3415 lines (3082 sloc) | 119 KB ...

```
1  /* Copyright 2015 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
6
7      http://www.apache.org/licenses/LICENSE-2.0
8
9  Unless required by applicable law or agreed to in writing, software
10 distributed under the License is distributed on an "AS IS" BASIS,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 ==============================================================================*/
15
16 #include <algorithm>
17 #include <ostream>
18
19 #include "tensorflow/core/framework/common_shape_fns.h"
20 #include "tensorflow/core/framework/kernel_shape_util.h"
21 #include "tensorflow/core/framework/op.h"
22 #include "tensorflow/core/framework/shape_inference.h"
23 #include "tensorflow/core/framework/tensor.pb.h"
24 #include "tensorflow/core/framework/types.h"
25 #include "tensorflow/core/framework/types.pb.h"
26 #include "tensorflow/core/lib/core/errors.h"
27 #include "tensorflow/core/util/mirror_pad_mode.h"
28 #include "tensorflow/core/util/padding.h"
29 #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);
    }
```

```
177     } else {
178       for (int i = 0; i < rank; ++i) {
179         dims[i] = c->UnknownDim();
180       }
181     }
182
183     c->set_output(0, c->MakeShape(dims));
184     return Status::OK();
185   }
186
187   Status SetOutputShapeForReshape(InferenceContext* c) {
188     ShapeHandle in = c->input(0);
189     ShapeHandle out;
190     TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(1, &out));
191
192     if (!c->RankKnown(out)) {
193       // We have no information about the shape of the output.
194       c->set_output(0, out);
195       return Status::OK();
196     }
197     if (c->RankKnown(in)) {
198       // We don't know the number of output elements, but we can try to infer
199       // the missing dimension.
200       bool too_many_unknown = false;
201       int32_t out_unknown_idx = -1;
202
203       DimensionHandle known_out_elems = c->NumElements(out);
204       if (!c->ValueKnown(known_out_elems)) {
205         known_out_elems = c->MakeDim(1);
206         for (int32_t i = 0; i < c->Rank(out); ++i) {
207           DimensionHandle dim = c->Dim(out, i);
208           if (!c->ValueKnown(dim)) {
209             if (out_unknown_idx >= 0) {
210               too_many_unknown = true;
211               break;
212             }
213             out_unknown_idx = i;
214           } else {
215             TF_RETURN_IF_ERROR(
216                 c->Multiply(known_out_elems, dim, &known_out_elems));
217           }
218         }
219       }
220       int32_t in_unknown_idx = -1;
221       DimensionHandle known_in_elems = c->NumElements(in);
222       if (!c->ValueKnown(known_in_elems)) {
223         known_in_elems = c->MakeDim(1);
224         for (int32_t i = 0; i < c->Rank(in); ++i) {
225           DimensionHandle dim = c->Dim(in, i);
```

```
226         if (!c->ValueKnown(dim)) {
227           if (in_unknown_idx >= 0) {
228             too_many_unknown = true;
229             break;
230           }
231           in_unknown_idx = i;
232         } else {
233           TF_RETURN_IF_ERROR(c->Multiply(known_in_elems, dim, &known_in_elems));
234         }
235       }
236     }
237
238     if (!too_many_unknown) {
239       if (in_unknown_idx < 0 && out_unknown_idx < 0) {
240         // Just check that the dimensions match.
241         if (c->Value(known_in_elems) != c->Value(known_out_elems)) {
242           return errors::InvalidArgument(
243               "Cannot reshape a tensor with ", c->DebugString(known_in_elems),
244               " elements to shape ", c->DebugString(out), " (",
245               c->DebugString(known_out_elems), " elements)");
246         }
247       } else if (in_unknown_idx < 0 && out_unknown_idx >= 0 &&
248                  c->Value(known_out_elems) > 0) {
249         // Input fully known, infer the one missing output dim
250         DimensionHandle inferred_dim;
251         TF_RETURN_IF_ERROR(c->Divide(known_in_elems, c->Value(known_out_elems),
252                                      true /* evenly_divisible */,
253                                      &inferred_dim));
254         TF_RETURN_IF_ERROR(
255             c->ReplaceDim(out, out_unknown_idx, inferred_dim, &out));
256
257       } else if (in_unknown_idx >= 0 && out_unknown_idx < 0 &&
258                  c->Value(known_in_elems) != 0) {
259         // Output fully known, infer the one missing input dim
260         DimensionHandle inferred_dim;
261         TF_RETURN_IF_ERROR(c->Divide(known_out_elems, c->Value(known_in_elems),
262                                      true /* evenly_divisible */,
263                                      &inferred_dim));
264         DimensionHandle unknown_in_dim = c->Dim(in, in_unknown_idx);
265         TF_RETURN_IF_ERROR(
266             c->Merge(unknown_in_dim, inferred_dim, &unknown_in_dim));
267       } else if (in_unknown_idx >= 0 && out_unknown_idx >= 0) {
268         // Exactly one unknown dimension in both input and output. These 2 are
269         // equal iff the known elements are equal.
270         if (c->Value(known_in_elems) == c->Value(known_out_elems)) {
271           DimensionHandle unknown_in_dim = c->Dim(in, in_unknown_idx);
272           TF_RETURN_IF_ERROR(
273               c->ReplaceDim(out, out_unknown_idx, unknown_in_dim, &out));
274         }
```

```
      }
    }
  }
  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")
```

```
373     .SetIsStateful()
374     .SetShapeFn(UnchangedShape);
375
376 REGISTER_OP("InplaceUpdate")
377     .Input("x: T")
378     .Input("i: int32")
379     .Input("v: T")
380     .Output("y: T")
381     .Attr("T: type")
382     .SetShapeFn(UnchangedShape);
383
384 REGISTER_OP("InplaceAdd")
385     .Input("x: T")
386     .Input("i: int32")
387     .Input("v: T")
388     .Output("y: T")
389     .Attr("T: type")
390     .SetShapeFn(UnchangedShape);
391
392 REGISTER_OP("InplaceSub")
393     .Input("x: T")
394     .Input("i: int32")
395     .Input("v: T")
396     .Output("y: T")
397     .Attr("T: type")
398     .SetShapeFn(UnchangedShape);
399
400 REGISTER_OP("Empty")
401     .Input("shape: int32")
402     .Output("output: dtype")
403     .Attr("dtype: type")
404     .Attr("init: bool = false")
405     .SetDoNotOptimize()
406     .SetShapeFn([](InferenceContext* c) {
407       ShapeHandle out;
408       TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &out));
409       c->set_output(0, out);
410       return Status::OK();
411     });
412
413 // ----------------------------------------------------------------------------
414 REGISTER_OP("Unpack")
415     .Input("value: T")
416     .Output("output: num * T")
417     .Attr("num: int >= 0")
418     .Attr("T: type")
419     .Attr("axis: int = 0")
420     .SetShapeFn([](InferenceContext* c) {
421       ShapeHandle s = c->input(0);
```

```
422       ShapeHandle out;
423       if (c->RankKnown(s)) {
424         // Determine the axis that will be removed, converting from negative
425         // axes to a positive point per negative indexing rules.
426         int32_t rank = c->Rank(s);
427         int32_t axis;
428         TF_RETURN_IF_ERROR(GetAxisForPackAndUnpack(c, rank, &axis));
429
430         // The axis dim matches the number of outputs.
431         DimensionHandle unused;
432         TF_RETURN_IF_ERROR(
433             c->WithValue(c->Dim(s, axis), c->num_outputs(), &unused));
434
435         // Copy all dimensions, removing the <axis> dimension.
436         std::vector<DimensionHandle> dims;
437         for (int i = 0; i < rank; ++i) {
438           if (i != axis) dims.push_back(c->Dim(s, i));
439         }
440         out = c->MakeShape(dims);
441       } else {
442         // All outputs are the same shape, but it's not known.
443         out = c->UnknownShape();
444       }
445       for (int i = 0; i < c->num_outputs(); ++i) c->set_output(i, out);
446       return Status::OK();
447     });
448
449 REGISTER_OP("UnravelIndex")
450     .Input("indices: Tidx")
451     .Input("dims: Tidx")
452     .Output("output: Tidx")
453     .Attr("Tidx: {int32, int64} = DT_INT32")
454     .SetShapeFn([](InferenceContext* c) {
455       ShapeHandle indices = c->input(0);
456       ShapeHandle dims;
457       TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &dims));
458       if (c->RankKnown(indices) && c->Rank(indices) == 0) {
459         c->set_output(0, c->Vector(c->Dim(dims, 0)));
460       } else if (c->RankKnown(indices)) {
461         c->set_output(0, c->Matrix(c->Dim(dims, 0), c->NumElements(indices)));
462       } else {
463         c->set_output(0, c->UnknownShape());
464       }
465       return Status::OK();
466     });
467
468 REGISTER_OP("BroadcastTo")
469     .Input("input: T")
470     .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);
```

```
618       if (rank == InferenceContext::kUnknownRank) {
619         // If the rank of input tensor is unknown, then return unknown shapes.
620         // Note that the shape of each output can be different.
621         for (int i = 0; i < num_outputs; ++i) {
622           c->set_output(i, c->UnknownShape());
623         }
624       } else if (rank == 0) {
625         // Throw error if input is a scalar.
626         return errors::InvalidArgument("Can't split scalars");
627       } else if (size_splits == nullptr && c->ValueKnown(split_dimension)) {
628         // If split dimension is known, but the sizes are unknown, then
629         // only the split dimension is unknown
630         output_shape = input;
631         for (int i = 0; i < num_outputs; ++i) {
632           TF_RETURN_IF_ERROR(c->ReplaceDim(output_shape,
633                                            c->Value(split_dimension),
634                                            c->UnknownDim(), &output_shape));
635           c->set_output(i, output_shape);
636         }
637       } else if (size_splits == nullptr && !c->ValueKnown(split_dimension)) {
638         // If split dimension or tensor containing the split sizes is unknown,
639         // then return unknown shapes of same rank as input. Note that each
640         // output shape can be different since splitv doesn't always split
641         // tensors evenly.
642         for (int i = 0; i < num_outputs; ++i) {
643           c->set_output(i, c->UnknownShapeOfRank(rank));
644         }
645       } else {
646         // Determine the output shape if split dimension and split sizes are
647         // known.
648         int64_t split_dim = c->Value(split_dimension);
649         TF_RETURN_IF_ERROR(c->WithRankAtLeast(input, split_dim + 1, &input));
650         std::vector<int64_t> data;
651         if (size_splits->dtype() == DT_INT32) {
652           data = AsInt64<int32>(size_splits, size_splits->shape().dim_size(0));
653         } else {
654           data =
655               AsInt64<int64_t>(size_splits, size_splits->shape().dim_size(0));
656         }
657         if (num_outputs != data.size()) {
658           return errors::InvalidArgument(
659               "Length of size_splits should be equal to num_outputs");
660         }
661         int64_t total_size = 0;
662         bool has_neg_one = false;
663         for (const auto size : data) {
664           if (size == -1) {
665             if (has_neg_one) {
666               return errors::InvalidArgument(
```

```
667                   "size_splits can only have one -1");
668             }
669             has_neg_one = true;
670           } else {
671             total_size += size;
672           }
673         }
674         auto split_dim_size = c->Value(c->Dim(input, split_dim));
675         // If the sizes of the splits are known, then
676         // make sure that the sizes add up to the expected
677         // dimension size, with the possibility of a -1.
678         // Specify the full output shapes.
679         for (int i = 0; i < num_outputs; ++i) {
680           auto size = data[i];
681           if (data[i] == -1 && c->ValueKnown(split_dim_size)) {
682             size = split_dim_size - total_size;
683           }
684           // If we have a negative known size (either explicit, or computed
685           // via -1), then the split sizes are invalid.
686           if (size < -1 || (size == -1 && c->ValueKnown(split_dim_size))) {
687             return errors::InvalidArgument("Split size at index ", i,
688                                            " must be >= 0. Got: ", size);
689           }
690           TF_RETURN_IF_ERROR(
691               c->ReplaceDim(input, split_dim, c->MakeDim(size), &output_shape));
692           c->set_output(i, output_shape);
693         }
694         if (c->ValueKnown(split_dim_size)) {
695           if (has_neg_one ? total_size > split_dim_size
696                           : total_size != split_dim_size) {
697             return errors::InvalidArgument(
698                 "can't split axis of size ", split_dim_size,
699                 " into pieces of size [", absl::StrJoin(data, ","), "]");
700           }
701         }
702       }
703
704       return Status::OK();
705     });
706
707 // --------------------------------------------------------------------------
708 REGISTER_OP("Const")
709     .Output("output: dtype")
710     .Attr("value: tensor")
711     .Attr("dtype: type")
712     .SetShapeFn([](InferenceContext* c) {
713       const TensorProto* proto = nullptr;
714       TF_RETURN_IF_ERROR(c->GetAttr("value", &proto));
715       TF_RETURN_IF_ERROR(TensorShape::IsValidShape(proto->tensor_shape()));
```

```
716       TensorShape shape(proto->tensor_shape());
717       std::vector<DimensionHandle> dims;
718       dims.reserve(shape.dims());
719       for (int i = 0; i < shape.dims(); ++i) {
720         dims.push_back(c->MakeDim(shape.dim_size(i)));
721       }
722       c->set_output(0, c->MakeShape(dims));
723       return Status::OK();
724     });
725
726 // Returns a constant tensor on the host.  Useful for writing C++ tests
727 // and benchmarks which run on GPU but require arguments pinned to the host.
728 // Used by test::graph::HostConstant.
729 // value: Attr `value` is the tensor to return.
730 REGISTER_OP("HostConst")
731     .Output("output: dtype")
732     .Attr("value: tensor")
733     .Attr("dtype: type")
734     .SetShapeFn(shape_inference::UnknownShape);
735
736 // Used executing op-by-op to copy constants to the current device without
737 // serializing tensors as TensorProtos, after a host tensor has been
738 // created. Same behavior as Identity, but no gradient and potentially relaxed
739 // copy semantics.
740 REGISTER_OP("_EagerConst")
741     .Input("input: T")
742     .Output("output: T")
743     .Attr("T: type")
744     .SetShapeFn(shape_inference::UnchangedShape);
745
746 // --------------------------------------------------------------------------
747 // TODO(mgubin): Update the doc when the freeze_graph script supports converting
748 // into memmapped format.
749 REGISTER_OP("ImmutableConst")
750     .Attr("dtype: type")
751     .Attr("shape: shape")
752     .Attr("memory_region_name: string")
753     .Output("tensor: dtype")
754     .SetShapeFn(shape_inference::ExplicitShape);
755
756 REGISTER_OP("GuaranteeConst")
757     .Input("input: T")
758     .Output("output: T")
759     .Attr("T: type")
760     .SetShapeFn([](shape_inference::InferenceContext* c) {
761       return UnchangedShape(c);
762     })
763     // We don't want this to be optimized away.
764     .SetDoNotOptimize();
```

```
765
766 // --------------------------------------------------------------------------
767 REGISTER_OP("ZerosLike")
768     .Input("x: T")
769     .Output("y: T")
770     .Attr("T: type")
771     .SetShapeFn(shape_inference::UnchangedShape);
772
773 // --------------------------------------------------------------------------
774 REGISTER_OP("OnesLike")
775     .Input("x: T")
776     .Output("y: T")
777     .Attr(
778         "T: {bfloat16, half, float, double, int8, uint8, int16, uint16, int32, "
779         "uint32, int64, uint64, complex64, complex128, bool}")
780     .SetShapeFn(shape_inference::UnchangedShape);
781
782 // --------------------------------------------------------------------------
783 REGISTER_OP("Diag")
784     .Input("diagonal: T")
785     .Output("output: T")
786     .Attr(
787         "T: {bfloat16, half, float, double, int32, int64, complex64, "
788         "complex128}")
789     .SetShapeFn([](InferenceContext* c) {
790       ShapeHandle in = c->input(0);
791       TF_RETURN_IF_ERROR(c->WithRankAtLeast(in, 1, &in));
792       // Output shape is original concatenated with itself.
793       ShapeHandle out;
794       TF_RETURN_IF_ERROR(c->Concatenate(in, in, &out));
795       c->set_output(0, out);
796       return Status::OK();
797     });
798
799 // --------------------------------------------------------------------------
800 REGISTER_OP("DiagPart")
801     .Input("input: T")
802     .Output("diagonal: T")
803     .Attr(
804         "T: {bfloat16, half, float, double, int32, int64, complex64, "
805         "complex128}")
806     .SetShapeFn([](InferenceContext* c) {
807       ShapeHandle in = c->input(0);
808       if (!c->RankKnown(in)) {
809         c->set_output(0, c->UnknownShape());
810         return Status::OK();
811       }
812       // Rank must be even, and result will have rank <rank/2>.
813       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")
```

```
863     .Input("num_rows: int32")
864     .Input("num_cols: int32")
865     .Input("padding_value: T")
866     .Output("output: T")
867     .Attr("T: type")
868     .Attr(
869         "align: {'LEFT_RIGHT', 'RIGHT_LEFT', 'LEFT_LEFT', 'RIGHT_RIGHT'} = "
870         "'RIGHT_LEFT'")
871     .SetShapeFn(shape_inference::MatrixDiagV2Shape);
872
873 // ----------------------------------------------------------------------
874 REGISTER_OP("MatrixSetDiag")
875     .Input("input: T")
876     .Input("diagonal: T")
877     .Output("output: T")
878     .Attr("T: type")
879     .SetShapeFn([](InferenceContext* c) {
880       ShapeHandle input;
881       ShapeHandle diag;
882       TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &input));
883       TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 1, &diag));
884       if (c->RankKnown(input)) {
885         TF_RETURN_IF_ERROR(c->WithRank(c->input(1), c->Rank(input) - 1, &diag));
886       }
887       DimensionHandle smallest_dim;
888       TF_RETURN_IF_ERROR(
889           c->Min(c->Dim(input, -2), c->Dim(input, -1), &smallest_dim));
890       TF_RETURN_IF_ERROR(
891           c->Merge(smallest_dim, c->Dim(diag, -1), &smallest_dim));
892
893       ShapeHandle output = input;
894       if (c->RankKnown(diag) && !c->FullyDefined(input)) {
895         // Try to infer parts of shape from diag.
896         ShapeHandle diag_batch_shape;
897         TF_RETURN_IF_ERROR(c->Subshape(diag, 0, -1, &diag_batch_shape));
898         TF_RETURN_IF_ERROR(
899             c->Concatenate(diag_batch_shape, c->UnknownShapeOfRank(2), &diag));
900         TF_RETURN_IF_ERROR(c->Merge(input, diag, &output));
901       }
902       c->set_output(0, output);
903       return Status::OK();
904     });
905
906 REGISTER_OP("MatrixSetDiagV2")
907     .Input("input: T")
908     .Input("diagonal: T")
909     .Input("k: int32")
910     .Output("output: T")
911     .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")
```

```
961      .Input("padding_value: T")
962      .Output("diagonal: T")
963      .Attr("T: type")
964      .Attr(
965          "align: {'LEFT_RIGHT', 'RIGHT_LEFT', 'LEFT_LEFT', 'RIGHT_RIGHT'} = "
966          "'RIGHT_LEFT'")
967      .SetShapeFn(shape_inference::MatrixDiagPartV2Shape);
968
969  // ----------------------------------------------------------------------
970  REGISTER_OP("MatrixBandPart")
971      .Input("input: T")
972      .Input("num_lower: Tindex")
973      .Input("num_upper: Tindex")
974      .Output("band: T")
975      .Attr("T: type")
976      .Attr("Tindex: {int32, int64} = DT_INT64")
977      .SetShapeFn(shape_inference::UnchangedShape);
978
979  // ----------------------------------------------------------------------
980  REGISTER_OP("Reverse")
981      .Input("tensor: T")
982      .Input("dims: bool")
983      .Output("output: T")
984      .Attr(
985          "T: {uint8, int8, uint16, int16, uint32, int32, uint64, int64, bool, "
986          "bfloat16, half, float, double, complex64, complex128, string}")
987      .SetShapeFn([](InferenceContext* c) {
988        ShapeHandle input = c->input(0);
989        ShapeHandle dims;
990        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &dims));
991        DimensionHandle dims_dim = c->Dim(dims, 0);
992        if (c->ValueKnown(dims_dim)) {
993          TF_RETURN_IF_ERROR(c->WithRank(input, c->Value(dims_dim), &input));
994        }
995        if (c->Rank(input) > 8) {
996          return errors::InvalidArgument(
997              "reverse does not work on tensors with more than 8 dimensions");
998        }
999        c->set_output(0, input);
1000       return Status::OK();
1001     });
1002
1003 // ----------------------------------------------------------------------
1004 REGISTER_OP("ReverseV2")
1005     .Input("tensor: T")
1006     .Input("axis: Tidx")
1007     .Output("output: T")
1008     .Attr("Tidx: {int32, int64} = DT_INT32")
1009     .Attr(
```

```
1010          "T: {uint8, int8, uint16, int16, int32, uint32, int64, uint64, bool, "
1011          "bfloat16, half, float, double, complex64, complex128, string}")
1012      .SetShapeFn([](InferenceContext* c) {
1013        ShapeHandle input = c->input(0);
1014        ShapeHandle axis;
1015        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &axis));
1016        if (c->Rank(input) > 8) {
1017          return errors::InvalidArgument(
1018              "reverse does not work on tensors with more than 8 dimensions");
1019        }
1020        const Tensor* axis_tensor = c->input_tensor(1);
1021        if (axis_tensor != nullptr && c->RankKnown(input)) {
1022          int32_t rank = c->Rank(input);
1023          std::vector<int64_t> axis_value;
1024          if (axis_tensor->dtype() == DT_INT32) {
1025            axis_value = AsInt64<int32>(axis_tensor, axis_tensor->NumElements());
1026          } else {
1027            axis_value =
1028                AsInt64<int64_t>(axis_tensor, axis_tensor->NumElements());
1029          }
1030          std::vector<bool> axes_dense(c->Rank(input), false);
1031          for (int i = 0; i < axis_value.size(); i++) {
1032            int64_t canonical_axis =
1033                axis_value[i] < 0 ? rank + axis_value[i] : axis_value[i];
1034            if (canonical_axis < 0 || canonical_axis >= rank) {
1035              return errors::InvalidArgument("'axis'[", i, "] = ", axis_value[i],
1036                                             " is out of valid range [", 0, ", ",
1037                                             rank - 1);
1038            }
1039            if (axes_dense[canonical_axis]) {
1040              return errors::InvalidArgument("axis ", canonical_axis,
1041                                             " specified more than once.");
1042            }
1043            axes_dense[canonical_axis] = true;
1044          }
1045        }
1046        c->set_output(0, input);
1047        return Status::OK();
1048      });
1049
1050  // --------------------------------------------------------------------------
1051  REGISTER_OP("EditDistance")
1052      .Input("hypothesis_indices: int64")
1053      .Input("hypothesis_values: T")
1054      .Input("hypothesis_shape: int64")
1055      .Input("truth_indices: int64")
1056      .Input("truth_values: T")
1057      .Input("truth_shape: int64")
1058      .Attr("normalize: bool = true")
```

```
1059      .Attr("T: type")
1060      .Output("output: float")
1061      .SetShapeFn([](InferenceContext* c) {
1062        TF_RETURN_IF_ERROR(shape_inference::ValidateSparseTensor(
1063            c, c->input(0), c->input(1), c->input(2)));
1064        TF_RETURN_IF_ERROR(shape_inference::ValidateSparseTensor(
1065            c, c->input(3), c->input(4), c->input(5)));
1066        const Tensor* hypothesis_shape_t = c->input_tensor(2);
1067        const Tensor* truth_shape_t = c->input_tensor(5);
1068        if (hypothesis_shape_t == nullptr || truth_shape_t == nullptr) {
1069          // We need to know the runtime shape of the two tensors,
1070          // or else the output shape is unknown.
1071          return shape_inference::UnknownShape(c);
1072        }
1073
1074        if (hypothesis_shape_t->NumElements() != truth_shape_t->NumElements()) {
1075          return errors::InvalidArgument(
1076              "Num elements of hypothesis_shape does not match truth_shape: ",
1077              hypothesis_shape_t->NumElements(), " vs. ",
1078              truth_shape_t->NumElements());
1079        }
1080
1081        auto h_values = hypothesis_shape_t->flat<int64_t>();
1082        auto t_values = truth_shape_t->flat<int64_t>();
1083        std::vector<DimensionHandle> dims(hypothesis_shape_t->NumElements() - 1);
1084        for (int i = 0; i < dims.size(); ++i) {
1085          dims[i] = c->MakeDim(std::max(h_values(i), t_values(i)));
1086        }
1087
1088        c->set_output(0, c->MakeShape(dims));
1089        return Status::OK();
1090      });
1091
1092  // ----------------------------------------------------------------------
1093  REGISTER_OP("Fill")
1094      .Input("dims: index_type")
1095      .Input("value: T")
1096      .Output("output: T")
1097      .Attr("T: type")
1098      .Attr("index_type: {int32, int64} = DT_INT32")
1099      .SetShapeFn([](InferenceContext* c) {
1100        DataType index_type = DT_INT32;
1101        Status s = c->GetAttr("index_type", &index_type);
1102        if (!s.ok() && s.code() != error::NOT_FOUND) {
1103          return s;
1104        }
1105        ShapeHandle unused;
1106        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &unused));
1107        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));
```

```
1108
1109        const Tensor* t = c->input_tensor(0);
1110        if (t != nullptr) {
1111          for (int i = 0; i < t->NumElements(); ++i) {
1112            if ((index_type == DT_INT32 && t->vec<int32>()(i) < 0) ||
1113                (index_type == DT_INT64 && t->vec<int64_t>()(i) < 0)) {
1114              return errors::InvalidArgument("Fill dimensions must be >= 0");
1115            }
1116          }
1117        }
1118
1119        ShapeHandle out;
1120        TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &out));
1121        c->set_output(0, out);
1122
1123        auto* shape_and_type = c->input_handle_shapes_and_types(1);
1124        if (shape_and_type) {
1125          c->set_output_handle_shapes_and_types(0, *shape_and_type);
1126        }
1127
1128        return Status::OK();
1129      });
1130
1131  // ----------------------------------------------------------------------------
1132  REGISTER_OP("_ParallelConcatStart")
1133      .Output("output: dtype")
1134      .Attr("shape: shape")
1135      .Attr("dtype: type")
1136      .SetIsStateful()
1137      .SetShapeFn(shape_inference::ExplicitShape)
1138      .Doc(R"doc(
1139  Creates an empty Tensor with shape `shape` and type `dtype`.
1140
1141  The memory can optionally be initialized. This is usually useful in
1142  conjunction with inplace operations.
1143
1144  shape: 1-D `Tensor` indicating the shape of the output.
1145  dtype: The element type of the returned tensor.
1146  output: An empty Tensor of the specified type.
1147  )doc");
1148
1149  // ----------------------------------------------------------------------------
1150  REGISTER_OP("_ParallelConcatUpdate")
1151      .Input("value: T")
1152      .Input("update: T")
1153      .Output("output: T")
1154      .Attr("T: type")
1155      .Attr("loc: int")
1156      .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));
```

```
1206
1207        ShapeHandle indices_shape = c->input(1);
1208        ShapeHandle unused_axis_shape;
1209        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused_axis_shape));
1210        const Tensor* axis_t = c->input_tensor(2);
1211
1212        // If axis is unknown, we can only infer that the result is params_rank +
1213        // indices_rank - 1.
1214        if (axis_t == nullptr) {
1215          if (c->RankKnown(params_shape) && c->RankKnown(indices_shape)) {
1216            int32_t batch_dims;
1217            TF_RETURN_IF_ERROR(c->GetAttr("batch_dims", &batch_dims));
1218            c->set_output(0, c->UnknownShapeOfRank(c->Rank(params_shape) +
1219                                                   c->Rank(indices_shape) - 1 -
1220                                                   batch_dims));
1221          } else {
1222            c->set_output(0, c->UnknownShape());
1223          }
1224          return Status::OK();
1225        }
1226
1227        // Note, axis can be negative.
1228        int64_t axis = 0;
1229        if (axis_t->dtype() == DT_INT32) {
1230          axis = axis_t->scalar<int32>()();
1231        } else {
1232          axis = axis_t->scalar<int64_t>()();
1233        }
1234
1235        // Check that params has rank of at least axis + 1.
1236        ShapeHandle unused;
1237        TF_RETURN_IF_ERROR(c->WithRankAtLeast(
1238            params_shape, axis < 0 ? -axis : axis + 1, &unused));
1239
1240        // Note, batch_dims can be negative.
1241        int32_t batch_dims;
1242        TF_RETURN_IF_ERROR(c->GetAttr("batch_dims", &batch_dims));
1243        // -rank(indices) <= batch_dims <= rank(indices)
1244        TF_RETURN_IF_ERROR(
1245            c->WithRankAtLeast(indices_shape, std::abs(batch_dims), &unused));
1246        if (batch_dims < 0) {
1247          batch_dims += c->Rank(indices_shape);
1248        }
1249        // rank(params) > batch_dims
1250        TF_RETURN_IF_ERROR(
1251            c->WithRankAtLeast(params_shape, batch_dims + 1, &unused));
1252
1253        ShapeHandle params_outer_subshape;
1254        TF_RETURN_IF_ERROR(
```

```
1255            c->Subshape(params_shape, 0, axis, &params_outer_subshape));
1256
1257        ShapeHandle indices_inner_subshape;
1258        TF_RETURN_IF_ERROR(
1259            c->Subshape(indices_shape, batch_dims, &indices_inner_subshape));
1260
1261        ShapeHandle out;
1262        TF_RETURN_IF_ERROR(
1263            c->Concatenate(params_outer_subshape, indices_inner_subshape, &out));
1264
1265        // Slice from axis + 1 to the end of params_shape to collect the inner
1266        // dimensions of the result. Special case -1 here since -1 + 1 wraps, and
1267        // we slice from 0 to the end of shape. Subshape() handles all other
1268        // out-of-bounds checking.
1269        if (axis != -1) {
1270          ShapeHandle params_inner_subshape;
1271          TF_RETURN_IF_ERROR(
1272              c->Subshape(params_shape, axis + 1, &params_inner_subshape));
1273          TF_RETURN_IF_ERROR(c->Concatenate(out, params_inner_subshape, &out));
1274        }
1275
1276        c->set_output(0, out);
1277        return Status::OK();
1278      });
1279
1280  // ----------------------------------------------------------------------
1281  REGISTER_OP("GatherNd")
1282      .Input("params: Tparams")
1283      .Input("indices: Tindices")
1284      .Output("output: Tparams")
1285      .Attr("Tparams: type")
1286      .Attr("Tindices: {int32,int64}")
1287      .SetShapeFn(shape_inference::GatherNdShape);
1288
1289  // ----------------------------------------------------------------------
1290  REGISTER_OP("Identity")
1291      .Input("input: T")
1292      .Output("output: T")
1293      .Attr("T: type")
1294      .SetForwardTypeFn(full_type::ReplicateInput())
1295      .SetShapeFn(shape_inference::UnchangedShape);
1296
1297  REGISTER_OP("Snapshot")
1298      .Input("input: T")
1299      .Output("output: T")
1300      .Attr("T: type")
1301      .SetShapeFn(shape_inference::UnchangedShape);
1302
1303  #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)")
```

```
1353      .Attr("T: type")
1354      .SetShapeFn(shape_inference::UnchangedShape)
1355      .SetAllowsUninitializedInput();
1356
1357  // --------------------------------------------------------------------------
1358  REGISTER_OP("StopGradient")
1359      .Input("input: T")
1360      .Output("output: T")
1361      .Attr("T: type")
1362      .SetShapeFn(shape_inference::UnchangedShape);
1363
1364  REGISTER_OP("PreventGradient")
1365      .Input("input: T")
1366      .Output("output: T")
1367      .Attr("T: type")
1368      .Attr("message: string = ''")
1369      .SetShapeFn(shape_inference::UnchangedShape);
1370
1371  // --------------------------------------------------------------------------
1372  REGISTER_OP("CheckNumerics")
1373      .Input("tensor: T")
1374      .Output("output: T")
1375      .Attr("T: {bfloat16, half, float, double}")
1376      .Attr("message: string")
1377      .SetIsStateful()
1378      .SetShapeFn(shape_inference::UnchangedShape);
1379
1380  // --------------------------------------------------------------------------
1381  REGISTER_OP("CheckNumericsV2")
1382      .Input("tensor: T")
1383      .Output("output: T")
1384      .Attr("T: {bfloat16, half, float, double}")
1385      .Attr("message: string")
1386      .SetIsStateful()
1387      .SetShapeFn(shape_inference::UnchangedShape);
1388
1389  // --------------------------------------------------------------------------
1390  REGISTER_OP("Reshape")
1391      .Input("tensor: T")
1392      .Input("shape: Tshape")
1393      .Output("output: T")
1394      .Attr("T: type")
1395      .Attr("Tshape: {int32, int64} = DT_INT32")
1396      .SetShapeFn([](InferenceContext* c) {
1397        return SetOutputShapeForReshape(c);
1398      });
1399
1400  #ifdef INTEL_MKL
1401  REGISTER_OP("_MklReshape")
```

```
1402      .Input("tensor: T")
1403      .Input("shape: Tshape")
1404      .Input("mkl_tensor: uint8")
1405      .Input("mkl_shape: uint8")
1406      .Output("output: T")
1407      .Output("mkl_output: uint8")
1408      .Attr("T: type")
1409      .Attr("Tshape: {int32, int64} = DT_INT32")
1410      .SetShapeFn([](InferenceContext* c) { return SetOutputShapeForReshape(c); })
1411      .Doc(R"Doc( MKL implementation of ReshapeOp.
1412  )Doc");
1413  #endif  // INTEL_MKL
1414
1415  // --------------------------------------------------------------------------
1416  REGISTER_OP("InvertPermutation")
1417      .Input("x: T")
1418      .Output("y: T")
1419      .Attr("T: {int32, int64} = DT_INT32")
1420      .SetShapeFn([](InferenceContext* c) {
1421        ShapeHandle x;
1422        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &x));
1423        c->set_output(0, x);
1424        return Status::OK();
1425      });
1426
1427  // --------------------------------------------------------------------------
1428  REGISTER_OP("Transpose")
1429      .Input("x: T")
1430      .Input("perm: Tperm")
1431      .Output("y: T")
1432      .Attr("T: type")
1433      .Attr("Tperm: {int32, int64} = DT_INT32")
1434      .SetShapeFn(TransposeShapeFn);
1435
1436  #ifdef INTEL_MKL
1437  REGISTER_OP("_MklTranspose")
1438      .Input("x: T")
1439      .Input("perm: Tperm")
1440      .Output("y: T")
1441      .Attr("T: type")
1442      .Attr("Tperm: {int32, int64} = DT_INT32")
1443      .SetShapeFn(TransposeShapeFn);
1444  #endif  // INTEL_MKL
1445
1446  // --------------------------------------------------------------------------
1447  REGISTER_OP("ConjugateTranspose")
1448      .Input("x: T")
1449      .Input("perm: Tperm")
1450      .Output("y: T")
```

```
1451      .Attr("T: type")
1452      .Attr("Tperm: {int32, int64} = DT_INT32")
1453      .SetShapeFn(TransposeShapeFn);
1454
1455  #ifdef INTEL_MKL
1456  REGISTER_OP("_MklConjugateTranspose")
1457      .Input("x: T")
1458      .Input("perm: Tperm")
1459      .Output("y: T")
1460      .Attr("T: type")
1461      .Attr("Tperm: {int32, int64} = DT_INT32")
1462      .SetShapeFn(TransposeShapeFn);
1463  #endif  // INTEL_MKL
1464
1465  // --------------------------------------------------------------------------
1466  namespace {
1467  Status UniqueIdxShapeFn(InferenceContext* c) {
1468    ShapeHandle input = c->input(0);
1469    const Tensor* axis_t = c->input_tensor(1);
1470    if (axis_t == nullptr || !c->RankKnown(input)) {
1471      c->set_output(1, c->Vector(InferenceContext::kUnknownDim));
1472      return Status::OK();
1473    }
1474
1475    if (c->Rank(c->input(1)) != 1) {
1476      return errors::InvalidArgument("axis expects a 1D vector.");
1477    }
1478
1479    int32_t n = axis_t->NumElements();
1480    if (n == 0) {
1481      if (c->Rank(input) != 1) {
1482        return errors::InvalidArgument("x expects a 1D vector.");
1483      }
1484      c->set_output(1, input);
1485      return Status::OK();
1486    } else if (n == 1) {
1487      int64_t axis;
1488      if (axis_t->dtype() == DT_INT32) {
1489        axis = static_cast<int64_t>(axis_t->flat<int32>()(0));
1490      } else {
1491        axis = axis_t->flat<int64_t>()(0);
1492      }
1493
1494      int64_t input_rank = c->Rank(input);
1495      if (axis < -input_rank || axis >= input_rank) {
1496        return errors::InvalidArgument("axis expects to be in the range [",
1497                                       -input_rank, ", ", input_rank, ")");
1498      }
1499      if (axis < 0) {
```

```
1500        axis += input_rank;
1501      }
1502      c->set_output(1, c->Vector(c->Dim(input, axis)));
1503      return Status::OK();
1504    }
1505    return errors::InvalidArgument(
1506        "axis does not support input tensors larger than 1 elements.");
1507  }
1508  }  // namespace
1509
1510  REGISTER_OP("Unique")
1511      .Input("x: T")
1512      .Output("y: T")
1513      .Output("idx: out_idx")
1514      .Attr("T: type")
1515      .Attr("out_idx: {int32, int64} = DT_INT32")
1516      .SetShapeFn([](InferenceContext* c) {
1517        c->set_output(0, c->Vector(InferenceContext::kUnknownDim));
1518        c->set_output(1, c->input(0));
1519        // Assert that the input rank is 1.
1520        ShapeHandle dummy;
1521        return c->WithRank(c->input(0), 1, &dummy);
1522      });
1523
1524  REGISTER_OP("UniqueV2")
1525      .Input("x: T")
1526      .Input("axis: Taxis")
1527      .Output("y: T")
1528      .Output("idx: out_idx")
1529      .Attr("T: type")
1530      .Attr("Taxis: {int32,int64} = DT_INT64")
1531      .Attr("out_idx: {int32, int64} = DT_INT32")
1532      .SetShapeFn([](InferenceContext* c) {
1533        c->set_output(0, c->UnknownShapeOfRank(c->Rank(c->input(0))));
1534        TF_RETURN_IF_ERROR(UniqueIdxShapeFn(c));
1535        return Status::OK();
1536      });
1537
1538  // ----------------------------------------------------------------------
1539  REGISTER_OP("UniqueWithCounts")
1540      .Input("x: T")
1541      .Output("y: T")
1542      .Output("idx: out_idx")
1543      .Output("count: out_idx")
1544      .Attr("T: type")
1545      .Attr("out_idx: {int32, int64} = DT_INT32")
1546      .SetShapeFn([](InferenceContext* c) {
1547        auto uniq = c->Vector(InferenceContext::kUnknownDim);
1548        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")
```

```
1598      .Attr("N: int")
1599      .Attr("T: type")
1600      .Attr("out_type: {int32, int64} = DT_INT32")
1601      .SetShapeFn(ShapeShapeFn);
1602
1603  REGISTER_OP("EnsureShape")
1604      .Input("input: T")
1605      .Output("output: T")
1606      .Attr("shape: shape")
1607      .Attr("T: type")
1608      .SetShapeFn([](InferenceContext* c) {
1609        // Merges desired shape and statically known shape of input
1610        PartialTensorShape desired_shape;
1611        TF_RETURN_IF_ERROR(c->GetAttr("shape", &desired_shape));
1612
1613        int rank = desired_shape.dims();
1614        ShapeHandle input_shape_handle;
1615        ShapeHandle desired_shape_handle;
1616        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &input_shape_handle));
1617        TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(
1618            desired_shape, &desired_shape_handle));
1619
1620        ShapeHandle merged_shape;
1621        TF_RETURN_IF_ERROR(
1622            c->Merge(desired_shape_handle, input_shape_handle, &merged_shape));
1623        c->set_output(0, merged_shape);
1624        return Status::OK();
1625      });
1626
1627  // --------------------------------------------------------------------------
1628  REGISTER_OP("ReverseSequence")
1629      .Input("input: T")
1630      .Input("seq_lengths: Tlen")
1631      .Output("output: T")
1632      .Attr("seq_dim: int")
1633      .Attr("batch_dim: int = 0")
1634      .Attr("T: type")
1635      .Attr("Tlen: {int32, int64} = DT_INT64")
1636      .SetShapeFn([](InferenceContext* c) {
1637        ShapeHandle input = c->input(0);
1638        ShapeHandle seq_lens_shape;
1639        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &seq_lens_shape));
1640
1641        int64_t seq_dim;
1642        TF_RETURN_IF_ERROR(c->GetAttr("seq_dim", &seq_dim));
1643        int64_t batch_dim;
1644        TF_RETURN_IF_ERROR(c->GetAttr("batch_dim", &batch_dim));
1645
1646        if (!c->RankKnown(input)) {
```

```
1647          return shape_inference::UnknownShape(c);
1648        }
1649
1650        // Validate batch_dim and seq_dim against input.
1651        const int32_t input_rank = c->Rank(input);
1652        if (batch_dim >= input_rank) {
1653          return errors::InvalidArgument(
1654              "batch_dim must be < input rank: ", batch_dim, " vs. ", input_rank);
1655        }
1656        if (seq_dim >= input_rank) {
1657          return errors::InvalidArgument(
1658              "seq_dim must be < input rank: ", seq_dim, " vs. ", input_rank);
1659        }
1660
1661        DimensionHandle batch_dim_dim = c->Dim(input, batch_dim);
1662        TF_RETURN_IF_ERROR(
1663            c->Merge(batch_dim_dim, c->Dim(seq_lens_shape, 0), &batch_dim_dim));
1664
1665        // Replace batch_dim of input with batch_size
1666        ShapeHandle output_shape;
1667        TF_RETURN_IF_ERROR(
1668            c->ReplaceDim(input, batch_dim, batch_dim_dim, &output_shape));
1669        c->set_output(0, output_shape);
1670        return Status::OK();
1671      });
1672
1673  // --------------------------------------------------------------------------
1674  REGISTER_OP("Rank")
1675      .Input("input: T")
1676      .Output("output: int32")
1677      .Attr("T: type")
1678      .SetShapeFn(shape_inference::ScalarShape);
1679
1680  // --------------------------------------------------------------------------
1681  REGISTER_OP("Size")
1682      .Input("input: T")
1683      .Output("output: out_type")
1684      .Attr("T: type")
1685      .Attr("out_type: {int32, int64} = DT_INT32")
1686      .SetShapeFn(shape_inference::ScalarShape);
1687
1688  // --------------------------------------------------------------------------
1689  REGISTER_OP("Slice")
1690      .Input("input: T")
1691      .Input("begin: Index")
1692      .Input("size: Index")
1693      .Output("output: T")
1694      .Attr("T: type")
1695      .Attr("Index: {int32,int64}")
```

```
1696      .SetShapeFn(shape_inference::SliceShape);
1697
1698  #ifdef INTEL_MKL
1699  REGISTER_OP("_MklSlice")
1700      .Input("input: T")
1701      .Input("begin: Index")
1702      .Input("size: Index")
1703      .Input("mkl_input: uint8")
1704      .Input("mkl_begin: uint8")
1705      .Input("mkl_size: uint8")
1706      .Output("output: T")
1707      .Output("mkl_output: uint8")
1708      .Attr("T: type")
1709      .Attr("Index: {int32,int64}")
1710      .SetShapeFn(shape_inference::SliceShape);
1711  #endif
1712
1713  REGISTER_OP("StridedSlice")
1714      .Input("input: T")
1715      .Input("begin: Index")
1716      .Input("end: Index")
1717      .Input("strides: Index")
1718      .Output("output: T")
1719      .Attr("T: type")
1720      .Attr("Index: {int32, int64}")
1721      .Attr("begin_mask: int = 0")
1722      .Attr("end_mask: int = 0")
1723      .Attr("ellipsis_mask: int = 0")
1724      .Attr("new_axis_mask: int = 0")
1725      .Attr("shrink_axis_mask: int = 0")
1726      .SetShapeFn([](InferenceContext* c) {
1727        ShapeHandle input = c->input(0);
1728        ShapeHandle begin_shape, end_shape, strides_shape;
1729        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &begin_shape));
1730        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &end_shape));
1731        TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 1, &strides_shape));
1732        TF_RETURN_IF_ERROR(c->Merge(begin_shape, end_shape, &begin_shape));
1733        TF_RETURN_IF_ERROR(c->Merge(begin_shape, strides_shape, &begin_shape));
1734        DimensionHandle sparse_dims_dim = c->Dim(begin_shape, 0);
1735
1736        const Tensor* strides_value = c->input_tensor(3);
1737        // TODO(aselle,allenl): If we had a stride_mask it would be possible to do
1738        // more shape inference here (e.g. for x[3, ::T]).
1739        if (!c->RankKnown(input) || !c->ValueKnown(sparse_dims_dim) ||
1740            strides_value == nullptr) {
1741          c->set_output(0, c->UnknownShape());
1742          return Status::OK();
1743        }
1744
```

```
1745        PartialTensorShape input_shape({});
1746        for (int i = 0; i < c->Rank(input); ++i) {
1747          auto dim = c->Dim(input, i);
1748          input_shape.AddDim(c->ValueKnown(dim) ? c->Value(dim) : -1);
1749        }
1750
1751        int32_t begin_mask, end_mask, ellipsis_mask, new_axis_mask,
1752            shrink_axis_mask;
1753        TF_RETURN_IF_ERROR(c->GetAttr("begin_mask", &begin_mask));
1754        TF_RETURN_IF_ERROR(c->GetAttr("end_mask", &end_mask));
1755        TF_RETURN_IF_ERROR(c->GetAttr("ellipsis_mask", &ellipsis_mask));
1756        TF_RETURN_IF_ERROR(c->GetAttr("new_axis_mask", &new_axis_mask));
1757        TF_RETURN_IF_ERROR(c->GetAttr("shrink_axis_mask", &shrink_axis_mask));
1758
1759        const Tensor* begin_value = c->input_tensor(1);
1760        const Tensor* end_value = c->input_tensor(2);
1761
1762        PartialTensorShape processing_shape, final_shape;
1763        bool is_identity, is_simple_slice, slice_dim0;
1764        gtl::InlinedVector<int64, 4> begin, end, strides;
1765        TF_RETURN_IF_ERROR(ValidateStridedSliceOp(
1766            begin_value, end_value, *strides_value, input_shape, begin_mask,
1767            end_mask, ellipsis_mask, new_axis_mask, shrink_axis_mask,
1768            &processing_shape, &final_shape, &is_identity, &is_simple_slice,
1769            &slice_dim0, &begin, &end, &strides));
1770
1771        ShapeHandle out;
1772        TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(final_shape, &out));
1773        c->set_output(0, out);
1774
1775        auto* shape_and_type = c->input_handle_shapes_and_types(0);
1776        if (shape_and_type) {
1777          c->set_output_handle_shapes_and_types(0, *shape_and_type);
1778        }
1779
1780        return Status::OK();
1781      });
1782
1783  REGISTER_OP("StridedSliceGrad")
1784      .Input("shape: Index")
1785      .Input("begin: Index")
1786      .Input("end: Index")
1787      .Input("strides: Index")
1788      .Input("dy: T")
1789      .Output("output: T")
1790      .Attr("T: type")
1791      .Attr("Index: {int32, int64}")
1792      .Attr("begin_mask: int = 0")
1793      .Attr("end_mask: int = 0")
```

```
1794      .Attr("ellipsis_mask: int = 0")
1795      .Attr("new_axis_mask: int = 0")
1796      .Attr("shrink_axis_mask: int = 0")
1797      .SetShapeFn([](InferenceContext* c) {
1798        ShapeHandle out;
1799        TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &out));
1800        c->set_output(0, out);
1801        return Status::OK();
1802      });
1803
1804  REGISTER_OP("StridedSliceAssign")
1805      .Input("ref: Ref(T)")
1806      .Input("begin: Index")
1807      .Input("end: Index")
1808      .Input("strides: Index")
1809      .Input("value: T")
1810      .Output("output_ref: Ref(T)")
1811      .Attr("T: type")
1812      .Attr("Index: {int32, int64}")
1813      .Attr("begin_mask: int = 0")
1814      .Attr("end_mask: int = 0")
1815      .Attr("ellipsis_mask: int = 0")
1816      .Attr("new_axis_mask: int = 0")
1817      .Attr("shrink_axis_mask: int = 0")
1818      .SetShapeFn(shape_inference::UnchangedShape);
1819  // TODO(aselle): Fix this documentation once StridedSliceAssign Supports
1820  // broadcasting.
1821  // ----------------------------------------------------------------------
1822
1823  REGISTER_OP("ResourceStridedSliceAssign")
1824      .Input("ref: resource")
1825      .Input("begin: Index")
1826      .Input("end: Index")
1827      .Input("strides: Index")
1828      .Input("value: T")
1829      .Attr("T: type")
1830      .Attr("Index: {int32, int64}")
1831      .Attr("begin_mask: int = 0")
1832      .Attr("end_mask: int = 0")
1833      .Attr("ellipsis_mask: int = 0")
1834      .Attr("new_axis_mask: int = 0")
1835      .Attr("shrink_axis_mask: int = 0")
1836      .SetShapeFn(shape_inference::NoOutputs);
1837
1838  REGISTER_OP("TensorStridedSliceUpdate")
1839      .Input("input: T")
1840      .Input("begin: Index")
1841      .Input("end: Index")
1842      .Input("strides: Index")
```

```
1843      .Input("value: T")
1844      .Output("output: T")
1845      .Attr("T: type")
1846      .Attr("Index: {int32, int64}")
1847      .Attr("begin_mask: int = 0")
1848      .Attr("end_mask: int = 0")
1849      .Attr("ellipsis_mask: int = 0")
1850      .Attr("new_axis_mask: int = 0")
1851      .Attr("shrink_axis_mask: int = 0")
1852      .SetShapeFn(shape_inference::UnchangedShape);
1853
1854  REGISTER_OP("Tile")
1855      .Input("input: T")
1856      .Input("multiples: Tmultiples")
1857      .Output("output: T")
1858      .Attr("T: type")
1859      .Attr("Tmultiples: {int32, int64} = DT_INT32")
1860      .SetShapeFn([](InferenceContext* c) {
1861        ShapeHandle input = c->input(0);
1862        // NOTE(mrry): Represent `multiples` as a `TensorShape` because (i)
1863        // it is a vector of non-negative integers, and (ii) doing so allows
1864        // us to handle partially-known multiples.
1865        ShapeHandle multiples;
1866        TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(1, &multiples));
1867        if (c->RankKnown(input)) {
1868          TF_RETURN_IF_ERROR(c->WithRank(multiples, c->Rank(input), &multiples));
1869          ShapeHandle dummy;
1870          TF_RETURN_IF_ERROR(
1871              c->Merge(c->input(1), c->Vector(c->Rank(input)), &dummy));
1872        }
1873
1874        if (!c->RankKnown(multiples)) {
1875          return shape_inference::UnknownShape(c);
1876        }
1877
1878        int32_t rank = c->Rank(multiples);
1879        TF_RETURN_IF_ERROR(c->WithRank(input, rank, &input));
1880        std::vector<DimensionHandle> dims(rank);
1881        for (int i = 0; i < rank; ++i) {
1882          TF_RETURN_IF_ERROR(
1883              c->Multiply(c->Dim(input, i), c->Dim(multiples, i), &dims[i]));
1884        }
1885        c->set_output(0, c->MakeShape(dims));
1886        return Status::OK();
1887      });
1888
1889  // ----------------------------------------------------------------------
1890  REGISTER_OP("TileGrad")
1891      .Input("input: T")
```

```
1892      .Input("multiples: int32")
1893      .Output("output: T")
1894      .Attr("T: type")
1895      .Deprecated(3, "TileGrad has been replaced with reduce_sum")
1896      .SetShapeFn(tensorflow::shape_inference::UnknownShape);
1897
1898  // --------------------------------------------------------------------------
1899  REGISTER_OP("Where")
1900      .Input("input: T")
1901      .Attr("T: {numbertype, bool} = DT_BOOL")
1902      .Output("index: int64")
1903      .SetShapeFn([](InferenceContext* c) {
1904        c->set_output(0, c->Matrix(c->UnknownDim(), c->Rank(c->input(0))));
1905        return Status::OK();
1906      });
1907
1908  // --------------------------------------------------------------------------
1909  REGISTER_OP("BroadcastArgs")
1910      .Input("s0: T")
1911      .Input("s1: T")
1912      .Output("r0: T")
1913      .Attr("T: {int32, int64} = DT_INT32")
1914      .SetShapeFn([](InferenceContext* c) {
1915        ShapeHandle unused;
1916        ShapeHandle shape_x = c->input(0);
1917        ShapeHandle shape_y = c->input(1);
1918        TF_RETURN_IF_ERROR(c->WithRank(shape_x, 1, &unused));
1919        TF_RETURN_IF_ERROR(c->WithRank(shape_y, 1, &unused));
1920
1921        if (!c->ValueKnown(c->Dim(shape_x, 0)) ||
1922            !c->ValueKnown(c->Dim(shape_y, 0))) {
1923          c->set_output(0, c->Vector(InferenceContext::kUnknownDim));
1924          return Status::OK();
1925        }
1926
1927        int64_t x_dim = c->Value(c->Dim(shape_x, 0));
1928        int64_t y_dim = c->Value(c->Dim(shape_y, 0));
1929
1930        // Broadcasted shape is going to be as large as the largest dimension.
1931        c->set_output(0, c->Vector(std::max(x_dim, y_dim)));
1932        return Status::OK();
1933      });
1934
1935  // --------------------------------------------------------------------------
1936  REGISTER_OP("BroadcastGradientArgs")
1937      .Input("s0: T")
1938      .Input("s1: T")
1939      .Output("r0: T")
1940      .Output("r1: T")
```

```
1941      .Attr("T: {int32, int64} = DT_INT32")
1942      .SetShapeFn([](InferenceContext* c) {
1943        // TODO(mrry): Implement constant_value for BroadcastGradientArgs?
1944        ShapeHandle unused;
1945        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &unused));
1946        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &unused));
1947        c->set_output(0, c->Vector(InferenceContext::kUnknownDim));
1948        c->set_output(1, c->Vector(InferenceContext::kUnknownDim));
1949        return Status::OK();
1950      });
1951
1952  // ----------------------------------------------------------------------
1953  REGISTER_OP("Pad")
1954      .Input("input: T")
1955      .Input("paddings: Tpaddings")
1956      .Output("output: T")
1957      .Attr("T: type")
1958      .Attr("Tpaddings: {int32, int64} = DT_INT32")
1959      .SetShapeFn(PadShapeFn);
1960
1961  // ----------------------------------------------------------------------
1962  REGISTER_OP("PadV2")
1963      .Input("input: T")
1964      .Input("paddings: Tpaddings")
1965      .Input("constant_values: T")
1966      .Output("output: T")
1967      .Attr("T: type")
1968      .Attr("Tpaddings: {int32, int64} = DT_INT32")
1969      .SetShapeFn(PadShapeFn);
1970
1971  // ----------------------------------------------------------------------
1972  REGISTER_OP("MirrorPad")
1973      .Input("input: T")
1974      .Input("paddings: Tpaddings")
1975      .Output("output: T")
1976      .Attr("T: type")
1977      .Attr("Tpaddings: {int32, int64} = DT_INT32")
1978      .Attr(GetMirrorPadModeAttrString())
1979      .SetShapeFn(PadShapeFn);
1980
1981  // ----------------------------------------------------------------------
1982  namespace {
1983  template <typename T>
1984  Status MirrorPadKnown(InferenceContext* c, ShapeHandle input,
1985                        const Tensor* paddings_t, int64_t input_rank) {
1986    auto paddings_data = paddings_t->matrix<T>();
1987    std::vector<DimensionHandle> dims(input_rank);
1988    for (int64_t i = 0; i < input_rank; ++i) {
1989      const int64_t pad0 = static_cast<int64_t>(paddings_data(i, 0));
```

```
1990      const int64_t pad1 = static_cast<int64_t>(paddings_data(i, 1));
1991      if (pad0 < 0 || pad1 < 0) {
1992        return errors::InvalidArgument("Paddings must be non-negative");
1993      }
1994
1995      TF_RETURN_IF_ERROR(c->Subtract(c->Dim(input, i), pad0 + pad1, &dims[i]));
1996    }
1997    c->set_output(0, c->MakeShape(dims));
1998    return Status::OK();
1999  }
2000
2001  }  // namespace
2002
2003  REGISTER_OP("MirrorPadGrad")
2004      .Input("input: T")
2005      .Input("paddings: Tpaddings")
2006      .Output("output: T")
2007      .Attr("T: type")
2008      .Attr("Tpaddings: {int32, int64} = DT_INT32")
2009      .Attr(GetMirrorPadModeAttrString())
2010      .SetShapeFn([](InferenceContext* c) {
2011        ShapeHandle paddings;
2012        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 2, &paddings));
2013        DimensionHandle pad_0 = c->Dim(paddings, 0);
2014        if (!c->ValueKnown(pad_0)) {
2015          // We don't know the rank of the output since the first
2016          // padding dimension is unknown.
2017          c->set_output(0, c->UnknownShape());
2018          return Status::OK();
2019        }
2020
2021        int64_t input_rank = c->Value(pad_0);
2022        ShapeHandle input;
2023        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), input_rank, &input));
2024        TF_RETURN_IF_ERROR(
2025            c->Merge(paddings, c->Matrix(input_rank, 2), &paddings));
2026
2027        const Tensor* paddings_t = c->input_tensor(1);
2028        if (paddings_t == nullptr) {
2029          // Values of 'paddings' is not available, but we know the
2030          // input rank, so return the rank of the output with unknown
2031          // dimensions.
2032          c->set_output(0, c->UnknownShapeOfRank(input_rank));
2033          return Status::OK();
2034        }
2035
2036        if (paddings_t->dtype() == DT_INT32) {
2037          return MirrorPadKnown<int32>(c, input, paddings_t, input_rank);
2038        } else {
```

```
2039          return MirrorPadKnown<int64_t>(c, input, paddings_t, input_rank);
2040        }
2041      });
2042
2043  // ----------------------------------------------------------------------
2044  REGISTER_OP("Placeholder")
2045      .Output("output: dtype")
2046      .Attr("dtype: type")
2047      .Attr("shape: shape = { unknown_rank: true }")
2048      .SetShapeFn([](InferenceContext* c) {
2049        PartialTensorShape shape;
2050        TF_RETURN_IF_ERROR(c->GetAttr("shape", &shape));
2051
2052        // Placeholder has legacy behavior where we cannot tell the difference
2053        // between a scalar shape attribute and 'unknown shape'.  So if the shape
2054        // is a scalar, we return an unknown shape.
2055        if (c->graph_def_version() <= 21 && shape.dims() <= 0) {
2056          return shape_inference::UnknownShape(c);
2057        }
2058
2059        ShapeHandle out;
2060        TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(shape, &out));
2061        c->set_output(0, out);
2062        return Status::OK();
2063      });
2064
2065  // Placeholder was modified in a backwards compatible way to do what
2066  // PlaceholderV2 did, so we have deprecated V2 (no one was really
2067  // using it).
2068  REGISTER_OP("PlaceholderV2")
2069      .Output("output: dtype")
2070      .Attr("dtype: type")
2071      .Attr("shape: shape")
2072      .SetShapeFn(shape_inference::ExplicitShape)
2073      .Deprecated(23, "Placeholder now behaves the same as PlaceholderV2.");
2074
2075  // ----------------------------------------------------------------------
2076  REGISTER_OP("PlaceholderWithDefault")
2077      .Input("input: dtype")
2078      .Output("output: dtype")
2079      .Attr("dtype: type")
2080      .Attr("shape: shape")
2081      .SetShapeFn([](InferenceContext* c) {
2082        ShapeHandle input = c->input(0);
2083        PartialTensorShape shape;
2084        TF_RETURN_IF_ERROR(c->GetAttr("shape", &shape));
2085        ShapeHandle out;
2086        TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(shape, &out));
2087
```

```
2088        // We merge for compatibility checking, but return the output,
2089        // since output_shape may be less precise than input_shape.
2090        ShapeHandle unused;
2091        TF_RETURN_IF_ERROR(c->Merge(input, out, &unused));
2092        c->set_output(0, out);
2093        return Status::OK();
2094      });
2095
2096  // ----------------------------------------------------------------------
2097  REGISTER_OP("ExpandDims")
2098      .Input("input: T")
2099      .Input("dim: Tdim")
2100      .Output("output: T")
2101      .Attr("T: type")
2102      .Attr("Tdim: {int32, int64} = DT_INT32")
2103      .SetShapeFn([](InferenceContext* c) {
2104        ShapeHandle input = c->input(0);
2105
2106        const Tensor* dim_t = c->input_tensor(1);
2107        if (dim_t != nullptr && dim_t->NumElements() != 1) {
2108          return errors::InvalidArgument(
2109              "'dim' input must be a tensor with a single value");
2110        }
2111        if (dim_t == nullptr || !c->RankKnown(input)) {
2112          c->set_output(0, c->UnknownShape());
2113          return Status::OK();
2114        }
2115
2116        int64_t dim;
2117        if (dim_t->dtype() == DT_INT32) {
2118          dim = static_cast<int64_t>(dim_t->flat<int32>()(0));
2119        } else {
2120          dim = dim_t->flat<int64_t>()(0);
2121        }
2122
2123        const int32_t rank = c->Rank(input);
2124        const int32_t min_dim = -1 * rank - 1;
2125        if (dim < min_dim || dim > rank) {
2126          return errors::InvalidArgument("dim ", dim, " not in the interval [",
2127                                         min_dim, ", ", rank, "].");
2128        }
2129
2130        if (dim < 0) {
2131          dim += rank + 1;
2132        }
2133
2134        ShapeHandle end;
2135        TF_RETURN_IF_ERROR(c->Subshape(input, dim, &end));
2136
```

```
2137        // Build output as start + 1 + end.
2138        ShapeHandle output;
2139        TF_RETURN_IF_ERROR(c->Subshape(input, 0, dim, &output));
2140        TF_RETURN_IF_ERROR(c->Concatenate(output, c->Vector(1), &output));
2141        TF_RETURN_IF_ERROR(c->Concatenate(output, end, &output));
2142        c->set_output(0, output);
2143        return Status::OK();
2144      });
2145
2146  // --------------------------------------------------------------------------
2147  REGISTER_OP("Squeeze")
2148      .Input("input: T")
2149      .Output("output: T")
2150      .Attr("T: type")
2151      .Attr("squeeze_dims: list(int) >= 0 = []")
2152      .SetShapeFn([](InferenceContext* c) {
2153        ShapeHandle input = c->input(0);
2154        if (!c->RankKnown(input)) {
2155          // Input shape unknown.
2156          return shape_inference::UnknownShape(c);
2157        }
2158
2159        const int32_t input_rank = c->Rank(input);
2160
2161        // Validate and wrap squeeze dimensions.
2162        std::vector<int32> squeeze_dims;
2163        TF_RETURN_IF_ERROR(c->GetAttr("squeeze_dims", &squeeze_dims));
2164        for (int i = 0; i < squeeze_dims.size(); ++i) {
2165          if (squeeze_dims[i] < -input_rank || squeeze_dims[i] >= input_rank) {
2166            return errors::InvalidArgument("squeeze_dims[", i, "] not in [",
2167                                           -input_rank, ",", input_rank, ").");
2168          }
2169
2170          if (squeeze_dims[i] < 0) {
2171            squeeze_dims[i] += input_rank;
2172          }
2173        }
2174
2175        std::vector<DimensionHandle> result_shape;
2176        for (int i = 0; i < input_rank; ++i) {
2177          // True if squeeze_dims contains an entry to squeeze this
2178          // dimension.
2179          bool is_explicit_match =
2180              std::find(squeeze_dims.begin(), squeeze_dims.end(), i) !=
2181              squeeze_dims.end();
2182
2183          DimensionHandle dim = c->Dim(input, i);
2184
2185          if (!c->ValueKnown(dim)) {
```

```
2186            // Assume that the squeezed dimension will be 1 at runtime.
2187            if (is_explicit_match) continue;
2188
2189            // If squeezing all 1 dimensions, and we see an unknown value,
2190            // give up and return Unknown Shape.
2191            if (squeeze_dims.empty()) {
2192              c->set_output(0, c->UnknownShape());
2193              return Status::OK();
2194            }
2195          } else if (c->Value(dim) == 1) {
2196            if (is_explicit_match || squeeze_dims.empty()) {
2197              // If explicitly squeezing, or squeezing all 1s, remove
2198              // this dimension.
2199              continue;
2200            }
2201          } else if (is_explicit_match) {
2202            return errors::InvalidArgument("Can not squeeze dim[", i,
2203                                           "], expected a dimension of 1, got ",
2204                                           c->Value(c->Dim(input, i)));
2205          }
2206
2207          result_shape.emplace_back(dim);
2208        }
2209
2210        c->set_output(0, c->MakeShape(result_shape));
2211        return Status::OK();
2212      });
2213
2214  // ----------------------------------------------------------------------
2215  REGISTER_OP("ListDiff")
2216      .Input("x: T")
2217      .Input("y: T")
2218      .Output("out: T")
2219      .Output("idx: out_idx")
2220      .Attr("T: type")
2221      .Attr("out_idx: {int32, int64} = DT_INT32")
2222      .SetShapeFn([](InferenceContext* c) {
2223        ShapeHandle unused;
2224        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 1, &unused));
2225        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &unused));
2226        // TODO(mrry): Indicate that the length falls within an interval?
2227        ShapeHandle out = c->Vector(InferenceContext::kUnknownDim);
2228        c->set_output(0, out);
2229        c->set_output(1, out);
2230        return Status::OK();
2231      });
2232
2233  namespace {
2234
```

```
2235  // Converts Tensor to flat std::vector<int64_t>.
2236  template <typename InputType>
2237  std::vector<int64_t> GetFlatInt64(const Tensor& t) {
2238    std::vector<int64_t> output(t.shape().num_elements());
2239    if (t.shape().num_elements() > 0) {
2240      auto eigen_vec = t.flat<InputType>();
2241      std::copy_n(&eigen_vec(0), output.size(), output.begin());
2242    }
2243    return output;
2244  }
2245
2246  // Converts int32 or int64 Tensor to flat std::vector<int64_t>.
2247  std::vector<int64_t> GetFlatInt64(const Tensor& t) {
2248    if (t.dtype() == DT_INT32) {
2249      return GetFlatInt64<int32>(t);
2250    } else {
2251      return GetFlatInt64<int64_t>(t);
2252    }
2253  }
2254
2255  Status SpaceToBatchShapeHelper(InferenceContext* c, ShapeHandle input_shape,
2256                                 ShapeHandle block_shape_shape,
2257                                 const Tensor* block_shape_t,
2258                                 ShapeHandle paddings_shape,
2259                                 const Tensor* paddings_t) {
2260    if (c->Rank(block_shape_shape) != 1) {
2261      return errors::InvalidArgument("block_shape must have rank 1.");
2262    }
2263
2264    const DimensionHandle num_block_dims_handle = c->Dim(block_shape_shape, 0);
2265    if (!c->ValueKnown(num_block_dims_handle)) {
2266      return errors::InvalidArgument("block_shape must have known size.");
2267    }
2268
2269    const int64_t num_block_dims = c->Value(num_block_dims_handle);
2270
2271    TF_RETURN_IF_ERROR(
2272        c->WithRankAtLeast(input_shape, num_block_dims + 1, &input_shape));
2273
2274    TF_RETURN_IF_ERROR(
2275        c->Merge(paddings_shape, c->Matrix(num_block_dims, 2), &paddings_shape));
2276
2277    DimensionHandle batch_size = c->Dim(input_shape, 0);
2278    std::vector<int64_t> block_shape_vec;
2279    if (block_shape_t && (block_shape_t->NumElements() > 0)) {
2280      block_shape_vec = GetFlatInt64(*block_shape_t);
2281      for (int64_t dim = 0; dim < num_block_dims; ++dim) {
2282        const int64_t block_shape_value = block_shape_vec[dim];
2283        if (block_shape_value < 1) {
```

```
2284          return errors::InvalidArgument("block_shape must be positive");
2285        }
2286        if (c->ValueKnown(batch_size)) {
2287          TF_RETURN_IF_ERROR(
2288              c->Multiply(batch_size, block_shape_value, &batch_size));
2289        } else {
2290          batch_size = c->UnknownDim();
2291        }
2292      }
2293    } else if (num_block_dims > 0) {
2294      batch_size = c->UnknownDim();
2295    }
2296
2297    std::vector<DimensionHandle> output_dims{batch_size};
2298    output_dims.resize(num_block_dims + 1, c->UnknownDim());
2299
2300    if (paddings_t && (paddings_t->NumElements() > 0)) {
2301      const std::vector<int64_t> paddings_vec = GetFlatInt64(*paddings_t);
2302      for (int64_t dim = 0; dim < num_block_dims; ++dim) {
2303        const int64_t pad_start = paddings_vec[dim * 2],
2304                      pad_end = paddings_vec[dim * 2 + 1];
2305        if (pad_start < 0 || pad_end < 0) {
2306          return errors::InvalidArgument("paddings cannot be negative");
2307        }
2308        if (block_shape_t) {
2309          DimensionHandle padded_size;
2310          TF_RETURN_IF_ERROR(
2311              c->Add(c->Dim(input_shape, dim + 1), pad_start, &padded_size));
2312          TF_RETURN_IF_ERROR(c->Add(padded_size, pad_end, &padded_size));
2313          TF_RETURN_IF_ERROR(c->Divide(padded_size, block_shape_vec[dim],
2314                                       /*evenly_divisible=*/true,
2315                                       &output_dims[dim + 1]));
2316        }
2317      }
2318    }
2319
2320    ShapeHandle remaining_input_shape;
2321    TF_RETURN_IF_ERROR(
2322        c->Subshape(input_shape, 1 + num_block_dims, &remaining_input_shape));
2323
2324    ShapeHandle result;
2325    TF_RETURN_IF_ERROR(c->Concatenate(c->MakeShape(output_dims),
2326                                      remaining_input_shape, &result));
2327    c->set_output(0, result);
2328    return Status::OK();
2329  }
2330
2331  Status BatchToSpaceShapeHelper(InferenceContext* c, ShapeHandle input_shape,
2332                                 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");
```

```
2382        }
2383        if (block_shape_t) {
2384          DimensionHandle cropped_size;
2385          TF_RETURN_IF_ERROR(c->Multiply(c->Dim(input_shape, dim + 1),
2386                                         block_shape_vec[dim], &cropped_size));
2387          TF_RETURN_IF_ERROR(
2388              c->Subtract(cropped_size, crop_start, &cropped_size));
2389          TF_RETURN_IF_ERROR(
2390              c->Subtract(cropped_size, crop_end, &output_dims[dim + 1]));
2391        }
2392      }
2393    }
2394
2395    ShapeHandle remaining_input_shape;
2396    TF_RETURN_IF_ERROR(
2397        c->Subshape(input_shape, 1 + num_block_dims, &remaining_input_shape));
2398
2399    ShapeHandle result;
2400    TF_RETURN_IF_ERROR(c->Concatenate(c->MakeShape(output_dims),
2401                                      remaining_input_shape, &result));
2402    c->set_output(0, result);
2403    return Status::OK();
2404  }
2405
2406  }  // namespace
2407
2408  // ----------------------------------------------------------------------
2409  REGISTER_OP("SpaceToBatchND")
2410      .Input("input: T")
2411      .Input("block_shape: Tblock_shape")
2412      .Input("paddings: Tpaddings")
2413      .Output("output: T")
2414      .Attr("T: type")
2415      .Attr("Tblock_shape: {int32, int64} = DT_INT32")
2416      .Attr("Tpaddings: {int32, int64} = DT_INT32")
2417      .SetShapeFn([](InferenceContext* c) {
2418        return SpaceToBatchShapeHelper(c, c->input(0), c->input(1),
2419                                       c->input_tensor(1), c->input(2),
2420                                       c->input_tensor(2));
2421      });
2422
2423  // ----------------------------------------------------------------------
2424  REGISTER_OP("SpaceToBatch")
2425      .Input("input: T")
2426      .Input("paddings: Tpaddings")
2427      .Output("output: T")
2428      .Attr("T: type")
2429      .Attr("Tpaddings: {int32, int64} = DT_INT32")
2430      .Attr("block_size: int >= 2")
```

```
2431      .SetShapeFn([](InferenceContext* c) {
2432        ShapeHandle input_shape;
2433        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));
2434
2435        int32_t block_size;
2436        TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));
2437
2438        Tensor block_shape(tensorflow::DT_INT64, TensorShape({2}));
2439        auto block_shape_vec = block_shape.vec<int64_t>();
2440        block_shape_vec(0) = block_size;
2441        block_shape_vec(1) = block_size;
2442
2443        return SpaceToBatchShapeHelper(c, input_shape, c->MakeShape({2}),
2444                                       &block_shape, c->input(1),
2445                                       c->input_tensor(1));
2446      });
2447
2448  // --------------------------------------------------------------------------
2449  REGISTER_OP("BatchToSpaceND")
2450      .Input("input: T")
2451      .Input("block_shape: Tblock_shape")
2452      .Input("crops: Tcrops")
2453      .Output("output: T")
2454      .Attr("T: type")
2455      .Attr("Tblock_shape: {int32, int64} = DT_INT32")
2456      .Attr("Tcrops: {int32, int64} = DT_INT32")
2457      .SetShapeFn([](InferenceContext* c) {
2458        return BatchToSpaceShapeHelper(c, c->input(0), c->input(1),
2459                                       c->input_tensor(1), c->input(2),
2460                                       c->input_tensor(2));
2461      });
2462
2463  // --------------------------------------------------------------------------
2464  REGISTER_OP("BatchToSpace")
2465      .Input("input: T")
2466      .Input("crops: Tidx")
2467      .Output("output: T")
2468      .Attr("T: type")
2469      .Attr("block_size: int >= 2")
2470      .Attr("Tidx: {int32, int64} = DT_INT32")
2471      .SetShapeFn([](InferenceContext* c) {
2472        ShapeHandle input_shape;
2473        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));
2474
2475        int32_t block_size;
2476        TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));
2477
2478        Tensor block_shape(tensorflow::DT_INT64, TensorShape({2}));
2479        auto block_shape_vec = block_shape.vec<int64_t>();
```

```
2480        block_shape_vec(0) = block_size;
2481        block_shape_vec(1) = block_size;
2482
2483        return BatchToSpaceShapeHelper(c, input_shape, c->MakeShape({2}),
2484                                       &block_shape, c->input(1),
2485                                       c->input_tensor(1));
2486      });
2487
2488  // ----------------------------------------------------------------------
2489  REGISTER_OP("SpaceToDepth")
2490      .Input("input: T")
2491      .Output("output: T")
2492      .Attr("T: type")
2493      .Attr("block_size: int >= 2")
2494      .Attr("data_format: {'NHWC', 'NCHW', 'NCHW_VECT_C'} = 'NHWC'")
2495      // TODO(pauldonnelly): Implement GPU kernels for NCHW_VECT_C.
2496      .SetShapeFn([](InferenceContext* c) {
2497        string data_format_str;
2498        TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
2499        TensorFormat data_format;
2500        FormatFromString(data_format_str, &data_format);
2501
2502        constexpr int num_spatial_dims = 2;
2503        const int dims =
2504            GetTensorDimsFromSpatialDims(num_spatial_dims, data_format);
2505        ShapeHandle input;
2506        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), dims, &input));
2507
2508        int32_t block_size;
2509        TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));
2510
2511        DimensionHandle batch_size =
2512            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'N'));
2513        DimensionHandle input_height =
2514            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'H'));
2515        DimensionHandle input_width =
2516            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'W'));
2517        DimensionHandle input_depth =
2518            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'C'));
2519
2520        DimensionHandle output_height;
2521        DimensionHandle output_width;
2522        DimensionHandle output_depth;
2523        // Will return an error if input height or width are not evenly divisible.
2524        TF_RETURN_IF_ERROR(c->Divide(input_height, block_size,
2525                                     true /* evenly_divisible */,
2526                                     &output_height));
2527        TF_RETURN_IF_ERROR(c->Divide(input_width, block_size,
2528                                     true /* evenly_divisible */, &output_width));
```

```
2529
2530        TF_RETURN_IF_ERROR(
2531            c->Multiply(input_depth, block_size * block_size, &output_depth));
2532
2533        ShapeHandle output_shape;
2534        TF_RETURN_IF_ERROR(MakeShapeFromFormat(data_format, batch_size,
2535                                               {output_height, output_width},
2536                                               output_depth, &output_shape, c));
2537
2538        c->set_output(0, output_shape);
2539        return Status::OK();
2540      });
2541
2542  // --------------------------------------------------------------------------
2543  REGISTER_OP("DepthToSpace")
2544      .Input("input: T")
2545      .Output("output: T")
2546      .Attr("T: type")
2547      .Attr("block_size: int >= 2")
2548      .Attr("data_format: {'NHWC', 'NCHW', 'NCHW_VECT_C'} = 'NHWC'")
2549      // TODO(pauldonnelly): Implement GPU kernels for NCHW and NCHW_VECT_C.
2550      .SetShapeFn([](InferenceContext* c) {
2551        string data_format_str;
2552        TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
2553        TensorFormat data_format;
2554        FormatFromString(data_format_str, &data_format);
2555
2556        constexpr int num_spatial_dims = 2;
2557        const int dims =
2558            GetTensorDimsFromSpatialDims(num_spatial_dims, data_format);
2559
2560        ShapeHandle input;
2561        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), dims, &input));
2562
2563        int32_t block_size;
2564        TF_RETURN_IF_ERROR(c->GetAttr("block_size", &block_size));
2565
2566        DimensionHandle batch_size =
2567            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'N'));
2568        DimensionHandle input_height =
2569            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'H'));
2570        DimensionHandle input_width =
2571            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'W'));
2572        DimensionHandle input_depth =
2573            c->Dim(input, GetTensorDimIndex<num_spatial_dims>(data_format, 'C'));
2574
2575        DimensionHandle output_height;
2576        DimensionHandle output_width;
2577        DimensionHandle output_depth;
```

```
2578        TF_RETURN_IF_ERROR(c->Multiply(input_height, block_size, &output_height));
2579        TF_RETURN_IF_ERROR(c->Multiply(input_width, block_size, &output_width));
2580
2581        // Will return an error if input_depth is not evenly divisible.
2582        TF_RETURN_IF_ERROR(c->Divide(input_depth, block_size * block_size,
2583                                     true /* evenly_divisible */, &output_depth));
2584
2585        ShapeHandle output_shape;
2586        TF_RETURN_IF_ERROR(MakeShapeFromFormat(data_format, batch_size,
2587                                               {output_height, output_width},
2588                                               output_depth, &output_shape, c));
2589
2590        c->set_output(0, output_shape);
2591        return Status::OK();
2592      });
2593
2594  // ----------------------------------------------------------------------
2595
2596  REGISTER_OP("ExtractImagePatches")
2597      .Input("images: T")
2598      .Output("patches: T")
2599      .Attr("ksizes: list(int) >= 4")
2600      .Attr("strides: list(int) >= 4")
2601      .Attr("rates: list(int) >= 4")
2602      .Attr(
2603          "T: {bfloat16, half, float, double, int8, int16, int32, int64, "
2604          "uint8, uint16, uint32, uint64, complex64, complex128, bool}")
2605      .Attr(GetPaddingAttrString())
2606      .SetShapeFn([](InferenceContext* c) {
2607        ShapeHandle input_shape;
2608        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));
2609
2610        std::vector<int32> ksizes;
2611        TF_RETURN_IF_ERROR(c->GetAttr("ksizes", &ksizes));
2612        if (ksizes.size() != 4) {
2613          return errors::InvalidArgument(
2614              "ExtractImagePatches requires the ksizes attribute to contain 4 "
2615              "values, but got: ",
2616              ksizes.size());
2617        }
2618
2619        std::vector<int32> strides;
2620        TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
2621        if (strides.size() != 4) {
2622          return errors::InvalidArgument(
2623              "ExtractImagePatches requires the stride attribute to contain 4 "
2624              "values, but got: ",
2625              strides.size());
2626        }
```

```
2627
2628        std::vector<int32> rates;
2629        TF_RETURN_IF_ERROR(c->GetAttr("rates", &rates));
2630        if (rates.size() != 4) {
2631          return errors::InvalidArgument(
2632              "ExtractImagePatches requires the rates attribute to contain 4 "
2633              "values, but got: ",
2634              rates.size());
2635        }
2636
2637        int32_t ksize_rows = ksizes[1];
2638        int32_t ksize_cols = ksizes[2];
2639
2640        int32_t stride_rows = strides[1];
2641        int32_t stride_cols = strides[2];
2642
2643        int32_t rate_rows = rates[1];
2644        int32_t rate_cols = rates[2];
2645
2646        int32_t ksize_rows_eff = ksize_rows + (ksize_rows - 1) * (rate_rows - 1);
2647        int32_t ksize_cols_eff = ksize_cols + (ksize_cols - 1) * (rate_cols - 1);
2648
2649        DimensionHandle batch_size_dim = c->Dim(input_shape, 0);
2650        DimensionHandle in_rows_dim = c->Dim(input_shape, 1);
2651        DimensionHandle in_cols_dim = c->Dim(input_shape, 2);
2652        DimensionHandle output_depth_dim;
2653        TF_RETURN_IF_ERROR(c->Multiply(
2654            c->Dim(input_shape, 3), ksize_rows * ksize_cols, &output_depth_dim));
2655
2656        if (!c->ValueKnown(in_rows_dim) || !c->ValueKnown(in_cols_dim)) {
2657          ShapeHandle output_shape =
2658              c->MakeShape({batch_size_dim, InferenceContext::kUnknownDim,
2659                            InferenceContext::kUnknownDim, output_depth_dim});
2660          c->set_output(0, output_shape);
2661          return Status::OK();
2662        }
2663        auto in_rows = c->Value(in_rows_dim);
2664        auto in_cols = c->Value(in_cols_dim);
2665
2666        Padding padding;
2667        TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
2668
2669        int64_t output_rows, output_cols;
2670        int64_t padding_before, padding_after;
2671        TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
2672            in_rows, ksize_rows_eff, stride_rows, padding, &output_rows,
2673            &padding_before, &padding_after));
2674        TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
2675            in_cols, ksize_cols_eff, stride_cols, padding, &output_cols,
```

```
2676            &padding_before, &padding_after));
2677        ShapeHandle output_shape = c->MakeShape(
2678            {batch_size_dim, output_rows, output_cols, output_depth_dim});
2679        c->set_output(0, output_shape);
2680        return Status::OK();
2681      });
2682
2683  // ----------------------------------------------------------------------
2684
2685  // To enable rates, uncomment all lines commented below and use ksize_*_eff
2686  // as the second parameter of all GetWindowedOutputSizeVerbose calls instead
2687  // of ksize_*.
2688  REGISTER_OP("ExtractVolumePatches")
2689      .Input("input: T")
2690      .Output("patches: T")
2691      .Attr("ksizes: list(int) >= 5")
2692      .Attr("strides: list(int) >= 5")
2693      /* .Attr("rates: list(int) >= 5") */
2694      .Attr("T: realnumbertype")
2695      .Attr(GetPaddingAttrString())
2696      .SetShapeFn([](InferenceContext* c) {
2697        ShapeHandle input_shape;
2698        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 5, &input_shape));
2699
2700        std::vector<int32> ksizes;
2701        TF_RETURN_IF_ERROR(c->GetAttr("ksizes", &ksizes));
2702        if (ksizes.size() != 5) {
2703          return errors::InvalidArgument(
2704              "ExtractVolumePatches requires the ksizes attribute to contain 5 "
2705              "values, but got: ",
2706              ksizes.size());
2707        }
2708
2709        std::vector<int32> strides;
2710        TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
2711        if (strides.size() != 5) {
2712          return errors::InvalidArgument(
2713              "ExtractVolumePatches requires the stride attribute to contain 5 "
2714              "values, but got: ",
2715              strides.size());
2716        }
2717
2718        /*
2719        // TODO(hsgkim): Enable rates.
2720        // See extract_volume_patches_op.cc for why rates are disabled now.
2721
2722        std::vector<int32> rates;
2723        TF_RETURN_IF_ERROR(c->GetAttr("rates", &rates));
2724        if (rates.size() != 5) {
```

```
2725          return errors::InvalidArgument(
2726              "ExtractVolumePatches requires the rates attribute to contain 5 "
2727              "values, but got: ",
2728              rates.size());
2729        }
2730        */
2731
2732        int32_t ksize_planes = ksizes[1];
2733        int32_t ksize_rows = ksizes[2];
2734        int32_t ksize_cols = ksizes[3];
2735
2736        int32_t stride_planes = strides[1];
2737        int32_t stride_rows = strides[2];
2738        int32_t stride_cols = strides[3];
2739
2740        /*
2741        int32 rate_planes = rates[1];
2742        int32 rate_rows = rates[2];
2743        int32 rate_cols = rates[3];
2744
2745        int32 ksize_planes_eff = ksize_planes +
2746                                 (ksize_planes - 1) * (rate_planes - 1);
2747        int32 ksize_rows_eff = ksize_rows + (ksize_rows - 1) * (rate_rows - 1);
2748        int32 ksize_cols_eff = ksize_cols + (ksize_cols - 1) * (rate_cols - 1);
2749        */
2750
2751        DimensionHandle batch_size_dim = c->Dim(input_shape, 0);
2752        DimensionHandle in_planes_dim = c->Dim(input_shape, 1);
2753        DimensionHandle in_rows_dim = c->Dim(input_shape, 2);
2754        DimensionHandle in_cols_dim = c->Dim(input_shape, 3);
2755        DimensionHandle output_depth_dim;
2756        TF_RETURN_IF_ERROR(c->Multiply(c->Dim(input_shape, 4),
2757                                       ksize_planes * ksize_rows * ksize_cols,
2758                                       &output_depth_dim));
2759
2760        if (!c->ValueKnown(in_planes_dim) || !c->ValueKnown(in_rows_dim) ||
2761            !c->ValueKnown(in_cols_dim)) {
2762          ShapeHandle output_shape =
2763              c->MakeShape({batch_size_dim, InferenceContext::kUnknownDim,
2764                            InferenceContext::kUnknownDim, output_depth_dim});
2765          c->set_output(0, output_shape);
2766          return Status::OK();
2767        }
2768        auto in_planes = c->Value(in_planes_dim);
2769        auto in_rows = c->Value(in_rows_dim);
2770        auto in_cols = c->Value(in_cols_dim);
2771
2772        Padding padding;
2773        TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
```

```
2774
2775        int64_t output_planes, output_rows, output_cols;
2776        int64_t padding_before, padding_after;
2777        TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
2778            in_planes, ksize_planes, stride_planes, padding, &output_planes,
2779            &padding_before, &padding_after));
2780        TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
2781            in_rows, ksize_rows, stride_rows, padding, &output_rows,
2782            &padding_before, &padding_after));
2783        TF_RETURN_IF_ERROR(GetWindowedOutputSizeVerbose(
2784            in_cols, ksize_cols, stride_cols, padding, &output_cols,
2785            &padding_before, &padding_after));
2786        ShapeHandle output_shape =
2787            c->MakeShape({batch_size_dim, output_planes, output_rows, output_cols,
2788                          output_depth_dim});
2789        c->set_output(0, output_shape);
2790        return Status::OK();
2791      });
2792
2793  // ----------------------------------------------------------------------
2794
2795  REGISTER_OP("OneHot")
2796      .Input("indices: TI")
2797      .Input("depth: int32")
2798      .Input("on_value: T")
2799      .Input("off_value: T")
2800      .Attr("axis: int = -1")
2801      .Output("output: T")
2802      .Attr("T: type")
2803      .Attr("TI: {uint8, int32, int64} = DT_INT64")
2804      .SetShapeFn([](InferenceContext* c) {
2805        int32_t axis;
2806        TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
2807        if (axis < -1) return errors::InvalidArgument("axis must be >= -1");
2808
2809        DimensionHandle depth;
2810        TF_RETURN_IF_ERROR(c->MakeDimForScalarInput(1, &depth));
2811
2812        ShapeHandle indices = c->input(0);
2813        if (!c->RankKnown(indices)) return shape_inference::UnknownShape(c);
2814
2815        int32_t new_rank = c->Rank(indices) + 1;
2816        // We need to add new_rank to axis in the case the axis is -1 because
2817        // C++ returns negative values from % if the dividend is negative.
2818        int32_t depth_index = (axis + new_rank) % new_rank;
2819        // Out shape is indices[0:depth_index] + [depth] + indices[depth_index:].
2820        ShapeHandle front;
2821        ShapeHandle back;
2822        ShapeHandle out;
```

```
2823        TF_RETURN_IF_ERROR(c->Subshape(indices, 0, depth_index, &front));
2824        TF_RETURN_IF_ERROR(c->Subshape(indices, depth_index, &back));
2825        TF_RETURN_IF_ERROR(c->Concatenate(front, c->Vector(depth), &front));
2826        TF_RETURN_IF_ERROR(c->Concatenate(front, back, &out));
2827        c->set_output(0, out);
2828        return Status::OK();
2829      });
2830
2831  // EXPERIMENTAL. DO NOT USE OR DEPEND ON THIS YET.
2832  REGISTER_OP("QuantizeAndDequantize")
2833      .Input("input: T")
2834      .Attr("signed_input: bool = true")
2835      .Attr("num_bits: int = 8")
2836      .Attr("range_given: bool = false")
2837      .Attr("input_min: float = 0")
2838      .Attr("input_max: float = 0")
2839      .Output("output: T")
2840      .Attr("T: {bfloat16, half, float, double}")
2841      .SetShapeFn(shape_inference::UnchangedShape)
2842      .Deprecated(22, "Replaced by QuantizeAndDequantizeV2");
2843
2844  // TODO(suharshs): Deprecate QuantizeAndDequantizeV2.
2845  REGISTER_OP("QuantizeAndDequantizeV2")
2846      .Input("input: T")
2847      .Input("input_min: T")
2848      .Input("input_max: T")
2849      .Attr("signed_input: bool = true")
2850      .Attr("num_bits: int = 8")
2851      .Attr("range_given: bool = false")
2852      .Output("output: T")
2853      .Attr("T: {bfloat16, half, float, double}")
2854      .Attr(
2855          "round_mode: {'HALF_TO_EVEN', 'HALF_UP'} = "
2856          "'HALF_TO_EVEN'")
2857      .Attr("narrow_range: bool = false")
2858      .Attr("axis: int = -1")
2859      .SetShapeFn([](InferenceContext* c) {
2860        int axis;
2861        TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
2862        const int minmax_rank = (axis == -1) ? 0 : 1;
2863        ShapeHandle minmax;
2864        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
2865        TF_RETURN_IF_ERROR(c->Merge(c->input(2), minmax, &minmax));
2866        if (axis < -1) {
2867          return errors::InvalidArgument("axis should be at least -1, got ",
2868                                         axis);
2869        } else if (axis != -1) {
2870          ShapeHandle input;
2871          TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
```

```
2872          DimensionHandle depth;
2873          TF_RETURN_IF_ERROR(
2874              c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
2875        }
2876        c->set_output(0, c->input(0));
2877        return Status::OK();
2878      });
2879
2880  REGISTER_OP("QuantizeAndDequantizeV4")
2881      .Input("input: T")
2882      .Input("input_min: T")
2883      .Input("input_max: T")
2884      .Attr("signed_input: bool = true")
2885      .Attr("num_bits: int = 8")
2886      .Attr("range_given: bool = false")
2887      .Output("output: T")
2888      .Attr("T: {bfloat16, half, float, double}")
2889      .Attr(
2890          "round_mode: {'HALF_TO_EVEN', 'HALF_UP'} = "
2891          "'HALF_TO_EVEN'")
2892      .Attr("narrow_range: bool = false")
2893      .Attr("axis: int = -1")
2894      .SetShapeFn([](InferenceContext* c) {
2895        int axis;
2896        TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
2897        const int minmax_rank = (axis == -1) ? 0 : 1;
2898        ShapeHandle minmax;
2899        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
2900        TF_RETURN_IF_ERROR(c->Merge(c->input(2), minmax, &minmax));
2901        if (axis < -1) {
2902          return errors::InvalidArgument("axis should be at least -1, got ",
2903                                         axis);
2904        } else if (axis != -1) {
2905          ShapeHandle input;
2906          TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
2907          DimensionHandle depth;
2908          TF_RETURN_IF_ERROR(
2909              c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
2910        }
2911        c->set_output(0, c->input(0));
2912        return Status::OK();
2913      });
2914
2915  REGISTER_OP("QuantizeAndDequantizeV4Grad")
2916      .Input("gradients: T")
2917      .Input("input: T")
2918      .Input("input_min: T")
2919      .Input("input_max: T")
2920      .Output("input_backprop: T")
```

```
2921      .Output("input_min_backprop: T")
2922      .Output("input_max_backprop: T")
2923      .Attr("T: {bfloat16, half, float, double}")
2924      .Attr("axis: int = -1")
2925      .SetShapeFn([](InferenceContext* c) {
2926        int axis;
2927        TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
2928        const int minmax_rank = (axis == -1) ? 0 : 1;
2929        ShapeHandle minmax;
2930        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), minmax_rank, &minmax));
2931        TF_RETURN_IF_ERROR(c->Merge(c->input(3), minmax, &minmax));
2932        if (axis < -1) {
2933          return errors::InvalidArgument("axis should be at least -1, got ",
2934                                         axis);
2935        } else if (axis != -1) {
2936          ShapeHandle input;
2937          TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
2938          DimensionHandle depth;
2939          TF_RETURN_IF_ERROR(
2940              c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
2941        }
2942        ShapeHandle inputs;
2943        TF_RETURN_IF_ERROR(c->Merge(c->input(0), c->input(1), &inputs));
2944        c->set_output(0, inputs);
2945        c->set_output(1, minmax);
2946        c->set_output(2, minmax);
2947        return Status::OK();
2948      });
2949
2950  REGISTER_OP("QuantizeAndDequantizeV3")
2951      .Input("input: T")
2952      .Input("input_min: T")
2953      .Input("input_max: T")
2954      .Input("num_bits: int32")
2955      .Attr("signed_input: bool = true")
2956      .Attr("range_given: bool = true")
2957      .Output("output: T")
2958      .Attr("T: {bfloat16, half, float, double}")
2959      .Attr("narrow_range: bool = false")
2960      .Attr("axis: int = -1")
2961      .SetShapeFn([](InferenceContext* c) {
2962        int axis;
2963        TF_RETURN_IF_ERROR(c->GetAttr("axis", &axis));
2964        const int minmax_rank = (axis == -1) ? 0 : 1;
2965        ShapeHandle minmax;
2966        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
2967        TF_RETURN_IF_ERROR(c->Merge(c->input(2), minmax, &minmax));
2968        if (axis < -1) {
2969          return errors::InvalidArgument("axis should be at least -1, got ",
```

```
2970                                        axis);
2971       } else if (axis != -1) {
2972         ShapeHandle input;
2973         TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
2974         DimensionHandle depth;
2975         TF_RETURN_IF_ERROR(
2976             c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
2977       }
2978       ShapeHandle unused;
2979       TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 0, &unused));
2980       c->set_output(0, c->input(0));
2981       return Status::OK();
2982     });
2983
2984 REGISTER_OP("QuantizeV2")
2985     .Input("input: float")
2986     .Input("min_range: float")
2987     .Input("max_range: float")
2988     .Output("output: T")
2989     .Output("output_min: float")
2990     .Output("output_max: float")
2991     .Attr("T: quantizedtype")
2992     .Attr("mode: {'MIN_COMBINED', 'MIN_FIRST', 'SCALED'} = 'MIN_COMBINED'")
2993     .Attr(
2994         "round_mode: {'HALF_AWAY_FROM_ZERO', 'HALF_TO_EVEN'} = "
2995         "'HALF_AWAY_FROM_ZERO'")
2996     .Attr("narrow_range: bool = false")
2997     .Attr("axis: int = -1")
•••  .Attr("ensure_minimum_range: float = 0.01")
2999     .SetShapeFn(shape_inference::QuantizeV2Shape);
3000
3001 REGISTER_OP("Dequantize")
3002     .Input("input: T")
3003     .Input("min_range: float")
3004     .Input("max_range: float")
3005     .Output("output: dtype")
3006     .Attr("T: quantizedtype")
3007     .Attr("mode: {'MIN_COMBINED', 'MIN_FIRST', 'SCALED'} = 'MIN_COMBINED'")
3008     .Attr("narrow_range: bool = false")
3009     .Attr("axis: int = -1")
3010     .Attr("dtype: {bfloat16, float} = DT_FLOAT")
3011     .SetShapeFn([](InferenceContext* c) {
3012       int axis = -1;
3013       Status s = c->GetAttr("axis", &axis);
3014       if (!s.ok() && s.code() != error::NOT_FOUND) {
3015         return s;
3016       }
3017       if (axis < -1) {
3018         return errors::InvalidArgument("axis should be at least -1, got ",
```

```
3019                                        axis);
3020       }
3021       const int minmax_rank = (axis == -1) ? 0 : 1;
3022       TF_RETURN_IF_ERROR(shape_inference::UnchangedShape(c));
3023       ShapeHandle minmax;
3024       TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
3025       TF_RETURN_IF_ERROR(c->WithRank(c->input(2), minmax_rank, &minmax));
3026       if (axis != -1) {
3027         ShapeHandle input;
3028         TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
3029         DimensionHandle depth;
3030         TF_RETURN_IF_ERROR(
3031             c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
3032       }
3033       return Status::OK();
3034     });
3035
3036 REGISTER_OP("QuantizedConcat")
3037     .Input("concat_dim: int32")
3038     .Input("values: N * T")
3039     .Input("input_mins: N * float32")
3040     .Input("input_maxes: N * float32")
3041     .Output("output: T")
3042     .Output("output_min: float")
3043     .Output("output_max: float")
3044     .Attr("N: int >= 2")
3045     .Attr("T: type")
3046     .SetShapeFn([](InferenceContext* c) {
3047       const int n = (c->num_inputs() - 1) / 3;
3048       TF_RETURN_IF_ERROR(shape_inference::ConcatShape(c, n));
3049       ShapeHandle unused;
3050       for (int i = n + 1; i < c->num_inputs(); ++i) {
3051         TF_RETURN_IF_ERROR(c->WithRank(c->input(i), 0, &unused));
3052       }
3053       c->set_output(1, c->Scalar());
3054       c->set_output(2, c->Scalar());
3055       return Status::OK();
3056     });
3057
3058 REGISTER_OP("QuantizedReshape")
3059     .Input("tensor: T")
3060     .Input("shape: Tshape")
3061     .Input("input_min: float")
3062     .Input("input_max: float")
3063     .Output("output: T")
3064     .Output("output_min: float")
3065     .Output("output_max: float")
3066     .Attr("T: type")
3067     .Attr("Tshape: {int32, int64} = DT_INT32")
```

```
3068      .SetShapeFn([](InferenceContext* c) {
3069        TF_RETURN_IF_ERROR(SetOutputShapeForReshape(c));
3070        ShapeHandle unused;
3071        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
3072        TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 0, &unused));
3073        c->set_output(1, c->Scalar());
3074        c->set_output(2, c->Scalar());
3075        return Status::OK();
3076      });
3077
3078  REGISTER_OP("QuantizedInstanceNorm")
3079      .Input("x: T")
3080      .Input("x_min: float")
3081      .Input("x_max: float")
3082      .Output("y: T")
3083      .Output("y_min: float")
3084      .Output("y_max: float")
3085      .Attr("T: quantizedtype")
3086      .Attr("output_range_given: bool = false")
3087      .Attr("given_y_min: float = 0")
3088      .Attr("given_y_max: float = 0")
3089      .Attr("variance_epsilon: float = 1e-5")
3090      .Attr("min_separation: float = 1e-3")
3091      .SetShapeFn([](shape_inference::InferenceContext* c) {
3092        shape_inference::ShapeHandle unused;
3093        // x should be a rank 4 tensor.
3094        TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &unused));
3095        // Assert x_min and x_max are scalars (rank 0).
3096        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));
3097        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
3098        // y has the same shape as x.
3099        TF_RETURN_IF_ERROR(shape_inference::UnchangedShape(c));
3100        // y_min and y_max are scalars.
3101        c->set_output(1, c->Scalar());
3102        c->set_output(2, c->Scalar());
3103        return Status::OK();
3104      });
3105
3106  namespace {
3107
3108  Status ScatterNdTensorShape(InferenceContext* c) {
3109    ShapeHandle output_shape;
3110    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &output_shape));
3111    ShapeHandle indices_shape;
3112    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 1, &indices_shape));
3113    ShapeHandle updates_shape;
3114    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(2), 1, &updates_shape));
3115    return shape_inference::ScatterNdShapeHelper(c, indices_shape, updates_shape,
3116                                                 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);
    });
```

```
3166
3167 REGISTER_OP("TensorScatterUpdate")
3168     .Input("tensor: T")
3169     .Input("indices: Tindices")
3170     .Input("updates: T")
3171     .Output("output: T")
3172     .Attr("T: type")
3173     .Attr("Tindices: {int32, int64}")
3174     .SetShapeFn(ScatterNdTensorShape);
3175
3176 REGISTER_OP("TensorScatterAdd")
3177     .Input("tensor: T")
3178     .Input("indices: Tindices")
3179     .Input("updates: T")
3180     .Output("output: T")
3181     .Attr("T: type")
3182     .Attr("Tindices: {int32, int64}")
3183     .SetShapeFn(ScatterNdTensorShape);
3184
3185 REGISTER_OP("TensorScatterSub")
3186     .Input("tensor: T")
3187     .Input("indices: Tindices")
3188     .Input("updates: T")
3189     .Output("output: T")
3190     .Attr("T: type")
3191     .Attr("Tindices: {int32, int64}")
3192     .SetShapeFn(ScatterNdTensorShape);
3193
3194 REGISTER_OP("TensorScatterMin")
3195     .Input("tensor: T")
3196     .Input("indices: Tindices")
3197     .Input("updates: T")
3198     .Output("output: T")
3199     .Attr("T: type")
3200     .Attr("Tindices: {int32, int64}")
3201     .SetShapeFn(ScatterNdTensorShape);
3202
3203 REGISTER_OP("TensorScatterMax")
3204     .Input("tensor: T")
3205     .Input("indices: Tindices")
3206     .Input("updates: T")
3207     .Output("output: T")
3208     .Attr("T: type")
3209     .Attr("Tindices: {int32, int64}")
3210     .SetShapeFn(ScatterNdTensorShape);
3211
3212 REGISTER_OP("ScatterNdNonAliasingAdd")
3213     .Input("input: T")
3214     .Input("indices: Tindices")
```

```
3215      .Input("updates: T")
3216      .Output("output: T")
3217      .Attr("T: {numbertype, bool}")
3218      .Attr("Tindices: {int32, int64}")
3219      .SetShapeFn(ScatterNdTensorShape);
3220
3221  REGISTER_OP("FakeQuantWithMinMaxArgs")
3222      .Attr("min: float = -6.0")
3223      .Attr("max: float = 6.0")
3224      .Attr("num_bits: int = 8")
3225      .Attr("narrow_range: bool = false")
3226      .Input("inputs: float")
3227      .Output("outputs: float")
3228      .SetShapeFn(shape_inference::UnchangedShape);
3229
3230  REGISTER_OP("FakeQuantWithMinMaxArgsGradient")
3231      .Attr("min: float = -6.0")
3232      .Attr("max: float = 6.0")
3233      .Attr("num_bits: int = 8")
3234      .Attr("narrow_range: bool = false")
3235      .Input("gradients: float")
3236      .Input("inputs: float")
3237      .Output("backprops: float")
3238      .SetShapeFn(shape_inference::UnchangedShape);
3239
3240  REGISTER_OP("FakeQuantWithMinMaxVars")
3241      .Attr("num_bits: int = 8")
3242      .Attr("narrow_range: bool = false")
3243      .Input("inputs: float")
3244      .Input("min: float")
3245      .Input("max: float")
3246      .Output("outputs: float")
3247      .SetShapeFn([](InferenceContext* c) {
3248        TF_RETURN_IF_ERROR(shape_inference::UnchangedShape(c));
3249        ShapeHandle unused;
3250        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));
3251        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
3252        return Status::OK();
3253      });
3254
3255  REGISTER_OP("FakeQuantWithMinMaxVarsGradient")
3256      .Attr("num_bits: int = 8")
3257      .Attr("narrow_range: bool = false")
3258      .Input("gradients: float")
3259      .Input("inputs: float")
3260      .Input("min: float")
3261      .Input("max: float")
3262      .Output("backprops_wrt_input: float")
3263      .Output("backprop_wrt_min: float")
```

```
3264      .Output("backprop_wrt_max: float")
3265      .SetShapeFn([](InferenceContext* c) {
3266        // gradients and inputs are same size.
3267        ShapeHandle inputs;
3268        TF_RETURN_IF_ERROR(c->Merge(c->input(0), c->input(1), &inputs));
3269
3270        // min and max are scalars
3271        ShapeHandle min_max;
3272        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &min_max));
3273        TF_RETURN_IF_ERROR(c->Merge(min_max, c->input(3), &min_max));
3274
3275        c->set_output(0, inputs);
3276        c->set_output(1, min_max);
3277        c->set_output(2, min_max);
3278        return Status::OK();
3279      });
3280
3281  REGISTER_OP("FakeQuantWithMinMaxVarsPerChannel")
3282      .Attr("num_bits: int = 8")
3283      .Attr("narrow_range: bool = false")
3284      .Input("inputs: float")
3285      .Input("min: float")
3286      .Input("max: float")
3287      .Output("outputs: float")
3288      .SetShapeFn([](InferenceContext* c) {
3289        ShapeHandle input, min, max;
3290        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &input));
3291        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &min));
3292        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &max));
3293
3294        DimensionHandle unused;
3295        TF_RETURN_IF_ERROR(c->Merge(c->Dim(input, -1), c->Dim(min, 0), &unused));
3296        TF_RETURN_IF_ERROR(c->Merge(c->Dim(input, -1), c->Dim(max, 0), &unused));
3297        TF_RETURN_IF_ERROR(c->Merge(c->Dim(min, 0), c->Dim(max, 0), &unused));
3298
3299        c->set_output(0, input);
3300        return Status::OK();
3301      });
3302
3303  REGISTER_OP("FakeQuantWithMinMaxVarsPerChannelGradient")
3304      .Attr("num_bits: int = 8")
3305      .Attr("narrow_range: bool = false")
3306      .Input("gradients: float")
3307      .Input("inputs: float")
3308      .Input("min: float")
3309      .Input("max: float")
3310      .Output("backprops_wrt_input: float")
3311      .Output("backprop_wrt_min: float")
3312      .Output("backprop_wrt_max: float")
```

```
3313      .SetShapeFn([](InferenceContext* c) {
3314        ShapeHandle inputs;
3315        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &inputs));
3316        TF_RETURN_IF_ERROR(c->WithRankAtMost(inputs, 4, &inputs));
3317        TF_RETURN_IF_ERROR(c->Merge(inputs, c->input(1), &inputs));
3318
3319        ShapeHandle last_dim = c->Vector(c->Dim(inputs, -1));
3320
3321        ShapeHandle min_max;
3322        TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &min_max));
3323        TF_RETURN_IF_ERROR(c->Merge(min_max, last_dim, &min_max));
3324        TF_RETURN_IF_ERROR(c->Merge(c->input(3), min_max, &min_max));
3325
3326        c->set_output(0, inputs);
3327        c->set_output(1, min_max);
3328        c->set_output(2, min_max);
3329        return Status::OK();
3330      });
3331
3332  REGISTER_OP("Fingerprint")
3333      .Input("data: T")
3334      .Input("method: string")
3335      .Output("fingerprint: uint8")
3336      .Attr("T: type")
3337      .SetShapeFn([](InferenceContext* c) {
3338        ShapeHandle unused;
3339        TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &unused));
3340        TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));
3341
3342        DimensionHandle fingerprint_size;
3343        const Tensor* method = c->input_tensor(1);
3344        if (method == nullptr) {
3345          fingerprint_size = c->UnknownDim();
3346        } else {
3347          if (method->dims() != 0) {
3348            return errors::InvalidArgument("`method` must be rank 0: ",
3349                                           method->shape());
3350          }
3351          const string& method_string = method->scalar<tstring>()();
3352          if (method_string != "farmhash64") {
3353            return errors::InvalidArgument("Unsupported method: ", method_string);
3354          }
3355          fingerprint_size = c->MakeDim(sizeof(uint64));
3356        }
3357
3358        DimensionHandle batch = c->Dim(c->input(0), 0);
3359        c->set_output(0, c->MakeShape({batch, fingerprint_size}));
3360        return Status::OK();
3361      });
```

```
3362
3363  #ifdef INTEL_MKL
3364  REGISTER_OP("_MklConcat")
3365      .Input("concat_dim: int32")
3366      .Input("values: N * T")
3367      .Input("mkl_concat_dim: uint8")
3368      .Input("mkl_values: N * uint8")
3369      .Output("output: T")
3370      .Output("mkl_output: uint8")
3371      .Attr("N: int >= 2")
3372      .Attr("T: type")
3373      .SetShapeFn([](InferenceContext* c) {
3374        return shape_inference::ConcatShape(c, c->num_inputs() - 3);
3375      })
3376      .Doc(R"doc(
3377  MKL version of Concat operator. Uses MKL DNN APIs to perform concatenation.
3378
3379  NOTE Do not invoke this operator directly in Python. Graph rewrite pass is
3380  expected to invoke these operators.
3381  )doc");
3382  #endif
3383
3384  // Deprecated op registrations:
3385
3386  // The following can be deleted after 10mar2017.
3387  REGISTER_OP("BatchMatrixDiag")
3388      .Input("diagonal: T")
3389      .Output("output: T")
3390      .Attr("T: type")
3391      .Deprecated(14, "Use MatrixDiag")
3392      .SetShapeFn(shape_inference::UnknownShape);
3393  REGISTER_OP("BatchMatrixSetDiag")
3394      .Input("input: T")
3395      .Input("diagonal: T")
3396      .Output("output: T")
3397      .Attr("T: type")
3398      .Deprecated(14, "Use MatrixSetDiag")
3399      .SetShapeFn(shape_inference::UnknownShape);
3400  REGISTER_OP("BatchMatrixDiagPart")
3401      .Input("input: T")
3402      .Output("diagonal: T")
3403      .Attr("T: type")
3404      .Deprecated(14, "Use MatrixDiagPart")
3405      .SetShapeFn(shape_inference::UnknownShape);
3406  REGISTER_OP("BatchMatrixBandPart")
3407      .Input("input: T")
3408      .Input("num_lower: int64")
3409      .Input("num_upper: int64")
3410      .Output("band: T")
```

```
3411      .Attr("T: type")
3412      .Deprecated(14, "Use MatrixBandPart")
3413      .SetShapeFn(shape_inference::UnknownShape);
3414
3415  }  // namespace tensorflow
```