miaout17 Prevent OOB access in QuantizeV2 shape inference ... History

42 contributors +27

2624 lines (2332 sloc) | 100 KB ...

```
1  /* Copyright 2016 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 #include "tensorflow/core/framework/common_shape_fns.h"
16
17 #include "absl/container/flat_hash_map.h"
18 #include "absl/container/flat_hash_set.h"
19 #include "absl/strings/match.h"
20 #include "absl/strings/str_split.h"
21 #include "absl/strings/string_view.h"
22 #include "tensorflow/core/framework/attr_value.pb.h"
23 #include "tensorflow/core/framework/shape_inference.h"
24 #include "tensorflow/core/lib/core/errors.h"
25 #include "tensorflow/core/lib/gtl/inlined_vector.h"
26 #include "tensorflow/core/util/einsum_op_util.h"
27
28 namespace tensorflow {
29
```

```
namespace shape_inference {

// The V2 version computes windowed output size with arbitrary dilation_rate and
// explicit padding, while the original version only handles the cases where
// dilation_rates equal to 1 and the padding is SAME or VALID.
Status GetWindowedOutputSizeFromDimsV2(
    shape_inference::InferenceContext* c,
    shape_inference::DimensionHandle input_size,
    shape_inference::DimensionOrConstant filter_size, int64_t dilation_rate,
    int64_t stride, Padding padding_type, int64_t padding_before,
    int64_t padding_after, shape_inference::DimensionHandle* output_size) {
  if (stride <= 0) {
    return errors::InvalidArgument("Stride must be > 0, but got ", stride);
  }

  if (dilation_rate < 1) {
    return errors::InvalidArgument("Dilation rate must be >= 1, but got ",
                                   dilation_rate);
  }

  // See also the parallel implementation in GetWindowedOutputSizeVerbose.
  switch (padding_type) {
    case Padding::VALID:
      padding_before = padding_after = 0;
      TF_FALLTHROUGH_INTENDED;
    case Padding::EXPLICIT:
      TF_RETURN_IF_ERROR(
          c->Add(input_size, padding_before + padding_after, &input_size));
      if (dilation_rate > 1) {
        DimensionHandle window_size;
        TF_RETURN_IF_ERROR(
            c->Subtract(c->MakeDim(filter_size), 1, &window_size));
        TF_RETURN_IF_ERROR(
            c->Multiply(window_size, dilation_rate, &window_size));
        TF_RETURN_IF_ERROR(c->Add(window_size, 1, &window_size));
        TF_RETURN_IF_ERROR(c->Subtract(input_size, window_size, output_size));
      } else {
        TF_RETURN_IF_ERROR(c->Subtract(input_size, filter_size, output_size));
      }
      TF_RETURN_IF_ERROR(c->Add(*output_size, stride, output_size));
      TF_RETURN_IF_ERROR(c->Divide(*output_size, stride,
                                   /*evenly_divisible=*/false, output_size));
      break;
    case Padding::SAME:
      TF_RETURN_IF_ERROR(c->Add(input_size, stride - 1, output_size));
      TF_RETURN_IF_ERROR(c->Divide(*output_size, stride,
                                   /*evenly_divisible=*/false, output_size));
      break;
  }
```

```
  return Status::OK();
}

Status GetWindowedOutputSizeFromDims(
    shape_inference::InferenceContext* c,
    shape_inference::DimensionHandle input_size,
    shape_inference::DimensionOrConstant filter_size, int64_t stride,
    Padding padding_type, shape_inference::DimensionHandle* output_size) {
  if (padding_type == Padding::EXPLICIT) {
    return errors::Internal(
        "GetWindowedOutputSizeFromDims does not handle EXPLICIT padding; call "
        "GetWindowedOutputSizeFromDimsV2 instead");
  }
  return GetWindowedOutputSizeFromDimsV2(c, input_size, filter_size,
                                         /*dilation_rate=*/1, stride,
                                         padding_type,
                                         // Give dummy values of -1 to
                                         // padding_before and padding_after,
                                         // since explicit padding is not used.
                                         -1, -1, output_size);
}

Status UnchangedShape(shape_inference::InferenceContext* c) {
  c->set_output(0, c->input(0));
  auto* handle_data = c->input_handle_shapes_and_types(0);
  if (handle_data != nullptr) {
    c->set_output_handle_shapes_and_types(0, *handle_data);
  }
  return Status::OK();
}

Status MatMulShape(shape_inference::InferenceContext* c) {
  ShapeHandle a;
  TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 2, &a));

  ShapeHandle b;
  TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 2, &b));

  bool transpose_a, transpose_b;
  TF_RETURN_IF_ERROR(c->GetAttr("transpose_a", &transpose_a));
  TF_RETURN_IF_ERROR(c->GetAttr("transpose_b", &transpose_b));
  DimensionHandle output_rows = transpose_a ? c->Dim(a, 1) : c->Dim(a, 0);
  DimensionHandle output_cols = transpose_b ? c->Dim(b, 0) : c->Dim(b, 1);

  // Validate that the inner shapes are compatible.
  DimensionHandle inner_a = transpose_a ? c->Dim(a, 0) : c->Dim(a, 1);
  DimensionHandle inner_b = transpose_b ? c->Dim(b, 1) : c->Dim(b, 0);
  DimensionHandle merged;
  TF_RETURN_IF_ERROR(c->Merge(inner_a, inner_b, &merged));
```

```
128
129   c->set_output(0, c->Matrix(output_rows, output_cols));
130   return Status::OK();
131 }
132
133 namespace {
134
135 // Validate that an Einsum subscript contains exactly one or zero ellipsis; and
136 // that periods (.) occur only within an ellipses (...).
137 Status ValidateEinsumEllipsis(absl::string_view subscript,
138                               bool* found_ellipsis) {
139   const int num_periods = absl::c_count(subscript, '.');
140   if (num_periods != 0 && num_periods != 3) {
141     return errors::InvalidArgument(
142         "Expected at most one ellipsis (...), but found ", num_periods,
143         " periods (.) in the input subscript: ", subscript);
144   }
145   if (num_periods == 3 && !absl::StrContains(subscript, "...")) {
146     return errors::InvalidArgument(
147         "Periods found outside of ellipsis in subscript: ", subscript);
148   }
149   *found_ellipsis = num_periods > 0;
150   return Status::OK();
151 }
152
153 }  // namespace
154
155 Status EinsumShape(shape_inference::InferenceContext* c) {
156   // We assume that the equation has a valid format. Either (x),(y)->(z)
157   // or (x)->(z), where each of (x), (y) and (z) are concatenation of zero or
158   // more latin alphabets and contains at most one ellipsis ('...').
159   string equation;
160   TF_RETURN_IF_ERROR(c->GetAttr("equation", &equation));
161   gtl::InlinedVector<string, 2> input_labels;
162   string output_labels;
163   TF_RETURN_IF_ERROR(
164       ParseEinsumEquation(equation, &input_labels, &output_labels));
165
166   if (c->num_inputs() == 0 || c->num_inputs() > 2) {
167     return errors::InvalidArgument("Expected either 1 or 2 inputs but got: ",
168                                    c->num_inputs());
169   }
170   const int input_labels_size = input_labels.size();
171   if (c->num_inputs() != input_labels_size) {
172     return errors::InvalidArgument("Expected ", input_labels.size(),
173                                    " inputs for equation ", equation,
174                                    " but got: ", c->num_inputs());
175   }
176
```

```
  // Validate input subscripts, build the label to dimension mapping and obtain
  // the broadcast shapes that map to ellipsis.
  absl::flat_hash_map<char, DimensionHandle> label_to_dimension;
  gtl::InlinedVector<ShapeHandle, 2> input_bcast_shapes(c->num_inputs());
  for (int i = 0, end = c->num_inputs(); i < end; ++i) {
    bool has_ellipsis = false;
    TF_RETURN_IF_ERROR(ValidateEinsumEllipsis(input_labels[i], &has_ellipsis));
    ShapeHandle input_shape = c->input(i);
    // Validate that the input rank is sufficient for the given number of named
    // labels.
    if (c->RankKnown(input_shape)) {
      if (has_ellipsis) {
        const int num_named_labels =
            static_cast<int>(input_labels[i].size()) - 3;
        TF_RETURN_WITH_CONTEXT_IF_ERROR(
            c->WithRankAtLeast(input_shape, num_named_labels, &input_shape),
            " for ", i, "th input and equation: ", equation);
      } else {
        const int num_named_labels = static_cast<int>(input_labels[i].size());
        TF_RETURN_WITH_CONTEXT_IF_ERROR(
            c->WithRank(input_shape, num_named_labels, &input_shape), " for ",
            i, "th input and equation: ", equation);
      }
    }

    bool seen_ellipsis = false;
    input_bcast_shapes[i] = c->Scalar();
    // Run through the input labels; populate label_to_dimension mapping and
    // compute the broadcast shapes corresponding to the ellipsis (if present).
    for (int label_idx = 0, end = input_labels[i].size(); label_idx < end;
         ++label_idx) {
      const char label = input_labels[i][label_idx];
      // Calculate the input axis that the current label is referring to. After
      // the ellipsis, the axis may be found by using negative indices; i.e the
      // (rank - k)th dimension corresponds to the (num_labels - k)th label.
      const int64_t axis_before_ellipsis = label_idx;
      const int64_t axis_after_ellipsis =
          c->RankKnown(input_shape)
              ? label_idx + c->Rank(input_shape) - input_labels[i].size()
              : -1;

      // Populate the input broadcast shape when we encounter an ellipsis (...).
      if (label == '.') {
        if (!c->RankKnown(input_shape)) {
          input_bcast_shapes[i] = c->UnknownShape();
        } else {
          // The broadcast shape runs till the named label right after the
          // ellipsis, the label with index (label_idx + 3).
          TF_RETURN_IF_ERROR(c->Subshape(input_shape, axis_before_ellipsis,
```

```
                                      axis_after_ellipsis + 3,
                                      &input_bcast_shapes[i]));
        }
        label_idx += 2;  // Skip the rest of the ellipsis.
        seen_ellipsis = true;
        continue;
      }
      // Obtain the dimension that the current label corresponds to.
      int64_t axis = seen_ellipsis ? axis_after_ellipsis : axis_before_ellipsis;
      DimensionHandle new_dim = c->RankKnown(input_shape)
                                    ? c->Dim(input_shape, axis)
                                    : c->UnknownDim();
      // If we've seen this label before, make sure previous and current
      // dimensions are compatible.
      if (label_to_dimension.contains(label)) {
        DimensionHandle merged;
        TF_RETURN_IF_ERROR(
            c->Merge(label_to_dimension[label], new_dim, &merged));
        label_to_dimension[label] = merged;
      } else {
        label_to_dimension[label] = new_dim;
      }
    }
  }

  // For two inputs, broadcast the two input broadcast shapes to create the
  // output broadcast shape. For one input, just copy the single broadcast
  // shape.
  ShapeHandle output_bcast_shape;
  if (input_bcast_shapes.size() == 1) {
    output_bcast_shape = input_bcast_shapes[0];
  } else if (input_bcast_shapes.size() == 2) {
    TF_RETURN_IF_ERROR(BroadcastBinaryOpOutputShapeFnHelper(
        c, input_bcast_shapes[0], input_bcast_shapes[1], true,
        &output_bcast_shape));
  }

  bool output_has_ellipsis = false;
  TF_RETURN_IF_ERROR(
      ValidateEinsumEllipsis(output_labels, &output_has_ellipsis));
  if (output_has_ellipsis) {
    // If the output subscript has ellipsis and the output broadcast rank is
    // unknown, then the output shape should have unknown rank.
    if (!c->RankKnown(output_bcast_shape)) {
      c->set_output(0, c->UnknownShape());
      return Status::OK();
    }
  } else {
    // If the output subscripts don't have ellipsis then make sure the output
```

```
    // broadcasting shape is empty.
    TF_RETURN_WITH_CONTEXT_IF_ERROR(
        c->WithRankAtMost(output_bcast_shape, 0, &output_bcast_shape),
        " for einsum equation '", equation,
        "' without ellipsis (...) in the output subscripts where input(s) have "
        "non-empty broadcasting shape");
    output_bcast_shape = c->Scalar();
  }

  // Create the output shape from output labels and label_to_dimension mapping.
  std::vector<DimensionHandle> output_dims;
  for (int label_idx = 0, end = output_labels.size(); label_idx < end;
       ++label_idx) {
    const char label = output_labels[label_idx];
    // Append the output_bcast_shape when the ellipsis is encountered.
    if (label == '.') {
      for (int k = 0; k < c->Rank(output_bcast_shape); ++k) {
        output_dims.push_back(c->Dim(output_bcast_shape, k));
      }
      label_idx += 2;  // Skip the rest of the ellipsis.
      continue;
    }
    auto dimension_it = label_to_dimension.find(label);
    if (dimension_it == label_to_dimension.end()) {
      return errors::InvalidArgument(
          "Einsum output subscripts for equation '", equation, "' has label '",
          label, "' which is not present in the input subscripts");
    }
    output_dims.push_back(dimension_it->second);
  }
  c->set_output(0, c->MakeShape(output_dims));
  return Status::OK();
}

Status BatchMatMulV2Shape(shape_inference::InferenceContext* c) {
  ShapeHandle a_shape;
  ShapeHandle b_shape;
  TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &a_shape));
  TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 2, &b_shape));

  // Determine output rows and columns.
  bool adj_x;
  bool adj_y;
  TF_RETURN_IF_ERROR(c->GetAttr("adj_x", &adj_x));
  TF_RETURN_IF_ERROR(c->GetAttr("adj_y", &adj_y));
  DimensionHandle output_rows = c->Dim(a_shape, adj_x ? -1 : -2);
  DimensionHandle output_cols = c->Dim(b_shape, adj_y ? -2 : -1);

  // Inner dimensions should be compatible.
```

```
  DimensionHandle inner_merged;
  TF_RETURN_IF_ERROR(c->Merge(c->Dim(a_shape, adj_x ? -2 : -1),
                              c->Dim(b_shape, adj_y ? -1 : -2), &inner_merged));

  // Batch dimensions should broadcast with each other.
  ShapeHandle a_batch_shape;
  ShapeHandle b_batch_shape;
  ShapeHandle output_batch_shape;
  TF_RETURN_IF_ERROR(c->Subshape(a_shape, 0, -2, &a_batch_shape));
  TF_RETURN_IF_ERROR(c->Subshape(b_shape, 0, -2, &b_batch_shape));

  TF_RETURN_IF_ERROR(BroadcastBinaryOpOutputShapeFnHelper(
      c, a_batch_shape, b_batch_shape, true, &output_batch_shape));

  ShapeHandle output_shape;
  TF_RETURN_IF_ERROR(c->Concatenate(
      output_batch_shape, c->Matrix(output_rows, output_cols), &output_shape));

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

Status BatchMatMulShape(shape_inference::InferenceContext* c) {
  ShapeHandle a_shape;
  ShapeHandle b_shape;
  TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &a_shape));
  TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 2, &b_shape));

  // Determine output rows and cols.
  bool adj_x;
  bool adj_y;
  TF_RETURN_IF_ERROR(c->GetAttr("adj_x", &adj_x));
  TF_RETURN_IF_ERROR(c->GetAttr("adj_y", &adj_y));
  DimensionHandle output_rows = c->Dim(a_shape, adj_x ? -1 : -2);
  DimensionHandle output_cols = c->Dim(b_shape, adj_y ? -2 : -1);

  // Batch dims match between inputs.
  ShapeHandle a_batch_dims;
  ShapeHandle b_batch_dims;
  ShapeHandle batch_dims;
  TF_RETURN_IF_ERROR(c->Subshape(a_shape, 0, -2, &a_batch_dims));
  TF_RETURN_IF_ERROR(c->Subshape(b_shape, 0, -2, &b_batch_dims));
  TF_RETURN_IF_ERROR(c->Merge(a_batch_dims, b_batch_dims, &batch_dims));

  // Assert inner dims match.
  DimensionHandle unused;
  TF_RETURN_IF_ERROR(c->Merge(c->Dim(a_shape, adj_x ? -2 : -1),
                              c->Dim(b_shape, adj_y ? -1 : -2), &unused));

```

```
373   ShapeHandle out;
374   TF_RETURN_IF_ERROR(
375       c->Concatenate(batch_dims, c->Matrix(output_rows, output_cols), &out));
376   c->set_output(0, out);
377   return Status::OK();
378 }
379
380 // ----------------------------------------------------------------------------
381
382 Status BiasAddShape(shape_inference::InferenceContext* c) {
383   ShapeHandle input_shape;
384
385   // Fetch the data_format attribute, which may not exist.
386   string data_format;
387   Status s = c->GetAttr("data_format", &data_format);
388
389   if (s.ok() && data_format == "NCHW") {
390     TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 3, &input_shape));
391   } else {
392     TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &input_shape));
393   }
394
395   ShapeHandle bias_shape;
396   TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &bias_shape));
397   DimensionHandle bias_dim = c->Dim(bias_shape, 0);
398
399   // If rank unknown, return unknown shape.
400   if (!c->RankKnown(input_shape)) {
401     c->set_output(0, c->UnknownShape());
402     return Status::OK();
403   }
404
405   // Output has the same shape as the input, and matches the length of
406   // the bias in its bias dimension.
407   ShapeHandle output_shape;
408   if (s.ok() && data_format == "NCHW") {
409     // Merge the length of bias_shape into the third to last dimension
410     ShapeHandle first;
411     TF_RETURN_IF_ERROR(c->Subshape(input_shape, 0, 1, &first));
412
413     ShapeHandle last;
414     TF_RETURN_IF_ERROR(c->Subshape(input_shape, 2, &last));
415
416     DimensionHandle input_bias_dim = c->Dim(input_shape, 1);
417     DimensionHandle merged_bias_dim;
418     TF_RETURN_IF_ERROR(c->Merge(input_bias_dim, bias_dim, &merged_bias_dim));
419     ShapeHandle merged_bias = c->Vector(merged_bias_dim);
420
421     ShapeHandle temp;
```

```
422       TF_RETURN_IF_ERROR(c->Concatenate(first, merged_bias, &temp));
423       TF_RETURN_IF_ERROR(c->Concatenate(temp, last, &output_shape));
424     } else {
425       ShapeHandle all_but_bias;
426       TF_RETURN_IF_ERROR(c->Subshape(input_shape, 0, -1, &all_but_bias));
427
428       DimensionHandle input_bias_dim = c->Dim(input_shape, -1);
429       DimensionHandle merged_bias_dim;
430       TF_RETURN_IF_ERROR(c->Merge(input_bias_dim, bias_dim, &merged_bias_dim));
431
432       ShapeHandle merged_bias = c->Vector(merged_bias_dim);
433       TF_RETURN_IF_ERROR(
434           c->Concatenate(all_but_bias, merged_bias, &output_shape));
435     }
436
437     c->set_output(0, output_shape);
438     return Status::OK();
439   }
440
441   Status BiasAddGradShape(shape_inference::InferenceContext* c) {
442     ShapeHandle input_shape;
443     // Fetch the data_format attribute, which may not exist.
444     string data_format;
445     Status s = c->GetAttr("data_format", &data_format);
446
447     if (s.ok() && data_format == "NCHW") {
448       TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 3, &input_shape));
449       c->set_output(0, c->Vector(c->Dim(input_shape, 1)));
450     } else {
451       TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &input_shape));
452       c->set_output(0, c->Vector(c->Dim(input_shape, -1)));
453     }
454
455     return Status::OK();
456   }
457
458   Status CheckFormatConstraintsOnShape(const TensorFormat tensor_format,
459                                        const ShapeHandle shape_handle,
460                                        const string& tensor_name,
461                                        shape_inference::InferenceContext* c) {
462     if (tensor_format == FORMAT_NCHW_VECT_C) {
463       // Check that the vect dim has size 4 or 32.
464       const int num_dims = c->Rank(shape_handle);
465       DimensionHandle vect_dim = c->Dim(
466           shape_handle, GetTensorInnerFeatureDimIndex(num_dims, tensor_format));
467       int64_t vect_dim_val = c->Value(vect_dim);
468       if (vect_dim_val != 4 && vect_dim_val != 32) {
469         return errors::InvalidArgument(
470             "VECT_C dimension must be 4 or 32, but is ", vect_dim_val);
```

```
    }
  }

  return Status::OK();
}

Status DatasetIteratorShape(shape_inference::InferenceContext* c) {
  shape_inference::ShapeHandle unused;
  TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 0, &unused));
  std::vector<PartialTensorShape> output_shapes;
  TF_RETURN_IF_ERROR(c->GetAttr("output_shapes", &output_shapes));
  const int output_shapes_size = output_shapes.size();
  if (output_shapes_size != c->num_outputs()) {
    return errors::InvalidArgument(
        "`output_shapes` must be the same length as `output_types` (",
        output_shapes.size(), " vs. ", c->num_outputs());
  }
  for (size_t i = 0; i < output_shapes.size(); ++i) {
    shape_inference::ShapeHandle output_shape_handle;
    TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(
        output_shapes[i], &output_shape_handle));
    c->set_output(static_cast<int>(i), output_shape_handle);
  }
  return Status::OK();
}

Status MakeShapeFromFormat(TensorFormat format, DimensionOrConstant N,
                           const std::vector<DimensionOrConstant>& spatial,
                           DimensionOrConstant C, ShapeHandle* out,
                           shape_inference::InferenceContext* context) {
  const int num_dims = GetTensorDimsFromSpatialDims(spatial.size(), format);
  std::vector<DimensionHandle> dims_actual(num_dims);
  dims_actual[GetTensorBatchDimIndex(num_dims, format)] = context->MakeDim(N);
  int outer_c_index = GetTensorFeatureDimIndex(num_dims, format);
  dims_actual[outer_c_index] = context->MakeDim(C);
  if (format == FORMAT_NCHW_VECT_C) {
    dims_actual[GetTensorInnerFeatureDimIndex(num_dims, format)] =
        context->MakeDim(4);
  } else if (format == FORMAT_NHWC_VECT_W) {
    dims_actual[GetTensorInnerWidthDimIndex(num_dims, format)] =
        context->MakeDim(4);
  }
  for (int spatial_dim = 0, end = spatial.size(); spatial_dim < end;
       spatial_dim++) {
    dims_actual[GetTensorSpatialDimIndex(num_dims, format, spatial_dim)] =
        context->MakeDim(spatial[spatial_dim]);
  }
  *out = context->MakeShape(dims_actual);
  return Status::OK();
```

```
}

Status DimensionsFromShape(ShapeHandle shape, TensorFormat format,
                           DimensionHandle* batch_dim,
                           gtl::MutableArraySlice<DimensionHandle> spatial_dims,
                           DimensionHandle* filter_dim,
                           InferenceContext* context) {
  const int32_t rank =
      GetTensorDimsFromSpatialDims(spatial_dims.size(), format);
  // Batch.
  *batch_dim = context->Dim(shape, GetTensorBatchDimIndex(rank, format));
  // Spatial.
  for (int spatial_dim_index = 0, end = spatial_dims.size();
       spatial_dim_index < end; ++spatial_dim_index) {
    spatial_dims[spatial_dim_index] = context->Dim(
        shape, GetTensorSpatialDimIndex(rank, format, spatial_dim_index));
  }
  // Channel.
  *filter_dim = context->Dim(shape, GetTensorFeatureDimIndex(rank, format));
  if (format == FORMAT_NCHW_VECT_C) {
    TF_RETURN_IF_ERROR(context->Multiply(
        *filter_dim,
        context->Dim(shape, GetTensorInnerFeatureDimIndex(rank, format)),
        filter_dim));
  }
  return Status::OK();
}

// vect_size must be provided if format is NCHW_VECT_C.
Status ShapeFromDimensions(DimensionHandle batch_dim,
                           gtl::ArraySlice<DimensionHandle> spatial_dims,
                           DimensionHandle filter_dim, TensorFormat format,
                           absl::optional<DimensionHandle> vect_size,
                           InferenceContext* context, ShapeHandle* shape) {
  const int32_t rank =
      GetTensorDimsFromSpatialDims(spatial_dims.size(), format);
  std::vector<DimensionHandle> out_dims(rank);

  // Batch.
  out_dims[tensorflow::GetTensorBatchDimIndex(rank, format)] = batch_dim;
  // Spatial.
  for (int spatial_dim_index = 0, end = spatial_dims.size();
       spatial_dim_index < end; ++spatial_dim_index) {
    out_dims[tensorflow::GetTensorSpatialDimIndex(
        rank, format, spatial_dim_index)] = spatial_dims[spatial_dim_index];
  }
  // Channel.
  if (format == tensorflow::FORMAT_NCHW_VECT_C) {
    // When format is NCHW_VECT_C, factor the feature map count into the outer
```

```
    // feature count and the inner feature count (4 or 32).
    CHECK(vect_size.has_value());  // Crash ok.
    TF_RETURN_IF_ERROR(context->Divide(
        filter_dim, *vect_size, /*evenly_divisible=*/true,
        &out_dims[tensorflow::GetTensorFeatureDimIndex(rank, format)]));
    out_dims[GetTensorInnerFeatureDimIndex(rank, format)] = *vect_size;
  } else {
    out_dims[tensorflow::GetTensorFeatureDimIndex(rank, format)] = filter_dim;
  }

  *shape = context->MakeShape(out_dims);
  return tensorflow::Status::OK();
}

namespace {

Status Conv2DShapeImpl(shape_inference::InferenceContext* c,
                       bool supports_explicit_padding) {
  string data_format_str, filter_format_str;
  if (!c->GetAttr("data_format", &data_format_str).ok()) {
    data_format_str = "NHWC";
  }
  if (!c->GetAttr("filter_format", &filter_format_str).ok()) {
    filter_format_str = "HWIO";
  }

  TensorFormat data_format;
  if (!FormatFromString(data_format_str, &data_format)) {
    return errors::InvalidArgument("Invalid data format string: ",
                                   data_format_str);
  }
  FilterTensorFormat filter_format;
  if (!FilterFormatFromString(filter_format_str, &filter_format)) {
    return errors::InvalidArgument("Invalid filter format string: ",
                                   filter_format_str);
  }

  constexpr int num_spatial_dims = 2;
  const int rank = GetTensorDimsFromSpatialDims(num_spatial_dims, data_format);
  ShapeHandle conv_input_shape;
  TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &conv_input_shape));
  TF_RETURN_IF_ERROR(CheckFormatConstraintsOnShape(
      data_format, conv_input_shape, "conv_input", c));

  // The filter rank should match the input (4 for NCHW, 5 for NCHW_VECT_C).
  ShapeHandle filter_shape;
  TF_RETURN_IF_ERROR(c->WithRank(c->input(1), rank, &filter_shape));
  TF_RETURN_IF_ERROR(
      CheckFormatConstraintsOnShape(data_format, filter_shape, "filter", c));
```

```
618
619   std::vector<int32> dilations;
620   TF_RETURN_IF_ERROR(c->GetAttr("dilations", &dilations));
621
622   if (dilations.size() != 4) {
623     return errors::InvalidArgument(
624         "Conv2D requires the dilation attribute to contain 4 values, but got: ",
625         dilations.size());
626   }
627
628   std::vector<int32> strides;
629   TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
630
631   // strides.size() should be 4 (NCHW) even if the input is 5 (NCHW_VECT_C).
632   if (strides.size() != 4) {
633     return errors::InvalidArgument("Conv2D on data format ", data_format_str,
634                                    " requires the stride attribute to contain"
635                                    " 4 values, but got: ",
636                                    strides.size());
637   }
638
639   const int32_t stride_rows = GetTensorDim(strides, data_format, 'H');
640   const int32_t stride_cols = GetTensorDim(strides, data_format, 'W');
641   const int32_t dilation_rows = GetTensorDim(dilations, data_format, 'H');
642   const int32_t dilation_cols = GetTensorDim(dilations, data_format, 'W');
643
644   DimensionHandle batch_size_dim;
645   DimensionHandle input_depth_dim;
646   gtl::InlinedVector<DimensionHandle, 2> input_spatial_dims(2);
647   TF_RETURN_IF_ERROR(DimensionsFromShape(
648       conv_input_shape, data_format, &batch_size_dim,
649       absl::MakeSpan(input_spatial_dims), &input_depth_dim, c));
650
651   DimensionHandle output_depth_dim = c->Dim(
652       filter_shape, GetFilterDimIndex<num_spatial_dims>(filter_format, 'O'));
653   DimensionHandle filter_rows_dim = c->Dim(
654       filter_shape, GetFilterDimIndex<num_spatial_dims>(filter_format, 'H'));
655   DimensionHandle filter_cols_dim = c->Dim(
656       filter_shape, GetFilterDimIndex<num_spatial_dims>(filter_format, 'W'));
657   DimensionHandle filter_input_depth_dim;
658   if (filter_format == FORMAT_OIHW_VECT_I) {
659     TF_RETURN_IF_ERROR(c->Multiply(
660         c->Dim(filter_shape,
661                GetFilterDimIndex<num_spatial_dims>(filter_format, 'I')),
662         c->Dim(filter_shape,
663                GetFilterTensorInnerInputChannelsDimIndex(rank, filter_format)),
664         &filter_input_depth_dim));
665   } else {
666     filter_input_depth_dim = c->Dim(
```

```
667         filter_shape, GetFilterDimIndex<num_spatial_dims>(filter_format, 'I'));
668   }
669
670   // Check that the input tensor and the filter tensor agree on the channel
671   // count.
672   if (c->ValueKnown(input_depth_dim) && c->ValueKnown(filter_input_depth_dim)) {
673     int64_t input_depth_value = c->Value(input_depth_dim),
674             filter_input_depth_value = c->Value(filter_input_depth_dim);
675     if (filter_input_depth_value == 0)
676       return errors::InvalidArgument("Depth of filter must not be 0");
677     if (input_depth_value % filter_input_depth_value != 0)
678       return errors::InvalidArgument(
679           "Depth of input (", input_depth_value,
680           ") is not a multiple of input depth of filter (",
681           filter_input_depth_value, ")");
682     if (input_depth_value != filter_input_depth_value) {
683       int64_t num_groups = input_depth_value / filter_input_depth_value;
684       if (c->ValueKnown(output_depth_dim)) {
685         int64_t output_depth_value = c->Value(output_depth_dim);
686         if (num_groups == 0)
687           return errors::InvalidArgument("Number of groups must not be 0");
688         if (output_depth_value % num_groups != 0)
689           return errors::InvalidArgument(
690               "Depth of output (", output_depth_value,
691               ") is not a multiple of the number of groups (", num_groups, ")");
692       }
693     }
694   }
695
696   Padding padding;
697   TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
698
699   std::vector<int64_t> explicit_paddings;
700   if (supports_explicit_padding) {
701     Status s = c->GetAttr("explicit_paddings", &explicit_paddings);
702     // Use the default value, which is an empty list, if the attribute is not
703     // found. Otherwise return the error to the caller.
704     if (!s.ok() && !errors::IsNotFound(s)) {
705       return s;
706     }
707     TF_RETURN_IF_ERROR(CheckValidPadding(padding, explicit_paddings,
708                                          /*num_dims=*/4, data_format));
709   } else {
710     CHECK(padding != Padding::EXPLICIT);  // Crash ok.
711   }
712
713   DimensionHandle output_rows, output_cols;
714   int64_t pad_rows_before = -1, pad_rows_after = -1;
715   int64_t pad_cols_before = -1, pad_cols_after = -1;
```

```
716   if (padding == Padding::EXPLICIT) {
717     GetExplicitPaddingForDim(explicit_paddings, data_format, 'H',
718                              &pad_rows_before, &pad_rows_after);
719     GetExplicitPaddingForDim(explicit_paddings, data_format, 'W',
720                              &pad_cols_before, &pad_cols_after);
721   }
722   TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
723       c, input_spatial_dims[0], filter_rows_dim, dilation_rows, stride_rows,
724       padding, pad_rows_before, pad_rows_after, &output_rows));
725   TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
726       c, input_spatial_dims[1], filter_cols_dim, dilation_cols, stride_cols,
727       padding, pad_cols_before, pad_cols_after, &output_cols));
728
729   absl::optional<DimensionHandle> vect_size;
730   if (data_format == FORMAT_NCHW_VECT_C) {
731     vect_size.emplace(c->Dim(conv_input_shape,
732                              GetTensorInnerFeatureDimIndex(rank, data_format)));
733   }
734   ShapeHandle output_shape;
735   TF_RETURN_IF_ERROR(ShapeFromDimensions(
736       batch_size_dim, {output_rows, output_cols}, output_depth_dim, data_format,
737       vect_size, c, &output_shape));
738   c->set_output(0, output_shape);
739   return Status::OK();
740 }
741
742 }  // namespace
743
744 // Shape function for Conv2D-like operations that support explicit padding.
745 Status Conv2DShapeWithExplicitPadding(shape_inference::InferenceContext* c) {
746   return Conv2DShapeImpl(c, true);
747 }
748
749 // Shape function for Conv2D-like operations that do not support explicit
750 // padding.
751 Status Conv2DShape(shape_inference::InferenceContext* c) {
752   return Conv2DShapeImpl(c, false);
753 }
754
755 // TODO(mjanusz): Unify all conv/pooling shape functions.
756 Status Conv3DShape(shape_inference::InferenceContext* c) {
757   ShapeHandle input_shape;
758   TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 5, &input_shape));
759   ShapeHandle filter_shape;
760   TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 5, &filter_shape));
761
762   string data_format;
763   Status s = c->GetAttr("data_format", &data_format);
764
```

```
  std::vector<int32> dilations;
  TF_RETURN_IF_ERROR(c->GetAttr("dilations", &dilations));

  if (dilations.size() != 5) {
    return errors::InvalidArgument(
        "Conv3D requires the dilation attribute to contain 5 values, but got: ",
        dilations.size());
  }

  std::vector<int32> strides;
  TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
  if (strides.size() != 5) {
    return errors::InvalidArgument(
        "Conv3D requires the stride attribute to contain 5 values, but got: ",
        strides.size());
  }

  int32_t stride_planes, stride_rows, stride_cols;
  int32_t dilation_planes, dilation_rows, dilation_cols;
  if (s.ok() && data_format == "NCDHW") {
    // Convert input_shape to NDHWC.
    auto dim = [&](char dimension) {
      return c->Dim(input_shape, GetTensorDimIndex<3>(FORMAT_NCHW, dimension));
    };
    input_shape =
        c->MakeShape({{dim('N'), dim('0'), dim('1'), dim('2'), dim('C')}});
    stride_planes = strides[2];
    stride_rows = strides[3];
    stride_cols = strides[4];
    dilation_planes = dilations[2];
    dilation_cols = dilations[3];
    dilation_rows = dilations[4];
  } else {
    stride_planes = strides[1];
    stride_rows = strides[2];
    stride_cols = strides[3];
    dilation_planes = dilations[1];
    dilation_cols = dilations[2];
    dilation_rows = dilations[3];
  }

  DimensionHandle batch_size_dim = c->Dim(input_shape, 0);
  DimensionHandle in_planes_dim = c->Dim(input_shape, 1);
  DimensionHandle in_rows_dim = c->Dim(input_shape, 2);
  DimensionHandle in_cols_dim = c->Dim(input_shape, 3);
  DimensionHandle input_depth_dim = c->Dim(input_shape, 4);

  DimensionHandle filter_planes_dim = c->Dim(filter_shape, 0);
  DimensionHandle filter_rows_dim = c->Dim(filter_shape, 1);
```

```
  DimensionHandle filter_cols_dim = c->Dim(filter_shape, 2);
  DimensionHandle filter_input_depth_dim = c->Dim(filter_shape, 3);
  DimensionHandle output_depth_dim = c->Dim(filter_shape, 4);

  // Check that the input tensor and the filter tensor agree on the channel
  // count.
  if (c->ValueKnown(input_depth_dim) && c->ValueKnown(filter_input_depth_dim)) {
    int64_t input_depth_value = c->Value(input_depth_dim),
            filter_input_depth_value = c->Value(filter_input_depth_dim);
    if (filter_input_depth_value == 0)
      return errors::InvalidArgument("Depth of filter must not be 0");
    if (input_depth_value % filter_input_depth_value != 0)
      return errors::InvalidArgument(
          "Depth of input (", input_depth_value,
          ") is not a multiple of input depth of filter (",
          filter_input_depth_value, ")");
    if (input_depth_value != filter_input_depth_value) {
      int64_t num_groups = input_depth_value / filter_input_depth_value;
      if (c->ValueKnown(output_depth_dim)) {
        int64_t output_depth_value = c->Value(output_depth_dim);
        if (num_groups == 0)
          return errors::InvalidArgument("Number of groups must not be 0");
        if (output_depth_value % num_groups != 0)
          return errors::InvalidArgument(
              "Depth of output (", output_depth_value,
              ") is not a multiple of the number of groups (", num_groups, ")");
      }
    }
  }

  Padding padding;
  TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
  DimensionHandle output_planes, output_rows, output_cols;

  TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
      c, in_planes_dim, filter_planes_dim, dilation_planes, stride_planes,
      padding, -1, -1, &output_planes));
  TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
      c, in_rows_dim, filter_rows_dim, dilation_rows, stride_rows, padding, -1,
      -1, &output_rows));
  TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
      c, in_cols_dim, filter_cols_dim, dilation_cols, stride_cols, padding, -1,
      -1, &output_cols));

  ShapeHandle output_shape;
  if (data_format == "NCDHW") {
    output_shape = c->MakeShape({batch_size_dim, output_depth_dim,
                                 output_planes, output_rows, output_cols});
  } else {
```

```
    output_shape = c->MakeShape({batch_size_dim, output_planes, output_rows,
                                 output_cols, output_depth_dim});
  }
  c->set_output(0, output_shape);
  return Status::OK();
}

Status Conv2DBackpropInputShape(shape_inference::InferenceContext* c) {
  string data_format_str;
  if (!c->GetAttr("data_format", &data_format_str).ok()) {
    data_format_str = "NHWC";
  }
  TensorFormat data_format;
  if (!FormatFromString(data_format_str, &data_format)) {
    return errors::InvalidArgument("Invalid data format string: ",
                                   data_format_str);
  }

  // For the rest of this function, output_grad_* describes out_backprop and
  // input_grad_* describes in_backprop.
  ShapeHandle output_grad_shape = c->input(2);
  TF_RETURN_IF_ERROR(c->WithRank(output_grad_shape, 4, &output_grad_shape));
  ShapeHandle filter_shape = c->input(1);
  TF_RETURN_IF_ERROR(c->WithRank(filter_shape, 4, &filter_shape));

  DimensionHandle batch_size_dim;
  DimensionHandle output_grad_depth_dim;
  gtl::InlinedVector<DimensionHandle, 2> output_grad_spatial_dims(2);
  TF_RETURN_IF_ERROR(DimensionsFromShape(
      output_grad_shape, data_format, &batch_size_dim,
      absl::MakeSpan(output_grad_spatial_dims), &output_grad_depth_dim, c));
  DimensionHandle unused;
  TF_RETURN_IF_ERROR(
      c->Merge(output_grad_depth_dim, c->Dim(filter_shape, 3), &unused));

  ShapeHandle specified_input_grad_shape;
  TF_RETURN_IF_ERROR(
      c->MakeShapeFromShapeTensor(0, &specified_input_grad_shape));
  if (c->Rank(specified_input_grad_shape) == InferenceContext::kUnknownRank) {
    TF_RETURN_IF_ERROR(c->WithRank(specified_input_grad_shape, 4,
                                   &specified_input_grad_shape));
  }

  // input_grad_depth_dim doesn't equal c->Dim(filter_shape,2) when the number
  // of groups is larger than 1. If input_sizes is a 4D shape, we collect
  // input_grad_depth_dim from input_sizes; otherwise we compute it as
  // c->Dim(filter_shape,2).
  DimensionHandle input_grad_depth_dim;
  gtl::InlinedVector<DimensionHandle, 2> specified_input_grad_spatial_dims(2);
```

```
  int specified_input_grad_rank = c->Rank(specified_input_grad_shape);
  if (specified_input_grad_rank == 4) {
    DimensionHandle specified_batch_size_dim;
    TF_RETURN_IF_ERROR(DimensionsFromShape(
        specified_input_grad_shape, data_format, &specified_batch_size_dim,
        absl::MakeSpan(specified_input_grad_spatial_dims),
        &input_grad_depth_dim, c));
    TF_RETURN_IF_ERROR(
        c->Merge(specified_batch_size_dim, batch_size_dim, &unused));
  } else if (specified_input_grad_rank == 2) {
    specified_input_grad_spatial_dims[0] =
        c->Dim(specified_input_grad_shape, 0);
    specified_input_grad_spatial_dims[1] =
        c->Dim(specified_input_grad_shape, 1);
    input_grad_depth_dim = c->Dim(filter_shape, 2);
  } else {
    return errors::InvalidArgument(
        "Conv2DBackpropInput requires input_sizes to contain 4 values or 2 "
        "values, but got: ",
        specified_input_grad_rank);
  }

  ShapeHandle input_grad_shape;
  TF_RETURN_IF_ERROR(ShapeFromDimensions(
      batch_size_dim, specified_input_grad_spatial_dims, input_grad_depth_dim,
      data_format, /*vect_size=*/absl::nullopt, c, &input_grad_shape));
  c->set_output(0, input_grad_shape);
  return Status::OK();
}

Status Conv2DBackpropFilterWithBiasShape(shape_inference::InferenceContext* c) {
  ShapeHandle input_shape;
  // Fetch the data_format attribute, which may not exist.
  string data_format;
  Status s = c->GetAttr("data_format", &data_format);

  TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));
  if (s.ok() && data_format == "NCHW") {
    c->set_output(1, c->Vector(c->Dim(input_shape, -3)));
  } else {
    c->set_output(1, c->Vector(c->Dim(input_shape, -1)));
  }
  ShapeHandle sh;
  TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(1, &sh));
  TF_RETURN_IF_ERROR(c->WithRank(sh, 4, &sh));
  c->set_output(0, sh);
  return Status::OK();
}

```

```
961 namespace {
962
963 Status DepthwiseConv2DNativeShapeImpl(shape_inference::InferenceContext* c,
964                                       bool supports_explicit_padding) {
965   ShapeHandle input_shape;
966   TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &input_shape));
967   ShapeHandle filter_shape;
968   TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 4, &filter_shape));
969
970   std::vector<int32> strides;
971   TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
972
973   if (strides.size() != 4) {
974     return errors::InvalidArgument(
975         "DepthwiseConv2D requires the stride attribute to contain 4 values, "
976         "but got: ",
977         strides.size());
978   }
979
980   std::vector<int32> dilations;
981   if (!c->GetAttr("dilations", &dilations).ok()) {
982     dilations.resize(4, 1);
983   }
984
985   if (dilations.size() != 4) {
986     return errors::InvalidArgument(
987         "DepthwiseConv2D requires the dilations attribute to contain 4 values, "
988         "but got: ",
989         dilations.size());
990   }
991
992   string data_format_str;
993   Status s = c->GetAttr("data_format", &data_format_str);
994   TensorFormat data_format;
995   if (!s.ok() || !FormatFromString(data_format_str, &data_format)) {
996     data_format = FORMAT_NHWC;
997   }
998   int32_t stride_rows;
999   int32_t stride_cols;
1000  int32_t dilation_rows;
1001  int32_t dilation_cols;
1002  if (data_format == FORMAT_NCHW) {
1003    // Canonicalize input shape to NHWC so the shape inference code below can
1004    // process it.
1005    input_shape =
1006        c->MakeShape({{c->Dim(input_shape, 0), c->Dim(input_shape, 2),
1007                       c->Dim(input_shape, 3), c->Dim(input_shape, 1)}});
1008    stride_rows = strides[2];
1009    stride_cols = strides[3];
```

```
1010      dilation_rows = dilations[2];
1011      dilation_cols = dilations[3];
1012    } else {
1013      stride_rows = strides[1];
1014      stride_cols = strides[2];
1015      dilation_rows = dilations[1];
1016      dilation_cols = dilations[2];
1017    }
1018
1019    DimensionHandle batch_size_dim = c->Dim(input_shape, 0);
1020    DimensionHandle in_rows_dim = c->Dim(input_shape, 1);
1021    DimensionHandle in_cols_dim = c->Dim(input_shape, 2);
1022
1023    DimensionHandle filter_rows_dim = c->Dim(filter_shape, 0);
1024    DimensionHandle filter_cols_dim = c->Dim(filter_shape, 1);
1025    DimensionHandle input_depth = c->Dim(filter_shape, 2);
1026    DimensionHandle depth_multiplier = c->Dim(filter_shape, 3);
1027
1028    // Check that the input depths are compatible.
1029    TF_RETURN_IF_ERROR(
1030        c->Merge(c->Dim(input_shape, 3), input_depth, &input_depth));
1031
1032    DimensionHandle output_depth;
1033    TF_RETURN_IF_ERROR(c->Multiply(input_depth, depth_multiplier, &output_depth));
1034
1035    Padding padding;
1036    TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
1037
1038    std::vector<int64_t> explicit_paddings;
1039    if (supports_explicit_padding) {
1040      Status status = c->GetAttr("explicit_paddings", &explicit_paddings);
1041      // Use the default value, which is an empty list, if the attribute is not
1042      // found. Otherwise return the error to the caller.
1043      if (!status.ok() && !errors::IsNotFound(status)) {
1044        return status;
1045      }
1046      TF_RETURN_IF_ERROR(CheckValidPadding(padding, explicit_paddings,
1047                                           /*num_dims=*/4, data_format));
1048    } else {
1049      DCHECK(padding != Padding::EXPLICIT);
1050    }
1051
1052    // TODO(mrry,shlens): Raise an error if the stride would cause
1053    // information in the input to be ignored. This will require a change
1054    // in the kernel implementation.
1055    DimensionHandle output_rows, output_cols;
1056    int64_t pad_rows_before = -1, pad_rows_after = -1;
1057    int64_t pad_cols_before = -1, pad_cols_after = -1;
1058    if (padding == Padding::EXPLICIT) {
```

```
1059      GetExplicitPaddingForDim(explicit_paddings, data_format, 'H',
1060                               &pad_rows_before, &pad_rows_after);
1061      GetExplicitPaddingForDim(explicit_paddings, data_format, 'W',
1062                               &pad_cols_before, &pad_cols_after);
1063    }
1064    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
1065        c, in_rows_dim, filter_rows_dim, dilation_rows, stride_rows, padding,
1066        pad_rows_before, pad_rows_after, &output_rows));
1067    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
1068        c, in_cols_dim, filter_cols_dim, dilation_cols, stride_cols, padding,
1069        pad_cols_before, pad_cols_after, &output_cols));
1070
1071    ShapeHandle output_shape;
1072    if (data_format == FORMAT_NCHW) {
1073      output_shape =
1074          c->MakeShape({batch_size_dim, output_depth, output_rows, output_cols});
1075    } else {
1076      output_shape =
1077          c->MakeShape({batch_size_dim, output_rows, output_cols, output_depth});
1078    }
1079    c->set_output(0, output_shape);
1080    return Status::OK();
1081  }
1082
1083  };  // namespace
1084
1085  Status DepthwiseConv2DNativeShape(shape_inference::InferenceContext* c) {
1086    return DepthwiseConv2DNativeShapeImpl(c, false);
1087  }
1088
1089  Status DepthwiseConv2DNativeShapeWithExplicitPadding(
1090      shape_inference::InferenceContext* c) {
1091    return DepthwiseConv2DNativeShapeImpl(c, true);
1092  }
1093
1094  Status AvgPoolShape(shape_inference::InferenceContext* c) {
1095    string data_format_str;
1096    TensorFormat data_format;
1097    Status s = c->GetAttr("data_format", &data_format_str);
1098    if (s.ok()) {
1099      FormatFromString(data_format_str, &data_format);
1100    } else {
1101      data_format = FORMAT_NHWC;
1102    }
1103
1104    const int rank = (data_format == FORMAT_NCHW_VECT_C) ? 5 : 4;
1105    ShapeHandle input_shape;
1106    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &input_shape));
1107
```

```
  TF_RETURN_IF_ERROR(
      CheckFormatConstraintsOnShape(data_format, input_shape, "input", c));

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

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

  int32_t stride_rows = GetTensorDim(strides, data_format, 'H');
  int32_t stride_cols = GetTensorDim(strides, data_format, 'W');
  int32_t kernel_rows = GetTensorDim(kernel_sizes, data_format, 'H');
  int32_t kernel_cols = GetTensorDim(kernel_sizes, data_format, 'W');

  constexpr int num_spatial_dims = 2;
  DimensionHandle batch_size_dim = c->Dim(
      input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'N'));
  DimensionHandle in_rows_dim = c->Dim(
      input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'H'));
  DimensionHandle in_cols_dim = c->Dim(
      input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'W'));
  DimensionHandle depth_dim = c->Dim(
      input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'C'));

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

  // TODO(mrry,shlens): Raise an error if the stride would cause
  // information in the input to be ignored. This will require a change
  // in the kernel implementation.

  DimensionHandle output_rows, output_cols;
  TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
      c, in_rows_dim, kernel_rows, stride_rows, padding, &output_rows));
  TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
      c, in_cols_dim, kernel_cols, stride_cols, padding, &output_cols));

  ShapeHandle output_shape;
  TF_RETURN_IF_ERROR(MakeShapeFromFormat(data_format, batch_size_dim,
```

```
                                        {output_rows, output_cols}, depth_dim,
                                        &output_shape, c));
  c->set_output(0, output_shape);
  return Status::OK();
}

Status AvgPoolGradShape(shape_inference::InferenceContext* c) {
  ShapeHandle s;
  TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &s));
  TF_RETURN_IF_ERROR(c->WithRank(s, 4, &s));
  c->set_output(0, s);
  return Status::OK();
}

Status FusedBatchNormShape(shape_inference::InferenceContext* c) {
  string data_format_str;
  TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
  TensorFormat data_format;
  if (!FormatFromString(data_format_str, &data_format)) {
    return errors::InvalidArgument("Invalid data format string: ",
                                   data_format_str);
  }
  const int rank =
      (data_format_str == "NDHWC" || data_format_str == "NCDHW") ? 5 : 4;
  ShapeHandle x;
  TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &x));

  bool is_training;
  TF_RETURN_IF_ERROR(c->GetAttr("is_training", &is_training));
  float exponential_avg_factor;
  if (!c->GetAttr("exponential_avg_factor", &exponential_avg_factor).ok()) {
    exponential_avg_factor = 1.0f;  // default value
  }
  int number_inputs = (is_training && exponential_avg_factor == 1.0f) ? 3 : 5;

  int channel_dim_index = GetTensorFeatureDimIndex(rank, data_format);
  DimensionHandle channel_dim = c->Dim(x, channel_dim_index);

  // covers scale, offset, and if is_training is false, mean, variance
  for (int i = 1; i < number_inputs; ++i) {
    ShapeHandle vec;
    TF_RETURN_IF_ERROR(c->WithRank(c->input(i), 1, &vec));
    TF_RETURN_IF_ERROR(c->Merge(channel_dim, c->Dim(vec, 0), &channel_dim));
  }

  ShapeHandle y;
  TF_RETURN_IF_ERROR(c->ReplaceDim(x, channel_dim_index, channel_dim, &y));
  c->set_output(0, y);
  ShapeHandle vector_shape = c->Vector(channel_dim);
```

```
1206    c->set_output(1, vector_shape);
1207    c->set_output(2, vector_shape);
1208    c->set_output(3, vector_shape);
1209    c->set_output(4, vector_shape);
1210    return Status::OK();
1211  }
1212
1213  Status FusedBatchNormV3Shape(shape_inference::InferenceContext* c) {
1214    TF_RETURN_IF_ERROR(FusedBatchNormShape(c));
1215    c->set_output(5, c->UnknownShape());
1216    return Status::OK();
1217  }
1218
1219  Status FusedBatchNormExShape(shape_inference::InferenceContext* c) {
1220    TF_RETURN_IF_ERROR(FusedBatchNormV3Shape(c));
1221
1222    string data_format_str;
1223    TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
1224    TensorFormat data_format;
1225    if (!FormatFromString(data_format_str, &data_format)) {
1226      return errors::InvalidArgument("Invalid data format string: ",
1227                                     data_format_str);
1228    }
1229    ShapeHandle x;
1230    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &x));
1231
1232    int channel_dim_index = GetTensorFeatureDimIndex(4, data_format);
1233    DimensionHandle channel_dim = c->Dim(x, channel_dim_index);
1234
1235    // This is a cuDNN implementation constraint.
1236    if (c->ValueKnown(channel_dim) && c->Value(channel_dim) % 4 != 0) {
1237      return errors::InvalidArgument(
1238          "_FusedBatchNormEx channel dimension must be divisible by 4.");
1239    }
1240
1241    return Status::OK();
1242  }
1243
1244  Status FusedBatchNormGradShape(shape_inference::InferenceContext* c) {
1245    string data_format_str;
1246    TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
1247    TensorFormat data_format;
1248    if (!FormatFromString(data_format_str, &data_format)) {
1249      return errors::InvalidArgument("Invalid data format string: ",
1250                                     data_format_str);
1251    }
1252    const int rank =
1253        (data_format_str == "NDHWC" || data_format_str == "NCDHW") ? 5 : 4;
1254    ShapeHandle y_backprop;
```

```
1255    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &y_backprop));
1256    ShapeHandle x;
1257    TF_RETURN_IF_ERROR(c->WithRank(c->input(1), rank, &x));
1258
1259    bool is_training;
1260    TF_RETURN_IF_ERROR(c->GetAttr("is_training", &is_training));
1261
1262    int channel_dim_index = GetTensorFeatureDimIndex(rank, data_format);
1263    DimensionHandle channel_dim = c->Dim(y_backprop, channel_dim_index);
1264    TF_RETURN_IF_ERROR(
1265        c->Merge(channel_dim, c->Dim(x, channel_dim_index), &channel_dim));
1266
1267    // covers scale, mean (reserve_space_1), variance (reserve_space_2)
1268    for (int i = 2; i < 5; ++i) {
1269      ShapeHandle vec;
1270      TF_RETURN_IF_ERROR(c->WithRank(c->input(i), 1, &vec));
1271      TF_RETURN_IF_ERROR(c->Merge(channel_dim, c->Dim(vec, 0), &channel_dim));
1272    }
1273
1274    ShapeHandle x_backprop;
1275    TF_RETURN_IF_ERROR(
1276        c->ReplaceDim(y_backprop, channel_dim_index, channel_dim, &x_backprop));
1277    c->set_output(0, x_backprop);
1278    c->set_output(1, c->Vector(channel_dim));
1279    c->set_output(2, c->Vector(channel_dim));
1280    c->set_output(3, c->Vector(0));
1281    c->set_output(4, c->Vector(0));
1282    return Status::OK();
1283  }
1284
1285  Status FusedBatchNormGradExShape(shape_inference::InferenceContext* c) {
1286    TF_RETURN_IF_ERROR(FusedBatchNormGradShape(c));
1287
1288    int num_side_inputs;
1289    TF_RETURN_IF_ERROR(c->GetAttr("num_side_inputs", &num_side_inputs));
1290    if (num_side_inputs == 0) {
1291      return Status::OK();
1292    }
1293
1294    string data_format_str;
1295    TF_RETURN_IF_ERROR(c->GetAttr("data_format", &data_format_str));
1296    TensorFormat data_format;
1297    if (!FormatFromString(data_format_str, &data_format)) {
1298      return errors::InvalidArgument("Invalid data format string: ",
1299                                     data_format_str);
1300    }
1301    const int rank =
1302        (data_format_str == "NDHWC" || data_format_str == "NCDHW") ? 5 : 4;
1303    ShapeHandle y_backprop;
```

```
1304    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &y_backprop));
1305    ShapeHandle x;
1306    TF_RETURN_IF_ERROR(c->WithRank(c->input(1), rank, &x));
1307
1308    int channel_dim_index = GetTensorFeatureDimIndex(rank, data_format);
1309    DimensionHandle channel_dim = c->Dim(y_backprop, channel_dim_index);
1310    TF_RETURN_IF_ERROR(
1311        c->Merge(channel_dim, c->Dim(x, channel_dim_index), &channel_dim));
1312
1313    ShapeHandle side_input_backprop;
1314    TF_RETURN_IF_ERROR(c->ReplaceDim(y_backprop, channel_dim_index, channel_dim,
1315                                     &side_input_backprop));
1316
1317    c->set_output(5, side_input_backprop);
1318    return Status::OK();
1319  }
1320
1321  Status ReadDiagIndex(InferenceContext* c, const Tensor* diag_index_tensor,
1322                       int32* lower_diag_index, int32* upper_diag_index) {
1323    // This function assumes that the shape of diag_index_tensor is fully defined.
1324    if (diag_index_tensor->dims() == 0) {
1325      *lower_diag_index = diag_index_tensor->scalar<int32>()();
1326      *upper_diag_index = *lower_diag_index;
1327    } else {
1328      int32_t num_elements = diag_index_tensor->dim_size(0);
1329      if (num_elements == 1) {
1330        *lower_diag_index = diag_index_tensor->vec<int32>()(0);
1331        *upper_diag_index = *lower_diag_index;
1332      } else if (num_elements == 2) {
1333        *lower_diag_index = diag_index_tensor->vec<int32>()(0);
1334        *upper_diag_index = diag_index_tensor->vec<int32>()(1);
1335      } else {
1336        return errors::InvalidArgument(
1337            "diag_index must be a vector with one or two elements. It has ",
1338            num_elements, " elements.");
1339      }
1340    }
1341    return Status::OK();
1342  }
1343
1344  Status MatrixDiagPartV2Shape(shape_inference::InferenceContext* c) {
1345    ShapeHandle input_shape, diag_index_shape, unused_shape;
1346    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &input_shape));
1347    TF_RETURN_IF_ERROR(c->WithRankAtMost(c->input(1), 1, &diag_index_shape));
1348    TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused_shape));
1349
1350    const Tensor* diag_index_tensor = c->input_tensor(1);
1351    if (!c->RankKnown(input_shape) || !c->FullyDefined(diag_index_shape) ||
1352        diag_index_tensor == nullptr) {
```

```
1353      c->set_output(0, c->UnknownShape());
1354      return Status::OK();
1355    }
1356    int32_t lower_diag_index = 0;
1357    int32_t upper_diag_index = 0;
1358    TF_RETURN_IF_ERROR(ReadDiagIndex(c, diag_index_tensor, &lower_diag_index,
1359                                     &upper_diag_index));
1360    if (lower_diag_index > upper_diag_index) {
1361      return errors::InvalidArgument(
1362          "lower_diag_index is greater than upper_diag_index");
1363    }
1364
1365    // Validates lower_diag_index and upper_diag_index.
1366    const int32_t input_rank = c->Rank(input_shape);
1367    const int32_t num_rows = c->Value(c->Dim(input_shape, input_rank - 2));
1368    const int32_t num_cols = c->Value(c->Dim(input_shape, input_rank - 1));
1369    int32_t max_diag_len = InferenceContext::kUnknownDim;
1370    if (num_rows != InferenceContext::kUnknownDim &&
1371        num_cols != InferenceContext::kUnknownDim) {
1372      if (lower_diag_index != 0 &&  // For when num_rows or num_cols == 0.
1373          (-num_rows >= lower_diag_index || lower_diag_index >= num_cols)) {
1374        return errors::InvalidArgument("lower_diag_index is out of bound.");
1375      }
1376      if (upper_diag_index != 0 &&  // For when num_rows or num_cols == 0.
1377          (-num_rows >= upper_diag_index || upper_diag_index >= num_cols)) {
1378        return errors::InvalidArgument("upper_diag_index is out of bound.");
1379      }
1380      max_diag_len = std::min(num_rows + std::min(upper_diag_index, 0),
1381                              num_cols - std::max(lower_diag_index, 0));
1382    }
1383
1384    std::vector<DimensionHandle> dims;
1385    dims.reserve(input_rank - 2);
1386    for (int i = 0; i < input_rank - 2; ++i) {
1387      dims.push_back(c->Dim(input_shape, i));
1388    }
1389    if (lower_diag_index < upper_diag_index) {
1390      dims.push_back(c->MakeDim(upper_diag_index - lower_diag_index + 1));
1391    }
1392    dims.push_back(c->MakeDim(max_diag_len));
1393    c->set_output(0, c->MakeShape(dims));
1394    return Status::OK();
1395  }
1396
1397  Status MatrixDiagV2Shape(shape_inference::InferenceContext* c) {
1398    // Checks input ranks.
1399    ShapeHandle input_shape, diag_index_shape, unused_shape;
1400    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 1, &input_shape));
1401    TF_RETURN_IF_ERROR(c->WithRankAtMost(c->input(1), 1, &diag_index_shape));
```

```
1402    TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused_shape));
1403    TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 0, &unused_shape));
1404    TF_RETURN_IF_ERROR(c->WithRank(c->input(4), 0, &unused_shape));
1405
1406    // Reads the diagonal indices.
1407    const Tensor* diag_index_tensor = c->input_tensor(1);
1408    if (!c->RankKnown(input_shape) || !c->FullyDefined(diag_index_shape) ||
1409        diag_index_tensor == nullptr) {
1410      c->set_output(0, c->UnknownShape());
1411      return Status::OK();
1412    }
1413    int32_t lower_diag_index = 0;
1414    int32_t upper_diag_index = 0;
1415    TF_RETURN_IF_ERROR(ReadDiagIndex(c, diag_index_tensor, &lower_diag_index,
1416                                     &upper_diag_index));
1417    if (lower_diag_index > upper_diag_index) {
1418      return errors::InvalidArgument(
1419          "lower_diag_index is greater than upper_diag_index");
1420    }
1421
1422    // Checks if the number of diagonals provided matches what we imply from
1423    // lower_diag_index and upper_diag_index.
1424    const int32_t input_rank = c->Rank(input_shape);
1425    if (lower_diag_index < upper_diag_index) {
1426      const int32_t num_diags = c->Value(c->Dim(input_shape, input_rank - 2));
1427      const int32_t other_dim = c->Value(c->Dim(input_shape, input_rank - 1));
1428
1429      if (num_diags != (upper_diag_index - lower_diag_index + 1)) {
1430        return errors::InvalidArgument(
1431            "The number of rows of `diagonal` doesn't match the number of "
1432            "diagonals implied from `d_lower` and `d_upper`.\n",
1433            "num_diags = ", num_diags, ", d_lower = ", lower_diag_index,
1434            ", d_upper = ", upper_diag_index, " ", input_rank, " ", other_dim);
1435      }
1436    }
1437
1438    // Reads num_rows and num_cols.
1439    const Tensor* num_rows_tensor = c->input_tensor(2);
1440    const Tensor* num_cols_tensor = c->input_tensor(3);
1441    int64_t num_rows = -1;
1442    int64_t num_cols = -1;
1443    if (num_rows_tensor != nullptr) {
1444      TF_RETURN_IF_ERROR(c->GetScalarFromTensor(num_rows_tensor, &num_rows));
1445    }
1446    if (num_cols_tensor != nullptr) {
1447      TF_RETURN_IF_ERROR(c->GetScalarFromTensor(num_cols_tensor, &num_cols));
1448    }
1449
1450    // Infers the missing num_rows or num_cols: If both are missing, assume
```

```
1451    // output is square. Otherwise, use the smallest possible value. Also
1452    // validates the provided values.
1453    const int32_t max_diag_len = c->Value(c->Dim(input_shape, input_rank - 1));
1454    const int32_t min_num_rows = max_diag_len - std::min(upper_diag_index, 0);
1455    const int32_t min_num_cols = max_diag_len + std::max(lower_diag_index, 0);
1456    if (num_rows == -1 && num_cols == -1) {  // Special case.
1457      num_rows = std::max(min_num_rows, min_num_cols);
1458      num_cols = num_rows;
1459    }
1460    if (num_rows == -1) {
1461      num_rows = min_num_rows;
1462    } else if (num_rows < min_num_rows) {
1463      return errors::InvalidArgument("num_rows is too small");
1464    }
1465    if (num_cols == -1) {
1466      num_cols = min_num_cols;
1467    } else if (num_cols < min_num_cols) {
1468      return errors::InvalidArgument("num_cols is too small.");
1469    }
1470    // At least one of them must match the minimum length.
1471    if (num_rows != min_num_rows && num_cols != min_num_cols) {
1472      return errors::InvalidArgument(
1473          "num_rows and num_cols are not consistent with lower_diag_index, "
1474          "upper_diag_index, and the length of the given diagonals.\n",
1475          "num_rows = ", num_rows, " != min_num_rows = ", min_num_rows,
1476          ", num_cols = ", num_cols, " != min_num_cols = ", min_num_cols);
1477    }
1478
1479    // Sets output shape.
1480    ShapeHandle output_shape;
1481    const DimensionHandle output_row_dim = c->MakeDim(num_rows);
1482    const DimensionHandle output_col_dim = c->MakeDim(num_cols);
1483    if (lower_diag_index == upper_diag_index) {
1484      TF_RETURN_IF_ERROR(c->ReplaceDim(input_shape, input_rank - 1,
1485                                       output_row_dim, &output_shape));
1486      TF_RETURN_IF_ERROR(
1487          c->Concatenate(output_shape, c->Vector(output_col_dim), &output_shape));
1488    } else {
1489      TF_RETURN_IF_ERROR(c->ReplaceDim(input_shape, input_rank - 2,
1490                                       output_row_dim, &output_shape));
1491      TF_RETURN_IF_ERROR(c->ReplaceDim(output_shape, input_rank - 1,
1492                                       output_col_dim, &output_shape));
1493    }
1494    c->set_output(0, output_shape);
1495    return Status::OK();
1496  }
1497
1498  Status MatrixSetDiagV2Shape(shape_inference::InferenceContext* c) {
1499    ShapeHandle input_shape, diag_shape, diag_index_shape;
```

```
1500    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), 2, &input_shape));
1501    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 1, &diag_shape));
1502    TF_RETURN_IF_ERROR(c->WithRankAtMost(c->input(2), 1, &diag_index_shape));
1503
1504    int32_t lower_diag_index = 0;
1505    int32_t upper_diag_index = 0;
1506    bool diag_index_known = false;
1507    const Tensor* diag_index_tensor = c->input_tensor(2);
1508    if (diag_index_tensor != nullptr && c->FullyDefined(diag_index_shape)) {
1509      diag_index_known = true;
1510      TF_RETURN_IF_ERROR(ReadDiagIndex(c, diag_index_tensor, &lower_diag_index,
1511                                       &upper_diag_index));
1512      if (lower_diag_index > upper_diag_index) {
1513        return errors::InvalidArgument(
1514            "lower_diag_index is greater than upper_diag_index");
1515      }
1516    }
1517
1518    // Do more checks when input rank is known.
1519    if (c->RankKnown(input_shape)) {
1520      int32_t input_rank = c->Rank(input_shape);
1521
1522      // If diag_index is set, we know the exact rank of diagonal.
1523      if (diag_index_known) {
1524        TF_RETURN_IF_ERROR(c->WithRank(
1525            c->input(1),
1526            (lower_diag_index == upper_diag_index) ? input_rank - 1 : input_rank,
1527            &diag_shape));
1528      } else {
1529        TF_RETURN_IF_ERROR(
1530            c->WithRankAtLeast(c->input(1), input_rank - 1, &diag_shape));
1531        TF_RETURN_IF_ERROR(
1532            c->WithRankAtMost(c->input(1), input_rank, &diag_shape));
1533      }
1534
1535      // Validates lower_diag_index and upper_diag_index.
1536      const int32_t num_rows = c->Value(c->Dim(input_shape, input_rank - 2));
1537      const int32_t num_cols = c->Value(c->Dim(input_shape, input_rank - 1));
1538      if (num_rows != InferenceContext::kUnknownDim &&
1539          num_cols != InferenceContext::kUnknownDim) {
1540        if (lower_diag_index != 0 &&  // For when num_rows or num_cols == 0.
1541            (-num_rows >= lower_diag_index || lower_diag_index >= num_cols)) {
1542          return errors::InvalidArgument("lower_diag_index is out of bound.");
1543        }
1544        if (upper_diag_index != 0 &&  // For when num_rows or num_cols == 0.
1545            (-num_rows >= upper_diag_index || upper_diag_index >= num_cols)) {
1546          return errors::InvalidArgument("upper_diag_index is out of bound.");
1547        }
1548      }
```

```
1549    }
1550
1551    ShapeHandle output_shape = input_shape;
1552    if (c->RankKnown(diag_shape) && !c->FullyDefined(input_shape)) {
1553      // Try to infer parts of shape from diag.
1554      ShapeHandle diag_prefix;
1555      TF_RETURN_IF_ERROR(c->Subshape(
1556          diag_shape, 0, (lower_diag_index == upper_diag_index) ? -1 : -2,
1557          &diag_prefix));
1558
1559      // The inner matrices can be rectangular, so we can't pinpoint their
1560      // exact height and width by just lower_diag_index, upper_diag_index,
1561      // and the longest length of given diagonals.
1562      TF_RETURN_IF_ERROR(
1563          c->Concatenate(diag_prefix, c->UnknownShapeOfRank(2), &diag_shape));
1564      TF_RETURN_IF_ERROR(c->Merge(input_shape, diag_shape, &output_shape));
1565    }
1566    c->set_output(0, output_shape);
1567    return Status::OK();
1568  }
1569
1570  Status MaxPoolShapeImpl(shape_inference::InferenceContext* c,
1571                          bool supports_explicit_padding) {
1572    string data_format_str;
1573    TensorFormat data_format;
1574    Status s = c->GetAttr("data_format", &data_format_str);
1575    if (s.ok()) {
1576      FormatFromString(data_format_str, &data_format);
1577    } else {
1578      data_format = FORMAT_NHWC;
1579    }
1580
1581    const int rank = (data_format == FORMAT_NCHW_VECT_C) ? 5 : 4;
1582    ShapeHandle input_shape;
1583    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &input_shape));
1584
1585    TF_RETURN_IF_ERROR(
1586        CheckFormatConstraintsOnShape(data_format, input_shape, "input", c));
1587
1588    std::vector<int32> strides;
1589    TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
1590    if (strides.size() != 4) {
1591      return errors::InvalidArgument(
1592          "MaxPool requires the stride attribute to contain 4 values, but got: ",
1593          strides.size());
1594    }
1595
1596    std::vector<int32> kernel_sizes;
1597    TF_RETURN_IF_ERROR(c->GetAttr("ksize", &kernel_sizes));
```

```
1598    if (kernel_sizes.size() != 4) {
1599      return errors::InvalidArgument(
1600          "MaxPool requires the ksize attribute to contain 4 values, but got: ",
1601          kernel_sizes.size());
1602    }
1603
1604    int32_t stride_depth = GetTensorDim(strides, data_format, 'C');
1605    int32_t stride_rows = GetTensorDim(strides, data_format, 'H');
1606    int32_t stride_cols = GetTensorDim(strides, data_format, 'W');
1607    int32_t kernel_depth = GetTensorDim(kernel_sizes, data_format, 'C');
1608    int32_t kernel_rows = GetTensorDim(kernel_sizes, data_format, 'H');
1609    int32_t kernel_cols = GetTensorDim(kernel_sizes, data_format, 'W');
1610
1611    constexpr int num_spatial_dims = 2;
1612    DimensionHandle batch_size_dim = c->Dim(
1613        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'N'));
1614    DimensionHandle in_rows_dim = c->Dim(
1615        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'H'));
1616    DimensionHandle in_cols_dim = c->Dim(
1617        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'W'));
1618    DimensionHandle in_depth_dim = c->Dim(
1619        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'C'));
1620
1621    Padding padding;
1622    TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
1623
1624    std::vector<int64_t> explicit_paddings;
1625    if (supports_explicit_padding) {
1626      Status status = c->GetAttr("explicit_paddings", &explicit_paddings);
1627      // Use the default value, which is an empty list, if the attribute is not
1628      // found. Otherwise return the error to the caller.
1629      if (!status.ok() && !errors::IsNotFound(status)) {
1630        return status;
1631      }
1632      TF_RETURN_IF_ERROR(CheckValidPadding(padding, explicit_paddings,
1633                                           /*num_dims=*/4, data_format));
1634    } else {
1635      DCHECK(padding != Padding::EXPLICIT);
1636    }
1637
1638    ShapeHandle output_shape;
1639    DimensionHandle output_rows, output_cols, output_depth;
1640    int64_t pad_rows_before = -1, pad_rows_after = -1;
1641    int64_t pad_cols_before = -1, pad_cols_after = -1;
1642    if (padding == Padding::EXPLICIT) {
1643      GetExplicitPaddingForDim(explicit_paddings, data_format, 'H',
1644                               &pad_rows_before, &pad_rows_after);
1645      GetExplicitPaddingForDim(explicit_paddings, data_format, 'W',
1646                               &pad_cols_before, &pad_cols_after);
```

```
1647    }
1648    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
1649        c, in_rows_dim, kernel_rows, /*dilation_rate=*/1, stride_rows, padding,
1650        pad_rows_before, pad_rows_after, &output_rows));
1651    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
1652        c, in_cols_dim, kernel_cols, /*dilation_rate=*/1, stride_cols, padding,
1653        pad_cols_before, pad_cols_after, &output_cols));
1654    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDimsV2(
1655        c, in_depth_dim, kernel_depth, /*dilation_rate=*/1, stride_depth, padding,
1656        /*pad_before*/ 0, /*pad_after*/ 0, &output_depth));
1657
1658    TF_RETURN_IF_ERROR(MakeShapeFromFormat(data_format, batch_size_dim,
1659                                           {output_rows, output_cols},
1660                                           output_depth, &output_shape, c));
1661
1662    c->set_output(0, output_shape);
1663    return Status::OK();
1664  }
1665
1666  Status MaxPoolShape(shape_inference::InferenceContext* c) {
1667    return MaxPoolShapeImpl(c, /*supports_explicit_padding=*/false);
1668  }
1669
1670  Status MaxPoolGradShape(shape_inference::InferenceContext* c) {
1671    return UnchangedShapeWithRank(c, 4);
1672  }
1673
1674  Status MaxPoolShapeWithExplicitPadding(shape_inference::InferenceContext* c) {
1675    return MaxPoolShapeImpl(c, /*supports_explicit_padding=*/true);
1676  }
1677
1678  Status MaxPoolV2Shape(shape_inference::InferenceContext* c, int num_inputs) {
1679    string data_format_str;
1680    TensorFormat data_format;
1681    Status s = c->GetAttr("data_format", &data_format_str);
1682    if (s.ok()) {
1683      FormatFromString(data_format_str, &data_format);
1684    } else {
1685      data_format = FORMAT_NHWC;
1686    }
1687
1688    const int rank = (data_format == FORMAT_NCHW_VECT_C) ? 5 : 4;
1689    ShapeHandle input_shape;
1690    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), rank, &input_shape));
1691
1692    TF_RETURN_IF_ERROR(
1693        CheckFormatConstraintsOnShape(data_format, input_shape, "input", c));
1694
1695    std::vector<int32> kernel_sizes;
```

```
1696    std::vector<int32> strides;
1697
1698    if (c->num_inputs() + 2 == num_inputs) {
1699      TF_RETURN_IF_ERROR(c->GetAttr("ksize", &kernel_sizes));
1700
1701      TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
1702    } else {
1703      // Verify shape of ksize and strides input.
1704      ShapeHandle size;
1705      DimensionHandle unused;
1706      TF_RETURN_IF_ERROR(c->WithRank(c->input(c->num_inputs() - 2), 1, &size));
1707      TF_RETURN_IF_ERROR(c->WithValue(c->Dim(size, 0), 4, &unused));
1708      TF_RETURN_IF_ERROR(c->WithRank(c->input(c->num_inputs() - 1), 1, &size));
1709      TF_RETURN_IF_ERROR(c->WithValue(c->Dim(size, 0), 4, &unused));
1710
1711      const Tensor* kernel_sizes_tensor = c->input_tensor(c->num_inputs() - 2);
1712      if (kernel_sizes_tensor == nullptr) {
1713        c->set_output(0, c->UnknownShape());
1714        return Status::OK();
1715      }
1716      kernel_sizes.resize(kernel_sizes_tensor->shape().num_elements());
1717      auto kernel_sizes_vec = kernel_sizes_tensor->flat<int32>();
1718      std::copy_n(&kernel_sizes_vec(0), kernel_sizes.size(),
1719                  kernel_sizes.begin());
1720
1721      const Tensor* strides_tensor = c->input_tensor(c->num_inputs() - 1);
1722      if (strides_tensor == nullptr) {
1723        c->set_output(0, c->UnknownShape());
1724        return Status::OK();
1725      }
1726      strides.resize(strides_tensor->shape().num_elements());
1727      auto strides_vec = strides_tensor->flat<int32>();
1728      std::copy_n(&strides_vec(0), strides.size(), strides.begin());
1729    }
1730
1731    if (strides.size() != 4) {
1732      return errors::InvalidArgument(
1733          "MaxPool requires the stride attribute to contain 4 values, but "
1734          "got: ",
1735          strides.size());
1736    }
1737    if (kernel_sizes.size() != 4) {
1738      return errors::InvalidArgument(
1739          "MaxPool requires the ksize attribute to contain 4 values, but got: ",
1740          kernel_sizes.size());
1741    }
1742
1743    int32_t stride_depth = GetTensorDim(strides, data_format, 'C');
1744    int32_t stride_rows = GetTensorDim(strides, data_format, 'H');
```

```
1745    int32_t stride_cols = GetTensorDim(strides, data_format, 'W');
1746    int32_t kernel_depth = GetTensorDim(kernel_sizes, data_format, 'C');
1747    int32_t kernel_rows = GetTensorDim(kernel_sizes, data_format, 'H');
1748    int32_t kernel_cols = GetTensorDim(kernel_sizes, data_format, 'W');
1749
1750    constexpr int num_spatial_dims = 2;
1751    DimensionHandle batch_size_dim = c->Dim(
1752        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'N'));
1753    DimensionHandle in_rows_dim = c->Dim(
1754        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'H'));
1755    DimensionHandle in_cols_dim = c->Dim(
1756        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'W'));
1757    DimensionHandle in_depth_dim = c->Dim(
1758        input_shape, GetTensorDimIndex<num_spatial_dims>(data_format, 'C'));
1759
1760    Padding padding;
1761    TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
1762
1763    ShapeHandle output_shape;
1764    DimensionHandle output_rows, output_cols, output_depth;
1765    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
1766        c, in_rows_dim, kernel_rows, stride_rows, padding, &output_rows));
1767    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
1768        c, in_cols_dim, kernel_cols, stride_cols, padding, &output_cols));
1769    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
1770        c, in_depth_dim, kernel_depth, stride_depth, padding, &output_depth));
1771
1772    TF_RETURN_IF_ERROR(MakeShapeFromFormat(data_format, batch_size_dim,
1773                                           {output_rows, output_cols},
1774                                           output_depth, &output_shape, c));
1775
1776    c->set_output(0, output_shape);
1777    return Status::OK();
1778  }
1779
1780  Status Pool3DShape(shape_inference::InferenceContext* c) {
1781    ShapeHandle input_shape;
1782    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 5, &input_shape));
1783
1784    string data_format;
1785    Status s = c->GetAttr("data_format", &data_format);
1786
1787    std::vector<int32> strides;
1788    TF_RETURN_IF_ERROR(c->GetAttr("strides", &strides));
1789    if (strides.size() != 5) {
1790      return errors::InvalidArgument(
1791          "Pool3D ops require the stride attribute to contain 5 values, but "
1792          "got: ",
1793          strides.size());
```

```
1794    }
1795
1796    std::vector<int32> kernel_sizes;
1797    TF_RETURN_IF_ERROR(c->GetAttr("ksize", &kernel_sizes));
1798    if (kernel_sizes.size() != 5) {
1799      return errors::InvalidArgument(
1800          "Pool3D requires the ksize attribute to contain 5 values, but got: ",
1801          kernel_sizes.size());
1802    }
1803
1804    int32_t stride_planes, stride_rows, stride_cols;
1805    int32_t kernel_planes, kernel_rows, kernel_cols;
1806
1807    if (s.ok() && data_format == "NCDHW") {
1808      // Convert input_shape to NDHWC.
1809      auto dim = [&](char dimension) {
1810        return c->Dim(input_shape, GetTensorDimIndex<3>(FORMAT_NCHW, dimension));
1811      };
1812      input_shape =
1813          c->MakeShape({{dim('N'), dim('0'), dim('1'), dim('2'), dim('C')}});
1814      stride_planes = strides[2];
1815      stride_rows = strides[3];
1816      stride_cols = strides[4];
1817      kernel_planes = kernel_sizes[2];
1818      kernel_rows = kernel_sizes[3];
1819      kernel_cols = kernel_sizes[4];
1820    } else {
1821      stride_planes = strides[1];
1822      stride_rows = strides[2];
1823      stride_cols = strides[3];
1824      kernel_planes = kernel_sizes[1];
1825      kernel_rows = kernel_sizes[2];
1826      kernel_cols = kernel_sizes[3];
1827    }
1828
1829    DimensionHandle batch_size_dim = c->Dim(input_shape, 0);
1830    DimensionHandle in_planes_dim = c->Dim(input_shape, 1);
1831    DimensionHandle in_rows_dim = c->Dim(input_shape, 2);
1832    DimensionHandle in_cols_dim = c->Dim(input_shape, 3);
1833    DimensionHandle output_depth_dim = c->Dim(input_shape, 4);
1834
1835    Padding padding;
1836    TF_RETURN_IF_ERROR(c->GetAttr("padding", &padding));
1837
1838    // TODO(mrry,shlens): Raise an error if the stride would cause
1839    // information in the input to be ignored. This will require a change
1840    // in the kernel implementation.
1841    DimensionHandle output_planes, output_rows, output_cols;
1842    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
```

```
1843        c, in_planes_dim, kernel_planes, stride_planes, padding, &output_planes));
1844    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
1845        c, in_rows_dim, kernel_rows, stride_rows, padding, &output_rows));
1846    TF_RETURN_IF_ERROR(GetWindowedOutputSizeFromDims(
1847        c, in_cols_dim, kernel_cols, stride_cols, padding, &output_cols));
1848
1849    ShapeHandle output_shape;
1850    if (data_format == "NCDHW") {
1851      output_shape = c->MakeShape({batch_size_dim, output_depth_dim,
1852                                   output_planes, output_rows, output_cols});
1853    } else {
1854      output_shape = c->MakeShape({batch_size_dim, output_planes, output_rows,
1855                                   output_cols, output_depth_dim});
1856    }
1857
1858    c->set_output(0, output_shape);
1859    return Status::OK();
1860  }
1861
1862  Status MaxPool3DGradShape(shape_inference::InferenceContext* c) {
1863    return UnchangedShapeWithRank(c, 5);
1864  }
1865
1866  Status AvgPool3DGradShape(shape_inference::InferenceContext* c) {
1867    ShapeHandle s;
1868    TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &s));
1869    TF_RETURN_IF_ERROR(c->WithRank(s, 5, &s));
1870    c->set_output(0, s);
1871    return Status::OK();
1872  }
1873
1874  Status UnknownShape(shape_inference::InferenceContext* c) {
1875    for (int i = 0; i < c->num_outputs(); ++i) {
1876      c->set_output(i, c->UnknownShape());
1877    }
1878    return Status::OK();
1879  }
1880
1881  template <typename T>
1882  Status ReductionShapeHelper(const Tensor* reduction_indices_t,
1883                              const int32_t input_rank,
1884                              std::set<int64_t>* true_indices) {
1885    auto reduction_indices = reduction_indices_t->flat<T>();
1886    for (int i = 0; i < reduction_indices_t->NumElements(); ++i) {
1887      const T reduction_index = reduction_indices(i);
1888      if (reduction_index < -input_rank || reduction_index >= input_rank) {
1889        return errors::InvalidArgument("Invalid reduction dimension ",
1890                                       reduction_index, " for input with ",
1891                                       input_rank, " dimensions.");
```

```
1892      }
1893
1894      auto wrapped_index = reduction_index;
1895      if (wrapped_index < 0) {
1896        wrapped_index += input_rank;
1897      }
1898
1899      true_indices->insert(wrapped_index);
1900    }
1901    return Status::OK();
1902  }
1903
1904  Status ReductionShape(InferenceContext* c) {
1905    ShapeHandle input = c->input(0);
1906
1907    ShapeHandle indices;
1908    // Older versions of TensorFlow accidentally allowed higher rank tensors like
1909    // [[1,2]] or [[1],[2]] to represent axis=[1,2].
1910    if (c->graph_def_version() < 21) {
1911      indices = c->input(1);
1912    } else {
1913      TF_RETURN_IF_ERROR(c->WithRankAtMost(c->input(1), 1, &indices));
1914    }
1915
1916    bool keep_dims;
1917    TF_RETURN_IF_ERROR(c->GetAttr("keep_dims", &keep_dims));
1918
1919    const Tensor* reduction_indices_t = c->input_tensor(1);
1920    if (reduction_indices_t == nullptr || !c->RankKnown(input)) {
1921      // If we do not have the reduction values at runtime, or the
1922      // rank of the input, we don't know the output shape.
1923
1924      if (keep_dims && c->RankKnown(input)) {
1925        // output rank matches input input if <keep_dims>.
1926        c->set_output(0, c->UnknownShapeOfRank(c->Rank(input)));
1927        return Status::OK();
1928      } else {
1929        return shape_inference::UnknownShape(c);
1930      }
1931    }
1932
1933    const int32_t input_rank = c->Rank(input);
1934    std::set<int64_t> true_indices;
1935    if (reduction_indices_t->dtype() == DataType::DT_INT32) {
1936      TF_RETURN_IF_ERROR(ReductionShapeHelper<int32>(reduction_indices_t,
1937                                                     input_rank, &true_indices));
1938    } else if (reduction_indices_t->dtype() == DataType::DT_INT64) {
1939      TF_RETURN_IF_ERROR(ReductionShapeHelper<int64_t>(
1940          reduction_indices_t, input_rank, &true_indices));
```

```
1941    } else {
1942      return errors::InvalidArgument(
1943          "reduction_indices can only be int32 or int64");
1944    }
1945
1946    std::vector<DimensionHandle> dims;
1947    for (int i = 0; i < input_rank; ++i) {
1948      if (true_indices.count(i) > 0) {
1949        if (keep_dims) {
1950          dims.emplace_back(c->MakeDim(1));
1951        }
1952      } else {
1953        dims.emplace_back(c->Dim(input, i));
1954      }
1955    }
1956
1957    c->set_output(0, c->MakeShape(dims));
1958    return Status::OK();
1959  }
1960
1961  Status ConcatShapeHelper(InferenceContext* c, int start_value_index,
1962                           int end_value_index, int dim_index) {
1963    ShapeHandle unused;
1964    TF_RETURN_IF_ERROR(c->WithRank(c->input(dim_index), 0, &unused));
1965    const Tensor* concat_dim_t = c->input_tensor(dim_index);
1966    if (concat_dim_t == nullptr) {
1967      // Return an unknown shape with same rank as inputs, or an unknown rank
1968      // if no input's rank is known.
1969
1970      // Find rank.
1971      int32_t rank = InferenceContext::kUnknownRank;
1972      for (int i = start_value_index; i < end_value_index; ++i) {
1973        if (rank == InferenceContext::kUnknownRank) rank = c->Rank(c->input(i));
1974        if (rank != InferenceContext::kUnknownRank) {
1975          break;
1976        }
1977      }
1978      if (rank == InferenceContext::kUnknownRank) {
1979        c->set_output(0, c->UnknownShape());
1980        return Status::OK();
1981      } else if (rank == 0) {
1982        return errors::InvalidArgument(
1983            "Can't concatenate scalars (use tf.stack instead)");
1984      } else {
1985        for (int i = start_value_index; i < end_value_index; ++i) {
1986          // Check that all the inputs are of the correct rank.
1987          TF_RETURN_IF_ERROR(c->WithRank(c->input(i), rank, &unused));
1988        }
1989      }
```

```
1990      // Build result of <rank> different unknown dims.
1991      std::vector<DimensionHandle> dims;
1992      dims.reserve(rank);
1993      for (int i = 0; i < rank; ++i) dims.push_back(c->UnknownDim());
1994      c->set_output(0, c->MakeShape(dims));
1995      return Status::OK();
1996    }
1997
1998    // Merge all the non-concat dims, and sum the concat dim to make an output
1999    // shape.
2000    int64_t concat_dim;
2001    if (concat_dim_t->dtype() == DT_INT32) {
2002      concat_dim = static_cast<int64_t>(concat_dim_t->flat<int32>()(0));
2003    } else {
2004      concat_dim = concat_dim_t->flat<int64_t>()(0);
2005    }
2006
2007    // Minimum required number of dimensions.
2008    const int min_rank = concat_dim < 0 ? -concat_dim : concat_dim + 1;
2009
2010    ShapeHandle output_before;
2011    ShapeHandle output_after;
2012
2013    ShapeHandle input = c->input(end_value_index - 1);
2014    TF_RETURN_IF_ERROR(c->WithRankAtLeast(input, min_rank, &input));
2015    TF_RETURN_IF_ERROR(c->Subshape(input, 0, concat_dim, &output_before));
2016    DimensionHandle output_middle = c->Dim(input, concat_dim);
2017    if (concat_dim == -1) {
2018      output_after = c->Scalar();  // no dimensions.
2019    } else {
2020      TF_RETURN_IF_ERROR(c->Subshape(input, concat_dim + 1, &output_after));
2021    }
2022
2023    for (int i = end_value_index - 2; i >= start_value_index; --i) {
2024      ShapeHandle before;
2025      ShapeHandle after;
2026      input = c->input(i);
2027      TF_RETURN_IF_ERROR(c->WithRankAtLeast(input, min_rank, &input));
2028      TF_RETURN_IF_ERROR(c->Subshape(input, 0, concat_dim, &before));
2029      DimensionHandle middle = c->Dim(input, concat_dim);
2030      if (concat_dim == -1) {
2031        after = c->Scalar();
2032      } else {
2033        TF_RETURN_IF_ERROR(c->Subshape(input, concat_dim + 1, &after));
2034      }
2035
2036      TF_RETURN_IF_ERROR(c->Merge(before, output_before, &output_before));
2037      TF_RETURN_IF_ERROR(c->Add(output_middle, middle, &output_middle));
2038      TF_RETURN_IF_ERROR(c->Merge(after, output_after, &output_after));
```

```
2039    }
2040
2041    ShapeHandle s;
2042    TF_RETURN_IF_ERROR(
2043        c->Concatenate(output_before, c->Vector(output_middle), &s));
2044    TF_RETURN_IF_ERROR(c->Concatenate(s, output_after, &s));
2045    c->set_output(0, s);
2046    return Status::OK();
2047  }
2048
2049  Status ConcatShape(InferenceContext* c, int num_inputs_to_concat) {
2050    return ConcatShapeHelper(c, 1 /* start_value_index */,
2051                             1 + num_inputs_to_concat /* end_value_index */,
2052                             0 /* dim_index */);
2053  }
2054
2055  Status ConcatV2Shape(InferenceContext* c) {
2056    return ConcatShapeHelper(c, 0 /* start_value_index */,
2057                             c->num_inputs() - 1 /* end_value_index */,
2058                             c->num_inputs() - 1 /* dim_index */);
2059  }
2060
2061  Status QuantizedConcatV2Shape(InferenceContext* c, int num_inputs_to_concat) {
2062    return ConcatShapeHelper(c, 0 /* start_value_index */,
2063                             num_inputs_to_concat /* end_value_index */,
2064                             num_inputs_to_concat /* dim_index */);
2065  }
2066
2067  Status BroadcastBinaryOpOutputShapeFnHelper(InferenceContext* c,
2068                                              ShapeHandle shape_x,
2069                                              ShapeHandle shape_y,
2070                                              bool incompatible_shape_error,
2071                                              ShapeHandle* out) {
2072    CHECK_NOTNULL(out);
2073    if (!c->RankKnown(shape_x) || !c->RankKnown(shape_y)) {
2074      *out = c->UnknownShape();
2075      return Status::OK();
2076    }
2077    const int32_t rank_x = c->Rank(shape_x);
2078    const int32_t rank_y = c->Rank(shape_y);
2079    const int32_t rank_out = std::max(rank_x, rank_y);
2080
2081    // To compute the broadcast dimensions, we zip together shape_x and shape_y
2082    // and
2083    // pad with 1 to make them the same length.
2084    std::vector<DimensionHandle> dims;
2085    DimensionHandle dim_one;
2086    if (rank_x != rank_y) dim_one = c->MakeDim(1);
2087    for (int i = 0; i < rank_out; ++i) {
```

```
2088      const auto dim_x = i < (rank_out - rank_x)
2089                             ? dim_one
2090                             : c->Dim(shape_x, i - (rank_out - rank_x));
2091      const bool dim_y_is_one = (i < (rank_out - rank_y));
2092      const auto dim_y =
2093          dim_y_is_one ? dim_one : c->Dim(shape_y, i - (rank_out - rank_y));
2094      if (!c->ValueKnown(dim_x) || !c->ValueKnown(dim_y)) {
2095        // One or both dimensions is unknown.
2096        //
2097        // - If either dimension is greater than 1, we assume that the program is
2098        // correct, and the other dimension will be broadcast to match it.
2099        // TODO(cwhipkey): For shape inference, if we eliminate the shape checks
2100        // in C++ op code, we must still assert that the unknown dim is either 1
2101        // or the same as the known dim.
2102        // - If either dimension is 1, the other dimension is the output.
2103        // - If both are unknown then dimension is unknown
2104        if (c->Value(dim_x) > 1) {
2105          if (!incompatible_shape_error) {
2106            *out = c->UnknownShape();
2107            return Status::OK();
2108          }
2109          dims.push_back(dim_x);
2110        } else if (c->Value(dim_y) > 1) {
2111          if (!incompatible_shape_error) {
2112            *out = c->UnknownShape();
2113            return Status::OK();
2114          }
2115          dims.push_back(dim_y);
2116        } else if (c->Value(dim_x) == 1) {
2117          dims.push_back(dim_y);
2118        } else if (c->Value(dim_y) == 1) {
2119          dims.push_back(dim_x);
2120        } else if (dim_y.SameHandle(dim_x)) {
2121          dims.push_back(dim_x);
2122        } else if (!c->ValueKnown(dim_x) && !c->ValueKnown(dim_y)) {
2123          dims.push_back(c->UnknownDim());
2124        } else {
2125          if (!incompatible_shape_error) {
2126            *out = c->UnknownShape();
2127            return Status::OK();
2128          }
2129          dims.push_back(c->UnknownDim());
2130        }
2131      } else if (c->Value(dim_x) == 1 || c->Value(dim_y) == 1) {
2132        if (c->Value(dim_x) == 1 && !dim_y_is_one) {
2133          // We will broadcast dim_x to dim_y.
2134          dims.push_back(dim_y);
2135        } else {
2136          DCHECK_EQ(c->Value(dim_y), 1);
```

```
2137          // We will broadcast dim_y to dim_x.
2138          dims.push_back(dim_x);
2139        }
2140      } else {
2141        DimensionHandle dim;
2142        Status s = c->Merge(dim_x, dim_y, &dim);
2143        if (!s.ok()) {
2144          if (!incompatible_shape_error) {
2145            *out = c->MakeShape({});
2146            return Status::OK();
2147          }
2148          return s;
2149        }
2150        dims.push_back(dim);
2151      }
2152    }
2153
2154    *out = c->MakeShape(dims);
2155    return Status::OK();
2156  }
2157
2158  Status RandomShape(shape_inference::InferenceContext* c) {
2159    shape_inference::ShapeHandle out;
2160    TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(0, &out));
2161    c->set_output(0, out);
2162    return Status::OK();
2163  }
2164
2165  Status UnsortedSegmentReductionShapeFn(InferenceContext* c) {
2166    ShapeHandle s_data = c->input(0);
2167    ShapeHandle s_segment_ids = c->input(1);
2168    ShapeHandle s_num_segments = c->input(2);
2169    TF_RETURN_IF_ERROR(c->WithRank(s_num_segments, 0, &s_num_segments));
2170
2171    ShapeHandle out;
2172
2173    // Leading dimensions of data must be compatible with dimensions of
2174    // <s_segment_ids>.
2175    if (c->RankKnown(s_segment_ids)) {
2176      TF_RETURN_IF_ERROR(
2177          c->MergePrefix(s_data, s_segment_ids, &s_data, &s_segment_ids));
2178
2179      // Get the value of the num_segments input tensor.
2180      DimensionHandle num_segments_dim;
2181      TF_RETURN_IF_ERROR(c->MakeDimForScalarInput(2, &num_segments_dim));
2182
2183      // Output is {segment_id_rank} + s_data[segment_id_rank:].
2184      ShapeHandle s_data_suffix;
2185      TF_RETURN_IF_ERROR(
```

```
2186          c->Subshape(s_data, c->Rank(s_segment_ids), &s_data_suffix));
2187      TF_RETURN_IF_ERROR(
2188          c->Concatenate(c->Vector(num_segments_dim), s_data_suffix, &out));
2189    } else {
2190      out = c->UnknownShape();
2191    }
2192    c->set_output(0, out);
2193    return Status::OK();
2194  }
2195
2196  namespace {
2197
2198  // This SliceHelper processes the output shape of the `slice`
2199  // when the tensor of `sizes` is available.
2200  template <typename T>
2201  Status SliceHelper(InferenceContext* c, ShapeHandle begin_value,
2202                     const Tensor* sizes_value,
2203                     std::vector<DimensionHandle>* dims) {
2204    auto sizes_vec = sizes_value->vec<T>();
2205    for (int i = 0; i < sizes_value->NumElements(); ++i) {
2206      DimensionHandle dim = c->Dim(c->input(0), i);
2207      if (sizes_vec(i) != -1) {
2208        auto dim_val = c->Value(dim);
2209        if (sizes_vec(i) < 0) {
2210          return errors::InvalidArgument(
2211              "Out of bounds slicing on dimension ", i, " of length ", dim_val,
2212              ": sizes vector cannot be < -1, but was ", sizes_vec(i));
2213        }
2214
2215        dims->emplace_back(c->MakeDim(sizes_vec(i)));
2216      } else {
2217        DimensionHandle result;
2218        TF_RETURN_IF_ERROR(c->Subtract(dim, c->Dim(begin_value, i), &result));
2219        dims->emplace_back(result);
2220      }
2221    }
2222
2223    return Status::OK();
2224  }
2225  }  // namespace
2226
2227  Status SliceShape(InferenceContext* c) {
2228    ShapeHandle input = c->input(0);
2229    ShapeHandle begin_shape;
2230    TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 1, &begin_shape));
2231    ShapeHandle sizes_shape;
2232    TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &sizes_shape));
2233
2234    // Merge to check compatibility of begin and sizes tensors.
```

```
2235    TF_RETURN_IF_ERROR(c->Merge(begin_shape, sizes_shape, &begin_shape));
2236
2237    DimensionHandle ndims = c->Dim(begin_shape, 0);
2238    if (c->ValueKnown(ndims)) {
2239      TF_RETURN_IF_ERROR(c->WithRank(input, c->Value(ndims), &input));
2240    }
2241
2242    // NOTE(mrry): Use MakeShapeFromShapeTensor to handle partially-known
2243    // values, even though the `begin` value does not represent a shape.
2244    ShapeHandle begin_value;
2245    TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(1, &begin_value));
2246
2247    // We check the tensor value here and will only use
2248    // `MakeShapeFromShapeTensor` when `sizes_value` is null.
2249    // The reason is that `sizes` might contain -1, which can't
2250    // be represented (-1 in the ShapeHandle would mean "unknown").
2251    const Tensor* sizes_value = c->input_tensor(2);
2252
2253    if (sizes_value != nullptr) {
2254      TF_RETURN_IF_ERROR(
2255          c->WithRank(begin_value, sizes_value->NumElements(), &begin_value));
2256      std::vector<DimensionHandle> dims;
2257      // If the begin and sizes tensors are available, then
2258      // we can be precise about the shape of the output.
2259      if (sizes_value->dtype() == DT_INT64) {
2260        TF_RETURN_IF_ERROR(
2261            SliceHelper<int64_t>(c, begin_value, sizes_value, &dims));
2262      } else {
2263        TF_RETURN_IF_ERROR(
2264            SliceHelper<int32>(c, begin_value, sizes_value, &dims));
2265      }
2266      c->set_output(0, c->MakeShape(dims));
2267      return Status::OK();
2268    } else {
2269      // In case `sizes` is not available (`sizes_value` is null),
2270      // we could try to use `MakeShapeFromShapeTensor` here.
2271      // If sizes contain -1, we will simply consider it as `Unknown`.
2272      // This is less than ideal but still an improvement of shape inference.
2273      // The following is an example that returns [None, 1, None] with this
2274      // code path:
2275      //   z = tf.zeros((1, 2, 3))
2276      //   m = tf.slice(z, [0, 0, 0], [tf.constant(1) + 0, 1, -1])
2277      //   m.get_shape().as_list()
2278      ShapeHandle sizes_value;
2279      TF_RETURN_IF_ERROR(c->MakeShapeFromShapeTensor(2, &sizes_value));
2280      if (c->RankKnown(sizes_value)) {
2281        TF_RETURN_IF_ERROR(
2282            c->WithRank(begin_value, c->Rank(sizes_value), &begin_value));
2283        std::vector<DimensionHandle> dims;
```

```
2284        dims.reserve(c->Rank(sizes_value));
2285        for (int i = 0; i < c->Rank(sizes_value); ++i) {
2286          dims.emplace_back(c->Dim(sizes_value, i));
2287        }
2288        c->set_output(0, c->MakeShape(dims));
2289        return Status::OK();
2290      }
2291      // We might know the rank of the input.
2292      if (c->RankKnown(input)) {
2293        c->set_output(0, c->UnknownShapeOfRank(c->Rank(input)));
2294        return Status::OK();
2295      } else {
2296        return shape_inference::UnknownShape(c);
2297      }
2298    }
2299
2300    return Status::OK();
2301  }
2302
2303  Status ValidateSparseTensor(InferenceContext* c, ShapeHandle indices_shape,
2304                              ShapeHandle values_shape, ShapeHandle shape_shape) {
2305    // Validate ranks.
2306    ShapeHandle unused_shape;
2307    TF_RETURN_IF_ERROR(c->WithRank(indices_shape, 2, &unused_shape));
2308    TF_RETURN_IF_ERROR(c->WithRank(values_shape, 1, &unused_shape));
2309    TF_RETURN_IF_ERROR(c->WithRank(shape_shape, 1, &unused_shape));
2310
2311    // Number of elements in indices and values must match.
2312    DimensionHandle num_index_elements_dim = c->Dim(indices_shape, 0);
2313    if (c->ValueKnown(num_index_elements_dim)) {
2314      DimensionHandle num_values_elements_dim = c->Dim(values_shape, 0);
2315      if (c->ValueKnown(num_values_elements_dim)) {
2316        int64_t num_index_elements = c->Value(num_index_elements_dim);
2317        int64_t num_values_elements = c->Value(num_values_elements_dim);
2318        if (num_index_elements != num_values_elements) {
2319          return errors::InvalidArgument("Number of elements in index (",
2320                                         num_index_elements, ") and values (",
2321                                         num_values_elements, ") do not match.");
2322        }
2323      }
2324    }
2325
2326    // Rank embedded in indices must match shape.
2327    DimensionHandle index_rank_dim = c->Dim(indices_shape, 1);
2328    if (c->ValueKnown(index_rank_dim)) {
2329      DimensionHandle shape_rank_dim = c->Dim(shape_shape, 0);
2330      if (c->ValueKnown(shape_rank_dim)) {
2331        int64_t index_rank = c->Value(index_rank_dim);
2332        int32_t shape_rank = c->Value(shape_rank_dim);
```

```
2333        if (index_rank != shape_rank) {
2334          return errors::InvalidArgument("Index rank (", index_rank,
2335                                         ") and shape rank (", shape_rank,
2336                                         ") do not match.");
2337        }
2338      }
2339    }
2340
2341    return Status::OK();
2342  }
2343
2344  Status ValidateVariableResourceHandle(
2345      InferenceContext* c, std::vector<ShapeAndType>* shape_and_type) {
2346    auto* handle_data = c->input_handle_shapes_and_types(0);
2347    if (handle_data == nullptr || handle_data->empty()) {
2348      shape_and_type->emplace_back(c->UnknownShape(), DT_INVALID);
2349    } else {
2350      *shape_and_type = *handle_data;
2351      DataType value_dtype;
2352      TF_RETURN_IF_ERROR(c->GetAttr("dtype", &value_dtype));
2353      if (shape_and_type->at(0).dtype != value_dtype) {
2354        return errors::InvalidArgument(
2355            "Trying to read variable with wrong dtype. "
2356            "Expected ",
2357            DataTypeString(shape_and_type->at(0).dtype), " got ",
2358            DataTypeString(value_dtype));
2359      }
2360    }
2361    return Status::OK();
2362  }
2363
2364  Status GatherNdShape(InferenceContext* c) {
2365    ShapeHandle params;
2366    std::vector<ShapeAndType> handle_shape_and_type;
2367    if (c->input_handle_shapes_and_types(0) != nullptr) {
2368      TF_RETURN_IF_ERROR(
2369          ValidateVariableResourceHandle(c, &handle_shape_and_type));
2370      params = handle_shape_and_type[0].shape;
2371    } else {
2372      params = c->input(0);
2373    }
2374    ShapeHandle indices;
2375    TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(1), 1, &indices));
2376    DimensionHandle r_dim = c->Dim(indices, -1);
2377
2378    if (!c->RankKnown(params) || !c->ValueKnown(r_dim)) {
2379      c->set_output(0, c->UnknownShape());
2380      return Status::OK();
2381    }
```

```
2382
2383    if (c->Value(r_dim) > c->Rank(params)) {
2384      return errors::InvalidArgument(
2385          "indices.shape[-1] must be <= params.rank, but saw indices shape: ",
2386          c->DebugString(indices), " and params shape: ", c->DebugString(params));
2387    }
2388
2389    // Remove r_dim from indices to get output.
2390    ShapeHandle indices_slice;
2391    ShapeHandle params_slice;
2392    TF_RETURN_IF_ERROR(c->Subshape(indices, 0, -1, &indices_slice));
2393    TF_RETURN_IF_ERROR(c->Subshape(params, c->Value(r_dim), &params_slice));
2394    ShapeHandle out;
2395    TF_RETURN_IF_ERROR(c->Concatenate(indices_slice, params_slice, &out));
2396    c->set_output(0, out);
2397    return Status::OK();
2398  }
2399
2400  Status ScatterNdShapeHelper(InferenceContext* c, ShapeHandle indices_shape,
2401                              ShapeHandle updates_shape,
2402                              ShapeHandle input_shape) {
2403    if (c->Value(c->NumElements(input_shape)) == 0 &&
2404        (c->Value(c->NumElements(indices_shape)) > 0 ||
2405         c->Value(c->NumElements(updates_shape)) > 0)) {
2406      return errors::InvalidArgument(
2407          "Indices and updates specified for empty input");
2408    }
2409
2410    if (c->RankKnown(indices_shape) && c->RankKnown(updates_shape)) {
2411      const int64_t outer_dims = c->Rank(indices_shape) - 1;
2412      const DimensionHandle ixdim = c->Dim(indices_shape, -1);
2413
2414      // We can only do more validation if the last dimension of indices
2415      // is a known value.
2416      if (c->ValueKnown(ixdim)) {
2417        int64_t ix = c->Value(ixdim);
2418        ShapeHandle unused;
2419        ShapeHandle prefix_indices;
2420        TF_RETURN_IF_ERROR(
2421            c->Subshape(indices_shape, 0, outer_dims, &prefix_indices));
2422        ShapeHandle prefix_updates;
2423        TF_RETURN_IF_ERROR(
2424            c->Subshape(updates_shape, 0, outer_dims, &prefix_updates));
2425
2426        Status s = c->Merge(prefix_indices, prefix_updates, &unused);
2427        if (!s.ok()) {
2428          return errors::InvalidArgument(
2429              "Dimensions [0,", outer_dims,
2430              ") of indices[shape=", c->DebugString(indices_shape),
```

```
2431              "] = ", c->DebugString(prefix_indices),
2432              " must match dimensions [0,", outer_dims,
2433              ") of updates[shape=", c->DebugString(updates_shape),
2434              "] = ", c->DebugString(prefix_updates), ": ", s.error_message());
2435        }
2436
2437        ShapeHandle suffix_output;
2438        TF_RETURN_IF_ERROR(c->Subshape(input_shape, ix, &suffix_output));
2439        ShapeHandle suffix_updates;
2440        TF_RETURN_IF_ERROR(
2441            c->Subshape(updates_shape, outer_dims, &suffix_updates));
2442        s = c->Merge(suffix_output, suffix_updates, &unused);
2443        if (!s.ok()) {
2444          return errors::InvalidArgument(
2445              "Dimensions [", ix, ",", c->Rank(input_shape),
2446              ") of input[shape=", c->DebugString(input_shape),
2447              "] = ", c->DebugString(suffix_output), " must match dimensions [",
2448              outer_dims, ",", c->Rank(updates_shape),
2449              ") of updates[shape=", c->DebugString(updates_shape),
2450              "] = ", c->DebugString(suffix_updates), ": ", s.error_message());
2451        }
2452      }
2453    }
2454
2455    if (c->input_handle_shapes_and_types(0) == nullptr && c->num_outputs() > 0) {
2456      // This is called for tf.scatter_nd; output is a tensor with this shape.
2457      c->set_output(0, input_shape);
2458    }
2459    return Status::OK();
2460  }
2461
2462  Status ExplicitShape(InferenceContext* c) {
2463    PartialTensorShape shape;
2464    TF_RETURN_IF_ERROR(c->GetAttr("shape", &shape));
2465    ShapeHandle output_shape;
2466    TF_RETURN_IF_ERROR(c->MakeShapeFromPartialTensorShape(shape, &output_shape));
2467    c->set_output(0, output_shape);
2468    return Status::OK();
2469  }
2470
2471  Status ExplicitShapes(InferenceContext* c) {
2472    std::vector<PartialTensorShape> shapes;
2473    TF_RETURN_IF_ERROR(c->GetAttr("shapes", &shapes));
2474    if (shapes.empty()) {
2475      return errors::Internal("shapes attribute is empty");
2476    }
2477    for (int i = 0, end = shapes.size(); i < end; ++i) {
2478      ShapeHandle output_shape;
2479      TF_RETURN_IF_ERROR(
```

```
2480          c->MakeShapeFromPartialTensorShape(shapes[i], &output_shape));
2481      c->set_output(i, output_shape);
2482    }
2483    return Status::OK();
2484  }
2485
2486  Status SparseReduceShapeFn(InferenceContext* c) {
2487    // Input 0: input_indices
2488    // Input 1: input_values
2489    // Input 2: input_shape
2490    // Input 3: reduction_axes
2491    // Attr: keep_dims
2492    bool keep_dims = false;
2493    TF_RETURN_IF_ERROR(c->GetAttr("keep_dims", &keep_dims));
2494
2495    const Tensor* shape_tensor = c->input_tensor(2);
2496    const Tensor* axes_tensor = c->input_tensor(3);
2497    if (shape_tensor != nullptr && axes_tensor != nullptr) {
2498      auto shape_vec = shape_tensor->flat<int64_t>();
2499      auto axes_vec = axes_tensor->flat<int32>();
2500
2501      int64_t ndims = shape_vec.size();
2502      absl::flat_hash_set<int64_t> axes;
2503      if (ndims == 0)
2504        return errors::InvalidArgument(
2505            "Number of dims in shape tensor must not be 0");
2506      for (int i = 0; i < axes_vec.size(); i++) {
2507        axes.insert((axes_vec(i) + ndims) % ndims);
2508      }
2509
2510      std::vector<DimensionHandle> dims;
2511      if (keep_dims) {
2512        dims.reserve(ndims);
2513        for (int d = 0; d < ndims; ++d) {
2514          if (axes.find(d) == axes.end()) {
2515            dims.push_back(c->MakeDim(shape_vec(d)));
2516          } else {
2517            dims.push_back(c->MakeDim(1));
2518          }
2519        }
2520      } else {
2521        for (int d = 0; d < ndims; ++d) {
2522          if (axes.find(d) == axes.end()) {
2523            dims.push_back(c->MakeDim(shape_vec(d)));
2524          }
2525        }
2526      }
2527
2528      c->set_output(0, c->MakeShape(dims));
```

```
2529      return Status::OK();
2530    }
2531    return UnknownShape(c);
2532  }
2533
2534  Status QuantizedConv2DShape(InferenceContext* c) {
2535    TF_RETURN_IF_ERROR(shape_inference::Conv2DShape(c));
2536    ShapeHandle unused;
2537    TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
2538    TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 0, &unused));
2539    TF_RETURN_IF_ERROR(c->WithRank(c->input(4), 0, &unused));
2540    TF_RETURN_IF_ERROR(c->WithRank(c->input(5), 0, &unused));
2541    c->set_output(1, c->Scalar());
2542    c->set_output(2, c->Scalar());
2543    return Status::OK();
2544  }
2545
2546  Status QuantizedAvgPoolShape(InferenceContext* c) {
2547    TF_RETURN_IF_ERROR(shape_inference::AvgPoolShape(c));
2548    ShapeHandle unused;
2549    TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 0, &unused));
2550    TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 0, &unused));
2551    c->set_output(1, c->Scalar());
2552    c->set_output(2, c->Scalar());
2553    return Status::OK();
2554  }
2555
2556  Status QuantizeV2Shape(InferenceContext* c) {
2557    int axis = -1;
2558    Status s = c->GetAttr("axis", &axis);
2559    if (!s.ok() && s.code() != error::NOT_FOUND) {
2560      return s;
2561    }
2562    if (axis < -1) {
2563      return errors::InvalidArgument("axis should be at least -1, got ", axis);
2564    }
2565    const int minmax_rank = (axis == -1) ? 0 : 1;
2566    TF_RETURN_IF_ERROR(shape_inference::UnchangedShape(c));
2567    ShapeHandle minmax;
2568    TF_RETURN_IF_ERROR(c->WithRank(c->input(1), minmax_rank, &minmax));
2569    TF_RETURN_IF_ERROR(c->WithRank(c->input(2), minmax_rank, &minmax));
2570    if (axis != -1) {
2571      ShapeHandle input;
2572      TF_RETURN_IF_ERROR(c->WithRankAtLeast(c->input(0), axis + 1, &input));
2573      DimensionHandle depth;
2574      TF_RETURN_IF_ERROR(
2575          c->Merge(c->Dim(minmax, 0), c->Dim(input, axis), &depth));
2576    }
2577    c->set_output(1, minmax);
```

```
2578    c->set_output(2, minmax);
2579    return Status::OK();
2580  }
2581
2582  Status ReduceScatterShape(shape_inference::InferenceContext* c) {
2583    shape_inference::ShapeHandle in = c->input(0);
2584    if (!c->RankKnown(in)) {
2585      // Input shape unknown, so set unknown output shape.
2586      c->set_output(0, in);
2587      return Status::OK();
2588    }
2589
2590    shape_inference::ShapeHandle group_assignment_shape = c->input(1);
2591    if (c->Rank(group_assignment_shape) != 2)
2592      return errors::InvalidArgument(
2593          "ReduceScatter group_assignment should be rank 2");
2594
2595    const Tensor* scatter_dimension = c->input_tensor(2);
2596    if (!scatter_dimension) {
2597      c->set_output(0, c->UnknownShape());
2598      return Status::OK();
2599    }
2600    int64_t scatter_dim;
2601    TF_RETURN_IF_ERROR(c->GetScalarFromTensor(scatter_dimension, &scatter_dim));
2602
2603    std::vector<shape_inference::DimensionHandle> out_dims;
2604    out_dims.reserve(c->Rank(in));
2605    for (int i = 0; i < c->Rank(in); ++i) {
2606      // If the dimension is the scatter_dimension, then divide the dimension
2607      // by the partition size in the group_assignment.
2608      if (i == scatter_dim) {
2609        shape_inference::DimensionHandle dim = c->Dim(in, i);
2610        shape_inference::DimensionHandle out_dim;
2611        TF_RETURN_IF_ERROR(c->Divide(dim, c->Dim(group_assignment_shape, 1),
2612                                     /*evenly_divisible=*/true, &out_dim));
2613        out_dims.push_back(out_dim);
2614      } else {
2615        out_dims.emplace_back(c->Dim(in, i));
2616      }
2617    }
2618    c->set_output(0, c->MakeShape(out_dims));
2619    return Status::OK();
2620  }
2621
2622  }  // namespace shape_inference
2623
2624  }  // namespace tensorflow
```