Mesh TensorFlow

Running models of sufficiently large size pose practical difficulties

- taking advantage of the inherent parallelism afforded by multiple computational units (processors, cores)
- insufficient memory per processor to contain a model's parameters

Fortunately, there are programming abstractions that mitigate these difficulties.

Mesh-TensorFlow (https://arxiv.org/pdf/1811.02084.pdf) is an extension of TensorFlow that provides a clean API for dealing with various forms of parallelism.

We provide a very brief introduction

- motivated by its use in implementing the MoE in the FFN of the Switch Transformer

Note that this is an extension of "low-level" TensorFlow, **not** higher level Keras.

Forms of parallelism

We assume

- a collection of computational units (referred to as *processors*)
- that are able to communicate with one another
    - by a arbitary communication fabric

We want our models to be able to take advantage of the multiple processors.

The two common forms of parallelism are

- data parallelism
    - each example in a mini-batch is independent
    - split the batch across processors
- model parallelism
    - when a model's parameters are too large to fit into memory of a single processor
    - split the parameters (and computation) across multiple processors

To illustrate: Consider

- A batch $\mathbf{X}$ of m examples, each a one-dimensional vector of length n: $\mathbf{X} \in \mathbb{R}^{m \times n}$
- A vector $\mathbf{w}$ of length n: $\mathbf{w} \in \mathbb{R}^n$
    - e.g., one row of a weight matrix $\mathbf{W}$ implementing the `Dense` operation of $\mathbf{X} * \mathbf{W}$
- We want to compute the dot product of every example with $\mathbf{w}$
  $$\mathbf{X}^{(\mathbf{i})} \cdot \mathbf{w}$$
  for every example i

Data parallelism

This is the simplest form of parallelism

- split the examples into groups

$$\mathbf{X} = \begin{pmatrix} \mathbf{X}^{(0:g-1)} \\ \mathbf{X}^{(g:2*g-1)} \\ \vdots \\ X^{(m-g:m-1)} \end{pmatrix}$$

Dispatch

- a single group (e.g., $\mathbf{X}^{(s:e)}$) to a single processor
- the weights $\mathbf{w}$ to each processor

Each processor (e.g., the one assigned examples $\mathbf{X}^{(s:e)}$) computes

$$\mathbf{X}^{(s:e)} * \mathbf{w}$$

n.b., multiplication of matrix and vector results in vector output

- that is dot product of each row of matrix and the right vector

Model parallelism

Suppose dimension n is so large that a single processor's memory cannot accommodate either vector

- corresponding to a single example $\mathbf{X}^{(\mathbf{i})}$
- corresponding to $\mathbf{w}$

We will split each vector into (horizontal) groups

$$\begin{aligned}
\mathbf{X}^{(\mathbf{i})} &= [\mathbf{X}^{(\mathbf{i})}_{0:g-1}, \mathbf{X}^{(\mathbf{i})}_{g,2*g-1}, \ldots, \mathbf{X}^{(\mathbf{i})}_{n-g,n-1}] \\
\mathbf{w} &= [\mathbf{w}_{0:g-1}, \mathbf{w}_{g,2*g-1}, \ldots, \mathbf{w}_{n-g,n-1}]
\end{aligned}$$

Dispatch a group

- $\mathbf{X}^{(\mathbf{i})}_{s:e}$ and $\mathbf{w}_{s:e}$ to a single processor p which computes the dot product
$$h^{(p)} = \mathbf{X}^{(\mathbf{i})}_{s:e} \cdot \mathbf{w}_{s:e}$$

Note that $h^{(p)}$ is a scalar.

Gather (using the communication network) the $h_{(p)}$ from all processors p
$$\mathbf{h} = [h^{(0)}, h^{(1)}, \ldots, h^{(\frac{n}{g})}]$$

Reduce the vector $\mathbf{h}$ to a scalar

- by summing over its elements

Backward pass

We have illustrated the fundamental ideas of parallelism

- using a computation from the forward pass of a Neural Network

The backward pass (gradient flow) can also be implemented

- but requires Gather and Reduce operations

Defining parallelism in Mesh-TensorFlow

As per our illustration, implementing the two forms of parallelism involves

- splitting Tensors
- and possibly gathering/reducing sub-results

Mesh-TensorFlow provides a simple notation for describing how to split a Tensor

- everything else (dispatching, gathering, reducing) is automatic

A Tensor has multiple dimensions.

In Mesh-TensorFlow, we can give each dimension a *name* and a size.

For example, the first dimension of most Tensors in a Neural Network is the "batch" dimension.

- We can specify a dimension named "batch", of size 100:

  ```
  batch_dim = mtf.Dimension("batch", 100)
  ```

Mesh-TensorFlow also allows the user to specify

- The logical (not physical connectivity) organization of processors.
    - Treating 4 processors as a one-dimensional vector

      ```
      mesh_shape = [("all_processors", 4)]
      ```

- *Layout* rules: how a named dimension is split into groups
    - To specify data parallelism (split batch across processors)

      ```
      layout_rules = [("batch", "all_processors")]
      ```

Mesh-TensorFlow thus provides a simple but powerful API

- for distributing data and computation
- across multiple processors

We illustrate with an example taken from the Mesh TensorFlow github (https://github.com/tensorflow/mesh#example-network-mnist)

Mesh-TensorFlow program for MNIST classification task (https://github.com/tensorflow/mesh#example-network-mnist)

Name the dimensions

```
# 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)
```

Compute logits loss, and update weights via Gradient Descent

```
    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_di
m]))
    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)
```

Specify mesh of processors and layout of computation

Layout for data parallelism

The following layout implements *data parallelism*

- Any Tensor with a dimension named "batch" dimension
    - `images, h, logits` and their gradients
- is split across all devices ("all_processors")

```
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.PlacementMeshImpl(
    mesh_shape, layout_rules, devices)
```

Layout for model parallelism

Alternatively, we can use a layout to implement *model parallelism*

- Any Tensor with a dimension named `hidden_dim`
    - `hidden, w1, w2`
- is split across all devices ("all_processors")

```
layout_rules=[("hidden_dim", "all_processors")]
```

Layout for data and model parallelism

We create a 2D mesh of processors

- rows are named `processor_rows`, columns are named `processor_cols`

  ```
  mesh_shape = [("processor_rows", 2), ("processor_cols", 2)]
  ```

And use the layout

```
layout_rules = [("batch", "processor_rows"), ("hidden", "processor_cols")]
```

Layout splits

- Tensors with a "batch" dimension across the processor rows (data parallelism)
    - replicating them for every processor column
- Tensors with a "hidden" dimension across processor columns (model parallelism)
    - replicating them for every processor row layout_rules = [("batch", "processor_rows"), ("hidden", "processor_cols")]

So the processors

- in first row has half the examples
- in the second row has half the examples
- in the first column has half the weights
- in the second column has half the weights

So each quadrant (one processor)

- computes the dot product of half the examples on half the weights
- for the `hidden` tensor
    - has **both** "batch" and "hidden" dimensions
    - so is distributed across all 4 quadrants of the mesh

These are combined into a single result batch via a reduction

- `allreduce` operation
- communication across processors
    - or partially reduced dot products (i.e., scalars)

”Lower” the logical layout unto the physical devices

```
lowering = mtf.Lowering(graph, {mesh:mesh_impl})

tf_update_ops = [lowering.lowered_operation(update_w1_op),
                 lowering.lowered_operation(update_w2_op)]
```

Illustrations of layouts

Appendix A (https://arxiv.org/pdf/1811.02084.pdf#page=11) provides nice illustrations

- of layouts
- for an example similar to the MNIST program

Notation

- Tensors are split by vertical and horizontal lines
- Blue integers indicate which processor a piece of a Tensor has been dispatched to
    - some pieces are dispatched to multiple processors

Data parallelism

Data parallelism

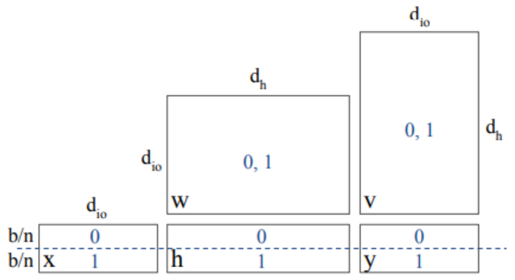

Figure 2: The data-parallel layout for the Two Fully-Connected Layers example, with $n = 2$ processors $\in \{0, 1\}$. Blue numbers on matrices indicate the ranks of the processors the matrix slices reside on. The *batch* dimension is split among all processors. w and v are fully replicated.

Attribution: https://arxiv.org/pdf/1811.02084.pdf#page=12

- $\mathbf{X}$ (and result $\mathbf{h} = \mathbf{X} * \mathbf{W}$) are split along the batch dimension
    - groups assigned to processors $0, 1$
- $\mathbf{W}$ is *not split*
    - full $\mathbf{W}$ is replicated across processors $0, 1$

Thus, processors $0, 1$ are able to compute their subset of $\mathbf{X}$, times $\mathbf{W}$

- resulting $\mathbf{h}$ is split (along batch dimension) across processors $0, 1$

Model parallelism

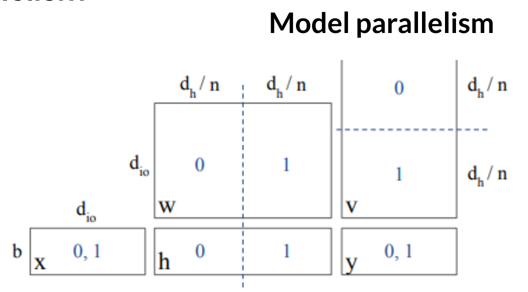

Figure 3: The model-parallel layout for the Two Fully-Connected Layers example, with $n = 2$ processors $\in \{0, 1\}$. Blue numbers on matrices indicate the ranks of the processors the matrix slices reside on. The *hidden* dimension is split among all processors. x and y are fully replicated.

Attribution: https://arxiv.org/pdf/1811.02084.pdf#page=12

- $\mathbf{X}$ is not split
    - replicated to processors $0, 1$
- $\mathbf{W}$ is split (across columns)
    - groups assigned to processors $0, 1$

Thus

- processor 0 can compute the full $\mathbf{X}$ times the first group of $\mathbf{W}$
- processor 1 can compute the full $\mathbf{X}$ times the second group of $\mathbf{W}$
- resulting in $\mathbf{h}$ split across processors $0, 1$

Data and Model parallelism

Here, 4 processors are arranged into a 2×2 mesh.

Data and Model parallelism

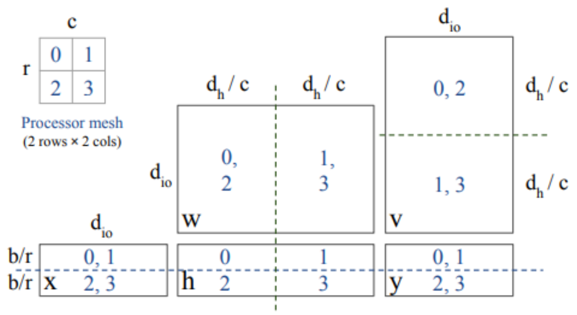

Figure 4: The mixed data-and-model-parallel layout for the Two Fully-Connected Layers example. There are 4 processors, arranged into a 2-by-2 mesh. Each processor is assigned a serialized rank which is used to label matrix slices that it owns.

Attribution: https://arxiv.org/pdf/1811.02084.pdf#page=12

$\mathbf{X}$ (and result $\mathbf{h} = \mathbf{X} * \mathbf{W}$) are split along the batch dimension

- one group is replicated
  - assigned to processors $0, 1$
- second group is replicated
  - assigned to processors $2, 3$
- $\mathbf{W}$ is split (across columns)
- one group is replicated
  - assigned to processors $0, 2$
- second group is replicated
  - assigned to processors (1, 3)

Thus

- processor 0 can compute first group of $\mathbf{X}$ times first group of $\mathbf{W}$
- processor 1 can compute first group of $\mathbf{X}$ times second group of $\mathbf{W}$
- processor 2 can compute second group of $\mathbf{X}$ times first group of $\mathbf{W}$
- processor 3 can compute second group of $\mathbf{X}$ times second group of $\mathbf{W}$

Note that result $\mathbf{h}$ is split across **both**

- batch dimension (due to the data parallel split of $\mathbf{X}$)
- model dimension (due to model parallel split of $\mathbf{W}$)

In [2]:
```python
print("Done")
```

```
Done
```