
                      const TransformFuncContext& context,
                      GraphDef* output_graph_def) {
  GraphDef replaced_graph_def;
  TF_RETURN_IF_ERROR(ReplaceMatchingOpTypes(
      input_graph_def,  // clang-format off
      {"Mul",                 // mul_node
       {
         {"Conv2D|MatMul|DepthwiseConv2dNative",  // conv_node
          {
            {"*"},         // input_node
            {"Const"},     // weights_node
          }
         },
         {"Const"},         // mul_values_node
       }
      },  // clang-format on
      [](const NodeMatch& match, const std::set<string>& input_nodes,
         const std::set<string>& output_nodes,
         std::vector<NodeDef>* new_nodes) {
        // Find all the nodes we expect in the subgraph.
        const NodeDef& mul_node = match.node;
        const NodeDef& conv_node = match.inputs[0].node;
        const NodeDef& input_node = match.inputs[0].inputs[0].node;
        const NodeDef& weights_node = match.inputs[0].inputs[1].node;
        const NodeDef& mul_values_node = match.inputs[1].node;

        // Check that nodes that we use are not used somewhere else.
        for (const auto& node : {conv_node, weights_node, mul_values_node}) {
          if (output_nodes.count(node.name())) {
            // Return original nodes.
            new_nodes->insert(new_nodes->end(),
                              {mul_node, conv_node, input_node, weights_node,
                               mul_values_node});
            return Status::OK();
          }
        }

        Tensor weights = GetNodeTensorAttr(weights_node, "value");
        Tensor mul_values = GetNodeTensorAttr(mul_values_node, "value");

        // Make sure all the inputs really are vectors, with as many entries as
        // there are columns in the weights.
        int64 weights_cols;
        if (conv_node.op() == "Conv2D") {
          weights_cols = weights.shape().dim_size(3);
        } else if (conv_node.op() == "DepthwiseConv2dNative") {
          weights_cols =
              weights.shape().dim_size(2) * weights.shape().dim_size(3);
        } else {
          weights_cols = weights.shape().dim_size(1);
        }
        if ((mul_values.shape().dims() != 1) ||
            (mul_values.shape().dim_size(0) != weights_cols)) {
          return errors::InvalidArgument(
              "Mul constant input to batch norm has bad shape: ",
              mul_values.shape().DebugString());
        }

        // Multiply the original weights by the scale vector.
        auto weights_vector = weights.flat<float>();
        Tensor scaled_weights(DT_FLOAT, weights.shape());
        auto scaled_weights_vector = scaled_weights.flat<float>();
        for (int64 row = 0; row < weights_vector.dimension(0); ++row) {
          scaled_weights_vector(row) =
              weights_vector(row) *
              mul_values.flat<float>()(row % weights_cols);
        }

        // Construct the new nodes.
        NodeDef scaled_weights_node;
        scaled_weights_node.set_op("Const");
        scaled_weights_node.set_name(weights_node.name());
        SetNodeAttr("dtype", DT_FLOAT, &scaled_weights_node);
        SetNodeTensorAttr<float>("value", scaled_weights, &scaled_weights_node);
        new_nodes->push_back(scaled_weights_node);

        new_nodes->push_back(input_node);

        NodeDef new_conv_node;
        new_conv_node = conv_node;
        new_conv_node.set_name(mul_node.name());
        new_nodes->push_back(new_conv_node);

        return Status::OK();
      },
      {}, &replaced_graph_def));
  *output_graph_def = replaced_graph_def;
  return Status::OK();
}

REGISTER_GRAPH_TRANSFORM("fold_batch_norms", FoldBatchNorms);

}  // namespace graph_transforms
}  // namespace tensorflow
```

Limitations of Rule-based Optimizations

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



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



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

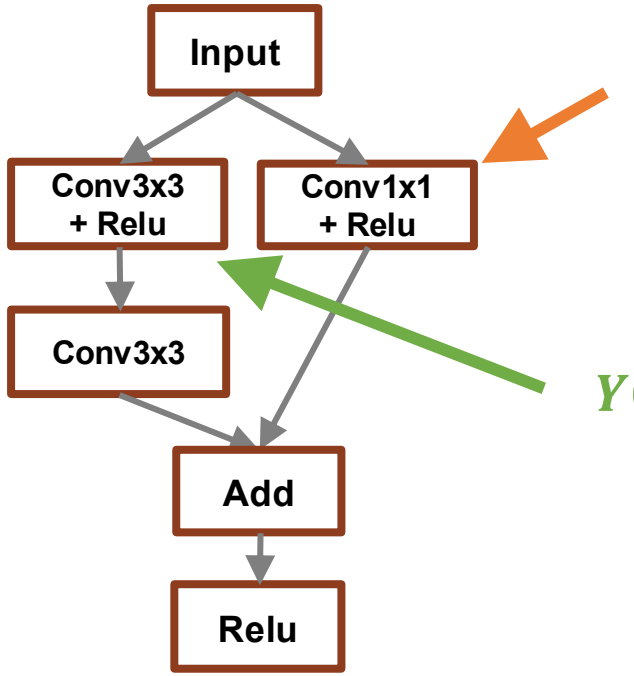
Limitations of Rule-based Optimizations

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*)

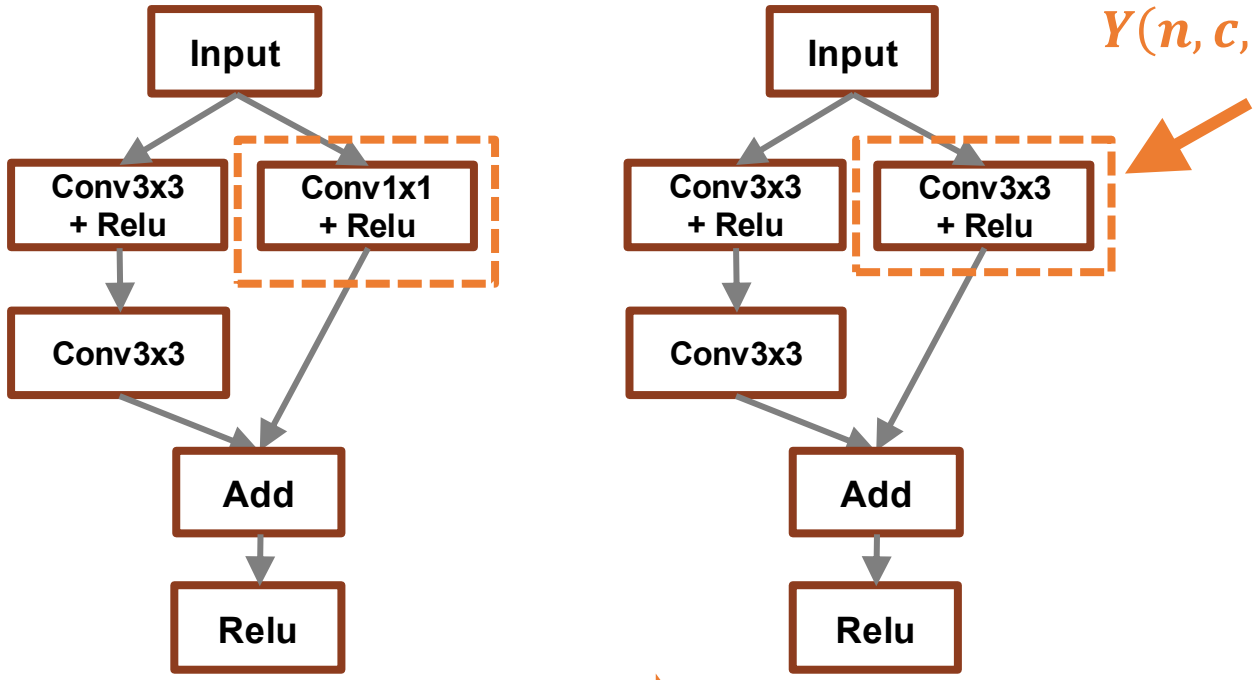


$$Y(n,c,h,w) = \sum_{d}^{D} \sum_{u=1}^{1} \sum_{v=1}^{1} X(n,d,h+u,w+v) * W(c,d,u,v)$$

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

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

Motivating Example (ResNet*)



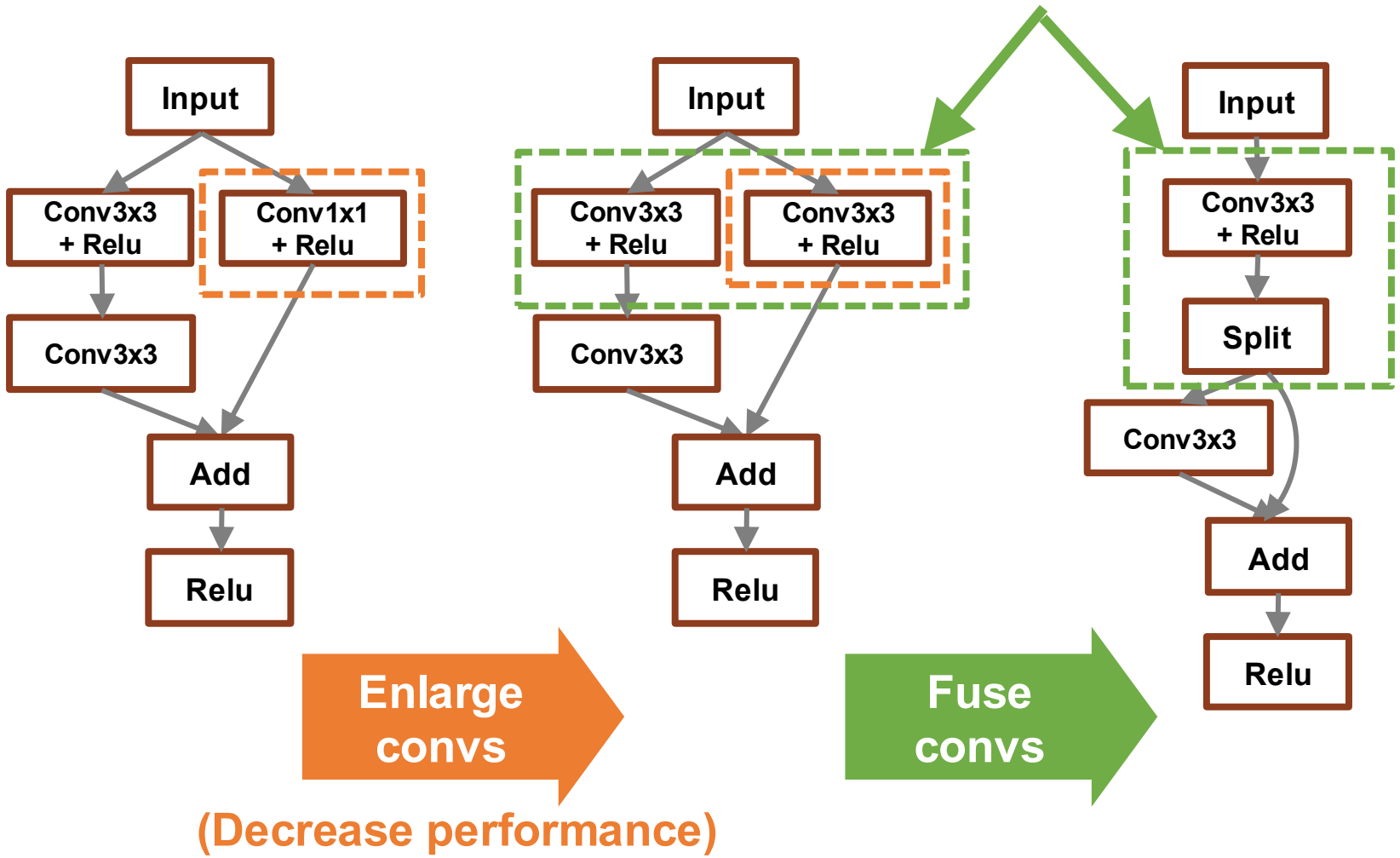
$$Y(n,c,h,w) = \sum_{d} \sum_{u=1}^{3} \sum_{v=1}^{3} X(n,d,h+u,w+v) * W(c,d,u,v)$$

Enlarge convs

(Decrease performance)

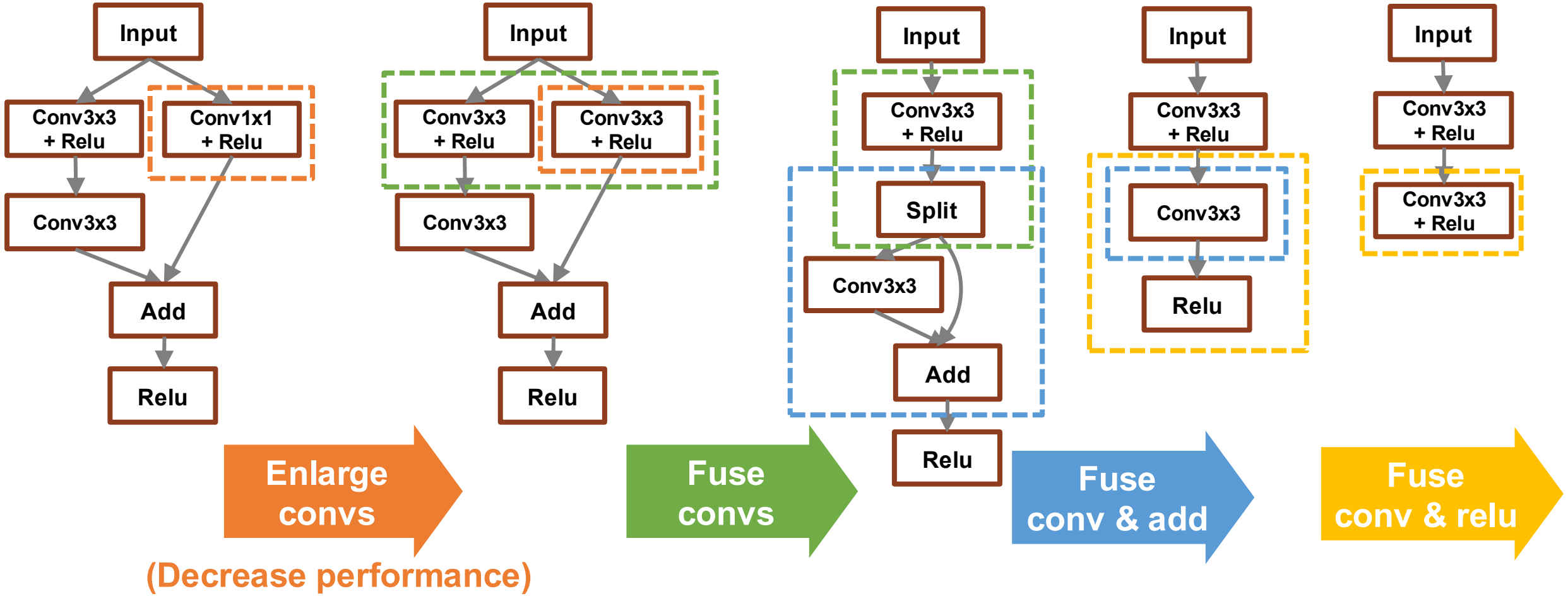
* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015

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



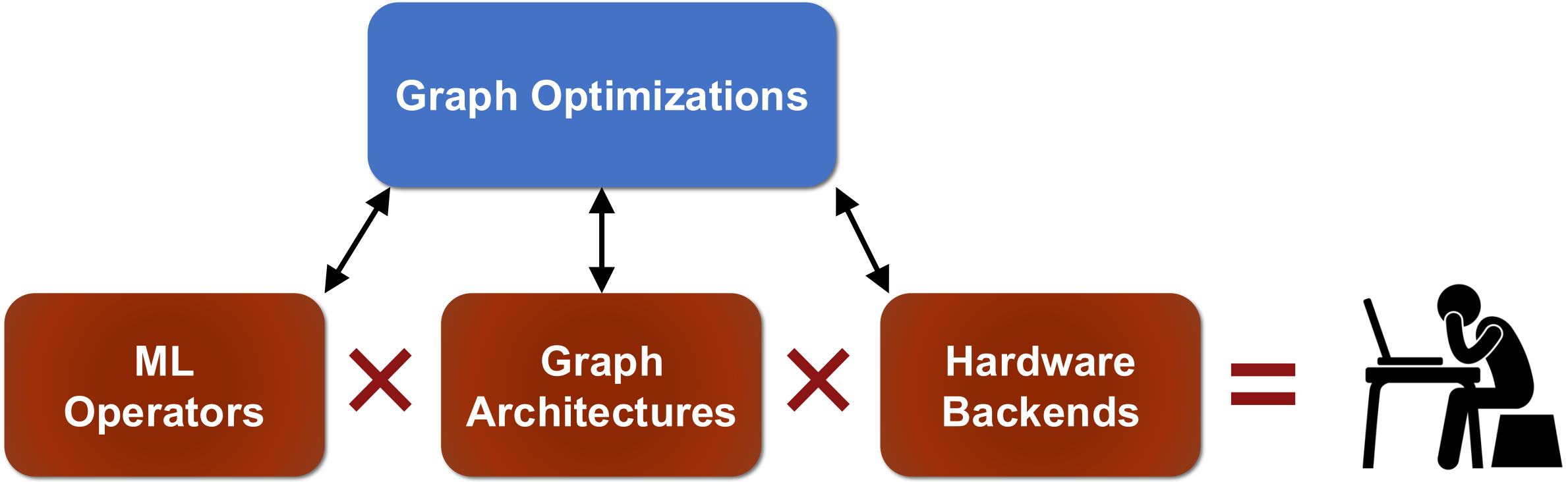
* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015

Motivating Example (ResNet*)



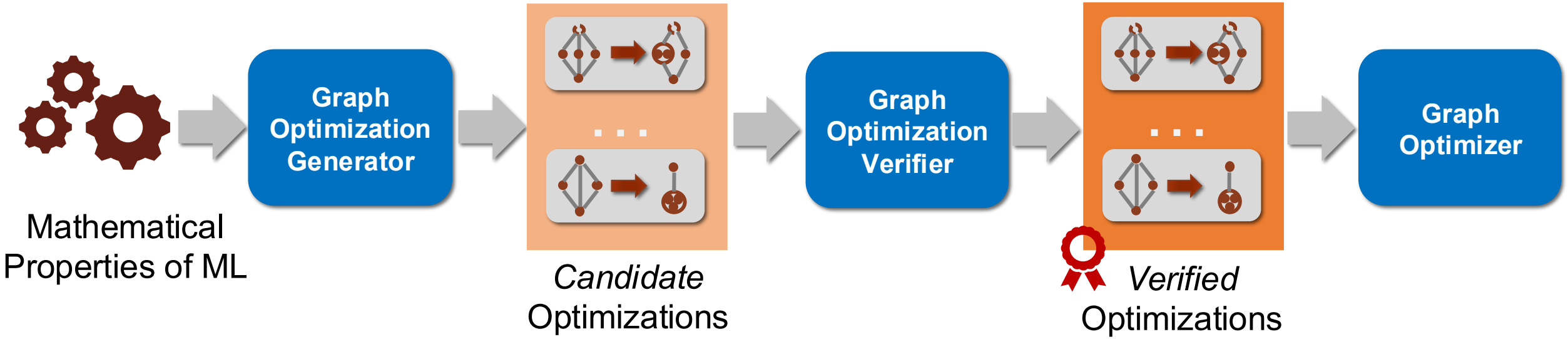
The final graph is <u>30% faster</u> on V100 GPU but <u>10% slower</u> on K80 GPU.

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

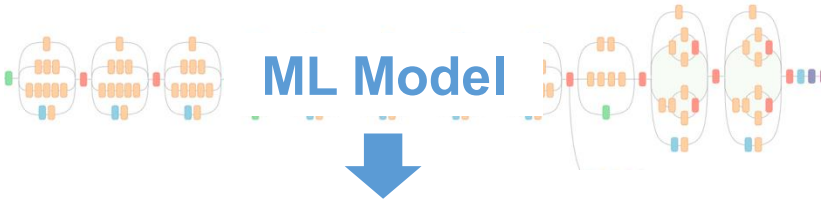


Infeasible to manually design graph optimizations for all cases

Automated Graph Optimizations



Layer 3: Parallelizing ML Computations



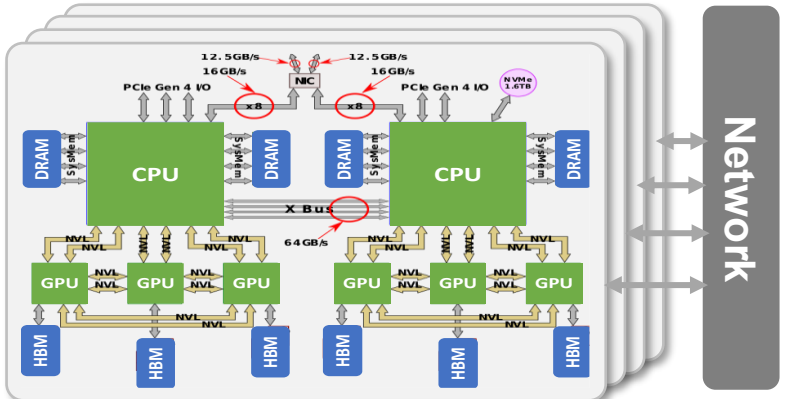
Automatic Differentiation

Graph-Level Optimization

Parallelization

Kernel Generation

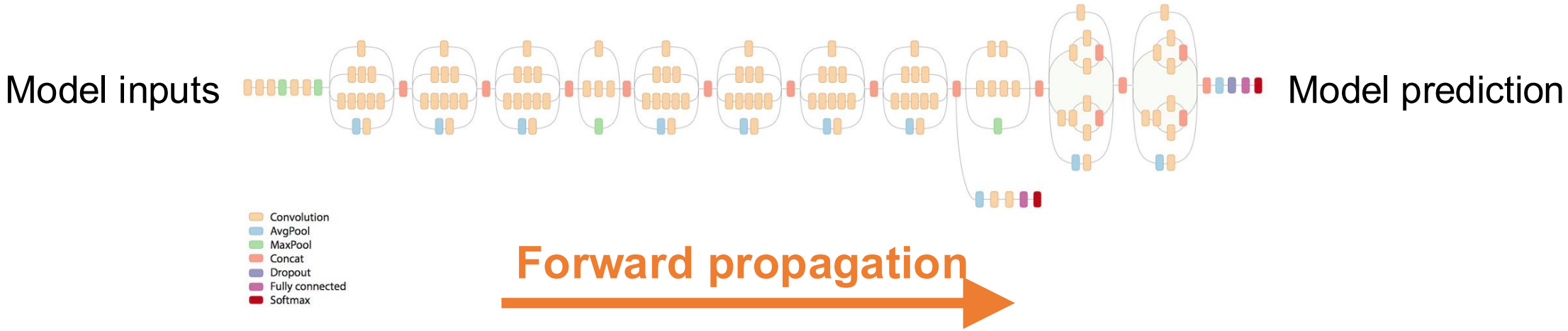
Memory Optimization



Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

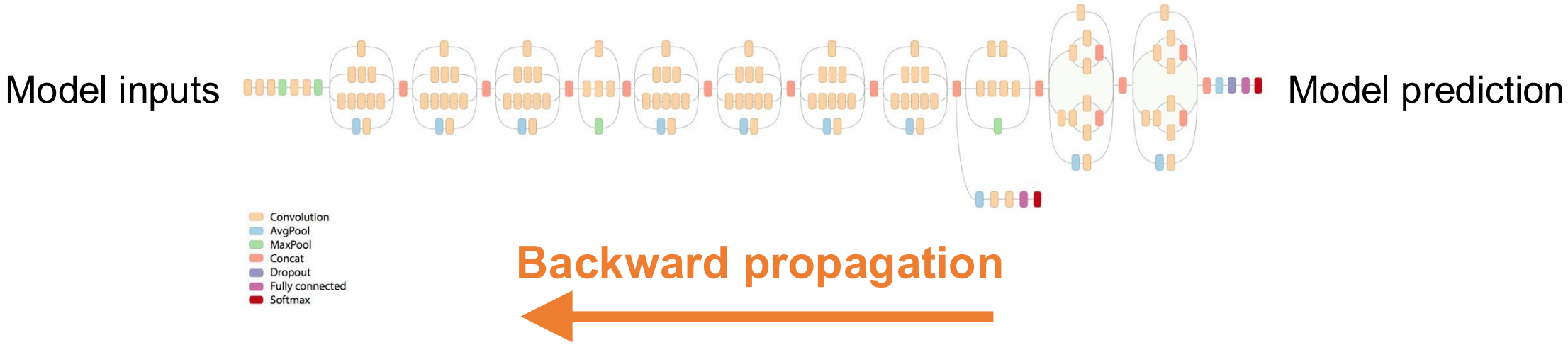
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights



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

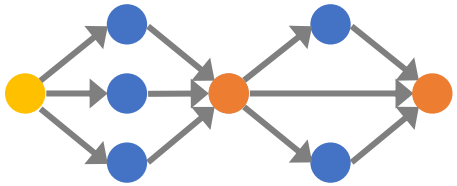
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

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

How can we parallelize ML training?

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

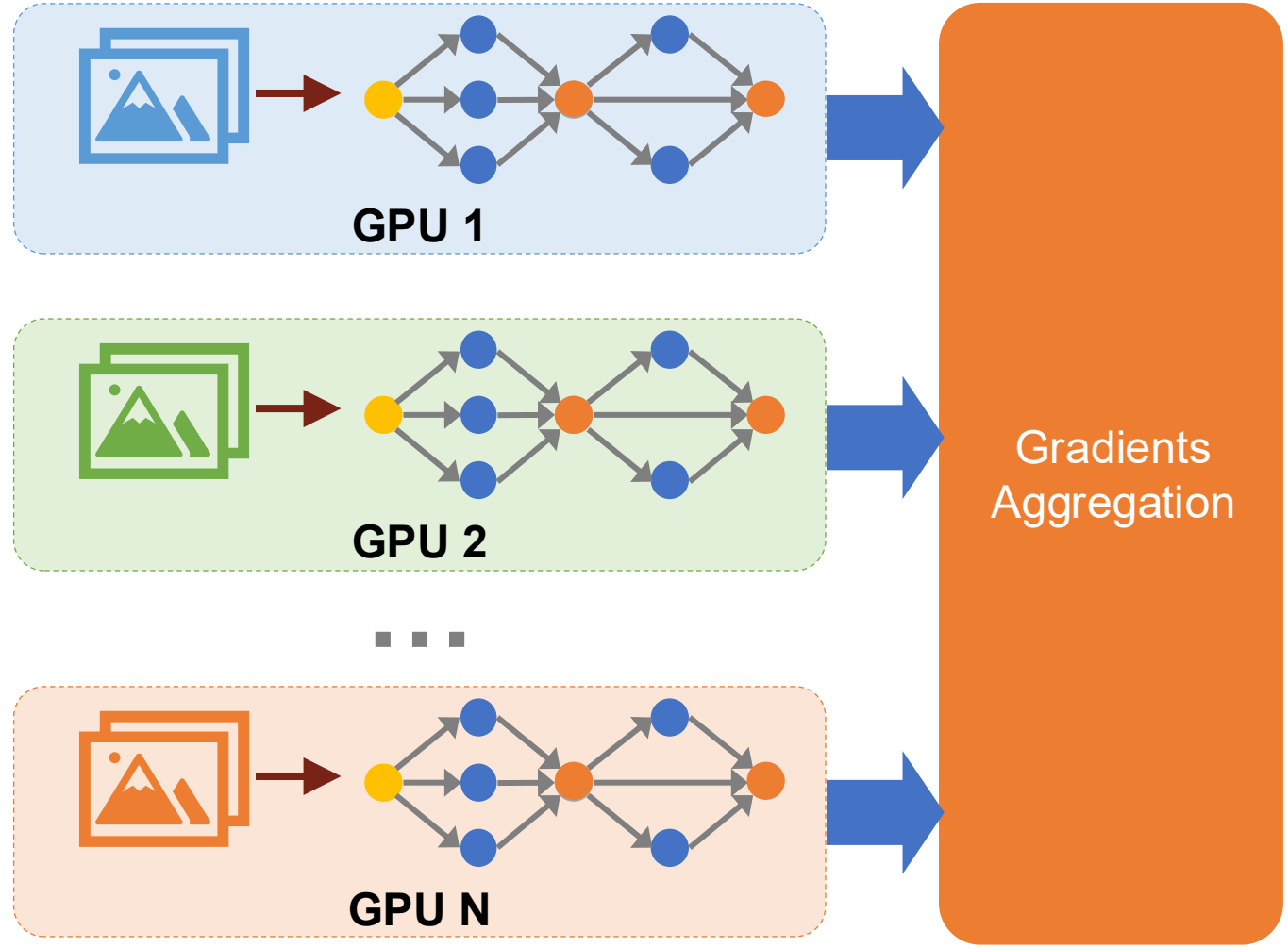
Data Parallelism



ML Model

Training Dataset

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



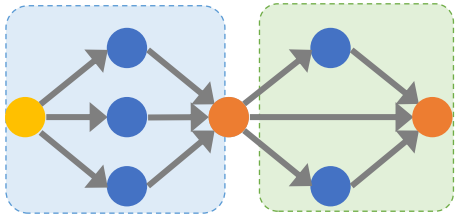
1. Partition training data into batches

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

Model Parallelism

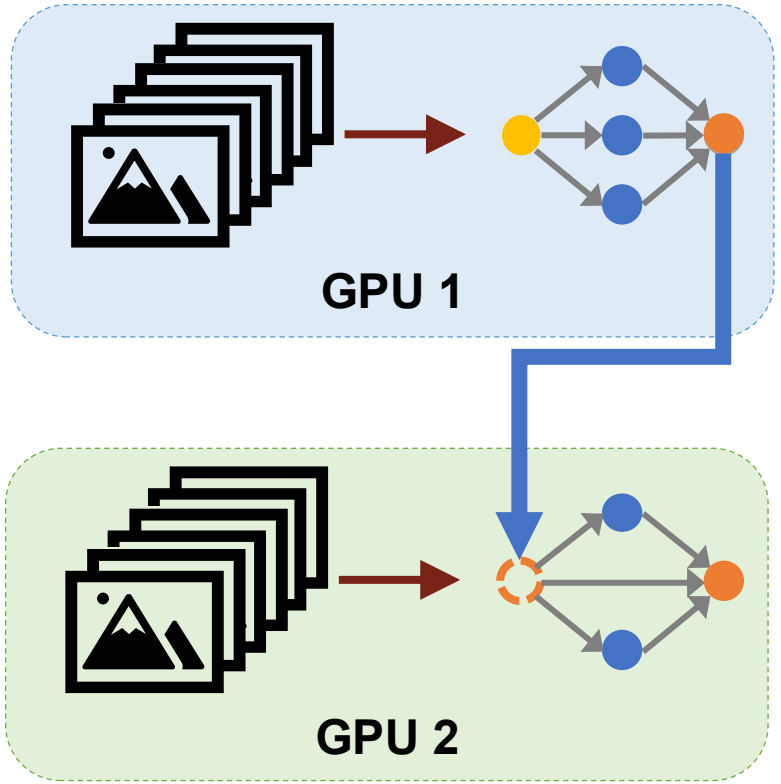
- Split a model into multiple subgraphs and assign them to different devices



ML Model

Training Dataset

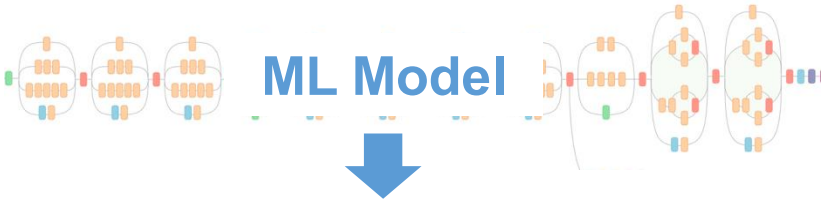
Model Parallelism



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



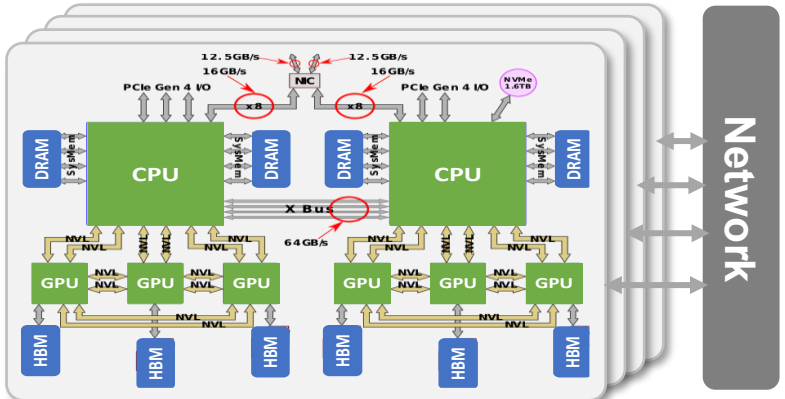
Automatic Differentiation

Graph-Level Optimization

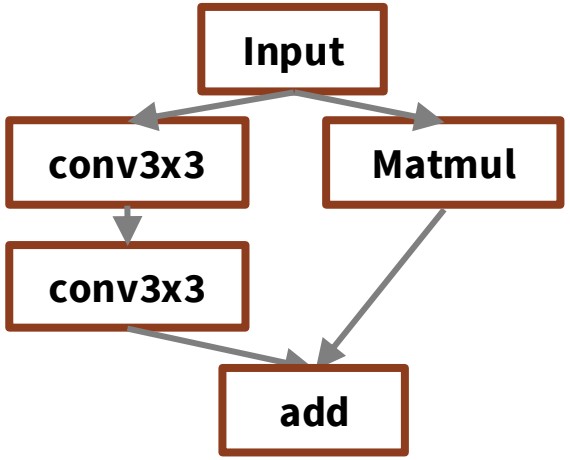
Parallelization

Kernel Generation

Memory Optimization



Kernel Generation: How to find performant programs for each operator?




```
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]
```

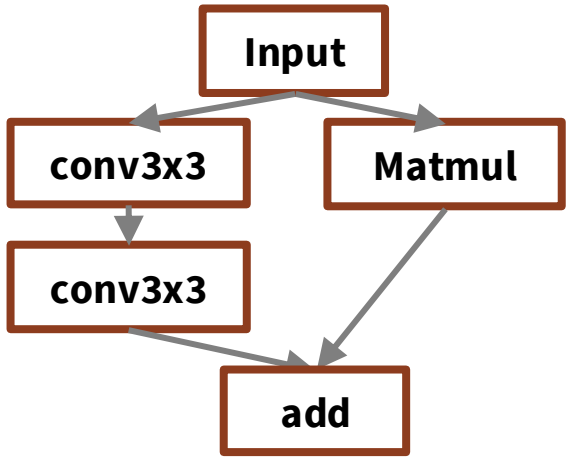
Existing Approach: Engineer Optimized Tensor Programs

- 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

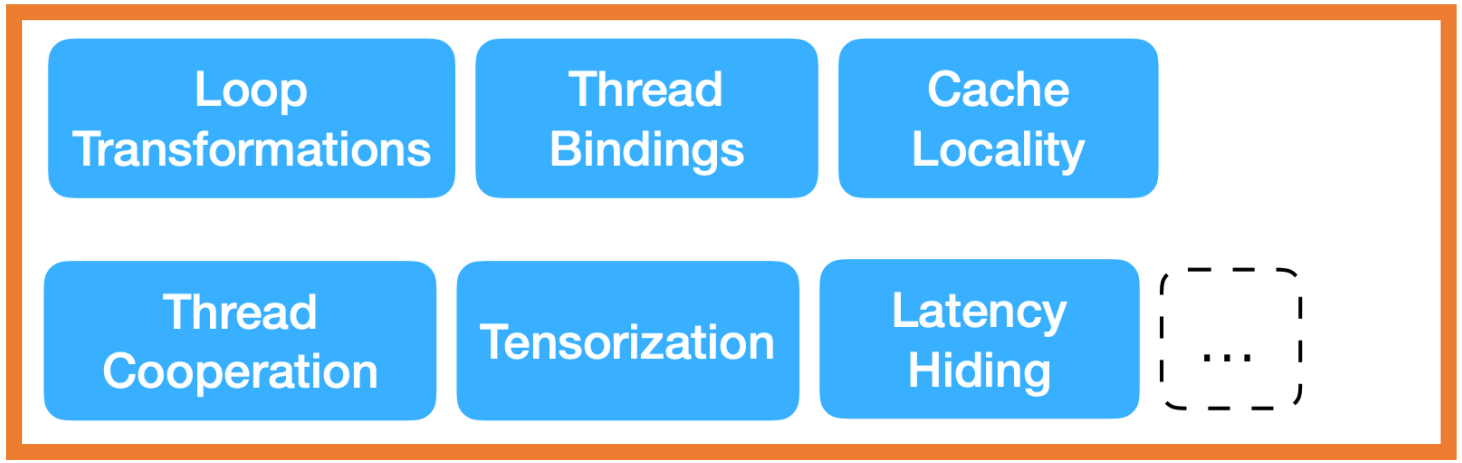
Automated Code Generation




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

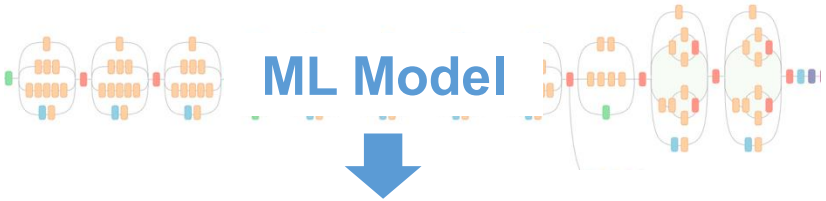
Automated search for performant programs:

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



* Slides from Tianqi Chen

An Overview of Deep Learning Systems



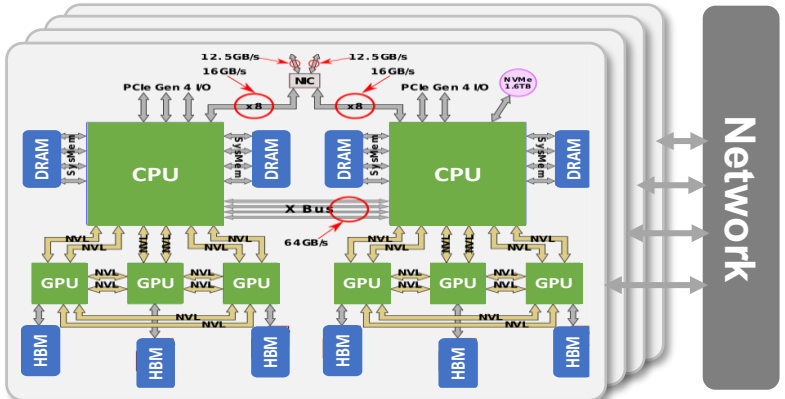
Automatic Differentiation

Graph-Level Optimization

Parallelization / Distributed Training

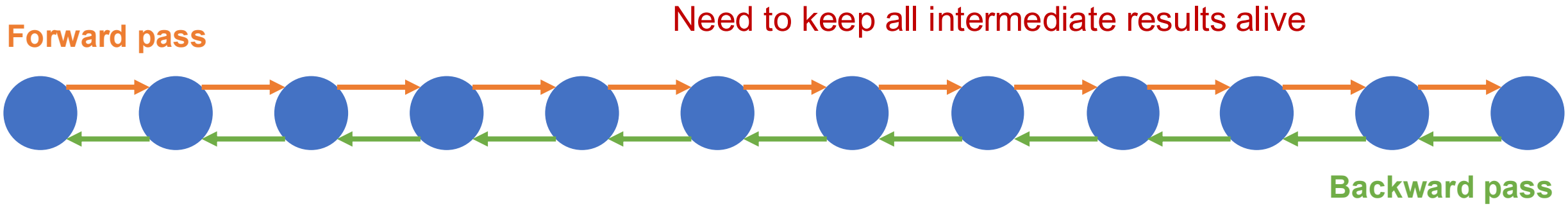
Code Optimization

Memory Optimization

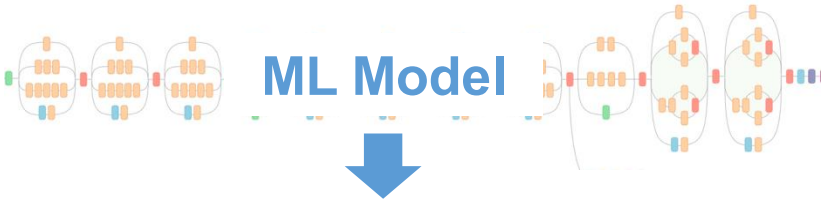


GPU Memory is the Bottleneck in DNN Training

- 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