Shapes and Layout

The XLA **Shape** proto (xla_data.proto (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/xla_data.proto)) describes the rank, size, and data type of an N-dimensional array (*array* in short).

Terminology, Notation, and Conventions

- The rank of an array is equal to the number of dimensions. The *true rank* of an array is the number of dimensions which have a size greater than 1.
- Dimensions are numbered from `0` up to `N-1` for an `N` dimensional array. The dimension numbers are arbitrary labels for convenience. The order of these dimension numbers does not imply a particular minor/major ordering in the layout of the shape. The layout is determined by the `Layout` proto.
- By convention, dimensions are listed in increasing order of dimension number. For example, for a 3-dimensional array of size `[A x B x C]`, dimension 0 has size `A`, dimension 1 has size `B` and dimension 2 has size `C`.

  Some utilities in XLA also support negative indexing, similarly to Python; dimension -1 is the last dimension (equivalent to `N-1` for an `N` dimensional array). For example, for the 3-dimensional array described above, dimension -1 has size `C`, dimension -2 has size `B` and so on.
- Two, three, and four dimensional arrays often have specific letters associated with dimensions. For example, for a 2D array:
  - dimension 0: `y`
  - dimension 1: `x`

  For a 3D array:
  - dimension 0: `z`
  - dimension 1: `y`
  - dimension 2: `x`

  For a 4D array:
  - dimension 0: `p`
  - dimension 1: `z`
  - dimension 2: `y`
  - dimension 3: `x`
- Functions in the XLA API which take dimensions do so in increasing order of dimension number. This matches the ordering used when passing dimensions as an `initializer_list`; e.g.

  `ShapeUtil::MakeShape(F32, {A, B, C, D})`

  Will create a shape whose dimension size array consists of the sequence `[A, B, C, D]`.

Layout

The `Layout` proto describes how an array is represented in memory. The `Layout` proto includes the following fields:

```
message Layout {
  repeated int64 minor_to_major = 1;
  repeated int64 padded_dimensions = 2;
  optional PaddingValue padding_value = 3;
}
```

Minor-to-major dimension ordering

The only required field is `minor_to_major`. This field describes the minor-to-major ordering of the dimensions within a shape. Values in `minor_to_major` are an ordering of the dimensions of the array (`0` to `N-1` for an `N` dimensional array) with the first value being the most-minor dimension up to the last value which is the most-major dimension. The most-minor dimension is the dimension which changes most rapidly when stepping through the elements of the array laid out in linear memory.

For example, consider the following 2D array of size `[2 x 3]`:

```
a b c
d e f
```

Here dimension `0` is size 2, and dimension `1` is size 3. If the `minor_to_major` field in the layout is `[0, 1]` then dimension `0` is the most-minor dimension and dimension `1` is the most-major dimension. This corresponds to the following layout in linear memory:

```
a d b e c f
```

This minor-to-major dimension order of `0` up to `N-1` is akin to *column-major* (at rank 2). Assuming a monotonic ordering of dimensions, another name we may use to refer to this layout in the code is simply "dim 0 is minor".

On the other hand, if the `minor_to_major` field in the layout is `[1, 0]` then the layout in linear memory is:

```
a b c d e f
```

A minor-to-major dimension order of `N-1` down to `0` for an `N` dimensional array is akin to *row-major* (at rank 2). Assuming a monotonic ordering of dimensions, another name we may use to refer to this layout in the code is simply "dim 0 is major".

Default minor-to-major ordering

The default layout for newly created Shapes is "dimension order is major-to-minor" (akin to row-major at rank 2).

Padding

Padding is defined in the optional `padded_dimensions` and `padding_value` fields. The field `padded_dimensions` describes the sizes (widths) to which each dimension is padded. If present, the number of elements in `padded_dimensions` must equal the rank of the shape.

For example, given the `[2 x 3]` array defined above, if `padded_dimension` is `[3, 5]` then dimension 0 is padded to a width of 3 and dimension 1 is padded to a width of 5. The layout in linear memory (assuming a padding value of 0 and column-major layout) is:

```
a d 0 b e 0 c f 0 0 0 0 0 0 0
```

This is equivalent to the layout of the following array with the same minor-to-major dimension order:

```
a b c 0 0
d e f 0 0
0 0 0 0 0
```

Indexing into arrays

The class `IndexUtil` in index_util.h (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/index_util.h) provides utilities for converting between multidimensional indices and linear indices given a shape and layout. Multidimensional indices include a `int64` index for each dimension. Linear indices are a single `int64` value which indexes into the buffer holding the array. See `shape_util.h` and `layout_util.h` in the same directory for utilities that simplify creation and manipulation of shapes and layouts.
