qqfish Fix a NPE issue in invalid Exit op. Now it will report an error inste... ... History

6 contributors






380 lines (336 sloc) | 13 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/common_runtime/immutable_executor_state.h"

#include "absl/memory/memory.h"
#include "tensorflow/core/framework/function.h"
#include "tensorflow/core/framework/metrics.h"
#include "tensorflow/core/framework/node_def_util.h"
#include "tensorflow/core/graph/edgeset.h"
#include "tensorflow/core/graph/graph.h"
#include "tensorflow/core/graph/graph_node_util.h"
#include "tensorflow/core/platform/errors.h"
#include "tensorflow/core/platform/logging.h"

namespace tensorflow {

```

```
namespace {
bool IsInitializationOp(const Node* node) {
  return node->op_def().allows_uninitialized_input();
}
}  // namespace

ImmutableExecutorState::~ImmutableExecutorState() {
  for (int32_t i = 0; i < gview_.num_nodes(); i++) {
    NodeItem* item = gview_.node(i);
    if (item != nullptr) {
      params_.delete_kernel(item->kernel);
    }
  }
}

namespace {
void GetMaxPendingCounts(const Node* n, size_t* max_pending,
                         size_t* max_dead_count) {
  const size_t num_in_edges = n->in_edges().size();
  size_t initial_count;
  if (IsMerge(n)) {
    // merge waits all control inputs so we initialize the pending
    // count to be the number of control edges.
    int32_t num_control_edges = 0;
    for (const Edge* edge : n->in_edges()) {
      if (edge->IsControlEdge()) {
        num_control_edges++;
      }
    }
    // Use bit 0 to indicate if we are waiting for a ready live data input.
    initial_count = 1 + (num_control_edges << 1);
  } else {
    initial_count = num_in_edges;
  }

  *max_pending = initial_count;
  *max_dead_count = num_in_edges;
}
}  // namespace

ImmutableExecutorState::FrameInfo* ImmutableExecutorState::EnsureFrameInfo(
    const string& fname) {
  auto iter = frame_info_.find(fname);
  if (iter != frame_info_.end()) {
    return iter->second.get();
  } else {
    auto frame_info = absl::make_unique<FrameInfo>(fname);
    absl::string_view fname_view = frame_info->name;
    auto emplace_result =
```

```
        frame_info_.emplace(fname_view, std::move(frame_info));
    return emplace_result.first->second.get();
  }
}

Status ImmutableExecutorState::Initialize(const Graph& graph) {
  TF_RETURN_IF_ERROR(gview_.Initialize(&graph));

  // Build the information about frames in this subgraph.
  ControlFlowInfo cf_info;
  TF_RETURN_IF_ERROR(BuildControlFlowInfo(&graph, &cf_info));

  for (auto& it : cf_info.unique_frame_names) {
    EnsureFrameInfo(it)->nodes =
        absl::make_unique<std::vector<const NodeItem*>>();
  }
  root_frame_info_ = frame_info_[""].get();

  pending_ids_.resize(gview_.num_nodes());

  // Preprocess every node in the graph to create an instance of op
  // kernel for each node.
  requires_control_flow_ = false;
  for (const Node* n : graph.nodes()) {
    if (IsSink(n)) continue;
    if (IsSwitch(n) || IsMerge(n) || IsEnter(n) || IsExit(n)) {
      requires_control_flow_ = true;
    } else if (IsRecv(n)) {
      // A Recv node from a different device may produce dead tensors from
      // non-local control-flow nodes.
      //
      // TODO(mrry): Track whether control flow was present in the
      // pre-partitioned graph, and enable the caller (e.g.
      // `DirectSession`) to relax this constraint.
      string send_device;
      string recv_device;
      TF_RETURN_IF_ERROR(GetNodeAttr(n->attrs(), "send_device", &send_device));
      TF_RETURN_IF_ERROR(GetNodeAttr(n->attrs(), "recv_device", &recv_device));
      if (send_device != recv_device) {
        requires_control_flow_ = true;
      }
    }

    const int id = n->id();
    const string& frame_name = cf_info.frame_names[id];
    FrameInfo* frame_info = EnsureFrameInfo(frame_name);

    NodeItem* item = gview_.node(id);
    item->node_id = id;
```

```
128
129     item->input_start = frame_info->total_inputs;
130     frame_info->total_inputs += n->num_inputs();
131
132     Status s = params_.create_kernel(n->properties(), &item->kernel);
133     if (!s.ok()) {
134       item->kernel = nullptr;
135       s = AttachDef(s, *n);
136       return s;
137     }
138     CHECK(item->kernel);
139     item->kernel_is_async = (item->kernel->AsAsync() != nullptr);
140     item->is_merge = IsMerge(n);
141     item->is_any_consumer_merge_or_control_trigger = false;
142     for (const Node* consumer : n->out_nodes()) {
143       if (IsMerge(consumer) || IsControlTrigger(consumer)) {
144         item->is_any_consumer_merge_or_control_trigger = true;
145         break;
146       }
147     }
148     const Tensor* const_tensor = item->kernel->const_tensor();
149     if (const_tensor) {
150       // Hold onto a shallow copy of the constant tensor in `*this` so that the
151       // reference count does not drop to 1. This prevents the constant tensor
152       // from being forwarded, and its buffer reused.
153       const_tensors_.emplace_back(*const_tensor);
154     }
155     item->const_tensor = const_tensor;
156     item->is_noop = (item->kernel->type_string_view() == "NoOp");
157     item->is_enter = IsEnter(n);
158     if (item->is_enter) {
159       bool is_constant_enter;
160       TF_RETURN_IF_ERROR(
161           GetNodeAttr(n->attrs(), "is_constant", &is_constant_enter));
162       item->is_constant_enter = is_constant_enter;
163
164       string frame_name;
165       TF_RETURN_IF_ERROR(GetNodeAttr(n->attrs(), "frame_name", &frame_name));
166       FrameInfo* frame_info = frame_info_[frame_name].get();
167
168       int parallel_iterations;
169       TF_RETURN_IF_ERROR(
170           GetNodeAttr(n->attrs(), "parallel_iterations", &parallel_iterations));
171
172       if (frame_info->parallel_iterations == -1) {
173         frame_info->parallel_iterations = parallel_iterations;
174       } else if (frame_info->parallel_iterations != parallel_iterations) {
175         LOG(WARNING) << "Loop frame \"" << frame_name
176                      << "\" had two different values for parallel_iterations: "
```

```
177                     << frame_info->parallel_iterations << " vs. "
178                     << parallel_iterations << ".";
179       }
180
181       if (enter_frame_info_.size() <= id) {
182         enter_frame_info_.resize(id + 1);
183       }
184       enter_frame_info_[id] = frame_info;
185     } else {
186       item->is_constant_enter = false;
187     }
188     item->is_exit = IsExit(n);
189     item->is_control_trigger = IsControlTrigger(n);
190     item->is_source = IsSource(n);
191     item->is_enter_exit_or_next_iter =
192         (IsEnter(n) || IsExit(n) || IsNextIteration(n));
193     item->is_transfer_node = IsTransferNode(n);
194     item->is_initialization_op = IsInitializationOp(n);
195     item->is_recv_or_switch = IsRecv(n) || IsSwitch(n);
196     item->is_next_iteration = IsNextIteration(n);
197     item->is_distributed_communication = IsDistributedCommunication(n);
198
199     // Compute the maximum values we'll store for this node in the
200     // pending counts data structure, and allocate a handle in
201     // that frame's pending counts data structure that has enough
202     // space to store these maximal count values.
203     size_t max_pending, max_dead;
204     GetMaxPendingCounts(n, &max_pending, &max_dead);
205     pending_ids_[id] =
206         frame_info->pending_counts_layout.CreateHandle(max_pending, max_dead);
207
208     // See if this node is a root node, and if so, add item to root_nodes_.
209     if (n->in_edges().empty()) {
210       root_nodes_.push_back(item);
211     }
212
213     // Initialize static information about the frames in the graph.
214     frame_info->nodes->push_back(item);
215     if (item->is_enter) {
216       string enter_name;
217       TF_RETURN_IF_ERROR(GetNodeAttr(n->attrs(), "frame_name", &enter_name));
218       EnsureFrameInfo(enter_name)->input_count++;
219     }
220
221     // Record information about whether each output of the op is used.
222     std::unique_ptr<bool[]> outputs_required(new bool[n->num_outputs()]);
223     std::fill(&outputs_required[0], &outputs_required[n->num_outputs()], false);
224     int32_t unused_outputs = n->num_outputs();
225     for (const Edge* e : n->out_edges()) {
```

```
      if (IsSink(e->dst())) continue;
      if (e->src_output() >= 0) {
        if (!outputs_required[e->src_output()]) {
          --unused_outputs;
          outputs_required[e->src_output()] = true;
        }
      }
    }
    if (unused_outputs > 0) {
      for (int i = 0; i < n->num_outputs(); ++i) {
        if (!outputs_required[i]) {
          metrics::RecordUnusedOutput(n->type_string());
        }
      }
      item->outputs_required = std::move(outputs_required);
    }
  }

  // Rewrite each `EdgeInfo::input_slot` member to refer directly to the input
  // location.
  for (const Node* n : graph.nodes()) {
    if (IsSink(n)) continue;
    const int id = n->id();
    NodeItem* item = gview_.node(id);

    for (EdgeInfo& e : item->mutable_output_edges()) {
      const int dst_id = e.dst_id;
      NodeItem* dst_item = gview_.node(dst_id);
      e.input_slot += dst_item->input_start;
    }
  }

  // Initialize PendingCounts only after pending_ids_[node.id] is initialized
  // for all nodes.
  InitializePending(&graph, cf_info);
  return gview_.SetAllocAttrs(&graph, params_.device);
}

namespace {
// If a Node has been marked to use a ScopedAllocator x for output i, then
// sc_attr will contain the subsequence (i, x) at an even offset.  This function
// extracts and transfers that ScopedAllocator id to alloc_attr.  For now, we
// only allow one ScopedAllocator use per Node.
bool ExtractScopedAllocatorAttr(const std::vector<int>& sc_attr,
                                int output_index,
                                AllocatorAttributes* alloc_attr) {
  DCHECK_LE(2, sc_attr.size());
  for (int i = 0; i < sc_attr.size(); i += 2) {
    if (sc_attr[i] == output_index) {
```

```
      CHECK_EQ(alloc_attr->scope_id, 0);
      alloc_attr->scope_id = sc_attr[i + 1];
      return true;
    }
  }
  return false;
}
}  // namespace

Status ImmutableExecutorState::BuildControlFlowInfo(const Graph* g,
                                                    ControlFlowInfo* cf_info) {
  const int num_nodes = g->num_node_ids();
  cf_info->frame_names.resize(num_nodes);
  std::vector<Node*> parent_nodes;
  parent_nodes.resize(num_nodes);
  std::vector<bool> visited;
  visited.resize(num_nodes);

  string frame_name;
  std::deque<Node*> ready;

  // Initialize with the root nodes.
  for (Node* n : g->nodes()) {
    if (n->in_edges().empty()) {
      visited[n->id()] = true;
      cf_info->unique_frame_names.insert(frame_name);
      ready.push_back(n);
    }
  }

  while (!ready.empty()) {
    Node* curr_node = ready.front();
    int curr_id = curr_node->id();
    ready.pop_front();

    Node* parent = nullptr;
    if (IsEnter(curr_node)) {
      // Enter a child frame.
      TF_RETURN_IF_ERROR(
          GetNodeAttr(curr_node->attrs(), "frame_name", &frame_name));
      parent = curr_node;
    } else if (IsExit(curr_node)) {
      // Exit to the parent frame.
      parent = parent_nodes[curr_id];
      if (!parent) {
        return errors::InvalidArgument(
            "Invalid Exit op: Cannot find a corresponding Enter op.");
      }
      frame_name = cf_info->frame_names[parent->id()];
```

```
      parent = parent_nodes[parent->id()];
    } else {
      parent = parent_nodes[curr_id];
      frame_name = cf_info->frame_names[curr_id];
    }

    for (const Edge* out_edge : curr_node->out_edges()) {
      Node* out = out_edge->dst();
      if (IsSink(out)) continue;
      const int out_id = out->id();

      // Add to ready queue if not visited.
      bool is_visited = visited[out_id];
      if (!is_visited) {
        ready.push_back(out);
        visited[out_id] = true;

        // Process the node 'out'.
        cf_info->frame_names[out_id] = frame_name;
        parent_nodes[out_id] = parent;
        cf_info->unique_frame_names.insert(frame_name);
      }
    }
  }

  return Status::OK();
}

void ImmutableExecutorState::InitializePending(const Graph* graph,
                                               const ControlFlowInfo& cf_info) {
  for (auto& it : cf_info.unique_frame_names) {
    FrameInfo* finfo = EnsureFrameInfo(it);
    DCHECK_EQ(finfo->pending_counts.get(), nullptr);
    finfo->pending_counts =
        absl::make_unique<PendingCounts>(finfo->pending_counts_layout);
  }

  if (!requires_control_flow_) {
    atomic_pending_counts_.reset(new std::atomic<int32>[gview_.num_nodes()]);
    std::fill(atomic_pending_counts_.get(),
              atomic_pending_counts_.get() + gview_.num_nodes(), 0);
  }

  for (const Node* n : graph->nodes()) {
    if (IsSink(n)) continue;
    const int id = n->id();
    const string& name = cf_info.frame_names[id];
    size_t max_pending, max_dead;
    GetMaxPendingCounts(n, &max_pending, &max_dead);
```

```
373       auto& counts = EnsureFrameInfo(name)->pending_counts;
374       counts->set_initial_count(pending_ids_[id], max_pending);
375       if (!requires_control_flow_) {
376         atomic_pending_counts_[id] = max_pending;
377       }
378     }
379   }
380   }  // namespace tensorflow
```