GPipe
Jun Huang
CSE5449 AU21
How do we train models that cannot be put into single GPU?
import torch
import torch.nn as nn
import torch.optim as optim

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to('cuda:1')

    def forward(self, x):
        x = self.relu(self.net1(x.to('cuda:0')))
        return self.net2(x.to('cuda:1'))

Reference:
https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html
Before Gpipe:

- Node 1
- Node 2
- Node 3

*Step 1:* Data

*Step 2:* Activation

*Step 3:* Activation
Gpipe forward (pipelining):

Data #1

step1

Data #2

step2

Data #3

step3

Activation #1

Activation #2

Activation #1
Gpipe pipeline overview:

Device #4
Device #3
Device #2
Device #1

Before Gpipe

Device #4
Device #3
Device #2
Device #1

Gpipe
Improve over Naïve solution:

1. Less bubble / less waste of computation cycle

2. Closer to linear scale (comparing to 1 node, n nodes achieve 1/n run time), naïve solution: more node, the same run time (even worst, because introduced communication overhead)
Why previous research can not figure such a simple solution?

1. Limited attention

2. Google has the need to create large models, leading position in other research (CV/NLP) create the needs.
Limitation:

1. All forwards + all backwards is not an optimal schedule.
2. Splitting batch into micro batches lead to potential low computation efficiency (more micro-batches mean less bubble).
3. Memory overhead, larger amount batches lead to less bubble.
Gshard

Jun Huang

CSE5449 AU21
Gshard paper organization:
1. System side
2. Model side
History of Gshard
1. mesh-tensorflow
2. XLA
3. Gshard
Mesh-Tensorflow

1. Independent framework that provides distributed strategy space to users.
2. Only run on TPU
3. Not compatible with Tensorflow

Reference: https://github.com/tensorflow/mesh
Model definition:

```python
# tf_images is a tf.Tensor with shape [100, 28, 28] and dtype tf.float32
# tf_labels is a tf.Tensor with shape [100] and dtype tf.int32
graph = mtf.Graph()
mesh = mtf.Mesh(graph, "my_mesh")
batch_dim = mtf.Dimension("batch", 100)
rows_dim = mtf.Dimension("rows", 28)
cols_dim = mtf.Dimension("cols", 28)
hidden_dim = mtf.Dimension("hidden", 1024)
classes_dim = mtf.Dimension("classes", 10)
images = mtf.import_tf_tensor(
    mesh, tf_images, shape=[batch_dim, rows_dim, cols_dim])
labels = mtf.import_tf_tensor(mesh, tf_labels, [batch_dim])
w1 = mtf.get_variable(mesh, "w1", [rows_dim, cols_dim, hidden_dim])
w2 = mtf.get_variable(mesh, "w2", [hidden_dim, classes_dim])
# einsum is a generalization of matrix multiplication (see numpy.einsum)
hidden = mtf.relu(mtf.einsum(images, w1, output_shape=[batch_dim, hidden_dim]))
logits = mtf.einsum(hidden, w2, output_shape=[batch_dim, classes_dim])
loss = mtf.reduce_mean(mtf.layers.softmax_cross_entropy_with_logits(
    logits, mtf.one_hot(labels, classes_dim), classes_dim))
w1_grad, w2_grad = mtf.gradients([loss], [w1, w2])
update_w1_op = mtf.assign(w1, w1 - w1_grad * 0.001)
update_w2_op = mtf.assign(w2, w2 - w2_grad * 0.001)
```
Model placement:

```python
devices = ["gpu:0", "gpu:1", "gpu:2", "gpu:3"]
mesh_shape = ["all_processors", 4]
layout_rules = ["batch", "all_processors"]
mesh_impl = mtf.placement_mesh_impl.PlasemntMeshImpl(
    mesh_shape, layout_rules, devices)
lowering = mtf.Lowering(graph, {mesh: mesh_impl})
tf_update_ops = [lowering.lowered_operation(update_w1_op),
                 lowering.lowered_operation(update_w2_op)]
```
XLA

1. Compiler extension to Tensorflow

2. Integrate mesh-tensorflow functionality into Tensorflow

3. And many more functionalities.
Gshard system

1. Using XLA to provide three type of decorator to users.

2. Replicate, Split, Shard.
GShard lightweight annotation API and XLA compiler extensions.

```python
# Partition inputs along the first (group G) dim across D devices.
+ inputs = split(inputs, 0, D)

# Replicate the gating weights across all devices
+ wg = replicate(wg)

gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
combine_weights, dispatch_mask = Top2Gating(gates)
dispatched_inputs = einsum("GSEC,GSM->EGCM", dispatch_mask, inputs)

# Partition dispatched inputs along expert (E) dim.
+ dispatched_inputs = split(dispatched_inputs, 0, D)

h = einsum("EGCM,EMH->EGCH", dispatched_inputs, wi)
```

Minimal sharding annotations are needed to train models on thousands of accelerators. XLA compiler ensures efficiency.

No need to define model placement.

No need to change training process.
MoE (Mixture of Expert)

1. Propose a question: Why large model provides higher accuracy? Is it only because it has larger model size or because of the larger computation?

2. What if we only increase the model size, but keep the computation the same, will it give us higher accuracy?
Solutions to keep computation the same while model size gets larger:

1. Sparsity in computation

2. Ensemble idea
Baseline (1 GPU):

Tensor model parallelism paradigm (3*parameters, Computation = 3*Baseline)

Moe model paradigm (3*parameter, Computation = Baseline)
Intuition:

1. certain words should be processed by certain “expert” (dog/cat should be processed by animal-word “expert”).

2. Sparsity among computation. Each word will only be processed by a partial model, which achieves the goal that increasing the model size but the computation stays the same.
### Results:

<table>
<thead>
<tr>
<th>Id</th>
<th>Model</th>
<th>Cores</th>
<th>Steps per sec.</th>
<th>Batch sz. (Tokens)</th>
<th>TPU core years</th>
<th>Training time (days)</th>
<th>BLEU avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>MoE(2048E, 36L)</td>
<td>2048</td>
<td>0.72</td>
<td>4M</td>
<td>22.4</td>
<td>4.0</td>
<td>44.3</td>
</tr>
<tr>
<td>(2)</td>
<td>MoE(2048E, 12L)</td>
<td>2048</td>
<td>2.15</td>
<td>4M</td>
<td>7.5</td>
<td>1.4</td>
<td>41.3</td>
</tr>
<tr>
<td>(3)</td>
<td>MoE(512E, 36L)</td>
<td>512</td>
<td>1.05</td>
<td>1M</td>
<td>15.5</td>
<td>11.0</td>
<td>43.7</td>
</tr>
<tr>
<td>(4)</td>
<td>MoE(512E, 12L)</td>
<td>512</td>
<td>3.28</td>
<td>1M</td>
<td>4.9</td>
<td>3.5</td>
<td>40.0</td>
</tr>
<tr>
<td>(5)</td>
<td>MoE(128E, 36L)</td>
<td>128</td>
<td>0.67</td>
<td>1M</td>
<td>6.1</td>
<td>17.3</td>
<td>39.0</td>
</tr>
<tr>
<td>(6)</td>
<td>MoE(128E, 12L)</td>
<td>128</td>
<td>2.16</td>
<td>1M</td>
<td>1.9</td>
<td>5.4</td>
<td>36.7</td>
</tr>
<tr>
<td>*</td>
<td>T(96L)</td>
<td>2048</td>
<td>-</td>
<td>4M</td>
<td>~235.5</td>
<td>~42</td>
<td>36.9</td>
</tr>
</tbody>
</table>

Table 3: Performance of MoE models with different number of experts and layers.

V-Moe, Vit-Moe, Switch transformer take leadership in many task.
Insights:

1. Propose a really good question.

2. This paper is aimed to make more people use TPU (TPU unlike GPU, using 2D interconnect, MoE heavily rely on All-to-All communication, GPU software/hardware haven’t had a good enough solution for All-to-All)
Reference: