2692 lines (2437 sloc) | 110 KB

```
/* Copyright 2017 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
==============================================================================*/

#include "tensorflow/core/grappler/costs/op_level_cost_estimator.h"

#include "absl/strings/match.h"
#include "third_party/eigen3/Eigen/Core"
#include "tensorflow/core/framework/attr_value.pb.h"
#include "tensorflow/core/framework/attr_value_util.h"
#include "tensorflow/core/framework/tensor.pb.h"
#include "tensorflow/core/framework/tensor_shape.pb.h"
#include "tensorflow/core/framework/types.h"
#include "tensorflow/core/grappler/clusters/utils.h"
#include "tensorflow/core/grappler/costs/op_context.h"
#include "tensorflow/core/grappler/costs/utils.h"
#include "tensorflow/core/platform/errors.h"
```

```

namespace tensorflow {
namespace grappler {

// TODO(dyoon): update op to Predict method map for TF ops with V2 or V3 suffix.
constexpr int kOpsPerMac = 2;
constexpr char kGuaranteeConst[] = "GuaranteeConst";
constexpr char kAddN[] = "AddN";
constexpr char kBitCast[] = "BitCast";
constexpr char kConcatV2[] = "ConcatV2";
constexpr char kConv2d[] = "Conv2D";
constexpr char kConv2dBackpropFilter[] = "Conv2DBackpropFilter";
constexpr char kConv2dBackpropInput[] = "Conv2DBackpropInput";
constexpr char kFusedConv2dBiasActivation[] = "FusedConv2DBiasActivation";
constexpr char kDataFormatVecPermute[] = "DataFormatVecPermute";
constexpr char kDepthToSpace[] = "DepthToSpace";
constexpr char kDepthwiseConv2dNative[] = "DepthwiseConv2dNative";
constexpr char kDepthwiseConv2dNativeBackpropFilter[] =
    "DepthwiseConv2dNativeBackpropFilter";
constexpr char kDepthwiseConv2dNativeBackpropInput[] =
    "DepthwiseConv2dNativeBackpropInput";
constexpr char kMatMul[] = "MatMul";
constexpr char kXlaEinsum[] = "XlaEinsum";
constexpr char kEinsum[] = "Einsum";
constexpr char kExpandDims[] = "ExpandDims";
constexpr char kFill[] = "Fill";
constexpr char kSparseMatMul[] = "SparseMatMul";
constexpr char kSparseTensorDenseMatMul[] = "SparseTensorDenseMatMul";
constexpr char kPlaceholder[] = "Placeholder";
constexpr char kIdentity[] = "Identity";
constexpr char kIdentityN[] = "IdentityN";
constexpr char kRefIdentity[] = "RefIdentity";
constexpr char kNoOp[] = "NoOp";
constexpr char kReshape[] = "Reshape";
constexpr char kSplit[] = "Split";
constexpr char kSqueeze[] = "Squeeze";
constexpr char kRecv[] = "_Recv";
constexpr char kSend[] = "_Send";
constexpr char kBatchMatMul[] = "BatchMatMul";
constexpr char kBatchMatMulV2[] = "BatchMatMulV2";
constexpr char kOneHot[] = "OneHot";
constexpr char kPack[] = "Pack";
constexpr char kRank[] = "Rank";
constexpr char kRange[] = "Range";
constexpr char kShape[] = "Shape";
constexpr char kShapeN[] = "ShapeN";
constexpr char kSize[] = "Size";
constexpr char kStopGradient[] = "StopGradient";
constexpr char kPreventGradient[] = "PreventGradient";
```

```
constexpr char kGather[] = "Gather";
constexpr char kGatherNd[] = "GatherNd";
constexpr char kGatherV2[] = "GatherV2";
constexpr char kScatterAdd[] = "ScatterAdd";
constexpr char kScatterDiv[] = "ScatterDiv";
constexpr char kScatterMax[] = "ScatterMax";
constexpr char kScatterMin[] = "ScatterMin";
constexpr char kScatterMul[] = "ScatterMul";
constexpr char kScatterSub[] = "ScatterSub";
constexpr char kScatterUpdate[] = "ScatterUpdate";
constexpr char kSlice[] = "Slice";
constexpr char kStridedSlice[] = "StridedSlice";
constexpr char kSpaceToDepth[] = "SpaceToDepth";
constexpr char kTranspose[] = "Transpose";
constexpr char kTile[] = "Tile";
constexpr char kMaxPool[] = "MaxPool";
constexpr char kMaxPoolGrad[] = "MaxPoolGrad";
constexpr char kAvgPool[] = "AvgPool";
constexpr char kAvgPoolGrad[] = "AvgPoolGrad";
constexpr char kFusedBatchNorm[] = "FusedBatchNorm";
constexpr char kFusedBatchNormGrad[] = "FusedBatchNormGrad";
constexpr char kQuantizedMatMul[] = "QuantizedMatMul";
constexpr char kQuantizedMatMulV2[] = "QuantizedMatMulV2";
constexpr char kUnpack[] = "Unpack";
constexpr char kSoftmax[] = "Softmax";
constexpr char kResizeBilinear[] = "ResizeBilinear";
constexpr char kCropAndResize[] = "CropAndResize";
// Dynamic control flow ops.
constexpr char kSwitch[] = "Switch";
constexpr char kMerge[] = "Merge";
constexpr char kEnter[] = "Enter";
constexpr char kExit[] = "Exit";
constexpr char kNextIteration[] = "NextIteration";
// Persistent ops.
constexpr char kConst[] = "Const";
constexpr char kVariable[] = "Variable";
constexpr char kVariableV2[] = "VariableV2";
constexpr char kAutoReloadVariable[] = "AutoReloadVariable";
constexpr char kVarHandleOp[] = "VarHandleOp";
constexpr char kVarHandlesOp[] = "_VarHandlesOp";
constexpr char kReadVariableOp[] = "ReadVariableOp";
constexpr char kReadVariablesOp[] = "_ReadVariablesOp";
constexpr char kAssignVariableOp[] = "AssignVariableOp";
constexpr char kAssignAddVariableOp[] = "AssignAddVariableOp";
constexpr char kAssignSubVariableOp[] = "AssignSubVariableOp";

static const Costs::Duration kMinComputeTime(1);
static const int64_t kMinComputeOp = 1;

```

```
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,
```

```
        wrap(&OpLevelCostEstimator::PredictConv2DBackpropFilter));
    device_cost_impl_.emplace(
        kConv2dBackpropInput,
        wrap(&OpLevelCostEstimator::PredictConv2DBackpropInput));
    device_cost_impl_.emplace(
        kFusedConv2dBiasActivation,
        wrap(&OpLevelCostEstimator::PredictFusedConv2DBiasActivation));
    // reuse Conv2D for DepthwiseConv2dNative because the calculation is the
    // same although the actual meaning of the parameters are different. See
    // comments in PredictConv2D and related functions
    device_cost_impl_.emplace(kDepthwiseConv2dNative,
                              wrap(&OpLevelCostEstimator::PredictConv2D));
    device_cost_impl_.emplace(
        kDepthwiseConv2dNativeBackpropFilter,
        wrap(&OpLevelCostEstimator::PredictConv2DBackpropFilter));
    device_cost_impl_.emplace(
        kDepthwiseConv2dNativeBackpropInput,
        wrap(&OpLevelCostEstimator::PredictConv2DBackpropInput));
    device_cost_impl_.emplace(kMatMul,
                              wrap(&OpLevelCostEstimator::PredictMatMul));
    device_cost_impl_.emplace(kSparseMatMul,
                              wrap(&OpLevelCostEstimator::PredictMatMul));
    device_cost_impl_.emplace(
        kSparseTensorDenseMatMul,
        wrap(&OpLevelCostEstimator::PredictSparseTensorDenseMatMul));
    device_cost_impl_.emplace(kBatchMatMul,
                              wrap(&OpLevelCostEstimator::PredictBatchMatMul));
    device_cost_impl_.emplace(kBatchMatMulV2,
                              wrap(&OpLevelCostEstimator::PredictBatchMatMul));
    device_cost_impl_.emplace(kQuantizedMatMul,
                              wrap(&OpLevelCostEstimator::PredictMatMul));
    device_cost_impl_.emplace(kQuantizedMatMulV2,
                              wrap(&OpLevelCostEstimator::PredictMatMul));
    device_cost_impl_.emplace(kXlaEinsum,
                              wrap(&OpLevelCostEstimator::PredictEinsum));
    device_cost_impl_.emplace(kEinsum,
                              wrap(&OpLevelCostEstimator::PredictEinsum));

    device_cost_impl_.emplace(kNoOp, wrap(&OpLevelCostEstimator::PredictNoOp));
    device_cost_impl_.emplace(kGuaranteeConst,
                              wrap(&OpLevelCostEstimator::PredictNoOp));

    device_cost_impl_.emplace(kGather,
                              wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
    device_cost_impl_.emplace(kGatherNd,
                              wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
    device_cost_impl_.emplace(kGatherV2,
                              wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
    device_cost_impl_.emplace(kScatterAdd,
```

```
                          wrap(&OpLevelCostEstimator::PredictScatter));
  device_cost_impl_.emplace(kScatterDiv,
                          wrap(&OpLevelCostEstimator::PredictScatter));
  device_cost_impl_.emplace(kScatterMax,
                          wrap(&OpLevelCostEstimator::PredictScatter));
  device_cost_impl_.emplace(kScatterMin,
                          wrap(&OpLevelCostEstimator::PredictScatter));
  device_cost_impl_.emplace(kScatterMul,
                          wrap(&OpLevelCostEstimator::PredictScatter));
  device_cost_impl_.emplace(kScatterSub,
                          wrap(&OpLevelCostEstimator::PredictScatter));
  device_cost_impl_.emplace(kScatterUpdate,
                          wrap(&OpLevelCostEstimator::PredictScatter));

  device_cost_impl_.emplace(kSlice,
                          wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
  device_cost_impl_.emplace(kStridedSlice,
                          wrap(&OpLevelCostEstimator::PredictGatherOrSlice));

  device_cost_impl_.emplace(kPlaceholder,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kIdentity,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kIdentityN,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kRefIdentity,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kStopGradient,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kPreventGradient,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kReshape,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kRecv,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kSend,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kSwitch,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kMerge,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kEnter,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kExit,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kNextIteration,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
  device_cost_impl_.emplace(kBitCast,
                          wrap(&OpLevelCostEstimator::PredictIdentity));
```

```
  device_cost_impl_.emplace(kConcatV2,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kDataFormatVecPermute,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kDepthToSpace,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kExpandDims,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kFill,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kOneHot,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kPack,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kRange,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kSpaceToDepth,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kSplit,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kSqueeze,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kTranspose,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kTile,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));
  device_cost_impl_.emplace(kUnpack,
                            wrap(&OpLevelCostEstimator::PredictPureMemoryOp));

  device_cost_impl_.emplace(kRank,
                            wrap(&OpLevelCostEstimator::PredictMetadata));
  device_cost_impl_.emplace(kShape,
                            wrap(&OpLevelCostEstimator::PredictMetadata));
  device_cost_impl_.emplace(kShapeN,
                            wrap(&OpLevelCostEstimator::PredictMetadata));
  device_cost_impl_.emplace(kSize,
                            wrap(&OpLevelCostEstimator::PredictMetadata));
  device_cost_impl_.emplace(kMaxPool,
                            wrap(&OpLevelCostEstimator::PredictMaxPool));
  device_cost_impl_.emplace(kMaxPoolGrad,
                            wrap(&OpLevelCostEstimator::PredictMaxPoolGrad));
  device_cost_impl_.emplace(kAvgPool,
                            wrap(&OpLevelCostEstimator::PredictAvgPool));
  device_cost_impl_.emplace(kAvgPoolGrad,
                            wrap(&OpLevelCostEstimator::PredictAvgPoolGrad));
  device_cost_impl_.emplace(kFusedBatchNorm,
                            wrap(&OpLevelCostEstimator::PredictFusedBatchNorm));
  device_cost_impl_.emplace(
```

```
      kFusedBatchNormGrad,
      wrap(&OpLevelCostEstimator::PredictFusedBatchNormGrad));
  device_cost_impl_.emplace(kSoftmax,
                            wrap(&OpLevelCostEstimator::PredictSoftmax));
  device_cost_impl_.emplace(kResizeBilinear,
                            wrap(&OpLevelCostEstimator::PredictResizeBilinear));
  device_cost_impl_.emplace(kCropAndResize,
                            wrap(&OpLevelCostEstimator::PredictCropAndResize));
  device_cost_impl_.emplace(
      kAssignVariableOp, wrap(&OpLevelCostEstimator::PredictAssignVariableOps));
  device_cost_impl_.emplace(
      kAssignAddVariableOp,
      wrap(&OpLevelCostEstimator::PredictAssignVariableOps));
  device_cost_impl_.emplace(
      kAssignSubVariableOp,
      wrap(&OpLevelCostEstimator::PredictAssignVariableOps));
  device_cost_impl_.emplace(kAddN, wrap(&OpLevelCostEstimator::PredictNaryOp));

  persistent_ops_ = {
      kConst,       kVariable,       kVariableV2,   kAutoReloadVariable,
      kVarHandleOp, kReadVariableOp, kVarHandlesOp, kReadVariablesOp};

#define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost

  // Quantize = apply min and max bounds, multiply by scale factor and round.
  const int quantize_v2_cost =
      EIGEN_COST(scalar_product_op<float>) + EIGEN_COST(scalar_max_op<float>) +
      EIGEN_COST(scalar_min_op<float>) + EIGEN_COST(scalar_round_op<float>);
  const int quantize_and_dequantize_v2_cost =
      quantize_v2_cost + EIGEN_COST(scalar_product_op<float>);

  // Unary ops alphabetically sorted
  elementwise_ops_.emplace("Acos", EIGEN_COST(scalar_acos_op<float>));
  elementwise_ops_.emplace("All", EIGEN_COST(scalar_boolean_and_op));
  elementwise_ops_.emplace("ArgMax", EIGEN_COST(scalar_max_op<float>));
  elementwise_ops_.emplace("Asin", EIGEN_COST(scalar_asin_op<float>));
  elementwise_ops_.emplace("Atan", EIGEN_COST(scalar_atan_op<float>));
  elementwise_ops_.emplace("Atan2", EIGEN_COST(scalar_quotient_op<float>) +
                                        EIGEN_COST(scalar_atan_op<float>));
  // For now, we use Eigen cost model for float to int16 cast as an example
  // case; Eigen cost model is zero when src and dst types are identical,
  // and it uses AddCost (1) when different. We may implement a separate
  // cost functions for cast ops, using the actual input and output types.
  elementwise_ops_.emplace(
      "Cast", Eigen::internal::functor_traits<
                  Eigen::internal::scalar_cast_op<float, int16>>::Cost);
  elementwise_ops_.emplace("Ceil", EIGEN_COST(scalar_ceil_op<float>));
  elementwise_ops_.emplace("Cos", EIGEN_COST(scalar_cos_op<float>));
  elementwise_ops_.emplace("Dequantize", EIGEN_COST(scalar_product_op<float>));
```

```
  elementwise_ops_.emplace("Erf", 1);
  elementwise_ops_.emplace("Erfc", 1);
  elementwise_ops_.emplace("Exp", EIGEN_COST(scalar_exp_op<float>));
  elementwise_ops_.emplace("Expm1", EIGEN_COST(scalar_expm1_op<float>));
  elementwise_ops_.emplace("Floor", EIGEN_COST(scalar_floor_op<float>));
  elementwise_ops_.emplace("Inv", EIGEN_COST(scalar_inverse_op<float>));
  elementwise_ops_.emplace("InvGrad", 1);
  elementwise_ops_.emplace("Lgamma", 1);
  elementwise_ops_.emplace("Log", EIGEN_COST(scalar_log_op<float>));
  elementwise_ops_.emplace("Log1p", EIGEN_COST(scalar_log1p_op<float>));
  elementwise_ops_.emplace("Max", EIGEN_COST(scalar_max_op<float>));
  elementwise_ops_.emplace("Min", EIGEN_COST(scalar_min_op<float>));
  elementwise_ops_.emplace("Neg", EIGEN_COST(scalar_opposite_op<float>));
  elementwise_ops_.emplace("Prod", EIGEN_COST(scalar_product_op<float>));
  elementwise_ops_.emplace("QuantizeAndDequantizeV2",
                           quantize_and_dequantize_v2_cost);
  elementwise_ops_.emplace("QuantizeAndDequantizeV4",
                           quantize_and_dequantize_v2_cost);
  elementwise_ops_.emplace("QuantizedSigmoid",
                           EIGEN_COST(scalar_logistic_op<float>));
  elementwise_ops_.emplace("QuantizeV2", quantize_v2_cost);
  elementwise_ops_.emplace("Reciprocal", EIGEN_COST(scalar_inverse_op<float>));
  elementwise_ops_.emplace("Relu", EIGEN_COST(scalar_max_op<float>));
  elementwise_ops_.emplace("Relu6", EIGEN_COST(scalar_max_op<float>));
  elementwise_ops_.emplace("Rint", 1);
  elementwise_ops_.emplace("Round", EIGEN_COST(scalar_round_op<float>));
  elementwise_ops_.emplace("Rsqrt", EIGEN_COST(scalar_rsqrt_op<float>));
  elementwise_ops_.emplace("Sigmoid", EIGEN_COST(scalar_logistic_op<float>));
  elementwise_ops_.emplace("Sign", EIGEN_COST(scalar_sign_op<float>));
  elementwise_ops_.emplace("Sin", EIGEN_COST(scalar_sin_op<float>));
  elementwise_ops_.emplace("Sqrt", EIGEN_COST(scalar_sqrt_op<float>));
  elementwise_ops_.emplace("Square", EIGEN_COST(scalar_square_op<float>));
  elementwise_ops_.emplace("Sum", EIGEN_COST(scalar_sum_op<float>));
  elementwise_ops_.emplace("Tan", EIGEN_COST(scalar_tan_op<float>));
  elementwise_ops_.emplace("Tanh", EIGEN_COST(scalar_tanh_op<float>));
  elementwise_ops_.emplace("TopKV2", EIGEN_COST(scalar_max_op<float>));
  // Binary ops alphabetically sorted
  elementwise_ops_.emplace("Add", EIGEN_COST(scalar_sum_op<float>));
  elementwise_ops_.emplace("AddV2", EIGEN_COST(scalar_sum_op<float>));
  elementwise_ops_.emplace("ApproximateEqual", 1);
  elementwise_ops_.emplace("BiasAdd", EIGEN_COST(scalar_sum_op<float>));
  elementwise_ops_.emplace("QuantizedBiasAdd",
                           EIGEN_COST(scalar_sum_op<float>));
  elementwise_ops_.emplace("Div", EIGEN_COST(scalar_quotient_op<float>));
  elementwise_ops_.emplace("Equal", 1);
  elementwise_ops_.emplace("FloorDiv", EIGEN_COST(scalar_quotient_op<float>));
  elementwise_ops_.emplace("FloorMod", EIGEN_COST(scalar_mod_op<float>));
  elementwise_ops_.emplace("Greater", 1);
  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;
```

```
      costs.intermediate_memory_time = 0;
      costs.intermediate_memory_read_time = 0;
      costs.intermediate_memory_write_time = 0;
    } else {
      // Convert NodeCosts to Costs.
      costs = PredictOpCountBasedCost(
          node_costs.num_compute_ops, node_costs.num_total_read_bytes(),
          node_costs.num_total_write_bytes(), op_context.op_info);
    }
    VLOG(1) << "Operation " << op_context.op_info.op() << " takes "
            << costs.execution_time.count() << " ns.";
    // Copy additional stats from NodeCosts to Costs.
    costs.max_memory = node_costs.max_memory;
    costs.persistent_memory = node_costs.persistent_memory;
    costs.temporary_memory = node_costs.temporary_memory;
    costs.inaccurate = node_costs.inaccurate;
    costs.num_ops_with_unknown_shapes =
        node_costs.num_nodes_with_unknown_shapes;
    costs.num_ops_total = node_costs.num_nodes;
    return costs;
  }
  // Errors during node cost estimate.
  LOG(WARNING) << "Error in PredictCost() for the op: "
               << op_context.op_info.ShortDebugString();
  costs = Costs::ZeroCosts(/*inaccurate=*/true);
  costs.num_ops_with_unknown_shapes = node_costs.num_nodes_with_unknown_shapes;
  return costs;
}

Status OpLevelCostEstimator::PredictNodeCosts(const OpContext& op_context,
                                              NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  auto it = device_cost_impl_.find(op_info.op());
  if (it != device_cost_impl_.end()) {
    std::function<Status(const OpContext&, NodeCosts*)> estimator = it->second;
    return estimator(op_context, node_costs);
  }

  if (persistent_ops_.find(op_info.op()) != persistent_ops_.end()) {
    return PredictVariable(op_context, node_costs);
  }

  if (elementwise_ops_.find(op_info.op()) != elementwise_ops_.end()) {
    return PredictCwiseOp(op_context, node_costs);
  }

  VLOG(1) << "Missing accurate estimator for op: " << op_info.op();

  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);
}
```

```

Status OpLevelCostEstimator::PredictCostOfAnUnknownOp(
    const OpContext& op_context, NodeCosts* node_costs) const {
  // Don't assume the operation is cwise, return cost based on input/output size
  // and admit that it is inaccurate...
  bool found_unknown_shapes = false;
  node_costs->inaccurate = true;
  return PredictDefaultNodeCosts(0, op_context, &found_unknown_shapes,
                                 node_costs);
}

Costs OpLevelCostEstimator::PredictOpCountBasedCost(
    double operations, const OpInfo& op_info) const {
  bool unknown_shapes = false;
  const double input_size = CalculateInputSize(op_info, &unknown_shapes);
  const double output_size = CalculateOutputSize(op_info, &unknown_shapes);
  Costs costs =
      PredictOpCountBasedCost(operations, input_size, output_size, op_info);
  costs.inaccurate = unknown_shapes;
  costs.num_ops_with_unknown_shapes = unknown_shapes;
  costs.max_memory = output_size;
  return costs;
}

Costs OpLevelCostEstimator::PredictOpCountBasedCost(
    double operations, double input_io_bytes, double output_io_bytes,
    const OpInfo& op_info) const {
  double total_io_bytes = input_io_bytes + output_io_bytes;
  const DeviceInfo device_info = GetDeviceInfo(op_info.device());
  if (device_info.gigaops <= 0 || device_info.gb_per_sec <= 0 ||
      device_info.intermediate_read_gb_per_sec <= 0 ||
      device_info.intermediate_write_gb_per_sec <= 0) {
    VLOG(1) << "BAD DEVICE. Op:" << op_info.op()
            << " device type:" << op_info.device().type()
            << " device model:" << op_info.device().model();
  }

  Costs::NanoSeconds compute_cost(std::ceil(operations / device_info.gigaops));
  VLOG(1) << "Op:" << op_info.op() << " GOps:" << operations / 1e9
          << " Compute Time (ns):" << compute_cost.count();

  Costs::NanoSeconds memory_cost(
      std::ceil(total_io_bytes / device_info.gb_per_sec));
  VLOG(1) << "Op:" << op_info.op() << " Size (KB):" << (total_io_bytes) / 1e3
          << " Memory Time (ns):" << memory_cost.count();

  // Check if bytes > 0.  If it's not and the bandwidth is set to infinity
  // then the result would be undefined.
  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") {
```

```
    major_channel_index = 1;
    y_index = 2;
    x_index = 3;
  } else if (data_format == "NCHW_VECT_C") {
    // Use NCHW_VECT_C
    minor_channel_index = 1;
    y_index = 2;
    x_index = 3;
    major_channel_index = 4;
  } else {
    // Use NHWC.
    y_index = 1;
    x_index = 2;
    major_channel_index = 3;
  }
  const std::string& filter_format = GetFilterFormat(op_info);
  int filter_x_index, filter_y_index, in_major_channel_index, out_channel_index,
      in_minor_channel_index = -1;
  if (filter_format == "HWIO") {
    filter_y_index = 0;
    filter_x_index = 1;
    in_major_channel_index = 2;
    out_channel_index = 3;
  } else if (filter_format == "OIHW_VECT_I") {
    out_channel_index = 0;
    in_minor_channel_index = 1;
    filter_y_index = 2;
    filter_x_index = 3;
    in_major_channel_index = 4;
  } else {
    // Use OIHW
    out_channel_index = 0;
    in_major_channel_index = 1;
    filter_y_index = 2;
    filter_x_index = 3;
  }

  auto image_shape = MaybeGetMinimumShape(original_image_shape,
                                          minor_channel_index >= 0 ? 5 : 4,
                                          found_unknown_shapes);
  auto filter_shape = MaybeGetMinimumShape(original_filter_shape,
                                           in_minor_channel_index >= 0 ? 5 : 4,
                                           found_unknown_shapes);
  VLOG(2) << "Image shape: " << image_shape.DebugString();
  VLOG(2) << "Filter shape: " << filter_shape.DebugString();

  int64_t batch = image_shape.dim(0).size();
  int64_t ix = image_shape.dim(x_index).size();
  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";
```

```

  if (op_info.inputs_size() < 2) {  // Unexpected inputs.
    *found_unknown_shapes = true;
    return 0;
  }

  ConvolutionDimensions conv_dims = ConvolutionDimensionsFromInputs(
      op_info.inputs(0).shape(), op_info.inputs(1).shape(), op_info,
      found_unknown_shapes);

  //  in DepthwiseConv2dNative conv_dims.oz is actually the channel depth
  //  multiplier; The effective output channel depth oz_effective is
  //  conv_dims.iz * conv_dims.oz. thus # ops = N x H x W x oz_effective x 2RS.
  //  Compare to Conv2D where # ops =  N x H x W x kz x oz x 2RS,
  //  oz = oz_effective,  then Conv2D_ops / Depthwise_conv2d_native_ops = kz.
  int64_t ops = conv_dims.batch;
  ops *= conv_dims.ox * conv_dims.oy;
  ops *= conv_dims.kx * conv_dims.ky;
  if (op_info.op() == kConv2d) {
    ops *= conv_dims.kz * conv_dims.oz;
  } else {
    // To ensure output tensor dims to be correct for DepthwiseConv2DNative,
    // although ops are the same as Conv2D.
    conv_dims.oz *= conv_dims.iz;
    ops *= conv_dims.oz;
  }
  ops *= kOpsPerMac;

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

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

// TODO(nishantpatil): Create separate estimator for Sparse Matmul
int64_t OpLevelCostEstimator::CountMatMulOperations(
    const OpInfo& op_info, MatMulDimensions* mat_mul,
    bool* found_unknown_shapes) {
  double ops = 0;

  if (op_info.inputs_size() < 2) {
    LOG(ERROR) << "Need 2 inputs but got " << op_info.inputs_size();
    // TODO(pcma): Try to separate invalid inputs from unknown shapes
    *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);
```

```
  }

  ops = m_dim * n_dim * k_dim * 2;
  VLOG(1) << "Operations for Matmul: " << ops;

  if (mat_mul != nullptr) {
    mat_mul->m = m_dim;
    mat_mul->n = n_dim;
    mat_mul->k = k_dim;
  }
  return ops;
}

bool OpLevelCostEstimator::GenerateBatchMatmulContextFromEinsum(
    const OpContext& einsum_context, OpContext* batch_matmul_context,
    bool* found_unknown_shapes) const {
  // This auxiliary function transforms an einsum OpContext into its equivalent
  // Batch Matmul OpContext. The function returns a boolean, which determines
  // whether it was successful in generating the output OpContext or not.

  // Einsum computes a generalized contraction between tensors of arbitrary
  // dimension as defined by the equation written in the Einstein summation
  // convention. The number of tensors in the computation and the number of
  // contractions can be arbitrarily long. The current model only contemplates
  // Einsum equations, which can be translated into a single BatchMatMul
  // operation. Einsum operations with more than two operands are not currently
  // supported. Subscripts where an axis appears more than once for a single
  // input and ellipsis are currently also excluded. See:
  // https://www.tensorflow.org/api_docs/python/tf/einsum
  // We distinguish four kinds of dimensions, depending on their placement in
  // the equation:
  // + B: Batch dimensions: Dimensions which appear in both operands and RHS.
  // + K: Contracting dimensions: These appear in both inputs but not RHS.
  // + M: Operand A dimensions: These appear in the first operand and the RHS.
  // + N: Operand B dimensions: These appear in the second operand and the RHS.
  // Then, the operation to estimate is BatchMatMul([B,M,K],[B,K,N])

  if (batch_matmul_context == nullptr) {
    VLOG(1) << "Output context should not be a nullptr.";
    return false;
  }
  if (!IsEinsumCorrectlyFormed(einsum_context)) return false;
  const auto& op_info = einsum_context.op_info;
  std::vector<std::string> equation_split =
      absl::StrSplit(op_info.attr().find("equation")->second.s(), "->");
  std::vector<absl::string_view> input_split =
      absl::StrSplit(equation_split[0], ',');
  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];

  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);

  *found_unknown_shapes = a_input_shape_unknown || b_input_shape_unknown ||
                          (a_input.shape().dim_size() < kMatrixRank) ||
                          (b_input.shape().dim_size() < kMatrixRank);

  OpInfo batch_matmul_op_info = op_info;
  batch_matmul_op_info.mutable_inputs()->Clear();
  batch_matmul_op_info.set_op("BatchMatMul");

  AttrValue transpose_attribute;
  transpose_attribute.set_b(false);
  (*batch_matmul_op_info.mutable_attr())["transpose_a"] = transpose_attribute;
  (*batch_matmul_op_info.mutable_attr())["transpose_b"] = transpose_attribute;

  OpInfo::TensorProperties* a_matrix = batch_matmul_op_info.add_inputs();
  TensorShapeProto* a_matrix_shape = a_matrix->mutable_shape();
  a_matrix->set_dtype(a_input.dtype());

  OpInfo::TensorProperties* b_matrix = batch_matmul_op_info.add_inputs();
  b_matrix->set_dtype(b_input.dtype());
  TensorShapeProto* b_matrix_shape = b_matrix->mutable_shape();

  TensorShapeProto_Dim m_dim;
  TensorShapeProto_Dim n_dim;
  TensorShapeProto_Dim k_dim;

  m_dim.set_size(1);
  n_dim.set_size(1);
  k_dim.set_size(1);

  for (int i_idx = 0, a_input_str_size = a_input_str.size();
       i_idx < a_input_str_size; ++i_idx) {
    if (b_input_str.find(a_input_str[i_idx]) == std::string::npos) {
      if (rhs_str.find(a_input_str[i_idx]) == std::string::npos) {
```

```
        VLOG(1) << "Missing accurate estimator for op: " << op_info.op();
        return false;
      }

      m_dim.set_size(m_dim.size() * a_input_shape.dim(i_idx).size());
      continue;
    } else if (rhs_str.find(a_input_str[i_idx]) == std::string::npos) {
      // The dimension does not appear in the RHS, therefore it is a contracting
      // dimension.
      k_dim.set_size(k_dim.size() * a_input_shape.dim(i_idx).size());
      continue;
    }
    // It appears in both input operands, therefore we place it as an outer
    // dimension for the Batch Matmul.
    *(a_matrix_shape->add_dim()) = a_input_shape.dim(i_idx);
    *(b_matrix_shape->add_dim()) = a_input_shape.dim(i_idx);
  }
  for (int i_idx = 0, b_input_str_size = b_input_str.size();
       i_idx < b_input_str_size; ++i_idx) {
    if (a_input_str.find(b_input_str[i_idx]) == std::string::npos) {
      if (rhs_str.find(b_input_str[i_idx]) == std::string::npos) {
        VLOG(1) << "Missing accurate estimator for op: " << op_info.op();
        return false;
      }
      n_dim.set_size(n_dim.size() * b_input_shape.dim(i_idx).size());
    }
  }

  // The two inner-most dimensions of the Batch Matmul are added.
  *(a_matrix_shape->add_dim()) = m_dim;
  *(a_matrix_shape->add_dim()) = k_dim;
  *(b_matrix_shape->add_dim()) = k_dim;
  *(b_matrix_shape->add_dim()) = n_dim;

  *batch_matmul_context = einsum_context;
  batch_matmul_context->op_info = batch_matmul_op_info;
  return true;
}

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

int64_t OpLevelCostEstimator::CountBatchMatMulOperations(
    const OpInfo& op_info, BatchMatMulDimensions* batch_mat_mul,
    bool* found_unknown_shapes) {
  if (op_info.op() != kBatchMatMul && op_info.op() != kBatchMatMulV2) {
    LOG(ERROR) << "Invalid Operation: " << op_info.op();
```

```
    // TODO(pcma): Try to separate invalid inputs from unknown shapes
    *found_unknown_shapes = true;
    return 0;
  }
  if (op_info.inputs_size() != 2) {
    LOG(ERROR) << "Expected 2 inputs but got " << op_info.inputs_size();
    // TODO(pcma): Try to separate invalid inputs from unknown shapes
    *found_unknown_shapes = true;
    return 0;
  }

  double ops = 0;
  const auto& a_input = op_info.inputs(0);
  const auto& b_input = op_info.inputs(1);

  // BatchMatMul requires inputs of at least matrix shape (rank 2).
  // The two most minor dimensions of each input are matrices that
  // need to be multiplied together. The other dimensions determine
  // the number of such MatMuls.  For example, if the BatchMatMul has
  // inputs of shape:
  //   a_input_shape = [2, 3, 4, 5]
  //   b_input_shape = [2, 3, 5, 6]
  // then there are 2*3 = 6 MatMuls of dimensions m = 4, k = 5, n = 6
  // in this BatchMatMul.
  const int matrix_rank = 2;

  bool a_input_shape_unknown = false;
  bool b_input_shape_unknown = false;

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

  *found_unknown_shapes = a_input_shape_unknown || b_input_shape_unknown ||
                          (a_input.shape().dim_size() < matrix_rank) ||
                          (b_input.shape().dim_size() < matrix_rank);

  // Compute the number of matmuls as the max indicated at each dimension
  // by either input. Note that the shapes do not have to have
  // the same rank due to incompleteness.
  TensorShapeProto* bigger_rank_shape = &a_input_shape;
  TensorShapeProto* smaller_rank_shape = &b_input_shape;
  if (b_input_shape.dim_size() > a_input_shape.dim_size()) {
    bigger_rank_shape = &b_input_shape;
    smaller_rank_shape = &a_input_shape;
  }
```

```
  int num_matmuls = 1;
  for (int b_i = 0,
           s_i = smaller_rank_shape->dim_size() - bigger_rank_shape->dim_size();
       b_i < bigger_rank_shape->dim_size() - matrix_rank; ++b_i, ++s_i) {
    int b_dim = bigger_rank_shape->dim(b_i).size();
    int s_dim = 1;
    if (s_i >= 0) {
      s_dim = smaller_rank_shape->dim(s_i).size();
    }
    if (batch_mat_mul != nullptr) {
      batch_mat_mul->batch_dims.push_back(s_dim);
    }
    num_matmuls *= std::max(b_dim, s_dim);
  }

  // Build the MatMul. Note that values are ignored here since we are just
  // counting ops (e.g. only shapes matter).
  OpInfo matmul_op_info;
  matmul_op_info.set_op("MatMul");

  AttrValue transpose_a;
  transpose_a.set_b(false);
  if (op_info.attr().find("adj_x") != op_info.attr().end()) {
    transpose_a.set_b(op_info.attr().at("adj_x").b());
  }
  (*matmul_op_info.mutable_attr())["transpose_a"] = transpose_a;

  AttrValue transpose_b;
  transpose_b.set_b(false);
  if (op_info.attr().find("adj_y") != op_info.attr().end()) {
    transpose_b.set_b(op_info.attr().at("adj_y").b());
  }
  (*matmul_op_info.mutable_attr())["transpose_b"] = transpose_b;

  OpInfo::TensorProperties* a_matrix = matmul_op_info.add_inputs();
  a_matrix->set_dtype(a_input.dtype());
  TensorShapeProto* a_matrix_shape = a_matrix->mutable_shape();
  for (int i = std::max(0, a_input_shape.dim_size() - matrix_rank);
       i < a_input_shape.dim_size(); ++i) {
    *(a_matrix_shape->add_dim()) = a_input_shape.dim(i);
  }

  OpInfo::TensorProperties* b_matrix = matmul_op_info.add_inputs();
  b_matrix->set_dtype(b_input.dtype());
  TensorShapeProto* b_matrix_shape = b_matrix->mutable_shape();
  for (int i = std::max(0, b_input_shape.dim_size() - matrix_rank);
       i < b_input_shape.dim_size(); ++i) {
    *(b_matrix_shape->add_dim()) = b_input_shape.dim(i);
  }
```

```
  if (batch_mat_mul != nullptr) {
    batch_mat_mul->matmul_dims.m = (transpose_a.b())
                                       ? a_matrix_shape->dim(1).size()
                                       : a_matrix_shape->dim(0).size();
    batch_mat_mul->matmul_dims.k = (transpose_a.b())
                                       ? a_matrix_shape->dim(0).size()
                                       : a_matrix_shape->dim(1).size();
    batch_mat_mul->matmul_dims.n = (transpose_b.b())
                                       ? b_matrix_shape->dim(0).size()
                                       : b_matrix_shape->dim(1).size();
  }

  for (int i = 0; i < num_matmuls; ++i) {
    bool matmul_unknown_shapes = false;
    ops += CountMatMulOperations(matmul_op_info, &matmul_unknown_shapes);
    *found_unknown_shapes |= matmul_unknown_shapes;
  }
  return ops;
}

bool GetTensorShapeProtoFromTensorProto(const TensorProto& tensor_proto,
                                        TensorShapeProto* tensor_shape_proto) {
  tensor_shape_proto->Clear();
  // First convert TensorProto into Tensor class so that it correctly parses
  // data values within TensorProto (whether it's in int_val, int64_val,
  // tensor_content, or anything.
  Tensor tensor(tensor_proto.dtype());
  if (!tensor.FromProto(tensor_proto)) {
    LOG(WARNING) << "GetTensorShapeProtoFromTensorProto() -- "
                 << "failed to parse TensorProto: "
                 << tensor_proto.DebugString();
    return false;
  }
  if (tensor.dims() != 1) {
    LOG(WARNING) << "GetTensorShapeProtoFromTensorProto() -- "
                 << "tensor is not 1D: " << tensor.dims();
    return false;
  }
  // Then, convert it back to TensorProto using AsProtoField, which makes sure
  // the data is in int_val, int64_val, or such repeated data fields, not in
  // tensor_content.
  TensorProto temp_tensor;
  tensor.AsProtoField(&temp_tensor);

#define TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(type)        \
  do {                                                   \
    for (const auto& value : temp_tensor.type##_val()) { \
      tensor_shape_proto->add_dim()->set_size(value);    \
    }                                                    \
```

```
  } 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();
  }
```

```
  return tensor_size;
}

int64_t OpLevelCostEstimator::CalculateTensorSize(
    const OpInfo::TensorProperties& tensor, bool* found_unknown_shapes) {
  int64_t count = CalculateTensorElementCount(tensor, found_unknown_shapes);
  int size = DataTypeSize(BaseType(tensor.dtype()));
  VLOG(2) << "Count: " << count << " DataTypeSize: " << size;
  return count * size;
}

int64_t OpLevelCostEstimator::CalculateInputSize(const OpInfo& op_info,
                                                 bool* found_unknown_shapes) {
  int64_t total_input_size = 0;
  for (auto& input : op_info.inputs()) {
    int64_t input_size = CalculateTensorSize(input, found_unknown_shapes);
    total_input_size += input_size;
    VLOG(1) << "Input Size: " << input_size
            << " Total Input Size:" << total_input_size;
  }
  return total_input_size;
}

std::vector<int64_t> OpLevelCostEstimator::CalculateInputTensorSize(
    const OpInfo& op_info, bool* found_unknown_shapes) {
  std::vector<int64_t> input_tensor_size;
  input_tensor_size.reserve(op_info.inputs().size());
  for (auto& input : op_info.inputs()) {
    input_tensor_size.push_back(
        CalculateTensorSize(input, found_unknown_shapes));
  }
  return input_tensor_size;
}

int64_t OpLevelCostEstimator::CalculateLargestInputCount(
    const OpInfo& op_info, bool* found_unknown_shapes) {
  int64_t largest_input_count = 0;
  for (auto& input : op_info.inputs()) {
    int64_t input_count =
        CalculateTensorElementCount(input, found_unknown_shapes);
    if (input_count > largest_input_count) {
      largest_input_count = input_count;
    }
    VLOG(1) << "Input Count: " << input_count
            << " Largest Input Count:" << largest_input_count;
  }
  return largest_input_count;
}

```

```cpp
int64_t OpLevelCostEstimator::CalculateOutputSize(const OpInfo& op_info,
                                                  bool* found_unknown_shapes) {
  int64_t total_output_size = 0;
  // Use float as default for calculations.
  for (const auto& output : op_info.outputs()) {
    DataType dt = output.dtype();
    const auto& original_output_shape = output.shape();
    int64_t output_size = DataTypeSize(BaseType(dt));
    int num_dims = std::max(1, original_output_shape.dim_size());
    auto output_shape = MaybeGetMinimumShape(original_output_shape, num_dims,
                                             found_unknown_shapes);
    for (const auto& dim : output_shape.dim()) {
      output_size *= dim.size();
    }
    total_output_size += output_size;
    VLOG(1) << "Output Size: " << output_size
            << " Total Output Size:" << total_output_size;
  }
  return total_output_size;
}

std::vector<int64_t> OpLevelCostEstimator::CalculateOutputTensorSize(
    const OpInfo& op_info, bool* found_unknown_shapes) {
  std::vector<int64_t> output_tensor_size;
  output_tensor_size.reserve(op_info.outputs().size());
  // Use float as default for calculations.
  for (const auto& output : op_info.outputs()) {
    DataType dt = output.dtype();
    const auto& original_output_shape = output.shape();
    int64_t output_size = DataTypeSize(BaseType(dt));
    int num_dims = std::max(1, original_output_shape.dim_size());
    auto output_shape = MaybeGetMinimumShape(original_output_shape, num_dims,
                                             found_unknown_shapes);
    for (const auto& dim : output_shape.dim()) {
      output_size *= dim.size();
    }
    output_tensor_size.push_back(output_size);
  }
  return output_tensor_size;
}

Status OpLevelCostEstimator::PredictDefaultNodeCosts(
    const int64_t num_compute_ops, const OpContext& op_context,
    bool* found_unknown_shapes, NodeCosts* node_costs) {
  const auto& op_info = op_context.op_info;
  node_costs->num_compute_ops = num_compute_ops;
  node_costs->num_input_bytes_accessed =
      CalculateInputTensorSize(op_info, found_unknown_shapes);
  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,
```

```
        ") for op: ", op_context.op_info.ShortDebugString());
  }
  std::string filter_format = GetFilterFormat(op_context.op_info);
  if (filter_format != "HWIO" && filter_format != "OIHW" &&
      filter_format != "OIHW_VECT_I") {
    return errors::InvalidArgument(
        "Unsupported filter format (", filter_format,
        ") for op: ", op_context.op_info.ShortDebugString());
  }

  auto& conv_input = op_context.op_info.inputs(0);
  auto& filter = op_context.op_info.inputs(1);
  auto& side_input = op_context.op_info.inputs(3);
  auto& conv_input_scale = op_context.op_info.inputs(4);
  auto& side_input_scale = op_context.op_info.inputs(5);

  // Manually compute our convolution dimensions.
  bool found_unknown_shapes = false;
  auto dims = ConvolutionDimensionsFromInputs(
      conv_input.shape(), filter.shape(), op_context.op_info,
      &found_unknown_shapes);
  OpInfo::TensorProperties output;
  if (data_format == "NCHW" || data_format == "NCHW_VECT_C") {
    output = DescribeTensor(DT_FLOAT, {dims.batch, dims.oz, dims.oy, dims.ox});
  } else if (data_format == "NHWC") {
    output = DescribeTensor(DT_FLOAT, {dims.batch, dims.oy, dims.ox, dims.oz});
  }

  // Add the operations the fused op always computes.
  std::vector<OpContext> component_ops = {
      FusedChildContext(op_context, "Conv2D", output, {conv_input, filter}),
      FusedChildContext(op_context, "Mul", output, {output, conv_input_scale}),
      FusedChildContext(
          op_context, "BiasAdd", output,
          {output, output}),  // Note we're no longer using bias at all
      FusedChildContext(op_context, "Relu", output, {output})};

  // Add our side_input iff it's non-empty.
  if (side_input.shape().dim_size() > 0) {
    component_ops.push_back(FusedChildContext(op_context, "Mul", side_input,
                                              {side_input, side_input_scale}));
    component_ops.push_back(FusedChildContext(
        op_context, "Add", output,
        {output, output}));  // Note that we're not using side_input here
  }

  // Construct an op_context which definitely has our output shape.
  auto op_context_with_output = op_context;
  op_context_with_output.op_info.mutable_outputs()->Clear();
```

```
  *op_context_with_output.op_info.mutable_outputs()->Add() = output;

  // Construct component operations and run the cost computation.
  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return PredictFusedOp(op_context_with_output, component_ops, node_costs);
}

Status OpLevelCostEstimator::PredictMatMul(const OpContext& op_context,
                                           NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  bool found_unknown_shapes = false;
  int64_t num_compute_ops =
      CountMatMulOperations(op_info, &found_unknown_shapes);
  return PredictDefaultNodeCosts(num_compute_ops, op_context,
                                 &found_unknown_shapes, node_costs);
}

Status OpLevelCostEstimator::PredictEinsum(const OpContext& op_context,
                                           NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;

  auto it = op_info.attr().find("equation");
  if (it == op_info.attr().end()) {
    return errors::InvalidArgument("Einsum op doesn't have equation attr: ",
                                   op_info.ShortDebugString());
  }

  OpContext batch_matmul_op_context;
  bool found_unknown_shapes = false;
  bool success = GenerateBatchMatmulContextFromEinsum(
      op_context, &batch_matmul_op_context, &found_unknown_shapes);
  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  if (!success) {
    return PredictCostOfAnUnknownOp(op_context, node_costs);
  }
  return PredictNodeCosts(batch_matmul_op_context, node_costs);
}

Status OpLevelCostEstimator::PredictSparseTensorDenseMatMul(
    const OpContext& op_context, NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  bool found_unknown_shapes = false;
  // input[0]: indices in sparse matrix a
```

```
  // input[1]: values in sparse matrix a
  // input[2]: shape of matrix a
  // input[3]: matrix b
  // See
  // https://github.com/tensorflow/tensorflow/blob/9a43dfeac5/tensorflow/core/ops/sparse_ops.cc#L8
  int64_t num_elems_in_a =
      CalculateTensorElementCount(op_info.inputs(1), &found_unknown_shapes);
  auto b_matrix = op_info.inputs(3);
  auto b_matrix_shape =
      MaybeGetMinimumShape(b_matrix.shape(), 2, &found_unknown_shapes);
  int64_t n_dim = b_matrix_shape.dim(1).size();

  // Each element in A is multiplied and added with an element from each column
  // in b.
  const int64_t op_count = kOpsPerMac * num_elems_in_a * n_dim;

  int64_t a_indices_input_size =
      CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
  int64_t a_values_input_size =
      CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
  int64_t a_shape_input_size =
      CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
  int64_t b_input_size =
      num_elems_in_a * n_dim * DataTypeSize(BaseType(b_matrix.dtype()));
  int64_t output_size = CalculateOutputSize(op_info, &found_unknown_shapes);

  node_costs->num_compute_ops = op_count;
  node_costs->num_input_bytes_accessed = {a_indices_input_size,
                                          a_values_input_size,
                                          a_shape_input_size, b_input_size};
  node_costs->num_output_bytes_accessed = {output_size};
  if (found_unknown_shapes) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

Status OpLevelCostEstimator::PredictNoOp(const OpContext& op_context,
                                         NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  VLOG(1) << "Op:" << op_info.op() << " Execution Time 0 (ns)";
  // By default, NodeCosts is initialized to zero ops and bytes.
  return Status::OK();
}

Status OpLevelCostEstimator::PredictPureMemoryOp(const OpContext& op_context,
                                                 NodeCosts* node_costs) const {
  // Each output element is a copy of some element from input, with no required
```

```
  // computation, so just compute memory costs.
  bool found_unknown_shapes = false;
  node_costs->num_nodes_with_pure_memory_op = 1;
  return PredictDefaultNodeCosts(0, op_context, &found_unknown_shapes,
                                 node_costs);
}

Status OpLevelCostEstimator::PredictIdentity(const OpContext& op_context,
                                             NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  VLOG(1) << "Op:" << op_info.op() << " Minimum cost for Identity";
  node_costs->minimum_cost_op = true;
  node_costs->num_compute_ops = kMinComputeOp;
  // Identity op internally pass input tensor buffer's pointer to the output
  // tensor buffer; no actual memory operation.
  node_costs->num_input_bytes_accessed = {0};
  node_costs->num_output_bytes_accessed = {0};
  bool inaccurate = false;
  node_costs->max_memory = CalculateOutputSize(op_info, &inaccurate);
  if (inaccurate) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

Status OpLevelCostEstimator::PredictVariable(const OpContext& op_context,
                                             NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  VLOG(1) << "Op:" << op_info.op() << " Minimum cost for Variable";
  node_costs->minimum_cost_op = true;
  node_costs->num_compute_ops = kMinComputeOp;
  // Variables are persistent ops; initialized before step; hence, no memory
  // cost.
  node_costs->num_input_bytes_accessed = {0};
  node_costs->num_output_bytes_accessed = {0};
  bool inaccurate = false;
  node_costs->persistent_memory = CalculateOutputSize(op_info, &inaccurate);
  if (inaccurate) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

Status OpLevelCostEstimator::PredictBatchMatMul(const OpContext& op_context,
                                                NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  bool found_unknown_shapes = false;
```

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

Status OpLevelCostEstimator::PredictMetadata(const OpContext& op_context,
                                             NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  node_costs->minimum_cost_op = true;
  node_costs->num_compute_ops = kMinComputeOp;
  node_costs->num_input_bytes_accessed = {0};
  node_costs->num_output_bytes_accessed = {0};
  bool inaccurate = false;
  node_costs->max_memory = CalculateOutputSize(op_info, &inaccurate);
  if (inaccurate) {
    node_costs->inaccurate = true;
    node_costs->num_nodes_with_unknown_shapes = 1;
  }
  return Status::OK();
}

Status OpLevelCostEstimator::PredictGatherOrSlice(const OpContext& op_context,
                                                  NodeCosts* node_costs) const {
  // Gather & Slice ops can have a very large input, but only access a small
  // part of it. For these op the size of the output determines the memory cost.
  const auto& op_info = op_context.op_info;

  const int inputs_needed = op_info.op() == "Slice" ? 3 : 2;
  if (op_info.outputs_size() == 0 || op_info.inputs_size() < inputs_needed) {
    return errors::InvalidArgument(
        op_info.op(),
        " Op doesn't have valid input / output: ", op_info.ShortDebugString());
  }

  bool unknown_shapes = false;

  // Each output element is a copy of some element from input.
  // For roofline estimate we assume each copy has a unit cost.
  const int64_t op_count =
      CalculateTensorElementCount(op_info.outputs(0), &unknown_shapes);
  node_costs->num_compute_ops = op_count;

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

  node_costs->num_input_bytes_accessed.reserve(op_info.inputs().size());
  int64_t input_size = output_size;
  // 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();
  }
```

```
  const int64_t op_count = num_indices * num_elems_in_ref_per_index;
  node_costs->num_compute_ops = op_count;

  // Sparsely access ref so input size depends on the number of operations
  int64_t ref_input_size =
      op_count * DataTypeSize(BaseType(op_info.inputs(0).dtype()));
  int64_t indices_input_size =
      CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
  int64_t updates_input_size =
      CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
  node_costs->num_input_bytes_accessed = {ref_input_size, indices_input_size,
                                          updates_input_size};

  // Sparsely access ref so output size depends on the number of operations
  int64_t output_size =
      op_count * DataTypeSize(BaseType(op_info.outputs(0).dtype()));
  node_costs->num_output_bytes_accessed = {output_size};

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

Status OpLevelCostEstimator::PredictFusedOp(
    const OpContext& op_context,
    const std::vector<OpContext>& fused_op_contexts,
    NodeCosts* node_costs) const {
  // Note that PredictDefaultNodeCosts will get the correct memory costs from
  // the node's inputs and outputs; but we don't want to have to re-implement
  // the logic for computing the operation count of each of our component
  // operations here; so we simply add the compute times of each component
  // operation, then update the cost.
  bool found_unknown_shapes = false;
  Status s =
      PredictDefaultNodeCosts(0, op_context, &found_unknown_shapes, node_costs);

  for (auto& fused_op : fused_op_contexts) {
    NodeCosts fused_node_costs;
    s.Update(PredictNodeCosts(fused_op, &fused_node_costs));
    node_costs->num_compute_ops += fused_node_costs.num_compute_ops;
    node_costs->inaccurate |= fused_node_costs.inaccurate;
    // Set, not increment. Note that we are predicting the cost of one fused
    // node, not a function node composed of many nodes.
    node_costs->num_nodes_with_unknown_shapes |=
        fused_node_costs.num_nodes_with_unknown_shapes;
    node_costs->num_nodes_with_unknown_op_type |=
        fused_node_costs.num_nodes_with_unknown_op_type;
```

```
    node_costs->num_nodes_with_pure_memory_op |=
        fused_node_costs.num_nodes_with_pure_memory_op;
  }

  return Status::OK();
}

/* static */
OpContext OpLevelCostEstimator::FusedChildContext(
    const OpContext& parent, const std::string& op_name,
    const OpInfo::TensorProperties& output,
    const std::vector<OpInfo::TensorProperties>& inputs) {
  // Setup the base parameters of our new context.
  OpContext new_context;
  new_context.name = op_name;
  new_context.device_name = parent.device_name;
  new_context.op_info = parent.op_info;
  new_context.op_info.set_op(op_name);

  // Setup the inputs of our new context.
  new_context.op_info.mutable_inputs()->Clear();
  for (const auto& input : inputs) {
    *new_context.op_info.mutable_inputs()->Add() = input;
  }

  // Setup the output of our new context.
  new_context.op_info.mutable_outputs()->Clear();
  *new_context.op_info.mutable_outputs()->Add() = output;

  return new_context;
}

/* static */
OpInfo::TensorProperties OpLevelCostEstimator::DescribeTensor(
    DataType type, const std::vector<int64_t>& dims) {
  OpInfo::TensorProperties ret;
  ret.set_dtype(type);

  auto shape = ret.mutable_shape();
  for (const int dim : dims) {
    shape->add_dim()->set_size(dim);
  }

  return ret;
}

/* static */
OpLevelCostEstimator::ConvolutionDimensions
OpLevelCostEstimator::OpDimensionsFromInputs(
```

```
    const TensorShapeProto& original_image_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();
  auto image_shape =
      MaybeGetMinimumShape(original_image_shape, 4, found_unknown_shapes);
  VLOG(2) << "Image shape: " << image_shape.DebugString();

  int x_index, y_index, channel_index;
  const std::string& data_format = GetDataFormat(op_info);
  if (data_format == "NCHW") {
    channel_index = 1;
    y_index = 2;
    x_index = 3;
  } else {
    y_index = 1;
    x_index = 2;
    channel_index = 3;
  }
  int64_t batch = image_shape.dim(0).size();
  int64_t ix = image_shape.dim(x_index).size();
  int64_t iy = image_shape.dim(y_index).size();
  int64_t iz = image_shape.dim(channel_index).size();

  // Note that FusedBatchNorm doesn't have ksize attr, but GetKernelSize returns
  // {1, 1, 1, 1} in that case.
  std::vector<int64_t> ksize = GetKernelSize(op_info);
  int64_t kx = ksize[x_index];
  int64_t ky = ksize[y_index];
  // These ops don't support groupwise operation, therefore kz == iz.
  int64_t kz = iz;

  std::vector<int64_t> strides = GetStrides(op_info);
  int64_t sx = strides[x_index];
  int64_t sy = strides[y_index];
  const auto padding = GetPadding(op_info);

  int64_t ox = GetOutputSize(ix, kx, sx, padding);
  int64_t oy = GetOutputSize(iy, ky, sy, padding);
  int64_t oz = iz;

  OpLevelCostEstimator::ConvolutionDimensions conv_dims = {
      batch, ix, iy, iz, kx, ky, kz, oz, ox, oy, sx, sy, padding};
  return conv_dims;
}

Status OpLevelCostEstimator::PredictMaxPool(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 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;
```

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

Status OpLevelCostEstimator::PredictAvgPoolGrad(const OpContext& op_context,
                                                NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;
  const auto& op_info = op_context.op_info;
  // x's shape: op_info.inputs(0)
  // y_grad: op_info.inputs(1)

  // Extract x_shape from op_info.inputs(0).value() or op_info.outputs(0).
  bool shape_found = false;
  TensorShapeProto x_shape;
  if (op_info.inputs_size() >= 1 && op_info.inputs(0).has_value()) {
    const TensorProto& value = op_info.inputs(0).value();
    shape_found = GetTensorShapeProtoFromTensorProto(value, &x_shape);
  }
  if (!shape_found && op_info.outputs_size() > 0) {
    x_shape = op_info.outputs(0).shape();
    shape_found = true;
  }
  if (!shape_found) {
    // Set the minimum shape that's feasible.
    x_shape.Clear();
    for (int i = 0; i < 4; ++i) {
      x_shape.add_dim()->set_size(1);
    }
    found_unknown_shapes = true;
  }

  ConvolutionDimensions dims =
      OpDimensionsFromInputs(x_shape, op_info, &found_unknown_shapes);

  int64_t ops = 0;
  if (dims.kx <= dims.sx && dims.ky <= dims.sy) {
    // Non-overlapping window.
    ops = dims.batch * dims.iz * (dims.ix * dims.iy + dims.ox * dims.oy);
  } else {
    // Overlapping window.
    ops = dims.batch * dims.iz *
          (dims.ix * dims.iy + dims.ox * dims.oy * (dims.kx * dims.ky + 1));
  }
  auto s = PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
                                   node_costs);
  node_costs->max_memory = node_costs->num_total_output_bytes();
  return s;
}

```

```
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();
}
```

```
Status OpLevelCostEstimator::PredictFusedBatchNormGrad(
    const OpContext& op_context, NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;
  const auto& op_info = op_context.op_info;
  // y_backprop: op_info.inputs(0)
  // x: op_info.inputs(1)
  // scale: op_info.inputs(2)
  // mean: op_info.inputs(3)
  // variance or inverse of variance: op_info.inputs(4)
  ConvolutionDimensions dims = OpDimensionsFromInputs(
      op_info.inputs(1).shape(), op_info, &found_unknown_shapes);

  int64_t ops = 0;
  const auto rsqrt_cost = Eigen::internal::functor_traits<
      Eigen::internal::scalar_rsqrt_op<float>>::Cost;
  ops = dims.iz * (dims.batch * dims.ix * dims.iy * 11 + 5 + rsqrt_cost);
  node_costs->num_compute_ops = ops;

  const int64_t size_nhwc =
      CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
  const int64_t size_c =
      CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
  // TODO(dyoon): fix missing memory cost for variance input (size_c) and
  // yet another read of y_backprop (size_nhwc) internally.
  node_costs->num_input_bytes_accessed = {size_nhwc, size_nhwc, size_c, size_c};
  node_costs->num_output_bytes_accessed = {size_nhwc, size_c, size_c};
  // FusedBatchNormGrad has to read y_backprop internally.
  node_costs->internal_read_bytes = 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();
}

Status OpLevelCostEstimator::PredictNaryOp(const OpContext& op_context,
                                           NodeCosts* node_costs) const {
  const auto& op_info = op_context.op_info;
  bool found_unknown_shapes = false;
  // Calculate the largest known tensor size across all inputs and output.
  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));
  }
  // Also calculate the output shape possibly resulting from broadcasting.
  // Note that the some Nary ops (such as AddN) do not support broadcasting,
  // but we're including this here for completeness.
  if (op_info.inputs_size() >= 2) {
    op_count = std::max(op_count, CwiseOutputElementCount(op_info));
  }

  // Nary ops perform one operation for every element in every input tensor.
  op_count *= op_info.inputs_size() - 1;

  const auto sum_cost = Eigen::internal::functor_traits<
      Eigen::internal::scalar_sum_op<float>>::Cost;
  return PredictDefaultNodeCosts(op_count * sum_cost, op_context,
                                 &found_unknown_shapes, node_costs);
}

// softmax[i, j] = exp(logits[i, j]) / sum_j(exp(logits[i, j]))
Status OpLevelCostEstimator::PredictSoftmax(const OpContext& op_context,
                                            NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;
  const int64_t logits_size = CalculateTensorElementCount(
      op_context.op_info.inputs(0), &found_unknown_shapes);
  // Softmax input rank should be >=1.
  TensorShapeProto logits_shape = op_context.op_info.inputs(0).shape();
  if (logits_shape.unknown_rank() || logits_shape.dim_size() == 0) {
    return errors::InvalidArgument("Softmax op has invalid input: ",
                                   op_context.op_info.ShortDebugString());
  }

#define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost

  // Every element of <logits> will be exponentiated, have that result included
  // in a sum across j, and also have that result multiplied by the reciprocal
  // of the sum_j. In addition, we'll compute 1/sum_j for every i.
  auto ops =
      (EIGEN_COST(scalar_exp_op<float>) + EIGEN_COST(scalar_sum_op<float>) +
       EIGEN_COST(scalar_product_op<float>)) *
          logits_size +
      EIGEN_COST(scalar_inverse_op<float>) * logits_shape.dim(0).size();

#undef EIGEN_COST
  return PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
                                 node_costs);
}

Status OpLevelCostEstimator::PredictResizeBilinear(
    const OpContext& op_context, NodeCosts* node_costs) const {
```

```
  bool found_unknown_shapes = false;

  if (op_context.op_info.outputs().empty() ||
      op_context.op_info.inputs().empty()) {
    return errors::InvalidArgument(
        "ResizeBilinear op has invalid input / output ",
        op_context.op_info.ShortDebugString());
  }

  const int64_t output_elements = CalculateTensorElementCount(
      op_context.op_info.outputs(0), &found_unknown_shapes);

  const auto half_pixel_centers =
      op_context.op_info.attr().find("half_pixel_centers");
  bool use_half_pixel_centers = false;
  if (half_pixel_centers == op_context.op_info.attr().end()) {
    LOG(WARNING) << "half_pixel_centers attr not set for ResizeBilinear.";
    return PredictCostOfAnUnknownOp(op_context, node_costs);
  } else {
    use_half_pixel_centers = half_pixel_centers->second.b();
  }

  // Compose cost of bilinear interpolation.
  int64_t ops = 0;

#define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost
  const auto sub_cost_float = EIGEN_COST(scalar_difference_op<float>);
  const auto sub_cost_int = EIGEN_COST(scalar_difference_op<int64_t>);
  const auto add_cost = EIGEN_COST(scalar_sum_op<float>);
  const auto mul_cost = EIGEN_COST(scalar_product_op<float>);
  const auto floor_cost = EIGEN_COST(scalar_floor_op<float>);
  const auto max_cost = EIGEN_COST(scalar_max_op<int64_t>);
  const auto min_cost = EIGEN_COST(scalar_min_op<int64_t>);
  const auto cast_to_int_cost = Eigen::internal::functor_traits<
      Eigen::internal::scalar_cast_op<float, int64_t>>::Cost;
  const auto cast_to_float_cost = Eigen::internal::functor_traits<
      Eigen::internal::scalar_cast_op<int64_t, float>>::Cost;
  const auto ceil_cost = EIGEN_COST(scalar_ceil_op<float>);
#undef EIGEN_COST

  // Ops calculated from tensorflow/core/kernels/image/resize_bilinear_op.cc.

  // Op counts taken from resize_bilinear implementation on 07/21/2020.
  // Computed op counts may become inaccurate if resize_bilinear implementation
  // changes.

  // resize_bilinear has an optimization where the interpolation weights are
  // precomputed and cached. Given input tensors of size [B,H1,W1,C] and output
  // tensors of size [B,H2,W2,C], the last dimension C that needs to be accessed
```

```
  // in the input for interpolation are identical at every point in the output.
  // These values are cached in the compute_interpolation_weights function. For
  // a particular y in [0...H2-1], the rows to be accessed in the input are the
  // same. Likewise, for a particular x in [0...H2-1], the columns to be accsed
  // are the same. So the precomputation only needs to be done for H2 + W2
  // values.
  const auto output_shape = MaybeGetMinimumShape(
      op_context.op_info.outputs(0).shape(), 4, &found_unknown_shapes);
  // Assume H is dim 1 and W is dim 2 to match logic in resize_bilinear, which
  // also makes this assumption.
  const int64_t output_height = output_shape.dim(1).size();
  const int64_t output_width = output_shape.dim(2).size();
  // Add the ops done outside of the scaler function in
  // compute_interpolation_weights.
  int64_t interp_weight_cost = floor_cost + max_cost + min_cost +
                               sub_cost_float + sub_cost_int + ceil_cost +
                               cast_to_int_cost * 2;
  // There are two options for computing the weight of each pixel in the
  // interpolation. Algorithm can use pixel centers, or corners, for the
  // weight. Ops depend on the scaler function passed into
  // compute_interpolation_weights.
  if (use_half_pixel_centers) {
    // Ops for HalfPixelScalaer.
    interp_weight_cost +=
        add_cost + mul_cost + sub_cost_float + cast_to_float_cost;
  } else {
    // Ops for LegacyScaler.
    interp_weight_cost += cast_to_float_cost + mul_cost;
  }
  // Cost for the interpolation is multiplied by (H2 + w2), as mentioned above.
  ops += interp_weight_cost * (output_height + output_width);

  // Ops for computing the new values, done for every element. Logic is from
  // compute_lerp in the inner loop of resize_image which consists of:
  //   const float top = top_left + (top_right - top_left) * x_lerp;
  //   const float bottom = bottom_left + (bottom_right - bottom_left) * x_lerp;
  //   return top + (bottom - top) * y_lerp;
  ops += (add_cost * 3 + sub_cost_float * 3 + mul_cost * 3) * output_elements;

  return PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
                                 node_costs);
}

Status OpLevelCostEstimator::PredictCropAndResize(const OpContext& op_context,
                                                  NodeCosts* node_costs) const {
  bool found_unknown_shapes = false;

  const auto method = op_context.op_info.attr().find("method");
  bool use_bilinear_interp;
```

```
  if (method == op_context.op_info.attr().end() ||
      method->second.s() == "bilinear") {
    use_bilinear_interp = true;
  } else if (method->second.s() == "nearest") {
    use_bilinear_interp = false;
  } else {
    LOG(WARNING) << "method attr in CropAndResize invalid; expected bilinear "
                    "or nearest.";
    return PredictCostOfAnUnknownOp(op_context, node_costs);
  }

  const int64_t num_boxes = op_context.op_info.inputs(1).shape().dim(0).size();
  const auto crop_shape = MaybeGetMinimumShape(
      op_context.op_info.outputs(0).shape(), 4, &found_unknown_shapes);
  const int64_t crop_height = crop_shape.dim(1).size();
  const int64_t crop_width = crop_shape.dim(2).size();
  const int64_t output_elements = CalculateTensorElementCount(
      op_context.op_info.outputs(0), &found_unknown_shapes);

#define EIGEN_COST(X) Eigen::internal::functor_traits<Eigen::internal::X>::Cost
  const auto sub_cost = EIGEN_COST(scalar_difference_op<float>);
  const auto add_cost = EIGEN_COST(scalar_sum_op<float>);
  const auto mul_cost = EIGEN_COST(scalar_product_op<float>);
  auto div_cost = EIGEN_COST(scalar_div_cost<float>);
  const auto floor_cost = EIGEN_COST(scalar_floor_op<float>);
  const auto ceil_cost = EIGEN_COST(scalar_ceil_op<float>);
  auto round_cost = EIGEN_COST(scalar_round_op<float>);
  const auto cast_to_float_cost = Eigen::internal::functor_traits<
      Eigen::internal::scalar_cast_op<int64_t, float>>::Cost;
#undef EIGEN_COST

  // Computing ops following
  // tensorflow/core/kernels/image/crop_and_resize_op.cc at 08/25/2020. Op
  // calculation differs from rough estimate in implementation, as it separates
  // out cost per box from cost per pixel and cost per element.

  // Ops for variables height_scale and width_scale.
  int64_t ops = (sub_cost * 6 + mul_cost * 2 + div_cost * 2) * num_boxes;
  // Ops for variable in_y.
  ops += (mul_cost * 2 + sub_cost + add_cost) * crop_height * num_boxes;
  // Ops for variable in_x (same computation across both branches).
  ops += (mul_cost * 2 + sub_cost + add_cost) * crop_height * crop_width *
         num_boxes;
  // Specify op_cost based on the method.
  if (use_bilinear_interp) {
    // Ops for variables top_y_index, bottom_y_index, y_lerp.
    ops += (floor_cost + ceil_cost + sub_cost) * crop_height * num_boxes;
    // Ops for variables left_x, right_x, x_lerp;
    ops += (floor_cost + ceil_cost + sub_cost) * crop_height * crop_width *
```

```
              num_boxes;
      // Ops for innermost loop across depth.
      ops +=
          (cast_to_float_cost * 4 + add_cost * 3 + sub_cost * 3 + mul_cost * 3) *
          output_elements;
    } else /* method == "nearest" */ {
      // Ops for variables closest_x_index and closest_y_index.
      ops += round_cost * 2 * crop_height * crop_width * num_boxes;
      // Ops for innermost loop across depth.
      ops += cast_to_float_cost * output_elements;
    }
    return PredictDefaultNodeCosts(ops, op_context, &found_unknown_shapes,
                                   node_costs);
  }

}  // end namespace grappler
}  // end namespace tensorflow
```