

ffa202a17a ▾

...

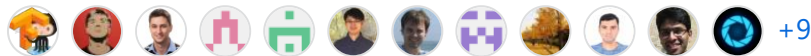
tensorflow / tensorflow / core / grappler / costs / op_level_cost_estimator.cc



tensorflow-gardener Merge pull request #51035 from slowy07:minor-fixing ... ✖

History

21 contributors



2692 lines (2437 sloc) | 110 KB

...

```

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

```

```

30
31 namespace tensorflow {
32 namespace grappler {
33
34 // TODO(dyoon): update op to Predict method map for TF ops with V2 or V3 suffix.
35 constexpr int kOpsPerMac = 2;
36 constexpr char kGuaranteeConst[] = "GuaranteeConst";
37 constexpr char kAddN[] = "AddN";
38 constexpr char kBitCast[] = "BitCast";
39 constexpr char kConcatV2[] = "ConcatV2";
40 constexpr char kConv2d[] = "Conv2D";
41 constexpr char kConv2dBackpropFilter[] = "Conv2DBackpropFilter";
42 constexpr char kConv2dBackpropInput[] = "Conv2DBackpropInput";
43 constexpr char kFusedConv2dBiasActivation[] = "FusedConv2dBiasActivation";
44 constexpr char kDataFormatVecPermute[] = "DataFormatVecPermute";
45 constexpr char kDepthToSpace[] = "DepthToSpace";
46 constexpr char kDepthwiseConv2dNative[] = "DepthwiseConv2dNative";
47 constexpr char kDepthwiseConv2dNativeBackpropFilter[] =
48     "DepthwiseConv2dNativeBackpropFilter";
49 constexpr char kDepthwiseConv2dNativeBackpropInput[] =
50     "DepthwiseConv2dNativeBackpropInput";
51 constexpr char kMatMul[] = "MatMul";
52 constexpr char kXlaEinsum[] = "XlaEinsum";
53 constexpr char kEinsum[] = "Einsum";
54 constexpr char kExpandDims[] = "ExpandDims";
55 constexpr char kFill[] = "Fill";
56 constexpr char kSparseMatMul[] = "SparseMatMul";
57 constexpr char kSparseTensorDenseMatMul[] = "SparseTensorDenseMatMul";
58 constexpr char kPlaceholder[] = "Placeholder";
59 constexpr char kIdentity[] = "Identity";
60 constexpr char kIdentityN[] = "IdentityN";
61 constexpr char kRefIdentity[] = "RefIdentity";
62 constexpr char kNoOp[] = "NoOp";
63 constexpr char kReshape[] = "Reshape";
64 constexpr char kSplit[] = "Split";
65 constexpr char kSqueeze[] = "Squeeze";
66 constexpr char kRecv[] = "_Recv";
67 constexpr char kSend[] = "_Send";
68 constexpr char kBatchMatMul[] = "BatchMatMul";
69 constexpr char kBatchMatMulV2[] = "BatchMatMulV2";
70 constexpr char kOneHot[] = "OneHot";
71 constexpr char kPack[] = "Pack";
72 constexpr char kRank[] = "Rank";
73 constexpr char kRange[] = "Range";
74 constexpr char kShape[] = "Shape";
75 constexpr char kShapeN[] = "ShapeN";
76 constexpr char kSize[] = "Size";
77 constexpr char kStopGradient[] = "StopGradient";
78 constexpr char kPreventGradient[] = "PreventGradient";

```

```

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

```

```

128 namespace {
129
130 std::string GetDataFormat(const OpInfo& op_info) {
131     std::string data_format = "NHWC"; // Default format.
132     if (op_info.attr().find("data_format") != op_info.attr().end()) {
133         data_format = op_info.attr().at("data_format").s();
134     }
135     return data_format;
136 }
137
138 std::string GetFilterFormat(const OpInfo& op_info) {
139     std::string filter_format = "HWIO"; // Default format.
140     if (op_info.attr().find("filter_format") != op_info.attr().end()) {
141         filter_format = op_info.attr().at("filter_format").s();
142     }
143     return filter_format;
144 }
145
146 Padding GetPadding(const OpInfo& op_info) {
147     if (op_info.attr().find("padding") != op_info.attr().end() &&
148         op_info.attr().at("padding").s() == "VALID") {
149         return Padding::VALID;
150     }
151     return Padding::SAME; // Default padding.
152 }
153
154 bool IsTraining(const OpInfo& op_info) {
155     if (op_info.attr().find("is_training") != op_info.attr().end() &&
156         op_info.attr().at("is_training").b()) {
157         return true;
158     }
159     return false;
160 }
161
162 // TODO(dyoon): support non-4D tensors in the cost functions of convolution
163 // related ops (Conv, Pool, BatchNorm, and their backprops) and the related
164 // helper functions.
165 std::vector<int64_t> GetStrides(const OpInfo& op_info) {
166     if (op_info.attr().find("strides") != op_info.attr().end()) {
167         const auto strides = op_info.attr().at("strides").list().i();
168         DCHECK(strides.size() == 4)
169             << "Attr strides is not a length-4 vector: " << op_info.DebugString();
170         if (strides.size() != 4) return {1, 1, 1, 1};
171         return {strides[0], strides[1], strides[2], strides[3]};
172     }
173     return {1, 1, 1, 1};
174 }
175
176 std::vector<int64_t> GetKernelSize(const OpInfo& op_info) {

```

```

177     if (op_info.attr().find("ksize") != op_info.attr().end()) {
178         const auto ksize = op_info.attr().at("ksize").list().i();
179         DCHECK(ksize.size() == 4)
180             << "Attr ksize is not a length-4 vector: " << op_info.DebugString();
181         if (ksize.size() != 4) return {1, 1, 1, 1};
182         return {ksize[0], ksize[1], ksize[2], ksize[3]};
183     }
184     // Note that FusedBatchNorm doesn't have ksize attr, but GetKernelSize returns
185     // {1, 1, 1, 1} in that case.
186     return {1, 1, 1, 1};
187 }
188
189 int64_t GetOutputSize(const int64_t input, const int64_t filter,
190                      const int64_t stride, const Padding& padding) {
191     // Logic for calculating output shape is from GetWindowedOutputSizeVerbose()
192     // function in third_party/tensorflow/core/framework/common_shape_fns.cc.
193     if (padding == Padding::VALID) {
194         return (input - filter + stride) / stride;
195     } else { // SAME.
196         return (input + stride - 1) / stride;
197     }
198 }
199
200 // Return the output element count of a multi-input element-wise op considering
201 // broadcasting.
202 int64_t CwiseOutputElementCount(const OpInfo& op_info) {
203     int max_rank = 1;
204     for (const OpInfo::TensorProperties& input_properties : op_info.inputs()) {
205         max_rank = std::max(max_rank, input_properties.shape().dim_size());
206     }
207
208     TensorShapeProto output_shape;
209     output_shape.mutable_dim()->Reserve(max_rank);
210     for (int i = 0; i < max_rank; ++i) {
211         output_shape.add_dim();
212     }
213
214     // Expand the shape of the output to follow the numpy-style broadcast rule
215     // which matches each input starting with the trailing dimensions and working
216     // its way forward. To do this, iterate through each input shape's dimensions
217     // in reverse order, and potentially increase the corresponding output
218     // dimension.
219     for (const OpInfo::TensorProperties& input_properties : op_info.inputs()) {
220         const TensorShapeProto& input_shape = input_properties.shape();
221         for (int i = input_shape.dim_size() - 1; i >= 0; --i) {
222             int output_shape_dim_index =
223                 i + output_shape.dim_size() - input_shape.dim_size();
224             output_shape.mutable_dim(output_shape_dim_index)
225                 ->set_size(std::max(output_shape.dim(output_shape_dim_index).size(),

```

```

226         input_shape.dim(i).size());
227     }
228 }
229
230 int64_t count = 1;
231 for (int i = 0; i < output_shape.dim_size(); i++) {
232     count *= output_shape.dim(i).size();
233 }
234 return count;
235 }
236
237 // Helper function for determining whether there are repeated indices in the
238 // input Einsum equation.
239 bool CheckRepeatedDimensions(const absl::string_view dim_str) {
240     int str_size = dim_str.size();
241     for (int idx = 0; idx < str_size - 1; idx++) {
242         if (dim_str.find(dim_str[idx], idx + 1) != std::string::npos) {
243             return true;
244         }
245     }
246     return false;
247 }
248
249 // Auxiliary function for determining whether OpLevelCostEstimator is compatible
250 // with a given Einsum.
251 bool IsEinsumCorrectlyFormed(const OpContext& einsum_context) {
252     const auto& op_info = einsum_context.op_info;
253
254     auto it = op_info.attr().find("equation");
255     if (it == op_info.attr().end()) return false;
256     const absl::string_view equation = it->second.s();
257     std::vector<std::string> equation_split = absl::StrSplit(equation, "->");
258
259     if (equation_split.empty()) {
260         LOG(WARNING) << "Einsum with malformed equation";
261         return false;
262     }
263     std::vector<absl::string_view> input_split =
264         absl::StrSplit(equation_split[0], ',');
265
266     // The current model covers Einsum operations with two operands and a RHS
267     if (op_info.inputs_size() != 2 || equation_split.size() != 2) {
268         VLOG(1) << "Missing accurate estimator for op: " << op_info.op();
269         return false;
270     }
271     const auto& a_input = op_info.inputs(0);
272     const auto& b_input = op_info.inputs(1);
273     absl::string_view rhs_str = equation_split[1];
274     absl::string_view a_input_str = input_split[0];

```

```

275     absl::string_view b_input_str = input_split[1];
276
277     // Ellipsis are not currently supported
278     if (absl::StrContains(a_input_str, "...") ||
279         absl::StrContains(b_input_str, "...")) {
280         VLOG(1) << "Missing accurate estimator for op: " << op_info.op()
281             << ", ellipsis not supported";
282         return false;
283     }
284
285     constexpr int kMatrixRank = 2;
286
287     bool a_input_shape_unknown = false;
288     bool b_input_shape_unknown = false;
289
290     TensorShapeProto a_input_shape = MaybeGetMinimumShape(
291         a_input.shape(), std::max(kMatrixRank, a_input.shape().dim_size()),
292         &a_input_shape_unknown);
293     TensorShapeProto b_input_shape = MaybeGetMinimumShape(
294         b_input.shape(), std::max(kMatrixRank, b_input.shape().dim_size()),
295         &b_input_shape_unknown);
296
297     if (a_input_str.size() != static_cast<size_t>(a_input_shape.dim_size()) ||
298         b_input_str.size() != static_cast<size_t>(b_input_shape.dim_size())) {
299         VLOG(1) << "Missing accurate estimator for op: " << op_info.op()
300             << ", equation subscripts don't match tensor rank.";
301         return false;
302     }
303
304     // Subscripts where axis appears more than once for a single input are not yet
305     // supported
306     if (CheckRepeatedDimensions(a_input_str) ||
307         CheckRepeatedDimensions(b_input_str) ||
308         CheckRepeatedDimensions(rhs_str)) {
309         VLOG(1) << "Missing accurate estimator for op: " << op_info.op()
310             << ", Subscripts where axis appears more than once for a single "
311             << "input are not yet supported";
312         return false;
313     }
314
315     return true;
316 }
317
318 } // namespace
319
320 // Return a minimum shape if the shape is unknown. If known, return the original
321 // shape.
322 TensorShapeProto MaybeGetMinimumShape(const TensorShapeProto& original_shape,
323     int rank, bool* found_unknown_shapes) {

```

```

324     auto shape = original_shape;
325     bool is_scalar = !shape.unknown_rank() && shape.dim_size() == 0;
326
327     if (shape.unknown_rank() || (!is_scalar && shape.dim_size() < rank)) {
328         *found_unknown_shapes = true;
329         VLOG(2) << "Use minimum shape because the rank is unknown.";
330         // The size of each dimension is at least 1, if unknown.
331         for (int i = shape.dim_size(); i < rank; i++) {
332             shape.add_dim()->set_size(1);
333         }
334     } else if (is_scalar) {
335         for (int i = 0; i < rank; i++) {
336             shape.add_dim()->set_size(1);
337         }
338     } else if (shape.dim_size() > rank) {
339         *found_unknown_shapes = true;
340         shape.clear_dim();
341         for (int i = 0; i < rank; i++) {
342             shape.add_dim()->set_size(original_shape.dim(i).size());
343         }
344     } else {
345         for (int i = 0; i < shape.dim_size(); i++) {
346             if (shape.dim(i).size() < 0) {
347                 *found_unknown_shapes = true;
348                 VLOG(2) << "Use minimum dim size 1 because the shape is unknown.";
349                 // The size of each dimension is at least 1, if unknown.
350                 shape.mutable_dim(i)->set_size(1);
351             }
352         }
353     }
354     return shape;
355 }
356
357 OpLevelCostEstimator::OpLevelCostEstimator() {
358     // Syntactic sugar to build and return a lambda that takes an OpInfo and
359     // returns a cost.
360     typedef Status (OpLevelCostEstimator::*CostImpl)(const OpContext& op_context,
361                                                         NodeCosts*) const;
362     auto wrap = [this](CostImpl impl)
363         -> std::function<Status(const OpContext&, NodeCosts*)> {
364         return [this, impl](const OpContext& op_context, NodeCosts* node_costs) {
365             return (this->*impl)(op_context, node_costs);
366         };
367     };
368
369     device_cost_impl_.emplace(kConv2d,
370                               wrap(&OpLevelCostEstimator::PredictConv2D));
371     device_cost_impl_.emplace(
372         kConv2dBackpropFilter,

```



```

373         wrap(&OpLevelCostEstimator::PredictConv2DBackpropFilter));
374 device_cost_impl_.emplace(
375     kConv2dBackpropInput,
376     wrap(&OpLevelCostEstimator::PredictConv2DBackpropInput));
377 device_cost_impl_.emplace(
378     kFusedConv2dBiasActivation,
379     wrap(&OpLevelCostEstimator::PredictFusedConv2dBiasActivation));
380 // reuse Conv2D for DepthwiseConv2dNative because the calculation is the
381 // same although the actual meaning of the parameters are different. See
382 // comments in PredictConv2D and related functions
383 device_cost_impl_.emplace(kDepthwiseConv2dNative,
384     wrap(&OpLevelCostEstimator::PredictConv2D));
385 device_cost_impl_.emplace(
386     kDepthwiseConv2dNativeBackpropFilter,
387     wrap(&OpLevelCostEstimator::PredictConv2DBackpropFilter));
388 device_cost_impl_.emplace(
389     kDepthwiseConv2dNativeBackpropInput,
390     wrap(&OpLevelCostEstimator::PredictConv2DBackpropInput));
391 device_cost_impl_.emplace(kMatMul,
392     wrap(&OpLevelCostEstimator::PredictMatMul));
393 device_cost_impl_.emplace(kSparseMatMul,
394     wrap(&OpLevelCostEstimator::PredictMatMul));
395 device_cost_impl_.emplace(
396     kSparseTensorDenseMatMul,
397     wrap(&OpLevelCostEstimator::PredictSparseTensorDenseMatMul));
398 device_cost_impl_.emplace(kBatchMatMul,
399     wrap(&OpLevelCostEstimator::PredictBatchMatMul));
400 device_cost_impl_.emplace(kBatchMatMulV2,
401     wrap(&OpLevelCostEstimator::PredictBatchMatMul));
402 device_cost_impl_.emplace(kQuantizedMatMul,
403     wrap(&OpLevelCostEstimator::PredictMatMul));
404 device_cost_impl_.emplace(kQuantizedMatMulV2,
405     wrap(&OpLevelCostEstimator::PredictMatMul));
406 device_cost_impl_.emplace(kXlaEinsum,
407     wrap(&OpLevelCostEstimator::PredictEinsum));
408 device_cost_impl_.emplace(kEinsum,
409     wrap(&OpLevelCostEstimator::PredictEinsum));
410
411 device_cost_impl_.emplace(kNoOp, wrap(&OpLevelCostEstimator::PredictNoOp));
412 device_cost_impl_.emplace(kGuaranteeConst,
413     wrap(&OpLevelCostEstimator::PredictNoOp));
414
415 device_cost_impl_.emplace(kGather,
416     wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
417 device_cost_impl_.emplace(kGatherNd,
418     wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
419 device_cost_impl_.emplace(kGatherV2,
420     wrap(&OpLevelCostEstimator::PredictGatherOrSlice));
421 device_cost_impl_.emplace(kScatterAdd,

```

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

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

```

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

```

```

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

```

```

618     elementwise_ops_.emplace("Less", 1);
619     elementwise_ops_.emplace("LessEqual", 1);
620     elementwise_ops_.emplace("LogicalAnd", EIGEN_COST(scalar_boolean_and_op));
621     elementwise_ops_.emplace("LogicalNot", 1);
622     elementwise_ops_.emplace("LogicalOr", EIGEN_COST(scalar_boolean_or_op));
623     elementwise_ops_.emplace("Maximum", EIGEN_COST(scalar_max_op<float>));
624     elementwise_ops_.emplace("Minimum", EIGEN_COST(scalar_min_op<float>));
625     elementwise_ops_.emplace("Mod", EIGEN_COST(scalar_mod_op<float>));
626     elementwise_ops_.emplace("Mul", EIGEN_COST(scalar_product_op<float>));
627     elementwise_ops_.emplace("NotEqual", 1);
628     elementwise_ops_.emplace("QuantizedAdd", EIGEN_COST(scalar_sum_op<float>));
629     elementwise_ops_.emplace("QuantizedMul",
630                               EIGEN_COST(scalar_product_op<float>));
631     elementwise_ops_.emplace("RealDiv", EIGEN_COST(scalar_quotient_op<float>));
632     elementwise_ops_.emplace("ReluGrad", EIGEN_COST(scalar_max_op<float>));
633     elementwise_ops_.emplace("Select", EIGEN_COST(scalar_boolean_or_op));
634     elementwise_ops_.emplace("SelectV2", EIGEN_COST(scalar_boolean_or_op));
635     elementwise_ops_.emplace("SquaredDifference",
636                               EIGEN_COST(scalar_square_op<float>) +
637                               EIGEN_COST(scalar_difference_op<float>));
638     elementwise_ops_.emplace("Sub", EIGEN_COST(scalar_difference_op<float>));
639     elementwise_ops_.emplace("TruncateDiv",
640                               EIGEN_COST(scalar_quotient_op<float>));
641     elementwise_ops_.emplace("TruncateMod", EIGEN_COST(scalar_mod_op<float>));
642     elementwise_ops_.emplace("Where", 1);
643
644     #undef EIGEN_COST
645
646     // By default, use sum of memory_time and compute_time for execution_time.
647     compute_memory_overlap_ = false;
648 }
649
650 Costs OpLevelCostEstimator::PredictCosts(const OpContext& op_context) const {
651     Costs costs;
652     NodeCosts node_costs;
653     if (PredictNodeCosts(op_context, &node_costs).ok()) {
654         if (node_costs.has_costs) {
655             return node_costs.costs;
656         }
657         // Convert NodeCosts to Costs.
658         if (node_costs.minimum_cost_op) {
659             // Override to minimum cost; Note that some ops with minimum cost may have
660             // non-typical device (e.g., channel for _Send), which may fail with
661             // GetDeviceInfo(), called from PredictOpCountBasedCost(). Make sure we
662             // directly set minimum values to Costs here, not calling
663             // PredictOpCountBasedCost().
664             costs.compute_time = kMinComputeTime;
665             costs.execution_time = kMinComputeTime;
666             costs.memory_time = 0;

```



```

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

```

```

716     return PredictCostOfAnUnknownOp(op_context, node_costs);
717 }
718
719 // This method assumes a typical system composed of CPUs and GPUs, connected
720 // through PCIe. To define device info more precisely, override this method.
721 DeviceInfo OpLevelCostEstimator::GetDeviceInfo(
722     const DeviceProperties& device) const {
723     double gflops = -1;
724     double gb_per_sec = -1;
725
726     if (device.type() == "CPU") {
727         // Check if vector instructions are available, and refine performance
728         // prediction based on this.
729         // Frequencies are stored in MHz in the DeviceProperties.
730         gflops = device.num_cores() * device.frequency() * 1e-3;
731         if (gb_per_sec < 0) {
732             if (device.bandwidth() > 0) {
733                 gb_per_sec = device.bandwidth() / 1e6;
734             } else {
735                 gb_per_sec = 32;
736             }
737         }
738     } else if (device.type() == "GPU") {
739         const auto& device_env = device.environment();
740         auto it = device_env.find("architecture");
741         if (it != device_env.end()) {
742             const std::string architecture = device_env.at("architecture");
743             int cores_per_multiprocessor;
744             if (architecture < "3") {
745                 // Fermi
746                 cores_per_multiprocessor = 32;
747             } else if (architecture < "4") {
748                 // Kepler
749                 cores_per_multiprocessor = 192;
750             } else if (architecture < "6") {
751                 // Maxwell
752                 cores_per_multiprocessor = 128;
753             } else {
754                 // Pascal (compute capability version 6) and Volta (compute capability
755                 // version 7)
756                 cores_per_multiprocessor = 64;
757             }
758             gflops = device.num_cores() * device.frequency() * 1e-3 *
759                 cores_per_multiprocessor * kOpsPerMac;
760             if (device.bandwidth() > 0) {
761                 gb_per_sec = device.bandwidth() / 1e6;
762             } else {
763                 gb_per_sec = 100;
764             }
765         }
766     }
767 }

```



```

765     } else {
766         // Architecture is not available (ex: pluggable device), return default
767         // value.
768         gflops = 100;    // Dummy value;
769         gb_per_sec = 12; // default PCIe x16 gen3.
770     }
771 } else {
772     LOG_EVERY_N(WARNING, 1000) << "Unknown device type: " << device.type()
773         << ", assuming PCIe between CPU and GPU.";
774     gflops = 1; // Dummy value; data transfer ops would not have compute ops.
775     gb_per_sec = 12; // default PCIe x16 gen3.
776 }
777 VLOG(1) << "Device: " << device.type() << " gflops: " << gflops
778     << " gb_per_sec: " << gb_per_sec;
779
780 return DeviceInfo(gflops, gb_per_sec);
781 }
782
783 Status OpLevelCostEstimator::PredictCwiseOp(const OpContext& op_context,
784                                             NodeCosts* node_costs) const {
785     const auto& op_info = op_context.op_info;
786     bool found_unknown_shapes = false;
787     // For element-wise operations, op count is the element count of any input. We
788     // use the count for the largest input here to be more robust in case that the
789     // shape is unknown or partially known for other input.
790     int64_t op_count = CalculateLargestInputCount(op_info, &found_unknown_shapes);
791     // If output shape is available, try to use the element count calculated from
792     // that.
793     if (op_info.outputs_size() > 0) {
794         op_count = std::max(
795             op_count,
796             CalculateTensorElementCount(op_info.outputs(0), &found_unknown_shapes));
797     }
798     // Calculate the output shape possibly resulting from broadcasting.
799     if (op_info.inputs_size() >= 2) {
800         op_count = std::max(op_count, CwiseOutputElementCount(op_info));
801     }
802
803     int op_cost = 1;
804     auto it = elementwise_ops_.find(op_info.op());
805     if (it != elementwise_ops_.end()) {
806         op_cost = it->second;
807     } else {
808         return errors::InvalidArgument("Not a wise op: ", op_info.op());
809     }
810
811     return PredictDefaultNodeCosts(op_count * op_cost, op_context,
812                                     &found_unknown_shapes, node_costs);
813 }

```

```

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

```

```

863         (input_io_bytes > 0)
864         ? std::ceil(input_io_bytes / device_info.intermediate_read_gb_per_sec)
865         : 0;
866
867     double intermediate_write_time =
868         (output_io_bytes > 0)
869         ? std::ceil(output_io_bytes /
870                     device_info.intermediate_write_gb_per_sec)
871         : 0;
872
873     Costs::NanoSeconds intermediate_memory_cost =
874         compute_memory_overlap_
875         ? std::max(intermediate_read_time, intermediate_write_time)
876         : (intermediate_read_time + intermediate_write_time);
877     VLOG(1) << "Op:" << op_info.op() << " Size (KB):" << (total_io_bytes) / 1e3
878         << " Intermediate Memory Time (ns):"
879         << intermediate_memory_cost.count();
880
881     Costs costs = Costs::ZeroCosts();
882     costs.compute_time = compute_cost;
883     costs.memory_time = memory_cost;
884     costs.intermediate_memory_time = intermediate_memory_cost;
885     costs.intermediate_memory_read_time =
886         Costs::NanoSeconds(intermediate_read_time);
887     costs.intermediate_memory_write_time =
888         Costs::NanoSeconds(intermediate_write_time);
889     CombineCostsAndUpdateExecutionTime(compute_memory_overlap_, &costs);
890     return costs;
891 }
892
893 int64_t OpLevelCostEstimator::CountConv2D0Operations(
894     const OpInfo& op_info, bool* found_unknown_shapes) {
895     return CountConv2D0Operations(op_info, nullptr, found_unknown_shapes);
896 }
897
898 // Helper to translate the positional arguments into named fields.
899 /* static */
900 OpLevelCostEstimator::ConvolutionDimensions
901 OpLevelCostEstimator::ConvolutionDimensionsFromInputs(
902     const TensorShapeProto& original_image_shape,
903     const TensorShapeProto& original_filter_shape, const OpInfo& op_info,
904     bool* found_unknown_shapes) {
905     VLOG(2) << "op features: " << op_info.DebugString();
906     VLOG(2) << "Original image shape: " << original_image_shape.DebugString();
907     VLOG(2) << "Original filter shape: " << original_filter_shape.DebugString();
908
909     int x_index, y_index, major_channel_index, minor_channel_index = -1;
910     const std::string& data_format = GetDataFormat(op_info);
911     if (data_format == "NCHW") {

```

```

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

```

```

961     int64_t iz = minor_channel_index >= 0
962         ? image_shape.dim(minor_channel_index).size() *
963           image_shape.dim(major_channel_index).size()
964         : image_shape.dim(major_channel_index).size();
965     int64_t kx = filter_shape.dim(filter_x_index).size();
966     int64_t ky = filter_shape.dim(filter_y_index).size();
967     int64_t kz = in_minor_channel_index >= 0
968         ? filter_shape.dim(in_major_channel_index).size() *
969           filter_shape.dim(in_minor_channel_index).size()
970         : filter_shape.dim(in_major_channel_index).size();
971     std::vector<int64_t> strides = GetStrides(op_info);
972     const auto padding = GetPadding(op_info);
973     int64_t sx = strides[x_index];
974     int64_t sy = strides[y_index];
975     int64_t ox = GetOutputSize(ix, kx, sx, padding);
976     int64_t oy = GetOutputSize(iy, ky, sy, padding);
977     int64_t oz = filter_shape.dim(out_channel_index).size();
978     // Only check equality when both sizes are known (in other words, when
979     // neither is set to a minimum dimension size of 1).
980     if (iz != 1 && kz != 1) {
981         DCHECK_EQ(iz % kz, 0) << "Input channel " << iz
982             << " is not a multiple of filter channel " << kz
983             << ".";
984         if (iz % kz) {
985             *found_unknown_shapes = true;
986         }
987     } else {
988         iz = kz = std::max<int64_t>(iz, kz);
989     }
990     OpLevelCostEstimator::ConvolutionDimensions conv_dims = {
991         batch, ix, iy, iz, kx, ky, kz, oz, ox, oy, sx, sy, padding};
992
993     VLOG(1) << "Batch Size:" << batch;
994     VLOG(1) << "Image Dims:" << ix << ", " << iy;
995     VLOG(1) << "Input Depth:" << iz;
996     VLOG(1) << "Kernel Dims:" << kx << ", " << ky;
997     VLOG(1) << "Kernel Depth:" << kz;
998     VLOG(1) << "Output Dims:" << ox << ", " << oy;
999     VLOG(1) << "Output Depth:" << oz;
1000    VLOG(1) << "Strides:" << sx << ", " << sy;
1001    VLOG(1) << "Padding:" << (padding == Padding::VALID ? "VALID" : "SAME");
1002    return conv_dims;
1003 }
1004
1005 int64_t OpLevelCostEstimator::CountConv2DOperations(
1006     const OpInfo& op_info, ConvolutionDimensions* conv_info,
1007     bool* found_unknown_shapes) {
1008     DCHECK(op_info.op() == kConv2d || op_info.op() == kDepthwiseConv2dNative)
1009         << "Invalid Operation: not Conv2D nor DepthwiseConv2dNative";

```

```

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

```

```

1059     return 0;
1060 }
1061
1062 auto& a_matrix = op_info.inputs(0);
1063 auto& b_matrix = op_info.inputs(1);
1064
1065 bool transpose_a = false;
1066 bool transpose_b = false;
1067
1068 double m_dim, n_dim, k_dim, k_dim_b = 0;
1069
1070 for (const auto& item : op_info.attr()) {
1071     VLOG(1) << "Key:" << item.first
1072             << " Value:" << SummarizeAttrValue(item.second);
1073     if (item.first == "transpose_a" && item.second.b() == true)
1074         transpose_a = true;
1075     if (item.first == "transpose_b" && item.second.b() == true)
1076         transpose_b = true;
1077 }
1078 VLOG(1) << "transpose_a:" << transpose_a;
1079 VLOG(1) << "transpose_b:" << transpose_b;
1080 auto a_matrix_shape =
1081     MaybeGetMinimumShape(a_matrix.shape(), 2, found_unknown_shapes);
1082 auto b_matrix_shape =
1083     MaybeGetMinimumShape(b_matrix.shape(), 2, found_unknown_shapes);
1084 if (transpose_a) {
1085     m_dim = a_matrix_shape.dim(1).size();
1086     k_dim = a_matrix_shape.dim(0).size();
1087 } else {
1088     m_dim = a_matrix_shape.dim(0).size();
1089     k_dim = a_matrix_shape.dim(1).size();
1090 }
1091 if (transpose_b) {
1092     k_dim_b = b_matrix_shape.dim(1).size();
1093     n_dim = b_matrix_shape.dim(0).size();
1094 } else {
1095     k_dim_b = b_matrix_shape.dim(0).size();
1096     n_dim = b_matrix_shape.dim(1).size();
1097 }
1098
1099 VLOG(1) << "M, N, K: " << m_dim << "," << n_dim << "," << k_dim;
1100 // Only check equality when both sizes are known (in other words, when
1101 // neither is set to a minimum dimension size of 1).
1102 if (k_dim_b != 1 && k_dim != 1 && k_dim_b != k_dim) {
1103     LOG(ERROR) << "Incompatible Matrix dimensions";
1104     return ops;
1105 } else {
1106     // One of k_dim and k_dim_b might be 1 (minimum dimension size).
1107     k_dim = std::max(k_dim, k_dim_b);

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1402     } while (0)
1403
1404     if (tensor.dtype() == DT_INT32 || tensor.dtype() == DT_INT16 ||
1405         tensor.dtype() == DT_INT8 || tensor.dtype() == DT_UINT8) {
1406         TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(int);
1407     } else if (tensor.dtype() == DT_INT64) {
1408         TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(int64);
1409     } else if (tensor.dtype() == DT_UINT32) {
1410         TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(uint32);
1411     } else if (tensor.dtype() == DT_UINT64) {
1412         TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO(uint64);
1413     } else {
1414         LOG(WARNING) << "GetTensorShapeProtoFromTensorProto() -- "
1415             << "Unsupported dtype: " << tensor.dtype();
1416         return false;
1417     }
1418 #undef TENSOR_VALUES_TO_TENSOR_SHAPE_PROTO
1419
1420     return true;
1421 }
1422
1423 // TODO(cliffy): Dedup this method and CountConv2DBackpropFilterOperations.
1424 int64_t OplevelCostEstimator::CountConv2DBackpropInputOperations(
1425     const OpInfo& op_info, ConvolutionDimensions* returned_conv_dims,
1426     bool* found_unknown_shapes) {
1427     int64_t ops = 0;
1428
1429     DCHECK(op_info.op() == kConv2dBackpropInput ||
1430         op_info.op() == kDepthwiseConv2dNativeBackpropInput)
1431         << "Invalid Operation: not kConv2dBackpropInput nor"
1432         "kDepthwiseConv2dNativeBackpropInput";
1433
1434     if (op_info.inputs_size() < 2) {
1435         // TODO(pcma): Try to separate invalid inputs from unknown shapes
1436         *found_unknown_shapes = true;
1437         return ops;
1438     }
1439
1440     TensorShapeProto input_shape;
1441     bool shape_found = false;
1442     if (op_info.inputs(0).has_value()) {
1443         const TensorProto& value = op_info.inputs(0).value();
1444         shape_found = GetTensorShapeProtoFromTensorProto(value, &input_shape);
1445     }
1446     if (!shape_found && op_info.outputs_size() == 1) {
1447         input_shape = op_info.outputs(0).shape();
1448         shape_found = true;
1449     }
1450     if (!shape_found) {

```

```

1451     // Set the minimum filter size that's feasible.
1452     input_shape.Clear();
1453     for (int i = 0; i < 4; ++i) {
1454         input_shape.add_dim()->set_size(1);
1455     }
1456     *found_unknown_shapes = true;
1457 }
1458
1459 ConvolutionDimensions conv_dims = ConvolutionDimensionsFromInputs(
1460     input_shape, op_info.inputs(1).shape(), op_info, found_unknown_shapes);
1461
1462 ops = conv_dims.batch;
1463 ops *= conv_dims.ox * conv_dims.oy;
1464 ops *= conv_dims.kx * conv_dims.ky;
1465 if (op_info.op() == kConv2dBackpropInput) {
1466     ops *= conv_dims.kz * conv_dims.oz;
1467 } else {
1468     // conv_dims always use forward path definition regardless
1469     conv_dims.oz *= conv_dims.iz;
1470     ops *= conv_dims.oz;
1471 }
1472 ops *= kOpsPerMac;
1473
1474 VLOG(1) << "Operations for" << op_info.op() << " " << ops;
1475
1476 if (returned_conv_dims != nullptr) {
1477     *returned_conv_dims = conv_dims;
1478 }
1479 return ops;
1480 }
1481
1482 int64_t OpLevelCostEstimator::CountConv2DBackpropFilterOperations(
1483     const OpInfo& op_info, ConvolutionDimensions* returned_conv_dims,
1484     bool* found_unknown_shapes) {
1485     int64_t ops = 0;
1486
1487     DCHECK(op_info.op() == kConv2dBackpropFilter ||
1488         op_info.op() == kDepthwiseConv2dNativeBackpropFilter)
1489         << "Invalid Operation: not kConv2dBackpropFilter nor"
1490         "kDepthwiseConv2dNativeBackpropFilter";
1491
1492     TensorShapeProto filter_shape;
1493     bool shape_found = false;
1494     if (op_info.inputs_size() >= 2 && op_info.inputs(1).has_value()) {
1495         const TensorProto& value = op_info.inputs(1).value();
1496         shape_found = GetTensorShapeProtoFromTensorProto(value, &filter_shape);
1497     }
1498     if (!shape_found && op_info.outputs_size() == 1) {
1499         filter_shape = op_info.outputs(0).shape();

```

```

1500     shape_found = true;
1501 }
1502 if (!shape_found) {
1503     // Set the minimum filter size that's feasible.
1504     filter_shape.Clear();
1505     for (int i = 0; i < 4; ++i) {
1506         filter_shape.add_dim()->set_size(1);
1507     }
1508     *found_unknown_shapes = true;
1509 }
1510
1511 if (op_info.inputs_size() < 1) {
1512     // TODO(pcma): Try to separate invalid inputs from unknown shapes
1513     *found_unknown_shapes = true;
1514     return ops;
1515 }
1516 ConvolutionDimensions conv_dims = ConvolutionDimensionsFromInputs(
1517     op_info.inputs(0).shape(), filter_shape, op_info, found_unknown_shapes);
1518
1519 ops = conv_dims.batch;
1520 ops *= conv_dims.ox * conv_dims.oy;
1521 ops *= conv_dims.kx * conv_dims.ky;
1522 if (op_info.op() == kConv2dBackpropFilter) {
1523     ops *= conv_dims.kz * conv_dims.oz;
1524 } else {
1525     // conv_dims always use forward path definition regardless
1526     conv_dims.oz *= conv_dims.iz;
1527     ops *= conv_dims.oz;
1528 }
1529 ops *= kOpsPerMac;
1530 VLOG(1) << "Operations for" << op_info.op() << " " << ops;
1531
1532 if (returned_conv_dims != nullptr) {
1533     *returned_conv_dims = conv_dims;
1534 }
1535 return ops;
1536 }
1537
1538 int64_t OpLevelCostEstimator::CalculateTensorElementCount(
1539     const OpInfo::TensorProperties& tensor, bool* found_unknown_shapes) {
1540     VLOG(2) << "    with " << DataTypeString(tensor.dtype()) << " tensor of shape "
1541         << tensor.shape().DebugString();
1542     int64_t tensor_size = 1;
1543     int num_dims = std::max(1, tensor.shape().dim_size());
1544     auto tensor_shape =
1545         MaybeGetMinimumShape(tensor.shape(), num_dims, found_unknown_shapes);
1546     for (const auto& dim : tensor_shape.dim()) {
1547         tensor_size *= dim.size();
1548     }

```



```

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

```

```

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

```

```

1647     CalculateOutputTensorSize(op_info, found_unknown_shapes);
1648     node_costs->max_memory = node_costs->num_total_output_bytes();
1649     if (*found_unknown_shapes) {
1650         node_costs->inaccurate = true;
1651         node_costs->num_nodes_with_unknown_shapes = 1;
1652     }
1653     return Status::OK();
1654 }
1655
1656 bool HasZeroDim(const OpInfo& op_info) {
1657     for (int i = 0; i < op_info.inputs_size(); ++i) {
1658         const auto& input = op_info.inputs(i);
1659         for (int j = 0; j < input.shape().dim_size(); ++j) {
1660             const auto& dim = input.shape().dim(j);
1661             if (dim.size() == 0) {
1662                 VLOG(1) << "Convolution config has zero dim "
1663                     << op_info.ShortDebugString();
1664                 return true;
1665             }
1666         }
1667     }
1668     return false;
1669 }
1670
1671 Status OpLevelCostEstimator::PredictConv2D(const OpContext& op_context,
1672                                             NodeCosts* node_costs) const {
1673     const auto& op_info = op_context.op_info;
1674     if (HasZeroDim(op_info)) {
1675         node_costs->num_nodes_with_unknown_shapes = 1;
1676         return errors::InvalidArgument("Conv2D op includes zero dimension: ",
1677                                         op_info.ShortDebugString());
1678     }
1679     bool found_unknown_shapes = false;
1680     int64_t num_compute_ops =
1681         CountConv2DOperations(op_info, &found_unknown_shapes);
1682     return PredictDefaultNodeCosts(num_compute_ops, op_context,
1683                                     &found_unknown_shapes, node_costs);
1684 }
1685
1686 Status OpLevelCostEstimator::PredictConv2DBackpropInput(
1687     const OpContext& op_context, NodeCosts* node_costs) const {
1688     const auto& op_info = op_context.op_info;
1689     if (HasZeroDim(op_info)) {
1690         node_costs->num_nodes_with_unknown_shapes = 1;
1691         return errors::InvalidArgument(
1692             "Conv2DBackpropInput op includes zero dimension",
1693             op_info.ShortDebugString());
1694     }
1695     bool found_unknown_shapes = false;

```

```

1696     int64_t num_compute_ops = CountConv2DBackpropInputOperations(
1697         op_info, nullptr, &found_unknown_shapes);
1698     return PredictDefaultNodeCosts(num_compute_ops, op_context,
1699         &found_unknown_shapes, node_costs);
1700 }
1701
1702 Status OpLevelCostEstimator::PredictConv2DBackpropFilter(
1703     const OpContext& op_context, NodeCosts* node_costs) const {
1704     const auto& op_info = op_context.op_info;
1705     if (HasZeroDim(op_info)) {
1706         node_costs->num_nodes_with_unknown_shapes = 1;
1707         return errors::InvalidArgument(
1708             "Conv2DBackpropFilter op includes zero dimension",
1709             op_info.ShortDebugString());
1710     }
1711     bool found_unknown_shapes = false;
1712     int64_t num_compute_ops = CountConv2DBackpropFilterOperations(
1713         op_info, nullptr, &found_unknown_shapes);
1714     return PredictDefaultNodeCosts(num_compute_ops, op_context,
1715         &found_unknown_shapes, node_costs);
1716 }
1717
1718 Status OpLevelCostEstimator::PredictFusedConv2DBiasActivation(
1719     const OpContext& op_context, NodeCosts* node_costs) const {
1720     // FusedConv2DBiasActivation computes a fused kernel which implements:
1721     // 2D convolution, adds side input with separate scaling on convolution and
1722     // side inputs, then adds bias, and finally applies the ReLU activation
1723     // function to the result:
1724     //
1725     // Input -> Conv2D -> Add -> BiasAdd -> ReLU
1726     //           ^         ^         ^
1727     //         Filter  Side Input  Bias
1728     //
1729     // Note that when adding the side input, the operation multiplies the output
1730     // of Conv2D by conv_input_scale, confusingly, and the side_input by
1731     // side_input_scale.
1732     //
1733     // Note that in the special case that side_input_scale is 0, which we infer
1734     // from side_input having dimensions [], we skip that addition operation.
1735     //
1736     // For more information, see
1737     // contrib/fused_conv/kernels/fused_conv2d_bias_activation_op.cc
1738
1739     // TODO(yaozhang): Support NHWC_VECT_W.
1740     std::string data_format = GetDataFormat(op_context.op_info);
1741     if (data_format != "NCHW" && data_format != "NHWC" &&
1742         data_format != "NCHW_VECT_C") {
1743         return errors::InvalidArgument(
1744             "Unsupported data format (" , data_format,

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

1990 // It's equal to the output size; though, input access is indexed gather or
1991 // slice (ignore duplicate indices).
1992 node_costs->num_input_bytes_accessed.push_back(input_size);
1993 int begin_input_index = 1;
1994 int end_input_index;
1995 if (op_info.op() == "Slice") {
1996     // Slice: 'input' (omitted), 'begin', 'size'
1997     end_input_index = 3;
1998 } else if (op_info.op() == "StridedSlice") {
1999     // StridedSlice: 'input' (omitted), 'begin', 'end', 'strides'
2000     end_input_index = 4;
2001 } else {
2002     // Gather, GatherV2, GatherNd: 'params' (omitted), 'indices'
2003     end_input_index = 2;
2004 }
2005 for (int i = begin_input_index; i < end_input_index; ++i) {
2006     node_costs->num_input_bytes_accessed.push_back(
2007         CalculateTensorElementCount(op_info.inputs(i), &unknown_shapes));
2008 }
2009 if (unknown_shapes) {
2010     node_costs->inaccurate = true;
2011     node_costs->num_nodes_with_unknown_shapes = 1;
2012 }
2013 return Status::OK();
2014 }
2015
2016 Status OpLevelCostEstimator::PredictScatter(const OpContext& op_context,
2017                                             NodeCosts* node_costs) const {
2018     // Scatter ops sparsely access a reference input and output tensor.
2019     const auto& op_info = op_context.op_info;
2020     bool found_unknown_shapes = false;
2021
2022     // input[0]: ref tensor that will be sparsely accessed
2023     // input[1]: indices - A tensor of indices into the first dimension of ref.
2024     // input[2]: updates where updates.shape = indices.shape + ref.shape[1:]
2025     // See
2026     // https://www.tensorflow.org/api_docs/python/tf/scatter_add and
2027     // https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/state_ops.cc#L146
2028
2029     const int64_t num_indices =
2030         CalculateTensorElementCount(op_info.inputs(1), &found_unknown_shapes);
2031
2032     int64_t num_elems_in_ref_per_index = 1;
2033     auto ref_tensor_shape = MaybeGetMinimumShape(
2034         op_info.inputs(0).shape(), op_info.inputs(0).shape().dim_size(),
2035         &found_unknown_shapes);
2036     for (int i = 1; i < ref_tensor_shape.dim().size(); ++i) {
2037         num_elems_in_ref_per_index *= ref_tensor_shape.dim(i).size();
2038     }

```

```

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

```

```

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

```

```

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

```

```

2186     const auto& op_info = op_context.op_info;
2187     // x: op_info.inputs(0)
2188     ConvolutionDimensions dims = OpDimensionsFromInputs(
2189         op_info.inputs(0).shape(), op_info, &found_unknown_shapes);
2190     // kx * ky - 1 comparisons per output (kx * ky > 1)
2191     // or 1 copy per output (kx * ky == 1).
2192     int per_output_ops = dims.kx * dims.ky == 1 ? 1 : dims.kx * dims.ky - 1;
2193     int64_t ops = dims.batch * dims.ox * dims.oy * dims.oz * per_output_ops;
2194     node_costs->num_compute_ops = ops;
2195
2196     int64_t input_size = 0;
2197     if (dims.ky >= dims.sy) {
2198         input_size = CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
2199     } else { // dims.ky < dims.sy
2200         // Vertical stride is larger than vertical kernel; assuming row-major
2201         // format, skip unnecessary rows (or read every kx rows per sy rows, as the
2202         // others are not used for output).
2203         const auto data_size = DataTypeSize(BaseType(op_info.inputs(0).dtype()));
2204         input_size = data_size * dims.batch * dims.ix * dims.ky * dims.oy * dims.iz;
2205     }
2206     node_costs->num_input_bytes_accessed = {input_size};
2207     const int64_t output_size =
2208         CalculateOutputSize(op_info, &found_unknown_shapes);
2209     node_costs->num_output_bytes_accessed = {output_size};
2210     node_costs->max_memory = output_size;
2211     if (found_unknown_shapes) {
2212         node_costs->inaccurate = true;
2213         node_costs->num_nodes_with_unknown_shapes = 1;
2214     }
2215     return Status::OK();
2216 }
2217
2218 Status OpLevelCostEstimator::PredictMaxPoolGrad(const OpContext& op_context,
2219                                                  NodeCosts* node_costs) const {
2220     bool found_unknown_shapes = false;
2221     const auto& op_info = op_context.op_info;
2222     // x: op_info.inputs(0)
2223     // y: op_info.inputs(1)
2224     // y_grad: op_info.inputs(2)
2225     if (op_info.inputs_size() < 3) {
2226         return errors::InvalidArgument("MaxPoolGrad op has invalid inputs: ",
2227                                         op_info.ShortDebugString());
2228     }
2229
2230     ConvolutionDimensions dims = OpDimensionsFromInputs(
2231         op_info.inputs(0).shape(), op_info, &found_unknown_shapes);
2232
2233     int64_t ops = 0;
2234     if (dims.kx == 1 && dims.ky == 1) {

```

```

2235     // 1x1 window. No need to know which input was max.
2236     ops = dims.batch * dims.ix * dims.iy * dims.iz;
2237 } else if (dims.kx <= dims.sx && dims.ky <= dims.sy) {
2238     // Non-overlapping window: re-run maxpool, then assign zero or y_grad.
2239     ops = dims.batch * dims.iz *
2240         (dims.ox * dims.oy * (dims.kx * dims.ky - 1) + dims.ix * dims.iy);
2241 } else {
2242     // Overlapping window: initialize with zeros, re-run maxpool, then
2243     // accumulate y_grad to proper x_grad locations.
2244     ops = dims.batch * dims.iz *
2245         (dims.ox * dims.oy * (dims.kx * dims.ky - 1) + dims.ix * dims.iy * 2);
2246 }
2247 node_costs->num_compute_ops = ops;
2248
2249 // Just read x and y_grad; no need to read y as we assume MaxPoolGrad re-run
2250 // MaxPool internally.
2251 const int64_t input0_size =
2252     CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
2253 const int64_t input2_size =
2254     CalculateTensorSize(op_info.inputs(2), &found_unknown_shapes);
2255 node_costs->num_input_bytes_accessed = {input0_size, 0, input2_size};
2256 // Write x_grad; size equal to x.
2257 const int64_t output_size =
2258     CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
2259 node_costs->num_output_bytes_accessed = {output_size};
2260 node_costs->max_memory = output_size;
2261
2262 if (found_unknown_shapes) {
2263     node_costs->inaccurate = true;
2264     node_costs->num_nodes_with_unknown_shapes = 1;
2265 }
2266 return Status::OK();
2267 }
2268
2269 /* This predict function handles three types of tensorflow ops
2270 * AssignVariableOp/AssignAddVariableOp/AssignSubVariableOp, broadcasting
2271 * was not possible for these ops, therefore the input tensor's shapes is
2272 * enough to compute the cost */
2273 Status OpLevelCostEstimator::PredictAssignVariableOps(
2274     const OpContext& op_context, NodeCosts* node_costs) const {
2275     bool found_unknown_shapes = false;
2276     const auto& op_info = op_context.op_info;
2277     /* First input of these ops are reference to the assignee. */
2278     if (op_info.inputs_size() != 2) {
2279         return errors::InvalidArgument("AssignVariable op has invalid input: ",
2280             op_info.ShortDebugString());
2281     }
2282
2283     const int64_t ops = op_info.op() == kAssignVariableOp

```

```

2284         ? 0
2285         : CalculateTensorElementCount(op_info.inputs(1),
2286                                     &found_unknown_shapes);
2287     node_costs->num_compute_ops = ops;
2288     const int64_t input_size = CalculateInputSize(op_info, &found_unknown_shapes);
2289     node_costs->num_input_bytes_accessed = {input_size};
2290     // TODO(dyoon): check these ops' behavior whether it writes data;
2291     // Op itself doesn't have output tensor, but it may modify the input (ref or
2292     // resource). Maybe use node_costs->internal_write_bytes.
2293     node_costs->num_output_bytes_accessed = {0};
2294     if (found_unknown_shapes) {
2295         node_costs->inaccurate = true;
2296         node_costs->num_nodes_with_unknown_shapes = 1;
2297     }
2298     return Status::OK();
2299 }
2300
2301 Status OpLevelCostEstimator::PredictAvgPool(const OpContext& op_context,
2302                                             NodeCosts* node_costs) const {
2303     bool found_unknown_shapes = false;
2304     const auto& op_info = op_context.op_info;
2305     // x: op_info.inputs(0)
2306     ConvolutionDimensions dims = OpDimensionsFromInputs(
2307         op_info.inputs(0).shape(), op_info, &found_unknown_shapes);
2308
2309     // kx * ky - 1 additions and 1 multiplication per output.
2310     int64_t ops = dims.batch * dims.ox * dims.oy * dims.oz * dims.kx * dims.ky;
2311     node_costs->num_compute_ops = ops;
2312
2313     int64_t input_size;
2314     if (dims.ky >= dims.sy) {
2315         input_size = CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
2316     } else { // dims.ky < dims.sy
2317         // vertical stride is larger than vertical kernel; assuming row-major
2318         // format, skip unnecessary rows (or read every kx rows per sy rows, as the
2319         // others are not used for output).
2320         const auto data_size = DataTypeSize(BaseType(op_info.inputs(0).dtype()));
2321         input_size = data_size * dims.batch * dims.ix * dims.ky * dims.oy * dims.iz;
2322     }
2323     node_costs->num_input_bytes_accessed = {input_size};
2324
2325     const int64_t output_size =
2326         CalculateOutputSize(op_info, &found_unknown_shapes);
2327     node_costs->num_output_bytes_accessed = {output_size};
2328     node_costs->max_memory = output_size;
2329
2330     if (found_unknown_shapes) {
2331         node_costs->inaccurate = true;
2332         node_costs->num_nodes_with_unknown_shapes = 1;

```



```

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

```

```

2382 Status OpLevelCostEstimator::PredictFusedBatchNorm(
2383     const OpContext& op_context, NodeCosts* node_costs) const {
2384     bool found_unknown_shapes = false;
2385     const auto& op_info = op_context.op_info;
2386     // x: op_info.inputs(0)
2387     // scale: op_info.inputs(1)
2388     // offset: op_info.inputs(2)
2389     // mean: op_info.inputs(3) --> only for inference
2390     // variance: op_info.inputs(4) --> only for inference
2391     ConvolutionDimensions dims = OpDimensionsFromInputs(
2392         op_info.inputs(0).shape(), op_info, &found_unknown_shapes);
2393     const bool is_training = IsTraining(op_info);
2394
2395     int64_t ops = 0;
2396     const auto rsqrt_cost = Eigen::internal::functor_traits<
2397         Eigen::internal::scalar_rsqrt_op<float>>::Cost;
2398     if (is_training) {
2399         ops = dims.iz * (dims.batch * dims.ix * dims.iy * 4 + 6 + rsqrt_cost);
2400     } else {
2401         ops = dims.batch * dims.ix * dims.iy * dims.iz * 2;
2402     }
2403     node_costs->num_compute_ops = ops;
2404
2405     const int64_t size_nhwc =
2406         CalculateTensorSize(op_info.inputs(0), &found_unknown_shapes);
2407     const int64_t size_c =
2408         CalculateTensorSize(op_info.inputs(1), &found_unknown_shapes);
2409     if (is_training) {
2410         node_costs->num_input_bytes_accessed = {size_nhwc, size_c, size_c};
2411         node_costs->num_output_bytes_accessed = {size_nhwc, size_c, size_c, size_c,
2412             size_c};
2413         // FusedBatchNorm in training mode internally re-reads the input tensor:
2414         // one for mean/variance, and the 2nd internal read for the actual scaling.
2415         // Assume small intermediate data such as mean / variance (size_c) can be
2416         // cached on-chip.
2417         node_costs->internal_read_bytes = size_nhwc;
2418     } else {
2419         node_costs->num_input_bytes_accessed = {size_nhwc, size_c, size_c, size_c,
2420             size_c};
2421         node_costs->num_output_bytes_accessed = {size_nhwc};
2422     }
2423     node_costs->max_memory = node_costs->num_total_output_bytes();
2424
2425     if (found_unknown_shapes) {
2426         node_costs->inaccurate = true;
2427         node_costs->num_nodes_with_unknown_shapes = 1;
2428     }
2429     return Status::OK();
2430 }

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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