tensorflow / **tensorflow** Public

<> **Code** ⊙ Issues 2.1k Pull requests 283 Actions Projects 1 ···

5100e359ae ▾ ···

tensorflow / tensorflow / core / ops / **array_ops.cc**

frgossen [MLIR][KernelGen] Add experimental JIT-compiled GPU kernels for tf.... … ✕ **History**

70 contributors +44

3415 lines (3082 sloc) | 119 KB ···

```
/* Copyright 2015 The TensorFlow Authors. All Rights Reserved.

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

    http://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.
==============================================================================*/

#include <algorithm>
#include <ostream>

#include "tensorflow/core/framework/common_shape_fns.h"
#include "tensorflow/core/framework/kernel_shape_util.h"
#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/shape_inference.h"
#include "tensorflow/core/framework/tensor.pb.h"
#include "tensorflow/core/framework/types.h"
#include "tensorflow/core/framework/types.pb.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/util/mirror_pad_mode.h"
#include "tensorflow/core/util/padding.h"
#include "tensorflow/core/util/strided_slice_op.h"
```

```cpp
#include "tensorflow/core/util/tensor_format.h"

namespace tensorflow {

using shape_inference::DimensionHandle;
using shape_inference::InferenceContext;
using shape_inference::ShapeHandle;
using shape_inference::UnchangedShape;

namespace {

Status GetAxisForPackAndUnpack(InferenceContext* c, int32_t rank_after_pack,
                               int32* axis) {
  TF_RETURN_IF_ERROR(c->GetAttr("axis", axis));
  if (*axis < -1 * rank_after_pack || *axis >= rank_after_pack) {
    return errors::InvalidArgument("Invalid axis: ", *axis, "; must be in [",
                                   -1 * rank_after_pack, ",", rank_after_pack,
                                   ")");
  }
  if (*axis < 0) *axis = (rank_after_pack + *axis);
  return Status::OK();
}

template <typename T>
std::vector<int64_t> AsInt64(const Tensor* tensor, int64_t num_elements) {
  std::vector<int64_t> ret(num_elements);
  auto data = tensor->vec<T>();
  for (int64_t i = 0; i < num_elements; ++i) {
    ret[i] = data(i);
  }
  return ret;
}

template <typename T>
Status PadKnown(InferenceContext* c, ShapeHandle input,
                const Tensor* paddings_t, int64_t num_dims) {
  // paddings_t is known.
  std::vector<DimensionHandle> dims(num_dims);
  auto paddings_data = paddings_t->matrix<T>();
  for (int64_t i = 0; i < num_dims; ++i) {
    const T pad0 = paddings_data(i, 0);
    const T pad1 = paddings_data(i, 1);
    if (pad0 < 0 || pad1 < 0) {
      return errors::InvalidArgument("Paddings must be non-negative");
    }
    TF_RETURN_IF_ERROR(c->Add(c->Dim(input, i), pad0 + pad1, &dims[i]));
  }
  c->set_output(0, c->MakeShape(dims));
  return Status::OK();
```

```
}

Status PadShapeFn(InferenceContext* c) {
  // Paddings is a matrix of [input_rank, 2].
  ShapeHandle paddings;
  TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 2, &paddings));
  DimensionHandle unused;
  TF_RETURN_IF_ERROR(c->WithValue(c->Dim(paddings, 1), 2, &unused));

  // n_dim and input.rank are equivalent.
  ShapeHandle input = c->input(0);
  DimensionHandle n_dim = c->Dim(paddings, 0);
  if (c->ValueKnown(n_dim)) {
    TF_RETURN_IF_ERROR(c->WithRank(input, c->Value(n_dim), &input));
  } else if (c->RankKnown(input)) {
    TF_RETURN_IF_ERROR(c->WithValue(n_dim, c->Rank(input), &n_dim));
  }

  const Tensor* paddings_t = c->input_tensor(1);

  // paddings_t is unknown
  if (paddings_t == nullptr) {
    if (c->ValueKnown(n_dim)) {
      // Make output with n_dim unknown dims.
      c->set_output(0, c->UnknownShapeOfRank(c->Value(n_dim)));
    } else {
      c->set_output(0, c->UnknownShape());
    }
    return Status::OK();
  }

  const int64_t num_dims = paddings_t->shape().dim_size(0);
  TF_RETURN_IF_ERROR(c->WithRank(input, num_dims, &input));
  TF_RETURN_IF_ERROR(c->WithValue(n_dim, num_dims, &n_dim));

  if (paddings_t->dtype() == DT_INT32) {
    return PadKnown<int32>(c, input, paddings_t, num_dims);
  } else {
    return PadKnown<int64_t>(c, input, paddings_t, num_dims);
  }
}

Status TransposeShapeFn(InferenceContext* c) {
  ShapeHandle input = c->input(0);
  ShapeHandle perm_shape = c->input(1);
  const Tensor* perm = c->input_tensor(1);
  DimensionHandle perm_elems = c->NumElements(perm_shape);
  // If we don't have rank information on the input or value information on
  // perm we can't return any shape information, otherwise we have enough
  // information to at least find the rank of the output.
  if (!c->RankKnown(input) && !c->ValueKnown(perm_elems) && perm == nullptr) {
    c->set_output(0, c->UnknownShape());
    return Status::OK();
  }

  // Find our value of the rank.
  int64_t rank;
  if (c->RankKnown(input)) {
    rank = c->Rank(input);
  } else if (c->ValueKnown(perm_elems)) {
    rank = c->Value(perm_elems);
  } else {
    rank = perm->NumElements();
  }
  if (!c->RankKnown(input) && rank < 2) {
    // A permutation array containing a single element is ambiguous. It could
    // indicate either a scalar or a 1-dimensional array, both of which the
    // transpose op returns unchanged.
    c->set_output(0, input);
    return Status::OK();
  }

  std::vector<DimensionHandle> dims;
  dims.resize(rank);
  TF_RETURN_IF_ERROR(c->WithRank(input, rank, &input));
  // Ensure that perm is a vector and has rank elements.
  TF_RETURN_IF_ERROR(c->WithRank(perm_shape, 1, &perm_shape));
  TF_RETURN_IF_ERROR(c->WithValue(perm_elems, rank, &perm_elems));

  // If we know the rank of the input and the value of perm, we can return
  // all shape information, otherwise we can only return rank information,
  // but no information for the dimensions.
  if (perm != nullptr) {
    std::vector<int64_t> data;
    if (perm->dtype() == DT_INT32) {
      data = AsInt64<int32>(perm, rank);
    } else {
      data = AsInt64<int64_t>(perm, rank);
    }

    for (int32_t i = 0; i < rank; ++i) {
      int64_t in_idx = data[i];
      if (in_idx >= rank || in_idx <= -rank) {
        return errors::InvalidArgument("perm dim ", in_idx,
                                       " is out of range of input rank ", rank);
      }
      dims[i] = c->Dim(input, in_idx);
    }
```

```cpp
  } else {
    for (int i = 0; i < rank; ++i) {
      dims[i] = c->UnknownDim();
    }
  }

  c->set_output(0, c->MakeShape(dims));
  return Status::OK();
}

Status SetOutputShapeForReshape(InferenceContext* c) {
  ShapeHandle in = c->input(0);
  ShapeHandle out;
  TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(1, &out));

  if (!c->RankKnown(out)) {
    // We have no information about the shape of the output.
    c->set_output(0, out);
    return Status::OK();
  }
  if (c->RankKnown(in)) {
    // We don't know the number of output elements, but we can try to infer
    // the missing dimension.
    bool too_many_unknown = false;
    int32_t out_unknown_idx = -1;

    DimensionHandle known_out_elems = c->NumElements(out);
    if (!c->ValueKnown(known_out_elems)) {
      known_out_elems = c->MakeDim(1);
      for (int32_t i = 0; i < c->Rank(out); ++i) {
        DimensionHandle dim = c->Dim(out, i);
        if (!c->ValueKnown(dim)) {
          if (out_unknown_idx >= 0) {
            too_many_unknown = true;
            break;
          }
          out_unknown_idx = i;
        } else {
          TF_RETURN_IF_ERROR(
              c->Multiply(known_out_elems, dim, &known_out_elems));
        }
      }
    }
    int32_t in_unknown_idx = -1;
    DimensionHandle known_in_elems = c->NumElements(in);
    if (!c->ValueKnown(known_in_elems)) {
      known_in_elems = c->MakeDim(1);
      for (int32_t i = 0; i < c->Rank(in); ++i) {
        DimensionHandle dim = c->Dim(in, i);
        if (!c->ValueKnown(dim)) {
          if (in_unknown_idx >= 0) {
            too_many_unknown = true;
            break;
          }
          in_unknown_idx = i;
        } else {
          TF_RETURN_IF_ERROR(c->Multiply(known_in_elems, dim, &known_in_elems));
        }
      }
    }

    if (!too_many_unknown) {
      if (in_unknown_idx < 0 && out_unknown_idx < 0) {
        // Just check that the dimensions match.
        if (c->Value(known_in_elems) != c->Value(known_out_elems)) {
          return errors::InvalidArgument(
              "Cannot reshape a tensor with ", c->DebugString(known_in_elems),
              " elements to shape ", c->DebugString(out), " (",
              c->DebugString(known_out_elems), " elements)");
        }
      } else if (in_unknown_idx < 0 && out_unknown_idx >= 0 &&
                 c->Value(known_out_elems) > 0) {
        // Input fully known, infer the one missing output dim
        DimensionHandle inferred_dim;
        TF_RETURN_IF_ERROR(c->Divide(known_in_elems, c->Value(known_out_elems),
                                     true /* evenly_divisible */,
                                     &inferred_dim));
        TF_RETURN_IF_ERROR(
            c->ReplaceDim(out, out_unknown_idx, inferred_dim, &out));

      } else if (in_unknown_idx >= 0 && out_unknown_idx < 0 &&
                 c->Value(known_in_elems) != 0) {
        // Output fully known, infer the one missing input dim
        DimensionHandle inferred_dim;
        TF_RETURN_IF_ERROR(c->Divide(known_out_elems, c->Value(known_in_elems),
                                     true /* evenly_divisible */,
                                     &inferred_dim));
        DimensionHandle unknown_in_dim = c->Dim(in, in_unknown_idx);
        TF_RETURN_IF_ERROR(
            c->Merge(unknown_in_dim, inferred_dim, &unknown_in_dim));
      } else if (in_unknown_idx >= 0 && out_unknown_idx >= 0) {
        // Exactly one unknown dimension in both input and output. These 2 are
        // equal iff the known elements are equal.
        if (c->Value(known_in_elems) == c->Value(known_out_elems)) {
          DimensionHandle unknown_in_dim = c->Dim(in, in_unknown_idx);
          TF_RETURN_IF_ERROR(
              c->ReplaceDim(out, out_unknown_idx, unknown_in_dim, &out));
        }
      }
    }
  }
  c->set_output(0, out);
  return Status::OK();
}

}  // namespace

REGISTER_OP("ParallelConcat")
    .Input("values: N * T")
    .Output("output: T")
    .Attr("N: int >= 1")
    .Attr("T: type")
    .Attr("shape: shape")
    .SetShapeFn([](InferenceContext* c) {
      // Validate that the shape attr is correct.
      PartialTensorShape shape;
      TF_RETURN_IF_ERROR(c->GetAttr("shape", &shape));
      ShapeHandle passed_shape;
      TF_RETURN_IF_ERROR(
          c->MakeShapeFromPartialTensorShape(shape, &passed_shape));
      if (!c->FullyDefined(passed_shape)) {
        return errors::InvalidArgument("shape attr must be fully defined.");
      }
      ShapeHandle cur;
      TF_RETURN_IF_ERROR(c->ReplaceDim(
          passed_shape, 0, c->MakeDim(shape_inference::DimensionOrConstant(1)),
          &cur));
      for (int i = 0; i < c->num_inputs(); ++i) {
        if (!c->FullyDefined(c->input(i))) {
          return errors::InvalidArgument(
              "All input shapes must be fully defined.");
        }
        DimensionHandle unused;
        if (!c->WithValue(c->Dim(c->input(i), 0), 1, &unused).ok()) {
          return errors::InvalidArgument("Size of first dimension must be 1.");
        }
        TF_RETURN_WITH_CONTEXT_IF_ERROR(c->Merge(c->input(i), cur, &cur),
                                        "From merging shape ", i,
                                        " with other shapes.");
      }

      c->set_output(0, passed_shape);

      return Status::OK();
    });

REGISTER_OP("Pack")
```

```
    .Input("values: N * T")
    .Output("output: T")
    .Attr("N: int >= 1")
    .Attr("T: type")
    .Attr("axis: int = 0")
    .SetShapeFn([](InferenceContext* c) {
      // Validate shapes of all inputs are compatible
      ShapeHandle cur = c->input(c->num_inputs() - 1);
      for (int i = c->num_inputs() - 2; i >= 0; --i) {
        TF_RETURN_WITH_CONTEXT_IF_ERROR(c->Merge(c->input(i), cur, &cur),
                                        "From merging shape ", i,
                                        " with other shapes.");
      }
      if (!c->RankKnown(cur)) {
        c->set_output(0, c->UnknownShape());
        return Status::OK();
      }
      // Determine the axis that will be added, converting from negative
      // axes to a positive point per negative indexing rules.
      int32_t rank = c->Rank(cur);
      int32_t axis;
      TF_RETURN_IF_ERROR(GetAxisForPackAndUnpack(c, rank + 1, &axis));

      // Copy all dimensions over, inserting a dimension of value #inputs
      // at <axis>.
      std::vector<DimensionHandle> dims;
      int index = 0;
      while (index < axis) dims.push_back(c->Dim(cur, index++));
      dims.push_back(c->MakeDim(c->num_inputs()));
      while (index < rank) dims.push_back(c->Dim(cur, index++));

      c->set_output(0, c->MakeShape(dims));
      for (int i = 0; i < c->num_inputs(); ++i) {
        auto* shape_and_type = c->input_handle_shapes_and_types(i);
        if (shape_and_type) {
          if (!c->RelaxOutputHandleShapesAndMergeTypes(0, *shape_and_type)) {
            c->set_output_handle_shapes_and_types(
                0, std::vector<shape_inference::ShapeAndType>({}));
            break;
          }
        }
      }
      return Status::OK();
    });

REGISTER_OP("DeepCopy")
    .Input("x: T")
    .Output("y: T")
    .Attr("T: type")
    .SetIsStateful()
    .SetShapeFn(UnchangedShape);

REGISTER_OP("InplaceUpdate")
    .Input("x: T")
    .Input("i: int32")
    .Input("v: T")
    .Output("y: T")
    .Attr("T: type")
    .SetShapeFn(UnchangedShape);

REGISTER_OP("InplaceAdd")
    .Input("x: T")
    .Input("i: int32")
    .Input("v: T")
    .Output("y: T")
    .Attr("T: type")
    .SetShapeFn(UnchangedShape);

REGISTER_OP("InplaceSub")
    .Input("x: T")
    .Input("i: int32")
    .Input("v: T")
    .Output("y: T")
    .Attr("T: type")
    .SetShapeFn(UnchangedShape);

REGISTER_OP("Empty")
    .Input("shape: int32")
    .Output("output: dtype")
    .Attr("dtype: type")
    .Attr("init: bool = false")
    .SetDoNotOptimize()
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &out));
      c->set_output(0, out);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Unpack")
    .Input("value: T")
    .Output("output: num * T")
    .Attr("num: int >= 0")
    .Attr("T: type")
    .Attr("axis: int = 0")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle s = c->input(0);
      ShapeHandle out;
      if (c->RankKnown(s)) {
        // Determine the axis that will be removed, converting from negative
        // axes to a positive point per negative indexing rules.
        int32_t rank = c->Rank(s);
        int32_t axis;
        TF_RETURN_IF_ERROR(GetAxisForPackAndUnpack(c, rank, &axis));

        // The axis dim matches the number of outputs.
        DimensionHandle unused;
        TF_RETURN_IF_ERROR(
            c->WithValue(c->Dim(s, axis), c->num_outputs(), &unused));

        // Copy all dimensions, removing the <axis> dimension.
        std::vector<DimensionHandle> dims;
        for (int i = 0; i < rank; ++i) {
          if (i != axis) dims.push_back(c->Dim(s, i));
        }
        out = c->MakeShape(dims);
      } else {
        // All outputs are the same shape, but it's not known.
        out = c->UnknownShape();
      }
      for (int i = 0; i < c->num_outputs(); ++i) c->set_output(i, out);
      return Status::OK();
    });

REGISTER_OP("UnravelIndex")
    .Input("indices: Tidx")
    .Input("dims: Tidx")
    .Output("output: Tidx")
    .Attr("Tidx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle indices = c->input(0);
      ShapeHandle dims;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &dims));
      if (c->RankKnown(indices) && c->Rank(indices) == 0) {
        c->set_output(0, c->Vector(c->Dim(dims, 0)));
      } else if (c->RankKnown(indices)) {
        c->set_output(0, c->Matrix(c->Dim(dims, 0), c->NumElements(indices)));
      } else {
        c->set_output(0, c->UnknownShape());
      }
      return Status::OK();
    });

REGISTER_OP("BroadcastTo")
    .Input("input: T")
    .Input("shape: Tidx")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tidx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle shape_in = c->input(1);
      TF_RETURN_IF_ERROR(c->WithRank(shape_in, 1, &shape_in));
      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(1, &out));
      if (!c->RankKnown(out)) {
        // We have no information about the shape of the output.
        c->set_output(0, out);
        return Status::OK();
      }

      ShapeHandle in = c->input(0);
      if (!c->RankKnown(in)) {
        // We have no information about the shape of the input,
        // nothing to do here.
        c->set_output(0, out);
        return Status::OK();
      }
      int out_rank = c->Rank(out);
      TF_RETURN_IF_ERROR(c->WithRankAtMost(in, out_rank, &in));
      int in_rank = c->Rank(in);
      for (int i = 0; i < in_rank; ++i) {
        auto in_dim = c->Dim(in, in_rank - i - 1);
        if (c->Value(in_dim) > 1) {
          // If the input dimension is greater than 1 then the output dimension
          // must be equal to it, since we only broadcast "from left to right".
          auto out_dim = c->Dim(out, out_rank - i - 1);
          TF_RETURN_IF_ERROR(c->Merge(in_dim, out_dim, &out_dim));
          TF_RETURN_IF_ERROR(
              c->ReplaceDim(out, out_rank - i - 1, out_dim, &out));
        }
      }
      c->set_output(0, out);
      return Status::OK();
    });

// --------------------------------------------------------------------------
// TODO(josh11b): Remove the >= 2 constraint, once we can rewrite the graph
// in the N == 1 case to remove the node.
REGISTER_OP("Concat")
    .Input("concat_dim: int32")
    .Input("values: N * T")
    .Output("output: T")
    .Attr("N: int >= 2")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      return shape_inference::ConcatShape(c, c->num_inputs() - 1);
    });

REGISTER_OP("ConcatV2")
    .Input("values: N * T")
    .Input("axis: Tidx")
    .Output("output: T")
    .Attr("N: int >= 2")
    .Attr("T: type")
    .Attr("Tidx: {int32, int64} = DT_INT32")
    .SetShapeFn(shape_inference::ConcatV2Shape);

// TODO(vivek.v.rane@intel.com): Prefix the op names with underscore if the ops
// are not to be made user-accessible.
#ifdef INTEL_MKL
REGISTER_OP("_MklConcatV2")
    .Input("values: N * T")
    .Input("axis: Tidx")
    .Input("mkl_values: N * uint8")
    .Input("mkl_axis: uint8")
    .Output("output: T")
    .Output("mkl_output: uint8")
    .Attr("N: int >= 2")
    .Attr("T: type")
    .Attr("Tidx: {int32, int64} = DT_INT32")
    .SetShapeFn(shape_inference::ConcatV2Shape)
    .Doc(R"doc(
MKL version of ConcatV2 operator. Uses MKL DNN APIs to perform concatenation.

NOTE Do not invoke this operator directly in Python. Graph rewrite pass is
expected to invoke these operators.
)doc");
#endif

REGISTER_OP("ConcatOffset")
    .Input("concat_dim: int32")
    .Input("shape: N * int32")
    .Output("offset: N * int32")
    .Attr("N: int >= 2")
    .SetShapeFn([](InferenceContext* c) {
      for (int i = 1; i < c->num_inputs(); ++i) {
        c->set_output(i - 1, c->input(i));
      }
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Split")
    .Input("split_dim: int32")
    .Input("value: T")
    .Output("output: num_split * T")
    .Attr("num_split: int >= 1")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      DimensionHandle split_dimension;
      ShapeHandle input = c->input(1);
      TF_RETURN_IF_ERROR(c->MakeDimForScalarInputWithNegativeIndexing(
          0, c->Rank(input), &split_dimension));
      int num_split = c->num_outputs();
      ShapeHandle out;
      if (!c->ValueKnown(split_dimension)) {
        if (c->RankKnown(input)) {
          out = c->UnknownShapeOfRank(c->Rank(input));
        } else {
          out = c->UnknownShape();
        }
      } else {
        int64_t split_dim = c->Value(split_dimension);
        TF_RETURN_IF_ERROR(c->WithRankAtLeast(input, split_dim + 1, &input));
        DimensionHandle split_dim_size;
        TF_RETURN_WITH_CONTEXT_IF_ERROR(
            c->Divide(c->Dim(input, split_dim), num_split,
                      true /* evenly_divisible */, &split_dim_size),
            "Number of ways to split should evenly divide the split dimension");
        TF_RETURN_IF_ERROR(
            c->ReplaceDim(input, split_dim, split_dim_size, &out));
      }
      for (int i = 0; i < num_split; ++i) c->set_output(i, out);
      return Status::OK();
    });

REGISTER_OP("SplitV")
    .Input("value: T")
    .Input("size_splits: Tlen")
    .Input("split_dim: int32")
    .Output("output: num_split * T")
    .Attr("num_split: int >= 1")
    .Attr("T: type")
    .Attr("Tlen: {int32, int64} = DT_INT64")
    .SetShapeFn([](InferenceContext* c) {
      DimensionHandle split_dimension;
      ShapeHandle input = c->input(0);
      TF_RETURN_IF_ERROR(c->MakeDimForScalarInputWithNegativeIndexing(
          2, c->Rank(input), &split_dimension));
      int32_t num_outputs = c->num_outputs();
      int32_t rank = c->Rank(input);
      ShapeHandle output_shape;
      const Tensor* size_splits = c->input_tensor(1);
      if (rank == InferenceContext::kUnknownRank) {
        // If the rank of input tensor is unknown, then return unknown shapes.
        // Note that the shape of each output can be different.
        for (int i = 0; i < num_outputs; ++i) {
          c->set_output(i, c->UnknownShape());
        }
      } else if (rank == 0) {
        // Throw error if input is a scalar.
        return errors::InvalidArgument("Can't split scalars");
      } else if (size_splits == nullptr && c->ValueKnown(split_dimension)) {
        // If split dimension is known, but the sizes are unknown, then
        // only the split dimension is unknown
        output_shape = input;
        for (int i = 0; i < num_outputs; ++i) {
          TF_RETURN_IF_ERROR(c->ReplaceDim(output_shape,
                                           c->Value(split_dimension),
                                           c->UnknownDim(), &output_shape));
          c->set_output(i, output_shape);
        }
      } else if (size_splits == nullptr && !c->ValueKnown(split_dimension)) {
        // If split dimension or tensor containing the split sizes is unknown,
        // then return unknown shapes of same rank as input. Note that each
        // output shape can be different since splitv doesn't always split
        // tensors evenly.
        for (int i = 0; i < num_outputs; ++i) {
          c->set_output(i, c->UnknownShapeOfRank(rank));
        }
      } else {
        // Determine the output shape if split dimension and split sizes are
        // known.
        int64_t split_dim = c->Value(split_dimension);
        TF_RETURN_IF_ERROR(c->WithRankAtLeast(input, split_dim + 1, &input));
        std::vector<int64_t> data;
        if (size_splits->dtype() == DT_INT32) {
          data = AsInt64<int32>(size_splits, size_splits->shape().dim_size(0));
        } else {
          data =
              AsInt64<int64_t>(size_splits, size_splits->shape().dim_size(0));
        }
        if (num_outputs != data.size()) {
          return errors::InvalidArgument(
              "Length of size_splits should be equal to num_outputs");
        }
        int64_t total_size = 0;
        bool has_neg_one = false;
        for (const auto size : data) {
          if (size == -1) {
            if (has_neg_one) {
              return errors::InvalidArgument(
                  "size_splits can only have one -1");
            }
            has_neg_one = true;
          } else {
            total_size += size;
          }
        }
        auto split_dim_size = c->Value(c->Dim(input, split_dim));
        // If the sizes of the splits are known, then
        // make sure that the sizes add up to the expected
        // dimension size, with the possibility of a -1.
        // Specify the full output shapes.
        for (int i = 0; i < num_outputs; ++i) {
          auto size = data[i];
          if (data[i] == -1 && c->ValueKnown(split_dim_size)) {
            size = split_dim_size - total_size;
          }
          // If we have a negative known size (either explicit, or computed
          // via -1), then the split sizes are invalid.
          if (size < -1 || (size == -1 && c->ValueKnown(split_dim_size))) {
            return errors::InvalidArgument("Split size at index ", i,
                                           " must be >= 0. Got: ", size);
          }
          TF_RETURN_IF_ERROR(
              c->ReplaceDim(input, split_dim, c->MakeDim(size), &output_shape));
          c->set_output(i, output_shape);
        }
        if (c->ValueKnown(split_dim_size)) {
          if (has_neg_one ? total_size > split_dim_size
                          : total_size != split_dim_size) {
            return errors::InvalidArgument(
                "can't split axis of size ", split_dim_size,
                " into pieces of size [", absl::StrJoin(data, ","), "]");
          }
        }
      }

      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Const")
    .Output("output: dtype")
    .Attr("value: tensor")
    .Attr("dtype: type")
    .SetShapeFn([](InferenceContext* c) {
      const TensorProto* proto = nullptr;
      TF_RETURN_IF_ERROR(c->GetAttr("value", &proto));
      TF_RETURN_IF_ERROR(TensorShape::IsValidShape(proto->tensor_shape()));
      TensorShape shape(proto->tensor_shape());
      std::vector<DimensionHandle> dims;
      dims.reserve(shape.dims());
      for (int i = 0; i < shape.dims(); ++i) {
        dims.push_back(c->MakeDim(shape.dim_size(i)));
      }
      c->set_output(0, c->MakeShape(dims));
      return Status::OK();
    });

// Returns a constant tensor on the host.  Useful for writing C++ tests
// and benchmarks which run on GPU but require arguments pinned to the host.
// Used by test::graph::HostConstant.
// value: Attr `value` is the tensor to return.
REGISTER_OP("HostConst")
    .Output("output: dtype")
    .Attr("value: tensor")
    .Attr("dtype: type")
    .SetShapeFn(shape_inference::UnknownShape);

// Used executing op-by-op to copy constants to the current device without
// serializing tensors as TensorProtos, after a host tensor has been
// created. Same behavior as Identity, but no gradient and potentially relaxed
// copy semantics.
REGISTER_OP("_EagerConst")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape);

// --------------------------------------------------------------------------
// TODO(mgubin): Update the doc when the freeze_graph script supports converting
// into memmapped format.
REGISTER_OP("ImmutableConst")
    .Attr("dtype: type")
    .Attr("shape: shape")
    .Attr("memory_region_name: string")
    .Output("tensor: dtype")
    .SetShapeFn(shape_inference::ExplicitShape);

REGISTER_OP("GuaranteeConst")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn([](shape_inference::InferenceContext* c) {
      return UnchangedShape(c);
    })
    // We don't want this to be optimized away.
    .SetDoNotOptimize();
```

```

// --------------------------------------------------------------------------
REGISTER_OP("ZerosLike")
    .Input("x: T")
    .Output("y: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape);

// --------------------------------------------------------------------------
REGISTER_OP("OnesLike")
    .Input("x: T")
    .Output("y: T")
    .Attr(
        "T: {bfloat16, half, float, double, int8, uint8, int16, uint16, int32, "
        "uint32, int64, uint64, complex64, complex128, bool}")
    .SetShapeFn(shape_inference::UnchangedShape);

// --------------------------------------------------------------------------
REGISTER_OP("Diag")
    .Input("diagonal: T")
    .Output("output: T")
    .Attr(
        "T: {bfloat16, half, float, double, int32, int64, complex64, "
        "complex128}")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle in = c->input(0);
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(in, 1, &in));
      // Output shape is original concatenated with itself.
      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->Concatenate(in, in, &out));
      c->set_output(0, out);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("DiagPart")
    .Input("input: T")
    .Output("diagonal: T")
    .Attr(
        "T: {bfloat16, half, float, double, int32, int64, complex64, "
        "complex128}")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle in = c->input(0);
      if (!c->RankKnown(in)) {
        c->set_output(0, c->UnknownShape());
        return Status::OK();
      }
      // Rank must be even, and result will have rank <rank/2>.
      const int32_t rank = c->Rank(in);
      if ((rank % 2) != 0 || rank <= 0) {
        return errors::InvalidArgument(
            "Input must have even and non-zero rank, input rank is ", rank);
      }
      const int32_t mid = rank / 2;

      // output dim[i] is the merge of in.dim[i] and in.dim[i+mid].
      std::vector<DimensionHandle> dims(mid);
      for (int i = 0; i < mid; ++i) {
        TF_RETURN_IF_ERROR(
            c->Merge(c->Dim(in, i), c->Dim(in, i + mid), &dims[i]));
      }
      c->set_output(0, c->MakeShape(dims));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("MatrixDiag")
    .Input("diagonal: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle in;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &in));
      if (!c->RankKnown(in)) {
        c->set_output(0, c->UnknownShape());
        return Status::OK();
      }
      const int32_t rank = c->Rank(in);
      ShapeHandle out;
      TF_RETURN_IF_ERROR(
          c->Concatenate(in, c->Vector(c->Dim(in, rank - 1)), &out));
      c->set_output(0, out);
      return Status::OK();
    });

REGISTER_OP("MatrixDiagV2")
    .Input("diagonal: T")
    .Input("k: int32")
    .Input("num_rows: int32")
    .Input("num_cols: int32")
    .Input("padding_value: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::MatrixDiagV2Shape);

REGISTER_OP("MatrixDiagV3")
    .Input("diagonal: T")
    .Input("k: int32")
    .Input("num_rows: int32")
    .Input("num_cols: int32")
    .Input("padding_value: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr(
        "align: {'LEFT_RIGHT', 'RIGHT_LEFT', 'LEFT_LEFT', 'RIGHT_RIGHT'} = "
        "'RIGHT_LEFT'")
    .SetShapeFn(shape_inference::MatrixDiagV2Shape);

// --------------------------------------------------------------------------
REGISTER_OP("MatrixSetDiag")
    .Input("input: T")
    .Input("diagonal: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input;
      ShapeHandle diag;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &input));
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 1, &diag));
      if (c->RankKnown(input)) {
        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), c->Rank(input) - 1, &diag));
      }
      DimensionHandle smallest_dim;
      TF_RETURN_IF_ERROR(
          c->Min(c->Dim(input, -2), c->Dim(input, -1), &smallest_dim));
      TF_RETURN_IF_ERROR(
          c->Merge(smallest_dim, c->Dim(diag, -1), &smallest_dim));

      ShapeHandle output = input;
      if (c->RankKnown(diag) && !c->FullyDefined(input)) {
        // Try to infer parts of shape from diag.
        ShapeHandle diag_batch_shape;
        TF_RETURN_IF_ERROR(c->Subshape(diag, 0, -1, &diag_batch_shape));
        TF_RETURN_IF_ERROR(
            c->Concatenate(diag_batch_shape, c->UnknownShapeOfRank(2), &diag));
        TF_RETURN_IF_ERROR(c->Merge(input, diag, &output));
      }
      c->set_output(0, output);
      return Status::OK();
    });

REGISTER_OP("MatrixSetDiagV2")
    .Input("input: T")
    .Input("diagonal: T")
    .Input("k: int32")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::MatrixSetDiagV2Shape);

REGISTER_OP("MatrixSetDiagV3")
    .Input("input: T")
    .Input("diagonal: T")
    .Input("k: int32")
    .Output("output: T")
    .Attr("T: type")
    .Attr(
        "align: {'LEFT_RIGHT', 'RIGHT_LEFT', 'LEFT_LEFT', 'RIGHT_RIGHT'} = "
        "'RIGHT_LEFT'")
    .SetShapeFn(shape_inference::MatrixSetDiagV2Shape);

// --------------------------------------------------------------------------
REGISTER_OP("MatrixDiagPart")
    .Input("input: T")
    .Output("diagonal: T")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle in;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &in));
      if (!c->RankKnown(in)) {
        c->set_output(0, c->UnknownShape());
        return Status::OK();
      }
      const int32_t rank = c->Rank(in);
      std::vector<DimensionHandle> dims;
      dims.reserve(rank - 2);
      for (int i = 0; i < rank - 2; ++i) dims.push_back(c->Dim(in, i));

      DimensionHandle min_dim;
      TF_RETURN_IF_ERROR(
          c->Min(c->Dim(in, rank - 2), c->Dim(in, rank - 1), &min_dim));
      dims.push_back(min_dim);
      c->set_output(0, c->MakeShape(dims));
      return Status::OK();
    });

REGISTER_OP("MatrixDiagPartV2")
    .Input("input: T")
    .Input("k: int32")
    .Input("padding_value: T")
    .Output("diagonal: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::MatrixDiagPartV2Shape);

REGISTER_OP("MatrixDiagPartV3")
    .Input("input: T")
    .Input("k: int32")
    .Input("padding_value: T")
    .Output("diagonal: T")
    .Attr("T: type")
    .Attr(
        "align: {'LEFT_RIGHT', 'RIGHT_LEFT', 'LEFT_LEFT', 'RIGHT_RIGHT'} = "
        "'RIGHT_LEFT'")
    .SetShapeFn(shape_inference::MatrixDiagPartV2Shape);

// --------------------------------------------------------------------------
REGISTER_OP("MatrixBandPart")
    .Input("input: T")
    .Input("num_lower: Tindex")
    .Input("num_upper: Tindex")
    .Output("band: T")
    .Attr("T: type")
    .Attr("Tindex: {int32, int64} = DT_INT64")
    .SetShapeFn(shape_inference::UnchangedShape);

// --------------------------------------------------------------------------
REGISTER_OP("Reverse")
    .Input("tensor: T")
    .Input("dims: bool")
    .Output("output: T")
    .Attr(
        "T: {uint8, int8, uint16, int16, uint32, int32, uint64, int64, bool, "
        "bfloat16, half, float, double, complex64, complex128, string}")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);
      ShapeHandle dims;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &dims));
      DimensionHandle dims_dim = c->Dim(dims, 0);
      if (c->ValueKnown(dims_dim)) {
        TF_RETURN_IF_ERROR(c->WithRank(input, c->Value(dims_dim), &input));
      }
      if (c->Rank(input) > 8) {
        return errors::InvalidArgument(
            "reverse does not work on tensors with more than 8 dimensions");
      }
      c->set_output(0, input);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("ReverseV2")
    .Input("tensor: T")
    .Input("axis: Tidx")
    .Output("output: T")
    .Attr("Tidx: {int32, int64} = DT_INT32")
    .Attr(
        "T: {uint8, int8, uint16, int16, int32, uint32, int64, uint64, bool, "
        "bfloat16, half, float, double, complex64, complex128, string}")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);
      ShapeHandle axis;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &axis));
      if (c->Rank(input) > 8) {
        return errors::InvalidArgument(
            "reverse does not work on tensors with more than 8 dimensions");
      }
      const Tensor* axis_tensor = c->input_tensor(1);
      if (axis_tensor != nullptr && c->RankKnown(input)) {
        int32_t rank = c->Rank(input);
        std::vector<int64_t> axis_value;
        if (axis_tensor->dtype() == DT_INT32) {
          axis_value = AsInt64<int32>(axis_tensor, axis_tensor->NumElements());
        } else {
          axis_value =
              AsInt64<int64_t>(axis_tensor, axis_tensor->NumElements());
        }
        std::vector<bool> axes_dense(c->Rank(input), false);
        for (int i = 0; i < axis_value.size(); i++) {
          int64_t canonical_axis =
              axis_value[i] < 0 ? rank + axis_value[i] : axis_value[i];
          if (canonical_axis < 0 || canonical_axis >= rank) {
            return errors::InvalidArgument("'axis'[", i, "] = ", axis_value[i],
                                           " is out of valid range [", 0, ", ",
                                           rank - 1);
          }
          if (axes_dense[canonical_axis]) {
            return errors::InvalidArgument("axis ", canonical_axis,
                                           " specified more than once.");
          }
          axes_dense[canonical_axis] = true;
        }
      }
      c->set_output(0, input);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("EditDistance")
    .Input("hypothesis_indices: int64")
    .Input("hypothesis_values: T")
    .Input("hypothesis_shape: int64")
    .Input("truth_indices: int64")
    .Input("truth_values: T")
    .Input("truth_shape: int64")
    .Attr("normalize: bool = true")
    .Attr("T: type")
    .Output("output: float")
    .SetShapeFn([](InferenceContext* c) {
      TF_RETURN_IF_ERROR(shape_inference::ValidateSparseTensor(
          c, c->input(0), c->input(1), c->input(2)));
      TF_RETURN_IF_ERROR(shape_inference::ValidateSparseTensor(
          c, c->input(3), c->input(4), c->input(5)));
      const Tensor* hypothesis_shape_t = c->input_tensor(2);
      const Tensor* truth_shape_t = c->input_tensor(5);
      if (hypothesis_shape_t == nullptr || truth_shape_t == nullptr) {
        // We need to know the runtime shape of the two tensors,
        // or else the output shape is unknown.
        return shape_inference::UnknownShape(c);
      }

      if (hypothesis_shape_t->NumElements() != truth_shape_t->NumElements()) {
        return errors::InvalidArgument(
            "Num elements of hypothesis_shape does not match truth_shape: ",
            hypothesis_shape_t->NumElements(), " vs. ",
            truth_shape_t->NumElements());
      }

      auto h_values = hypothesis_shape_t->flat<int64_t>();
      auto t_values = truth_shape_t->flat<int64_t>();
      std::vector<DimensionHandle> dims(hypothesis_shape_t->NumElements() - 1);
      for (int i = 0; i < dims.size(); ++i) {
        dims[i] = c->MakeDim(std::max(h_values(i), t_values(i)));
      }

      c->set_output(0, c->MakeShape(dims));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Fill")
    .Input("dims: index_type")
    .Input("value: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("index_type: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      DataType index_type = DT_INT32;
      Status s = c->GetAttr("index_type", &index_type);
      if (!s.ok() && s.code() != error::NOT_FOUND) {
        return s;
      }
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));

      const Tensor* t = c->input_tensor(0);
      if (t != nullptr) {
        for (int i = 0; i < t->NumElements(); ++i) {
          if ((index_type == DT_INT32 && t->vec<int32>()(i) < 0) ||
              (index_type == DT_INT64 && t->vec<int64_t>()(i) < 0)) {
            return errors::InvalidArgument("Fill dimensions must be >= 0");
          }
        }
      }

      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &out));
      c->set_output(0, out);

      auto* shape_and_type = c->input_handle_shapes_and_types(1);
      if (shape_and_type) {
        c->set_output_handle_shapes_and_types(0, *shape_and_type);
      }

      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("_ParallelConcatStart")
    .Output("output: dtype")
    .Attr("shape: shape")
    .Attr("dtype: type")
    .SetIsStateful()
    .SetShapeFn(shape_inference::ExplicitShape)
    .Doc(R"doc(
Creates an empty Tensor with shape `shape` and type `dtype`.

The memory can optionally be initialized. This is usually useful in
conjunction with inplace operations.

shape: 1-D `Tensor` indicating the shape of the output.
dtype: The element type of the returned tensor.
output: An empty Tensor of the specified type.
)doc");

// --------------------------------------------------------------------------
REGISTER_OP("_ParallelConcatUpdate")
    .Input("value: T")
    .Input("update: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("loc: int")
    .SetShapeFn(shape_inference::UnchangedShape)
    .Doc(R"doc(
Updates input `value` at `loc` with `update`.

If you use this function you will almost certainly want to add
a control dependency as done in the implementation of parallel_stack to
avoid race conditions.

value: A `Tensor` object that will be updated in-place.
loc: A scalar indicating the index of the first dimension such that
         value[loc, :] is updated.
update: A `Tensor` of rank one less than `value` if `loc` is a scalar,
        otherwise of rank equal to `value` that contains the new values
        for `value`.
output: `value` that has been updated accordingly.
)doc");

// --------------------------------------------------------------------------
REGISTER_OP("Gather")
    .Input("params: Tparams")
    .Input("indices: Tindices")
    .Attr("validate_indices: bool = true")
    .Output("output: Tparams")
    .Attr("Tparams: type")
    .Attr("Tindices: {int32,int64}")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &unused));
      ShapeHandle params_subshape;
      TF_RETURN_IF_ERROR(c->Subshape(c->input(0), 1, &params_subshape));
      ShapeHandle indices_shape = c->input(1);
      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->Concatenate(indices_shape, params_subshape, &out));
      c->set_output(0, out);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("GatherV2")
    .Input("params: Tparams")
    .Input("indices: Tindices")
    .Input("axis: Taxis")
    .Attr("batch_dims: int = 0")
    .Output("output: Tparams")
    .Attr("Tparams: type")
    .Attr("Tindices: {int32,int64}")
    .Attr("Taxis: {int32,int64}")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle params_shape;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &params_shape));

      ShapeHandle indices_shape = c->input(1);
      ShapeHandle unused_axis_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused_axis_shape));
      const Tensor* axis_t = c->input_tensor(2);

      // If axis is unknown, we can only infer that the result is params_rank +
      // indices_rank - 1.
      if (axis_t == nullptr) {
        if (c->RankKnown(params_shape) && c->RankKnown(indices_shape)) {
          int32_t batch_dims;
          TF_RETURN_IF_ERROR(c->GetAttr("batch_dims", &batch_dims));
          c->set_output(0, c->UnknownShapeOfRank(c->Rank(params_shape) +
                                                 c->Rank(indices_shape) - 1 -
                                                 batch_dims));
        } else {
          c->set_output(0, c->UnknownShape());
        }
        return Status::OK();
      }

      // Note, axis can be negative.
      int64_t axis = 0;
      if (axis_t->dtype() == DT_INT32) {
        axis = axis_t->scalar<int32>()();
      } else {
        axis = axis_t->scalar<int64_t>()();
      }

      // Check that params has rank of at least axis + 1.
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(
          params_shape, axis < 0 ? -axis : axis + 1, &unused));

      // Note, batch_dims can be negative.
      int32_t batch_dims;
      TF_RETURN_IF_ERROR(c->GetAttr("batch_dims", &batch_dims));
      // -rank(indices) <= batch_dims <= rank(indices)
      TF_RETURN_IF_ERROR(
          c->WithRankAtLeast(indices_shape, std::abs(batch_dims), &unused));
      if (batch_dims < 0) {
        batch_dims += c->Rank(indices_shape);
      }
      // rank(params) > batch_dims
      TF_RETURN_IF_ERROR(
          c->WithRankAtLeast(params_shape, batch_dims + 1, &unused));

      ShapeHandle params_outer_subshape;
      TF_RETURN_IF_ERROR(
          c->Subshape(params_shape, 0, axis, &params_outer_subshape));

      ShapeHandle indices_inner_subshape;
      TF_RETURN_IF_ERROR(
          c->Subshape(indices_shape, batch_dims, &indices_inner_subshape));

      ShapeHandle out;
      TF_RETURN_IF_ERROR(
          c->Concatenate(params_outer_subshape, indices_inner_subshape, &out));

      // Slice from axis + 1 to the end of params_shape to collect the inner
      // dimensions of the result. Special case -1 here since -1 + 1 wraps, and
      // we slice from 0 to the end of shape. Subshape() handles all other
      // out-of-bounds checking.
      if (axis != -1) {
        ShapeHandle params_inner_subshape;
        TF_RETURN_IF_ERROR(
            c->Subshape(params_shape, axis + 1, &params_inner_subshape));
        TF_RETURN_IF_ERROR(c->Concatenate(out, params_inner_subshape, &out));
      }

      c->set_output(0, out);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("GatherNd")
    .Input("params: Tparams")
    .Input("indices: Tindices")
    .Output("output: Tparams")
    .Attr("Tparams: type")
    .Attr("Tindices: {int32,int64}")
    .SetShapeFn(shape_inference::GatherNdShape);

// --------------------------------------------------------------------------
REGISTER_OP("Identity")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .SetForwardTypeFn(full_type::ReplicateInput())
    .SetShapeFn(shape_inference::UnchangedShape);

REGISTER_OP("Snapshot")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape);

#ifdef INTEL_MKL
```

```
REGISTER_OP("_MklIdentity")
    .Input("input: T")
    .Input("mkl_input: uint8")
    .Output("output: T")
    .Output("mkl_output: uint8")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape)
    .Doc(R"Doc( Mkl implementation of IdentityOp
)Doc");
#endif

REGISTER_OP("IdentityN")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: list(type)")
    .SetShapeFn([](shape_inference::InferenceContext* c) {
      std::vector<ShapeHandle> input;
      TF_RETURN_IF_ERROR(c->input("input", &input));
      TF_RETURN_IF_ERROR(c->set_output("output", input));
      // If any of the input shapes are not known, we should return error.
      for (int i = 0; i < input.size(); i++) {
        if (!input[i].Handle()) {
          return errors::InvalidArgument(absl::StrCat(
              "Cannot infer output shape #", i,
              " for IdentityN node because input shape #", i, " is unknown."));
        }
      }
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("RefIdentity")
    .Input("input: Ref(T)")
    .Output("output: Ref(T)")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape)
    .SetAllowsUninitializedInput();

// --------------------------------------------------------------------------
REGISTER_OP("DebugGradientIdentity")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape)
    .SetAllowsUninitializedInput();

REGISTER_OP("DebugGradientRefIdentity")
    .Input("input: Ref(T)")
    .Output("output: Ref(T)")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape)
    .SetAllowsUninitializedInput();

// --------------------------------------------------------------------------
REGISTER_OP("StopGradient")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .SetShapeFn(shape_inference::UnchangedShape);

REGISTER_OP("PreventGradient")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("message: string = ''")
    .SetShapeFn(shape_inference::UnchangedShape);

// --------------------------------------------------------------------------
REGISTER_OP("CheckNumerics")
    .Input("tensor: T")
    .Output("output: T")
    .Attr("T: {bfloat16, half, float, double}")
    .Attr("message: string")
    .SetIsStateful()
    .SetShapeFn(shape_inference::UnchangedShape);

// --------------------------------------------------------------------------
REGISTER_OP("CheckNumericsV2")
    .Input("tensor: T")
    .Output("output: T")
    .Attr("T: {bfloat16, half, float, double}")
    .Attr("message: string")
    .SetIsStateful()
    .SetShapeFn(shape_inference::UnchangedShape);

// --------------------------------------------------------------------------
REGISTER_OP("Reshape")
    .Input("tensor: T")
    .Input("shape: Tshape")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tshape: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      return SetOutputShapeForReshape(c);
    });

#ifdef INTEL_MKL
REGISTER_OP("_MklReshape")
```

```
    .Input("tensor: T")
    .Input("shape: Tshape")
    .Input("mkl_tensor: uint8")
    .Input("mkl_shape: uint8")
    .Output("output: T")
    .Output("mkl_output: uint8")
    .Attr("T: type")
    .Attr("Tshape: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) { return SetOutputShapeForReshape(c); })
    .Doc(R"Doc( MKL implementation of ReshapeOp.
)Doc");
#endif  // INTEL_MKL

// --------------------------------------------------------------------------
REGISTER_OP("InvertPermutation")
    .Input("x: T")
    .Output("y: T")
    .Attr("T: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle x;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &x));
      c->set_output(0, x);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Transpose")
    .Input("x: T")
    .Input("perm: Tperm")
    .Output("y: T")
    .Attr("T: type")
    .Attr("Tperm: {int32, int64} = DT_INT32")
    .SetShapeFn(TransposeShapeFn);

#ifdef INTEL_MKL
REGISTER_OP("_MklTranspose")
    .Input("x: T")
    .Input("perm: Tperm")
    .Output("y: T")
    .Attr("T: type")
    .Attr("Tperm: {int32, int64} = DT_INT32")
    .SetShapeFn(TransposeShapeFn);
#endif  // INTEL_MKL

// --------------------------------------------------------------------------
REGISTER_OP("ConjugateTranspose")
    .Input("x: T")
    .Input("perm: Tperm")
    .Output("y: T")
    .Attr("T: type")
    .Attr("Tperm: {int32, int64} = DT_INT32")
    .SetShapeFn(TransposeShapeFn);

#ifdef INTEL_MKL
REGISTER_OP("_MklConjugateTranspose")
    .Input("x: T")
    .Input("perm: Tperm")
    .Output("y: T")
    .Attr("T: type")
    .Attr("Tperm: {int32, int64} = DT_INT32")
    .SetShapeFn(TransposeShapeFn);
#endif  // INTEL_MKL

// --------------------------------------------------------------------------
namespace {
Status UniqueIdxShapeFn(InferenceContext* c) {
  ShapeHandle input = c->input(0);
  const Tensor* axis_t = c->input_tensor(1);
  if (axis_t == nullptr || !c->RankKnown(input)) {
    c->set_output(1, c->Vector(InferenceContext::kUnknownDim));
    return Status::OK();
  }

  if (c->Rank(c->input(1)) != 1) {
    return errors::InvalidArgument("axis expects a 1D vector.");
  }

  int32_t n = axis_t->NumElements();
  if (n == 0) {
    if (c->Rank(input) != 1) {
      return errors::InvalidArgument("x expects a 1D vector.");
    }
    c->set_output(1, input);
    return Status::OK();
  } else if (n == 1) {
    int64_t axis;
    if (axis_t->dtype() == DT_INT32) {
      axis = static_cast<int64_t>(axis_t->flat<int32>()(0));
    } else {
      axis = axis_t->flat<int64_t>()(0);
    }

    int64_t input_rank = c->Rank(input);
    if (axis < -input_rank || axis >= input_rank) {
      return errors::InvalidArgument("axis expects to be in the range [",
                                     -input_rank, ", ", input_rank, ")");
    }
    if (axis < 0) {
      axis += input_rank;
    }
    c->set_output(1, c->Vector(c->Dim(input, axis)));
    return Status::OK();
  }
  return errors::InvalidArgument(
      "axis does not support input tensors larger than 1 elements.");
}
}  // namespace

REGISTER_OP("Unique")
    .Input("x: T")
    .Output("y: T")
    .Output("idx: out_idx")
    .Attr("T: type")
    .Attr("out_idx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      c->set_output(0, c->Vector(InferenceContext::kUnknownDim));
      c->set_output(1, c->input(0));
      // Assert that the input rank is 1.
      ShapeHandle dummy;
      return c->WithRank(c->input(0), 1, &dummy);
    });

REGISTER_OP("UniqueV2")
    .Input("x: T")
    .Input("axis: Taxis")
    .Output("y: T")
    .Output("idx: out_idx")
    .Attr("T: type")
    .Attr("Taxis: {int32,int64} = DT_INT64")
    .Attr("out_idx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      c->set_output(0, c->UnknownShapeOfRank(c->Rank(c->input(0))));
      TF_RETURN_IF_ERROR(UniqueIdxShapeFn(c));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("UniqueWithCounts")
    .Input("x: T")
    .Output("y: T")
    .Output("idx: out_idx")
    .Output("count: out_idx")
    .Attr("T: type")
    .Attr("out_idx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      auto uniq = c->Vector(InferenceContext::kUnknownDim);
      c->set_output(0, uniq);
      c->set_output(1, c->input(0));
      c->set_output(2, uniq);
      return Status::OK();
    });

REGISTER_OP("UniqueWithCountsV2")
    .Input("x: T")
    .Input("axis: Taxis")
    .Output("y: T")
    .Output("idx: out_idx")
    .Output("count: out_idx")
    .Attr("T: type")
    .Attr("Taxis: {int32,int64} = DT_INT64")
    .Attr("out_idx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      c->set_output(0, c->UnknownShapeOfRank(c->Rank(c->input(0))));
      TF_RETURN_IF_ERROR(UniqueIdxShapeFn(c));
      c->set_output(2, c->Vector(InferenceContext::kUnknownDim));
      return Status::OK();
    });

namespace {

Status ShapeShapeFn(InferenceContext* c) {
  for (int i = 0; i < c->num_inputs(); ++i) {
    DimensionHandle dim;
    if (c->RankKnown(c->input(i))) {
      dim = c->MakeDim(c->Rank(c->input(i)));
    } else {
      dim = c->UnknownDim();
    }
    c->set_output(i, c->Vector(dim));
  }
  return Status::OK();
}

}  // namespace

// --------------------------------------------------------------------------
REGISTER_OP("Shape")
    .Input("input: T")
    .Output("output: out_type")
    .Attr("T: type")
    .Attr("out_type: {int32, int64} = DT_INT32")
    .SetShapeFn(ShapeShapeFn);

REGISTER_OP("ShapeN")
    .Input("input: N * T")
    .Output("output: N * out_type")
    .Attr("N: int")
    .Attr("T: type")
    .Attr("out_type: {int32, int64} = DT_INT32")
    .SetShapeFn(ShapeShapeFn);

REGISTER_OP("EnsureShape")
    .Input("input: T")
    .Output("output: T")
    .Attr("shape: shape")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      // Merges desired shape and statically known shape of input
      PartialTensorShape desired_shape;
      TF_RETURN_IF_ERROR(c->GetAttr("shape", &desired_shape));

      int rank = desired_shape.dims();
      ShapeHandle input_shape_handle;
      ShapeHandle desired_shape_handle;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &input_shape_handle));
      TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(
          desired_shape, &desired_shape_handle));

      ShapeHandle merged_shape;
      TF_RETURN_IF_ERROR(
          c->Merge(desired_shape_handle, input_shape_handle, &merged_shape));
      c->set_output(0, merged_shape);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("ReverseSequence")
    .Input("input: T")
    .Input("seq_lengths: Tlen")
    .Output("output: T")
    .Attr("seq_dim: int")
    .Attr("batch_dim: int = 0")
    .Attr("T: type")
    .Attr("Tlen: {int32, int64} = DT_INT64")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);
      ShapeHandle seq_lens_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &seq_lens_shape));

      int64_t seq_dim;
      TF_RETURN_IF_ERROR(c->GetAttr("seq_dim", &seq_dim));
      int64_t batch_dim;
      TF_RETURN_IF_ERROR(c->GetAttr("batch_dim", &batch_dim));

      if (!c->RankKnown(input)) {
        return shape_inference::UnknownShape(c);
      }

      // Validate batch_dim and seq_dim against input.
      const int32_t input_rank = c->Rank(input);
      if (batch_dim >= input_rank) {
        return errors::InvalidArgument(
            "batch_dim must be < input rank: ", batch_dim, " vs. ", input_rank);
      }
      if (seq_dim >= input_rank) {
        return errors::InvalidArgument(
            "seq_dim must be < input rank: ", seq_dim, " vs. ", input_rank);
      }

      DimensionHandle batch_dim_dim = c->Dim(input, batch_dim);
      TF_RETURN_IF_ERROR(
          c->Merge(batch_dim_dim, c->Dim(seq_lens_shape, 0), &batch_dim_dim));

      // Replace batch_dim of input with batch_size
      ShapeHandle output_shape;
      TF_RETURN_IF_ERROR(
          c->ReplaceDim(input, batch_dim, batch_dim_dim, &output_shape));
      c->set_output(0, output_shape);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Rank")
    .Input("input: T")
    .Output("output: int32")
    .Attr("T: type")
    .SetShapeFn(shape_inference::ScalarShape);

// --------------------------------------------------------------------------
REGISTER_OP("Size")
    .Input("input: T")
    .Output("output: out_type")
    .Attr("T: type")
    .Attr("out_type: {int32, int64} = DT_INT32")
    .SetShapeFn(shape_inference::ScalarShape);

// --------------------------------------------------------------------------
REGISTER_OP("Slice")
    .Input("input: T")
    .Input("begin: Index")
    .Input("size: Index")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Index: {int32,int64}")
    .SetShapeFn(shape_inference::SliceShape);

#ifdef INTEL_MKL
REGISTER_OP("_MklSlice")
    .Input("input: T")
    .Input("begin: Index")
    .Input("size: Index")
    .Input("mkl_input: uint8")
    .Input("mkl_begin: uint8")
    .Input("mkl_size: uint8")
    .Output("output: T")
    .Output("mkl_output: uint8")
    .Attr("T: type")
    .Attr("Index: {int32,int64}")
    .SetShapeFn(shape_inference::SliceShape);
#endif

REGISTER_OP("StridedSlice")
    .Input("input: T")
    .Input("begin: Index")
    .Input("end: Index")
    .Input("strides: Index")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Index: {int32, int64}")
    .Attr("begin_mask: int = 0")
    .Attr("end_mask: int = 0")
    .Attr("ellipsis_mask: int = 0")
    .Attr("new_axis_mask: int = 0")
    .Attr("shrink_axis_mask: int = 0")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);
      ShapeHandle begin_shape, end_shape, strides_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &begin_shape));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &end_shape));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 1, &strides_shape));
      TF_RETURN_IF_ERROR(c->Merge(begin_shape, end_shape, &begin_shape));
      TF_RETURN_IF_ERROR(c->Merge(begin_shape, strides_shape, &begin_shape));
      DimensionHandle sparse_dims_dim = c->Dim(begin_shape, 0);

      const Tensor* strides_value = c->input_tensor(3);
      // TODO(aselle,allenl): If we had a stride_mask it would be possible to do
      // more shape inference here (e.g. for x[3, ::T]).
      if (!c->RankKnown(input) || !c->ValueKnown(sparse_dims_dim) ||
          strides_value == nullptr) {
        c->set_output(0, c->UnknownShape());
        return Status::OK();
      }

      PartialTensorShape input_shape({});
      for (int i = 0; i < c->Rank(input); ++i) {
        auto dim = c->Dim(input, i);
        input_shape.AddDim(c->ValueKnown(dim) ? c->Value(dim) : -1);
      }

      int32_t begin_mask, end_mask, ellipsis_mask, new_axis_mask,
          shrink_axis_mask;
      TF_RETURN_IF_ERROR(c->GetAttr("begin_mask", &begin_mask));
      TF_RETURN_IF_ERROR(c->GetAttr("end_mask", &end_mask));
      TF_RETURN_IF_ERROR(c->GetAttr("ellipsis_mask", &ellipsis_mask));
      TF_RETURN_IF_ERROR(c->GetAttr("new_axis_mask", &new_axis_mask));
      TF_RETURN_IF_ERROR(c->GetAttr("shrink_axis_mask", &shrink_axis_mask));

      const Tensor* begin_value = c->input_tensor(1);
      const Tensor* end_value = c->input_tensor(2);

      PartialTensorShape processing_shape, final_shape;
      bool is_identity, is_simple_slice, slice_dim0;
      gtl::InlinedVector<int64, 4> begin, end, strides;
      TF_RETURN_IF_ERROR(ValidateStridedSliceOp(
          begin_value, end_value, *strides_value, input_shape, begin_mask,
          end_mask, ellipsis_mask, new_axis_mask, shrink_axis_mask,
          &processing_shape, &final_shape, &is_identity, &is_simple_slice,
          &slice_dim0, &begin, &end, &strides));

      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(final_shape, &out));
      c->set_output(0, out);

      auto* shape_and_type = c->input_handle_shapes_and_types(0);
      if (shape_and_type) {
        c->set_output_handle_shapes_and_types(0, *shape_and_type);
      }

      return Status::OK();
    });

REGISTER_OP("StridedSliceGrad")
    .Input("shape: Index")
    .Input("begin: Index")
    .Input("end: Index")
    .Input("strides: Index")
    .Input("dy: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Index: {int32, int64}")
    .Attr("begin_mask: int = 0")
    .Attr("end_mask: int = 0")
    .Attr("ellipsis_mask: int = 0")
    .Attr("new_axis_mask: int = 0")
    .Attr("shrink_axis_mask: int = 0")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &out));
      c->set_output(0, out);
      return Status::OK();
    });

REGISTER_OP("StridedSliceAssign")
    .Input("ref: Ref(T)")
    .Input("begin: Index")
    .Input("end: Index")
    .Input("strides: Index")
    .Input("value: T")
    .Output("output_ref: Ref(T)")
    .Attr("T: type")
    .Attr("Index: {int32, int64}")
    .Attr("begin_mask: int = 0")
    .Attr("end_mask: int = 0")
    .Attr("ellipsis_mask: int = 0")
    .Attr("new_axis_mask: int = 0")
    .Attr("shrink_axis_mask: int = 0")
    .SetShapeFn(shape_inference::UnchangedShape);
// TODO(aselle): Fix this documentation once StridedSliceAssign Supports
// broadcasting.
// --------------------------------------------------------------------------

REGISTER_OP("ResourceStridedSliceAssign")
    .Input("ref: resource")
    .Input("begin: Index")
    .Input("end: Index")
    .Input("strides: Index")
    .Input("value: T")
    .Attr("T: type")
    .Attr("Index: {int32, int64}")
    .Attr("begin_mask: int = 0")
    .Attr("end_mask: int = 0")
    .Attr("ellipsis_mask: int = 0")
    .Attr("new_axis_mask: int = 0")
    .Attr("shrink_axis_mask: int = 0")
    .SetShapeFn(shape_inference::NoOutputs);

REGISTER_OP("TensorStridedSliceUpdate")
    .Input("input: T")
    .Input("begin: Index")
    .Input("end: Index")
    .Input("strides: Index")
    .Input("value: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Index: {int32, int64}")
    .Attr("begin_mask: int = 0")
    .Attr("end_mask: int = 0")
    .Attr("ellipsis_mask: int = 0")
    .Attr("new_axis_mask: int = 0")
    .Attr("shrink_axis_mask: int = 0")
    .SetShapeFn(shape_inference::UnchangedShape);

REGISTER_OP("Tile")
    .Input("input: T")
    .Input("multiples: Tmultiples")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tmultiples: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);
      // NOTE(mrry): Represent `multiples` as a `TensorShape` because (i)
      // it is a vector of non-negative integers, and (ii) doing so allows
      // us to handle partially-known multiples.
      ShapeHandle multiples;
      TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(1, &multiples));
      if (c->RankKnown(input)) {
        TF_RETURN_IF_ERROR(c->WithRank(multiples, c->Rank(input), &multiples));
        ShapeHandle dummy;
        TF_RETURN_IF_ERROR(
            c->Merge(c->input(1), c->Vector(c->Rank(input)), &dummy));
      }

      if (!c->RankKnown(multiples)) {
        return shape_inference::UnknownShape(c);
      }

      int32_t rank = c->Rank(multiples);
      TF_RETURN_IF_ERROR(c->WithRank(input, rank, &input));
      std::vector<DimensionHandle> dims(rank);
      for (int i = 0; i < rank; ++i) {
        TF_RETURN_IF_ERROR(
            c->Multiply(c->Dim(input, i), c->Dim(multiples, i), &dims[i]));
      }
      c->set_output(0, c->MakeShape(dims));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("TileGrad")
    .Input("input: T")
    .Input("multiples: int32")
    .Output("output: T")
    .Attr("T: type")
    .Deprecated(3, "TileGrad has been replaced with reduce_sum")
    .SetShapeFn(tensorflow::shape_inference::UnknownShape);

// --------------------------------------------------------------------------
REGISTER_OP("Where")
    .Input("input: T")
    .Attr("T: {numbertype, bool} = DT_BOOL")
    .Output("index: int64")
    .SetShapeFn([](InferenceContext* c) {
      c->set_output(0, c->Matrix(c->UnknownDim(), c->Rank(c->input(0))));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("BroadcastArgs")
    .Input("s0: T")
    .Input("s1: T")
    .Output("r0: T")
    .Attr("T: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle unused;
      ShapeHandle shape_x = c->input(0);
      ShapeHandle shape_y = c->input(1);
      TF_RETURN_IF_ERROR(c->WithRank(shape_x, 1, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(shape_y, 1, &unused));

      if (!c->ValueKnown(c->Dim(shape_x, 0)) ||
          !c->ValueKnown(c->Dim(shape_y, 0))) {
        c->set_output(0, c->Vector(InferenceContext::kUnknownDim));
        return Status::OK();
      }

      int64_t x_dim = c->Value(c->Dim(shape_x, 0));
      int64_t y_dim = c->Value(c->Dim(shape_y, 0));

      // Broadcasted shape is going to be as large as the largest dimension.
      c->set_output(0, c->Vector(std::max(x_dim, y_dim)));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("BroadcastGradientArgs")
    .Input("s0: T")
    .Input("s1: T")
    .Output("r0: T")
    .Output("r1: T")
    .Attr("T: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      // TODO(mrry): Implement constant_value for BroadcastGradientArgs?
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &unused));
      c->set_output(0, c->Vector(InferenceContext::kUnknownDim));
      c->set_output(1, c->Vector(InferenceContext::kUnknownDim));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Pad")
    .Input("input: T")
    .Input("paddings: Tpaddings")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tpaddings: {int32, int64} = DT_INT32")
    .SetShapeFn(PadShapeFn);

// --------------------------------------------------------------------------
REGISTER_OP("PadV2")
    .Input("input: T")
    .Input("paddings: Tpaddings")
    .Input("constant_values: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tpaddings: {int32, int64} = DT_INT32")
    .SetShapeFn(PadShapeFn);

// --------------------------------------------------------------------------
REGISTER_OP("MirrorPad")
    .Input("input: T")
    .Input("paddings: Tpaddings")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tpaddings: {int32, int64} = DT_INT32")
    .Attr(GetMirrorPadModeAttrString())
    .SetShapeFn(PadShapeFn);

// --------------------------------------------------------------------------
namespace {
template <typename T>
Status MirrorPadKnown(InferenceContext* c, ShapeHandle input,
                      const Tensor* paddings_t, int64_t input_rank) {
  auto paddings_data = paddings_t->matrix<T>();
  std::vector<DimensionHandle> dims(input_rank);
  for (int64_t i = 0; i < input_rank; ++i) {
    const int64_t pad0 = static_cast<int64_t>(paddings_data(i, 0));
    const int64_t pad1 = static_cast<int64_t>(paddings_data(i, 1));
    if (pad0 < 0 || pad1 < 0) {
      return errors::InvalidArgument("Paddings must be non-negative");
    }

    TF_RETURN_IF_ERROR(c->Subtract(c->Dim(input, i), pad0 + pad1, &dims[i]));
  }
  c->set_output(0, c->MakeShape(dims));
  return Status::OK();
}

}  // namespace

REGISTER_OP("MirrorPadGrad")
    .Input("input: T")
    .Input("paddings: Tpaddings")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tpaddings: {int32, int64} = DT_INT32")
    .Attr(GetMirrorPadModeAttrString())
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle paddings;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 2, &paddings));
      DimensionHandle pad_0 = c->Dim(paddings, 0);
      if (!c->ValueKnown(pad_0)) {
        // We don't know the rank of the output since the first
        // padding dimension is unknown.
        c->set_output(0, c->UnknownShape());
        return Status::OK();
      }

      int64_t input_rank = c->Value(pad_0);
      ShapeHandle input;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), input_rank, &input));
      TF_RETURN_IF_ERROR(
          c->Merge(paddings, c->Matrix(input_rank, 2), &paddings));

      const Tensor* paddings_t = c->input_tensor(1);
      if (paddings_t == nullptr) {
        // Values of 'paddings' is not available, but we know the
        // input rank, so return the rank of the output with unknown
        // dimensions.
        c->set_output(0, c->UnknownShapeOfRank(input_rank));
        return Status::OK();
      }

      if (paddings_t->dtype() == DT_INT32) {
        return MirrorPadKnown<int32>(c, input, paddings_t, input_rank);
      } else {
        return MirrorPadKnown<int64_t>(c, input, paddings_t, input_rank);
      }
    });

// --------------------------------------------------------------------------
REGISTER_OP("Placeholder")
    .Output("output: dtype")
    .Attr("dtype: type")
    .Attr("shape: shape = { unknown_rank: true }")
    .SetShapeFn([](InferenceContext* c) {
      PartialTensorShape shape;
      TF_RETURN_IF_ERROR(c->GetAttr("shape", &shape));

      // Placeholder has legacy behavior where we cannot tell the difference
      // between a scalar shape attribute and 'unknown shape'.  So if the shape
      // is a scalar, we return an unknown shape.
      if (c->graph_def_version() <= 21 && shape.dims() <= 0) {
        return shape_inference::UnknownShape(c);
      }

      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(shape, &out));
      c->set_output(0, out);
      return Status::OK();
    });

// Placeholder was modified in a backwards compatible way to do what
// PlaceholderV2 did, so we have deprecated V2 (no one was really
// using it).
REGISTER_OP("PlaceholderV2")
    .Output("output: dtype")
    .Attr("dtype: type")
    .Attr("shape: shape")
    .SetShapeFn(shape_inference::ExplicitShape)
    .Deprecated(23, "Placeholder now behaves the same as PlaceholderV2.");

// --------------------------------------------------------------------------
REGISTER_OP("PlaceholderWithDefault")
    .Input("input: dtype")
    .Output("output: dtype")
    .Attr("dtype: type")
    .Attr("shape: shape")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);
      PartialTensorShape shape;
      TF_RETURN_IF_ERROR(c->GetAttr("shape", &shape));
      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(shape, &out));

      // We merge for compatibility checking, but return the output,
      // since output_shape may be less precise than input_shape.
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->Merge(input, out, &unused));
      c->set_output(0, out);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("ExpandDims")
    .Input("input: T")
    .Input("dim: Tdim")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tdim: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);

      const Tensor* dim_t = c->input_tensor(1);
      if (dim_t != nullptr && dim_t->NumElements() != 1) {
        return errors::InvalidArgument(
            "'dim' input must be a tensor with a single value");
      }
      if (dim_t == nullptr || !c->RankKnown(input)) {
        c->set_output(0, c->UnknownShape());
        return Status::OK();
      }

      int64_t dim;
      if (dim_t->dtype() == DT_INT32) {
        dim = static_cast<int64_t>(dim_t->flat<int32>()(0));
      } else {
        dim = dim_t->flat<int64_t>()(0);
      }

      const int32_t rank = c->Rank(input);
      const int32_t min_dim = -1 * rank - 1;
      if (dim < min_dim || dim > rank) {
        return errors::InvalidArgument("dim ", dim, " not in the interval [",
                                       min_dim, ", ", rank, "].");
      }

      if (dim < 0) {
        dim += rank + 1;
      }

      ShapeHandle end;
      TF_RETURN_IF_ERROR(c->Subshape(input, dim, &end));

      // Build output as start + 1 + end.
      ShapeHandle output;
      TF_RETURN_IF_ERROR(c->Subshape(input, 0, dim, &output));
      TF_RETURN_IF_ERROR(c->Concatenate(output, c->Vector(1), &output));
      TF_RETURN_IF_ERROR(c->Concatenate(output, end, &output));
      c->set_output(0, output);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("Squeeze")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("squeeze_dims: list(int) >= 0 = []")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input = c->input(0);
      if (!c->RankKnown(input)) {
        // Input shape unknown.
        return shape_inference::UnknownShape(c);
      }

      const int32_t input_rank = c->Rank(input);

      // Validate and wrap squeeze dimensions.
      std::vector<int32> squeeze_dims;
      TF_RETURN_IF_ERROR(c->GetAttr("squeeze_dims", &squeeze_dims));
      for (int i = 0; i < squeeze_dims.size(); ++i) {
        if (squeeze_dims[i] < -input_rank || squeeze_dims[i] >= input_rank) {
          return errors::InvalidArgument("squeeze_dims[", i, "] not in [",
                                         -input_rank, ",", input_rank, ").");
        }

        if (squeeze_dims[i] < 0) {
          squeeze_dims[i] += input_rank;
        }
      }

      std::vector<DimensionHandle> result_shape;
      for (int i = 0; i < input_rank; ++i) {
        // True if squeeze_dims contains an entry to squeeze this
        // dimension.
        bool is_explicit_match =
            std::find(squeeze_dims.begin(), squeeze_dims.end(), i) !=
            squeeze_dims.end();

        DimensionHandle dim = c->Dim(input, i);

        if (!c->ValueKnown(dim)) {
          // Assume that the squeezed dimension will be 1 at runtime.
          if (is_explicit_match) continue;

          // If squeezing all 1 dimensions, and we see an unknown value,
          // give up and return Unknown Shape.
          if (squeeze_dims.empty()) {
            c->set_output(0, c->UnknownShape());
            return Status::OK();
          }
        } else if (c->Value(dim) == 1) {
          if (is_explicit_match || squeeze_dims.empty()) {
            // If explicitly squeezing, or squeezing all 1s, remove
            // this dimension.
            continue;
          }
        } else if (is_explicit_match) {
          return errors::InvalidArgument("Can not squeeze dim[", i,
                                         "], expected a dimension of 1, got ",
                                         c->Value(c->Dim(input, i)));
        }

        result_shape.emplace_back(dim);
      }

      c->set_output(0, c->MakeShape(result_shape));
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("ListDiff")
    .Input("x: T")
    .Input("y: T")
    .Output("out: T")
    .Output("idx: out_idx")
    .Attr("T: type")
    .Attr("out_idx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &unused));
      // TODO(mrry): Indicate that the length falls within an interval?
      ShapeHandle out = c->Vector(InferenceContext::kUnknownDim);
      c->set_output(0, out);
      c->set_output(1, out);
      return Status::OK();
    });

namespace {

```

```
// Converts Tensor to flat std::vector<int64_t>.
template <typename InputType>
std::vector<int64_t> GetFlatInt64(const Tensor& t) {
  std::vector<int64_t> output(t.shape().num_elements());
  if (t.shape().num_elements() > 0) {
    auto eigen_vec = t.flat<InputType>();
    std::copy_n(&eigen_vec(0), output.size(), output.begin());
  }
  return output;
}

// Converts int32 or int64 Tensor to flat std::vector<int64_t>.
std::vector<int64_t> GetFlatInt64(const Tensor& t) {
  if (t.dtype() == DT_INT32) {
    return GetFlatInt64<int32>(t);
  } else {
    return GetFlatInt64<int64_t>(t);
  }
}

Status SpaceToBatchShapeHelper(InferenceContext* c, ShapeHandle input_shape,
                               ShapeHandle block_shape_shape,
                               const Tensor* block_shape_t,
                               ShapeHandle paddings_shape,
                               const Tensor* paddings_t) {
  if (c->Rank(block_shape_shape) != 1) {
    return errors::InvalidArgument("block_shape must have rank 1.");
  }

  const DimensionHandle num_block_dims_handle = c->Dim(block_shape_shape, 0);
  if (!c->ValueKnown(num_block_dims_handle)) {
    return errors::InvalidArgument("block_shape must have known size.");
  }

  const int64_t num_block_dims = c->Value(num_block_dims_handle);

  TF_RETURN_IF_ERROR(
      c->WithRankAtLeast(input_shape, num_block_dims + 1, &input_shape));

  TF_RETURN_IF_ERROR(
      c->Merge(paddings_shape, c->Matrix(num_block_dims, 2), &paddings_shape));

  DimensionHandle batch_size = c->Dim(input_shape, 0);
  std::vector<int64_t> block_shape_vec;
  if (block_shape_t && (block_shape_t->NumElements() > 0)) {
    block_shape_vec = GetFlatInt64(*block_shape_t);
    for (int64_t dim = 0; dim < num_block_dims; ++dim) {
      const int64_t block_shape_value = block_shape_vec[dim];
      if (block_shape_value < 1) {
        return errors::InvalidArgument("block_shape must be positive");
      }
      if (c->ValueKnown(batch_size)) {
        TF_RETURN_IF_ERROR(
            c->Multiply(batch_size, block_shape_value, &batch_size));
      } else {
        batch_size = c->UnknownDim();
      }
    }
  } else if (num_block_dims > 0) {
    batch_size = c->UnknownDim();
  }

  std::vector<DimensionHandle> output_dims{batch_size};
  output_dims.resize(num_block_dims + 1, c->UnknownDim());

  if (paddings_t && (paddings_t->NumElements() > 0)) {
    const std::vector<int64_t> paddings_vec = GetFlatInt64(*paddings_t);
    for (int64_t dim = 0; dim < num_block_dims; ++dim) {
      const int64_t pad_start = paddings_vec[dim * 2],
                    pad_end = paddings_vec[dim * 2 + 1];
      if (pad_start < 0 || pad_end < 0) {
        return errors::InvalidArgument("paddings cannot be negative");
      }
      if (block_shape_t) {
        DimensionHandle padded_size;
        TF_RETURN_IF_ERROR(
            c->Add(c->Dim(input_shape, dim + 1), pad_start, &padded_size));
        TF_RETURN_IF_ERROR(c->Add(padded_size, pad_end, &padded_size));
        TF_RETURN_IF_ERROR(c->Divide(padded_size, block_shape_vec[dim],
                                     /*evenly_divisible=*/true,
                                     &output_dims[dim + 1]));
      }
    }
  }

  ShapeHandle remaining_input_shape;
  TF_RETURN_IF_ERROR(
      c->Subshape(input_shape, 1 + num_block_dims, &remaining_input_shape));

  ShapeHandle result;
  TF_RETURN_IF_ERROR(c->Concatenate(c->MakeShape(output_dims),
                                    remaining_input_shape, &result));
  c->set_output(0, result);
  return Status::OK();
}

Status BatchToSpaceShapeHelper(InferenceContext* c, ShapeHandle input_shape,
                               ShapeHandle block_shape_shape,
                               const Tensor* block_shape_t,
                               ShapeHandle crops_shape, const Tensor* crops_t) {
  if (c->Rank(block_shape_shape) != 1) {
    return errors::InvalidArgument("block_shape must have rank 1.");
  }

  const DimensionHandle num_block_dims_handle = c->Dim(block_shape_shape, 0);
  if (!c->ValueKnown(num_block_dims_handle)) {
    return errors::InvalidArgument("block_shape must have known size.");
  }

  const int64_t num_block_dims = c->Value(num_block_dims_handle);

  TF_RETURN_IF_ERROR(
      c->WithRankAtLeast(input_shape, num_block_dims + 1, &input_shape));

  TF_RETURN_IF_ERROR(
      c->Merge(crops_shape, c->Matrix(num_block_dims, 2), &crops_shape));

  DimensionHandle batch_size = c->Dim(input_shape, 0);
  std::vector<int64_t> block_shape_vec;
  if (block_shape_t) {
    block_shape_vec = GetFlatInt64(*block_shape_t);
    for (int64_t dim = 0; dim < num_block_dims; ++dim) {
      const int64_t block_shape_value = block_shape_vec[dim];
      if (block_shape_value < 1) {
        return errors::InvalidArgument("block_shape must be positive");
      }
      if (c->ValueKnown(batch_size)) {
        TF_RETURN_IF_ERROR(c->Divide(batch_size, block_shape_value,
                                     /*evenly_divisible=*/true, &batch_size));
      } else {
        batch_size = c->UnknownDim();
      }
    }
  } else if (num_block_dims > 0) {
    batch_size = c->UnknownDim();
  }

  std::vector<DimensionHandle> output_dims{batch_size};
  output_dims.resize(num_block_dims + 1, c->UnknownDim());

  if (crops_t) {
    const std::vector<int64_t> crops_vec = GetFlatInt64(*crops_t);
    for (int64_t dim = 0; dim < num_block_dims; ++dim) {
      const int64_t crop_start = crops_vec[dim * 2],
                    crop_end = crops_vec[dim * 2 + 1];
      if (crop_start < 0 || crop_end < 0) {
        return errors::InvalidArgument("crops cannot be negative");
      }
      if (block_shape_t) {
        DimensionHandle cropped_size;
        TF_RETURN_IF_ERROR(c->Multiply(c->Dim(input_shape, dim + 1),
                                       block_shape_vec[dim], &cropped_size));
        TF_RETURN_IF_ERROR(
            c->Subtract(cropped_size, crop_start, &cropped_size));
        TF_RETURN_IF_ERROR(
            c->Subtract(cropped_size, crop_end, &output_dims[dim + 1]));
      }
    }
  }

  ShapeHandle remaining_input_shape;
  TF_RETURN_IF_ERROR(
      c->Subshape(input_shape, 1 + num_block_dims, &remaining_input_shape));

  ShapeHandle result;
  TF_RETURN_IF_ERROR(c->Concatenate(c->MakeShape(output_dims),
                                    remaining_input_shape, &result));
  c->set_output(0, result);
  return Status::OK();
}

}  // namespace

// --------------------------------------------------------------------------
REGISTER_OP("SpaceToBatchND")
    .Input("input: T")
    .Input("block_shape: Tblock_shape")
    .Input("paddings: Tpaddings")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tblock_shape: {int32, int64} = DT_INT32")
    .Attr("Tpaddings: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      return SpaceToBatchShapeHelper(c, c->input(0), c->input(1),
                                     c->input_tensor(1), c->input(2),
                                     c->input_tensor(2));
    });

// --------------------------------------------------------------------------
REGISTER_OP("SpaceToBatch")
    .Input("input: T")
    .Input("paddings: Tpaddings")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tpaddings: {int32, int64} = DT_INT32")
    .Attr("block_size: int >= 2")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));

      int32_t block_size;
      TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));

      Tensor block_shape(tensorflow::DT_INT64, TensorShape({2}));
      auto block_shape_vec = block_shape.vec<int64_t>();
      block_shape_vec(0) = block_size;
      block_shape_vec(1) = block_size;

      return SpaceToBatchShapeHelper(c, input_shape, c->MakeShape({2}),
                                     &block_shape, c->input(1),
                                     c->input_tensor(1));
    });

// --------------------------------------------------------------------------
REGISTER_OP("BatchToSpaceND")
    .Input("input: T")
    .Input("block_shape: Tblock_shape")
    .Input("crops: Tcrops")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tblock_shape: {int32, int64} = DT_INT32")
    .Attr("Tcrops: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      return BatchToSpaceShapeHelper(c, c->input(0), c->input(1),
                                     c->input_tensor(1), c->input(2),
                                     c->input_tensor(2));
    });

// --------------------------------------------------------------------------
REGISTER_OP("BatchToSpace")
    .Input("input: T")
    .Input("crops: Tidx")
    .Output("output: T")
    .Attr("T: type")
    .Attr("block_size: int >= 2")
    .Attr("Tidx: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));

      int32_t block_size;
      TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));

      Tensor block_shape(tensorflow::DT_INT64, TensorShape({2}));
      auto block_shape_vec = block_shape.vec<int64_t>();
      block_shape_vec(0) = block_size;
      block_shape_vec(1) = block_size;

      return BatchToSpaceShapeHelper(c, input_shape, c->MakeShape({2}),
                                     &block_shape, c->input(1),
                                     c->input_tensor(1));
    });

// --------------------------------------------------------------------------
REGISTER_OP("SpaceToDepth")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("block_size: int >= 2")
    .Attr("data_format: {'NHWC', 'NCHW', 'NCHW_VECT_C'} = 'NHWC'")
    // TODO(pauldonnelly): Implement GPU kernels for NCHW_VECT_C.
    .SetShapeFn([](InferenceContext* c) {
      string data_format_str;
      TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
      TensorFormat data_format;
      FormatFromString(data_format_str, &data_format);

      constexpr int num_spatial_dims = 2;
      const int dims =
          GetTensorDimsFromSpatialDims(num_spatial_dims, data_format);
      ShapeHandle input;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), dims, &input));

      int32_t block_size;
      TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));

      DimensionHandle batch_size =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'N'));
      DimensionHandle input_height =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'H'));
      DimensionHandle input_width =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'W'));
      DimensionHandle input_depth =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'C'));

      DimensionHandle output_height;
      DimensionHandle output_width;
      DimensionHandle output_depth;
      // Will return an error if input height or width are not evenly divisible.
      TF_RETURN_IF_ERROR(c->Divide(input_height, block_size,
                                   true /* evenly_divisible */,
                                   &output_height));
      TF_RETURN_IF_ERROR(c->Divide(input_width, block_size,
                                   true /* evenly_divisible */, &output_width));

      TF_RETURN_IF_ERROR(
          c->Multiply(input_depth, block_size * block_size, &output_depth));

      ShapeHandle output_shape;
      TF_RETURN_IF_ERROR(MakeShapeFromFormat(data_format, batch_size,
                                             {output_height, output_width},
                                             output_depth, &output_shape, c));

      c->set_output(0, output_shape);
      return Status::OK();
    });

// --------------------------------------------------------------------------
REGISTER_OP("DepthToSpace")
    .Input("input: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("block_size: int >= 2")
    .Attr("data_format: {'NHWC', 'NCHW', 'NCHW_VECT_C'} = 'NHWC'")
    // TODO(pauldonnelly): Implement GPU kernels for NCHW and NCHW_VECT_C.
    .SetShapeFn([](InferenceContext* c) {
      string data_format_str;
      TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
      TensorFormat data_format;
      FormatFromString(data_format_str, &data_format);

      constexpr int num_spatial_dims = 2;
      const int dims =
          GetTensorDimsFromSpatialDims(num_spatial_dims, data_format);

      ShapeHandle input;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), dims, &input));

      int32_t block_size;
      TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));

      DimensionHandle batch_size =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'N'));
      DimensionHandle input_height =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'H'));
      DimensionHandle input_width =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'W'));
      DimensionHandle input_depth =
          c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'C'));

      DimensionHandle output_height;
      DimensionHandle output_width;
      DimensionHandle output_depth;
      TF_RETURN_IF_ERROR(c->Multiply(input_height, block_size, &output_height));
      TF_RETURN_IF_ERROR(c->Multiply(input_width, block_size, &output_width));

      // Will return an error if input_depth is not evenly divisible.
      TF_RETURN_IF_ERROR(c->Divide(input_depth, block_size * block_size,
                                   true /* evenly_divisible */, &output_depth));

      ShapeHandle output_shape;
      TF_RETURN_IF_ERROR(MakeShapeFromFormat(data_format, batch_size,
                                             {output_height, output_width},
                                             output_depth, &output_shape, c));

      c->set_output(0, output_shape);
      return Status::OK();
    });

// -----------------------------------------------------------------------------

REGISTER_OP("ExtractImagePatches")
    .Input("images: T")
    .Output("patches: T")
    .Attr("ksizes: list(int) >= 4")
    .Attr("strides: list(int) >= 4")
    .Attr("rates: list(int) >= 4")
    .Attr(
        "T: {bfloat16, half, float, double, int8, int16, int32, int64, "
        "uint8, uint16, uint32, uint64, complex64, complex128, bool}")
    .Attr(GetPaddingAttrString())
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));

      std::vector<int32> ksizes;
      TF_RETURN_IF_ERROR(c->GetAttr("ksizes", &ksizes));
      if (ksizes.size() != 4) {
        return errors::InvalidArgument(
            "ExtractImagePatches requires the ksizes attribute to contain 4 "
            "values, but got: ",
            ksizes.size());
      }

      std::vector<int32> strides;
      TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
      if (strides.size() != 4) {
        return errors::InvalidArgument(
            "ExtractImagePatches requires the stride attribute to contain 4 "
            "values, but got: ",
            strides.size());
      }
```

```cpp

    std::vector<int32> rates;
    TF_RETURN_IF_ERROR(c->GetAttr("rates", &rates));
    if (rates.size() != 4) {
      return errors::InvalidArgument(
          "ExtractImagePatches requires the rates attribute to contain 4 "
          "values, but got: ",
          rates.size());
    }

    int32_t ksize_rows = ksizes[1];
    int32_t ksize_cols = ksizes[2];

    int32_t stride_rows = strides[1];
    int32_t stride_cols = strides[2];

    int32_t rate_rows = rates[1];
    int32_t rate_cols = rates[2];

    int32_t ksize_rows_eff = ksize_rows + (ksize_rows - 1) * (rate_rows - 1);
    int32_t ksize_cols_eff = ksize_cols + (ksize_cols - 1) * (rate_cols - 1);

    DimensionHandle batch_size_dim = c->Dim(input_shape, 0);
    DimensionHandle in_rows_dim = c->Dim(input_shape, 1);
    DimensionHandle in_cols_dim = c->Dim(input_shape, 2);
    DimensionHandle output_depth_dim;
    TF_RETURN_IF_ERROR(c->Multiply(
        c->Dim(input_shape, 3), ksize_rows * ksize_cols, &output_depth_dim));

    if (!c->ValueKnown(in_rows_dim) || !c->ValueKnown(in_cols_dim)) {
      ShapeHandle output_shape =
          c->MakeShape({batch_size_dim, InferenceContext::kUnknownDim,
                        InferenceContext::kUnknownDim, output_depth_dim});
      c->set_output(0, output_shape);
      return Status::OK();
    }
    auto in_rows = c->Value(in_rows_dim);
    auto in_cols = c->Value(in_cols_dim);

    Padding padding;
    TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));

    int64_t output_rows, output_cols;
    int64_t padding_before, padding_after;
    TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
        in_rows, ksize_rows_eff, stride_rows, padding, &output_rows,
        &padding_before, &padding_after));
    TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
        in_cols, ksize_cols_eff, stride_cols, padding, &output_cols,
          &padding_before, &padding_after));
      ShapeHandle output_shape = c->MakeShape(
          {batch_size_dim, output_rows, output_cols, output_depth_dim});
      c->set_output(0, output_shape);
      return Status::OK();
    });

// ----------------------------------------------------------------------------

// To enable rates, uncomment all lines commented below and use ksize_*_eff
// as the second parameter of all GetWindowedOutputSizeVerbose calls instead
// of ksize_*.
REGISTER_OP("ExtractVolumePatches")
    .Input("input: T")
    .Output("patches: T")
    .Attr("ksizes: list(int) >= 5")
    .Attr("strides: list(int) >= 5")
    /* .Attr("rates: list(int) >= 5") */
    .Attr("T: realnumbertype")
    .Attr(GetPaddingAttrString())
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 5, &input_shape));

      std::vector<int32> ksizes;
      TF_RETURN_IF_ERROR(c->GetAttr("ksizes", &ksizes));
      if (ksizes.size() != 5) {
        return errors::InvalidArgument(
            "ExtractVolumePatches requires the ksizes attribute to contain 5 "
            "values, but got: ",
            ksizes.size());
      }

      std::vector<int32> strides;
      TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
      if (strides.size() != 5) {
        return errors::InvalidArgument(
            "ExtractVolumePatches requires the stride attribute to contain 5 "
            "values, but got: ",
            strides.size());
      }

      /*
      // TODO(hsgkim): Enable rates.
      // See extract_volume_patches_op.cc for why rates are disabled now.

      std::vector<int32> rates;
      TF_RETURN_IF_ERROR(c->GetAttr("rates", &rates));
      if (rates.size() != 5) {
```

```
      return errors::InvalidArgument(
          "ExtractVolumePatches requires the rates attribute to contain 5 "
          "values, but got: ",
          rates.size());
    }
    */

    int32_t ksize_planes = ksizes[1];
    int32_t ksize_rows = ksizes[2];
    int32_t ksize_cols = ksizes[3];

    int32_t stride_planes = strides[1];
    int32_t stride_rows = strides[2];
    int32_t stride_cols = strides[3];

    /*
    int32 rate_planes = rates[1];
    int32 rate_rows = rates[2];
    int32 rate_cols = rates[3];

    int32 ksize_planes_eff = ksize_planes +
                             (ksize_planes - 1) * (rate_planes - 1);
    int32 ksize_rows_eff = ksize_rows + (ksize_rows - 1) * (rate_rows - 1);
    int32 ksize_cols_eff = ksize_cols + (ksize_cols - 1) * (rate_cols - 1);
    */

    DimensionHandle batch_size_dim = c->Dim(input_shape, 0);
    DimensionHandle in_planes_dim = c->Dim(input_shape, 1);
    DimensionHandle in_rows_dim = c->Dim(input_shape, 2);
    DimensionHandle in_cols_dim = c->Dim(input_shape, 3);
    DimensionHandle output_depth_dim;
    TF_RETURN_IF_ERROR(c->Multiply(c->Dim(input_shape, 4),
                                   ksize_planes * ksize_rows * ksize_cols,
                                   &output_depth_dim));

    if (!c->ValueKnown(in_planes_dim) || !c->ValueKnown(in_rows_dim) ||
        !c->ValueKnown(in_cols_dim)) {
      ShapeHandle output_shape =
          c->MakeShape({batch_size_dim, InferenceContext::kUnknownDim,
                        InferenceContext::kUnknownDim, output_depth_dim});
      c->set_output(0, output_shape);
      return Status::OK();
    }
    auto in_planes = c->Value(in_planes_dim);
    auto in_rows = c->Value(in_rows_dim);
    auto in_cols = c->Value(in_cols_dim);

    Padding padding;
    TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
```

```

      int64_t output_planes, output_rows, output_cols;
      int64_t padding_before, padding_after;
      TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
          in_planes, ksize_planes, stride_planes, padding, &output_planes,
          &padding_before, &padding_after));
      TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
          in_rows, ksize_rows, stride_rows, padding, &output_rows,
          &padding_before, &padding_after));
      TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
          in_cols, ksize_cols, stride_cols, padding, &output_cols,
          &padding_before, &padding_after));
      ShapeHandle output_shape =
          c->MakeShape({batch_size_dim, output_planes, output_rows, output_cols,
                        output_depth_dim});
      c->set_output(0, output_shape);
      return Status::OK();
    });

// --------------------------------------------------------------------------

REGISTER_OP("OneHot")
    .Input("indices: TI")
    .Input("depth: int32")
    .Input("on_value: T")
    .Input("off_value: T")
    .Attr("axis: int = -1")
    .Output("output: T")
    .Attr("T: type")
    .Attr("TI: {uint8, int32, int64} = DT_INT64")
    .SetShapeFn([](InferenceContext* c) {
      int32_t axis;
      TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
      if (axis < -1) return errors::InvalidArgument("axis must be >= -1");

      DimensionHandle depth;
      TF_RETURN_IF_ERROR(c->MakeDimForScalarInput(1, &depth));

      ShapeHandle indices = c->input(0);
      if (!c->RankKnown(indices)) return shape_inference::UnknownShape(c);

      int32_t new_rank = c->Rank(indices) + 1;
      // We need to add new_rank to axis in the case the axis is -1 because
      // C++ returns negative values from % if the dividend is negative.
      int32_t depth_index = (axis + new_rank) % new_rank;
      // Out shape is indices[0:depth_index] + [depth] + indices[depth_index:].
      ShapeHandle front;
      ShapeHandle back;
      ShapeHandle out;
      TF_RETURN_IF_ERROR(c->Subshape(indices, 0, depth_index, &front));
      TF_RETURN_IF_ERROR(c->Subshape(indices, depth_index, &back));
      TF_RETURN_IF_ERROR(c->Concatenate(front, c->Vector(depth), &front));
      TF_RETURN_IF_ERROR(c->Concatenate(front, back, &out));
      c->set_output(0, out);
      return Status::OK();
    });

// EXPERIMENTAL. DO NOT USE OR DEPEND ON THIS YET.
REGISTER_OP("QuantizeAndDequantize")
    .Input("input: T")
    .Attr("signed_input: bool = true")
    .Attr("num_bits: int = 8")
    .Attr("range_given: bool = false")
    .Attr("input_min: float = 0")
    .Attr("input_max: float = 0")
    .Output("output: T")
    .Attr("T: {bfloat16, half, float, double}")
    .SetShapeFn(shape_inference::UnchangedShape)
    .Deprecated(22, "Replaced by QuantizeAndDequantizeV2");

// TODO(suharshs): Deprecate QuantizeAndDequantizeV2.
REGISTER_OP("QuantizeAndDequantizeV2")
    .Input("input: T")
    .Input("input_min: T")
    .Input("input_max: T")
    .Attr("signed_input: bool = true")
    .Attr("num_bits: int = 8")
    .Attr("range_given: bool = false")
    .Output("output: T")
    .Attr("T: {bfloat16, half, float, double}")
    .Attr(
        "round_mode: {'HALF_TO_EVEN', 'HALF_UP'} = "
        "'HALF_TO_EVEN'")
    .Attr("narrow_range: bool = false")
    .Attr("axis: int = -1")
    .SetShapeFn([](InferenceContext* c) {
      int axis;
      TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
      const int minmax_rank = (axis == -1) ? 0 : 1;
      ShapeHandle minmax;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
      TF_RETURN_IF_ERROR(c->Merge(c->input(2), minmax, &minmax));
      if (axis < -1) {
        return errors::InvalidArgument("axis should be at least -1, got ",
                                       axis);
      } else if (axis != -1) {
        ShapeHandle input;
        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
        DimensionHandle depth;
        TF_RETURN_IF_ERROR(
            c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
      }
      c->set_output(0, c->input(0));
      return Status::OK();
    });

REGISTER_OP("QuantizeAndDequantizeV4")
    .Input("input: T")
    .Input("input_min: T")
    .Input("input_max: T")
    .Attr("signed_input: bool = true")
    .Attr("num_bits: int = 8")
    .Attr("range_given: bool = false")
    .Output("output: T")
    .Attr("T: {bfloat16, half, float, double}")
    .Attr(
        "round_mode: {'HALF_TO_EVEN', 'HALF_UP'} = "
        "'HALF_TO_EVEN'")
    .Attr("narrow_range: bool = false")
    .Attr("axis: int = -1")
    .SetShapeFn([](InferenceContext* c) {
      int axis;
      TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
      const int minmax_rank = (axis == -1) ? 0 : 1;
      ShapeHandle minmax;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
      TF_RETURN_IF_ERROR(c->Merge(c->input(2), minmax, &minmax));
      if (axis < -1) {
        return errors::InvalidArgument("axis should be at least -1, got ",
                                       axis);
      } else if (axis != -1) {
        ShapeHandle input;
        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
        DimensionHandle depth;
        TF_RETURN_IF_ERROR(
            c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
      }
      c->set_output(0, c->input(0));
      return Status::OK();
    });

REGISTER_OP("QuantizeAndDequantizeV4Grad")
    .Input("gradients: T")
    .Input("input: T")
    .Input("input_min: T")
    .Input("input_max: T")
    .Output("input_backprop: T")
    .Output("input_min_backprop: T")
    .Output("input_max_backprop: T")
    .Attr("T: {bfloat16, half, float, double}")
    .Attr("axis: int = -1")
    .SetShapeFn([](InferenceContext* c) {
      int axis;
      TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
      const int minmax_rank = (axis == -1) ? 0 : 1;
      ShapeHandle minmax;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), minmax_rank, &minmax));
      TF_RETURN_IF_ERROR(c->Merge(c->input(3), minmax, &minmax));
      if (axis < -1) {
        return errors::InvalidArgument("axis should be at least -1, got ",
                                       axis);
      } else if (axis != -1) {
        ShapeHandle input;
        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
        DimensionHandle depth;
        TF_RETURN_IF_ERROR(
            c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
      }
      ShapeHandle inputs;
      TF_RETURN_IF_ERROR(c->Merge(c->input(0), c->input(1), &inputs));
      c->set_output(0, inputs);
      c->set_output(1, minmax);
      c->set_output(2, minmax);
      return Status::OK();
    });

REGISTER_OP("QuantizeAndDequantizeV3")
    .Input("input: T")
    .Input("input_min: T")
    .Input("input_max: T")
    .Input("num_bits: int32")
    .Attr("signed_input: bool = true")
    .Attr("range_given: bool = true")
    .Output("output: T")
    .Attr("T: {bfloat16, half, float, double}")
    .Attr("narrow_range: bool = false")
    .Attr("axis: int = -1")
    .SetShapeFn([](InferenceContext* c) {
      int axis;
      TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
      const int minmax_rank = (axis == -1) ? 0 : 1;
      ShapeHandle minmax;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
      TF_RETURN_IF_ERROR(c->Merge(c->input(2), minmax, &minmax));
      if (axis < -1) {
        return errors::InvalidArgument("axis should be at least -1, got ",
                                       axis);
      } else if (axis != -1) {
        ShapeHandle input;
        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
        DimensionHandle depth;
        TF_RETURN_IF_ERROR(
            c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
      }
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 0, &unused));
      c->set_output(0, c->input(0));
      return Status::OK();
    });

REGISTER_OP("QuantizeV2")
    .Input("input: float")
    .Input("min_range: float")
    .Input("max_range: float")
    .Output("output: T")
    .Output("output_min: float")
    .Output("output_max: float")
    .Attr("T: quantizedtype")
    .Attr("mode: {'MIN_COMBINED', 'MIN_FIRST', 'SCALED'} = 'MIN_COMBINED'")
    .Attr(
        "round_mode: {'HALF_AWAY_FROM_ZERO', 'HALF_TO_EVEN'} = "
        "'HALF_AWAY_FROM_ZERO'")
    .Attr("narrow_range: bool = false")
    .Attr("axis: int = -1")
    .Attr("ensure_minimum_range: float = 0.01")
    .SetShapeFn(shape_inference::QuantizeV2Shape);

REGISTER_OP("Dequantize")
    .Input("input: T")
    .Input("min_range: float")
    .Input("max_range: float")
    .Output("output: dtype")
    .Attr("T: quantizedtype")
    .Attr("mode: {'MIN_COMBINED', 'MIN_FIRST', 'SCALED'} = 'MIN_COMBINED'")
    .Attr("narrow_range: bool = false")
    .Attr("axis: int = -1")
    .Attr("dtype: {bfloat16, float} = DT_FLOAT")
    .SetShapeFn([](InferenceContext* c) {
      int axis = -1;
      Status s = c->GetAttr("axis", &axis);
      if (!s.ok() && s.code() != error::NOT_FOUND) {
        return s;
      }
      if (axis < -1) {
        return errors::InvalidArgument("axis should be at least -1, got ",
                                       axis);
      }
      const int minmax_rank = (axis == -1) ? 0 : 1;
      TF_RETURN_IF_ERROR(shape_inference::UnchangedShape(c));
      ShapeHandle minmax;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), minmax_rank, &minmax));
      if (axis != -1) {
        ShapeHandle input;
        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
        DimensionHandle depth;
        TF_RETURN_IF_ERROR(
            c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
      }
      return Status::OK();
    });

REGISTER_OP("QuantizedConcat")
    .Input("concat_dim: int32")
    .Input("values: N * T")
    .Input("input_mins: N * float32")
    .Input("input_maxes: N * float32")
    .Output("output: T")
    .Output("output_min: float")
    .Output("output_max: float")
    .Attr("N: int >= 2")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      const int n = (c->num_inputs() - 1) / 3;
      TF_RETURN_IF_ERROR(shape_inference::ConcatShape(c, n));
      ShapeHandle unused;
      for (int i = n + 1; i < c->num_inputs(); ++i) {
        TF_RETURN_IF_ERROR(c->WithRank(c->input(i), 0, &unused));
      }
      c->set_output(1, c->Scalar());
      c->set_output(2, c->Scalar());
      return Status::OK();
    });

REGISTER_OP("QuantizedReshape")
    .Input("tensor: T")
    .Input("shape: Tshape")
    .Input("input_min: float")
    .Input("input_max: float")
    .Output("output: T")
    .Output("output_min: float")
    .Output("output_max: float")
    .Attr("T: type")
    .Attr("Tshape: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      TF_RETURN_IF_ERROR(SetOutputShapeForReshape(c));
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 0, &unused));
      c->set_output(1, c->Scalar());
      c->set_output(2, c->Scalar());
      return Status::OK();
    });

REGISTER_OP("QuantizedInstanceNorm")
    .Input("x: T")
    .Input("x_min: float")
    .Input("x_max: float")
    .Output("y: T")
    .Output("y_min: float")
    .Output("y_max: float")
    .Attr("T: quantizedtype")
    .Attr("output_range_given: bool = false")
    .Attr("given_y_min: float = 0")
    .Attr("given_y_max: float = 0")
    .Attr("variance_epsilon: float = 1e-5")
    .Attr("min_separation: float = 1e-3")
    .SetShapeFn([](shape_inference::InferenceContext* c) {
      shape_inference::ShapeHandle unused;
      // x should be a rank 4 tensor.
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &unused));
      // Assert x_min and x_max are scalars (rank 0).
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
      // y has the same shape as x.
      TF_RETURN_IF_ERROR(shape_inference::UnchangedShape(c));
      // y_min and y_max are scalars.
      c->set_output(1, c->Scalar());
      c->set_output(2, c->Scalar());
      return Status::OK();
    });

namespace {

Status ScatterNdTensorShape(InferenceContext* c) {
  ShapeHandle output_shape;
  TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &output_shape));
  ShapeHandle indices_shape;
  TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 1, &indices_shape));
  ShapeHandle updates_shape;
  TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(2), 1, &updates_shape));
  return shape_inference::ScatterNdShapeHelper(c, indices_shape, updates_shape,
                                               output_shape);
}

}  // namespace

REGISTER_OP("UpperBound")
    .Input("sorted_inputs: T")
    .Input("values: T")
    .Output("output: out_type")
    .Attr("T: type")
    .Attr("out_type: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle unused_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 2, &unused_shape));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 2, &unused_shape));
      c->set_output(0, c->input(1));
      return Status::OK();
    });

REGISTER_OP("LowerBound")
    .Input("sorted_inputs: T")
    .Input("values: T")
    .Output("output: out_type")
    .Attr("T: type")
    .Attr("out_type: {int32, int64} = DT_INT32")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle unused_shape;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 2, &unused_shape));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 2, &unused_shape));
      c->set_output(0, c->input(1));
      return Status::OK();
    });

REGISTER_OP("ScatterNd")
    .Input("indices: Tindices")
    .Input("updates: T")
    .Input("shape: Tindices")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tindices: {int32, int64}")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle indices_shape;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &indices_shape));
      ShapeHandle updates_shape;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 1, &updates_shape));
      ShapeHandle output_shape;
      TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(2, &output_shape));
      return shape_inference::ScatterNdShapeHelper(c, indices_shape,
                                                   updates_shape, output_shape);
    });
```

```

REGISTER_OP("TensorScatterUpdate")
    .Input("tensor: T")
    .Input("indices: Tindices")
    .Input("updates: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tindices: {int32, int64}")
    .SetShapeFn(ScatterNdTensorShape);

REGISTER_OP("TensorScatterAdd")
    .Input("tensor: T")
    .Input("indices: Tindices")
    .Input("updates: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tindices: {int32, int64}")
    .SetShapeFn(ScatterNdTensorShape);

REGISTER_OP("TensorScatterSub")
    .Input("tensor: T")
    .Input("indices: Tindices")
    .Input("updates: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tindices: {int32, int64}")
    .SetShapeFn(ScatterNdTensorShape);

REGISTER_OP("TensorScatterMin")
    .Input("tensor: T")
    .Input("indices: Tindices")
    .Input("updates: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tindices: {int32, int64}")
    .SetShapeFn(ScatterNdTensorShape);

REGISTER_OP("TensorScatterMax")
    .Input("tensor: T")
    .Input("indices: Tindices")
    .Input("updates: T")
    .Output("output: T")
    .Attr("T: type")
    .Attr("Tindices: {int32, int64}")
    .SetShapeFn(ScatterNdTensorShape);

REGISTER_OP("ScatterNdNonAliasingAdd")
    .Input("input: T")
    .Input("indices: Tindices")
    .Input("updates: T")
    .Output("output: T")
    .Attr("T: {numbertype, bool}")
    .Attr("Tindices: {int32, int64}")
    .SetShapeFn(ScatterNdTensorShape);

REGISTER_OP("FakeQuantWithMinMaxArgs")
    .Attr("min: float = -6.0")
    .Attr("max: float = 6.0")
    .Attr("num_bits: int = 8")
    .Attr("narrow_range: bool = false")
    .Input("inputs: float")
    .Output("outputs: float")
    .SetShapeFn(shape_inference::UnchangedShape);

REGISTER_OP("FakeQuantWithMinMaxArgsGradient")
    .Attr("min: float = -6.0")
    .Attr("max: float = 6.0")
    .Attr("num_bits: int = 8")
    .Attr("narrow_range: bool = false")
    .Input("gradients: float")
    .Input("inputs: float")
    .Output("backprops: float")
    .SetShapeFn(shape_inference::UnchangedShape);

REGISTER_OP("FakeQuantWithMinMaxVars")
    .Attr("num_bits: int = 8")
    .Attr("narrow_range: bool = false")
    .Input("inputs: float")
    .Input("min: float")
    .Input("max: float")
    .Output("outputs: float")
    .SetShapeFn([](InferenceContext* c) {
      TF_RETURN_IF_ERROR(shape_inference::UnchangedShape(c));
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
      return Status::OK();
    });

REGISTER_OP("FakeQuantWithMinMaxVarsGradient")
    .Attr("num_bits: int = 8")
    .Attr("narrow_range: bool = false")
    .Input("gradients: float")
    .Input("inputs: float")
    .Input("min: float")
    .Input("max: float")
    .Output("backprops_wrt_input: float")
    .Output("backprop_wrt_min: float")
```

```cpp
    .Output("backprop_wrt_max: float")
    .SetShapeFn([](InferenceContext* c) {
      // gradients and inputs are same size.
      ShapeHandle inputs;
      TF_RETURN_IF_ERROR(c->Merge(c->input(0), c->input(1), &inputs));

      // min and max are scalars
      ShapeHandle min_max;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &min_max));
      TF_RETURN_IF_ERROR(c->Merge(min_max, c->input(3), &min_max));

      c->set_output(0, inputs);
      c->set_output(1, min_max);
      c->set_output(2, min_max);
      return Status::OK();
    });

REGISTER_OP("FakeQuantWithMinMaxVarsPerChannel")
    .Attr("num_bits: int = 8")
    .Attr("narrow_range: bool = false")
    .Input("inputs: float")
    .Input("min: float")
    .Input("max: float")
    .Output("outputs: float")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle input, min, max;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &input));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &min));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &max));

      DimensionHandle unused;
      TF_RETURN_IF_ERROR(c->Merge(c->Dim(input, -1), c->Dim(min, 0), &unused));
      TF_RETURN_IF_ERROR(c->Merge(c->Dim(input, -1), c->Dim(max, 0), &unused));
      TF_RETURN_IF_ERROR(c->Merge(c->Dim(min, 0), c->Dim(max, 0), &unused));

      c->set_output(0, input);
      return Status::OK();
    });

REGISTER_OP("FakeQuantWithMinMaxVarsPerChannelGradient")
    .Attr("num_bits: int = 8")
    .Attr("narrow_range: bool = false")
    .Input("gradients: float")
    .Input("inputs: float")
    .Input("min: float")
    .Input("max: float")
    .Output("backprops_wrt_input: float")
    .Output("backprop_wrt_min: float")
    .Output("backprop_wrt_max: float")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle inputs;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &inputs));
      TF_RETURN_IF_ERROR(c->WithRankAtMost(inputs, 4, &inputs));
      TF_RETURN_IF_ERROR(c->Merge(inputs, c->input(1), &inputs));

      ShapeHandle last_dim = c->Vector(c->Dim(inputs, -1));

      ShapeHandle min_max;
      TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &min_max));
      TF_RETURN_IF_ERROR(c->Merge(min_max, last_dim, &min_max));
      TF_RETURN_IF_ERROR(c->Merge(c->input(3), min_max, &min_max));

      c->set_output(0, inputs);
      c->set_output(1, min_max);
      c->set_output(2, min_max);
      return Status::OK();
    });

REGISTER_OP("Fingerprint")
    .Input("data: T")
    .Input("method: string")
    .Output("fingerprint: uint8")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      ShapeHandle unused;
      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &unused));
      TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));

      DimensionHandle fingerprint_size;
      const Tensor* method = c->input_tensor(1);
      if (method == nullptr) {
        fingerprint_size = c->UnknownDim();
      } else {
        if (method->dims() != 0) {
          return errors::InvalidArgument("`method` must be rank 0: ",
                                         method->shape());
        }
        const string& method_string = method->scalar<tstring>()();
        if (method_string != "farmhash64") {
          return errors::InvalidArgument("Unsupported method: ", method_string);
        }
        fingerprint_size = c->MakeDim(sizeof(uint64));
      }

      DimensionHandle batch = c->Dim(c->input(0), 0);
      c->set_output(0, c->MakeShape({batch, fingerprint_size}));
      return Status::OK();
    });
```

```

#ifdef INTEL_MKL
REGISTER_OP("_MklConcat")
    .Input("concat_dim: int32")
    .Input("values: N * T")
    .Input("mkl_concat_dim: uint8")
    .Input("mkl_values: N * uint8")
    .Output("output: T")
    .Output("mkl_output: uint8")
    .Attr("N: int >= 2")
    .Attr("T: type")
    .SetShapeFn([](InferenceContext* c) {
      return shape_inference::ConcatShape(c, c->num_inputs() - 3);
    })
    .Doc(R"doc(
MKL version of Concat operator. Uses MKL DNN APIs to perform concatenation.

NOTE Do not invoke this operator directly in Python. Graph rewrite pass is
expected to invoke these operators.
)doc");
#endif

// Deprecated op registrations:

// The following can be deleted after 10mar2017.
REGISTER_OP("BatchMatrixDiag")
    .Input("diagonal: T")
    .Output("output: T")
    .Attr("T: type")
    .Deprecated(14, "Use MatrixDiag")
    .SetShapeFn(shape_inference::UnknownShape);
REGISTER_OP("BatchMatrixSetDiag")
    .Input("input: T")
    .Input("diagonal: T")
    .Output("output: T")
    .Attr("T: type")
    .Deprecated(14, "Use MatrixSetDiag")
    .SetShapeFn(shape_inference::UnknownShape);
REGISTER_OP("BatchMatrixDiagPart")
    .Input("input: T")
    .Output("diagonal: T")
    .Attr("T: type")
    .Deprecated(14, "Use MatrixDiagPart")
    .SetShapeFn(shape_inference::UnknownShape);
REGISTER_OP("BatchMatrixBandPart")
    .Input("input: T")
    .Input("num_lower: int64")
    .Input("num_upper: int64")
    .Output("band: T")
    .Attr("T: type")
    .Deprecated(14, "Use MatrixBandPart")
    .SetShapeFn(shape_inference::UnknownShape);

}  // namespace tensorflow
```