```python
# Copyright 2018 The JAX Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import builtins
from functools import partial
import operator
from typing import Any, List, NamedTuple, Optional, Sequence, Tuple, Union

import numpy as np

from jax.interpreters import ad
from jax.interpreters import batching
from jax.interpreters import mlir
from jax._src import core
from jax._src import dtypes
from jax._src import util
from jax._src.lax import lax
from jax._src.lib import xla_client
from jax._src.lib.mlir.dialects import hlo

_max = builtins.max

Array = Any
DType = Any
Shape = core.Shape
```

[docs]

```python
class ConvDimensionNumbers(NamedTuple):
  """Describes batch, spatial, and feature dimensions of a convolution.

  Args:
    lhs_spec: a tuple of nonnegative integer dimension numbers containing
      `(batch dimension, feature dimension, spatial dimensions...)`.
    rhs_spec: a tuple of nonnegative integer dimension numbers containing
      `(out feature dimension, in feature dimension, spatial dimensions...)`.
    out_spec: a tuple of nonnegative integer dimension numbers containing
      `(batch dimension, feature dimension, spatial dimensions...)`.
  """
  lhs_spec: Sequence[int]
  rhs_spec: Sequence[int]
  out_spec: Sequence[int]


ConvGeneralDilatedDimensionNumbers = Union[
  None, ConvDimensionNumbers, Tuple[str, str, str]]
```

[docs]

```python
def conv_general_dilated(
  lhs: Array, rhs: Array, window_strides: Sequence[int],
  padding: Union[str, Sequence[Tuple[int, int]]],
  lhs_dilation: Optional[Sequence[int]] = None,
  rhs_dilation: Optional[Sequence[int]] = None,
  dimension_numbers: ConvGeneralDilatedDimensionNumbers  = None,
  feature_group_count: int = 1, batch_group_count: int = 1,
  precision: lax.PrecisionLike = None,
  preferred_element_type: Optional[DType] = None) -> Array:
  """General n-dimensional convolution operator, with optional dilation.

  Wraps XLA's `Conv
  <https://www.tensorflow.org/xla/operation_semantics#conv_convolution>`_
  operator.

  Args:
    lhs: a rank `n+2` dimensional input array.
    rhs: a rank `n+2` dimensional array of kernel weights.
    window_strides: a sequence of `n` integers, representing the inter-window
      strides.
    padding: either the string `'SAME'`, the string `'VALID'`, or a sequence of
      `n` `(low, high)` integer pairs that give the padding to apply before and
      after each spatial dimension.
    lhs_dilation: `None`, or a sequence of `n` integers, giving the
      dilation factor to apply in each spatial dimension of `lhs`. LHS dilation
      is also known as transposed convolution.
    rhs_dilation: `None`, or a sequence of `n` integers, giving the
      dilation factor to apply in each spatial dimension of `rhs`. RHS dilation
      is also known as atrous convolution.
    dimension_numbers: either `None`, a ``ConvDimensionNumbers`` object, or
      a 3-tuple ``(lhs_spec, rhs_spec, out_spec)``, where each element is a
      string of length `n+2`.
```

```
    feature_group_count: integer, default 1. See XLA HLO docs.
    batch_group_count: integer, default 1. See XLA HLO docs.
    precision: Optional. Either ``None``, which means the default precision for
      the backend, a :class:`~jax.lax.Precision` enum value (``Precision.DEFAULT``,
      ``Precision.HIGH`` or ``Precision.HIGHEST``), a string (e.g. 'highest' or
      'fastest', see the ``jax.default_matmul_precision`` context manager), or a
      tuple of two :class:`~jax.lax.Precision` enums or strings indicating precision
of
      ``lhs`` and ``rhs``.
    preferred_element_type: Optional. Either ``None``, which means the default
      accumulation type for the input types, or a datatype, indicating to
      accumulate results to and return a result with that datatype.

  Returns:
    An array containing the convolution result.

  In the string case of ``dimension_numbers``, each character identifies by
  position:

  - the batch dimensions in ``lhs``, ``rhs``, and the output with the character
    'N',
  - the feature dimensions in `lhs` and the output with the character 'C',
  - the input and output feature dimensions in rhs with the characters 'I'
    and 'O' respectively, and
  - spatial dimension correspondences between lhs, rhs, and the output using
    any distinct characters.

  For example, to indicate dimension numbers consistent with the ``conv``
  function with two spatial dimensions, one could use ``('NCHW', 'OIHW',
  'NCHW')``. As another example, to indicate dimension numbers consistent with
  the TensorFlow Conv2D operation, one could use ``('NHWC', 'HWIO', 'NHWC')``.
  When using the latter form of convolution dimension specification, window
  strides are associated with spatial dimension character labels according to
  the order in which the labels appear in the ``rhs_spec`` string, so that
  ``window_strides[0]`` is matched with the dimension corresponding to the first
  character appearing in rhs_spec that is not ``'I'`` or ``'O'``.

  If ``dimension_numbers`` is ``None``, the default is ``('NCHW', 'OIHW',
  'NCHW')`` (for a 2D convolution).
  """
  dnums = conv_dimension_numbers(lhs.shape, rhs.shape, dimension_numbers)
  if lhs_dilation is None:
    lhs_dilation = (1,) * (lhs.ndim - 2)
  elif isinstance(padding, str) and not len(lhs_dilation) == lhs_dilation.count(1):
    raise ValueError(
        "String padding is not implemented for transposed convolution "
        "using this op. Please either exactly specify the required padding or "
        "use conv_transpose.")
  if rhs_dilation is None:
    rhs_dilation = (1,) * (rhs.ndim - 2)
  if isinstance(padding, str):
    lhs_perm, rhs_perm, _ = dnums
    rhs_shape = np.take(rhs.shape, rhs_perm)[2:]  # type: ignore[index]
    effective_rhs_shape = [(k-1) * r + 1 for k, r in zip(rhs_shape, rhs_dilation)]
    padding = lax.padtype_to_pads(
        np.take(lhs.shape, lhs_perm)[2:], effective_rhs_shape,  # type:
ignore[index]
        window_strides, padding)
  else:
    try:
      padding = tuple((operator.index(lo), operator.index(hi))
                      for lo, hi in padding)
    except (ValueError, TypeError) as e:
      raise ValueError(
        "padding argument to conv_general_dilated should be a string or a "
        f"sequence of (low, high) pairs, got {padding}") from e

  preferred_element_type = (
      None if preferred_element_type is None else
      dtypes.canonicalize_dtype(np.dtype(preferred_element_type)))
  return conv_general_dilated_p.bind(
      lhs, rhs, window_strides=tuple(window_strides), padding=tuple(padding),
      lhs_dilation=tuple(lhs_dilation), rhs_dilation=tuple(rhs_dilation),
      dimension_numbers=dnums,
      feature_group_count=feature_group_count,
      batch_group_count=batch_group_count,
      lhs_shape=lhs.shape, rhs_shape=rhs.shape,
      precision=lax.canonicalize_precision(precision),
      preferred_element_type=preferred_element_type)
```

```
### convenience wrappers around traceables


def conv(lhs: Array, rhs: Array, window_strides: Sequence[int],
         padding: str, precision: lax.PrecisionLike = None,
         preferred_element_type: Optional[DType] = None) -> Array:
  """Convenience wrapper around `conv_general_dilated`.
```
[docs]

```
  Args:
    lhs: a rank `n+2` dimensional input array.
    rhs: a rank `n+2` dimensional array of kernel weights.
    window_strides: a sequence of `n` integers, representing the inter-window
      strides.
    padding: either the string `'SAME'`, the string `'VALID'`.
    precision: Optional. Either ``None``, which means the default precision for
      the backend, a :class:`~jax.lax.Precision` enum value (``Precision.DEFAULT``,
      ``Precision.HIGH`` or ``Precision.HIGHEST``) or a tuple of two
      :class:`~jax.lax.Precision` enums indicating precision of ``lhs``` and
``rhs``.
    preferred_element_type: Optional. Either ``None``, which means the default
      accumulation type for the input types, or a datatype, indicating to
      accumulate results to and return a result with that datatype.

  Returns:
    An array containing the convolution result.
  """
  return conv_general_dilated(lhs, rhs, window_strides, padding,
                              precision=precision,
                              preferred_element_type=preferred_element_type)


def conv_with_general_padding(lhs: Array, rhs: Array,
                              window_strides: Sequence[int],
                              padding: Union[str, Sequence[Tuple[int, int]]],
                              lhs_dilation: Optional[Sequence[int]],
                              rhs_dilation: Optional[Sequence[int]],
                              precision: lax.PrecisionLike = None,
                              preferred_element_type: Optional[DType] = None) ->
Array:
  """Convenience wrapper around `conv_general_dilated`.

  Args:
    lhs: a rank `n+2` dimensional input array.
    rhs: a rank `n+2` dimensional array of kernel weights.
    window_strides: a sequence of `n` integers, representing the inter-window
      strides.
    padding: either the string `'SAME'`, the string `'VALID'`, or a sequence of
      `n` `(low, high)` integer pairs that give the padding to apply before and
      after each spatial dimension.
    lhs_dilation: `None`, or a sequence of `n` integers, giving the
      dilation factor to apply in each spatial dimension of `lhs`. LHS dilation
      is also known as transposed convolution.
    rhs_dilation: `None`, or a sequence of `n` integers, giving the
      dilation factor to apply in each spatial dimension of `rhs`. RHS dilation
      is also known as atrous convolution.
    precision: Optional. Either ``None``, which means the default precision for
      the backend, a :class:`~jax.lax.Precision` enum value (``Precision.DEFAULT``,
      ``Precision.HIGH`` or ``Precision.HIGHEST``) or a tuple of two
      :class:`~jax.lax.Precision` enums indicating precision of ``lhs``` and
``rhs``.
    preferred_element_type: Optional. Either ``None``, which means the default
      accumulation type for the input types, or a datatype, indicating to
      accumulate results to and return a result with that datatype.

  Returns:
    An array containing the convolution result.
  """
  return conv_general_dilated(
      lhs, rhs, window_strides, padding, lhs_dilation=lhs_dilation,
      rhs_dilation=rhs_dilation, precision=precision,
      preferred_element_type=preferred_element_type)



def _conv_transpose_padding(k, s, padding):
  """Calculate before and after padding for a dim of transposed convolution.

  Args:
    k: int: kernel dimension.
    s: int: dimension stride value.
    padding: 'same' or 'valid' padding mode for original forward conv.

  Returns:
    2-tuple: ints: before and after padding for transposed convolution.
  """
  if padding == 'SAME':
    pad_len = k + s - 2
    if s > k - 1:
      pad_a = k - 1
    else:
      pad_a = int(np.ceil(pad_len / 2))
  elif padding == 'VALID':
    pad_len = k + s - 2 + _max(k - s, 0)
    pad_a = k - 1
  else:
    raise ValueError('Padding mode must be `SAME` or `VALID`.')
  pad_b = pad_len - pad_a
  return pad_a, pad_b
```

[docs]

```
def _flip_axes(x, axes):
  """Flip ndarray 'x' along each axis specified in axes tuple."""
  for axis in axes:
    x = np.flip(x, axis)
  return x


def conv_transpose(lhs: Array, rhs: Array, strides: Sequence[int],                    [docs]
                   padding: Union[str, Sequence[Tuple[int, int]]],
                   rhs_dilation: Optional[Sequence[int]] = None,
                   dimension_numbers: ConvGeneralDilatedDimensionNumbers = None,
                   transpose_kernel: bool = False,
                   precision: lax.PrecisionLike = None,
                   preferred_element_type: Optional[DType] = None) -> Array:
  """Convenience wrapper for calculating the N-d convolution "transpose".

  This function directly calculates a fractionally strided conv rather than
  indirectly calculating the gradient (transpose) of a forward convolution.

  Args:
    lhs: a rank `n+2` dimensional input array.
    rhs: a rank `n+2` dimensional array of kernel weights.
    strides: sequence of `n` integers, sets fractional stride.
    padding: 'SAME', 'VALID' will set as transpose of corresponding forward
      conv, or a sequence of `n` integer 2-tuples describing before-and-after
      padding for each `n` spatial dimension.
    rhs_dilation: `None`, or a sequence of `n` integers, giving the
      dilation factor to apply in each spatial dimension of `rhs`. RHS dilation
      is also known as atrous convolution.
    dimension_numbers: tuple of dimension descriptors as in
      lax.conv_general_dilated. Defaults to tensorflow convention.
    transpose_kernel: if True flips spatial axes and swaps the input/output
      channel axes of the kernel. This makes the output of this function identical
      to the gradient-derived functions like keras.layers.Conv2DTranspose
      applied to the same kernel. For typical use in neural nets this is completely
      pointless and just makes input/output channel specification confusing.
    precision: Optional. Either ``None``, which means the default precision for
      the backend, a :class:`~jax.lax.Precision` enum value (``Precision.DEFAULT``,
      ``Precision.HIGH`` or ``Precision.HIGHEST``) or a tuple of two
      :class:`~jax.lax.Precision` enums indicating precision of ``lhs``` and
``rhs``.
    preferred_element_type: Optional. Either ``None``, which means the default
      accumulation type for the input types, or a datatype, indicating to
      accumulate results to and return a result with that datatype.

  Returns:
    Transposed N-d convolution, with output padding following the conventions of
    keras.layers.Conv2DTranspose.
  """
  assert len(lhs.shape) == len(rhs.shape) and len(lhs.shape) >= 2
  ndims = len(lhs.shape)
  one = (1,) * (ndims - 2)
  # Set dimensional layout defaults if not specified.
  if dimension_numbers is None:
    if ndims == 2:
      dimension_numbers = ('NC', 'IO', 'NC')
    elif ndims == 3:
      dimension_numbers = ('NHC', 'HIO', 'NHC')
    elif ndims == 4:
      dimension_numbers = ('NHWC', 'HWIO', 'NHWC')
    elif ndims == 5:
      dimension_numbers = ('NHWDC', 'HWDIO', 'NHWDC')
    else:
      raise ValueError('No 4+ dimensional dimension_number defaults.')
  dn = conv_dimension_numbers(lhs.shape, rhs.shape, dimension_numbers)
  k_shape = np.take(rhs.shape, dn.rhs_spec)
  k_sdims = k_shape[2:]  # type: ignore[index]
  # Calculate correct output shape given padding and strides.
  pads: Union[str, Sequence[Tuple[int, int]]]
  if isinstance(padding, str) and padding in {'SAME', 'VALID'}:
    if rhs_dilation is None:
      rhs_dilation = (1,) * (rhs.ndim - 2)
    effective_k_size = map(lambda k, r: (k-1) * r + 1, k_sdims, rhs_dilation)
    pads = [_conv_transpose_padding(k, s, padding)
            for k,s in zip(effective_k_size, strides)]
  else:
    pads = padding
  if transpose_kernel:
    # flip spatial dims and swap input / output channel axes
    rhs = _flip_axes(rhs, np.array(dn.rhs_spec)[2:])
    rhs = np.swapaxes(rhs, dn.rhs_spec[0], dn.rhs_spec[1])
  return conv_general_dilated(lhs, rhs, one, pads, strides, rhs_dilation, dn,
                              precision=precision,
                              preferred_element_type=preferred_element_type)


def _conv_general_dilated_shape_rule(
```

```
    lhs: core.ShapedArray, rhs: core.ShapedArray, *, window_strides, padding,
    lhs_dilation, rhs_dilation, dimension_numbers, feature_group_count,
    batch_group_count, **unused_kwargs) -> Tuple[int, ...]:
  assert type(dimension_numbers) is ConvDimensionNumbers
  if len(lhs.shape) != len(rhs.shape):
    msg = ("conv_general_dilated lhs and rhs must have the same number of "
           "dimensions, but got {} and {}.")
    raise ValueError(msg.format(lhs.shape, rhs.shape))
  if not feature_group_count > 0:
    msg = ("conv_general_dilated feature_group_count "
           "must be a positive integer, got {}.")
    raise ValueError(msg.format(feature_group_count))
  lhs_feature_count = lhs.shape[dimension_numbers.lhs_spec[1]]
  quot, rem = divmod(lhs_feature_count, feature_group_count)
  if rem:
    msg = ("conv_general_dilated feature_group_count must divide lhs feature "
           "dimension size, but {} does not divide {}.")
    raise ValueError(msg.format(feature_group_count, lhs_feature_count))
  if not core.symbolic_equal_dim(quot, rhs.shape[dimension_numbers.rhs_spec[1]]):
    msg = ("conv_general_dilated lhs feature dimension size divided by "
           "feature_group_count must equal the rhs input feature dimension "
           "size, but {} // {} != {}.")
    raise ValueError(msg.format(lhs_feature_count, feature_group_count,
                                rhs.shape[dimension_numbers.rhs_spec[1]]))
  if rhs.shape[dimension_numbers.rhs_spec[0]] % feature_group_count:
    msg = ("conv_general_dilated rhs output feature dimension size must be a "
           "multiple of feature_group_count, but {} is not a multiple of {}.")
    raise ValueError(msg.format(rhs.shape[dimension_numbers.rhs_spec[0]],
                                feature_group_count))

  if not batch_group_count > 0:
    msg = ("conv_general_dilated batch_group_count "
           "must be a positive integer, got {}.")
    raise ValueError(msg.format(batch_group_count))
  lhs_batch_count = lhs.shape[dimension_numbers.lhs_spec[0]]
  if batch_group_count > 1 and lhs_batch_count % batch_group_count != 0:
    msg = ("conv_general_dilated batch_group_count must divide lhs batch "
           "dimension size, but {} does not divide {}.")
    raise ValueError(msg.format(batch_group_count, lhs_batch_count))

  if rhs.shape[dimension_numbers.rhs_spec[0]] % batch_group_count:
    msg = ("conv_general_dilated rhs output feature dimension size must be a "
           "multiple of batch_group_count, but {} is not a multiple of {}.")
    raise ValueError(msg.format(rhs.shape[dimension_numbers.rhs_spec[0]],
                                batch_group_count))

  if batch_group_count > 1 and feature_group_count > 1:
    msg = ("At most one of batch_group_count and feature_group_count may be > "
           "1, got batch_group_count={} and feature_group_count={}")
    raise ValueError(msg.format(batch_group_count, feature_group_count))

  if len(_conv_sdims(dimension_numbers.rhs_spec)) != len(window_strides):
    msg = ("conv_general_dilated window and window_strides must have "
           "the same number of dimensions, but got {} and {}")
    raise ValueError(
        msg.format(len(_conv_sdims(dimension_numbers.rhs_spec)),
len(window_strides)))

  lhs_perm, rhs_perm, out_perm = dimension_numbers
  lhs_trans = lax._dilate_shape(np.take(lhs.shape, lhs_perm), lhs_dilation)
  rhs_trans = lax._dilate_shape(np.take(rhs.shape, rhs_perm), rhs_dilation)
  out_trans = conv_shape_tuple(lhs_trans, rhs_trans, window_strides, padding,
                               batch_group_count)
  return tuple(np.take(out_trans, np.argsort(out_perm)))  # type: ignore[arg-type]


def _conv_general_dilated_dtype_rule(
    lhs, rhs, *, window_strides, padding, lhs_dilation, rhs_dilation,
    dimension_numbers, preferred_element_type, **unused_kwargs):
  input_dtype = lax.naryop_dtype_rule(lax._input_dtype, [lax._any, lax._any],
                                      'conv_general_dilated', lhs, rhs)
  if preferred_element_type is None:
    return input_dtype
  lax._validate_preferred_element_type(input_dtype, preferred_element_type)
  return preferred_element_type

_conv_spec_transpose = lambda spec: (spec[1], spec[0]) + spec[2:]
_conv_sdims = lambda spec: spec[2:]

# Understanding the convolution transpose rules:
# Ignoring the spatial dimensions, let m = batch, j = input feature,
# k = output feature.
#
# Convolution computes the following contraction:
# Forward: [m, j] [j, k] -> [m, k]
#
# The transposes are similar to the rules for transposing a matmul:
# LHS transpose: [m, k] [k, j] -> [m, j]
# RHS transpose: [j, m] [m, k] -> [j, k]
#
```

```python
# With feature grouping, we have the following signatures:
# Forward: [m, gj] [j, gk] -> [m, gk]
# LHS transpose: [m, gk] [k, gj] -> [m, gj]
# --> implemented as feature grouping after transposing the group from the
#     kernel input features to the kernel output features.
# RHS transpose: [gj, m] [m, gk] -> [j, gk]
# --> which is batch grouping.
#
# With batch grouping, we have the following signatures:
# Forward: [gm,j] [j,gk]->[m,gk]
# LHS transpose: [m, gk][gk, j] -> [gm, j]
# --> implemented as feature grouping with transposing the group on the kernel
#     and the output.
# RHS transpose: [j, gm][m, gk] -> [j, gk]
# --> which is feature grouping.

def _conv_general_dilated_transpose_lhs(
    g, rhs, *, window_strides, padding, lhs_dilation, rhs_dilation,
    dimension_numbers, feature_group_count, batch_group_count,
    lhs_shape, rhs_shape, precision, preferred_element_type):
  assert type(dimension_numbers) is ConvDimensionNumbers
  assert batch_group_count == 1 or feature_group_count == 1
  lhs_sdims, rhs_sdims, out_sdims = map(_conv_sdims, dimension_numbers)
  lhs_spec, rhs_spec, out_spec = dimension_numbers
  t_rhs_spec = _conv_spec_transpose(rhs_spec)
  if feature_group_count > 1:
    # in addition to switching the dims in the spec, need to move the feature
    # group axis into the transposed rhs's output feature dim
    rhs = _reshape_axis_out_of(rhs_spec[0], feature_group_count, rhs)
    rhs = _reshape_axis_into(rhs_spec[0], rhs_spec[1], rhs)
  elif batch_group_count > 1:
    rhs = _reshape_axis_out_of(rhs_spec[0], batch_group_count, rhs)
    rhs = _reshape_axis_into(rhs_spec[0], rhs_spec[1], rhs)
    feature_group_count = batch_group_count
  trans_dimension_numbers = ConvDimensionNumbers(out_spec, t_rhs_spec, lhs_spec)
  padding = _conv_general_vjp_lhs_padding(
      np.take(lhs_shape, lhs_sdims), np.take(rhs_shape, rhs_sdims),
      window_strides, np.take(g.shape, out_sdims), padding, lhs_dilation,
      rhs_dilation)
  revd_weights = lax.rev(rhs, rhs_sdims)
  out = conv_general_dilated(
      g, revd_weights, window_strides=lhs_dilation, padding=padding,
      lhs_dilation=window_strides, rhs_dilation=rhs_dilation,
      dimension_numbers=trans_dimension_numbers,
      feature_group_count=feature_group_count,
      batch_group_count=1, precision=precision,
      preferred_element_type=preferred_element_type)
  if batch_group_count > 1:
    out = _reshape_axis_out_of(lhs_spec[1], batch_group_count, out)
    out = _reshape_axis_into(lhs_spec[1], lhs_spec[0], out)
  return out

def _conv_general_dilated_transpose_rhs(
    g, lhs, *, window_strides, padding, lhs_dilation, rhs_dilation,
    dimension_numbers: ConvDimensionNumbers, feature_group_count: int,
    batch_group_count: int, lhs_shape, rhs_shape, precision,
    preferred_element_type):
  assert type(dimension_numbers) is ConvDimensionNumbers
  if np.size(g) == 0:
    # Avoids forming degenerate convolutions where the RHS has spatial size 0.
    # Awkwardly, we don't have an aval for the rhs readily available, so instead
    # of returning an ad_util.Zero instance here, representing a symbolic zero
    # value, we instead return a None, which is meant to represent having no
    # cotangent at all (and is thus incorrect for this situation), since the two
    # are treated the same operationally.
    # TODO(mattjj): adjust defbilinear so that the rhs aval is available here
    return None
  lhs_sdims, rhs_sdims, out_sdims = map(_conv_sdims, dimension_numbers)
  lhs_trans, rhs_trans, out_trans = map(_conv_spec_transpose, dimension_numbers)
  assert batch_group_count == 1 or feature_group_count == 1
  if batch_group_count > 1:
    feature_group_count = batch_group_count
    batch_group_count = 1
  elif feature_group_count > 1:
    batch_group_count = feature_group_count
    feature_group_count = 1
  trans_dimension_numbers = ConvDimensionNumbers(lhs_trans, out_trans, rhs_trans)
  padding = _conv_general_vjp_rhs_padding(
      np.take(lhs_shape, lhs_sdims), np.take(rhs_shape, rhs_sdims),
      window_strides, np.take(g.shape, out_sdims), padding, lhs_dilation,
      rhs_dilation)
  return conv_general_dilated(
      lhs, g, window_strides=rhs_dilation, padding=padding,
      lhs_dilation=lhs_dilation, rhs_dilation=window_strides,
      dimension_numbers=trans_dimension_numbers,
      feature_group_count=feature_group_count,
      batch_group_count=batch_group_count, precision=precision,
      preferred_element_type=preferred_element_type)

def _conv_general_dilated_batch_rule(
```

```python
  batched_args, batch_dims, *, window_strides, padding,
  lhs_dilation, rhs_dilation, dimension_numbers,
  feature_group_count, batch_group_count, precision,
  preferred_element_type, **unused_kwargs):
assert batch_group_count == 1 or feature_group_count == 1
lhs, rhs = batched_args
lhs_bdim, rhs_bdim = batch_dims
lhs_spec, rhs_spec, out_spec = dimension_numbers

# Some of the cases that reshape into batch or feature dimensions do not work
# with size 0 batch dimensions. The best fix would be to extend HLO to support
# multiple batch dimensions.
if ((lhs_bdim is not None and lhs.shape[lhs_bdim] == 0) or
    (rhs_bdim is not None and rhs.shape[rhs_bdim] == 0)):
  lhs_shape_unbatched, rhs_shape_unbatched = list(lhs.shape), list(rhs.shape)
  if lhs_bdim is not None:
    lhs_shape_unbatched.pop(lhs_bdim)
  if rhs_bdim is not None:
    rhs_shape_unbatched.pop(rhs_bdim)
  shape = _conv_general_dilated_shape_rule(
    core.ShapedArray(lhs_shape_unbatched, lhs.dtype),
    core.ShapedArray(rhs_shape_unbatched, rhs.dtype),
    window_strides=window_strides, padding=padding, lhs_dilation=lhs_dilation,
    rhs_dilation=rhs_dilation, dimension_numbers=dimension_numbers,
    feature_group_count=feature_group_count,
    batch_group_count=batch_group_count)
  return lax.full(
    (0,) + shape, 0,
    dtype=lhs.dtype if preferred_element_type is None
          else preferred_element_type), 0


if lhs_bdim is not None and rhs_bdim is not None:
  assert lhs.shape[lhs_bdim] == rhs.shape[rhs_bdim]
  if batch_group_count > 1:
    new_lhs = _reshape_axis_into(lhs_bdim, lhs_spec[0], lhs)
    batch_group_count *= lhs.shape[lhs_bdim]
  else:
    new_lhs = _reshape_axis_into(lhs_bdim, lhs_spec[1], lhs)
    feature_group_count *= lhs.shape[lhs_bdim]
  new_rhs = _reshape_axis_into(rhs_bdim, rhs_spec[0], rhs)
  out = conv_general_dilated(
    new_lhs, new_rhs, window_strides, padding, lhs_dilation, rhs_dilation,
    dimension_numbers, feature_group_count=feature_group_count,
    batch_group_count=batch_group_count, precision=precision,
    preferred_element_type=preferred_element_type)
  out = _reshape_axis_out_of(out_spec[1], lhs.shape[lhs_bdim], out)
  return out, out_spec[1]

elif lhs_bdim is not None:
  if batch_group_count == 1:
    new_lhs = _reshape_axis_into(lhs_bdim, lhs_spec[0], lhs)
    out = conv_general_dilated(new_lhs, rhs, window_strides, padding,
                               lhs_dilation, rhs_dilation, dimension_numbers,
                               feature_group_count, precision=precision,
                               preferred_element_type=preferred_element_type)
    out = _reshape_axis_out_of(out_spec[0], lhs.shape[lhs_bdim], out)
    return out, out_spec[0]
  else:
    new_lhs = _reshape_axis_out_of(lhs_spec[0] + int(lhs_bdim <= lhs_spec[0]),
                                   batch_group_count, lhs)
    new_lhs = _reshape_axis_into(lhs_bdim + int(lhs_spec[0] < lhs_bdim),
                                 lhs_spec[0] + 1,
                                 new_lhs)
    new_lhs = _reshape_axis_into(lhs_spec[0], lhs_spec[0], new_lhs)
    out = conv_general_dilated(new_lhs, rhs, window_strides, padding,
                               lhs_dilation, rhs_dilation, dimension_numbers,
                               feature_group_count, batch_group_count,
                               precision=precision,
                               preferred_element_type=preferred_element_type)
    out = _reshape_axis_out_of(out_spec[0], lhs.shape[lhs_bdim], out)
    return out, out_spec[0]

elif rhs_bdim is not None:
  if feature_group_count == 1 and batch_group_count == 1:
    new_rhs = _reshape_axis_into(rhs_bdim, rhs_spec[0], rhs)
    out = conv_general_dilated(lhs, new_rhs, window_strides, padding,
                               lhs_dilation, rhs_dilation, dimension_numbers,
                               feature_group_count, batch_group_count,
                               precision=precision,
                               preferred_element_type=preferred_element_type)
    out = _reshape_axis_out_of(out_spec[1], rhs.shape[rhs_bdim], out)
    return out, out_spec[1]
  else:
    # groups need to be outermost, so we need to factor them out of the
    # rhs output feature dim, then factor the batch dim into the remaining rhs
    # output feature dim, then put groups back in. We do something
    # similar on the output. An alternative which would require more FLOPs but
    # fewer reshapes would be to broadcast lhs.
    group_count = (feature_group_count if feature_group_count > 1
```

```
            else batch_group_count)
      new_rhs = _reshape_axis_out_of(rhs_spec[0] + int(rhs_bdim <= rhs_spec[0]),
                                     group_count, rhs)
      new_rhs = _reshape_axis_into(rhs_bdim + int(rhs_spec[0] < rhs_bdim),
                                   rhs_spec[0] + 1, new_rhs)
      new_rhs = _reshape_axis_into(rhs_spec[0], rhs_spec[0], new_rhs)
      out = conv_general_dilated(lhs, new_rhs, window_strides, padding,
                                 lhs_dilation, rhs_dilation, dimension_numbers,
                                 feature_group_count, batch_group_count,
                                 precision=precision,
                                 preferred_element_type=preferred_element_type)
      out = _reshape_axis_out_of(out_spec[1], group_count, out)
      out = _reshape_axis_out_of(out_spec[1] + 1, rhs.shape[rhs_bdim], out)
      out = _reshape_axis_into(out_spec[1], out_spec[1] + 1, out)
      return out, out_spec[1]

conv_general_dilated_p = lax.standard_primitive(
    _conv_general_dilated_shape_rule, _conv_general_dilated_dtype_rule,
    'conv_general_dilated')

ad.defbilinear(conv_general_dilated_p,
               _conv_general_dilated_transpose_lhs,
               _conv_general_dilated_transpose_rhs)
batching.primitive_batchers[conv_general_dilated_p] = \
    _conv_general_dilated_batch_rule

def _complex_mul(mul, x, y):
  # We use a trick for complex multiplication sometimes attributed to Gauss
  # which uses three multiplications and five additions; instead of the naive
  # method of four multiplications and two additions.
  #
https://en.wikipedia.org/wiki/Multiplication_algorithm#Complex_multiplication_algori
thm
  #
  # This performance win comes with a trade-off in accuracy; especially in
  # cases when the real and imaginary differ hugely in magnitude. The relative
  # error bound (e.g. 1p-24 in case of float32) would be relative to the
  # maximum of real and imaginary parts of the result instead of being
  # satisfied by the real and imaginary parts independently of each other.
  x_re, x_im = lax.real(x), lax.imag(x)
  y_re, y_im = lax.real(y), lax.imag(y)
  k1 = mul(lax.add(x_re, x_im), y_re)
  k2 = mul(x_re, lax.sub(y_im, y_re))
  k3 = mul(x_im, lax.add(y_re, y_im))
  return lax.complex(lax.sub(k1, k3), lax.add(k1, k2))


_real_dtype = lambda dtype: np.finfo(dtype).dtype

def _conv_general_dilated_lower(
    ctx, lhs, rhs, *, window_strides, padding,
    lhs_dilation, rhs_dilation, dimension_numbers, feature_group_count,
    batch_group_count, precision, preferred_element_type,
    expand_complex_convolutions=False, **unused_kwargs):
  lhs_aval, rhs_aval = ctx.avals_in
  aval_out, = ctx.avals_out
  assert isinstance(dimension_numbers, ConvDimensionNumbers)
  dtype = lhs_aval.dtype
  if expand_complex_convolutions and np.issubdtype(dtype, np.complexfloating):
    if preferred_element_type is not None:
      # Convert complex dtype to types used for real and imaginary parts
      assert np.issubdtype(preferred_element_type, np.complexfloating)
      preferred_element_type = _real_dtype(preferred_element_type)
    complex_conv = mlir.lower_fun(
      partial(
        _complex_mul,
        partial(conv_general_dilated, window_strides=window_strides,
                padding=padding, lhs_dilation=lhs_dilation,
                rhs_dilation=rhs_dilation, dimension_numbers=dimension_numbers,
                feature_group_count=feature_group_count,
                batch_group_count=batch_group_count, precision=precision,
                preferred_element_type=preferred_element_type)),
      multiple_results=False)
    return complex_conv(ctx, lhs, rhs)

  lhs_spec, rhs_spec, out_spec = dimension_numbers
  dnums = hlo.ConvDimensionNumbers.get(
    input_batch_dimension=lhs_spec[0],
    input_feature_dimension=lhs_spec[1],
    input_spatial_dimensions=list(lhs_spec[2:]),
    kernel_output_feature_dimension=rhs_spec[0],
    kernel_input_feature_dimension=rhs_spec[1],
    kernel_spatial_dimensions=list(rhs_spec[2:]),
    output_batch_dimension=out_spec[0],
    output_feature_dimension=out_spec[1],
    output_spatial_dimensions=list(out_spec[2:]))
  num_spatial_dims = len(rhs_spec) - 2
  if len(padding) == 0:
    padding = np.zeros((0, 2), dtype=np.int64)
  window_reversal = mlir.dense_bool_elements([False] * num_spatial_dims)
```

```python
  return [
      hlo.ConvolutionOp(
        mlir.aval_to_ir_type(aval_out),
        lhs,
        rhs,
        dimension_numbers=dnums,
        feature_group_count=mlir.i64_attr(feature_group_count),
        batch_group_count=mlir.i64_attr(batch_group_count),
        window_strides=mlir.dense_int_elements(window_strides),
        padding=mlir.dense_int_elements(padding),
        lhs_dilation=mlir.dense_int_elements(lhs_dilation),
        rhs_dilation=mlir.dense_int_elements(rhs_dilation),
        window_reversal=window_reversal,
        precision_config=lax.precision_attr(precision)).result
  ]

mlir.register_lowering(conv_general_dilated_p, _conv_general_dilated_lower)
# TODO(b/161124619, b/161126248): XLA does not support complex convolution on
# GPU, and on CPU it uses a slow loop-based implementation;
# on these backends, lower complex convolutions away.
mlir.register_lowering(
    conv_general_dilated_p,
    partial(_conv_general_dilated_lower, expand_complex_convolutions=True),
    platform='cpu')
mlir.register_lowering(
    conv_general_dilated_p,
    partial(_conv_general_dilated_lower, expand_complex_convolutions=True),
    platform='gpu')


def _reshape_axis_into(src, dst, x):
  # NB: `dst` is the number of the dimension that we should reshape into
  # *after* `src` is removed from `x`'s list of dimensions. For example, if
  # `src` is an added batch dimension, `dst` might name a target dimension in
  # the unbatched list of dimensions.
  perm = [i for i in range(x.ndim) if i != src]
  perm.insert(dst, src)
  new_shape = list(np.delete(x.shape, src))
  new_shape[dst] *= x.shape[src]
  return lax.reshape(x, new_shape, perm)

def _reshape_axis_out_of(src, size1, x):
  shape = list(x.shape)
  size2, ragged = divmod(shape[src], size1)
  assert not ragged
  shape[src:src+1] = [size1, size2]
  return lax.reshape(x, shape)


def _check_conv_shapes(name, lhs_shape, rhs_shape, window_strides):
  """Check that conv shapes are valid and are consistent with window_strides."""
  if len(lhs_shape) != len(rhs_shape):
    msg = "Arguments to {} must have same rank, got {} and {}."
    raise TypeError(msg.format(name, len(lhs_shape), len(rhs_shape)))
  if len(lhs_shape) < 2:
    msg = "Arguments to {} must have rank at least 2, got {} and {}."
    raise TypeError(msg.format(name, len(lhs_shape), len(rhs_shape)))
  if lhs_shape[1] != rhs_shape[1]:
    msg = "Arguments to {} must agree on input feature size, got {} and {}."
    raise TypeError(msg.format(name, lhs_shape[1], rhs_shape[1]))
  lax._check_shapelike(name, "window_strides", window_strides)
  if not np.all(np.greater(window_strides, 0)):
    msg = "All elements of window_strides must be positive, got {}."
    raise TypeError(msg.format(window_strides))
  if len(window_strides) != len(lhs_shape) - 2:
    msg = "{} window_strides has wrong length: expected {}, got {}."
    expected_length = len(lhs_shape) - 2
    raise TypeError(msg.format(name, expected_length, len(window_strides)))


def conv_shape_tuple(lhs_shape, rhs_shape, strides, pads, batch_group_count=1):
  """Compute the shape tuple of a conv given input shapes in canonical order."""
  if isinstance(pads, str):
    pads = lax.padtype_to_pads(lhs_shape[2:], rhs_shape[2:], strides, pads)
  if len(pads) != len(lhs_shape) - 2:
    msg = "Wrong number of explicit pads for convolution: expected {}, got {}."
    raise TypeError(msg.format(len(lhs_shape) - 2, len(pads)))

  lhs_padded = np.add(lhs_shape[2:], np.sum(np.array(pads).reshape(-1, 2),
                                              axis=1))
  if np.any(lhs_padded < 0):
    raise ValueError("Negative padding is larger than the size of the corresponding
dimension: "
                     f"got padding={pads} for lhs_shape[2:]={lhs_shape[2:]}")
  out_space = core.stride_shape(lhs_padded, rhs_shape[2:], strides)
  out_space = np.maximum(0, out_space)
  if batch_group_count > 1:
    assert lhs_shape[0] % batch_group_count == 0
    out_shape_0 = lhs_shape[0] // batch_group_count
  else:
```

```
    out_shape_0 = lhs_shape[0]
  out_shape = (out_shape_0, rhs_shape[0])
  return tuple(out_shape + tuple(out_space))


def conv_general_shape_tuple(lhs_shape, rhs_shape, window_strides, padding,
                             dimension_numbers):
  lhs_perm, rhs_perm, out_perm = conv_general_permutations(dimension_numbers)
  lhs_trans = np.take(lhs_shape, lhs_perm)
  rhs_trans = np.take(rhs_shape, rhs_perm)
  out_trans = conv_shape_tuple(lhs_trans, rhs_trans, window_strides, padding)
  return tuple(np.take(out_trans, np.argsort(out_perm)))


def conv_transpose_shape_tuple(lhs_shape, rhs_shape, window_strides, padding,
                               dimension_numbers):
  lhs_perm, rhs_perm, out_perm = conv_general_permutations(dimension_numbers)
  lhs_trans = np.take(lhs_shape, lhs_perm)
  rhs_trans = np.take(rhs_shape, rhs_perm)
  if isinstance(padding, str):
    padding = [_conv_transpose_padding(k, s, padding)
               for k,s in zip(rhs_trans[2:], window_strides)]
  padding = list(map(np.sum, padding))
  unpad_out_space = [(i-1) * s - k + 2
                     for i, k, s in zip(lhs_trans[2:],
                                        rhs_trans[2:],
                                        window_strides)]
  out_space = np.sum([unpad_out_space, padding], axis=0).tolist()
  out_trans = tuple((lhs_trans[0], rhs_trans[0]) + tuple(out_space))
  return tuple(np.take(out_trans, np.argsort(out_perm)))

def conv_dimension_numbers(lhs_shape, rhs_shape, dimension_numbers                [docs]
                           ) -> ConvDimensionNumbers:
  """Converts convolution `dimension_numbers` to a `ConvDimensionNumbers`.

  Args:
    lhs_shape: tuple of nonnegative integers, shape of the convolution input.
    rhs_shape: tuple of nonnegative integers, shape of the convolution kernel.
    dimension_numbers: None or a tuple/list of strings or a ConvDimensionNumbers
      object following the convolution dimension number specification format in
      xla_client.py.

  Returns:
    A `ConvDimensionNumbers` object that represents `dimension_numbers` in the
    canonical form used by lax functions.
  """
  if isinstance(dimension_numbers, ConvDimensionNumbers):
    return dimension_numbers
  if len(lhs_shape) != len(rhs_shape):
    msg = "convolution requires lhs and rhs ndim to be equal, got {} and {}."
    raise TypeError(msg.format(len(lhs_shape), len(rhs_shape)))

  if dimension_numbers is None:
    iota = tuple(range(len(lhs_shape)))
    return ConvDimensionNumbers(iota, iota, iota)
  elif isinstance(dimension_numbers, (list, tuple)):
    if len(dimension_numbers) != 3:
      msg = "convolution dimension_numbers list/tuple must be length 3, got {}."
      raise TypeError(msg.format(len(dimension_numbers)))
    if not all(isinstance(elt, str) for elt in dimension_numbers):
      msg = "convolution dimension_numbers elements must be strings, got {}."
      raise TypeError(msg.format(tuple(map(type, dimension_numbers))))
    msg = ("convolution dimension_numbers[{}] must have len equal to the ndim "
           "of lhs and rhs, got {} for lhs and rhs shapes {} and {}.")
    for i, elt in enumerate(dimension_numbers):
      if len(elt) != len(lhs_shape):
        raise TypeError(msg.format(i, len(elt), lhs_shape, rhs_shape))

    lhs_spec, rhs_spec, out_spec = conv_general_permutations(dimension_numbers)
    return ConvDimensionNumbers(lhs_spec, rhs_spec, out_spec)
  else:
    msg = "convolution dimension_numbers must be tuple/list or None, got {}."
    raise TypeError(msg.format(type(dimension_numbers)))



def conv_general_permutations(dimension_numbers):
  """Utility for convolution dimension permutations relative to Conv HLO."""
  lhs_spec, rhs_spec, out_spec = dimension_numbers
  lhs_char, rhs_char, out_char = charpairs = ("N", "C"), ("O", "I"), ("N", "C")
  for i, (a, b) in enumerate(charpairs):
    if not dimension_numbers[i].count(a) == dimension_numbers[i].count(b) == 1:
      msg = ("convolution dimension_numbers[{}] must contain the characters "
             "'{}' and '{}' exactly once, got {}.")
      raise TypeError(msg.format(i, a, b, dimension_numbers[i]))
    if len(dimension_numbers[i]) != len(set(dimension_numbers[i])):
      msg = ("convolution dimension_numbers[{}] cannot have duplicate "
             "characters, got {}.")
      raise TypeError(msg.format(i, dimension_numbers[i]))
  if not (set(lhs_spec) - set(lhs_char) == set(rhs_spec) - set(rhs_char) ==
```

```
        set(out_spec) - set(out_char)):
      msg = ("convolution dimension_numbers elements must each have the same "
             "set of spatial characters, got {}.")
      raise TypeError(msg.format(dimension_numbers))

  def getperm(spec, charpair):
    spatial = (i for i, c in enumerate(spec) if c not in charpair)
    if spec is not rhs_spec:
      spatial = sorted(spatial, key=lambda i: rhs_spec.index(spec[i]))
    return (spec.index(charpair[0]), spec.index(charpair[1])) + tuple(spatial)

  lhs_perm, rhs_perm, out_perm = map(getperm, dimension_numbers, charpairs)
  return lhs_perm, rhs_perm, out_perm


def _conv_general_proto(dimension_numbers):
  assert type(dimension_numbers) is ConvDimensionNumbers
  lhs_spec, rhs_spec, out_spec = dimension_numbers
  proto = xla_client.ConvolutionDimensionNumbers()
  proto.input_batch_dimension = lhs_spec[0]
  proto.input_feature_dimension = lhs_spec[1]
  proto.output_batch_dimension = out_spec[0]
  proto.output_feature_dimension = out_spec[1]
  proto.kernel_output_feature_dimension = rhs_spec[0]
  proto.kernel_input_feature_dimension = rhs_spec[1]
  proto.input_spatial_dimensions.extend(lhs_spec[2:])
  proto.kernel_spatial_dimensions.extend(rhs_spec[2:])
  proto.output_spatial_dimensions.extend(out_spec[2:])
  return proto


def _conv_general_vjp_lhs_padding(
    in_shape, window_dimensions, window_strides, out_shape, padding,
    lhs_dilation, rhs_dilation) -> List[Tuple[int, int]]:
  lhs_dilated_shape = lax._dilate_shape(in_shape, lhs_dilation)
  rhs_dilated_shape = lax._dilate_shape(window_dimensions, rhs_dilation)
  out_dilated_shape = lax._dilate_shape(out_shape, window_strides)
  pad_before = np.subtract(rhs_dilated_shape, [lo for lo, _ in padding]) - 1
  pad_after = (np.add(lhs_dilated_shape, rhs_dilated_shape) - 1
               - out_dilated_shape - pad_before)
  return util.safe_zip(pad_before, pad_after)


def _conv_general_vjp_rhs_padding(
    in_shape, window_dimensions, window_strides, out_shape, padding,
    lhs_dilation, rhs_dilation):

  if len(in_shape) == 0:  # 0D conv
    return []
  lhs_dilated_shape = lax._dilate_shape(in_shape, lhs_dilation)
  rhs_dilated_shape = lax._dilate_shape(window_dimensions, rhs_dilation)
  out_dilated_shape = lax._dilate_shape(out_shape, window_strides)
  pads_lo, _ = util.unzip2(padding)
  pads_from_lhs = core.diff_shape(out_dilated_shape, lhs_dilated_shape)
  pads_from_rhs = core.diff_shape(core.diff_shape(rhs_dilated_shape, pads_lo),
                                  (1,) * len(pads_lo))
  pads_hi = core.sum_shapes(pads_from_lhs, pads_from_rhs)
  return list(zip(pads_lo, pads_hi))
```