Dan Moldovan Clean up placer rules. ... ✓ History

42 contributors +24

1009 lines (888 sloc) | 33.3 KB ...

```
/* Copyright 2015 The TensorFlow Authors. All Rights Reserved.

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

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

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

#include "tensorflow/core/graph/graph.h"

#include <memory>
#include <vector>

#include "absl/container/flat_hash_map.h"
#include "tensorflow/core/framework/full_type.pb.h"
#include "tensorflow/core/framework/graph.pb.h"
#include "tensorflow/core/framework/node_def.pb.h"
#include "tensorflow/core/framework/node_properties.h"
#include "tensorflow/core/framework/op_def_builder.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/versions.pb.h"
#include "tensorflow/core/graph/graph_node_util.h"
```

```
#include "tensorflow/core/graph/while_context.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/gtl/map_util.h"
#include "tensorflow/core/lib/hash/hash.h"
#include "tensorflow/core/lib/strings/strcat.h"
#include "tensorflow/core/lib/strings/stringprintf.h"
#include "tensorflow/core/platform/errors.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/public/version.h"

namespace tensorflow {

const int Graph::kControlSlot = -1;

// Node
Node::NodeClass Node::GetNodeClassForOp(const std::string& ts) {
  static const absl::flat_hash_map<std::string, Node::NodeClass>*
      kNodeClassTable =
#define REF_CLASS(key, value) \
  {key, value}, { "Ref" key, value }
          new absl::flat_hash_map<std::string, Node::NodeClass>({
              // Keep in same order as NodeClass values
              REF_CLASS("Switch", NC_SWITCH),
              REF_CLASS("_SwitchN", NC_SWITCH),
              REF_CLASS("Merge", NC_MERGE),
              REF_CLASS("Enter", NC_ENTER),
              REF_CLASS("Exit", NC_EXIT),
              REF_CLASS("NextIteration", NC_NEXT_ITERATION),
              {"LoopCond", NC_LOOP_COND},
              {"ControlTrigger", NC_CONTROL_TRIGGER},
              {"_Send", NC_SEND},
              {"_HostSend", NC_HOST_SEND},
              {"_Recv", NC_RECV},
              {"_HostRecv", NC_HOST_RECV},
              {"Const", NC_CONSTANT},
              {"HostConst", NC_CONSTANT},
              {"Variable", NC_VARIABLE},
              {"VariableV2", NC_VARIABLE},
              REF_CLASS("Identity", NC_IDENTITY),
              {"GetSessionHandle", NC_GET_SESSION_HANDLE},
              {"GetSessionHandleV2", NC_GET_SESSION_HANDLE},
              {"GetSessionTensor", NC_GET_SESSION_TENSOR},
              {"DeleteSessionTensor", NC_DELETE_SESSION_TENSOR},
              {"Size", NC_METADATA},
              {"Shape", NC_METADATA},
              {"Rank", NC_METADATA},
              {"_ScopedAllocator", NC_SCOPED_ALLOCATOR},
              {"CollectiveReduce", NC_COLLECTIVE},
              {"CollectiveBcastSend", NC_COLLECTIVE},
```

```
              {"CollectiveBcastRecv", NC_COLLECTIVE},
              {"CollectiveGather", NC_COLLECTIVE},
              {"FakeParam", NC_FAKE_PARAM},
              {"PartitionedCall", NC_PARTITIONED_CALL},
              {"StatefulPartitionedCall", NC_PARTITIONED_CALL},
              {"SymbolicGradient", NC_SYMBOLIC_GRADIENT},
              {"If", NC_IF},
              {"StatelessIf", NC_IF},
              {"While", NC_WHILE},
              {"StatelessWhile", NC_WHILE},
              {"Case", NC_CASE},
              {"StatelessCase", NC_CASE},
              // Not using the constants defined in FunctionLibraryDefinition
              // for the
              // 4 ops below because android inference library does not link
              // tf.function related files.
              {"_Arg", NC_ARG},
              {"_DeviceArg", NC_ARG},
              {"_Retval", NC_RETVAL},
              {"_DeviceRetval", NC_RETVAL},
              {"_XlaMerge", NC_MERGE},
          });
#undef REF_CLASS

  auto it = kNodeClassTable->find(ts);
  if (it != kNodeClassTable->end()) {
    return it->second;
  } else {
    return NC_OTHER;
  }
}

std::string Node::DebugString() const {
  std::string ret = strings::StrCat("{name:'", name(), "' id:", id_);
  if (IsSource()) {
    strings::StrAppend(&ret, " source}");
  } else if (IsSink()) {
    strings::StrAppend(&ret, " sink}");
  } else {
    strings::StrAppend(&ret, " op device:", "{requested: '", requested_device(),
                       "', assigned: '", assigned_device_name(), "'}", " def:{",
                       SummarizeNode(*this), "}}");
  }
  return ret;
}

Node::Node()
    : id_(-1),
      cost_id_(-1),
```

```
      class_(NC_UNINITIALIZED),
      props_(nullptr),
      assigned_device_name_index_(0),
      while_ctx_(nullptr) {}

void Node::Initialize(int id, int cost_id,
                      std::shared_ptr<NodeProperties> props,
                      Node::NodeClass node_class) {
  DCHECK_EQ(id_, -1);
  DCHECK(in_edges_.empty());
  DCHECK(out_edges_.empty());
  id_ = id;
  cost_id_ = cost_id;

  props_ = std::move(props);
  class_ = node_class;
}

void Node::Clear() {
  in_edges_.clear();
  out_edges_.clear();
  id_ = -1;
  cost_id_ = -1;
  class_ = NC_UNINITIALIZED;
  props_.reset();
  assigned_device_name_index_ = 0;
}

void Node::UpdateProperties() {
  DataTypeVector inputs;
  DataTypeVector outputs;
  Status status =
      InOutTypesForNode(props_->node_def, *(props_->op_def), &inputs, &outputs);
  if (!status.ok()) {
    LOG(ERROR) << "Failed at updating node: " << status;
    return;
  }
  if (props_->input_types != inputs || props_->output_types != outputs) {
    if (TF_PREDICT_TRUE(props_.use_count() == 1)) {
      props_->input_types = inputs;
      props_->input_types_slice = props_->input_types;
      props_->output_types = outputs;
      props_->output_types_slice = props_->output_types;
    } else {
      props_ = std::make_shared<NodeProperties>(
          props_->op_def, std::move(props_->node_def), inputs, outputs);
    }
  }
}
```

```
177
178 void Node::ClearTypeInfo() {
179   if (props_->node_def.has_experimental_type()) {
180     MaybeCopyOnWrite();
181     props_->node_def.clear_experimental_type();
182   }
183 }
184
185 void Node::RunForwardTypeInference() {
186   VLOG(4) << "Forward type inference: " << props_->node_def.DebugString();
187
188   if (props_->fwd_type_fn == nullptr) {
189     return;
190   }
191
192   std::vector<Node*> input_nodes(props_->input_types.size(), nullptr);
193   std::vector<int> input_idx(props_->input_types.size(), 0);
194   for (const auto& edge : in_edges_) {
195     if (edge->IsControlEdge()) {
196       continue;
197     }
198     DCHECK(edge->dst_input() < input_nodes.size()) << DebugString();
199     int i = edge->dst_input();
200     input_nodes.at(i) = edge->src();
201     input_idx.at(i) = edge->src_output();
202   }
203
204   // Note: technically, we could use a very generic type when some of the inputs
205   // are unknown. But there is an expectation that a node will have complete
206   // inputs soon, so updating intermediate types is largely unnecessary.
207
208   for (const auto* node : input_nodes) {
209     if (node == nullptr) {
210       // Incomplete inputs, bail.
211       ClearTypeInfo();
212       return;
213     }
214   }
215
216   static FullTypeDef* no_type = new FullTypeDef();
217
218   std::vector<std::reference_wrapper<const FullTypeDef>> input_types;
219   for (int i = 0; i < input_nodes.size(); i++) {
220     const auto* node = input_nodes[i];
221     if (node->def().has_experimental_type()) {
222       const auto& node_t = node->def().experimental_type();
223       if (node_t.type_id() != TFT_UNSET) {
224         int ix = input_idx[i];
225         DCHECK(ix < node_t.args_size())
```

```
            << "input " << i << " should have an output " << ix
            << " but instead only has " << node_t.args_size()
            << " outputs: " << node_t.DebugString();
        input_types.emplace_back(node_t.args(ix));
      } else {
        input_types.emplace_back(*no_type);
      }
    } else {
      // Incomplete inputs, bail.
      ClearTypeInfo();
      return;
    }
  }

  const auto infer_type = props_->fwd_type_fn(input_types);
  const FullTypeDef infer_typedef = infer_type.ValueOrDie();
  if (infer_typedef.type_id() != TFT_UNSET) {
    MaybeCopyOnWrite();
    *(props_->node_def.mutable_experimental_type()) = infer_typedef;
  }
}

const std::string& Node::name() const { return props_->node_def.name(); }
const std::string& Node::type_string() const { return props_->node_def.op(); }
const NodeDef& Node::def() const { return props_->node_def; }
const OpDef& Node::op_def() const { return *props_->op_def; }

NodeDef* Node::mutable_def() { return &props_->node_def; }

int32 Node::num_inputs() const { return props_->input_types.size(); }
DataType Node::input_type(int32_t i) const { return props_->input_types[i]; }
const DataTypeVector& Node::input_types() const { return props_->input_types; }

int32 Node::num_outputs() const { return props_->output_types.size(); }
DataType Node::output_type(int32_t o) const { return props_->output_types[o]; }
const DataTypeVector& Node::output_types() const {
  return props_->output_types;
}

AttrSlice Node::attrs() const { return AttrSlice(def()); }

const protobuf::RepeatedPtrField<std::string>& Node::requested_inputs() const {
  return def().input();
}

const std::string& Node::requested_device() const { return def().device(); }

gtl::iterator_range<NeighborIter> Node::out_nodes() const {
  return gtl::make_range(NeighborIter(out_edges_.begin(), false),
```

```
275                         NeighborIter(out_edges_.end(), false));
276 }
277
278 gtl::iterator_range<NeighborIter> Node::in_nodes() const {
279   return gtl::make_range(NeighborIter(in_edges_.begin(), true),
280                          NeighborIter(in_edges_.end(), true));
281 }
282
283 void Node::MaybeCopyOnWrite() {
284   // TODO(mdan): As nodes become more dynamic, this may not be worth the cost.
285   // NodeProperties may be shared between Nodes. Make a copy if so.
286   if (!props_.unique()) {
287     props_ = std::make_shared<NodeProperties>(*props_);
288   }
289 }
290
291 AttrValue* Node::AddAttrHelper(const std::string& name) {
292   MaybeCopyOnWrite();
293   return &((*props_->node_def.mutable_attr())[name]);
294 }
295
296 void Node::ClearAttr(const std::string& name) {
297   MaybeCopyOnWrite();
298   (*props_->node_def.mutable_attr()).erase(name);
299 }
300
301 void Node::set_name(std::string name) {
302   MaybeCopyOnWrite();
303   props_->node_def.set_name(std::move(name));
304 }
305
306 void Node::set_requested_device(const std::string& device) {
307   MaybeCopyOnWrite();
308   props_->node_def.set_device(device);
309 }
310
311 void Node::set_original_node_names(const std::vector<std::string>& names) {
312   MaybeCopyOnWrite();
313   props_->node_def.mutable_experimental_debug_info()
314       ->clear_original_node_names();
315   if (!names.empty()) {
316     *props_->node_def.mutable_experimental_debug_info()
317          ->mutable_original_node_names() = {names.begin(), names.end()};
318   }
319 }
320
321 void Node::set_original_func_names(const std::vector<std::string>& names) {
322   MaybeCopyOnWrite();
323   props_->node_def.mutable_experimental_debug_info()
```

```
      ->clear_original_func_names();
  if (!names.empty()) {
    *props_->node_def.mutable_experimental_debug_info()
         ->mutable_original_func_names() = {names.begin(), names.end()};
  }
}

Status Node::input_edge(int idx, const Edge** e) const {
  if (idx < 0 || idx >= num_inputs()) {
    return errors::InvalidArgument("Invalid input_edge index: ", idx, ", Node ",
                                   name(), " only has ", num_inputs(),
                                   " inputs.");
  }

  // This does a linear search over the edges.  In the common case,
  // the number of elements is small enough that this search isn't
  // expensive.  Should it become a bottleneck, one can make an
  // optimization where, if the number of edges is small, we use
  // linear iteration, and if the number of edges is large, we perform
  // an indexing step during construction that keeps an array of Edges
  // indexed by pointer.  This would keep the size of each Node small
  // in the common case but make this function faster when the number
  // of edges is large.
  for (const Edge* edge : in_edges()) {
    if (edge->dst_input() == idx) {
      *e = edge;
      return Status::OK();
    }
  }

  return errors::NotFound("Could not find input edge ", idx, " for ", name());
}

// Returns a vector of the non-control input edges to a node, indexed by ID.
Status Node::input_edges(std::vector<const Edge*>* input_edges) const {
  input_edges->clear();
  input_edges->resize(num_inputs(), nullptr);

  for (const Edge* edge : in_edges()) {
    if (edge->IsControlEdge()) continue;
    if (edge->dst_input() < 0 || edge->dst_input() >= num_inputs()) {
      return errors::Internal("Invalid edge input number ", edge->dst_input());
    }
    if ((*input_edges)[edge->dst_input()] != nullptr) {
      return errors::Internal("Duplicate edge input number: ",
                              edge->dst_input());
    }
    (*input_edges)[edge->dst_input()] = edge;
  }
```

```
373
374   for (int i = 0; i < num_inputs(); ++i) {
375     if ((*input_edges)[i] == nullptr) {
376       return errors::InvalidArgument("Missing edge input number: ", i);
377     }
378   }
379   return Status::OK();
380 }
381
382 Status Node::input_node(int idx, Node** n) const {
383   const Edge* e;
384   TF_RETURN_IF_ERROR(input_edge(idx, &e));
385   if (e == nullptr) {
386     *n = nullptr;
387   } else {
388     *n = e->src();
389   }
390   return Status::OK();
391 }
392
393 Status Node::input_node(int idx, const Node** const_n) const {
394   Node* n;
395   TF_RETURN_IF_ERROR(input_node(idx, &n));
396   *const_n = n;
397   return Status::OK();
398 }
399
400 Status Node::input_tensor(int idx, OutputTensor* t) const {
401   const Edge* e;
402   TF_RETURN_IF_ERROR(input_edge(idx, &e));
403   DCHECK(e != nullptr);
404   *t = OutputTensor(e->src(), e->src_output());
405   return Status::OK();
406 }
407
408 // NodeDebugInfo
409
410 NodeDebugInfo::NodeDebugInfo(const Node& n) : NodeDebugInfo(n.def()) {}
411 NodeDebugInfo::NodeDebugInfo(const NodeDef& ndef)
412     : NodeDebugInfo(ndef.name(), ndef.has_experimental_debug_info(),
413                     ndef.experimental_debug_info()) {}
414 NodeDebugInfo::NodeDebugInfo(
415     StringPiece node_name, bool has_experimental_debug_info,
416     const NodeDef_ExperimentalDebugInfo& experimental_debug_info)
417     : name(node_name) {
418   if (has_experimental_debug_info) {
419     const auto& node_names = experimental_debug_info.original_node_names();
420     original_node_names.assign(node_names.begin(), node_names.end());
421     const auto& func_names = experimental_debug_info.original_func_names();
```

```
    original_func_names.assign(func_names.begin(), func_names.end());
  }
}
// InputTensor

bool InputTensor::operator==(const InputTensor& other) const {
  return node == other.node && index == other.index;
}

uint64 InputTensor::Hash::operator()(InputTensor const& s) const {
  return Hash64Combine(std::hash<const Node*>()(s.node),
                       std::hash<int>()(s.index));
}

// OutputTensor

bool OutputTensor::operator==(const OutputTensor& other) const {
  return node == other.node && index == other.index;
}

uint64 OutputTensor::Hash::operator()(OutputTensor const& s) const {
  return Hash64Combine(std::hash<const Node*>()(s.node),
                       std::hash<int>()(s.index));
}

// Graph

Graph::Graph(const OpRegistryInterface* ops)
    : ops_(ops, FunctionDefLibrary()),
      versions_(new VersionDef),
      arena_(8 << 10 /* 8kB */) {
  versions_->set_producer(TF_GRAPH_DEF_VERSION);
  versions_->set_min_consumer(TF_GRAPH_DEF_VERSION_MIN_CONSUMER);

  // Initialize the name interning table for assigned_device_name.
  device_names_.push_back("");
  DCHECK_EQ(0, InternDeviceName(""));

  // Source and sink have no endpoints, just control edges.
  NodeDef def;
  def.set_name("_SOURCE");
  def.set_op("NoOp");
  Status status;
  Node* source = AddNode(def, &status);
  TF_CHECK_OK(status);
  CHECK_EQ(source->id(), kSourceId);

  def.set_name("_SINK");
  Node* sink = AddNode(def, &status);
```

```
471   TF_CHECK_OK(status);
472   CHECK_EQ(sink->id(), kSinkId);
473
474   AddControlEdge(source, sink);
475 }
476
477 Graph::Graph(const FunctionLibraryDefinition& flib_def)
478     : Graph(flib_def.default_registry()) {
479   // Need a new-enough consumer to support the functions we add to the graph.
480   if (flib_def.num_functions() > 0 && versions_->min_consumer() < 12) {
481     versions_->set_min_consumer(12);
482   }
483   Status s = ops_.AddLibrary(flib_def);
484   CHECK(s.ok()) << s.error_message();
485 }
486
487 Graph::~Graph() {
488   // Manually call the destructors for all the Nodes we constructed using
489   // placement new.
490   for (Node* node : nodes_) {
491     if (node != nullptr) {
492       node->~Node();
493     }
494   }
495   for (Node* node : free_nodes_) {
496     node->~Node();
497   }
498   // Edges have no destructor, and we arena-allocated them, so no need to
499   // destroy them.
500 }
501
502 std::unique_ptr<Graph> Graph::Clone() {
503   std::unique_ptr<Graph> new_graph(new Graph(flib_def()));
504   new_graph->Copy(*this);
505   return new_graph;
506 }
507
508 const VersionDef& Graph::versions() const { return *versions_; }
509 void Graph::set_versions(const VersionDef& versions) { *versions_ = versions; }
510
511 void Graph::Copy(const Graph& src) {
512   SetConstructionContext(src.GetConstructionContextInternal());
513   for (Node* n : nodes()) {
514     CHECK(n->IsSource() || n->IsSink()) << "*dest must be empty";
515   }
516
517   // Copy GraphDef versions
518   set_versions(src.versions());
519
```

```
520   // Copy the nodes.
521   // "Node in src" -> "Node in *dest"
522   gtl::FlatMap<const Node*, Node*> node_map;
523   node_map.reserve(src.num_nodes());
524   node_map[src.source_node()] = source_node();
525   node_map[src.sink_node()] = sink_node();
526   for (Node* n : src.op_nodes()) {
527     auto copy = CopyNode(n);
528     copy->in_edges_.reserve(n->in_edges().size());
529     copy->out_edges_.reserve(n->out_edges().size());
530     node_map[n] = copy;
531   }
532
533   // Copy the edges
534   edges_.reserve(src.num_edges());
535   for (const Edge* e : src.edges()) {
536     Node* src_copy = node_map[e->src()];
537     Node* dst_copy = node_map[e->dst()];
538     AddEdge(src_copy, e->src_output(), dst_copy, e->dst_input());
539   }
540 }
541
542 Node* Graph::AddNode(NodeDef node_def, Status* status) {
543   const OpRegistrationData* op_reg_data;
544   status->Update(ops_.LookUp(node_def.op(), &op_reg_data));
545   if (!status->ok()) return nullptr;
546
547   DataTypeVector inputs;
548   DataTypeVector outputs;
549   status->Update(
550       InOutTypesForNode(node_def, op_reg_data->op_def, &inputs, &outputs));
551   if (!status->ok()) {
552     *status = AttachDef(*status, node_def);
553     return nullptr;
554   }
555
556   Node::NodeClass node_class = op_reg_data->is_function_op
557                                    ? Node::NC_FUNCTION_OP
558                                    : Node::GetNodeClassForOp(node_def.op());
559
560   if (op_reg_data->type_ctor != nullptr) {
561     VLOG(3) << "AddNode: found type constructor for " << node_def.name();
562     const auto ctor_type =
563         full_type::SpecializeType(AttrSlice(node_def), op_reg_data->op_def);
564     const FullTypeDef ctor_typedef = ctor_type.ValueOrDie();
565     if (ctor_typedef.type_id() != TFT_UNSET) {
566       *(node_def.mutable_experimental_type()) = ctor_typedef;
567     }
568   } else {
```

```
569       VLOG(3) << "AddNode: no type constructor for " << node_def.name();
570     }
571
572     Node* node = AllocateNode(std::make_shared<NodeProperties>(
573                                   &op_reg_data->op_def, std::move(node_def),
574                                   inputs, outputs, op_reg_data->fwd_type_fn),
575                               nullptr, node_class);
576     return node;
577   }
578
579   Node* Graph::CopyNode(const Node* node) {
580     DCHECK(!node->IsSource());
581     DCHECK(!node->IsSink());
582     Node* copy = AllocateNode(node->props_, node, node->class_);
583     copy->set_assigned_device_name(node->assigned_device_name());
584
585     // Since the OpDef of a function may be owned by the Graph that owns 'node',
586     // relookup the OpDef in the target graph. If it differs, then clone the
587     // node properties with the updated OpDef.
588     const OpDef* op_def;
589     TF_CHECK_OK(ops_.LookUpOpDef(node->type_string(), &op_def));
590     if (op_def != node->props_->op_def) {
591       copy->MaybeCopyOnWrite();
592       copy->props_->op_def = op_def;
593     }
594     copy->SetStackTrace(node->GetStackTrace());
595
596     return copy;
597   }
598
599   void Graph::RemoveNode(Node* node) {
600     TF_DCHECK_OK(IsValidNode(node)) << node->DebugString();
601     DCHECK(!node->IsSource());
602     DCHECK(!node->IsSink());
603
604     // Remove any edges involving this node.
605     for (const Edge* e : node->in_edges_) {
606       CHECK_EQ(e->src_->out_edges_.erase(e), size_t{1});
607       edges_[e->id_] = nullptr;
608       RecycleEdge(e);
609       --num_edges_;
610     }
611     node->in_edges_.clear();
612     for (const Edge* e : node->out_edges_) {
613       CHECK_EQ(e->dst_->in_edges_.erase(e), size_t{1});
614       edges_[e->id_] = nullptr;
615       RecycleEdge(e);
616       --num_edges_;
617     }
```

```
618   node->out_edges_.clear();
619   ReleaseNode(node);
620 }
621
622 const Edge* Graph::AddEdge(Node* source, int x, Node* dest, int y) {
623   TF_DCHECK_OK(IsValidNode(source)) << source->DebugString();
624   TF_DCHECK_OK(IsValidNode(dest)) << dest->DebugString();
625
626   // source/sink must only be linked via control slots, and
627   // control slots must only be linked to control slots.
628   if (source == source_node() || dest == sink_node() || x == kControlSlot ||
629       y == kControlSlot) {
630     DCHECK_EQ(x, kControlSlot) << source->DebugString();
631     DCHECK_EQ(y, kControlSlot) << dest->DebugString();
632   }
633
634   Edge* e = nullptr;
635   if (free_edges_.empty()) {
636     e = new (arena_.Alloc(sizeof(Edge))) Edge;  // placement new
637   } else {
638     e = free_edges_.back();
639     free_edges_.pop_back();
640   }
641   e->id_ = edges_.size();
642   e->src_ = source;
643   e->dst_ = dest;
644   e->src_output_ = x;
645   e->dst_input_ = y;
646   CHECK(source->out_edges_.insert(e).second);
647   CHECK(dest->in_edges_.insert(e).second);
648   edges_.push_back(e);
649   ++num_edges_;
650
651   if (!e->IsControlEdge()) {
652     if (dest->in_edges_.size() >= dest->props_->input_types.size()) {
653       // Note: this only produces consistent results at graph construction,
654       // and only when all incoming edges are up-to-date.
655       // If the graph is subsequently modified, or if the node is added before
656       // any of its upstream nodes, this type information would change as well.
657       // In general, graph transformations should run shole-graph type inference
658       // when done, and should not rely on types being fully up to date
659       // after each AddNode.
660       // TODO(mdan): Should we even run type inference here any more?
661       dest->RunForwardTypeInference();
662     }
663   }
664
665   return e;
666 }
```

```
667
668 void Graph::RemoveEdge(const Edge* e) {
669   TF_DCHECK_OK(IsValidNode(e->src_)) << e->src_->DebugString();
670   TF_DCHECK_OK(IsValidNode(e->dst_)) << e->dst_->DebugString();
671   CHECK_EQ(e->src_->out_edges_.erase(e), size_t{1});
672   CHECK_EQ(e->dst_->in_edges_.erase(e), size_t{1});
673   CHECK_EQ(e, edges_[e->id_]);
674   CHECK_GT(num_edges_, 0);
675
676   edges_[e->id_] = nullptr;
677   RecycleEdge(e);
678   --num_edges_;
679
680   if (!e->IsControlEdge()) {
681     // This may clear the node type if enough edges are removed.
682     e->dst_->RunForwardTypeInference();
683   }
684 }
685
686 void Graph::RecycleEdge(const Edge* e) {
687   free_edges_.push_back(const_cast<Edge*>(e));
688 }
689
690 const Edge* Graph::AddControlEdge(Node* source, Node* dest,
691                                   bool allow_duplicates) {
692   if (!allow_duplicates) {
693     for (const Edge* edge : dest->in_edges()) {
694       if (edge->IsControlEdge() && edge->src() == source) {
695         // The requested edge already exists.
696         return nullptr;
697       }
698     }
699   }
700   // Modify dest's NodeDef if necessary.
701   if (!source->IsSource() && !dest->IsSink() && !allow_duplicates) {
702     // Check if this input is already in dest's NodeDef.
703     const std::string new_input = strings::StrCat("^", source->name());
704     bool input_exists = false;
705     for (const std::string& input : dest->props_->node_def.input()) {
706       if (input == new_input) {
707         input_exists = true;
708         break;
709       }
710     }
711     if (!input_exists) {
712       dest->MaybeCopyOnWrite();
713       dest->props_->node_def.add_input(new_input);
714     }
715   }
```

```
716   return AddEdge(source, kControlSlot, dest, kControlSlot);
717 }
718
719 void Graph::RemoveControlEdge(const Edge* e) {
720   if (!e->src_->IsSource() && !e->dst_->IsSink()) {
721     e->dst_->MaybeCopyOnWrite();
722     std::string e_src_name = strings::StrCat("^", e->src_->name());
723     auto* inputs = e->dst_->props_->node_def.mutable_input();
724     for (auto it = inputs->begin(); it != inputs->end(); ++it) {
725       if (*it == e_src_name) {
726         inputs->erase(it);
727         break;
728       }
729     }
730   }
731   RemoveEdge(e);
732 }
733
734 namespace {
735 const Edge* FindEdge(const Node* dst, int index) {
736   for (const Edge* e : dst->in_edges()) {
737     if (e->dst_input() == index) return e;
738   }
739   return nullptr;
740 }
741 }  // namespace
742
743 Status Graph::UpdateEdge(Node* new_src, int new_src_index, Node* dst,
744                          int dst_index) {
745   TF_RETURN_IF_ERROR(IsValidOutputTensor(new_src, new_src_index));
746   TF_RETURN_IF_ERROR(IsValidInputTensor(dst, dst_index));
747   const Edge* e = FindEdge(dst, dst_index);
748   if (e == nullptr) {
749     return errors::InvalidArgument("Couldn't find edge to ",
750                                    FormatNodeForError(*dst));
751   }
752   RemoveEdge(e);
753   AddEdge(new_src, new_src_index, dst, dst_index);
754   dst->MaybeCopyOnWrite();
755   (*dst->props_->node_def.mutable_input())[dst_index] =
756       strings::StrCat(new_src->name(), ":", new_src_index);
757   return Status::OK();
758 }
759
760 Status Graph::AddWhileInputHack(Node* new_src, int new_src_index, Node* dst) {
761   if (!dst->IsWhileNode()) {
762     return errors::Internal(
763         "dst argument to AddWhileEdgeHack should be a While op, got: ",
764         dst->DebugString());
```

```
765     }
766     TF_RETURN_IF_ERROR(IsValidOutputTensor(new_src, new_src_index));
767     // Find the current number of data inputs. We'll add the new edge to the next
768     // missing data input.
769     int dst_index = 0;
770     for (const Edge* edge : dst->in_edges()) {
771       if (edge->IsControlEdge()) continue;
772       ++dst_index;
773     }
774     TF_RETURN_IF_ERROR(IsValidInputTensor(dst, dst_index));
775     AddEdge(new_src, new_src_index, dst, dst_index);
776     dst->MaybeCopyOnWrite();
777     dst->props_->node_def.add_input(
778         strings::StrCat(new_src->name(), ":", new_src_index));
779     return Status::OK();
780   }
781
782   Status Graph::AddFunctionLibrary(const FunctionDefLibrary& fdef_lib) {
783     // Need a new-enough consumer to support the functions we add to the graph.
784     if (fdef_lib.function_size() > 0 && versions_->min_consumer() < 12) {
785       versions_->set_min_consumer(12);
786     }
787     return ops_.AddLibrary(fdef_lib);
788   }
789
790   namespace {
791
792   void AddInput(NodeDef* dst, StringPiece src_name, int src_slot) {
793     if (src_slot == Graph::kControlSlot) {
794       dst->add_input(strings::StrCat("^", src_name));
795     } else if (src_slot == 0) {
796       dst->add_input(src_name.data(), src_name.size());
797     } else {
798       dst->add_input(strings::StrCat(src_name, ":", src_slot));
799     }
800   }
801
802   }  // namespace
803
804   void Graph::ToGraphDef(GraphDef* graph_def) const {
805     ToGraphDefSubRange(graph_def, 0);
806   }
807
808   GraphDef Graph::ToGraphDefDebug() const {
809     GraphDef ret;
810     ToGraphDef(&ret);
811     return ret;
812   }
813
```

```
void Graph::ToGraphDefSubRange(GraphDef* graph_def, int from_node_id) const {
  graph_def->Clear();
  *graph_def->mutable_versions() = versions();
  *graph_def->mutable_library() = ops_.ToProto();

  graph_def->mutable_node()->Reserve(std::max(1, num_nodes() - from_node_id));

  std::vector<const Edge*>
      inputs;  // Construct this outside the loop for speed.
  for (auto id = from_node_id; id < num_node_ids(); ++id) {
    const Node* node = FindNodeId(id);
    if (node == nullptr || !node->IsOp()) continue;
    NodeDef* node_def = graph_def->add_node();
    *node_def = node->def();

    // Use the node's assigned device, if any, instead of the device requested
    // in the NodeDef.
    if (!node->assigned_device_name().empty()) {
      node_def->set_device(node->assigned_device_name());
    }

    // Get the inputs for this Node.  We make sure control inputs are
    // after data inputs, as required by GraphDef.
    inputs.clear();
    inputs.resize(node->num_inputs(), nullptr);
    for (const Edge* edge : node->in_edges()) {
      if (edge->IsControlEdge()) {
        inputs.push_back(edge);
      } else {
        DCHECK(edge->dst_input() < inputs.size())
            << "Edge " << edge->DebugString()
            << " is overflowing the expected number of inputs ("
            << node->num_inputs() << ") for node " << node->DebugString();
        CHECK(inputs[edge->dst_input()] == nullptr)
            << "Edge " << edge->src()->name() << "->" << edge->dst()->name()
            << " conflicts with pre-existing input edge "
            << inputs[edge->dst_input()]->src()->name() << "->"
            << inputs[edge->dst_input()]->dst()->name();

        inputs[edge->dst_input()] = edge;
      }
    }
    // Sort the control inputs for more predictable serialization.
    std::sort(inputs.begin() + node->num_inputs(), inputs.end(),
              [](const Edge* a, const Edge* b) -> bool {
                return a->src()->name() < b->src()->name();
              });
    node_def->clear_input();
    node_def->mutable_input()->Reserve(inputs.size());
```

```
863
864       for (size_t i = 0; i < inputs.size(); ++i) {
865         const Edge* edge = inputs[i];
866         if (edge == nullptr) {
867           if (i < node->requested_inputs().size()) {
868             node_def->add_input(node->requested_inputs()[i]);
869           } else {
870             node_def->add_input("");
871           }
872         } else {
873           const Node* src = edge->src();
874           if (!src->IsOp()) continue;
875           AddInput(node_def, src->name(), edge->src_output());
876         }
877       }
878     }
879   }
880
881   std::string Graph::NewName(StringPiece prefix) {
882     return strings::StrCat(prefix, "/_", name_counter_++);
883   }
884
885   Status Graph::IsValidNode(const Node* node) const {
886     if (node == nullptr) {
887       return errors::InvalidArgument("Node is null");
888     }
889     const int id = node->id();
890     if (id < 0) {
891       return errors::InvalidArgument("node id ", id, " is less than zero");
892     }
893     if (static_cast<size_t>(id) >= nodes_.size()) {
894       return errors::InvalidArgument(
895           "node id ", id, " is >= than number of nodes in graph ", nodes_.size());
896     }
897     if (nodes_[id] != node) {
898       return errors::InvalidArgument("Node with id ", id,
899                                      " is different from the passed in node. "
900                                      "Does it belong to a different graph?");
901     }
902     return Status::OK();
903   }
904
905   Status Graph::IsValidOutputTensor(const Node* node, int idx) const {
906     TF_RETURN_IF_ERROR(IsValidNode(node));
907     if (idx >= node->num_outputs() || idx < 0) {
908       return errors::OutOfRange("Node '", node->name(), "' (type: '",
909                                 node->op_def().name(),
910                                 "', num of outputs: ", node->num_outputs(),
911                                 ") does not have ", "output ", idx);
```

```
912   }
913   return Status::OK();
914 }
915
916 Status Graph::IsValidInputTensor(const Node* node, int idx) const {
917   TF_RETURN_IF_ERROR(IsValidNode(node));
918   if (idx >= node->num_inputs() || idx < 0) {
919     return errors::OutOfRange("Node '", node->name(), "' (type: '",
920                               node->op_def().name(),
921                               "', num of inputs: ", node->num_inputs(),
922                               ") does not have ", "input ", idx);
923   }
924   return Status::OK();
925 }
926
927 Node* Graph::AllocateNode(std::shared_ptr<NodeProperties> props,
928                           const Node* cost_node, Node::NodeClass node_class) {
929   Node* node = nullptr;
930   if (free_nodes_.empty()) {
931     node = new (arena_.Alloc(sizeof(Node))) Node;  // placement new
932   } else {
933     node = free_nodes_.back();
934     free_nodes_.pop_back();
935   }
936   node->graph_ = this;
937   const int id = nodes_.size();
938   int cost_id = cost_node ? cost_node->cost_id() : id;
939   node->Initialize(id, cost_id, std::move(props), node_class);
940   nodes_.push_back(node);
941   ++num_nodes_;
942   return node;
943 }
944
945 void Graph::ReleaseNode(Node* node) {
946   TF_DCHECK_OK(IsValidNode(node)) << node->DebugString();
947   nodes_[node->id()] = nullptr;
948   free_nodes_.push_back(node);
949   --num_nodes_;
950   node->Clear();
951 }
952
953 // Ensures that 'device_name' is present in the device name table, and returns
954 // the index of that device name. The index is stable, and can be used in
955 // calls to Node::set_assigned_device_name_index().
956 int Graph::InternDeviceName(const std::string& device_name) {
957   // Special case, very common.  Also, this allows us to use a single map
958   // lookup below, instead of two.  The 'if (index_cell > 0)' test below
959   // relies on this check.
960   if (device_name.empty()) {
```

```
961       return 0;
962     }
963
964     int& index_cell = device_names_map_[device_name];
965     if (index_cell > 0) {
966       return index_cell;
967     }
968
969     const int index = device_names_map_.size();
970     index_cell = index;
971     device_names_.push_back(device_name);
972     return index;
973   }
974
975   Status Graph::AddWhileContext(StringPiece frame_name,
976                                 std::vector<Node*> enter_nodes,
977                                 std::vector<Node*> exit_nodes,
978                                 OutputTensor cond_output,
979                                 std::vector<OutputTensor> body_inputs,
980                                 std::vector<OutputTensor> body_outputs,
981                                 WhileContext** result) {
982     auto pair = while_ctxs_.insert(std::pair<std::string, WhileContext>(
983         std::string(frame_name),
984         WhileContext(frame_name, std::move(enter_nodes), std::move(exit_nodes),
985                      cond_output, std::move(body_inputs),
986                      std::move(body_outputs))));
987     if (!pair.second) {
988       *result = nullptr;
989       return errors::InvalidArgument("WhileContext with frame name '", frame_name,
990                                      "' already exists");
991     }
992     *result = &pair.first->second;
993     return Status::OK();
994   }
995
996   std::unordered_map<std::string, Node*> Graph::BuildNodeNameIndex() const {
997     std::unordered_map<std::string, Node*> result;
998     for (Node* n : nodes()) {
999       result[n->name()] = n;
1000    }
1001    return result;
1002  }
1003
1004  std::string Edge::DebugString() const {
1005    return strings::Printf("[id=%d %s:%d -> %s:%d]", id_, src_->name().c_str(),
1006                           src_output_, dst_->name().c_str(), dst_input_);
1007  }
1008
1009  }  // namespace tensorflow
```