jpienaar Add default feedback in preparation for removal ... ✓ History

9 contributors

791 lines (736 sloc) | 30.7 KB ...

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

```
30  #include "tensorflow/core/lib/core/errors.h"
31  #include "tensorflow/core/lib/core/stringpiece.h"
32  #include "tensorflow/core/lib/gtl/inlined_vector.h"
33  #include "tensorflow/core/lib/strings/str_util.h"
34  #include "tensorflow/core/lib/strings/strcat.h"
35  #include "tensorflow/core/util/device_name_utils.h"
36
37  namespace tensorflow {
38  namespace grappler {
39
40  namespace {
41
42  bool RemoveControlInput(NodeDef* node, const string& control_input_to_remove,
43                          NodeMap* node_map) {
44    for (int pos = node->input_size() - 1; pos >= 0; --pos) {
45      const string& input = node->input(pos);
46      if (input[0] != '^') break;
47      if (input == control_input_to_remove) {
48        node->mutable_input()->SwapElements(pos, node->input_size() - 1);
49        node->mutable_input()->RemoveLast();
50        node_map->RemoveOutput(NodeName(input), node->name());
51        return true;
52      }
53    }
54    return false;
55  }
56
57  }  // namespace
58
59  bool DependencyOptimizer::SafeToRemoveIdentity(const NodeDef& node) const {
60    if (!IsIdentity(node) && !IsIdentityN(node)) {
61      return true;
62    }
63
64    if (nodes_to_preserve_.find(node.name()) != nodes_to_preserve_.end()) {
65      return false;
66    }
67    if (!fetch_nodes_known_) {
68      // The output values of this node may be needed.
69      return false;
70    }
71
72    if (node.input_size() < 1) {
73      // Node lacks input, is invalid
74      return false;
75    }
76
77    const NodeDef* input = node_map_->GetNode(NodeName(node.input(0)));
78    CHECK(input != nullptr) << "node = " << node.name()
```

```
                          << " input = " << node.input(0);
  // Don't remove Identity nodes corresponding to Variable reads or following
  // Recv.
  if (IsVariable(*input) || IsRecv(*input)) {
    return false;
  }
  for (const auto& consumer : node_map_->GetOutputs(node.name())) {
    if (node.input_size() > 1 && (IsRetval(*consumer) || IsMerge(*consumer))) {
      return false;
    }
    if (IsSwitch(*input)) {
      for (const string& consumer_input : consumer->input()) {
        if (consumer_input == AsControlDependency(node.name())) {
          return false;
        }
      }
    }
  }
  return true;
}

bool DependencyOptimizer::SafeToConvertToNoOp(const NodeDef& node) const {
  if (HasRegularOutputs(node, *node_map_)) {
    // The output values of this node may be needed.
    VLOG(3) << "Not safe to convert '" << node.name()
            << " to NoOp. Node has outputs.";
    return false;
  }
  if (!fetch_nodes_known_) {
    VLOG(3) << "Not safe to convert '" << node.name()
            << " to NoOp. Fetches unknown.";
    return false;
  }
  if (nodes_to_preserve_.find(node.name()) != nodes_to_preserve_.end()) {
    VLOG(3) << "Not safe to convert to NoOp: " << node.name()
            << " is in preserve set.";
    return false;
  }
  if (IsMerge(node) || IsSwitch(node) || ModifiesFrameInfo(node)) {
    VLOG(3) << "Not safe to convert '" << node.name()
            << " to NoOp. Node modifies frame info.";
    return false;
  }
  // Ops reading variables are marked as stateful, but are safe to remove if
  // redundant.
  static const absl::flat_hash_set<string>* gather_ops =
      new absl::flat_hash_set<string>{"Gather", "GatherV2", "GatherNd",
                                      "ResourceGather", "ResourceGatherNd"};
  const bool is_variable_read =
```

```
      IsReadVariableOp(node) || IsReadVariablesOp(node) ||
      gather_ops->find(node.op()) != gather_ops->end();
  if (!is_variable_read && !IsFreeOfSideEffect(node)) {
    VLOG(3) << "Not safe to convert '" << node.name()
            << " to NoOp. Node has side effect.";
    return false;
  }
  if (node.op().rfind("Submodel", 0) == 0) {
    return false;
  }
  const OpDef* op_def = nullptr;
  Status status = OpRegistry::Global()->LookUpOpDef(node.op(), &op_def);
  if (!status.ok() || op_def->output_arg_size() == 0) {
    return false;
  }
  const std::unordered_set<string> do_not_rewrite_ops{
      "Assert",     "CheckNumerics",         "_Retval",
      "_Arg",       "_ParallelConcatUpdate", "TPUExecute",
      "TPUCompile", "ControlTrigger"};
  if (do_not_rewrite_ops.find(node.op()) != do_not_rewrite_ops.end()) {
    return false;
  }
  if (!SafeToRemoveIdentity(node)) {
    return false;
  }
  return true;
}

int DependencyOptimizer::NumEdgesIfBypassed(
    const NodeDef& node, const std::vector<NodeDef*>& output_nodes) const {
  const bool is_multi_input_identity_n =
      IsIdentityN(node) && !IsIdentityNSingleInput(node);
  const int num_outputs = output_nodes.size();
  const int num_inputs = node.input_size();

  if (is_multi_input_identity_n) {
    // multi-input identity_n with input/output control dependencies will likely
    // increase number of edges after optimization.
    int num_edges_if_bypassed(0);
    for (const string& input_node_name : node.input()) {
      if (IsControlInput(input_node_name)) {
        num_edges_if_bypassed += num_outputs;
      } else {
        ++num_edges_if_bypassed;
      }
    }

    for (auto consumer : output_nodes) {
      for (int j = 0; j < consumer->input_size(); ++j) {
```

```
177         const TensorId consumer_input = ParseTensorName(consumer->input(j));
178         if (consumer_input.node() == node.name()) {
179           if (IsControlInput(consumer_input)) {
180             num_edges_if_bypassed += num_inputs;
181           } else {
182             ++num_edges_if_bypassed;
183           }
184         }
185       }
186     }
187     return num_edges_if_bypassed;
188   } else {
189     return num_inputs * num_outputs;
190   }
191 }
192
193 bool DependencyOptimizer::BypassingNodeIsBeneficial(
194     const NodeDef& node, const std::vector<NodeDef*>& input_nodes,
195     const std::vector<NodeDef*>& output_nodes) const {
196   const bool is_identity = IsIdentity(node) || IsIdentityNSingleInput(node);
197   const bool is_multi_input_identity_n =
198       IsIdentityN(node) && !IsIdentityNSingleInput(node);
199   const int num_outputs = output_nodes.size();
200   const int num_inputs = node.input_size();
201
202   if (NumEdgesIfBypassed(node, output_nodes) > num_inputs + num_outputs) {
203     return false;
204   }
205
206   // Make sure that we don't increase the number of edges that cross
207   // device boundaries.
208   if ((num_inputs == 1 && num_outputs > 1 &&
209        input_nodes[0]->device() != node.device()) ||
210       (num_inputs > 1 && num_outputs == 1 &&
211        output_nodes[0]->device() != node.device())) {
212     return false;
213   }
214
215   // TODO(rmlarsen): Not all device crossings are equally expensive.
216   // Assign a cost to each based on device affinity and compute a
217   // cost before and after.
218   const string& node_dev = node.device();
219   int num_cross_in = 0;
220   for (NodeDef* input_node : input_nodes) {
221     num_cross_in += static_cast<int>(input_node->device() != node_dev);
222   }
223   int num_cross_out = 0;
224   for (NodeDef* output_node : output_nodes) {
225     num_cross_out += static_cast<int>(output_node->device() != node_dev);
```

```
226     }
227
228     // Make sure we do not increase the number of device crossings.
229     const int num_cross_before = num_cross_in + num_cross_out;
230     int num_cross_after = 0;
231     for (NodeDef* input_node : input_nodes) {
232       for (NodeDef* output_node : output_nodes) {
233         num_cross_after +=
234             static_cast<int>(input_node->device() != output_node->device());
235       }
236     }
237     if (num_cross_after > num_cross_before) {
238       return false;
239     }
240
241     if ((is_identity || is_multi_input_identity_n) && num_cross_in > 0 &&
242         num_cross_out > 0 && num_cross_after > 0) {
243       // This identity node follows a device crossing, so it might be
244       // following a _Recv node after partitioning. Do not remove such nodes,
245       // unless they only have consumers on the same device as themselves.
246       return false;
247     }
248
249     return true;
250   }
251
252   void DependencyOptimizer::OptimizeNode(int node_idx,
253                                          SetVector<int>* nodes_to_simplify,
254                                          std::set<int>* nodes_to_delete) {
255     NodeDef* node = optimized_graph_->mutable_node(node_idx);
256     const bool is_noop = IsNoOp(*node);
257     const bool is_identity = IsIdentity(*node) || IsIdentityNSingleInput(*node);
258     const bool is_multi_input_identity =
259         IsIdentityN(*node) && !IsIdentityNSingleInput(*node);
260     const string node_name = node->name();
261     // Constant nodes with no input control dependency are always executed early,
262     // so we can prune all their output control dependencies.
263     if (IsConstant(*node) && node->input_size() == 0) {
264       const auto output_nodes = node_map_->GetOutputs(node_name);
265       for (NodeDef* fanout : output_nodes) {
266         bool optimize_fanout = false;
267         bool data_connection = false;
268         for (int i = fanout->input_size() - 1; i >= 0; --i) {
269           const TensorId input_tensor = ParseTensorName(fanout->input(i));
270           if (input_tensor.node() == node_name) {
271             if (input_tensor.index() < 0) {
272               fanout->mutable_input()->SwapElements(i, fanout->input_size() - 1);
273               fanout->mutable_input()->RemoveLast();
274               optimize_fanout = true;
```

```
275           } else {
276             data_connection = true;
277           }
278         }
279       }
280       if (optimize_fanout) {
281         nodes_to_simplify->PushBack(node_to_idx_[fanout]);
282         if (!data_connection) {
283           node_map_->RemoveOutput(node_name, fanout->name());
284         }
285       }
286     }
287     if (node_map_->GetOutputs(node_name).empty() && fetch_nodes_known_ &&
288         nodes_to_preserve_.find(node_name) == nodes_to_preserve_.end()) {
289       // Mark the node for deletion.
290       nodes_to_delete->insert(node_to_idx_[node]);
291     }
292     return;
293   }
294
295   // Change ops that only have control dependencies as outputs to NoOps.
296   if (!is_noop && SafeToConvertToNoOp(*node)) {
297     VLOG(2) << "***** Replacing  " << node_name << " (" << node->op()
298             << ") with NoOp.";
299     // The outputs of this node are not consumed. Replace its inputs with
300     // control dependencies and replace the op itself with the NoOp op.
301     std::unordered_set<string> ctrl_inputs;
302     int pos = 0;
303     while (pos < node->input_size()) {
304       const string old_input = node->input(pos);
305       if (IsControlInput(old_input)) {
306         if (!ctrl_inputs.insert(old_input).second) {
307           // We found a duplicate control input. Remove it.
308           node->mutable_input()->SwapElements(pos, node->input_size() - 1);
309           node->mutable_input()->RemoveLast();
310         } else {
311           ++pos;
312         }
313         continue;
314       }
315       // Replace a normal input with a control input.
316       const string ctrl_input = ConstantFolding::AddControlDependency(
317           old_input, optimized_graph_, node_map_.get());
318       ctrl_inputs.insert(ctrl_input);
319       node->set_input(pos, ctrl_input);
320       node_map_->UpdateInput(node_name, old_input, ctrl_input);
321       const NodeDef* old_input_node = node_map_->GetNode(old_input);
322       nodes_to_simplify->PushBack(node_to_idx_[old_input_node]);
323       ++pos;
```

```
    }
    node->set_op("NoOp");
    EraseRegularNodeAttributes(node);
    DedupControlInputs(node);
    nodes_to_simplify->PushBack(node_to_idx_[node]);
    return;
  }

  // Remove NoOp nodes if the product of their fan-in and fan-out is less than
  // or equal to the sum of the fan-in and fan-out. The non-trivial rewrites
  // take the following form:
  //
  // Case a)
  //    x --^> +------+               x --^> +---+
  //    y --^> | NoOp | --^> a   ==>   y --^> | a |
  //    ...    |      |                 ...  |   |
  //    z --^> +------+               z --^> +---+
  //
  // Case b)
  //           +------+ --^> a         +---+ --^> a
  //    x --^> | NoOp | --^> b  ==>    | x | --^> b
  //           |      | ...            |   | ...
  //           +------+ --^> c         +---+ --^> c
  // Case c)
  //           +------+               x ---^> a
  //    x --^> | NoOp | --^> a  ==>      \/
  //    y --^> |      | --^> b           /\
  //           +------+               y ---^> b
  //
  // We only apply this optimization if we don't increase the number of control
  // edges across device boundaries, e.g. in cases a) and b) if NoOp and
  // a and x, respectively, are on the same device. Control edges across device
  // boundaries require inter-device communication (Send/Recv pairs to be
  // inserted in the graph), which is very costly.
  //
  // We also remove identity nodes, subject to the same constraints on number of
  // resulting control edges and device boundary crossings:
  //
  // Case a)
  //          +----------+ ---> a       +---+ ---> a
  //    x --> | Identity | --^> b  ==>  | x | --^> b
  //          |          | ...          |   | ...
  //          +----------+ --^> c       +---+ --^> c
  //
  // Case b)
  //    x ---> +----------+ ---> a      x ---> +---+
  //    y --^> | Identity |        ==>  y --^> | a |
  //    ...    |          |               ...  |   |
  //    z --^> +----------+             z --^> +---+
```

```
373   //
374   // Case c)
375   //           +----------+             x ---> +---+
376   //    x ---> | Identity | ---> a ==>   \--^> | a |
377   //    y --^> |          | --^> b       /\    +---+
378   //           +----------+             y --^> b
379
380   if (is_noop || ((is_identity || is_multi_input_identity) &&
381                   SafeToRemoveIdentity(*node))) {
382     const int num_inputs = node->input_size();
383     std::vector<NodeDef*> input_nodes;
384     for (int i = 0; i < num_inputs; ++i) {
385       NodeDef* input_node = node_map_->GetNode(node->input(i));
386       if (input_node == nullptr) {
387         LOG(ERROR) << "Invalid input " << node->input(i);
388         return;
389       }
390       input_nodes.push_back(input_node);
391     }
392     const auto& output_node_set = node_map_->GetOutputs(node_name);
393     const std::vector<NodeDef*> output_nodes(output_node_set.begin(),
394                                              output_node_set.end());
395
396     if (!BypassingNodeIsBeneficial(*node, input_nodes, output_nodes)) {
397       return;
398     }
399
400     VLOG(2) << "***** Rerouting input around\n" << node->DebugString();
401     // Now remove the node and re-wire its inputs to its outputs.
402     for (auto consumer : output_nodes) {
403       bool updated_consumer = false;
404       VLOG(2) << "consumer before:\n" << consumer->DebugString();
405       // Remove dependency on node from consumer.
406       for (int i = 0; i < num_inputs; ++i) {
407         const NodeDef* input = input_nodes[i];
408         // Forward dependency from input to consumer if it doesn't already
409         // depend on it.
410         if ((is_identity && i == 0) ||
411             (is_multi_input_identity && !IsControlInput(node->input(i)))) {
412           // Replace regular input from Identity node.
413           string new_input;
414           const string& input_to_forward = node->input(i);
415           CHECK(!IsControlInput(input_to_forward));
416           for (int j = 0; j < consumer->input_size(); ++j) {
417             const TensorId old_input = ParseTensorName(consumer->input(j));
418             if (old_input.node() == node_name) {
419               if (old_input.index() == i) {
420                 // Regular input
421                 new_input = input_to_forward;
```

```
422                 node_map_->UpdateInput(consumer->name(),
423                                        string(old_input.node()), new_input);
424                 consumer->set_input(j, new_input);
425               } else if (old_input.index() == -1) {
426                 // Control dependency
427                 new_input = AsControlDependency(NodeName(input_to_forward));
428                 node_map_->UpdateInput(consumer->name(),
429                                        string(old_input.node()), new_input);
430                 consumer->set_input(j, new_input);
431               }
432             }
433           }
434           updated_consumer = true;
435         } else {
436           // Forward dependency from input to consumer if it doesn't already
437           // depend on it.
438           if (node_map_->GetOutputs(input->name()).count(consumer) == 0) {
439             consumer->add_input(AsControlDependency(input->name()));
440             node_map_->AddOutput(input->name(), consumer->name());
441             nodes_to_simplify->PushBack(node_to_idx_[input]);
442             updated_consumer = true;
443           }
444         }
445       }
446       updated_consumer |= RemoveControlInput(
447           consumer, AsControlDependency(node_name), node_map_.get());
448       if (updated_consumer) {
449         nodes_to_simplify->PushBack(node_to_idx_[consumer]);
450       }
451       VLOG(2) << "consumer after:\n" << consumer->DebugString();
452     }
453     node_map_->RemoveOutputs(node_name);
454     if (fetch_nodes_known_ &&
455         nodes_to_preserve_.find(node_name) == nodes_to_preserve_.end()) {
456       // Mark the node for deletion.
457       nodes_to_delete->insert(node_idx);
458
459       // Disconnect the node from its inputs to enable further optimizations.
460       node_map_->RemoveInputs(node_name);
461       node->clear_input();
462     }
463   }
464 }
465
466 void DependencyOptimizer::CleanControlInputs() {
467   for (int i = 0; i < optimized_graph_->node_size(); ++i) {
468     DedupControlInputs(optimized_graph_->mutable_node(i));
469   }
470 }
```

```
471
472 Status DependencyOptimizer::OptimizeDependencies() {
473   SetVector<int> nodes_to_simplify;
474   std::set<int> nodes_to_delete;
475   for (int i = 0; i < optimized_graph_->node_size(); ++i) {
476     const NodeDef& node = optimized_graph_->node(i);
477     if (IsNoOp(node) || IsIdentity(node) || IsIdentityN(node) ||
478         IsConstant(node) || SafeToConvertToNoOp(node)) {
479       nodes_to_simplify.PushBack(i);
480     }
481   }
482   while (!nodes_to_simplify.Empty()) {
483     int node_to_simplify = nodes_to_simplify.PopBack();
484     // Discard nodes that were marked for deletion already.
485     while (nodes_to_delete.find(node_to_simplify) != nodes_to_delete.end()) {
486       node_to_simplify = nodes_to_simplify.PopBack();
487     }
488     OptimizeNode(node_to_simplify, &nodes_to_simplify, &nodes_to_delete);
489   }
490
491   if (fetch_nodes_known_) {
492     VLOG(1) << "Deleted " << nodes_to_delete.size() << " out of "
493             << optimized_graph_->node_size() << " nodes.";
494     EraseNodesFromGraph(nodes_to_delete, optimized_graph_);
495     node_map_.reset(new NodeMap(optimized_graph_));
496     BuildNodeToIdx();
497   }
498   return Status::OK();
499 }
500
501 namespace {
502
503 enum DistanceFromSource : uint8 { ZERO = 0, ONE = 1, TWO_OR_GREATER = 2 };
504
505 void LongestPathsLowerBounds(
506     int source, const std::pair<int, int>& target_range,
507     const std::vector<std::vector<int>>& outputs,
508     std::vector<DistanceFromSource>* longest_distance) {
509   std::deque<int> queue;
510   queue.emplace_front(source);
511   while (!queue.empty()) {
512     int node = queue.front();
513     queue.pop_front();
514     for (int fanout : outputs[node]) {
515       // 1) Only nodes in the target range can be on paths from source to one of
516       //    its control outputs.
517       // 2) Since we only need a lower bound on the longest distance, we can
518       //    skip nodes for which we have already proven have a path of
519       //    length > 1 from the source.
```

```
      if (fanout >= target_range.first && fanout <= target_range.second &&
          (*longest_distance)[fanout] != TWO_OR_GREATER) {
        (*longest_distance)[fanout] =
            (*longest_distance)[fanout] == ZERO ? ONE : TWO_OR_GREATER;
        queue.emplace_front(fanout);
      }
    }
  }
}

}  // namespace

Status DependencyOptimizer::TransitiveReduction() {
  // PRECONDITION: optimized_graph_ must be sorted topologically.
  const int num_nodes = optimized_graph_->node_size();
  // Set up a compressed version of the graph to save a constant factor in the
  // expensive algorithm below. Also cache the set of control outputs and the
  // highest index of a target of any control output from each node.
  int num_controls = 0;
  std::vector<std::vector<int>> outputs(num_nodes);
  std::vector<gtl::InlinedVector<std::pair<int, int>, 2>> control_outputs(
      num_nodes);
  // target_range[i] contains the range of node indices for which to compute
  // longest paths starting from node i.
  std::vector<std::pair<int, int>> target_range(num_nodes, {num_nodes, -1});
  for (int node_idx = 0; node_idx < num_nodes; ++node_idx) {
    const NodeDef& node = optimized_graph_->node(node_idx);
    if (ModifiesFrameInfo(node) || !HasOpDef(node)) {
      // Ignore function nodes and nodes that modify frame info.
      continue;
    }
    for (int input_slot = 0; input_slot < node.input_size(); ++input_slot) {
      const string& input = node.input(input_slot);
      const NodeDef* input_node = node_map_->GetNode(input);
      if (ModifiesFrameInfo(*input_node) || IsMerge(*input_node)) {
        // Ignore edges from nodes that modify frame info and from Merge nodes,
        // because we cannot know which of it's input paths executes.
        continue;
      }
      const int input_node_idx = node_to_idx_[input_node];
      outputs[input_node_idx].push_back(node_idx);
      target_range[input_node_idx].first =
          std::min(target_range[input_node_idx].first, node_idx);
      if (IsControlInput(input)) {
        ++num_controls;
        control_outputs[input_node_idx].emplace_back(node_idx, input_slot);
        target_range[input_node_idx].second =
            std::max(target_range[input_node_idx].second, node_idx);
      }
```

```
    }
  }

  // Run the longest path in DAG algorithm for each source node that has control
  // outputs. If, for any target node of a control output, there exists a path
  // of length > 1, we can drop that control dependency.
  int num_controls_removed = 0;
  std::vector<DistanceFromSource> longest_distance(num_nodes);
  // Map from target_index -> set of (input_slot, source_index), representing
  // the control edges to remove. We sort them in reverse order by input slot,
  // such that when we swap them out so we don't clobber the
  // node(target).input() repeated field.
  typedef std::pair<int, int> InputSlotAndSource;
  absl::flat_hash_map<
      int, std::set<InputSlotAndSource, std::greater<InputSlotAndSource>>>
      control_edges_to_remove;
  for (int source = 0; source < num_nodes; ++source) {
    if (target_range[source].first >= target_range[source].second ||
        target_range[source].second <= source) {
      continue;
    }
    // Compute the set of nodes in the transitive fanout of source with
    // topological sort index in [target_range.first : target_range.second]]
    // to which there exists a path of length 2 or more from source.
    std::fill(longest_distance.begin() + target_range[source].first,
              longest_distance.begin() + target_range[source].second + 1, ZERO);
    LongestPathsLowerBounds(source, target_range[source], outputs,
                            &longest_distance);

    // If the longest path from source to target of a control dependency is
    // longer than 1, there exists an alternate path, and we can eliminate the
    // redundant direct control dependency.
    for (const auto& control_output : control_outputs[source]) {
      const int target = control_output.first;
      if (longest_distance[target] == TWO_OR_GREATER) {
        const int input_slot = control_output.second;
        control_edges_to_remove[target].emplace(input_slot, source);
      }
    }
  }
  for (const auto& it : control_edges_to_remove) {
    const int target = it.first;
    NodeDef* target_node = optimized_graph_->mutable_node(target);
    for (const InputSlotAndSource& slot_and_source : it.second) {
      const int input_slot = slot_and_source.first;
      const int source = slot_and_source.second;
      const NodeDef& source_node = optimized_graph_->node(source);
      CHECK_LT(input_slot, target_node->input_size());
      target_node->mutable_input()->SwapElements(input_slot,
```

```
                                                target_node->input_size() - 1);
      node_map_->RemoveOutput(source_node.name(), target_node->name());
      target_node->mutable_input()->RemoveLast();
      ++num_controls_removed;
    }
  }
  VLOG(1) << "Removed " << num_controls_removed << " out of " << num_controls
          << " control dependencies";
  return Status::OK();
}

void DependencyOptimizer::BuildNodeToIdx() {
  // Set up &node -> index map.
  node_to_idx_.clear();
  for (int i = 0; i < optimized_graph_->node_size(); ++i) {
    const NodeDef& node = optimized_graph_->node(i);
    node_to_idx_[&node] = i;
  }
}

// Suppose there are cross-device control inputs to node C from multiple nodes
// that are located on another device, e.g., we have control edges:
// A->C, B->C
// where A and B are on device X and C is on device Y.
// We can reduce cross-device communication by introducing an intermediate
// NoOp node C' on device X and rewriting the control edges to:
// A->C', B->C', C' -> C
void DependencyOptimizer::GroupCrossDeviceControlEdges(bool host_granularity) {
  VLOG(1)
      << "DependencyOptimizer::GroupCrossDeviceControlEdges host_granularity="
      << host_granularity;
  const int num_nodes = optimized_graph_->node_size();
  for (int i = 0; i < num_nodes; ++i) {
    NodeDef* node = optimized_graph_->mutable_node(i);
    if (node->device().empty()) continue;
    string rest, node_device = node->device();
    if (host_granularity) {
      DeviceNameUtils::SplitDeviceName(node->device(), &node_device, &rest);
    }

    // Creates new noop nodes for devices on which multiple control inputs are
    // located.

    // Map keyed by device name to the newly introduced Noop node for that
    // device. A nullptr value means that we have only seen a single node on
    // that device.
    std::map<string, NodeDef*> noops;
    int num_noops = 0;
    for (int j = 0; j < node->input_size(); ++j) {
```

```
667       if (IsControlInput(node->input(j))) {
668         const NodeDef* input = node_map_->GetNode(node->input(j));
669         if (input == nullptr || input->device().empty()) continue;
670         string input_device = input->device();
671         if (host_granularity) {
672           DeviceNameUtils::SplitDeviceName(input->device(), &input_device,
673                                            &rest);
674         }
675         if (input_device != node_device) {
676           VLOG(2) << "Cross-device " << node->name() << " " << input->device()
677                   << " -> " << node->device();
678           auto emplace_result = noops.emplace(input_device, nullptr);
679           if (!emplace_result.second &&
680               emplace_result.first->second == nullptr) {
681             VLOG(2) << "Duplicate input device from " << node->name();
682             // This is the second cross-device control input from the same
683             // device. Creates an intermediate noop node on that device.
684             string group_name;
685             NodeDef* noop;
686             // Creates a fresh node name; there may be conflicting names from
687             // a previous iteration of the optimizer.
688             do {
689               group_name = AddPrefixToNodeName(
690                   node->name(),
691                   strings::StrCat("GroupCrossDeviceControlEdges_", num_noops));
692               noop = node_map_->GetNode(group_name);
693               ++num_noops;
694             } while (noop != nullptr);
695             noop = optimized_graph_->add_node();
696             noop->set_name(group_name);
697             noop->set_device(input->device());
698             noop->set_op("NoOp");
699             node_map_->AddNode(noop->name(), noop);
700             emplace_result.first->second = noop;
701             VLOG(1) << "GroupCrossDeviceControlEdges: Added "
702                     << SummarizeNodeDef(*noop);
703           }
704         }
705       }
706     }
707
708     // Reroute existing control edges to go via the newly introduced NoOp nodes.
709     int pos = 0;
710     while (pos < node->input_size()) {
711       const string& input_name = node->input(pos);
712       if (IsControlInput(input_name)) {
713         NodeDef* input = node_map_->GetNode(input_name);
714         if (input == nullptr) {
715           ++pos;
```

```
        } else {
          string input_device = input->device();
          if (host_granularity) {
            DeviceNameUtils::SplitDeviceName(input->device(), &input_device,
                                             &rest);
          }
          auto it = noops.find(input_device);
          if (it == noops.end() || it->second == nullptr) {
            ++pos;
          } else {
            VLOG(2) << "Rewriting input from " << input_name;
            node->mutable_input()->SwapElements(pos, node->input_size() - 1);
            node->mutable_input()->RemoveLast();
            it->second->add_input(AsControlDependency(*input));
            node_map_->UpdateOutput(input_name, node->name(),
                                    it->second->name());
          }
        }
      } else {
        ++pos;
      }
    }
    for (const auto& entry : noops) {
      if (entry.second) {
        node->add_input(AsControlDependency(*entry.second));
        node_map_->AddOutput(entry.second->name(), node->name());
      }
    }
  }
}

Status DependencyOptimizer::Optimize(Cluster* cluster, const GrapplerItem& item,
                                     GraphDef* optimized_graph) {
  optimized_graph_ = optimized_graph;
  *optimized_graph_ = item.graph;
  nodes_to_preserve_ = item.NodesToPreserve();
  fetch_nodes_known_ = !item.fetch.empty();
  CleanControlInputs();

  const int num_iterations = 2;
  for (int iteration = 0; iteration < num_iterations; ++iteration) {
    GRAPPLER_RETURN_IF_DEADLINE_EXCEEDED();
    Status topo_sort_status;
    // Perform topological sort to prepare the graph for transitive reduction.
    topo_sort_status = TopologicalSort(optimized_graph_);
    // Set up index-based graph datastructures to speed up analysis steps below.
    node_map_.reset(new NodeMap(optimized_graph_));
    BuildNodeToIdx();

```

```
765     if (topo_sort_status.ok()) {
766       // Remove redundant control dependencies.
767       TF_RETURN_IF_ERROR(TransitiveReduction());
768     } else {
769       LOG(ERROR) << "Iteration = " << iteration
770                  << ", topological sort failed with message: "
771                  << topo_sort_status.error_message();
772     }
773     // Turn nodes with only control outputs into NoOps, prune NoOp and Identity
774     // nodes.
775     TF_RETURN_IF_ERROR(OptimizeDependencies());
776
777     // Dedup control inputs.
778     CleanControlInputs();
779
780     // Merge multiple control edges from the same device.
781     GroupCrossDeviceControlEdges(/*host_granularity=*/false);
782
783     // Merge control edges from the same host to reduce RPC traffic.
784     GroupCrossDeviceControlEdges(/*host_granularity=*/true);
785   }
786
787   return Status::OK();
788 }
789
790 }  // end namespace grappler
791 }  // end namespace tensorflow
```