Operation Semantics

The following describes the semantics of operations defined in the `ComputationBuilder` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) interface. Typically, these operations map one-to-one to operations defined in the RPC interface in `xla_data.proto` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/xla_data.proto).

A note on nomenclature: the generalized data type XLA deals with is an N-dimensional array holding elements of some uniform type (such as 32-bit float). Throughout the documentation, *array* is used to denote an arbitrary-dimensional array. For convenience, special cases have more specific and familiar names; for example a *vector* is a 1-dimensional array and a *matrix* is a 2-dimensional array.

Broadcast

See also `ComputationBuilder::Broadcast` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Adds dimensions to an array by duplicating the data in the array.

`Broadcast(operand, broadcast_sizes)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	The array to duplicate
`broadcast_sizes`	`ArraySlice<int64>`	The sizes of the new dimensions

The new dimensions are inserted on the left, i.e. if `broadcast_sizes` has values `{a0, ..., aN}` and the operand shape has dimensions `{b0, ..., bM}` then the shape of the output has dimensions `{a0, ..., aN, b0, ..., bM}`.

The new dimensions index into copies of the operand, i.e.

```
output[i0, ..., iN, j0, ..., jM] = operand[j0, ..., jM]
```

For example, if `operand` is a scalar `f32` with value `2.0f`, and `broadcast_sizes` is `{2, 3}`, then the result will be an array with shape `f32[2, 3]` and all the values in the result will be `2.0f`.

Call

See also `ComputationBuilder::Call` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) .

Invokes a computation with the given arguments.

`Call(computation, args...)`

Arguments	Type	Semantics
`computation`	`Computation`	computation of type `T_0, T_1, ..., T_N -> S` with N parameters of arbitrary type
`args`	sequence of N `ComputationDataHandle`s	N arguments of arbitrary type

The arity and types of the `args` must match the parameters of the `computation`. It is allowed to have no `args`.

Clamp

See also `ComputationBuilder::Clamp` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Clamps an operand to within the range between a minimum and maximum value.

`Clamp(computation, args...)`

Arguments	Type	Semantics
`computation`	`Computation`	computation of type `T_0, T_1, ..., T_N -> S` with N parameters of arbitrary type
`operand`	`ComputationDataHandle`	array of type T
`min`	`ComputationDataHandle`	array of type T
`max`	`ComputationDataHandle`	array of type T

Given an operand and minimum and maximum values, returns the operand if it is in the range between the minimum and maximum, else returns the minimum value if the operand is below this range or the maximum value if the operand is above this range. That is, `clamp(x, a, b) = max(min(x, a), b)`.

All three arrays must be the same shape. Alternately, as a restricted form of broadcasting (https://www.tensorflow.org/performance/xla/broadcasting), `min` and/or `max` can be a scalar of type `T`.

Example with scalar `min` and `max`:

```
let operand: s32[3] = {-1, 5, 9};
let min: s32 = 0;
let max: s32 = 6;
==>
Clamp(operand, min, max) = s32[3]{0, 5, 6};
```

Collapse

See also `ComputationBuilder::Collapse` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) and the `tf.reshape` (https://www.tensorflow.org/api_docs/python/tf/reshape) operation.

Collapses dimensions of an array into one dimension.

`Collapse(operand, dimensions)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	array of type T
`dimensions`	`int64` vector	in-order, consecutive subset of T's dimensions.

Collapse replaces the given subset of the operand's dimensions by a single dimension. The input arguments are an arbitrary array of type T and a compile-time-constant vector of dimension indices. The dimension indices must be an in-order (low to high dimension numbers), consecutive subset of T's dimensions. Thus, {0, 1, 2}, {0, 1}, or {1, 2} are all valid dimension sets, but {1, 0} or {0, 2} are not. They are replaced by a single new dimension, in the same position in the dimension sequence as those they replace, with the new dimension size equal to the product of original dimension sizes. The lowest dimension number in `dimensions` is the slowest varying dimension (most major) in the loop nest which collapses these dimension, and the highest dimension number is fastest varying (most minor). See the `tf.reshape` (https://www.tensorflow.org/api_docs/python/tf/reshape) operator if more general collapse ordering is needed.

For example, let v be an array of 24 elements:

```
let v = f32[4x2x3] { { {10, 11, 12},  {15, 16, 17}},
                    { {20, 21, 22},  {25, 26, 27}},
                    { {30, 31, 32},  {35, 36, 37}},
                    { {40, 41, 42},  {45, 46, 47}}};

// Collapse to a single dimension, leaving one dimension.
let v012 = Collapse(v, {0,1,2});
then v012 == f32[24] {10, 11, 12, 15, 16, 17,
                      20, 21, 22, 25, 26, 27,
                      30, 31, 32, 35, 36, 37,
```

```
                      40, 41, 42, 45, 46, 47};

// Collapse the two lower dimensions, leaving two dimensions.
let v01 = Collapse(v, {0,1});
then v01 == f32[4x6] { {10, 11, 12, 15, 16, 17},
                      {20, 21, 22, 25, 26, 27},
                      {30, 31, 32, 35, 36, 37},
                      {40, 41, 42, 45, 46, 47}};

// Collapse the two higher dimensions, leaving two dimensions.
let v12 = Collapse(v, {1,2});
then v12 == f32[8x3] { {10, 11, 12},
                      {15, 16, 17},
                      {20, 21, 22},
                      {25, 26, 27},
                      {30, 31, 32},
                      {35, 36, 37},
                      {40, 41, 42},
                      {45, 46, 47}};
```

Concatenate

See also **`ComputationBuilder::ConcatInDim`** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Concatenate composes an array from multiple array operands. The array is of the same rank as each of the input array operands (which must be of the same rank as each other) and contains the arguments in the order that they were specified.

`Concatenate(operands..., dimension)`

Arguments	Type	Semantics
`operands`	sequence of N `ComputationDataHandle`	N arrays of type T with dimensions [L0, L1, ...]. Requires N >= 1.
`dimension`	`int64`	A value in the interval `[0, N)` that names the dimension to be concatenated between the `operands`.

With the exception of `dimension` all dimensions must be the same. This is because XLA does not support "ragged" arrays Also note that rank-0 values cannot be concatenated (as it's impossible to name the dimension along which the concatenation occurs).

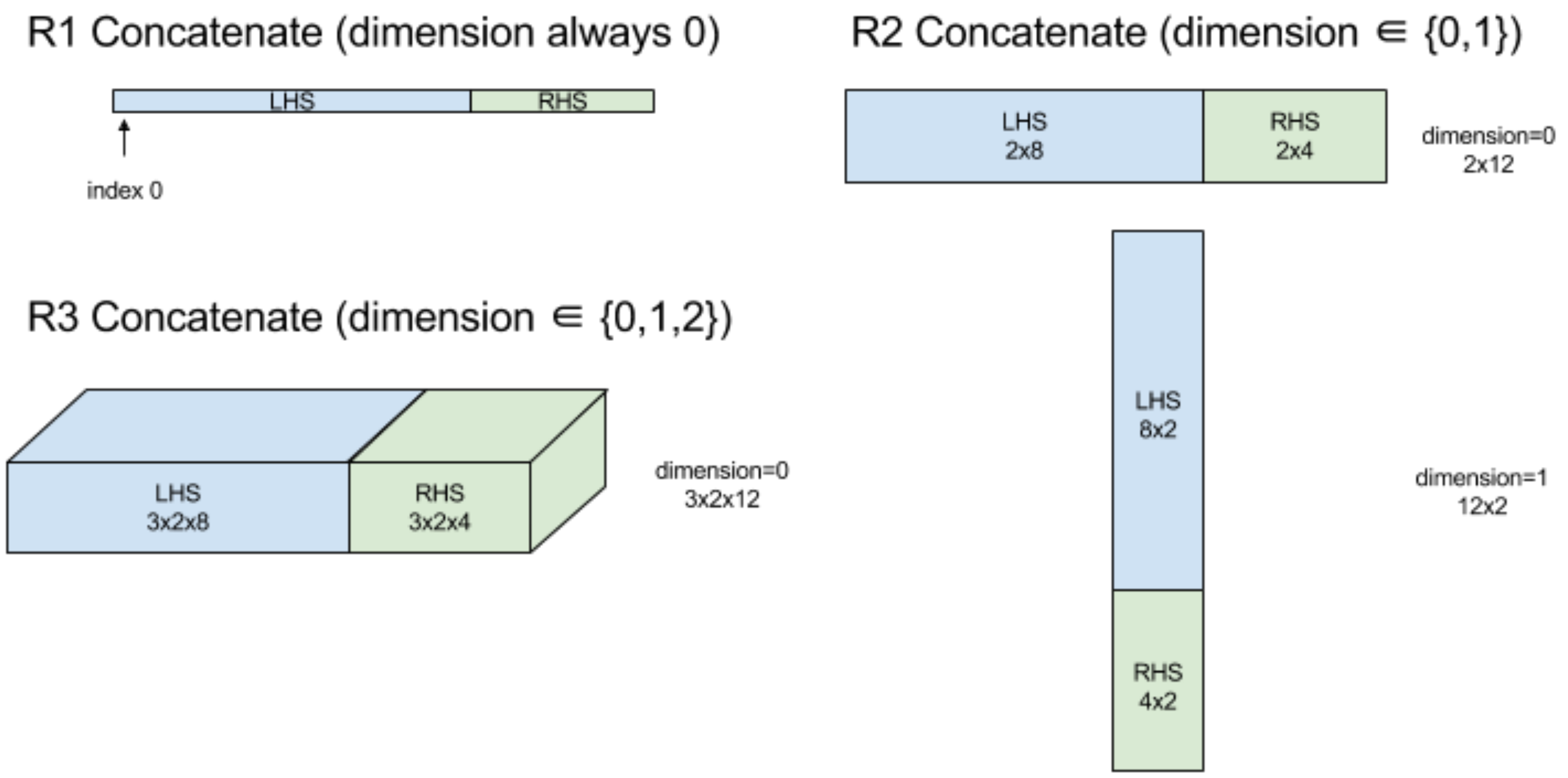
1-dimensional example:

```
Concat({ {2, 3}, {4, 5}, {6, 7}}, 0)
>>> {2, 3, 4, 5, 6, 7}
```

2-dimensional example:

```
let a = {
  {1, 2},
  {3, 4},
  {5, 6},
};
let b = {
  {7, 8},
};
Concat({a, b}, 0)
>>> {
  {1, 2},
  {3, 4},
  {5, 6},
  {7, 8},
}
```

Diagram:



ConvertElementType

See also **ComputationBuilder::ConvertElementType** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Similar to an element-wise `static_cast` in C++, performs an element-wise conversion operation from a data shape to a target shape. The dimensions must match, and the conversion is an element-wise one; e.g. `s32` elements become `f32` elements via an `s32`-to-`f32` conversion routine.

`ConvertElementType(operand, new_element_type)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	array of type T with dims D
`new_element_type`	`PrimitiveType`	type U

If the dimensions of the operand and the target shape do not match, or an invalid conversion is requested (e.g. to/from a tuple) an error will be produced.

A conversion such as `T=s32` to `U=f32` will perform a normalizing int-to-float conversion routine such as round-to-nearest-even.

Note: The precise float-to-int and visa-versa conversions are currently unspecified, but may become additional arguments to the convert operation in the future. Not all possible conversions have been implemented for all targets.

```
let a: s32[3] = {0, 1, 2};
let b: f32[3] = convert(a, f32);
then b == f32[3]{0.0, 1.0, 2.0}
```

Conv (convolution)

See also **ComputationBuilder::Conv** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h)
.

As ConvWithGeneralPadding, but the padding is specified in a short-hand way as either SAME or VALID. SAME padding pads the input (`lhs`) with zeroes so that the output has the same shape as the input when not taking striding into account. VALID padding simply means no padding.

ConvWithGeneralPadding (convolution)

See also `ComputationBuilder::ConvWithGeneralPadding` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Computes a convolution of the kind used in neural networks. Here, a convolution can be thought of as a n-dimensional window moving across a n-dimensional base area and a computation is performed for each possible position of the window.

Arguments	Type	Semantics
`lhs`	`ComputationDataHandle`	rank n+2 array of inputs
`rhs`	`ComputationDataHandle`	rank n+2 array of kernel weights
`window_strides`	`ArraySlice<int64>`	n-d array of kernel strides
`padding`	`ArraySlice<pair<int64, int64>>`	n-d array of (low, high) padding
`lhs_dilation`	`ArraySlice<int64>`	n-d lhs dilation factor array
`rhs_dilation`	`ArraySlice<int64>`	n-d rhs dilation factor array

Let n be the number of spatial dimensions. The `lhs` argument is a rank n+2 array describing the base area. This is called the input, even though of course the rhs is also an input. In a neural network, these are the input activations. The n+2 dimensions are, in this order:

- `batch`: Each coordinate in this dimension represents an independent input for which convolution is carried out.
- `z/depth/features`: Each (y,x) position in the base area has a vector associated to it, which goes into this dimension.
- `spatial_dims`: Describes the `n` spatial dimensions that define the base area that the window moves across.

The `rhs` argument is a rank n+2 array describing the convolutional filter/kernel/window. The dimensions are, in this order:

- `output-z`: The `z` dimension of the output.
- `input-z`: The size of this dimension should equal the size of the `z` dimension in lhs.
- `spatial_dims`: Describes the `n` spatial dimensions that define the n-d window that moves across the base area.

The `window_strides` argument specifies the stride of the convolutional window in the spatial dimensions. For example, if the stride in a the first spatial dimension is 3, then the window can only be placed at coordinates where the first spatial index is divisible by 3.

The `padding` argument specifies the amount of zero padding to be applied to the base area. The amount of padding can be negative -- the absolute value of negative padding indicates the number of elements to remove from the specified dimension before doing the convolution. `padding[0]` specifies the padding for dimension `y` and `padding[1]` specifies the padding for dimension `x`. Each pair has the low padding as the first element and the high padding as the second element. The low padding is applied in the direction of lower indices while the high padding is applied in the direction of higher indices. For example, if `padding[1]` is `(2,3)` then there will be a padding by 2 zeroes on the left and by 3 zeroes on the right in the second spatial dimension. Using padding is equivalent to inserting those same zero values into the input (`lhs`) before doing the convolution.

The `lhs_dilation` and `rhs_dilation` arguments specify the dilation factor to be applied to the lhs and rhs, respectively, in each spatial dimension. If the dilation factor in a spatial dimension is d, then d-1 holes are implicitly placed between each of the entries in that dimension, increasing the size of the array. The holes are filled with a no-op value, which for convolution means zeroes.

Dilation of the rhs is also called atrous convolution. For more details, see the `tf.nn.atrous_conv2d` (https://www.tensorflow.org/api_docs/python/tf/nn/atrous_conv2d). Dilation of the lhs is also called deconvolution.

The output shape has these dimensions, in this order:

- `batch`: Same size as `batch` on the input (`lhs`).
- `z`: Same size as `output-z` on the kernel (`rhs`).
- `spatial_dims`: One value for each valid placement of the convolutional window.

The valid placements of the convolutional window are determined by the strides and the size of the base area after padding.

To describe what a convolution does, consider a 2d convolution, and pick some fixed `batch`, `z`, `y`, `x` coordinates in the output. Then `(y,x)` is a position of a corner of the window within the base area (e.g. the upper left corner, depending on how you interpret the spatial dimensions). We now have a 2d window, taken from the base area, where each 2d point is associated to a 1d vector, so we get a 3d box.

From the convolutional kernel, since we fixed the output coordinate `z`, we also have a 3d box. The two boxes have the same dimensions, so we can take the sum of the element-wise products between the two boxes (similar to a dot product). That is the output value.

Note that if `output-z` is e.g., 5, then each position of the window produces 5 values in the output into the `z` dimension of the output. These values differ in what part of the convolutional kernel is used - there is a separate 3d box of values used for each `output-z` coordinate. So you could think of it as 5 separate convolutions with a different filter for each of them.

Here is pseudo-code for a 2d convolution with padding and striding:

```
for (b, oz, oy, ox) {  // output coordinates
  value = 0;
  for (iz, ky, kx) {  // kernel coordinates and input z
    iy = oy*stride_y + ky - pad_low_y;
    ix = ox*stride_x + kx - pad_low_x;
    if ((iy, ix) inside the base area considered without padding) {
      value += input(b, iz, iy, ix) * kernel(oz, iz, ky, kx);
    }
  }
  output(b, oz, oy, ox) = value;
}
```

CrossReplicaSum

See also `ComputationBuilder::CrossReplicaSum` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Computes a sum across replicas.

`CrossReplicaSum(operand)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	Array to sum across replicas.

The output shape is the same as the input shape. For example, if there are two replicas and the operand has the value `(1.0, 2.5)` and `(3.0, 5.1)` respectively on the two replicas, then the output value from this op will be `(4.0, 7.6)` on both replicas.

Computing the result of CrossReplicaSum requires having one input from each replica, so if one replica executes a CrossReplicaSum node more times than another, then the former replica will wait forever. Since the replicas are all running the same program, there are not a lot of ways for that to happen, but it is possible when a while loop's condition depends on data from infeed and the data that is infed causes the while loop to iterate more times on one replica than another.

CustomCall

See also `ComputationBuilder::CustomCall` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Call a user-provided function within a computation.

`CustomCall(target_name, args..., shape)`

Arguments	Type	Semantics
`target_name`	`string`	Name of the function. A call instruction will be emitted which targets this symbol name.
`args`	sequence of N `ComputationDataHandle`s	N arguments of arbitrary type, which will be passed to the function.
`shape`	`Shape`	Output shape of the function

The function signature is the same, regardless of the arity or type of args:

```
extern "C" void target_name(void* out, void** in);
```

For example, if CustomCall is used as follows:

```
let x = f32[2] {1,2};
let y = f32[2x3] { {10, 20, 30}, {40, 50, 60}};

CustomCall("myfunc", {x, y}, f32[3x3])
```

Here is an example of an implementation of `myfunc`:

```
extern "C" void myfunc(void* out, void** in) {
  float (&x)[2] = *static_cast<float(*)[2]>(in[0]);
  float (&y)[2][3] = *static_cast<float(*)[2][3]>(in[1]);
  EXPECT_EQ(1, x[0]);
  EXPECT_EQ(2, x[1]);
  EXPECT_EQ(10, y[0][0]);
  EXPECT_EQ(20, y[0][1]);
  EXPECT_EQ(30, y[0][2]);
  EXPECT_EQ(40, y[1][0]);
  EXPECT_EQ(50, y[1][1]);
  EXPECT_EQ(60, y[1][2]);
  float (&z)[3][3] = *static_cast<float(*)[3][3]>(out);
  z[0][0] = x[1] + y[1][0];
  // ...
}
```

The user-provided function must not have side-effects and its execution must be idempotent.

Note: The opaque nature of the user-provided function restricts optimization opportunities for the compiler. Try to express your computation in terms of native XLA ops whenever possible; only use CustomCall as a last resort.

Dot

See also `ComputationBuilder::Dot` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

`Dot(lhs, rhs)`

Arguments	Type	Semantics
`lhs`	`ComputationDataHandle`	array of type T
`rhs`	`ComputationDataHandle`	array of type T

The exact semantics of this operation depend on the ranks of the operands:

Input	Output	Semantics
vector [n] **`dot`** vector [n]	scalar	vector dot product
matrix [m x k] **`dot`** vector [k]	vector [m]	matrix-vector multiplication
matrix [m x k] **`dot`** matrix [k x n]	matrix [m x n]	matrix-matrix multiplication

The operation performs sum of products over the last dimension of `lhs` and the one-before-last dimension of `rhs`. These are the "contracted" dimensions. The contracted dimensions of `lhs` and `rhs` must be of the same size. In practice, it can be used to perform dot products between vectors, vector/matrix multiplications or matrix/matrix multiplications.

Element-wise binary arithmetic operations

See also `ComputationBuilder::Add` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

A set of element-wise binary arithmetic operations is supported.

`Op(lhs, rhs)`

Where `Op` is one of `Add` (addition), `Sub` (subtraction), `Mul` (multiplication), `Div` (division), `Rem` (remainder), `Max` (maximum), `Min` (minimum), `LogicalAnd` (logical AND), or `LogicalOr` (logical OR).

Arguments	Type	Semantics
`lhs`	`ComputationDataHandle`	left-hand-side operand: array of type T
`rhs`	`ComputationDataHandle`	right-hand-side operand: array of type T

The arguments' shapes have to be either similar or compatible. See the broadcasting (https://www.tensorflow.org/performance/xla/broadcasting) documentation about what it means for shapes to be compatible. The result of an operation has a shape which is the result of broadcasting the two input arrays. In this variant, operations between arrays of different ranks are *not* supported, unless one of the operands is a scalar.

When `Op` is `Rem`, the sign of the result is taken from the dividend, and the absolute value of the result is always less than the divisor's absolute value.

An alternative variant with different-rank broadcasting support exists for these operations:

`Op(lhs, rhs, broadcast_dimensions)`

Where `Op` is the same as above. This variant of the operation should be used for arithmetic operations between arrays of different ranks (such as adding a matrix to a vector).

The additional `broadcast_dimensions` operand is a slice of integers used to expand the rank of the lower-rank operand up to the rank of the higher-rank operand. `broadcast_dimensions` maps the dimensions of the lower-rank shape to the dimensions of the higher-rank shape. The unmapped dimensions of the expanded shape are filled with dimensions of size one. Degenerate-dimension broadcasting then broadcasts the shapes along these degenerate dimension to equalize the shapes of both operands. The semantics are described in detail on the broadcasting page (https://www.tensorflow.org/performance/xla/broadcasting).

Element-wise comparison operations

See also `ComputationBuilder::Eq` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

A set of standard element-wise binary comparison operations is supported. Note that standard IEEE 754 floating-point comparison semantics apply when comparing floating-point types.

`Op(lhs, rhs)`

Where `Op` is one of `Eq` (equal-to), `Ne` (not equal-to), `Ge` (greater-or-equal-than), `Gt` (greater-than), `Le` (less-or-equal-than), `Le` (less-than).

Arguments	Type	Semantics
`lhs`	`ComputationDataHandle`	left-hand-side operand: array of type T
`rhs`	`ComputationDataHandle`	right-hand-side operand: array of type T

The arguments' shapes have to be either similar or compatible. See the broadcasting (https://www.tensorflow.org/performance/xla/broadcasting) documentation about what it means for shapes to be compatible. The result of an operation has a shape which is the result of broadcasting the two input arrays with the element type `PRED`. In this variant, operations between arrays of different ranks are *not* supported, unless one of the operands is a scalar.

An alternative variant with different-rank broadcasting support exists for these operations:

`Op(lhs, rhs, broadcast_dimensions)`

Where `Op` is the same as above. This variant of the operation should be used for comparison operations between arrays of different ranks (such as adding a matrix to a vector).

The additional `broadcast_dimensions` operand is a slice of integers specifying the dimensions to use for broadcasting the operands. The semantics are described in detail on the broadcasting page (https://www.tensorflow.org/performance/xla/broadcasting).

Element-wise unary functions

ComputationBuilder supports these element-wise unary functions:

`Abs(operand)` Element-wise abs `x -> |x|`.

`Ceil(operand)` Element-wise ceil `x -> ⌈x⌉`.

`Cos(operand)` Element-wise cosine `x -> cos(x)`.

`Exp(operand)` Element-wise natural exponential `x -> e^x`.

`Floor(operand)` Element-wise floor `x -> ⌊x⌋`.

`IsFinite(operand)` Tests whether each element of `operand` is finite, i.e., is not positive or negative infinity, and is not `NaN`. Returns an array of `PRED` values with the same shape as the input, where each element is `true` if and only if the corresponding input element is finite.

`Log(operand)` Element-wise natural logarithm `x -> ln(x)`.

`LogicalNot(operand)` Element-wise logical not `x -> !(x)`.

`Neg(operand)` Element-wise negation `x -> -x`.

`Sign(operand)` Element-wise sign operation `x -> sgn(x)` where

$$\text{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

using the comparison operator of the element type of `operand`.

`Tanh(operand)` Element-wise hyperbolic tangent `x -> tanh(x)`.

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	The operand to the function

The function is applied to each element in the `operand` array, resulting in an array with the same shape. It is allowed for `operand` to be a scalar (rank 0).

BatchNormTraining

See also `ComputationBuilder::BatchNormTraining` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) and the original batch normalization paper (https://arxiv.org/abs/1502.03167) for a detailed description of the algorithm.

Warning: Not implemented on GPU backend yet.

Normalizes an array across batch and spatial dimensions.

`BatchNormTraining(operand, scale, offset, epsilon, feature_index)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	n dimensional array to be normalized
`scale`	`ComputationDataHandle`	1 dimensional array (γ)
`offset`	`ComputationDataHandle`	1 dimensional array (\\(\beta\ )
`epsilon`	`float`	Epsilon value (ϵ)
`feature_index`	`int64`	Index to feature dimension in `operand`

For each feature in the feature dimension (`feature_index` is the index for the feature dimension in `operand`), the operation calculates the mean and variance across all the other dimensions and use the mean and variance to normalize each element in `operand`. If an invalid `feature_index` is passed, an error is produced.

The algorithm goes as follows for each batch in `operand` x that contains `m` elements with `w` and `h` as the size of spatial dimensions (

assuming operand is an 4 dimensional array):

- Calculates batch mean μ_l for each feature l in feature dimension: $\mu_l = \frac{1}{mwh} \sum_{i=1}^m \sum_{j=1}^w \sum_{k=1}^h x_{ijkl}$
- Calculates batch variance σ_l^2: $\sigma_l^2 = \frac{1}{mwh} \sum_{i=1}^m \sum_{j=1}^w \sum_{k=1}^h (x_{ijkl} - \mu_l)^2$
- Normalizes, scales and shifts: $y_{ijkl} = \frac{\gamma_l(x_{ijkl} - \mu_l)}{\sqrt[2]{\sigma_l^2 + \epsilon}} + \beta_l$

The epsilon value, usually a small number, is added to avoid divide-by-zero errors.

The output type is a tuple of three ComputationDataHandles:

Outputs	Type	Semantics
output	ComputationDataHandle	n dimensional array with the same shape as input operand (y)
batch_mean	ComputationDataHandle	1 dimensional array (μ)
batch_var	ComputationDataHandle	1 dimensional array (σ^2)

The batch_mean and batch_var are moments calculated across the batch and spatial dimensions using the formulars above.

BatchNormInference

See also ComputationBuilder::BatchNormInference (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Warning: Not implemented yet.

Normalizes an array across batch and spatial dimensions.

BatchNormInference(operand, scale, offset, mean, variance, epsilon, feature_index)

Arguments	Type	Semantics
operand	ComputationDataHandle	n dimensional array to be normalized
scale	ComputationDataHandle	1 dimensional array
offset	ComputationDataHandle	1 dimensional array
mean	ComputationDataHandle	1 dimensional array
variance	ComputationDataHandle	1 dimensional array
epsilon	float	Epsilon value
feature_index	int64	Index to feature dimension in operand

For each feature in the feature dimension (feature_index is the index for the feature dimension in operand), the operation calculates the mean and variance across all the other dimensions and use the mean and variance to normalize each element in operand. If an invalid feature_index is passed, an error is produced.

BatchNormInference is equivalent to calling BatchNormTraining without computing mean and variance for each batch. It uses the input mean and variance instead as estimated values. The purpose of this op is to reduce latency in inference, hence the name BatchNormInference.

The output is a n dimensional, normalized array with the same shape as input operand.

BatchNormGrad

See also ComputationBuilder::BatchNormGrad (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Warning: Not implemented yet.

Calculates gradients of batch norm.

`BatchNormGrad(operand, scale, mean, variance, grad_output, epsilon, feature_index)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	n dimensional array to be normalized (x)
`scale`	`ComputationDataHandle`	1 dimensional array (γ)
`mean`	`ComputationDataHandle`	1 dimensional array (μ)
`variance`	`ComputationDataHandle`	1 dimensional array (σ^2)
`grad_output`	`ComputationDataHandle`	Gradients passed to `BatchNormTraining` (∇y)
`epsilon`	`float`	Epsilon value (ϵ)
`feature_index`	`int64`	Index to feature dimension in `operand`

For each feature in the feature dimension (`feature_index` is the index for the feature dimension in `operand`), the operation calculates the gradients with respect to `operand`, `offset` and `scale` across all the other dimensions. If an invalid `feature_index` is passed, an error is produced.

The three gradients are defined by the following formulas:

$$\nabla x = \nabla y * \gamma * \sqrt{\sigma^2+\epsilon}$$

$$\nabla \gamma = sum(\nabla y * (x - \mu) * \sqrt{\sigma^2+\epsilon})$$

$$\nabla \beta = sum(\nabla y)$$

The inputs `mean` and `variance` represents moments value across batch and spatial dimensions.

The output type is a tuple of three ComputationDataHandles:

Outputs	Type	Semantics
`grad_operand`	`ComputationDataHandle`	gradient with respect to input
`operand`	`grad_offset`	`ComputationDataHandle`
`grad_scale`	`ComputationDataHandle`	gradient with respect to input `scale`

GetTupleElement

See also `ComputationBuilder::GetTupleElement` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Indexes into a tuple with a compile-time-constant value.

The value must be a compile-time-constant so that shape inference can determine the type of the resulting value.

This is analogous to `std::get<int N>(t)` in C++. Conceptually:

```
let v: f32[10] = f32[10]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
let s: s32 = 5;
let t: (f32[10], s32) = tuple(v, s);
let element_1: s32 = gettupleelement(t, 1);  // Inferred shape matches s32.
```

See also `tf.tuple` (https://www.tensorflow.org/api_docs/python/tf/tuple).

Infeed

See also `ComputationBuilder::Infeed` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

`Infeed(shape)`

Argument	Type	Semantics
`shape`	`Shape`	Shape of the data read from the Infeed interface. The layout field of the shape must be set to match the layout of the data sent to the device; otherwise its behavior is undefined.

Reads a single data item from the implicit Infeed streaming interface of the device, interpreting the data as the given shape and its layout, and returns a `ComputationDataHandle` of the data. Multiple Infeed operations are allowed in a computation, but there must be a total order among the Infeed operations. For example, two Infeeds in the code below have a total order since there is a dependency between the while loops. The compiler issues an error if there isn't a total order.

```
result1 = while (condition, init = init_value) {
  Infeed(shape)
}

result2 = while (condition, init = result1) {
  Infeed(shape)
}
```

Nested tuple shapes are not supported. For an empty tuple shape, the Infeed operation is effectively a nop and proceeds without reading any data from the Infeed of the device.

Note: We plan to allow multiple Infeed operations without a total order, in which case the compiler will provide information about how the Infeed operations are serialized in the compiled program.

Map

See also [`ComputationBuilder::Map`](https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

`Map(operands..., computation)`

Arguments	Type	Semantics
`operands`	sequence of N `ComputationDataHandle`s	N arrays of types T_0..T_{N-1}
`computation`	`Computation`	computation of type `T_0, T_1, ..., T_{N + M -1} -> S` with N parameters of type T and M of arbitrary type
`static_operands`	sequence of M `ComputationDataHandle`s	M arrays of arbitrary type

Applies a scalar function over the given `operands` arrays, producing an array of the same dimensions where each element is the result of the mapped function applied to the corresponding elements in the input arrays with `static_operands` given as additional input to `computation`.

The mapped function is an arbitrary computation with the restriction that it has N inputs of scalar type `T` and a single output with type `S`. The output has the same dimensions as the operands except that the element type T is replaced with S.

For example: `Map(op1, op2, op3, computation, par1)` maps `elem_out <- computation(elem1, elem2, elem3, par1)` at each (multi-dimensional) index in the input arrays to produce the output array.

Pad

See also [`ComputationBuilder::Pad`](https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

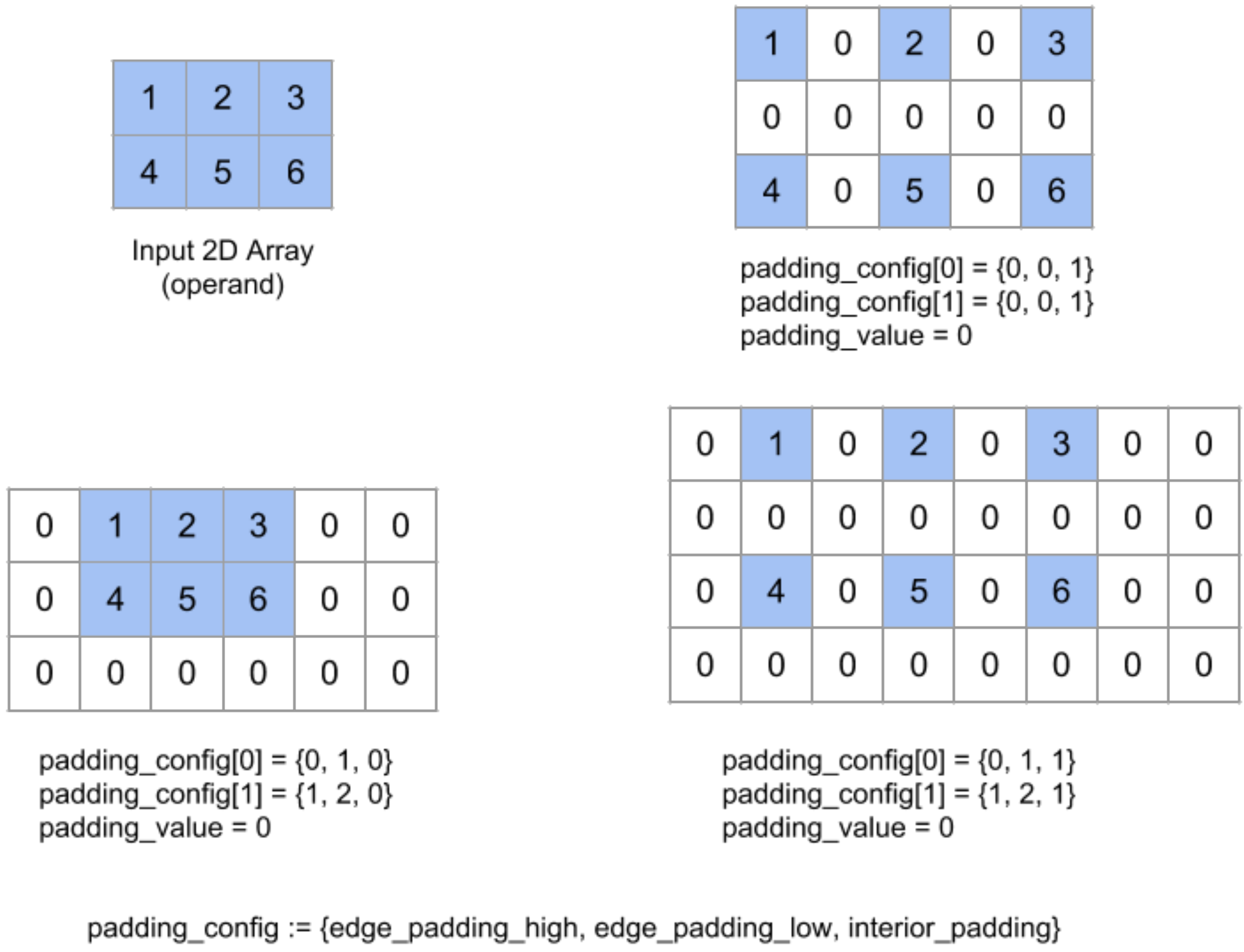
`Pad(operand, padding_value, padding_config)`

Arguments	Type	Semantics

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	array of type T
`padding_value`	`ComputationDataHandle`	scalar of type T to fill in the added padding
`padding_config`	`PaddingConfig`	padding amount on both edges (low, high) and between the elements of each dimension

Expands the given `operand` array by padding around the array as well as between the elements of the array with the given `padding_value`. `padding_config` specifies the amount of edge padding and the interior padding for each dimension.

`PaddingConfig` is a repeated field of `PaddingConfigDimension`, which contains three fields for each dimension: `edge_padding_low`, `edge_padding_high`, and `interior_padding`. `edge_padding_low` and `edge_padding_high` specifies the amount of padding added at the low-end (next to index 0) and the high-end (next to the highest index) of each dimension respectively. The amount of edge padding can be negative -- the absolute value of negative padding indicates the number of elements to remove from the specified dimension. `interior_padding` specifies the amount of padding added between any two elements in each dimension. Interior padding occurs logically before edge padding, so in the case of negative edge padding elements are removed from the interior-padded operand. This operation is a no-op if the edge padding pairs are all (0, 0) and the interior padding values are all 0. Figure below shows examples of different `edge_padding` and `interior_padding` values for a two dimensional array.



Reduce

See also [`ComputationBuilder::Reduce`](https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h)
(https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Applies a reduction function to an array.

`Reduce(operand, init_value, computation, dimensions)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	array of type T
`init_value`	`ComputationDataHandle`	scalar of type T
`computation`	`Computation`	computation of type `T, T -> T`
`dimensions`	`int64` array	unordered array of dimensions to reduce

Conceptually, this operation reduces one or more dimensions in the input array into scalars. The rank of the result array is `rank(operand) - len(dimensions)`. `init_value` is the initial value used for every reduction and may also be inserted anywhere during computation if the back-end chooses to do so. So in most cases `init_value` should be an identity of the reduction function (for example, 0 for addition).

The evaluation order of the reduction function is arbitrary and may be non-deterministic. Therefore, the reduction function should not be overly sensitive to reassociation.

Some reduction functions like addition are not strictly associative for floats. However, if the range of the data is limited, floating-point addition is close enough to being associative for most practical uses. It is possible to conceive of some completely non-associative reductions, however, and these will produce incorrect or unpredictable results in XLA reductions.

As an example, when reducing across the one dimension in a 1D array with values [10, 11, 12, 13], with reduction function `f` (this is `computation`) then that could be computed as

`f(10, f(11, f(12, f(init_value, 13)))`

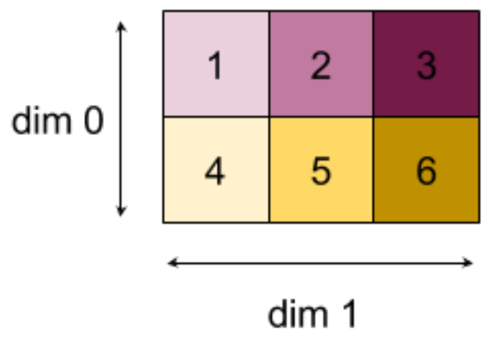
but there are also many other possibilities, e.g.

`f(init_value, f(f(10, f(init_value, 11)), f(f(init_value, 12), f(13, init_value))))`

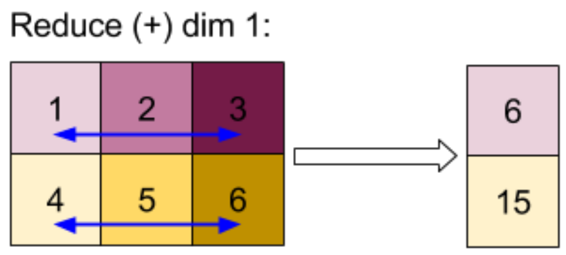
The following is a rough pseudo-code example of how reduction could be implemented, using summation as the reduction computation with an initial value of 0.

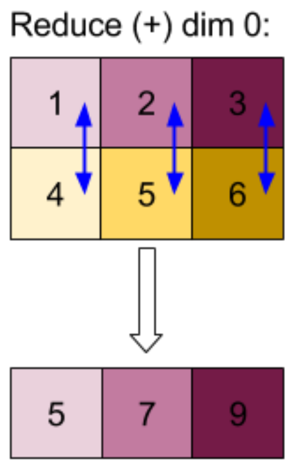
```
result_shape <- remove all dims in dimensions from operand_shape

# Iterate over all elements in result_shape. The number of r's here is equal
# to the rank of the result
for r0 in range(result_shape[0]), r1 in range(result_shape[1]), ...:
  # Initialize this result element
  result[r0, r1...] <- 0

  # Iterate over all the reduction dimensions
  for d0 in range(dimensions[0]), d1 in range(dimensions[1]), ...:
    # Increment the result element with the value of the operand's element.
    # The index of the operand's element is constructed from all ri's and di's
    # in the right order (by construction ri's and di's together index over the
    # whole operand shape).
    result[r0, r1...] += operand[ri... di]
```

Here's an example of reducing a 2D array (matrix). The shape has rank 2, dimension 0 of size 2 and dimension 1 of size 3:



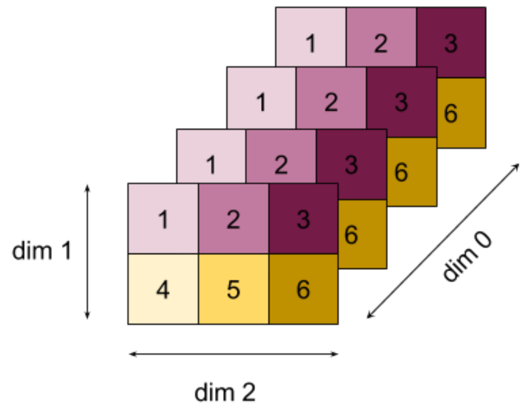
Results of reducing dimensions 0 or 1 with an "add" function:





Note that both reduction results are 1D arrays. The diagram shows one as column and another as row just for visual convenience.

For a more complex example, here is a 3D array. Its rank is 3, dimension 0 of size 4, dimension 1 of size 2 and dimension 2 of size 3. For simplicity, the values 1 to 6 are replicated across dimension 0.



Similarly to the 2D example, we can reduce just one dimension. If we reduce dimension 0, for example, we get a rank-2 array where all values across dimension 0 were folded into a scalar:

```
|  4   8  12 |
| 16  20  24 |
```

If we reduce dimension 2, we also get a rank-2 array where all values across dimension 2 were folded into a scalar:

```
| 6  15 |
| 6  15 |
| 6  15 |
| 6  15 |
```

Note that the relative order between the remaining dimensions in the input is preserved in the output, but some dimensions may get assigned new numbers (since the rank changes).

We can also reduce multiple dimensions. Add-reducing dimensions 0 and 1 produces the 1D array `| 20 28 36 |`.

Reducing the 3D array over all its dimensions produces the scalar `84`.

ReduceWindow

See also [`ComputationBuilder::ReduceWindow`](https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h)
(https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

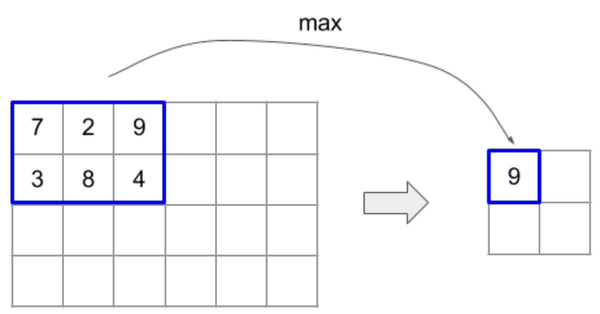
Applies a reduction function to all elements in each window of the input multi-dimensional array, producing an output multi-dimensional array with the same number of elements as the number of valid positions of the window. A pooling layer can be expressed as a `ReduceWindow`.

`ReduceWindow(operand, init_value, computation, window_dimensions, window_strides, padding)`

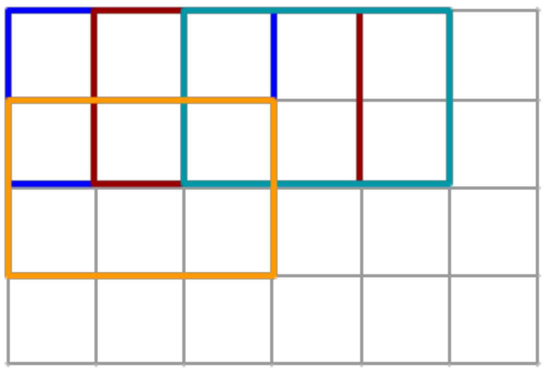
Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	N dimensional array containing elements of type T. This is the base area on which the window is placed.
`init_value`	`ComputationDataHandle`	Starting value for the reduction. See Reduce (#reduce) for details.
`computation`	`Computation`	Reduction function of type `T, T -> T`, to apply to all elements in each window
`window_dimensions`	`ArraySlice<int64>`	array of integers for window dimension values
`window_strides`	`ArraySlice<int64>`	array of integers for window stride values
`padding`	`Padding`	padding type for window (Padding\:\:kSame or Padding\:\:kValid)

Below code and figure shows an example of using `ReduceWindow`. Input is a matrix of size [4x6] and both window_dimensions and window_stride_dimensions are [2x3].

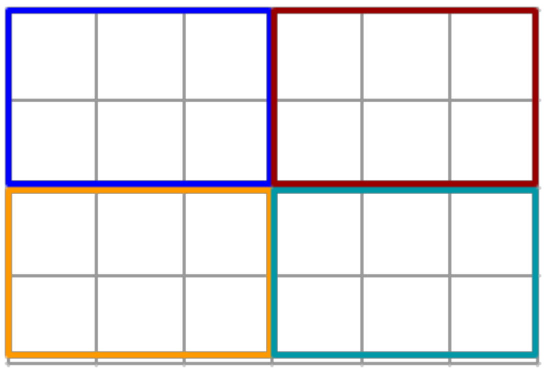
```
// Create a computation for the reduction (maximum).
Computation max;
{
  ComputationBuilder builder(client_, "max");
  auto y = builder.Parameter(0, ShapeUtil::MakeShape(F32, {}), "y");
  auto x = builder.Parameter(1, ShapeUtil::MakeShape(F32, {}), "x");
  builder.Max(y, x);
  max = builder.Build().ConsumeValueOrDie();
}

// Create a ReduceWindow computation with the max reduction computation.
ComputationBuilder builder(client_, "reduce_window_2x3");
auto shape = ShapeUtil::MakeShape(F32, {4, 6});
auto input = builder.Parameter(0, shape, "input");
builder.ReduceWindow(
    input, *max,
    /*init_val=*/builder.ConstantLiteral(LiteralUtil::MinValue(F32)),
    /*window_dimensions=*/{2, 3},
    /*window_stride_dimensions=*/{2, 3},
    Padding::kValid);
```



Stride of 1 in a dimension specifies that the position of a window in the dimension is 1 element away from its adjacent window. In order to specify that no windows overlap with each other, window_stride_dimensions should be equal to window_dimensions. The figure below illustrates the use of two different stride values. Padding is applied to each dimension of the input and the calculations are the same as though the input came in with the dimensions it has after padding.



window_dimensions = {2, 3}
window_stride_dimensions = {1, 1}



window_dimensions = {2, 3}
window_stride_dimensions = {2, 3}

The evaluation order of the reduction function is arbitrary and may be non-deterministic. Therefore, the reduction function should not be overly sensitive to reassociation. See the discussion about associativity in the context of **`Reduce`** (#reduce) for more details.

Reshape

See also **`ComputationBuilder::Reshape`** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) and the **`Collapse`** (#collapse) operation.

Reshapes the dimensions of an array into a new configuration.

`Reshape(operand, new_sizes)` **`Reshape(operand, dimensions, new_sizes)`**

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	array of type T
`dimensions`	`int64` vector	order in which dimensions are collapsed
`new_sizes`	`int64` vector	vector of sizes of new dimensions

Conceptually, reshape first flattens an array into a one-dimensional vector of data values, and then refines this vector into a new shape. The input arguments are an arbitrary array of type T, a compile-time-constant vector of dimension indices, and a compile-time-constant vector of dimension sizes for the result. The values in the `dimension` vector, if given, must be a permutation of all of T's dimensions; the default if not given is `{0, ..., rank - 1}`. The order of the dimensions in `dimensions` is from slowest-varying dimension (most major) to fastest-varying dimension (most minor) in the loop nest which collapses the input array into a single dimension. The `new_sizes` vector determines the size of the output array. The value at index 0 in `new_sizes` is the size of dimension 0, the value at index 1 is the size of dimension 1, and so on. The product of the `new_size` dimensions must equal the product of the operand's dimension sizes. When refining the collapsed array into the multidimensional array defined by `new_sizes`, the dimensions in `new_sizes` are ordered from slowest varying (most major) and to fastest varying (most minor).

For example, let v be an array of 24 elements:

```
let v = f32[4x2x3] { { {10, 11, 12}, {15, 16, 17}},
                    { {20, 21, 22}, {25, 26, 27}},
                    { {30, 31, 32}, {35, 36, 37}},
                    { {40, 41, 42}, {45, 46, 47}}};

In-order collapse:
let v012_24 = Reshape(v, {0,1,2}, {24});
then v012_24 == f32[24] {10, 11, 12, 15, 16, 17, 20, 21, 22, 25, 26, 27,
                         30, 31, 32, 35, 36, 37, 40, 41, 42, 45, 46, 47};

let v012_83 = Reshape(v, {0,1,2}, {8,3});
then v012_83 == f32[8x3] { {10, 11, 12}, {15, 16, 17},
                          {20, 21, 22}, {25, 26, 27},
                          {30, 31, 32}, {35, 36, 37},
                          {40, 41, 42}, {45, 46, 47}};

Out-of-order collapse:
let v021_24 = Reshape(v, {1,2,0}, {24});
```

```
then v012_24 == f32[24]  {10, 20, 30, 40, 11, 21, 31, 41, 12, 22, 32, 42,
                          15, 25, 35, 45, 16, 26, 36, 46, 17, 27, 37, 47};

let v021_83 = Reshape(v, {1,2,0}, {8,3});
then v021_83 == f32[8x3] { {10, 20, 30}, {40, 11, 21},
                          {31, 41, 12}, {22, 32, 42},
                          {15, 25, 35}, {45, 16, 26},
                          {36, 46, 17}, {27, 37, 47}};

let v021_262 = Reshape(v, {1,2,0}, {2,6,2});
then v021_262 == f32[2x6x2] { { {10, 20}, {30, 40},
                              {11, 21}, {31, 41},
                              {12, 22}, {32, 42}},
                             { {15, 25}, {35, 45},
                              {16, 26}, {36, 46},
                              {17, 27}, {37, 47}}};
```

As a special case, reshape can transform a single-element array to a scalar and vice versa. For example,

```
Reshape(f32[1x1] { {5}}, {0,1}, {}) == 5;
Reshape(5, {}, {1,1}) == f32[1x1] { {5}};
```

Rev (reverse)

See also `ComputationBuilder::Rev` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

`Rev(operand, dimensions)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	array of type T
`dimensions`	`ArraySlice<int64>`	dimensions to reverse

Reverses the order of elements in the `operand` array along the specified `dimensions`, generating an output array of the same shape. Each element of the operand array at a multidimensional index is stored into the output array at a transformed index. The multidimensional index is transformed by reversing the index in each dimension to be reversed (i.e., if a dimension of size N is one of the reversing dimensions, its index i is transformed into N - 1 - i).

One use for the `Rev` operation is to reverse the convolution weight array along the two window dimensions during the gradient computation in neural networks.

RngBernoulli

See also `ComputationBuilder::RngBernoulli` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Constructs an output of a given shape with random numbers generated following the Bernoulli distribution. The parameter needs to be a scalar valued F32 operand while the output shape needs to have elemental type U32.

`RngBernoulli(mean, shape)`

Arguments	Type	Semantics
`mean`	`ComputationDataHandle`	Scalar of type F32 specifying mean of generated numbers
`shape`	`Shape`	Output shape of type U32

RngNormal

See also `ComputationBuilder::RngNormal` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Constructs an output of a given shape with random numbers generated following

the

$$N(\mu, \sigma)$$

normal distribution. The parameters `mu` and `sigma`, and
output shape have to have elemental type F32. The parameters furthermore have to be scalar valued.

`RngNormal(mean, sigma, shape)`

Arguments	Type	Semantics
`mu`	`ComputationDataHandle`	Scalar of type F32 specifying mean of generated numbers
`sigma`	`ComputationDataHandle`	Scalar of type F32 specifying standard deviation of generated numbers
`shape`	`Shape`	Output shape of type F32

RngUniform

See also `ComputationBuilder::RngUniform`
(https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Constructs an output of a given shape with random numbers generated following

the uniform distribution over the interval

$$[a, b)$$

. The parameters and output
shape may be either F32, S32 or U32, but the types have to be consistent.
Furthermore, the parameters need to be scalar valued. If

$$b <= a$$

the result
is implementation-defined.

`RngUniform(a, b, shape)`

Arguments	Type	Semantics
`a`	`ComputationDataHandle`	Scalar of type T specifying lower limit of interval
`b`	`ComputationDataHandle`	Scalar of type T specifying upper limit of interval
`shape`	`Shape`	Output shape of type T

SelectAndScatter

See also `ComputationBuilder::SelectAndScatter`
(https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

This operation can be considered as a composite operation that first computes `ReduceWindow` on the `operand` array to select an element from each window, and then scatters the `source` array to the indices of the selected elements to construct an output array with the same shape as the operand array. The binary `select` function is used to select an element from each window by applying it across each window, and it is called with the property that the first parameter's index vector is lexicographically less than the second parameter's index vector. The `select` function returns `true` if the first parameter is selected and returns `false` if the second parameter is selected, and the function must hold transitivity (i.e., if `select(a, b)` and `select(b, c)` are `true`, then `select(a, c)` is also `true`) so that the selected element does not depend on the order of the elements traversed for a given window.

The function `scatter` is applied at each selected index in the output array. It takes two scalar parameters:

1. Current value at the selected index in the output array
2. The scatter value from `source` that applies to the selected index

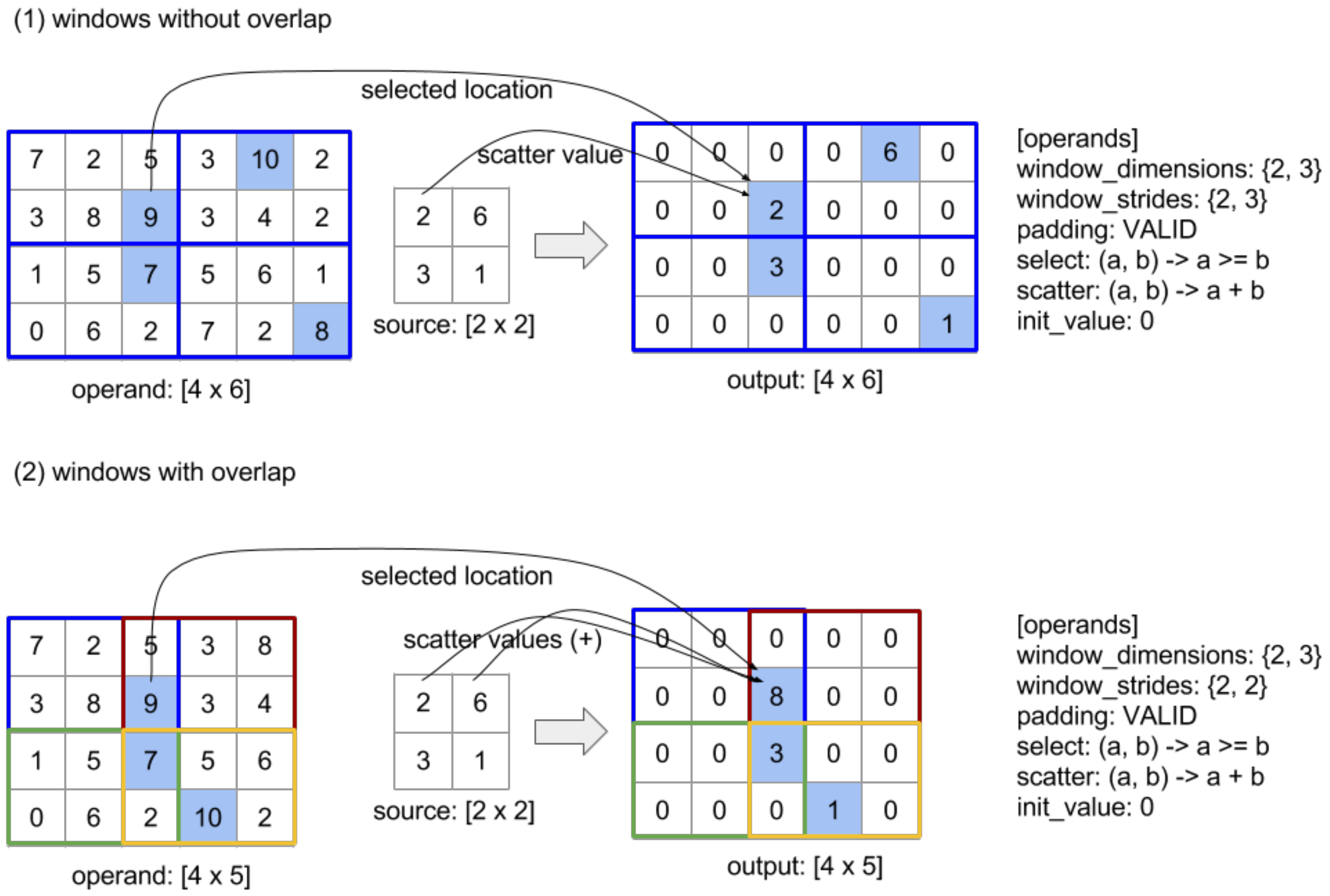
It combines the two parameters and returns a scalar value that's used to update the value at the selected index in the output array. Initially, all indices of the output array are set to `init_value`.

The output array has the same shape as the `operand` array and the `source` array must have the same shape as the result of applying a `ReduceWindow` operation on the `operand` array. `SelectAndScatter` can be used to backpropagate the gradient values for a pooling layer in a neural network.

`SelectAndScatter(operand, select, window_dimensions, window_strides, padding, source, init_value, scatter)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	array of type T over which the windows slide
`select`	`Computation`	binary computation of type `T, T -> PRED`, to apply to all elements in each window; returns `true` if the first parameter is selected and returns `false` if the second parameter is selected
`window_dimensions`	`ArraySlice<int64>`	array of integers for window dimension values
`window_strides`	`ArraySlice<int64>`	array of integers for window stride values
`padding`	`Padding`	padding type for window (Padding\:\:kSame or Padding\:\:kValid)
`source`	`ComputationDataHandle`	array of type T with the values to scatter
`init_value`	`ComputationDataHandle`	scalar value of type T for the initial value of the output array
`scatter`	`Computation`	binary computation of type `T, T -> T`, to apply each scatter source element with its destination element

The figure below shows examples of using `SelectAndScatter`, with the `select` function computing the maximal value among its parameters. Note that when the windows overlap, as in the figure (2) below, an index of the `operand` array may be selected multiple times by different windows. In the figure, the element of value 9 is selected by both of the top windows (blue and red) and the binary addition `scatter` function produces the output element of value 8 (2 + 6).



The evaluation order of the `scatter` function is arbitrary and may be non-deterministic. Therefore, the `scatter` function should not be overly sensitive to reassociation. See the discussion about associativity in the context of [`Reduce`](#reduce) (#reduce) for more details.

Select

See also [`ComputationBuilder::Select`](https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h)
(https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Constructs an output array from elements of two input arrays, based on the values of a predicate array.

`Select(pred, on_true, on_false)`

Arguments	Type	Semantics
`pred`	`ComputationDataHandle`	array of type PRED
`on_true`	`ComputationDataHandle`	array of type T
`on_false`	`ComputationDataHandle`	array of type T

The arrays `on_true` and `on_false` must have the same shape. This is also the shape of the output array. The array `pred` must have the same dimensionality as `on_true` and `on_false`, with the `PRED` element type.

For each element `P` of `pred`, the corresponding element of the output array is taken from `on_true` if the value of `P` is `true`, and from `on_false` if the value of `P` is `false`. As a restricted form of [broadcasting](https://www.tensorflow.org/performance/xla/broadcasting) (https://www.tensorflow.org/performance/xla/broadcasting) , `pred` can be a scalar of type `PRED`. In this case, the output array is taken wholly from `on_true` if `pred` is `true`, and from `on_false` if `pred` is `false`.

Example with non-scalar `pred`:

```
let pred: PRED[4] = {true, false, false, true};
let v1: s32[4] = {1, 2, 3, 4};
```

```
let v2: s32[4] = {100, 200, 300, 400};
==>
Select(pred, v1, v2) = s32[4]{1, 200, 300, 4};
```

Example with scalar `pred`:

```
let pred: PRED = true;
let v1: s32[4] = {1, 2, 3, 4};
let v2: s32[4] = {100, 200, 300, 400};
==>
Select(pred, v1, v2) = s32[4]{1, 2, 3, 4};
```

Selections between tuples are supported. Tuples are considered to be scalar types for this purpose. If `on_true` and `on_false` are tuples (which must have the same shape!) then `pred` has to be a scalar of type `PRED`.

Slice

See also `ComputationBuilder::Slice` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

Slicing extracts a sub-array from the input array. The sub-array is of the same rank as the input and contains the values inside a bounding box within the input array where the dimensions and indices of the bounding box are given as arguments to the slice operation.

`Slice(operand, start_indices, limit_indices)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	N dimensional array of type T
`start_indices`	`ArraySlice<int64>`	List of N integers containing the starting indices of the slice for each dimension. Values must be greater than or equal to zero.
`limit_indices`	`ArraySlice<int64>`	List of N integers containing the ending indices (exclusive) for the slice for each dimension. Each value must be strictly greater than the respective `start_indices` value for the dimension and less than or equal to the size of the dimension.

1-dimensional example:

```
let a = {0.0, 1.0, 2.0, 3.0, 4.0}
Slice(a, {2}, {4}) produces:
  {2.0, 3.0}
```

2-dimensional example:

```
let b =
 { {0.0,  1.0,  2.0},
   {3.0,  4.0,  5.0},
   {6.0,  7.0,  8.0},
   {9.0, 10.0, 11.0} }

Slice(b, {2, 1}, {4, 3}) produces:
  { { 7.0,  8.0},
    {10.0, 11.0} }
```

DynamicSlice

See also `ComputationBuilder::DynamicSlice` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

DynamicSlice extracts a sub-array from the input array at dynamic `start_indices`. The size of the slice in each dimension is passed in `size_indices`, which specify the end point of exclusive slice intervals in each dimension: [start, start + size). The shape of `start_indices` must be rank == 1, with dimension size equal to the rank of `operand`. Note: handling of out-of-bounds slice indices (generated by incorrect runtime calculation of 'start_indices') is currently implementation-defined. Currently, slice indices are computed modulo input dimension sizes to prevent out-of-bound array accesses, but this behavior may change in future implementations.

`DynamicSlice(operand, start_indices, size_indices)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	N dimensional array of type T
`start_indices`	`ComputationDataHandle`	Rank 1 array of N integers containing the starting indices of the slice for each dimension. Value must be greater than or equal to zero.
`size_indices`	`ArraySlice<int64>`	List of N integers containing the slice size for each dimension. Each value must be strictly greater than zero, and start + size must be less than or equal to the size of the dimension to avoid wrapping modulo dimension size.

1-dimensional example:

```
let a = {0.0, 1.0, 2.0, 3.0, 4.0}
let s = {2}

DynamicSlice(a, s, {2}) produces:
  {2.0, 3.0}
```

2-dimensional example:

```
let b =
 { {0.0,  1.0,  2.0},
   {3.0,  4.0,  5.0},
   {6.0,  7.0,  8.0},
   {9.0, 10.0, 11.0} }
let s = {2, 1}

DynamicSlice(b, s, {2, 2}) produces:
  { { 7.0,  8.0},
    {10.0, 11.0} }
```

DynamicUpdateSlice

See also `ComputationBuilder::DynamicUpdateSlice` (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

DynamicUpdateSlice generates a result which is the value of the input array `operand`, with a slice `update` overwritten at `start_indices`. The shape of `update` determines the shape of the sub-array of the result which is updated. The shape of `start_indices` must be rank == 1, with dimension size equal to the rank of `operand`. Note: handling of out-of-bounds slice indices (generated by incorrect runtime calculation of 'start_indices') is currently implementation-defined. Currently, slice indices are computed modulo update dimension sizes to prevent out-of-bound array accesses, but this behavior may change in future implementations.

`DynamicUpdateSlice(operand, update, start_indices)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	N dimensional array of type T
`update`	`ComputationDataHandle`	N dimensional array of type T containing the slice update. Each dimension of update shape must be strictly greater than zero, and start + update must be less than operand size for each dimension to avoid generating out-of-bounds update indices.
`start_indices`	`ComputationDataHandle`	Rank 1 array of N integers containing the starting indices of the slice for each dimension. Value must be greater than or equal to zero.

1-dimensional example:

```
let a = {0.0, 1.0, 2.0, 3.0, 4.0}
let u = {5.0, 6.0}
let s = {2}

DynamicUpdateSlice(a, u, s) produces:
  {0.0, 1.0, 5.0, 6.0, 4.0}
```

2-dimensional example:

```
let b =
 { {0.0,  1.0,  2.0},
   {3.0,  4.0,  5.0},
   {6.0,  7.0,  8.0},
   {9.0, 10.0, 11.0} }
let u =
 { {12.0,  13.0},
   {14.0,  15.0},
   {16.0,  17.0} }

let s = {1, 1}

DynamicUpdateSlice(b, u, s) produces:
 { {0.0,  1.0,  2.0},
   {3.0, 12.0, 13.0},
   {6.0, 14.0, 15.0},
   {9.0, 16.0, 17.0} }
```

Sort

See also **`ComputationBuilder::Sort`** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h) .

Sorts the elements in the operand.

`Sort(operand)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	The operand to sort

Transpose

See also the **`tf.reshape`** (https://www.tensorflow.org/api_docs/python/tf/reshape) operation.

`Transpose(operand)`

Arguments	Type	Semantics
`operand`	`ComputationDataHandle`	The operand to transpose.
`permutation`	`ArraySlice<int64>`	How to permute the dimensions.

Permutes the operand dimensions with the given permutation, so `∀ i . 0 ≤ i < rank ⇒ input_dimensions[permutation[i]] = output_dimensions[i]`.

This is the same as Reshape(operand, permutation, Permute(permutation, operand.shape.dimensions)).

Tuple

See also **`ComputationBuilder::Tuple`** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

A tuple containing a variable number of data handles, each of which has its own shape.

This is analogous to `std::tuple` in C++. Conceptually:

```
let v: f32[10] = f32[10]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
let s: s32 = 5;
let t: (f32[10], s32) = tuple(v, s);
```

Tuples can be deconstructed (accessed) via the **`GetTupleElement`** (#gettupleelement) operation.

While

See also **`ComputationBuilder::While`** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/client/computation_builder.h).

`While(condition, body, init)`

Arguments	Type	Semantics
`condition`	`Computation`	Computation of type `T -> PRED` which defines the termination condition of the loop.
`body`	`Computation`	Computation of type `T -> T` which defines the body of the loop.
`init`	`T`	Initial value for the parameter of `condition` and `body`.

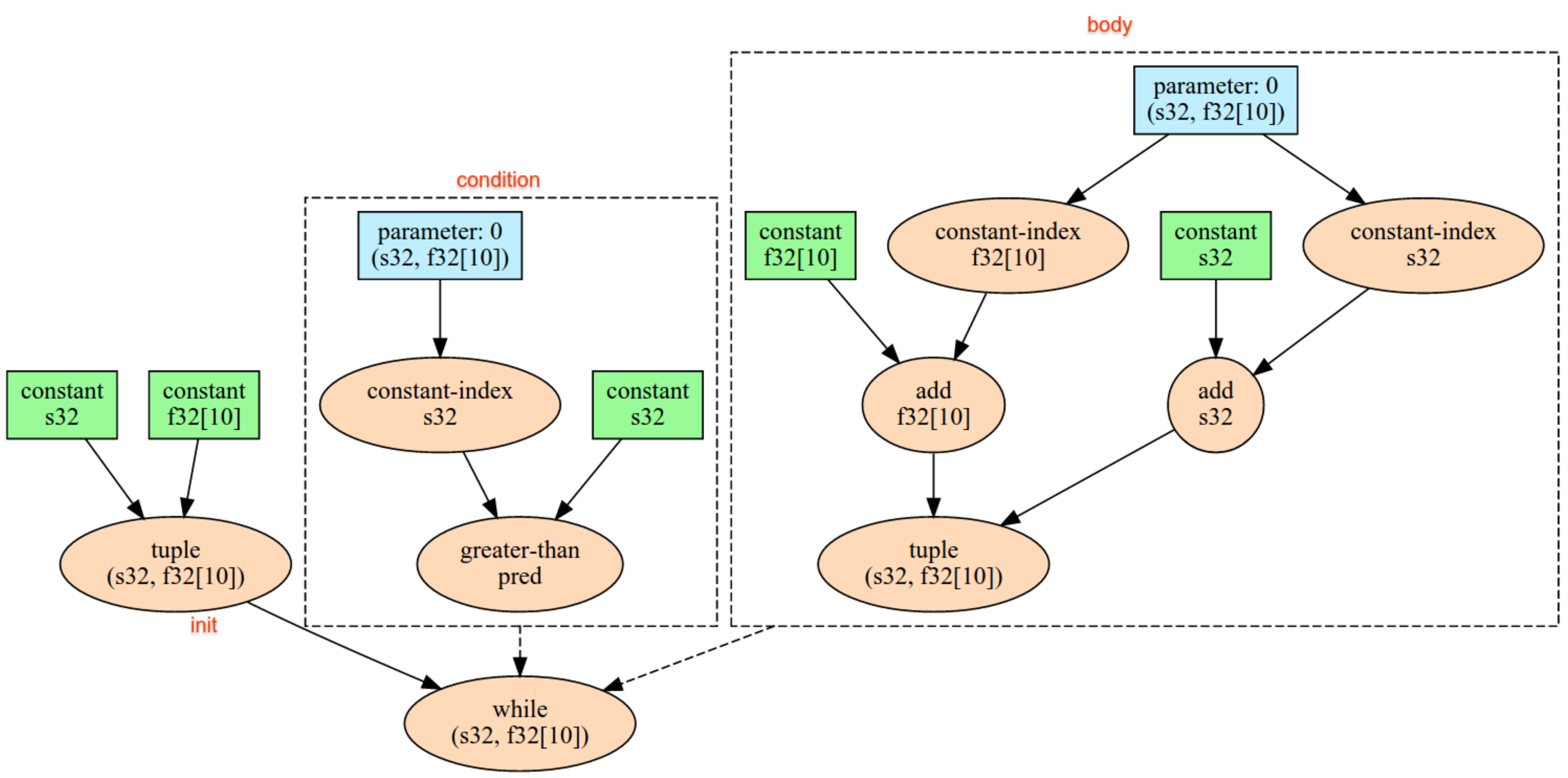
Sequentially executes the `body` until the `condition` fails. This is similar to a typical while loop in many other languages except for the differences and restrictions listed below.

- A `While` node returns a value of type `T`, which is the result from the last execution of the `body`.
- The shape of the type `T` is statically determined and must be the same across all iterations.
- `While` nodes are not allowed to be nested. (This restriction may be lifted in the future on some targets.)

The T parameters of the computations are initialized with the `init` value in the first iteration and are automatically updated to the new result from `body` in each subsequent iteration.

One main use case of the `While` node is to implement the repeated execution of training in neural networks. Simplified pseudocode is shown below with a graph that represents the computation. The code can be found in **`while_test.cc`** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/xla/tests/while_test.cc). The type `T` in this example is a `Tuple` consisting of an `int32` for the iteration count and a `vector[10]` for the accumulator. For 1000 iterations, the loop keeps adding a constant vector to the accumulator.

```
// Pseudocode for the computation.
init = {0, zero_vector[10]} // Tuple of int32 and float[10].
result = init;
while (result(0) < 1000) {
  iteration = result(0) + 1;
  new_vector = result(1) + constant_vector[10];
  result = {iteration, new_vector};
}
```



Last updated August 17, 2017.