tensorflower-gardener Merge pull request #51035 from slowy07:minor-fixing ... History

21 contributors +9

2692 lines (2437 sloc) | 110 KB ...

```
1
2  /* Copyright 2017 The TensorFlow Authors. All Rights Reserved.
3
4  Licensed under the Apache License, Version 2.0 (the "License");
5  you may not use this file except in compliance with the License.
6  You may obtain a copy of the License at
7
8      http://www.apache.org/licenses/LICENSE-2.0
9
10 Unless required by applicable law or agreed to in writing, software
11 distributed under the License is distributed on an "AS IS" BASIS,
12 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 See the License for the specific language governing permissions and
14 limitations under the License.
15 ==============================================================================*/
16
17 #include "tensorflow/core/grappler/costs/op_level_cost_estimator.h"
18
19 #include "absl/strings/match.h"
20 #include "third_party/eigen3/Eigen/Core"
21 #include "tensorflow/core/framework/attr_value.pb.h"
22 #include "tensorflow/core/framework/attr_value_util.h"
23 #include "tensorflow/core/framework/tensor.pb.h"
24 #include "tensorflow/core/framework/tensor_shape.pb.h"
25 #include "tensorflow/core/framework/types.h"
26 #include "tensorflow/core/grappler/clusters/utils.h"
27 #include "tensorflow/core/grappler/costs/op_context.h"
28 #include "tensorflow/core/grappler/costs/utils.h"
29 #include "tensorflow/core/platform/errors.h"
```

```
30
31  namespace tensorflow {
32  namespace grappler {
33
34  // TODO(dyoon): update op to Predict method map for TF ops with V2 or V3 suffix.
35  constexpr int kOpsPerMac = 2;
36  constexpr char kGuaranteeConst[] = "GuaranteeConst";
37  constexpr char kAddN[] = "AddN";
38  constexpr char kBitCast[] = "BitCast";
39  constexpr char kConcatV2[] = "ConcatV2";
40  constexpr char kConv2d[] = "Conv2D";
41  constexpr char kConv2dBackpropFilter[] = "Conv2DBackpropFilter";
42  constexpr char kConv2dBackpropInput[] = "Conv2DBackpropInput";
43  constexpr char kFusedConv2dBiasActivation[] = "FusedConv2DBiasActivation";
44  constexpr char kDataFormatVecPermute[] = "DataFormatVecPermute";
45  constexpr char kDepthToSpace[] = "DepthToSpace";
46  constexpr char kDepthwiseConv2dNative[] = "DepthwiseConv2dNative";
47  constexpr char kDepthwiseConv2dNativeBackpropFilter[] =
48      "DepthwiseConv2dNativeBackpropFilter";
49  constexpr char kDepthwiseConv2dNativeBackpropInput[] =
50      "DepthwiseConv2dNativeBackpropInput";
51  constexpr char kMatMul[] = "MatMul";
52  constexpr char kXlaEinsum[] = "XlaEinsum";
53  constexpr char kEinsum[] = "Einsum";
54  constexpr char kExpandDims[] = "ExpandDims";
55  constexpr char kFill[] = "Fill";
56  constexpr char kSparseMatMul[] = "SparseMatMul";
57  constexpr char kSparseTensorDenseMatMul[] = "SparseTensorDenseMatMul";
58  constexpr char kPlaceholder[] = "Placeholder";
59  constexpr char kIdentity[] = "Identity";
60  constexpr char kIdentityN[] = "IdentityN";
61  constexpr char kRefIdentity[] = "RefIdentity";
62  constexpr char kNoOp[] = "NoOp";
63  constexpr char kReshape[] = "Reshape";
64  constexpr char kSplit[] = "Split";
65  constexpr char kSqueeze[] = "Squeeze";
66  constexpr char kRecv[] = "_Recv";
67  constexpr char kSend[] = "_Send";
68  constexpr char kBatchMatMul[] = "BatchMatMul";
69  constexpr char kBatchMatMulV2[] = "BatchMatMulV2";
70  constexpr char kOneHot[] = "OneHot";
71  constexpr char kPack[] = "Pack";
72  constexpr char kRank[] = "Rank";
73  constexpr char kRange[] = "Range";
74  constexpr char kShape[] = "Shape";
75  constexpr char kShapeN[] = "ShapeN";
76  constexpr char kSize[] = "Size";
77  constexpr char kStopGradient[] = "StopGradient";
78  constexpr char kPreventGradient[] = "PreventGradient";
```

```
79  constexpr char kGather[] = "Gather";
80  constexpr char kGatherNd[] = "GatherNd";
81  constexpr char kGatherV2[] = "GatherV2";
82  constexpr char kScatterAdd[] = "ScatterAdd";
83  constexpr char kScatterDiv[] = "ScatterDiv";
84  constexpr char kScatterMax[] = "ScatterMax";
85  constexpr char kScatterMin[] = "ScatterMin";
86  constexpr char kScatterMul[] = "ScatterMul";
87  constexpr char kScatterSub[] = "ScatterSub";
88  constexpr char kScatterUpdate[] = "ScatterUpdate";
89  constexpr char kSlice[] = "Slice";
90  constexpr char kStridedSlice[] = "StridedSlice";
91  constexpr char kSpaceToDepth[] = "SpaceToDepth";
92  constexpr char kTranspose[] = "Transpose";
93  constexpr char kTile[] = "Tile";
94  constexpr char kMaxPool[] = "MaxPool";
95  constexpr char kMaxPoolGrad[] = "MaxPoolGrad";
96  constexpr char kAvgPool[] = "AvgPool";
97  constexpr char kAvgPoolGrad[] = "AvgPoolGrad";
98  constexpr char kFusedBatchNorm[] = "FusedBatchNorm";
99  constexpr char kFusedBatchNormGrad[] = "FusedBatchNormGrad";
100 constexpr char kQuantizedMatMul[] = "QuantizedMatMul";
101 constexpr char kQuantizedMatMulV2[] = "QuantizedMatMulV2";
102 constexpr char kUnpack[] = "Unpack";
103 constexpr char kSoftmax[] = "Softmax";
104 constexpr char kResizeBilinear[] = "ResizeBilinear";
105 constexpr char kCropAndResize[] = "CropAndResize";
106 // Dynamic control flow ops.
107 constexpr char kSwitch[] = "Switch";
108 constexpr char kMerge[] = "Merge";
109 constexpr char kEnter[] = "Enter";
110 constexpr char kExit[] = "Exit";
111 constexpr char kNextIteration[] = "NextIteration";
112 // Persistent ops.
113 constexpr char kConst[] = "Const";
114 constexpr char kVariable[] = "Variable";
115 constexpr char kVariableV2[] = "VariableV2";
116 constexpr char kAutoReloadVariable[] = "AutoReloadVariable";
117 constexpr char kVarHandleOp[] = "VarHandleOp";
118 constexpr char kVarHandlesOp[] = "_VarHandlesOp";
119 constexpr char kReadVariableOp[] = "ReadVariableOp";
120 constexpr char kReadVariablesOp[] = "_ReadVariablesOp";
121 constexpr char kAssignVariableOp[] = "AssignVariableOp";
122 constexpr char kAssignAddVariableOp[] = "AssignAddVariableOp";
123 constexpr char kAssignSubVariableOp[] = "AssignSubVariableOp";
124
125 static const Costs::Duration kMinComputeTime(1);
126 static const int64_t kMinComputeOp = 1;
127
```

```
namespace {

std::string GetDataFormat(const OpInfo& op_info) {
  std::string data_format = "NHWC";  // Default format.
  if (op_info.attr().find("data_format") != op_info.attr().end()) {
    data_format = op_info.attr().at("data_format").s();
  }
  return data_format;
}

std::string GetFilterFormat(const OpInfo& op_info) {
  std::string filter_format = "HWIO";  // Default format.
  if (op_info.attr().find("filter_format") != op_info.attr().end()) {
    filter_format = op_info.attr().at("filter_format").s();
  }
  return filter_format;
}

Padding GetPadding(const OpInfo& op_info) {
  if (op_info.attr().find("padding") != op_info.attr().end() &&
      op_info.attr().at("padding").s() == "VALID") {
    return Padding::VALID;
  }
  return Padding::SAME;  // Default padding.
}

bool IsTraining(const OpInfo& op_info) {
  if (op_info.attr().find("is_training") != op_info.attr().end() &&
      op_info.attr().at("is_training").b()) {
    return true;
  }
  return false;
}

// TODO(dyoon): support non-4D tensors in the cost functions of convolution
// related ops (Conv, Pool, BatchNorm, and their backprops) and the related
// helper functions.
std::vector<int64_t> GetStrides(const OpInfo& op_info) {
  if (op_info.attr().find("strides") != op_info.attr().end()) {
    const auto strides = op_info.attr().at("strides").list().i();
    DCHECK(strides.size() == 4)
        << "Attr strides is not a length-4 vector: " << op_info.DebugString();
    if (strides.size() != 4) return {1, 1, 1, 1};
    return {strides[0], strides[1], strides[2], strides[3]};
  }
  return {1, 1, 1, 1};
}

std::vector<int64_t> GetKernelSize(const OpInfo& op_info) {
```

```
  if (op_info.attr().find("ksize") != op_info.attr().end()) {
    const auto ksize = op_info.attr().at("ksize").list().i();
    DCHECK(ksize.size() == 4)
        << "Attr ksize is not a length-4 vector: " << op_info.DebugString();
    if (ksize.size() != 4) return {1, 1, 1, 1};
    return {ksize[0], ksize[1], ksize[2], ksize[3]};
  }
  // Note that FusedBatchNorm doesn't have ksize attr, but GetKernelSize returns
  // {1, 1, 1, 1} in that case.
  return {1, 1, 1, 1};
}

int64_t GetOutputSize(const int64_t input, const int64_t filter,
                      const int64_t stride, const Padding& padding) {
  // Logic for calculating output shape is from GetWindowedOutputSizeVerbose()
  // function in third_party/tensorflow/core/framework/common_shape_fns.cc.
  if (padding == Padding::VALID) {
    return (input - filter + stride) / stride;
  } else {  // SAME.
    return (input + stride - 1) / stride;
  }
}

// Return the output element count of a multi-input element-wise op considering
// broadcasting.
int64_t CwiseOutputElementCount(const OpInfo& op_info) {
  int max_rank = 1;
  for (const OpInfo::TensorProperties& input_properties : op_info.inputs()) {
    max_rank = std::max(max_rank, input_properties.shape().dim_size());
  }

  TensorShapeProto output_shape;
  output_shape.mutable_dim()->Reserve(max_rank);
  for (int i = 0; i < max_rank; ++i) {
    output_shape.add_dim();
  }

  // Expand the shape of the output to follow the numpy-style broadcast rule
  // which matches each input starting with the trailing dimensions and working
  // its way forward. To do this, iterate through each input shape's dimensions
  // in reverse order, and potentially increase the corresponding output
  // dimension.
  for (const OpInfo::TensorProperties& input_properties : op_info.inputs()) {
    const TensorShapeProto& input_shape = input_properties.shape();
    for (int i = input_shape.dim_size() - 1; i >= 0; --i) {
      int output_shape_dim_index =
          i + output_shape.dim_size() - input_shape.dim_size();
      output_shape.mutable_dim(output_shape_dim_index)
          ->set_size(std::max(output_shape.dim(output_shape_dim_index).size(),
```

```
                                input_shape.dim(i).size()));
    }
  }

  int64_t count = 1;
  for (int i = 0; i < output_shape.dim_size(); i++) {
    count *= output_shape.dim(i).size();
  }
  return count;
}

// Helper function for determining whether there are repeated indices in the
// input Einsum equation.
bool CheckRepeatedDimensions(const absl::string_view dim_str) {
  int str_size = dim_str.size();
  for (int idx = 0; idx < str_size - 1; idx++) {
    if (dim_str.find(dim_str[idx], idx + 1) != std::string::npos) {
      return true;
    }
  }
  return false;
}

// Auxiliary function for determining whether OpLevelCostEstimator is compatible
// with a given Einsum.
bool IsEinsumCorrectlyFormed(const OpContext& einsum_context) {
  const auto& op_info = einsum_context.op_info;

  auto it = op_info.attr().find("equation");
  if (it == op_info.attr().end()) return false;
  const absl::string_view equation = it->second.s();
  std::vector<std::string> equation_split = absl::StrSplit(equation, "->");

  if (equation_split.empty()) {
    LOG(WARNING) << "Einsum with malformed equation";
    return false;
  }
  std::vector<absl::string_view> input_split =
      absl::StrSplit(equation_split[0], ',');

  // The current model covers Einsum operations with two operands and a RHS
  if (op_info.inputs_size() != 2 || equation_split.size() != 2) {
    VLOG(1) << "Missing accurate estimator for op: " << op_info.op();
    return false;
  }
  const auto& a_input = op_info.inputs(0);
  const auto& b_input = op_info.inputs(1);
  absl::string_view rhs_str = equation_split[1];
  absl::string_view a_input_str = input_split[0];
```

```
  absl::string_view b_input_str = input_split[1];

  // Ellipsis are not currently supported
  if (absl::StrContains(a_input_str, "...") ||
      absl::StrContains(b_input_str, "...")) {
    VLOG(1) << "Missing accurate estimator for op: " << op_info.op()
            << ", ellipsis not supported";
    return false;
  }

  constexpr int kMatrixRank = 2;

  bool a_input_shape_unknown = false;
  bool b_input_shape_unknown = false;

  TensorShapeProto a_input_shape = MaybeGetMinimumShape(
      a_input.shape(), std::max(kMatrixRank, a_input.shape().dim_size()),
      &a_input_shape_unknown);
  TensorShapeProto b_input_shape = MaybeGetMinimumShape(
      b_input.shape(), std::max(kMatrixRank, b_input.shape().dim_size()),
      &b_input_shape_unknown);

  if (a_input_str.size() != static_cast<size_t>(a_input_shape.dim_size()) ||
      b_input_str.size() != static_cast<size_t>(b_input_shape.dim_size())) {
    VLOG(1) << "Missing accurate estimator for op: " << op_info.op()
            << ", equation subscripts don't match tensor rank.";
    return false;
  }

  // Subscripts where axis appears more than once for a single input are not yet
  // supported
  if (CheckRepeatedDimensions(a_input_str) ||
      CheckRepeatedDimensions(b_input_str) ||
      CheckRepeatedDimensions(rhs_str)) {
    VLOG(1) << "Missing accurate estimator for op: " << op_info.op()
            << ", Subscripts where axis appears more than once for a single "
               "input are not yet supported";
    return false;
  }

  return true;
}

}  // namespace

// Return a minimum shape if the shape is unknown. If known, return the original
// shape.
TensorShapeProto MaybeGetMinimumShape(const TensorShapeProto& original_shape,
                                      int rank, bool* found_unknown_shapes) {
```

```
  auto shape = original_shape;
  bool is_scalar = !shape.unknown_rank() && shape.dim_size() == 0;

  if (shape.unknown_rank() || (!is_scalar && shape.dim_size() < rank)) {
    *found_unknown_shapes = true;
    VLOG(2) << "Use minimum shape because the rank is unknown.";
    // The size of each dimension is at least 1, if unknown.
    for (int i = shape.dim_size(); i < rank; i++) {
      shape.add_dim()->set_size(1);
    }
  } else if (is_scalar) {
    for (int i = 0; i < rank; i++) {
      shape.add_dim()->set_size(1);
    }
  } else if (shape.dim_size() > rank) {
    *found_unknown_shapes = true;
    shape.clear_dim();
    for (int i = 0; i < rank; i++) {
      shape.add_dim()->set_size(original_shape.dim(i).size());
    }
  } else {
    for (int i = 0; i < shape.dim_size(); i++) {
      if (shape.dim(i).size() < 0) {
        *found_unknown_shapes = true;
        VLOG(2) << "Use minimum dim size 1 because the shape is unknown.";
        // The size of each dimension is at least 1, if unknown.
        shape.mutable_dim(i)->set_size(1);
      }
    }
  }
  return shape;
}

OpLevelCostEstimator::OpLevelCostEstimator() {
  // Syntactic sugar to build and return a lambda that takes an OpInfo and
  // returns a cost.
  typedef Status (OpLevelCostEstimator::*CostImpl)(const OpContext& op_context,
                                                   NodeCosts*) const;
  auto wrap = [this](CostImpl impl)
      -> std::function<Status(const OpContext&, NodeCosts*)> {
    return [this, impl](const OpContext& op_context, NodeCosts* node_costs) {
      return (this->*impl)(op_context, node_costs);
    };
  };

  device_cost_impl_.emplace(kConv2d,
                            wrap(&OpLevelCostEstimator::PredictConv2D));
  device_cost_impl_.emplace(
      kConv2dBackpropFilter,
```

```
373         wrap(&OpLevelCostEstimator::PredictConv2DBackpropFilter));
374     device_cost_impl_.emplace(
375         kConv2dBackpropInput,
376         wrap(&OpLevelCostEstimator::PredictConv2DBackpropInput));
377     device_cost_impl_.emplace(
378         kFusedConv2dBiasActivation,
379         wrap(&OpLevelCostEstimator::PredictFusedConv2DBiasActivation));
380     // reuse Conv2D for DepthwiseConv2dNative because the calculation is the
381     // same although the actual meaning of the parameters are different. See
382     // comments in PredictConv2D and related functions
383     device_cost_impl_.emplace(kDepthwiseConv2dNative,
384                               wrap(&OpLevelCostEstimator::PredictConv2D));
385     device_cost_impl_.emplace(
386         kDepthwiseConv2dNativeBackpropFilter,
387         wrap(&OpLevelCostEstimator::PredictConv2DBackpropFilter));
388     device_cost_impl_.emplace(
389         kDepthwiseConv2dNativeBackpropInput,
390         wrap(&OpLevelCostEstimator::PredictConv2DBackpropInput));
391     device_cost_impl_.emplace(kMatMul,
392                               wrap(&OpLevelCostEstimator::PredictMatMul));
393     device_cost_impl_.emplace(kSparseMatMul,
394                               wrap(&OpLevelCostEstimator::PredictMatMul));
395     device_cost_impl_.emplace(
396         kSparseTensorDenseMatMul,
397         wrap(&OpLevelCostEstimator::PredictSparseTensorDenseMatMul));
398     device_cost_impl_.emplace(kBatchMatMul,
399                               wrap(&OpLevelCostEstimator::PredictBatchMatMul));
400     device_cost_impl_.emplace(kBatchMatMulV2,
401                               wrap(&OpLevelCostEstimator::PredictBatchMatMul));
402     device_cost_impl_.emplace(kQuantizedMatMul,
403                               wrap(&OpLevelCostEstimator::PredictMatMul));
404     device_cost_impl_.emplace(kQuantizedMatMulV2,
405                               wrap(&OpLevelCostEstimator::PredictMatMul));
406     device_cost_impl_.emplace(kXlaEinsum,
407                               wrap(&OpLevelCostEstimator::PredictEinsum));
408     device_cost_impl_.emplace(kEinsum,
409                               wrap(&OpLevelCostEstimator::PredictEinsum));
410
411     device_cost_impl_.emplace(kNoOp, wrap(&OpLevelCostEstimator::PredictNoOp));
412     device_cost_impl_.emplace(kGuaranteeConst,
413                               wrap(&OpLevelCostEstimator::PredictNoOp));
414
415     device_cost_impl_.emplace(kGather,
416                               wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
417     device_cost_impl_.emplace(kGatherNd,
418                               wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
419     device_cost_impl_.emplace(kGatherV2,
420                               wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
421     device_cost_impl_.emplace(kScatterAdd,
```

```
422                             wrap(&OpLevelCostEstimator::PredictScatter));
423     device_cost_impl_.emplace(kScatterDiv,
424                             wrap(&OpLevelCostEstimator::PredictScatter));
425     device_cost_impl_.emplace(kScatterMax,
426                             wrap(&OpLevelCostEstimator::PredictScatter));
427     device_cost_impl_.emplace(kScatterMin,
428                             wrap(&OpLevelCostEstimator::PredictScatter));
429     device_cost_impl_.emplace(kScatterMul,
430                             wrap(&OpLevelCostEstimator::PredictScatter));
431     device_cost_impl_.emplace(kScatterSub,
432                             wrap(&OpLevelCostEstimator::PredictScatter));
433     device_cost_impl_.emplace(kScatterUpdate,
434                             wrap(&OpLevelCostEstimator::PredictScatter));
435
436     device_cost_impl_.emplace(kSlice,
437                             wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
438     device_cost_impl_.emplace(kStridedSlice,
439                             wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
440
441     device_cost_impl_.emplace(kPlaceholder,
442                             wrap(&OpLevelCostEstimator::PredictIdentity));
443     device_cost_impl_.emplace(kIdentity,
444                             wrap(&OpLevelCostEstimator::PredictIdentity));
445     device_cost_impl_.emplace(kIdentityN,
446                             wrap(&OpLevelCostEstimator::PredictIdentity));
447     device_cost_impl_.emplace(kRefIdentity,
448                             wrap(&OpLevelCostEstimator::PredictIdentity));
449     device_cost_impl_.emplace(kStopGradient,
450                             wrap(&OpLevelCostEstimator::PredictIdentity));
451     device_cost_impl_.emplace(kPreventGradient,
452                             wrap(&OpLevelCostEstimator::PredictIdentity));
453     device_cost_impl_.emplace(kReshape,
454                             wrap(&OpLevelCostEstimator::PredictIdentity));
455     device_cost_impl_.emplace(kRecv,
456                             wrap(&OpLevelCostEstimator::PredictIdentity));
457     device_cost_impl_.emplace(kSend,
458                             wrap(&OpLevelCostEstimator::PredictIdentity));
459     device_cost_impl_.emplace(kSwitch,
460                             wrap(&OpLevelCostEstimator::PredictIdentity));
461     device_cost_impl_.emplace(kMerge,
462                             wrap(&OpLevelCostEstimator::PredictIdentity));
463     device_cost_impl_.emplace(kEnter,
464                             wrap(&OpLevelCostEstimator::PredictIdentity));
465     device_cost_impl_.emplace(kExit,
466                             wrap(&OpLevelCostEstimator::PredictIdentity));
467     device_cost_impl_.emplace(kNextIteration,
468                             wrap(&OpLevelCostEstimator::PredictIdentity));
469     device_cost_impl_.emplace(kBitCast,
470                             wrap(&OpLevelCostEstimator::PredictIdentity));
```

```
471
472     device_cost_impl_.emplace(kConcatV2,
473                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
474     device_cost_impl_.emplace(kDataFormatVecPermute,
475                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
476     device_cost_impl_.emplace(kDepthToSpace,
477                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
478     device_cost_impl_.emplace(kExpandDims,
479                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
480     device_cost_impl_.emplace(kFill,
481                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
482     device_cost_impl_.emplace(kOneHot,
483                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
484     device_cost_impl_.emplace(kPack,
485                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
486     device_cost_impl_.emplace(kRange,
487                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
488     device_cost_impl_.emplace(kSpaceToDepth,
489                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
490     device_cost_impl_.emplace(kSplit,
491                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
492     device_cost_impl_.emplace(kSqueeze,
493                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
494     device_cost_impl_.emplace(kTranspose,
495                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
496     device_cost_impl_.emplace(kTile,
497                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
498     device_cost_impl_.emplace(kUnpack,
499                               wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
500
501     device_cost_impl_.emplace(kRank,
502                               wrap(&OpLevelCostEstimator::PredictMetadata));
503     device_cost_impl_.emplace(kShape,
504                               wrap(&OpLevelCostEstimator::PredictMetadata));
505     device_cost_impl_.emplace(kShapeN,
506                               wrap(&OpLevelCostEstimator::PredictMetadata));
507     device_cost_impl_.emplace(kSize,
508                               wrap(&OpLevelCostEstimator::PredictMetadata));
509     device_cost_impl_.emplace(kMaxPool,
510                               wrap(&OpLevelCostEstimator::PredictMaxPool));
511     device_cost_impl_.emplace(kMaxPoolGrad,
512                               wrap(&OpLevelCostEstimator::PredictMaxPoolGrad));
513     device_cost_impl_.emplace(kAvgPool,
514                               wrap(&OpLevelCostEstimator::PredictAvgPool));
515     device_cost_impl_.emplace(kAvgPoolGrad,
516                               wrap(&OpLevelCostEstimator::PredictAvgPoolGrad));
517     device_cost_impl_.emplace(kFusedBatchNorm,
518                               wrap(&OpLevelCostEstimator::PredictFusedBatchNorm));
519     device_cost_impl_.emplace(
```

```
520         kFusedBatchNormGrad,
521         wrap(&OpLevelCostEstimator::PredictFusedBatchNormGrad));
522     device_cost_impl_.emplace(kSoftmax,
523                               wrap(&OpLevelCostEstimator::PredictSoftmax));
524     device_cost_impl_.emplace(kResizeBilinear,
525                               wrap(&OpLevelCostEstimator::PredictResizeBilinear));
526     device_cost_impl_.emplace(kCropAndResize,
527                               wrap(&OpLevelCostEstimator::PredictCropAndResize));
528     device_cost_impl_.emplace(
529         kAssignVariableOp, wrap(&OpLevelCostEstimator::PredictAssignVariableOps));
530     device_cost_impl_.emplace(
531         kAssignAddVariableOp,
532         wrap(&OpLevelCostEstimator::PredictAssignVariableOps));
533     device_cost_impl_.emplace(
534         kAssignSubVariableOp,
535         wrap(&OpLevelCostEstimator::PredictAssignVariableOps));
536     device_cost_impl_.emplace(kAddN, wrap(&OpLevelCostEstimator::PredictNaryOp));
537
538     persistent_ops_ = {
539         kConst,       kVariable,       kVariableV2,   kAutoReloadVariable,
540         kVarHandleOp, kReadVariableOp, kVarHandlesOp, kReadVariablesOp};
541
542   #define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost
543
544     // Quantize = apply min and max bounds, multiply by scale factor and round.
545     const int quantize_v2_cost =
546         EIGEN_COST(scalar_product_op<float>) + EIGEN_COST(scalar_max_op<float>) +
547         EIGEN_COST(scalar_min_op<float>) + EIGEN_COST(scalar_round_op<float>);
548     const int quantize_and_dequantize_v2_cost =
549         quantize_v2_cost + EIGEN_COST(scalar_product_op<float>);
550
551     // Unary ops alphabetically sorted
552     elementwise_ops_.emplace("Acos", EIGEN_COST(scalar_acos_op<float>));
553     elementwise_ops_.emplace("All", EIGEN_COST(scalar_boolean_and_op));
554     elementwise_ops_.emplace("ArgMax", EIGEN_COST(scalar_max_op<float>));
555     elementwise_ops_.emplace("Asin", EIGEN_COST(scalar_asin_op<float>));
556     elementwise_ops_.emplace("Atan", EIGEN_COST(scalar_atan_op<float>));
557     elementwise_ops_.emplace("Atan2", EIGEN_COST(scalar_quotient_op<float>) +
558                                           EIGEN_COST(scalar_atan_op<float>));
559     // For now, we use Eigen cost model for float to int16 cast as an example
560     // case; Eigen cost model is zero when src and dst types are identical,
561     // and it uses AddCost (1) when different. We may implement a separate
562     // cost functions for cast ops, using the actual input and output types.
563     elementwise_ops_.emplace(
564         "Cast", Eigen::internal::functor_traits<
565                     Eigen::internal::scalar_cast_op<float, int16>>::Cost);
566     elementwise_ops_.emplace("Ceil", EIGEN_COST(scalar_ceil_op<float>));
567     elementwise_ops_.emplace("Cos", EIGEN_COST(scalar_cos_op<float>));
568     elementwise_ops_.emplace("Dequantize", EIGEN_COST(scalar_product_op<float>));
```

```
569     elementwise_ops_.emplace("Erf", 1);
570     elementwise_ops_.emplace("Erfc", 1);
571     elementwise_ops_.emplace("Exp", EIGEN_COST(scalar_exp_op<float>));
572     elementwise_ops_.emplace("Expm1", EIGEN_COST(scalar_expm1_op<float>));
573     elementwise_ops_.emplace("Floor", EIGEN_COST(scalar_floor_op<float>));
574     elementwise_ops_.emplace("Inv", EIGEN_COST(scalar_inverse_op<float>));
575     elementwise_ops_.emplace("InvGrad", 1);
576     elementwise_ops_.emplace("Lgamma", 1);
577     elementwise_ops_.emplace("Log", EIGEN_COST(scalar_log_op<float>));
578     elementwise_ops_.emplace("Log1p", EIGEN_COST(scalar_log1p_op<float>));
579     elementwise_ops_.emplace("Max", EIGEN_COST(scalar_max_op<float>));
580     elementwise_ops_.emplace("Min", EIGEN_COST(scalar_min_op<float>));
581     elementwise_ops_.emplace("Neg", EIGEN_COST(scalar_opposite_op<float>));
582     elementwise_ops_.emplace("Prod", EIGEN_COST(scalar_product_op<float>));
583     elementwise_ops_.emplace("QuantizeAndDequantizeV2",
584                              quantize_and_dequantize_v2_cost);
585     elementwise_ops_.emplace("QuantizeAndDequantizeV4",
586                              quantize_and_dequantize_v2_cost);
587     elementwise_ops_.emplace("QuantizedSigmoid",
588                              EIGEN_COST(scalar_logistic_op<float>));
589     elementwise_ops_.emplace("QuantizeV2", quantize_v2_cost);
590     elementwise_ops_.emplace("Reciprocal", EIGEN_COST(scalar_inverse_op<float>));
591     elementwise_ops_.emplace("Relu", EIGEN_COST(scalar_max_op<float>));
592     elementwise_ops_.emplace("Relu6", EIGEN_COST(scalar_max_op<float>));
593     elementwise_ops_.emplace("Rint", 1);
594     elementwise_ops_.emplace("Round", EIGEN_COST(scalar_round_op<float>));
595     elementwise_ops_.emplace("Rsqrt", EIGEN_COST(scalar_rsqrt_op<float>));
596     elementwise_ops_.emplace("Sigmoid", EIGEN_COST(scalar_logistic_op<float>));
597     elementwise_ops_.emplace("Sign", EIGEN_COST(scalar_sign_op<float>));
598     elementwise_ops_.emplace("Sin", EIGEN_COST(scalar_sin_op<float>));
599     elementwise_ops_.emplace("Sqrt", EIGEN_COST(scalar_sqrt_op<float>));
600     elementwise_ops_.emplace("Square", EIGEN_COST(scalar_square_op<float>));
601     elementwise_ops_.emplace("Sum", EIGEN_COST(scalar_sum_op<float>));
602     elementwise_ops_.emplace("Tan", EIGEN_COST(scalar_tan_op<float>));
603     elementwise_ops_.emplace("Tanh", EIGEN_COST(scalar_tanh_op<float>));
604     elementwise_ops_.emplace("TopKV2", EIGEN_COST(scalar_max_op<float>));
605     // Binary ops alphabetically sorted
606     elementwise_ops_.emplace("Add", EIGEN_COST(scalar_sum_op<float>));
607     elementwise_ops_.emplace("AddV2", EIGEN_COST(scalar_sum_op<float>));
608     elementwise_ops_.emplace("ApproximateEqual", 1);
609     elementwise_ops_.emplace("BiasAdd", EIGEN_COST(scalar_sum_op<float>));
610     elementwise_ops_.emplace("QuantizedBiasAdd",
611                              EIGEN_COST(scalar_sum_op<float>));
612     elementwise_ops_.emplace("Div", EIGEN_COST(scalar_quotient_op<float>));
613     elementwise_ops_.emplace("Equal", 1);
614     elementwise_ops_.emplace("FloorDiv", EIGEN_COST(scalar_quotient_op<float>));
615     elementwise_ops_.emplace("FloorMod", EIGEN_COST(scalar_mod_op<float>));
616     elementwise_ops_.emplace("Greater", 1);
617     elementwise_ops_.emplace("GreaterEqual", 1);
```

```
  elementwise_ops_.emplace("Less", 1);
  elementwise_ops_.emplace("LessEqual", 1);
  elementwise_ops_.emplace("LogicalAnd", EIGEN_COST(scalar_boolean_and_op));
  elementwise_ops_.emplace("LogicalNot", 1);
  elementwise_ops_.emplace("LogicalOr", EIGEN_COST(scalar_boolean_or_op));
  elementwise_ops_.emplace("Maximum", EIGEN_COST(scalar_max_op<float>));
  elementwise_ops_.emplace("Minimum", EIGEN_COST(scalar_min_op<float>));
  elementwise_ops_.emplace("Mod", EIGEN_COST(scalar_mod_op<float>));
  elementwise_ops_.emplace("Mul", EIGEN_COST(scalar_product_op<float>));
  elementwise_ops_.emplace("NotEqual", 1);
  elementwise_ops_.emplace("QuantizedAdd", EIGEN_COST(scalar_sum_op<float>));
  elementwise_ops_.emplace("QuantizedMul",
                           EIGEN_COST(scalar_product_op<float>));
  elementwise_ops_.emplace("RealDiv", EIGEN_COST(scalar_quotient_op<float>));
  elementwise_ops_.emplace("ReluGrad", EIGEN_COST(scalar_max_op<float>));
  elementwise_ops_.emplace("Select", EIGEN_COST(scalar_boolean_or_op));
  elementwise_ops_.emplace("SelectV2", EIGEN_COST(scalar_boolean_or_op));
  elementwise_ops_.emplace("SquaredDifference",
                           EIGEN_COST(scalar_square_op<float>) +
                               EIGEN_COST(scalar_difference_op<float>));
  elementwise_ops_.emplace("Sub", EIGEN_COST(scalar_difference_op<float>));
  elementwise_ops_.emplace("TruncateDiv",
                           EIGEN_COST(scalar_quotient_op<float>));
  elementwise_ops_.emplace("TruncateMod", EIGEN_COST(scalar_mod_op<float>));
  elementwise_ops_.emplace("Where", 1);

#undef EIGEN_COST

  // By default, use sum of memory_time and compute_time for execution_time.
  compute_memory_overlap_ = false;
}

Costs OpLevelCostEstimator::PredictCosts(const OpContext& op_context) const {
  Costs costs;
  NodeCosts node_costs;
  if (PredictNodeCosts(op_context, &node_costs).ok()) {
    if (node_costs.has_costs) {
      return node_costs.costs;
    }
    // Convert NodeCosts to Costs.
    if (node_costs.minimum_cost_op) {
      // Override to minimum cost; Note that some ops with minimum cost may have
      // non-typical device (e.g., channel for _Send), which may fail with
      // GetDeviceInfo(), called from PredictOpCountBasedCost(). Make sure we
      // directly set minimum values to Costs here, not calling
      // PredictOpCountBasedCost().
      costs.compute_time = kMinComputeTime;
      costs.execution_time = kMinComputeTime;
      costs.memory_time = 0;
```

```
667       costs.intermediate_memory_time = 0;
668       costs.intermediate_memory_read_time = 0;
669       costs.intermediate_memory_write_time = 0;
670     } else {
671       // Convert NodeCosts to Costs.
672       costs = PredictOpCountBasedCost(
673           node_costs.num_compute_ops, node_costs.num_total_read_bytes(),
674           node_costs.num_total_write_bytes(), op_context.op_info);
675     }
676     VLOG(1) << "Operation " << op_context.op_info.op() << " takes "
677             << costs.execution_time.count() << " ns.";
678     // Copy additional stats from NodeCosts to Costs.
679     costs.max_memory = node_costs.max_memory;
680     costs.persistent_memory = node_costs.persistent_memory;
681     costs.temporary_memory = node_costs.temporary_memory;
682     costs.inaccurate = node_costs.inaccurate;
683     costs.num_ops_with_unknown_shapes =
684         node_costs.num_nodes_with_unknown_shapes;
685     costs.num_ops_total = node_costs.num_nodes;
686     return costs;
687   }
688   // Errors during node cost estimate.
689   LOG(WARNING) << "Error in PredictCost() for the op: "
690                << op_context.op_info.ShortDebugString();
691   costs = Costs::ZeroCosts(/*inaccurate=*/true);
692   costs.num_ops_with_unknown_shapes = node_costs.num_nodes_with_unknown_shapes;
693   return costs;
694 }
695
696 Status OpLevelCostEstimator::PredictNodeCosts(const OpContext& op_context,
697                                               NodeCosts* node_costs) const {
698   const auto& op_info = op_context.op_info;
699   auto it = device_cost_impl_.find(op_info.op());
700   if (it != device_cost_impl_.end()) {
701     std::function<Status(const OpContext&, NodeCosts*)> estimator = it->second;
702     return estimator(op_context, node_costs);
703   }
704
705   if (persistent_ops_.find(op_info.op()) != persistent_ops_.end()) {
706     return PredictVariable(op_context, node_costs);
707   }
708
709   if (elementwise_ops_.find(op_info.op()) != elementwise_ops_.end()) {
710     return PredictCwiseOp(op_context, node_costs);
711   }
712
713   VLOG(1) << "Missing accurate estimator for op: " << op_info.op();
714
715   node_costs->num_nodes_with_unknown_op_type = 1;
```

```
  return PredictCostOfAnUnknownOp(op_context, node_costs);
}

// This method assumes a typical system composed of CPUs and GPUs, connected
// through PCIe. To define device info more precisely, override this method.
DeviceInfo OpLevelCostEstimator::GetDeviceInfo(
    const DeviceProperties& device) const {
  double gflops = -1;
  double gb_per_sec = -1;

  if (device.type() == "CPU") {
    // Check if vector instructions are available, and refine performance
    // prediction based on this.
    // Frequencies are stored in MHz in the DeviceProperties.
    gflops = device.num_cores() * device.frequency() * 1e-3;
    if (gb_per_sec < 0) {
      if (device.bandwidth() > 0) {
        gb_per_sec = device.bandwidth() / 1e6;
      } else {
        gb_per_sec = 32;
      }
    }
  } else if (device.type() == "GPU") {
    const auto& device_env = device.environment();
    auto it = device_env.find("architecture");
    if (it != device_env.end()) {
      const std::string architecture = device_env.at("architecture");
      int cores_per_multiprocessor;
      if (architecture < "3") {
        // Fermi
        cores_per_multiprocessor = 32;
      } else if (architecture < "4") {
        // Kepler
        cores_per_multiprocessor = 192;
      } else if (architecture < "6") {
        // Maxwell
        cores_per_multiprocessor = 128;
      } else {
        // Pascal (compute capability version 6) and Volta (compute capability
        // version 7)
        cores_per_multiprocessor = 64;
      }
      gflops = device.num_cores() * device.frequency() * 1e-3 *
               cores_per_multiprocessor * kOpsPerMac;
      if (device.bandwidth() > 0) {
        gb_per_sec = device.bandwidth() / 1e6;
      } else {
        gb_per_sec = 100;
      }
```

```
    } else {
      // Architecture is not available (ex: pluggable device), return default
      // value.
      gflops = 100;     // Dummy value;
      gb_per_sec = 12;  // default PCIe x16 gen3.
    }
  } else {
    LOG_EVERY_N(WARNING, 1000) << "Unknown device type: " << device.type()
                               << ", assuming PCIe between CPU and GPU.";
    gflops = 1;  // Dummy value; data transfer ops would not have compute ops.
    gb_per_sec = 12;  // default PCIe x16 gen3.
  }
  VLOG(1) << "Device: " << device.type() << " gflops: " << gflops
          << " gb_per_sec: " << gb_per_sec;

  return DeviceInfo(gflops, gb_per_sec);
}

Status OpLevelCostEstimator::PredictCwiseOp(const OpContext& op_context,
                                            NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  bool found_unknown_shapes = false;
  // For element-wise operations, op count is the element count of any input. We
  // use the count for the largest input here to be more robust in case that the
  // shape is unknown or partially known for other input.
  int64_t op_count = CalculateLargestInputCount(op_info, &found_unknown_shapes);
  // If output shape is available, try to use the element count calculated from
  // that.
  if (op_info.outputs_size() > 0) {
    op_count = std::max(
        op_count,
        CalculateTensorElementCount(op_info.outputs(0), &found_unknown_shapes));
  }
  // Calculate the output shape possibly resulting from broadcasting.
  if (op_info.inputs_size() >= 2) {
    op_count = std::max(op_count, CwiseOutputElementCount(op_info));
  }

  int op_cost = 1;
  auto it = elementwise_ops_.find(op_info.op());
  if (it != elementwise_ops_.end()) {
    op_cost = it->second;
  } else {
    return errors::InvalidArgument("Not a cwise op: ", op_info.op());
  }

  return PredictDefaultNodeCosts(op_count * op_cost, op_context,
                                 &found_unknown_shapes, node_costs);
}
```

```
814
815 Status OpLevelCostEstimator::PredictCostOfAnUnknownOp(
816     const OpContext& op_context, NodeCosts* node_costs) const {
817   // Don't assume the operation is cwise, return cost based on input/output size
818   // and admit that it is inaccurate...
819   bool found_unknown_shapes = false;
820   node_costs->inaccurate = true;
821   return PredictDefaultNodeCosts(0, op_context, &found_unknown_shapes,
822                                  node_costs);
823 }
824
825 Costs OpLevelCostEstimator::PredictOpCountBasedCost(
826     double operations, const OpInfo& op_info) const {
827   bool unknown_shapes = false;
828   const double input_size = CalculateInputSize(op_info, &unknown_shapes);
829   const double output_size = CalculateOutputSize(op_info, &unknown_shapes);
830   Costs costs =
831       PredictOpCountBasedCost(operations, input_size, output_size, op_info);
832   costs.inaccurate = unknown_shapes;
833   costs.num_ops_with_unknown_shapes = unknown_shapes;
834   costs.max_memory = output_size;
835   return costs;
836 }
837
838 Costs OpLevelCostEstimator::PredictOpCountBasedCost(
839     double operations, double input_io_bytes, double output_io_bytes,
840     const OpInfo& op_info) const {
841   double total_io_bytes = input_io_bytes + output_io_bytes;
842   const DeviceInfo device_info = GetDeviceInfo(op_info.device());
843   if (device_info.gigaops <= 0 || device_info.gb_per_sec <= 0 ||
844       device_info.intermediate_read_gb_per_sec <= 0 ||
845       device_info.intermediate_write_gb_per_sec <= 0) {
846     VLOG(1) << "BAD DEVICE. Op:" << op_info.op()
847             << " device type:" << op_info.device().type()
848             << " device model:" << op_info.device().model();
849   }
850
851   Costs::NanoSeconds compute_cost(std::ceil(operations / device_info.gigaops));
852   VLOG(1) << "Op:" << op_info.op() << " GOps:" << operations / 1e9
853           << " Compute Time (ns):" << compute_cost.count();
854
855   Costs::NanoSeconds memory_cost(
856       std::ceil(total_io_bytes / device_info.gb_per_sec));
857   VLOG(1) << "Op:" << op_info.op() << " Size (KB):" << (total_io_bytes) / 1e3
858           << " Memory Time (ns):" << memory_cost.count();
859
860   // Check if bytes > 0.  If it's not and the bandwidth is set to infinity
861   // then the result would be undefined.
862   double intermediate_read_time =
```

```
      (input_io_bytes > 0)
          ? std::ceil(input_io_bytes / device_info.intermediate_read_gb_per_sec)
          : 0;

  double intermediate_write_time =
      (output_io_bytes > 0)
          ? std::ceil(output_io_bytes /
                      device_info.intermediate_write_gb_per_sec)
          : 0;

  Costs::NanoSeconds intermediate_memory_cost =
      compute_memory_overlap_
          ? std::max(intermediate_read_time, intermediate_write_time)
          : (intermediate_read_time + intermediate_write_time);
  VLOG(1) << "Op:" << op_info.op() << " Size (KB):" << (total_io_bytes) / 1e3
          << " Intermediate Memory Time (ns):"
          << intermediate_memory_cost.count();

  Costs costs = Costs::ZeroCosts();
  costs.compute_time = compute_cost;
  costs.memory_time = memory_cost;
  costs.intermediate_memory_time = intermediate_memory_cost;
  costs.intermediate_memory_read_time =
      Costs::NanoSeconds(intermediate_read_time);
  costs.intermediate_memory_write_time =
      Costs::NanoSeconds(intermediate_write_time);
  CombineCostsAndUpdateExecutionTime(compute_memory_overlap_, &costs);
  return costs;
}

int64_t OpLevelCostEstimator::CountConv2DOperations(
    const OpInfo& op_info, bool* found_unknown_shapes) {
  return CountConv2DOperations(op_info, nullptr, found_unknown_shapes);
}

// Helper to translate the positional arguments into named fields.
/* static */
OpLevelCostEstimator::ConvolutionDimensions
OpLevelCostEstimator::ConvolutionDimensionsFromInputs(
    const TensorShapeProto& original_image_shape,
    const TensorShapeProto& original_filter_shape, const OpInfo& op_info,
    bool* found_unknown_shapes) {
  VLOG(2) << "op features: " << op_info.DebugString();
  VLOG(2) << "Original image shape: " << original_image_shape.DebugString();
  VLOG(2) << "Original filter shape: " << original_filter_shape.DebugString();

  int x_index, y_index, major_channel_index, minor_channel_index = -1;
  const std::string& data_format = GetDataFormat(op_info);
  if (data_format == "NCHW") {
```

```
912       major_channel_index = 1;
913       y_index = 2;
914       x_index = 3;
915     } else if (data_format == "NCHW_VECT_C") {
916       // Use NCHW_VECT_C
917       minor_channel_index = 1;
918       y_index = 2;
919       x_index = 3;
920       major_channel_index = 4;
921     } else {
922       // Use NHWC.
923       y_index = 1;
924       x_index = 2;
925       major_channel_index = 3;
926     }
927     const std::string& filter_format = GetFilterFormat(op_info);
928     int filter_x_index, filter_y_index, in_major_channel_index, out_channel_index,
929         in_minor_channel_index = -1;
930     if (filter_format == "HWIO") {
931       filter_y_index = 0;
932       filter_x_index = 1;
933       in_major_channel_index = 2;
934       out_channel_index = 3;
935     } else if (filter_format == "OIHW_VECT_I") {
936       out_channel_index = 0;
937       in_minor_channel_index = 1;
938       filter_y_index = 2;
939       filter_x_index = 3;
940       in_major_channel_index = 4;
941     } else {
942       // Use OIHW
943       out_channel_index = 0;
944       in_major_channel_index = 1;
945       filter_y_index = 2;
946       filter_x_index = 3;
947     }
948
949     auto image_shape = MaybeGetMinimumShape(original_image_shape,
950                                             minor_channel_index >= 0 ? 5 : 4,
951                                             found_unknown_shapes);
952     auto filter_shape = MaybeGetMinimumShape(original_filter_shape,
953                                              in_minor_channel_index >= 0 ? 5 : 4,
954                                              found_unknown_shapes);
955     VLOG(2) << "Image shape: " << image_shape.DebugString();
956     VLOG(2) << "Filter shape: " << filter_shape.DebugString();
957
958     int64_t batch = image_shape.dim(0).size();
959     int64_t ix = image_shape.dim(x_index).size();
960     int64_t iy = image_shape.dim(y_index).size();
```

```
  int64_t iz = minor_channel_index >= 0
                   ? image_shape.dim(minor_channel_index).size() *
                         image_shape.dim(major_channel_index).size()
                   : image_shape.dim(major_channel_index).size();
  int64_t kx = filter_shape.dim(filter_x_index).size();
  int64_t ky = filter_shape.dim(filter_y_index).size();
  int64_t kz = in_minor_channel_index >= 0
                   ? filter_shape.dim(in_major_channel_index).size() *
                         filter_shape.dim(in_minor_channel_index).size()
                   : filter_shape.dim(in_major_channel_index).size();
  std::vector<int64_t> strides = GetStrides(op_info);
  const auto padding = GetPadding(op_info);
  int64_t sx = strides[x_index];
  int64_t sy = strides[y_index];
  int64_t ox = GetOutputSize(ix, kx, sx, padding);
  int64_t oy = GetOutputSize(iy, ky, sy, padding);
  int64_t oz = filter_shape.dim(out_channel_index).size();
  // Only check equality when both sizes are known (in other words, when
  // neither is set to a minimum dimension size of 1).
  if (iz != 1 && kz != 1) {
    DCHECK_EQ(iz % kz, 0) << "Input channel " << iz
                          << " is not a multiple of filter channel " << kz
                          << ".";
    if (iz % kz) {
      *found_unknown_shapes = true;
    }
  } else {
    iz = kz = std::max<int64_t>(iz, kz);
  }
  OpLevelCostEstimator::ConvolutionDimensions conv_dims = {
      batch, ix, iy, iz, kx, ky, kz, oz, ox, oy, sx, sy, padding};

  VLOG(1) << "Batch Size:" << batch;
  VLOG(1) << "Image Dims:" << ix << "," << iy;
  VLOG(1) << "Input Depth:" << iz;
  VLOG(1) << "Kernel Dims:" << kx << "," << ky;
  VLOG(1) << "Kernel Depth:" << kz;
  VLOG(1) << "Output Dims:" << ox << "," << oy;
  VLOG(1) << "Output Depth:" << oz;
  VLOG(1) << "Strides:" << sx << "," << sy;
  VLOG(1) << "Padding:" << (padding == Padding::VALID ? "VALID" : "SAME");
  return conv_dims;
}

int64_t OpLevelCostEstimator::CountConv2DOperations(
    const OpInfo& op_info, ConvolutionDimensions* conv_info,
    bool* found_unknown_shapes) {
  DCHECK(op_info.op() == kConv2d || op_info.op() == kDepthwiseConv2dNative)
      << "Invalid Operation: not Conv2D nor DepthwiseConv2dNative";
```

```
1010
1011    if (op_info.inputs_size() < 2) {  // Unexpected inputs.
1012      *found_unknown_shapes = true;
1013      return 0;
1014    }
1015
1016    ConvolutionDimensions conv_dims = ConvolutionDimensionsFromInputs(
1017        op_info.inputs(0).shape(), op_info.inputs(1).shape(), op_info,
1018        found_unknown_shapes);
1019
1020    //  in DepthwiseConv2dNative conv_dims.oz is actually the channel depth
1021    //  multiplier; The effective output channel depth oz_effective is
1022    //  conv_dims.iz * conv_dims.oz. thus # ops = N x H x W x oz_effective x 2RS.
1023    //  Compare to Conv2D where # ops =  N x H x W x kz x oz x 2RS,
1024    //  oz = oz_effective,  then Conv2D_ops / Depthwise_conv2d_native_ops = kz.
1025    int64_t ops = conv_dims.batch;
1026    ops *= conv_dims.ox * conv_dims.oy;
1027    ops *= conv_dims.kx * conv_dims.ky;
1028    if (op_info.op() == kConv2d) {
1029      ops *= conv_dims.kz * conv_dims.oz;
1030    } else {
1031      // To ensure output tensor dims to be correct for DepthwiseConv2DNative,
1032      // although ops are the same as Conv2D.
1033      conv_dims.oz *= conv_dims.iz;
1034      ops *= conv_dims.oz;
1035    }
1036    ops *= kOpsPerMac;
1037
1038    if (conv_info != nullptr) {
1039      *conv_info = conv_dims;
1040    }
1041    return ops;
1042  }
1043
1044  int64_t OpLevelCostEstimator::CountMatMulOperations(
1045      const OpInfo& op_info, bool* found_unknown_shapes) {
1046    return CountMatMulOperations(op_info, nullptr, found_unknown_shapes);
1047  }
1048
1049  // TODO(nishantpatil): Create separate estimator for Sparse Matmul
1050  int64_t OpLevelCostEstimator::CountMatMulOperations(
1051      const OpInfo& op_info, MatMulDimensions* mat_mul,
1052      bool* found_unknown_shapes) {
1053    double ops = 0;
1054
1055    if (op_info.inputs_size() < 2) {
1056      LOG(ERROR) << "Need 2 inputs but got " << op_info.inputs_size();
1057      // TODO(pcma): Try to separate invalid inputs from unknown shapes
1058      *found_unknown_shapes = true;
```

```
    return 0;
  }

  auto& a_matrix = op_info.inputs(0);
  auto& b_matrix = op_info.inputs(1);

  bool transpose_a = false;
  bool transpose_b = false;

  double m_dim, n_dim, k_dim, k_dim_b = 0;

  for (const auto& item : op_info.attr()) {
    VLOG(1) << "Key:" << item.first
            << " Value:" << SummarizeAttrValue(item.second);
    if (item.first == "transpose_a" && item.second.b() == true)
      transpose_a = true;
    if (item.first == "transpose_b" && item.second.b() == true)
      transpose_b = true;
  }
  VLOG(1) << "transpose_a:" << transpose_a;
  VLOG(1) << "transpose_b:" << transpose_b;
  auto a_matrix_shape =
      MaybeGetMinimumShape(a_matrix.shape(), 2, found_unknown_shapes);
  auto b_matrix_shape =
      MaybeGetMinimumShape(b_matrix.shape(), 2, found_unknown_shapes);
  if (transpose_a) {
    m_dim = a_matrix_shape.dim(1).size();
    k_dim = a_matrix_shape.dim(0).size();
  } else {
    m_dim = a_matrix_shape.dim(0).size();
    k_dim = a_matrix_shape.dim(1).size();
  }
  if (transpose_b) {
    k_dim_b = b_matrix_shape.dim(1).size();
    n_dim = b_matrix_shape.dim(0).size();
  } else {
    k_dim_b = b_matrix_shape.dim(0).size();
    n_dim = b_matrix_shape.dim(1).size();
  }

  VLOG(1) << "M, N, K: " << m_dim << "," << n_dim << "," << k_dim;
  // Only check equality when both sizes are known (in other words, when
  // neither is set to a minimum dimension size of 1).
  if (k_dim_b != 1 && k_dim != 1 && k_dim_b != k_dim) {
    LOG(ERROR) << "Incompatible Matrix dimensions";
    return ops;
  } else {
    // One of k_dim and k_dim_b might be 1 (minimum dimension size).
    k_dim = std::max(k_dim, k_dim_b);
```

```
1108    }
1109
1110    ops = m_dim * n_dim * k_dim * 2;
1111    VLOG(1) << "Operations for Matmul: " << ops;
1112
1113    if (mat_mul != nullptr) {
1114      mat_mul->m = m_dim;
1115      mat_mul->n = n_dim;
1116      mat_mul->k = k_dim;
1117    }
1118    return ops;
1119  }
1120
1121  bool OpLevelCostEstimator::GenerateBatchMatmulContextFromEinsum(
1122      const OpContext& einsum_context, OpContext* batch_matmul_context,
1123      bool* found_unknown_shapes) const {
1124    // This auxiliary function transforms an einsum OpContext into its equivalent
1125    // Batch Matmul OpContext. The function returns a boolean, which determines
1126    // whether it was successful in generating the output OpContext or not.
1127
1128    // Einsum computes a generalized contraction between tensors of arbitrary
1129    // dimension as defined by the equation written in the Einstein summation
1130    // convention. The number of tensors in the computation and the number of
1131    // contractions can be arbitrarily long. The current model only contemplates
1132    // Einsum equations, which can be translated into a single BatchMatMul
1133    // operation. Einsum operations with more than two operands are not currently
1134    // supported. Subscripts where an axis appears more than once for a single
1135    // input and ellipsis are currently also excluded. See:
1136    // https://www.tensorflow.org/api_docs/python/tf/einsum
1137    // We distinguish four kinds of dimensions, depending on their placement in
1138    // the equation:
1139    // + B: Batch dimensions: Dimensions which appear in both operands and RHS.
1140    // + K: Contracting dimensions: These appear in both inputs but not RHS.
1141    // + M: Operand A dimensions: These appear in the first operand and the RHS.
1142    // + N: Operand B dimensions: These appear in the second operand and the RHS.
1143    // Then, the operation to estimate is BatchMatMul([B,M,K],[B,K,N])
1144
1145    if (batch_matmul_context == nullptr) {
1146      VLOG(1) << "Output context should not be a nullptr.";
1147      return false;
1148    }
1149    if (!IsEinsumCorrectlyFormed(einsum_context)) return false;
1150    const auto& op_info = einsum_context.op_info;
1151    std::vector<std::string> equation_split =
1152        absl::StrSplit(op_info.attr().find("equation")->second.s(), "->");
1153    std::vector<absl::string_view> input_split =
1154        absl::StrSplit(equation_split[0], ',');
1155    const auto& a_input = op_info.inputs(0);
1156    const auto& b_input = op_info.inputs(1);
```

```
1157    absl::string_view rhs_str = equation_split[1];
1158    absl::string_view a_input_str = input_split[0];
1159    absl::string_view b_input_str = input_split[1];
1160
1161    constexpr int kMatrixRank = 2;
1162
1163    bool a_input_shape_unknown = false;
1164    bool b_input_shape_unknown = false;
1165
1166    TensorShapeProto a_input_shape = MaybeGetMinimumShape(
1167        a_input.shape(), std::max(kMatrixRank, a_input.shape().dim_size()),
1168        &a_input_shape_unknown);
1169    TensorShapeProto b_input_shape = MaybeGetMinimumShape(
1170        b_input.shape(), std::max(kMatrixRank, b_input.shape().dim_size()),
1171        &b_input_shape_unknown);
1172
1173    *found_unknown_shapes = a_input_shape_unknown || b_input_shape_unknown ||
1174                            (a_input.shape().dim_size() < kMatrixRank) ||
1175                            (b_input.shape().dim_size() < kMatrixRank);
1176
1177    OpInfo batch_matmul_op_info = op_info;
1178    batch_matmul_op_info.mutable_inputs()->Clear();
1179    batch_matmul_op_info.set_op("BatchMatMul");
1180
1181    AttrValue transpose_attribute;
1182    transpose_attribute.set_b(false);
1183    (*batch_matmul_op_info.mutable_attr())["transpose_a"] = transpose_attribute;
1184    (*batch_matmul_op_info.mutable_attr())["transpose_b"] = transpose_attribute;
1185
1186    OpInfo::TensorProperties* a_matrix = batch_matmul_op_info.add_inputs();
1187    TensorShapeProto* a_matrix_shape = a_matrix->mutable_shape();
1188    a_matrix->set_dtype(a_input.dtype());
1189
1190    OpInfo::TensorProperties* b_matrix = batch_matmul_op_info.add_inputs();
1191    b_matrix->set_dtype(b_input.dtype());
1192    TensorShapeProto* b_matrix_shape = b_matrix->mutable_shape();
1193
1194    TensorShapeProto_Dim m_dim;
1195    TensorShapeProto_Dim n_dim;
1196    TensorShapeProto_Dim k_dim;
1197
1198    m_dim.set_size(1);
1199    n_dim.set_size(1);
1200    k_dim.set_size(1);
1201
1202    for (int i_idx = 0, a_input_str_size = a_input_str.size();
1203         i_idx < a_input_str_size; ++i_idx) {
1204      if (b_input_str.find(a_input_str[i_idx]) == std::string::npos) {
1205        if (rhs_str.find(a_input_str[i_idx]) == std::string::npos) {
```

```
1206          VLOG(1) << "Missing accurate estimator for op: " << op_info.op();
1207          return false;
1208        }
1209
1210        m_dim.set_size(m_dim.size() * a_input_shape.dim(i_idx).size());
1211        continue;
1212      } else if (rhs_str.find(a_input_str[i_idx]) == std::string::npos) {
1213        // The dimension does not appear in the RHS, therefore it is a contracting
1214        // dimension.
1215        k_dim.set_size(k_dim.size() * a_input_shape.dim(i_idx).size());
1216        continue;
1217      }
1218      // It appears in both input operands, therefore we place it as an outer
1219      // dimension for the Batch Matmul.
1220      *(a_matrix_shape->add_dim()) = a_input_shape.dim(i_idx);
1221      *(b_matrix_shape->add_dim()) = a_input_shape.dim(i_idx);
1222    }
1223    for (int i_idx = 0, b_input_str_size = b_input_str.size();
1224         i_idx < b_input_str_size; ++i_idx) {
1225      if (a_input_str.find(b_input_str[i_idx]) == std::string::npos) {
1226        if (rhs_str.find(b_input_str[i_idx]) == std::string::npos) {
1227          VLOG(1) << "Missing accurate estimator for op: " << op_info.op();
1228          return false;
1229        }
1230        n_dim.set_size(n_dim.size() * b_input_shape.dim(i_idx).size());
1231      }
1232    }
1233
1234    // The two inner-most dimensions of the Batch Matmul are added.
1235    *(a_matrix_shape->add_dim()) = m_dim;
1236    *(a_matrix_shape->add_dim()) = k_dim;
1237    *(b_matrix_shape->add_dim()) = k_dim;
1238    *(b_matrix_shape->add_dim()) = n_dim;
1239
1240    *batch_matmul_context = einsum_context;
1241    batch_matmul_context->op_info = batch_matmul_op_info;
1242    return true;
1243  }
1244
1245  int64_t OpLevelCostEstimator::CountBatchMatMulOperations(
1246      const OpInfo& op_info, bool* found_unknown_shapes) {
1247    return CountBatchMatMulOperations(op_info, nullptr, found_unknown_shapes);
1248  }
1249
1250  int64_t OpLevelCostEstimator::CountBatchMatMulOperations(
1251      const OpInfo& op_info, BatchMatMulDimensions* batch_mat_mul,
1252      bool* found_unknown_shapes) {
1253    if (op_info.op() != kBatchMatMul && op_info.op() != kBatchMatMulV2) {
1254      LOG(ERROR) << "Invalid Operation: " << op_info.op();
```

```
1255      // TODO(pcma): Try to separate invalid inputs from unknown shapes
1256      *found_unknown_shapes = true;
1257      return 0;
1258    }
1259    if (op_info.inputs_size() != 2) {
1260      LOG(ERROR) << "Expected 2 inputs but got " << op_info.inputs_size();
1261      // TODO(pcma): Try to separate invalid inputs from unknown shapes
1262      *found_unknown_shapes = true;
1263      return 0;
1264    }
1265
1266    double ops = 0;
1267    const auto& a_input = op_info.inputs(0);
1268    const auto& b_input = op_info.inputs(1);
1269
1270    // BatchMatMul requires inputs of at least matrix shape (rank 2).
1271    // The two most minor dimensions of each input are matrices that
1272    // need to be multiplied together. The other dimensions determine
1273    // the number of such MatMuls.  For example, if the BatchMatMul has
1274    // inputs of shape:
1275    //   a_input_shape = [2, 3, 4, 5]
1276    //   b_input_shape = [2, 3, 5, 6]
1277    // then there are 2*3 = 6 MatMuls of dimensions m = 4, k = 5, n = 6
1278    // in this BatchMatMul.
1279    const int matrix_rank = 2;
1280
1281    bool a_input_shape_unknown = false;
1282    bool b_input_shape_unknown = false;
1283
1284    TensorShapeProto a_input_shape = MaybeGetMinimumShape(
1285        a_input.shape(), std::max(matrix_rank, a_input.shape().dim_size()),
1286        &a_input_shape_unknown);
1287    TensorShapeProto b_input_shape = MaybeGetMinimumShape(
1288        b_input.shape(), std::max(matrix_rank, b_input.shape().dim_size()),
1289        &b_input_shape_unknown);
1290
1291    *found_unknown_shapes = a_input_shape_unknown || b_input_shape_unknown ||
1292                            (a_input.shape().dim_size() < matrix_rank) ||
1293                            (b_input.shape().dim_size() < matrix_rank);
1294
1295    // Compute the number of matmuls as the max indicated at each dimension
1296    // by either input. Note that the shapes do not have to have
1297    // the same rank due to incompleteness.
1298    TensorShapeProto* bigger_rank_shape = &a_input_shape;
1299    TensorShapeProto* smaller_rank_shape = &b_input_shape;
1300    if (b_input_shape.dim_size() > a_input_shape.dim_size()) {
1301      bigger_rank_shape = &b_input_shape;
1302      smaller_rank_shape = &a_input_shape;
1303    }
```

```
1304    int num_matmuls = 1;
1305    for (int b_i = 0,
1306             s_i = smaller_rank_shape->dim_size() - bigger_rank_shape->dim_size();
1307         b_i < bigger_rank_shape->dim_size() - matrix_rank; ++b_i, ++s_i) {
1308      int b_dim = bigger_rank_shape->dim(b_i).size();
1309      int s_dim = 1;
1310      if (s_i >= 0) {
1311        s_dim = smaller_rank_shape->dim(s_i).size();
1312      }
1313      if (batch_mat_mul != nullptr) {
1314        batch_mat_mul->batch_dims.push_back(s_dim);
1315      }
1316      num_matmuls *= std::max(b_dim, s_dim);
1317    }
1318
1319    // Build the MatMul. Note that values are ignored here since we are just
1320    // counting ops (e.g. only shapes matter).
1321    OpInfo matmul_op_info;
1322    matmul_op_info.set_op("MatMul");
1323
1324    AttrValue transpose_a;
1325    transpose_a.set_b(false);
1326    if (op_info.attr().find("adj_x") != op_info.attr().end()) {
1327      transpose_a.set_b(op_info.attr().at("adj_x").b());
1328    }
1329    (*matmul_op_info.mutable_attr())["transpose_a"] = transpose_a;
1330
1331    AttrValue transpose_b;
1332    transpose_b.set_b(false);
1333    if (op_info.attr().find("adj_y") != op_info.attr().end()) {
1334      transpose_b.set_b(op_info.attr().at("adj_y").b());
1335    }
1336    (*matmul_op_info.mutable_attr())["transpose_b"] = transpose_b;
1337
1338    OpInfo::TensorProperties* a_matrix = matmul_op_info.add_inputs();
1339    a_matrix->set_dtype(a_input.dtype());
1340    TensorShapeProto* a_matrix_shape = a_matrix->mutable_shape();
1341    for (int i = std::max(0, a_input_shape.dim_size() - matrix_rank);
1342         i < a_input_shape.dim_size(); ++i) {
1343      *(a_matrix_shape->add_dim()) = a_input_shape.dim(i);
1344    }
1345
1346    OpInfo::TensorProperties* b_matrix = matmul_op_info.add_inputs();
1347    b_matrix->set_dtype(b_input.dtype());
1348    TensorShapeProto* b_matrix_shape = b_matrix->mutable_shape();
1349    for (int i = std::max(0, b_input_shape.dim_size() - matrix_rank);
1350         i < b_input_shape.dim_size(); ++i) {
1351      *(b_matrix_shape->add_dim()) = b_input_shape.dim(i);
1352    }
```

```
1353    if (batch_mat_mul != nullptr) {
1354      batch_mat_mul->matmul_dims.m = (transpose_a.b())
1355                                         ? a_matrix_shape->dim(1).size()
1356                                         : a_matrix_shape->dim(0).size();
1357      batch_mat_mul->matmul_dims.k = (transpose_a.b())
1358                                         ? a_matrix_shape->dim(0).size()
1359                                         : a_matrix_shape->dim(1).size();
1360      batch_mat_mul->matmul_dims.n = (transpose_b.b())
1361                                         ? b_matrix_shape->dim(0).size()
1362                                         : b_matrix_shape->dim(1).size();
1363    }
1364
1365    for (int i = 0; i < num_matmuls; ++i) {
1366      bool matmul_unknown_shapes = false;
1367      ops += CountMatMulOperations(matmul_op_info, &matmul_unknown_shapes);
1368      *found_unknown_shapes |= matmul_unknown_shapes;
1369    }
1370    return ops;
1371  }
1372
1373  bool GetTensorShapeProtoFromTensorProto(const TensorProto& tensor_proto,
1374                                          TensorShapeProto* tensor_shape_proto) {
1375    tensor_shape_proto->Clear();
1376    // First convert TensorProto into Tensor class so that it correctly parses
1377    // data values within TensorProto (whether it's in int_val, int64_val,
1378    // tensor_content, or anything.
1379    Tensor tensor(tensor_proto.dtype());
1380    if (!tensor.FromProto(tensor_proto)) {
1381      LOG(WARNING) << "GetTensorShapeProtoFromTensorProto() -- "
1382                   << "failed to parse TensorProto: "
1383                   << tensor_proto.DebugString();
1384      return false;
1385    }
1386    if (tensor.dims() != 1) {
1387      LOG(WARNING) << "GetTensorShapeProtoFromTensorProto() -- "
1388                   << "tensor is not 1D: " << tensor.dims();
1389      return false;
1390    }
1391    // Then, convert it back to TensorProto using AsProtoField, which makes sure
1392    // the data is in int_val, int64_val, or such repeated data fields, not in
1393    // tensor_content.
1394    TensorProto temp_tensor;
1395    tensor.AsProtoField(&temp_tensor);
1396
1397  #define TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(type)        \
1398    do {                                                   \
1399      for (const auto& value : temp_tensor.type##_val()) { \
1400        tensor_shape_proto->add_dim()->set_size(value);    \
1401      }                                                    \
```

```
  } while (0)

  if (tensor.dtype() == DT_INT32 || tensor.dtype() == DT_INT16 ||
      tensor.dtype() == DT_INT8 || tensor.dtype() == DT_UINT8) {
    TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(int);
  } else if (tensor.dtype() == DT_INT64) {
    TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(int64);
  } else if (tensor.dtype() == DT_UINT32) {
    TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(uint32);
  } else if (tensor.dtype() == DT_UINT64) {
    TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(uint64);
  } else {
    LOG(WARNING) << "GetTensorShapeProtoFromTensorProto() -- "
                 << "Unsupported dtype: " << tensor.dtype();
    return false;
  }
#undef TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO

  return true;
}

// TODO(cliffy): Dedup this method and CountConv2DBackpropFilterOperations.
int64_t OpLevelCostEstimator::CountConv2DBackpropInputOperations(
    const OpInfo& op_info, ConvolutionDimensions* returned_conv_dims,
    bool* found_unknown_shapes) {
  int64_t ops = 0;

  DCHECK(op_info.op() == kConv2dBackpropInput ||
         op_info.op() == kDepthwiseConv2dNativeBackpropInput)
      << "Invalid Operation: not kConv2dBackpropInput nor"
         "kDepthwiseConv2dNativeBackpropInput";

  if (op_info.inputs_size() < 2) {
    // TODO(pcma): Try to separate invalid inputs from unknown shapes
    *found_unknown_shapes = true;
    return ops;
  }

  TensorShapeProto input_shape;
  bool shape_found = false;
  if (op_info.inputs(0).has_value()) {
    const TensorProto& value = op_info.inputs(0).value();
    shape_found = GetTensorShapeProtoFromTensorProto(value, &input_shape);
  }
  if (!shape_found && op_info.outputs_size() == 1) {
    input_shape = op_info.outputs(0).shape();
    shape_found = true;
  }
  if (!shape_found) {
```

```
    // Set the minimum filter size that's feasible.
    input_shape.Clear();
    for (int i = 0; i < 4; ++i) {
      input_shape.add_dim()->set_size(1);
    }
    *found_unknown_shapes = true;
  }

  ConvolutionDimensions conv_dims = ConvolutionDimensionsFromInputs(
      input_shape, op_info.inputs(1).shape(), op_info, found_unknown_shapes);

  ops = conv_dims.batch;
  ops *= conv_dims.ox * conv_dims.oy;
  ops *= conv_dims.kx * conv_dims.ky;
  if (op_info.op() == kConv2dBackpropInput) {
    ops *= conv_dims.kz * conv_dims.oz;
  } else {
    // conv_dims always use forward path definition regardless
    conv_dims.oz *= conv_dims.iz;
    ops *= conv_dims.oz;
  }
  ops *= kOpsPerMac;

  VLOG(1) << "Operations for" << op_info.op() << "  " << ops;

  if (returned_conv_dims != nullptr) {
    *returned_conv_dims = conv_dims;
  }
  return ops;
}

int64_t OpLevelCostEstimator::CountConv2DBackpropFilterOperations(
    const OpInfo& op_info, ConvolutionDimensions* returned_conv_dims,
    bool* found_unknown_shapes) {
  int64_t ops = 0;

  DCHECK(op_info.op() == kConv2dBackpropFilter ||
         op_info.op() == kDepthwiseConv2dNativeBackpropFilter)
      << "Invalid Operation: not kConv2dBackpropFilter nor"
         "kDepthwiseConv2dNativeBackpropFilter";

  TensorShapeProto filter_shape;
  bool shape_found = false;
  if (op_info.inputs_size() >= 2 && op_info.inputs(1).has_value()) {
    const TensorProto& value = op_info.inputs(1).value();
    shape_found = GetTensorShapeProtoFromTensorProto(value, &filter_shape);
  }
  if (!shape_found && op_info.outputs_size() == 1) {
    filter_shape = op_info.outputs(0).shape();
```

```
    shape_found = true;
  }
  if (!shape_found) {
    // Set the minimum filter size that's feasible.
    filter_shape.Clear();
    for (int i = 0; i < 4; ++i) {
      filter_shape.add_dim()->set_size(1);
    }
    *found_unknown_shapes = true;
  }

  if (op_info.inputs_size() < 1) {
    // TODO(pcma): Try to separate invalid inputs from unknown shapes
    *found_unknown_shapes = true;
    return ops;
  }
  ConvolutionDimensions conv_dims = ConvolutionDimensionsFromInputs(
      op_info.inputs(0).shape(), filter_shape, op_info, found_unknown_shapes);

  ops = conv_dims.batch;
  ops *= conv_dims.ox * conv_dims.oy;
  ops *= conv_dims.kx * conv_dims.ky;
  if (op_info.op() == kConv2dBackpropFilter) {
    ops *= conv_dims.kz * conv_dims.oz;
  } else {
    // conv_dims always use forward path definition regardless
    conv_dims.oz *= conv_dims.iz;
    ops *= conv_dims.oz;
  }
  ops *= kOpsPerMac;
  VLOG(1) << "Operations for" << op_info.op() << "  " << ops;

  if (returned_conv_dims != nullptr) {
    *returned_conv_dims = conv_dims;
  }
  return ops;
}

int64_t OpLevelCostEstimator::CalculateTensorElementCount(
    const OpInfo::TensorProperties& tensor, bool* found_unknown_shapes) {
  VLOG(2) << "   with " << DataTypeString(tensor.dtype()) << " tensor of shape "
          << tensor.shape().DebugString();
  int64_t tensor_size = 1;
  int num_dims = std::max(1, tensor.shape().dim_size());
  auto tensor_shape =
      MaybeGetMinimumShape(tensor.shape(), num_dims, found_unknown_shapes);
  for (const auto& dim : tensor_shape.dim()) {
    tensor_size *= dim.size();
  }
```

```
1549    return tensor_size;
1550  }
1551
1552  int64_t OpLevelCostEstimator::CalculateTensorSize(
1553      const OpInfo::TensorProperties& tensor, bool* found_unknown_shapes) {
1554    int64_t count = CalculateTensorElementCount(tensor, found_unknown_shapes);
1555    int size = DataTypeSize(BaseType(tensor.dtype()));
1556    VLOG(2) << "Count: " << count << " DataTypeSize: " << size;
1557    return count * size;
1558  }
1559
1560  int64_t OpLevelCostEstimator::CalculateInputSize(const OpInfo& op_info,
1561                                                   bool* found_unknown_shapes) {
1562    int64_t total_input_size = 0;
1563    for (auto& input : op_info.inputs()) {
1564      int64_t input_size = CalculateTensorSize(input, found_unknown_shapes);
1565      total_input_size += input_size;
1566      VLOG(1) << "Input Size: " << input_size
1567              << " Total Input Size:" << total_input_size;
1568    }
1569    return total_input_size;
1570  }
1571
1572  std::vector<int64_t> OpLevelCostEstimator::CalculateInputTensorSize(
1573      const OpInfo& op_info, bool* found_unknown_shapes) {
1574    std::vector<int64_t> input_tensor_size;
1575    input_tensor_size.reserve(op_info.inputs().size());
1576    for (auto& input : op_info.inputs()) {
1577      input_tensor_size.push_back(
1578          CalculateTensorSize(input, found_unknown_shapes));
1579    }
1580    return input_tensor_size;
1581  }
1582
1583  int64_t OpLevelCostEstimator::CalculateLargestInputCount(
1584      const OpInfo& op_info, bool* found_unknown_shapes) {
1585    int64_t largest_input_count = 0;
1586    for (auto& input : op_info.inputs()) {
1587      int64_t input_count =
1588          CalculateTensorElementCount(input, found_unknown_shapes);
1589      if (input_count > largest_input_count) {
1590        largest_input_count = input_count;
1591      }
1592      VLOG(1) << "Input Count: " << input_count
1593              << " Largest Input Count:" << largest_input_count;
1594    }
1595    return largest_input_count;
1596  }
1597
```

```
1598 int64_t OpLevelCostEstimator::CalculateOutputSize(const OpInfo& op_info,
1599                                                    bool* found_unknown_shapes) {
1600   int64_t total_output_size = 0;
1601   // Use float as default for calculations.
1602   for (const auto& output : op_info.outputs()) {
1603     DataType dt = output.dtype();
1604     const auto& original_output_shape = output.shape();
1605     int64_t output_size = DataTypeSize(BaseType(dt));
1606     int num_dims = std::max(1, original_output_shape.dim_size());
1607     auto output_shape = MaybeGetMinimumShape(original_output_shape, num_dims,
1608                                              found_unknown_shapes);
1609     for (const auto& dim : output_shape.dim()) {
1610       output_size *= dim.size();
1611     }
1612     total_output_size += output_size;
1613     VLOG(1) << "Output Size: " << output_size
1614             << " Total Output Size:" << total_output_size;
1615   }
1616   return total_output_size;
1617 }
1618
1619 std::vector<int64_t> OpLevelCostEstimator::CalculateOutputTensorSize(
1620     const OpInfo& op_info, bool* found_unknown_shapes) {
1621   std::vector<int64_t> output_tensor_size;
1622   output_tensor_size.reserve(op_info.outputs().size());
1623   // Use float as default for calculations.
1624   for (const auto& output : op_info.outputs()) {
1625     DataType dt = output.dtype();
1626     const auto& original_output_shape = output.shape();
1627     int64_t output_size = DataTypeSize(BaseType(dt));
1628     int num_dims = std::max(1, original_output_shape.dim_size());
1629     auto output_shape = MaybeGetMinimumShape(original_output_shape, num_dims,
1630                                              found_unknown_shapes);
1631     for (const auto& dim : output_shape.dim()) {
1632       output_size *= dim.size();
1633     }
1634     output_tensor_size.push_back(output_size);
1635   }
1636   return output_tensor_size;
1637 }
1638
1639 Status OpLevelCostEstimator::PredictDefaultNodeCosts(
1640     const int64_t num_compute_ops, const OpContext& op_context,
1641     bool* found_unknown_shapes, NodeCosts* node_costs) {
1642   const auto& op_info = op_context.op_info;
1643   node_costs->num_compute_ops = num_compute_ops;
1644   node_costs->num_input_bytes_accessed =
1645       CalculateInputTensorSize(op_info, found_unknown_shapes);
1646   node_costs->num_output_bytes_accessed =
```

```
      CalculateOutputTensorSize(op_info, found_unknown_shapes);
  node_costs->max_memory = node_costs->num_total_output_bytes();
  if (*found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

bool HasZeroDim(const OpInfo& op_info) {
  for (int i = 0; i < op_info.inputs_size(); ++i) {
    const auto& input = op_info.inputs(i);
    for (int j = 0; j < input.shape().dim_size(); ++j) {
      const auto& dim = input.shape().dim(j);
      if (dim.size() == 0) {
        VLOG(1) << "Convolution config has zero dim "
                << op_info.ShortDebugString();
        return true;
      }
    }
  }
  return false;
}

Status OpLevelCostEstimator::PredictConv2D(const OpContext& op_context,
                                           NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  if (HasZeroDim(op_info)) {
    node_costs->num_nodes_with_unknown_shapes = 1;
    return errors::InvalidArgument("Conv2D op includes zero dimension: ",
                                   op_info.ShortDebugString());
  }
  bool found_unknown_shapes = false;
  int64_t num_compute_ops =
      CountConv2DOperations(op_info, &found_unknown_shapes);
  return PredictDefaultNodeCosts(num_compute_ops, op_context,
                                 &found_unknown_shapes, node_costs);
}

Status OpLevelCostEstimator::PredictConv2DBackpropInput(
    const OpContext& op_context, NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  if (HasZeroDim(op_info)) {
    node_costs->num_nodes_with_unknown_shapes = 1;
    return errors::InvalidArgument(
        "Conv2DBackpropInput op includes zero dimension",
        op_info.ShortDebugString());
  }
  bool found_unknown_shapes = false;
```

```
  int64_t num_compute_ops = CountConv2DBackpropInputOperations(
      op_info, nullptr, &found_unknown_shapes);
  return PredictDefaultNodeCosts(num_compute_ops, op_context,
                                 &found_unknown_shapes, node_costs);
}

Status OpLevelCostEstimator::PredictConv2DBackpropFilter(
    const OpContext& op_context, NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  if (HasZeroDim(op_info)) {
    node_costs->num_nodes_with_unknown_shapes = 1;
    return errors::InvalidArgument(
        "Conv2DBackpropFilter op includes zero dimension",
        op_info.ShortDebugString());
  }
  bool found_unknown_shapes = false;
  int64_t num_compute_ops = CountConv2DBackpropFilterOperations(
      op_info, nullptr, &found_unknown_shapes);
  return PredictDefaultNodeCosts(num_compute_ops, op_context,
                                 &found_unknown_shapes, node_costs);
}

Status OpLevelCostEstimator::PredictFusedConv2DBiasActivation(
    const OpContext& op_context, NodeCosts* node_costs) const {
  // FusedConv2DBiasActivation computes a fused kernel which implements:
  // 2D convolution, adds side input with separate scaling on convolution and
  // side inputs, then adds bias, and finally applies the ReLU activation
  // function to the result:
  //
  // Input -> Conv2D  ->  Add  -> BiasAdd  -> ReLU
  //            ^          ^         ^
  //          Filter   Side Input   Bias
  //
  // Note that when adding the side input, the operation multiplies the output
  // of Conv2D by conv_input_scale, confusingly, and the side_input by
  // side_input_scale.
  //
  // Note that in the special case that side_input_scale is 0, which we infer
  // from side_input having dimensions [], we skip that addition operation.
  //
  // For more information, see
  // contrib/fused_conv/kernels/fused_conv2d_bias_activation_op.cc

  // TODO(yaozhang): Support NHWC_VECT_W.
  std::string data_format = GetDataFormat(op_context.op_info);
  if (data_format != "NCHW" && data_format != "NHWC" &&
      data_format != "NCHW_VECT_C") {
    return errors::InvalidArgument(
        "Unsupported data format (", data_format,
```

```
1745          ") for op: ", op_context.op_info.ShortDebugString());
1746    }
1747    std::string filter_format = GetFilterFormat(op_context.op_info);
1748    if (filter_format != "HWIO" && filter_format != "OIHW" &&
1749        filter_format != "OIHW_VECT_I") {
1750      return errors::InvalidArgument(
1751          "Unsupported filter format (", filter_format,
1752          ") for op: ", op_context.op_info.ShortDebugString());
1753    }
1754
1755    auto& conv_input = op_context.op_info.inputs(0);
1756    auto& filter = op_context.op_info.inputs(1);
1757    auto& side_input = op_context.op_info.inputs(3);
1758    auto& conv_input_scale = op_context.op_info.inputs(4);
1759    auto& side_input_scale = op_context.op_info.inputs(5);
1760
1761    // Manually compute our convolution dimensions.
1762    bool found_unknown_shapes = false;
1763    auto dims = ConvolutionDimensionsFromInputs(
1764        conv_input.shape(), filter.shape(), op_context.op_info,
1765        &found_unknown_shapes);
1766    OpInfo::TensorProperties output;
1767    if (data_format == "NCHW" || data_format == "NCHW_VECT_C") {
1768      output = DescribeTensor(DT_FLOAT, {dims.batch, dims.oz, dims.oy, dims.ox});
1769    } else if (data_format == "NHWC") {
1770      output = DescribeTensor(DT_FLOAT, {dims.batch, dims.oy, dims.ox, dims.oz});
1771    }
1772
1773    // Add the operations the fused op always computes.
1774    std::vector<OpContext> component_ops = {
1775        FusedChildContext(op_context, "Conv2D", output, {conv_input, filter}),
1776        FusedChildContext(op_context, "Mul", output, {output, conv_input_scale}),
1777        FusedChildContext(
1778            op_context, "BiasAdd", output,
1779            {output, output}),  // Note we're no longer using bias at all
1780        FusedChildContext(op_context, "Relu", output, {output})};
1781
1782    // Add our side_input iff it's non-empty.
1783    if (side_input.shape().dim_size() > 0) {
1784      component_ops.push_back(FusedChildContext(op_context, "Mul", side_input,
1785                                                {side_input, side_input_scale}));
1786      component_ops.push_back(FusedChildContext(
1787          op_context, "Add", output,
1788          {output, output}));  // Note that we're not using side_input here
1789    }
1790
1791    // Construct an op_context which definitely has our output shape.
1792    auto op_context_with_output = op_context;
1793    op_context_with_output.op_info.mutable_outputs()->Clear();
```

```
1794    *op_context_with_output.op_info.mutable_outputs()->Add() = output;
1795
1796    // Construct component operations and run the cost computation.
1797    if (found_unknown_shapes) {
1798      node_costs->inaccurate = true;
1799      node_costs->num_nodes_with_unknown_shapes = 1;
1800    }
1801    return PredictFusedOp(op_context_with_output, component_ops, node_costs);
1802  }
1803
1804  Status OpLevelCostEstimator::PredictMatMul(const OpContext& op_context,
1805                                             NodeCosts* node_costs) const {
1806    const auto& op_info = op_context.op_info;
1807    bool found_unknown_shapes = false;
1808    int64_t num_compute_ops =
1809        CountMatMulOperations(op_info, &found_unknown_shapes);
1810    return PredictDefaultNodeCosts(num_compute_ops, op_context,
1811                                   &found_unknown_shapes, node_costs);
1812  }
1813
1814  Status OpLevelCostEstimator::PredictEinsum(const OpContext& op_context,
1815                                             NodeCosts* node_costs) const {
1816    const auto& op_info = op_context.op_info;
1817
1818    auto it = op_info.attr().find("equation");
1819    if (it == op_info.attr().end()) {
1820      return errors::InvalidArgument("Einsum op doesn't have equation attr: ",
1821                                     op_info.ShortDebugString());
1822    }
1823
1824    OpContext batch_matmul_op_context;
1825    bool found_unknown_shapes = false;
1826    bool success = GenerateBatchMatmulContextFromEinsum(
1827        op_context, &batch_matmul_op_context, &found_unknown_shapes);
1828    if (found_unknown_shapes) {
1829      node_costs->inaccurate = true;
1830      node_costs->num_nodes_with_unknown_shapes = 1;
1831    }
1832    if (!success) {
1833      return PredictCostOfAnUnknownOp(op_context, node_costs);
1834    }
1835    return PredictNodeCosts(batch_matmul_op_context, node_costs);
1836  }
1837
1838  Status OpLevelCostEstimator::PredictSparseTensorDenseMatMul(
1839      const OpContext& op_context, NodeCosts* node_costs) const {
1840    const auto& op_info = op_context.op_info;
1841    bool found_unknown_shapes = false;
1842    // input[0]: indices in sparse matrix a
```

```
1843    // input[1]: values in sparse matrix a
1844    // input[2]: shape of matrix a
1845    // input[3]: matrix b
1846    // See
1847    // https://github.com/tensorflow/tensorflow/blob/9a43dfeac5/tensorflow/core/ops/sparse_ops.cc#L8
1848    int64_t num_elems_in_a =
1849        CalculateTensorElementCount(op_info.inputs(1), &found_unknown_shapes);
1850    auto b_matrix = op_info.inputs(3);
1851    auto b_matrix_shape =
1852        MaybeGetMinimumShape(b_matrix.shape(), 2, &found_unknown_shapes);
1853    int64_t n_dim = b_matrix_shape.dim(1).size();
1854
1855    // Each element in A is multiplied and added with an element from each column
1856    // in b.
1857    const int64_t op_count = kOpsPerMac * num_elems_in_a * n_dim;
1858
1859    int64_t a_indices_input_size =
1860        CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
1861    int64_t a_values_input_size =
1862        CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
1863    int64_t a_shape_input_size =
1864        CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
1865    int64_t b_input_size =
1866        num_elems_in_a * n_dim * DataTypeSize(BaseType(b_matrix.dtype()));
1867    int64_t output_size = CalculateOutputSize(op_info, &found_unknown_shapes);
1868
1869    node_costs->num_compute_ops = op_count;
1870    node_costs->num_input_bytes_accessed = {a_indices_input_size,
1871                                            a_values_input_size,
1872                                            a_shape_input_size, b_input_size};
1873    node_costs->num_output_bytes_accessed = {output_size};
1874    if (found_unknown_shapes) {
1875      node_costs->inaccurate = true;
1876      node_costs->num_nodes_with_unknown_shapes = 1;
1877    }
1878    return Status::OK();
1879  }
1880
1881  Status OpLevelCostEstimator::PredictNoOp(const OpContext& op_context,
1882                                           NodeCosts* node_costs) const {
1883    const auto& op_info = op_context.op_info;
1884    VLOG(1) << "Op:" << op_info.op() << " Execution Time 0 (ns)";
1885    // By default, NodeCosts is initialized to zero ops and bytes.
1886    return Status::OK();
1887  }
1888
1889  Status OpLevelCostEstimator::PredictPureMemoryOp(const OpContext& op_context,
1890                                                   NodeCosts* node_costs) const {
1891    // Each output element is a copy of some element from input, with no required
```

```
1892    // computation, so just compute memory costs.
1893    bool found_unknown_shapes = false;
1894    node_costs->num_nodes_with_pure_memory_op = 1;
1895    return PredictDefaultNodeCosts(0, op_context, &found_unknown_shapes,
1896                                   node_costs);
1897  }
1898
1899  Status OpLevelCostEstimator::PredictIdentity(const OpContext& op_context,
1900                                               NodeCosts* node_costs) const {
1901    const auto& op_info = op_context.op_info;
1902    VLOG(1) << "Op:" << op_info.op() << " Minimum cost for Identity";
1903    node_costs->minimum_cost_op = true;
1904    node_costs->num_compute_ops = kMinComputeOp;
1905    // Identity op internally pass input tensor buffer's pointer to the output
1906    // tensor buffer; no actual memory operation.
1907    node_costs->num_input_bytes_accessed = {0};
1908    node_costs->num_output_bytes_accessed = {0};
1909    bool inaccurate = false;
1910    node_costs->max_memory = CalculateOutputSize(op_info, &inaccurate);
1911    if (inaccurate) {
1912      node_costs->inaccurate = true;
1913      node_costs->num_nodes_with_unknown_shapes = 1;
1914    }
1915    return Status::OK();
1916  }
1917
1918  Status OpLevelCostEstimator::PredictVariable(const OpContext& op_context,
1919                                               NodeCosts* node_costs) const {
1920    const auto& op_info = op_context.op_info;
1921    VLOG(1) << "Op:" << op_info.op() << " Minimum cost for Variable";
1922    node_costs->minimum_cost_op = true;
1923    node_costs->num_compute_ops = kMinComputeOp;
1924    // Variables are persistent ops; initialized before step; hence, no memory
1925    // cost.
1926    node_costs->num_input_bytes_accessed = {0};
1927    node_costs->num_output_bytes_accessed = {0};
1928    bool inaccurate = false;
1929    node_costs->persistent_memory = CalculateOutputSize(op_info, &inaccurate);
1930    if (inaccurate) {
1931      node_costs->inaccurate = true;
1932      node_costs->num_nodes_with_unknown_shapes = 1;
1933    }
1934    return Status::OK();
1935  }
1936
1937  Status OpLevelCostEstimator::PredictBatchMatMul(const OpContext& op_context,
1938                                                  NodeCosts* node_costs) const {
1939    const auto& op_info = op_context.op_info;
1940    bool found_unknown_shapes = false;
```

```
1941    int64_t num_compute_ops =
1942        CountBatchMatMulOperations(op_info, &found_unknown_shapes);
1943    return PredictDefaultNodeCosts(num_compute_ops, op_context,
1944                                   &found_unknown_shapes, node_costs);
1945  }
1946
1947  Status OpLevelCostEstimator::PredictMetadata(const OpContext& op_context,
1948                                               NodeCosts* node_costs) const {
1949    const auto& op_info = op_context.op_info;
1950    node_costs->minimum_cost_op = true;
1951    node_costs->num_compute_ops = kMinComputeOp;
1952    node_costs->num_input_bytes_accessed = {0};
1953    node_costs->num_output_bytes_accessed = {0};
1954    bool inaccurate = false;
1955    node_costs->max_memory = CalculateOutputSize(op_info, &inaccurate);
1956    if (inaccurate) {
1957      node_costs->inaccurate = true;
1958      node_costs->num_nodes_with_unknown_shapes = 1;
1959    }
1960    return Status::OK();
1961  }
1962
1963  Status OpLevelCostEstimator::PredictGatherOrSlice(const OpContext& op_context,
1964                                                    NodeCosts* node_costs) const {
1965    // Gather & Slice ops can have a very large input, but only access a small
1966    // part of it. For these op the size of the output determines the memory cost.
1967    const auto& op_info = op_context.op_info;
1968
1969    const int inputs_needed = op_info.op() == "Slice" ? 3 : 2;
1970    if (op_info.outputs_size() == 0 || op_info.inputs_size() < inputs_needed) {
1971      return errors::InvalidArgument(
1972          op_info.op(),
1973          " Op doesn't have valid input / output: ", op_info.ShortDebugString());
1974    }
1975
1976    bool unknown_shapes = false;
1977
1978    // Each output element is a copy of some element from input.
1979    // For roofline estimate we assume each copy has a unit cost.
1980    const int64_t op_count =
1981        CalculateTensorElementCount(op_info.outputs(0), &unknown_shapes);
1982    node_costs->num_compute_ops = op_count;
1983
1984    const int64_t output_size = CalculateOutputSize(op_info, &unknown_shapes);
1985    node_costs->num_output_bytes_accessed = {output_size};
1986
1987    node_costs->num_input_bytes_accessed.reserve(op_info.inputs().size());
1988    int64_t input_size = output_size;
1989    // Note that input(0) byte accessed is not equal to input(0) tensor size.
```

```
  // It's equal to the output size; though, input access is indexed gather or
  // slice (ignore duplicate indices).
  node_costs->num_input_bytes_accessed.push_back(input_size);
  int begin_input_index = 1;
  int end_input_index;
  if (op_info.op() == "Slice") {
    // Slice: 'input' (omitted), 'begin', 'size'
    end_input_index = 3;
  } else if (op_info.op() == "StridedSlice") {
    // StridedSlice: 'input' (omitted), 'begin', 'end', 'strides'
    end_input_index = 4;
  } else {
    // Gather, GatherV2, GatherNd: 'params' (omitted), 'indices'
    end_input_index = 2;
  }
  for (int i = begin_input_index; i < end_input_index; ++i) {
    node_costs->num_input_bytes_accessed.push_back(
        CalculateTensorElementCount(op_info.inputs(i), &unknown_shapes));
  }
  if (unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

Status OpLevelCostEstimator::PredictScatter(const OpContext& op_context,
                                            NodeCosts* node_costs) const {
  // Scatter ops sparsely access a reference input and output tensor.
  const auto& op_info = op_context.op_info;
  bool found_unknown_shapes = false;

  // input[0]: ref tensor that will be sparsely accessed
  // input[1]: indices - A tensor of indices into the first dimension of ref.
  // input[2]: updates where updates.shape = indices.shape + ref.shape[1:]
  // See
  // https://www.tensorflow.org/api_docs/python/tf/scatter_add and
  // https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/state_ops.cc#L146

  const int64_t num_indices =
      CalculateTensorElementCount(op_info.inputs(1), &found_unknown_shapes);

  int64_t num_elems_in_ref_per_index = 1;
  auto ref_tensor_shape = MaybeGetMinimumShape(
      op_info.inputs(0).shape(), op_info.inputs(0).shape().dim_size(),
      &found_unknown_shapes);
  for (int i = 1; i < ref_tensor_shape.dim().size(); ++i) {
    num_elems_in_ref_per_index *= ref_tensor_shape.dim(i).size();
  }
```

```
2039    const int64_t op_count = num_indices * num_elems_in_ref_per_index;
2040    node_costs->num_compute_ops = op_count;
2041
2042    // Sparsely access ref so input size depends on the number of operations
2043    int64_t ref_input_size =
2044        op_count * DataTypeSize(BaseType(op_info.inputs(0).dtype()));
2045    int64_t indices_input_size =
2046        CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
2047    int64_t updates_input_size =
2048        CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
2049    node_costs->num_input_bytes_accessed = {ref_input_size, indices_input_size,
2050                                            updates_input_size};
2051
2052    // Sparsely access ref so output size depends on the number of operations
2053    int64_t output_size =
2054        op_count * DataTypeSize(BaseType(op_info.outputs(0).dtype()));
2055    node_costs->num_output_bytes_accessed = {output_size};
2056
2057    if (found_unknown_shapes) {
2058      node_costs->inaccurate = true;
2059      node_costs->num_nodes_with_unknown_shapes = 1;
2060    }
2061    return Status::OK();
2062  }
2063
2064  Status OpLevelCostEstimator::PredictFusedOp(
2065      const OpContext& op_context,
2066      const std::vector<OpContext>& fused_op_contexts,
2067      NodeCosts* node_costs) const {
2068    // Note that PredictDefaultNodeCosts will get the correct memory costs from
2069    // the node's inputs and outputs; but we don't want to have to re-implement
2070    // the logic for computing the operation count of each of our component
2071    // operations here; so we simply add the compute times of each component
2072    // operation, then update the cost.
2073    bool found_unknown_shapes = false;
2074    Status s =
2075        PredictDefaultNodeCosts(0, op_context, &found_unknown_shapes, node_costs);
2076
2077    for (auto& fused_op : fused_op_contexts) {
2078      NodeCosts fused_node_costs;
2079      s.Update(PredictNodeCosts(fused_op, &fused_node_costs));
2080      node_costs->num_compute_ops += fused_node_costs.num_compute_ops;
2081      node_costs->inaccurate |= fused_node_costs.inaccurate;
2082      // Set, not increment. Note that we are predicting the cost of one fused
2083      // node, not a function node composed of many nodes.
2084      node_costs->num_nodes_with_unknown_shapes |=
2085          fused_node_costs.num_nodes_with_unknown_shapes;
2086      node_costs->num_nodes_with_unknown_op_type |=
2087          fused_node_costs.num_nodes_with_unknown_op_type;
```

```
2088      node_costs->num_nodes_with_pure_memory_op |=
2089          fused_node_costs.num_nodes_with_pure_memory_op;
2090    }
2091
2092    return Status::OK();
2093  }
2094
2095  /* static */
2096  OpContext OpLevelCostEstimator::FusedChildContext(
2097      const OpContext& parent, const std::string& op_name,
2098      const OpInfo::TensorProperties& output,
2099      const std::vector<OpInfo::TensorProperties>& inputs) {
2100    // Setup the base parameters of our new context.
2101    OpContext new_context;
2102    new_context.name = op_name;
2103    new_context.device_name = parent.device_name;
2104    new_context.op_info = parent.op_info;
2105    new_context.op_info.set_op(op_name);
2106
2107    // Setup the inputs of our new context.
2108    new_context.op_info.mutable_inputs()->Clear();
2109    for (const auto& input : inputs) {
2110      *new_context.op_info.mutable_inputs()->Add() = input;
2111    }
2112
2113    // Setup the output of our new context.
2114    new_context.op_info.mutable_outputs()->Clear();
2115    *new_context.op_info.mutable_outputs()->Add() = output;
2116
2117    return new_context;
2118  }
2119
2120  /* static */
2121  OpInfo::TensorProperties OpLevelCostEstimator::DescribeTensor(
2122      DataType type, const std::vector<int64_t>& dims) {
2123    OpInfo::TensorProperties ret;
2124    ret.set_dtype(type);
2125
2126    auto shape = ret.mutable_shape();
2127    for (const int dim : dims) {
2128      shape->add_dim()->set_size(dim);
2129    }
2130
2131    return ret;
2132  }
2133
2134  /* static */
2135  OpLevelCostEstimator::ConvolutionDimensions
2136  OpLevelCostEstimator::OpDimensionsFromInputs(
```

```
2137      const TensorShapeProto& original_image_shape, const OpInfo& op_info,
2138      bool* found_unknown_shapes) {
2139    VLOG(2) << "op features: " << op_info.DebugString();
2140    VLOG(2) << "Original image shape: " << original_image_shape.DebugString();
2141    auto image_shape =
2142        MaybeGetMinimumShape(original_image_shape, 4, found_unknown_shapes);
2143    VLOG(2) << "Image shape: " << image_shape.DebugString();
2144
2145    int x_index, y_index, channel_index;
2146    const std::string& data_format = GetDataFormat(op_info);
2147    if (data_format == "NCHW") {
2148      channel_index = 1;
2149      y_index = 2;
2150      x_index = 3;
2151    } else {
2152      y_index = 1;
2153      x_index = 2;
2154      channel_index = 3;
2155    }
2156    int64_t batch = image_shape.dim(0).size();
2157    int64_t ix = image_shape.dim(x_index).size();
2158    int64_t iy = image_shape.dim(y_index).size();
2159    int64_t iz = image_shape.dim(channel_index).size();
2160
2161    // Note that FusedBatchNorm doesn't have ksize attr, but GetKernelSize returns
2162    // {1, 1, 1, 1} in that case.
2163    std::vector<int64_t> ksize = GetKernelSize(op_info);
2164    int64_t kx = ksize[x_index];
2165    int64_t ky = ksize[y_index];
2166    // These ops don't support groupwise operation, therefore kz == iz.
2167    int64_t kz = iz;
2168
2169    std::vector<int64_t> strides = GetStrides(op_info);
2170    int64_t sx = strides[x_index];
2171    int64_t sy = strides[y_index];
2172    const auto padding = GetPadding(op_info);
2173
2174    int64_t ox = GetOutputSize(ix, kx, sx, padding);
2175    int64_t oy = GetOutputSize(iy, ky, sy, padding);
2176    int64_t oz = iz;
2177
2178    OpLevelCostEstimator::ConvolutionDimensions conv_dims = {
2179        batch, ix, iy, iz, kx, ky, kz, oz, ox, oy, sx, sy, padding};
2180    return conv_dims;
2181  }
2182
2183  Status OpLevelCostEstimator::PredictMaxPool(const OpContext& op_context,
2184                                              NodeCosts* node_costs) const {
2185    bool found_unknown_shapes = false;
```

```
  const auto& op_info = op_context.op_info;
  // x: op_info.inputs(0)
  ConvolutionDimensions dims = OpDimensionsFromInputs(
      op_info.inputs(0).shape(), op_info, &found_unknown_shapes);
  // kx * ky - 1 comparisons per output (kx * xy > 1)
  // or 1 copy per output (kx * k1 = 1).
  int per_output_ops = dims.kx * dims.ky == 1 ? 1 : dims.kx * dims.ky - 1;
  int64_t ops = dims.batch * dims.ox * dims.oy * dims.oz * per_output_ops;
  node_costs->num_compute_ops = ops;

  int64_t input_size = 0;
  if (dims.ky >= dims.sy) {
    input_size = CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
  } else {  // dims.ky < dims.sy
    // Vertical stride is larger than vertical kernel; assuming row-major
    // format, skip unnecessary rows (or read every kx rows per sy rows, as the
    // others are not used for output).
    const auto data_size = DataTypeSize(BaseType(op_info.inputs(0).dtype()));
    input_size = data_size * dims.batch * dims.ix * dims.ky * dims.oy * dims.iz;
  }
  node_costs->num_input_bytes_accessed = {input_size};
  const int64_t output_size =
      CalculateOutputSize(op_info, &found_unknown_shapes);
  node_costs->num_output_bytes_accessed = {output_size};
  node_costs->max_memory = output_size;
  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

Status OpLevelCostEstimator::PredictMaxPoolGrad(const OpContext& op_context,
                                                NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;
  const auto& op_info = op_context.op_info;
  // x: op_info.inputs(0)
  // y: op_info.inputs(1)
  // y_grad: op_info.inputs(2)
  if (op_info.inputs_size() < 3) {
    return errors::InvalidArgument("MaxPoolGrad op has invalid inputs: ",
                                   op_info.ShortDebugString());
  }

  ConvolutionDimensions dims = OpDimensionsFromInputs(
      op_info.inputs(0).shape(), op_info, &found_unknown_shapes);

  int64_t ops = 0;
  if (dims.kx == 1 && dims.ky == 1) {
```

```
    // 1x1 window. No need to know which input was max.
    ops = dims.batch * dims.ix * dims.iy * dims.iz;
  } else if (dims.kx <= dims.sx && dims.ky <= dims.sy) {
    // Non-overlapping window: re-run maxpool, then assign zero or y_grad.
    ops = dims.batch * dims.iz *
          (dims.ox * dims.oy * (dims.kx * dims.ky - 1) + dims.ix * dims.iy);
  } else {
    // Overlapping window: initialize with zeros, re-run maxpool, then
    // accumulate y_gad to proper x_grad locations.
    ops = dims.batch * dims.iz *
          (dims.ox * dims.oy * (dims.kx * dims.ky - 1) + dims.ix * dims.iy * 2);
  }
  node_costs->num_compute_ops = ops;

  // Just read x and y_grad; no need to read y as we assume MaxPoolGrad re-run
  // MaxPool internally.
  const int64_t input0_size =
      CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
  const int64_t input2_size =
      CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
  node_costs->num_input_bytes_accessed = {input0_size, 0, input2_size};
  // Write x_grad; size equal to x.
  const int64_t output_size =
      CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
  node_costs->num_output_bytes_accessed = {output_size};
  node_costs->max_memory = output_size;

  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

/* This predict function handles three types of tensorflow ops
 * AssignVariableOp/AssignAddVariableOp/AssignSubVariableOp, broadcasting
 * was not possible for these ops, therefore the input tensor's shapes is
 * enough to compute the cost */
Status OpLevelCostEstimator::PredictAssignVariableOps(
    const OpContext& op_context, NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;
  const auto& op_info = op_context.op_info;
  /* First input of these ops are reference to the assignee. */
  if (op_info.inputs_size() != 2) {
    return errors::InvalidArgument("AssignVariable op has invalid input: ",
                                   op_info.ShortDebugString());
  }

  const int64_t ops = op_info.op() == kAssignVariableOp
```

```
                          ? 0
                          : CalculateTensorElementCount(op_info.inputs(1),
                                                        &found_unknown_shapes);
  node_costs->num_compute_ops = ops;
  const int64_t input_size = CalculateInputSize(op_info, &found_unknown_shapes);
  node_costs->num_input_bytes_accessed = {input_size};
  // TODO(dyoon): check these ops' behavior whether it writes data;
  // Op itself doesn't have output tensor, but it may modify the input (ref or
  // resource). Maybe use node_costs->internal_write_bytes.
  node_costs->num_output_bytes_accessed = {0};
  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

Status OpLevelCostEstimator::PredictAvgPool(const OpContext& op_context,
                                            NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;
  const auto& op_info = op_context.op_info;
  // x: op_info.inputs(0)
  ConvolutionDimensions dims = OpDimensionsFromInputs(
      op_info.inputs(0).shape(), op_info, &found_unknown_shapes);

  // kx * ky - 1 additions and 1 multiplication per output.
  int64_t ops = dims.batch * dims.ox * dims.oy * dims.oz * dims.kx * dims.ky;
  node_costs->num_compute_ops = ops;

  int64_t input_size;
  if (dims.ky >= dims.sy) {
    input_size = CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
  } else {  // dims.ky < dims.sy
    // vertical stride is larger than vertical kernel; assuming row-major
    // format, skip unnecessary rows (or read every kx rows per sy rows, as the
    // others are not used for output).
    const auto data_size = DataTypeSize(BaseType(op_info.inputs(0).dtype()));
    input_size = data_size * dims.batch * dims.ix * dims.ky * dims.oy * dims.iz;
  }
  node_costs->num_input_bytes_accessed = {input_size};

  const int64_t output_size =
      CalculateOutputSize(op_info, &found_unknown_shapes);
  node_costs->num_output_bytes_accessed = {output_size};
  node_costs->max_memory = output_size;

  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
```

```
2333    }
2334    return Status::OK();
2335  }
2336
2337  Status OpLevelCostEstimator::PredictAvgPoolGrad(const OpContext& op_context,
2338                                                  NodeCosts* node_costs) const {
2339    bool found_unknown_shapes = false;
2340    const auto& op_info = op_context.op_info;
2341    // x's shape: op_info.inputs(0)
2342    // y_grad: op_info.inputs(1)
2343
2344    // Extract x_shape from op_info.inputs(0).value() or op_info.outputs(0).
2345    bool shape_found = false;
2346    TensorShapeProto x_shape;
2347    if (op_info.inputs_size() >= 1 && op_info.inputs(0).has_value()) {
2348      const TensorProto& value = op_info.inputs(0).value();
2349      shape_found = GetTensorShapeProtoFromTensorProto(value, &x_shape);
2350    }
2351    if (!shape_found && op_info.outputs_size() > 0) {
2352      x_shape = op_info.outputs(0).shape();
2353      shape_found = true;
2354    }
2355    if (!shape_found) {
2356      // Set the minimum shape that's feasible.
2357      x_shape.Clear();
2358      for (int i = 0; i < 4; ++i) {
2359        x_shape.add_dim()->set_size(1);
2360      }
2361      found_unknown_shapes = true;
2362    }
2363
2364    ConvolutionDimensions dims =
2365        OpDimensionsFromInputs(x_shape, op_info, &found_unknown_shapes);
2366
2367    int64_t ops = 0;
2368    if (dims.kx <= dims.sx && dims.ky <= dims.sy) {
2369      // Non-overlapping window.
2370      ops = dims.batch * dims.iz * (dims.ix * dims.iy + dims.ox * dims.oy);
2371    } else {
2372      // Overlapping window.
2373      ops = dims.batch * dims.iz *
2374            (dims.ix * dims.iy + dims.ox * dims.oy * (dims.kx * dims.ky + 1));
2375    }
2376    auto s = PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
2377                                     node_costs);
2378    node_costs->max_memory = node_costs->num_total_output_bytes();
2379    return s;
2380  }
2381
```

```
Status OpLevelCostEstimator::PredictFusedBatchNorm(
    const OpContext& op_context, NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;
  const auto& op_info = op_context.op_info;
  // x: op_info.inputs(0)
  // scale: op_info.inputs(1)
  // offset: op_info.inputs(2)
  // mean: op_info.inputs(3)  --> only for inference
  // variance: op_info.inputs(4) --> only for inference
  ConvolutionDimensions dims = OpDimensionsFromInputs(
      op_info.inputs(0).shape(), op_info, &found_unknown_shapes);
  const bool is_training = IsTraining(op_info);

  int64_t ops = 0;
  const auto rsqrt_cost = Eigen::internal::functor_traits<
      Eigen::internal::scalar_rsqrt_op<float>>::Cost;
  if (is_training) {
    ops = dims.iz * (dims.batch * dims.ix * dims.iy * 4 + 6 + rsqrt_cost);
  } else {
    ops = dims.batch * dims.ix * dims.iy * dims.iz * 2;
  }
  node_costs->num_compute_ops = ops;

  const int64_t size_nhwc =
      CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
  const int64_t size_c =
      CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
  if (is_training) {
    node_costs->num_input_bytes_accessed = {size_nhwc, size_c, size_c};
    node_costs->num_output_bytes_accessed = {size_nhwc, size_c, size_c, size_c,
                                             size_c};
    // FusedBatchNorm in training mode internally re-reads the input tensor:
    // one for mean/variance, and the 2nd internal read forthe actual scaling.
    // Assume small intermediate data such as mean / variance (size_c) can be
    // cached on-chip.
    node_costs->internal_read_bytes = size_nhwc;
  } else {
    node_costs->num_input_bytes_accessed = {size_nhwc, size_c, size_c, size_c,
                                            size_c};
    node_costs->num_output_bytes_accessed = {size_nhwc};
  }
  node_costs->max_memory = node_costs->num_total_output_bytes();

  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}
```

```
2431
2432  Status OpLevelCostEstimator::PredictFusedBatchNormGrad(
2433      const OpContext& op_context, NodeCosts* node_costs) const {
2434    bool found_unknown_shapes = false;
2435    const auto& op_info = op_context.op_info;
2436    // y_backprop: op_info.inputs(0)
2437    // x: op_info.inputs(1)
2438    // scale: op_info.inputs(2)
2439    // mean: op_info.inputs(3)
2440    // variance or inverse of variance: op_info.inputs(4)
2441    ConvolutionDimensions dims = OpDimensionsFromInputs(
2442        op_info.inputs(1).shape(), op_info, &found_unknown_shapes);
2443
2444    int64_t ops = 0;
2445    const auto rsqrt_cost = Eigen::internal::functor_traits<
2446        Eigen::internal::scalar_rsqrt_op<float>>::Cost;
2447    ops = dims.iz * (dims.batch * dims.ix * dims.iy * 11 + 5 + rsqrt_cost);
2448    node_costs->num_compute_ops = ops;
2449
2450    const int64_t size_nhwc =
2451        CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
2452    const int64_t size_c =
2453        CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
2454    // TODO(dyoon): fix missing memory cost for variance input (size_c) and
2455    // yet another read of y_backprop (size_nhwc) internally.
2456    node_costs->num_input_bytes_accessed = {size_nhwc, size_nhwc, size_c, size_c};
2457    node_costs->num_output_bytes_accessed = {size_nhwc, size_c, size_c};
2458    // FusedBatchNormGrad has to read y_backprop internally.
2459    node_costs->internal_read_bytes = size_nhwc;
2460    node_costs->max_memory = node_costs->num_total_output_bytes();
2461
2462    if (found_unknown_shapes) {
2463      node_costs->inaccurate = true;
2464      node_costs->num_nodes_with_unknown_shapes = 1;
2465    }
2466    return Status::OK();
2467  }
2468
2469  Status OpLevelCostEstimator::PredictNaryOp(const OpContext& op_context,
2470                                             NodeCosts* node_costs) const {
2471    const auto& op_info = op_context.op_info;
2472    bool found_unknown_shapes = false;
2473    // Calculate the largest known tensor size across all inputs and output.
2474    int64_t op_count = CalculateLargestInputCount(op_info, &found_unknown_shapes);
2475    // If output shape is available, try to use the element count calculated from
2476    // that.
2477    if (op_info.outputs_size() > 0) {
2478      op_count = std::max(
2479          op_count,
```

```
2480          CalculateTensorElementCount(op_info.outputs(0), &found_unknown_shapes));
2481    }
2482    // Also calculate the output shape possibly resulting from broadcasting.
2483    // Note that the some Nary ops (such as AddN) do not support broadcasting,
2484    // but we're including this here for completeness.
2485    if (op_info.inputs_size() >= 2) {
2486      op_count = std::max(op_count, CwiseOutputElementCount(op_info));
2487    }
2488
2489    // Nary ops perform one operation for every element in every input tensor.
2490    op_count *= op_info.inputs_size() - 1;
2491
2492    const auto sum_cost = Eigen::internal::functor_traits<
2493        Eigen::internal::scalar_sum_op<float>>::Cost;
2494    return PredictDefaultNodeCosts(op_count * sum_cost, op_context,
2495                                   &found_unknown_shapes, node_costs);
2496  }
2497
2498  // softmax[i, j] = exp(logits[i, j]) / sum_j(exp(logits[i, j]))
2499  Status OpLevelCostEstimator::PredictSoftmax(const OpContext& op_context,
2500                                              NodeCosts* node_costs) const {
2501    bool found_unknown_shapes = false;
2502    const int64_t logits_size = CalculateTensorElementCount(
2503        op_context.op_info.inputs(0), &found_unknown_shapes);
2504    // Softmax input rank should be >=1.
2505    TensorShapeProto logits_shape = op_context.op_info.inputs(0).shape();
2506    if (logits_shape.unknown_rank() || logits_shape.dim_size() == 0) {
2507      return errors::InvalidArgument("Softmax op has invalid input: ",
2508                                     op_context.op_info.ShortDebugString());
2509    }
2510
2511  #define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost
2512
2513    // Every element of <logits> will be exponentiated, have that result included
2514    // in a sum across j, and also have that result multiplied by the reciprocal
2515    // of the sum_j. In addition, we'll compute 1/sum_j for every i.
2516    auto ops =
2517        (EIGEN_COST(scalar_exp_op<float>) + EIGEN_COST(scalar_sum_op<float>) +
2518         EIGEN_COST(scalar_product_op<float>)) *
2519            logits_size +
2520        EIGEN_COST(scalar_inverse_op<float>) * logits_shape.dim(0).size();
2521
2522  #undef EIGEN_COST
2523    return PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
2524                                   node_costs);
2525  }
2526
2527  Status OpLevelCostEstimator::PredictResizeBilinear(
2528      const OpContext& op_context, NodeCosts* node_costs) const {
```

```
2529    bool found_unknown_shapes = false;
2530
2531    if (op_context.op_info.outputs().empty() ||
2532        op_context.op_info.inputs().empty()) {
2533      return errors::InvalidArgument(
2534          "ResizeBilinear op has invalid input / output ",
2535          op_context.op_info.ShortDebugString());
2536    }
2537
2538    const int64_t output_elements = CalculateTensorElementCount(
2539        op_context.op_info.outputs(0), &found_unknown_shapes);
2540
2541    const auto half_pixel_centers =
2542        op_context.op_info.attr().find("half_pixel_centers");
2543    bool use_half_pixel_centers = false;
2544    if (half_pixel_centers == op_context.op_info.attr().end()) {
2545      LOG(WARNING) << "half_pixel_centers attr not set for ResizeBilinear.";
2546      return PredictCostOfAnUnknownOp(op_context, node_costs);
2547    } else {
2548      use_half_pixel_centers = half_pixel_centers->second.b();
2549    }
2550
2551    // Compose cost of bilinear interpolation.
2552    int64_t ops = 0;
2553
2554  #define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost
2555    const auto sub_cost_float = EIGEN_COST(scalar_difference_op<float>);
2556    const auto sub_cost_int = EIGEN_COST(scalar_difference_op<int64_t>);
2557    const auto add_cost = EIGEN_COST(scalar_sum_op<float>);
2558    const auto mul_cost = EIGEN_COST(scalar_product_op<float>);
2559    const auto floor_cost = EIGEN_COST(scalar_floor_op<float>);
2560    const auto max_cost = EIGEN_COST(scalar_max_op<int64_t>);
2561    const auto min_cost = EIGEN_COST(scalar_min_op<int64_t>);
2562    const auto cast_to_int_cost = Eigen::internal::functor_traits<
2563        Eigen::internal::scalar_cast_op<float, int64_t>>::Cost;
2564    const auto cast_to_float_cost = Eigen::internal::functor_traits<
2565        Eigen::internal::scalar_cast_op<int64_t, float>>::Cost;
2566    const auto ceil_cost = EIGEN_COST(scalar_ceil_op<float>);
2567  #undef EIGEN_COST
2568
2569    // Ops calculated from tensorflow/core/kernels/image/resize_bilinear_op.cc.
2570
2571    // Op counts taken from resize_bilinear implementation on 07/21/2020.
2572    // Computed op counts may become inaccurate if resize_bilinear implementation
2573    // changes.
2574
2575    // resize_bilinear has an optimization where the interpolation weights are
2576    // precomputed and cached. Given input tensors of size [B,H1,W1,C] and output
2577    // tensors of size [B,H2,W2,C], the last dimension C that needs to be accessed
```

```
2578    // in the input for interpolation are identical at every point in the output.
2579    // These values are cached in the compute_interpolation_weights function. For
2580    // a particular y in [0...H2-1], the rows to be accessed in the input are the
2581    // same. Likewise, for a particular x in [0...H2-1], the columns to be accsed
2582    // are the same. So the precomputation only needs to be done for H2 + W2
2583    // values.
2584    const auto output_shape = MaybeGetMinimumShape(
2585        op_context.op_info.outputs(0).shape(), 4, &found_unknown_shapes);
2586    // Assume H is dim 1 and W is dim 2 to match logic in resize_bilinear, which
2587    // also makes this assumption.
2588    const int64_t output_height = output_shape.dim(1).size();
2589    const int64_t output_width = output_shape.dim(2).size();
2590    // Add the ops done outside of the scaler function in
2591    // compute_interpolation_weights.
2592    int64_t interp_weight_cost = floor_cost + max_cost + min_cost +
2593                                 sub_cost_float + sub_cost_int + ceil_cost +
2594                                 cast_to_int_cost * 2;
2595    // There are two options for computing the weight of each pixel in the
2596    // interpolation. Algorithm can use pixel centers, or corners, for the
2597    // weight. Ops depend on the scaler function passed into
2598    // compute_interpolation_weights.
2599    if (use_half_pixel_centers) {
2600      // Ops for HalfPixelScalaer.
2601      interp_weight_cost +=
2602          add_cost + mul_cost + sub_cost_float + cast_to_float_cost;
2603    } else {
2604      // Ops for LegacyScaler.
2605      interp_weight_cost += cast_to_float_cost + mul_cost;
2606    }
2607    // Cost for the interpolation is multiplied by (H2 + w2), as mentioned above.
2608    ops += interp_weight_cost * (output_height + output_width);
2609
2610    // Ops for computing the new values, done for every element. Logic is from
2611    // compute_lerp in the inner loop of resize_image which consists of:
2612    //   const float top = top_left + (top_right - top_left) * x_lerp;
2613    //   const float bottom = bottom_left + (bottom_right - bottom_left) * x_lerp;
2614    //   return top + (bottom - top) * y_lerp;
2615    ops += (add_cost * 3 + sub_cost_float * 3 + mul_cost * 3) * output_elements;
2616
2617    return PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
2618                                   node_costs);
2619  }
2620
2621  Status OpLevelCostEstimator::PredictCropAndResize(const OpContext& op_context,
2622                                                    NodeCosts* node_costs) const {
2623    bool found_unknown_shapes = false;
2624
2625    const auto method = op_context.op_info.attr().find("method");
2626    bool use_bilinear_interp;
```

```
2627    if (method == op_context.op_info.attr().end() ||
2628        method->second.s() == "bilinear") {
2629      use_bilinear_interp = true;
2630    } else if (method->second.s() == "nearest") {
2631      use_bilinear_interp = false;
2632    } else {
2633      LOG(WARNING) << "method attr in CropAndResize invalid; expected bilinear "
2634                      "or nearest.";
2635      return PredictCostOfAnUnknownOp(op_context, node_costs);
2636    }
2637
2638    const int64_t num_boxes = op_context.op_info.inputs(1).shape().dim(0).size();
2639    const auto crop_shape = MaybeGetMinimumShape(
2640        op_context.op_info.outputs(0).shape(), 4, &found_unknown_shapes);
2641    const int64_t crop_height = crop_shape.dim(1).size();
2642    const int64_t crop_width = crop_shape.dim(2).size();
2643    const int64_t output_elements = CalculateTensorElementCount(
2644        op_context.op_info.outputs(0), &found_unknown_shapes);
2645
2646  #define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost
2647    const auto sub_cost = EIGEN_COST(scalar_difference_op<float>);
2648    const auto add_cost = EIGEN_COST(scalar_sum_op<float>);
2649    const auto mul_cost = EIGEN_COST(scalar_product_op<float>);
2650    auto div_cost = EIGEN_COST(scalar_div_cost<float>);
2651    const auto floor_cost = EIGEN_COST(scalar_floor_op<float>);
2652    const auto ceil_cost = EIGEN_COST(scalar_ceil_op<float>);
2653    auto round_cost = EIGEN_COST(scalar_round_op<float>);
2654    const auto cast_to_float_cost = Eigen::internal::functor_traits<
2655        Eigen::internal::scalar_cast_op<int64_t, float>>::Cost;
2656  #undef EIGEN_COST
2657
2658    // Computing ops following
2659    // tensorflow/core/kernels/image/crop_and_resize_op.cc at 08/25/2020. Op
2660    // calculation differs from rough estimate in implementation, as it separates
2661    // out cost per box from cost per pixel and cost per element.
2662
2663    // Ops for variables height_scale and width_scale.
2664    int64_t ops = (sub_cost * 6 + mul_cost * 2 + div_cost * 2) * num_boxes;
2665    // Ops for variable in_y.
2666    ops += (mul_cost * 2 + sub_cost + add_cost) * crop_height * num_boxes;
2667    // Ops for variable in_x (same computation across both branches).
2668    ops += (mul_cost * 2 + sub_cost + add_cost) * crop_height * crop_width *
2669           num_boxes;
2670    // Specify op_cost based on the method.
2671    if (use_bilinear_interp) {
2672      // Ops for variables top_y_index, bottom_y_index, y_lerp.
2673      ops += (floor_cost + ceil_cost + sub_cost) * crop_height * num_boxes;
2674      // Ops for variables left_x, right_x, x_lerp;
2675      ops += (floor_cost + ceil_cost + sub_cost) * crop_height * crop_width *
```

```
2676            num_boxes;
2677      // Ops for innermost loop across depth.
2678      ops +=
2679          (cast_to_float_cost * 4 + add_cost * 3 + sub_cost * 3 + mul_cost * 3) *
2680          output_elements;
2681    } else /* method == "nearest" */ {
2682      // Ops for variables closest_x_index and closest_y_index.
2683      ops += round_cost * 2 * crop_height * crop_width * num_boxes;
2684      // Ops for innermost loop across depth.
2685      ops += cast_to_float_cost * output_elements;
2686    }
2687    return PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
2688                                   node_costs);
2689  }
2690
2691  }  // end namespace grappler
2692  }  // end namespace tensorflow
```