quintinwang5 add DEVICE_DEFAULT for debug/stage ops History

10 contributors

323 lines (257 sloc) | 10.2 KB ...

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

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

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

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

#include <cstddef>
#include <deque>
#include <mutex>
#include <numeric>
#include <vector>

#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/resource_mgr.h"
#include "tensorflow/core/framework/tensor.h"
#include "tensorflow/core/framework/tensor_shape.h"
#include "tensorflow/core/lib/strings/strcat.h"
#include "tensorflow/core/platform/env.h"
#include "tensorflow/core/platform/mutex.h"

```

```
namespace tensorflow {
namespace {

class Buffer : public ResourceBase {
 public:
  using Tuple = std::vector<Tensor>;

  explicit Buffer(std::size_t capacity, std::size_t memory_limit)
      : capacity_(capacity), memory_limit_(memory_limit), current_bytes_(0) {}

  // the Buffer takes ownership of the Tuple
  Status Put(Tuple* tuple) {
    std::unique_lock<std::mutex> lock(mu_);

    std::size_t tuple_bytes = GetTupleBytes(*tuple);

    // Sanity check so that we don't block for ever below
    if (memory_limit_ > 0 && tuple_bytes > memory_limit_) {
      return Status(
          errors::ResourceExhausted("Attempted to insert "
                                    "tensors with combined size of '",
                                    tuple_bytes,
                                    "' bytes into "
                                    "Staging Area with a memory limit of '",
                                    memory_limit_, "'."));
    }

    // If buffer capacity is bounded wait until elements have been removed
    if (IsBounded()) {
      full_cond_var_.wait(lock, [tuple_bytes, this]() {
        // If there's a memory limit, check if there's space for insertion
        bool memory_limit_valid =
            memory_limit_ > 0 ? !WouldExceedMemoryLimit(tuple_bytes) : true;
        // If we're configured for capacity check if there's space for insertion
        bool capacity_valid = capacity_ > 0 ? !IsCapacityFull() : true;

        // Stop waiting upon success for both conditions
        return capacity_valid && memory_limit_valid;
      });
    }

    // Update bytes in the Staging Area
    current_bytes_ += tuple_bytes;

    // Store tuple
    buf_.push_back(std::move(*tuple));

    lock.unlock();
    // Notify all removers. Removers
```

```
    // may be peeking at a specific element or waiting
    // for the element at the front of the deque.
    // As we don't know the appropriate one to wake up
    // we should wake them all.
    non_empty_cond_var_.notify_all();

    return Status::OK();
  }

  // Get tuple at front of the buffer
  void Get(Tuple* tuple) {  // TODO(zhifengc): Support cancellation.
    std::unique_lock<std::mutex> lock(mu_);

    // Wait for data if the buffer is empty
    non_empty_cond_var_.wait(lock, [this]() { return !buf_.empty(); });

    // Move data into the output tuple
    *tuple = std::move(buf_.front());
    buf_.pop_front();

    // Update bytes in the Staging Area
    current_bytes_ -= GetTupleBytes(*tuple);

    notify_inserters_if_bounded(&lock);
  }

  // Return tuple at index
  Status Peek(std::size_t index, Tuple* tuple) {
    std::unique_lock<std::mutex> lock(mu_);

    // Wait if the requested index is not available
    non_empty_cond_var_.wait(
        lock, [index, this]() { return index < this->buf_.size(); });

    // Place tensors in the output tuple
    for (const auto& tensor : buf_[index]) {
      tuple->push_back(tensor);
    }

    return Status::OK();
  }

  // Buffer size
  size_t Size() {
    std::unique_lock<std::mutex> lock(mu_);
    return buf_.size();
  }

  void Clear() {
```

```
    std::unique_lock<std::mutex> lock(mu_);
    buf_.clear();
    current_bytes_ = 0;

    notify_inserters_if_bounded(&lock);
  }

  string DebugString() const override {
    std::unique_lock<std::mutex> lock(mu_);
    return strings::StrCat("Staging size: ", buf_.size());
  }

 private:
  // If the buffer is configured for bounded capacity, notify
  // waiting inserters that space is now available
  void notify_inserters_if_bounded(std::unique_lock<std::mutex>* lock) {
    if (IsBounded()) {
      lock->unlock();
      // Notify all inserters. The removal of an element
      // may make memory available for many inserters
      // to insert new elements
      full_cond_var_.notify_all();
    }
  }

  // Are there a limit number of elements or a memory limit
  // configured on this buffer?
  bool IsBounded() const { return capacity_ > 0 || memory_limit_ > 0; }

  bool IsCapacityFull() const { return buf_.size() >= capacity_; }

  bool WouldExceedMemoryLimit(std::size_t bytes) const {
    return bytes + current_bytes_ > memory_limit_;
  }

  std::size_t GetTupleBytes(const Tuple& tuple) {
    return std::accumulate(tuple.begin(), tuple.end(), 0,
                           [](const std::size_t& lhs, const Tensor& rhs) {
                             return lhs + rhs.TotalBytes();
                           });
  }

  std::size_t capacity_;
  std::size_t memory_limit_;
  std::size_t current_bytes_;
  mutable std::mutex mu_;
  std::condition_variable non_empty_cond_var_;
  std::condition_variable full_cond_var_;
  std::deque<Tuple> buf_;
```

```
};

Status GetBuffer(OpKernelContext* ctx, const NodeDef& ndef, Buffer** buf) {
  auto rm = ctx->resource_manager();
  ContainerInfo cinfo;

  // Lambda for creating the Staging Area
  auto create_fn = [&ndef](Buffer** ret) -> Status {
    int64_t capacity;
    int64_t memory_limit;
    TF_RETURN_IF_ERROR(GetNodeAttr(ndef, "capacity", &capacity));
    TF_RETURN_IF_ERROR(GetNodeAttr(ndef, "memory_limit", &memory_limit));
    *ret = new Buffer(capacity, memory_limit);
    return Status::OK();
  };

  TF_RETURN_IF_ERROR(cinfo.Init(rm, ndef, true /* use name() */));
  TF_RETURN_IF_ERROR(rm->LookupOrCreate<Buffer>(cinfo.container(), cinfo.name(),
                                                buf, create_fn));
  return Status::OK();
}

}  // namespace

class StageOp : public OpKernel {
 public:
  explicit StageOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  void Compute(OpKernelContext* ctx) override {
    Buffer* buf = nullptr;
    OP_REQUIRES_OK(ctx, GetBuffer(ctx, def(), &buf));
    core::ScopedUnref scope(buf);
    Buffer::Tuple tuple;
    tuple.reserve(ctx->num_inputs());
    for (int i = 0; i < ctx->num_inputs(); ++i) {
      tuple.push_back(ctx->input(i));
    }
    OP_REQUIRES_OK(ctx, buf->Put(&tuple));
  }
};

REGISTER_KERNEL_BUILDER(Name("Stage").Device(DEVICE_CPU), StageOp);
REGISTER_KERNEL_BUILDER(Name("Stage").Device(DEVICE_DEFAULT), StageOp);

class UnstageOp : public OpKernel {
 public:
  explicit UnstageOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  // Using this op in such a way that it blocks forever
```

```
  // is an error.  As such cancellation is not handled.
  void Compute(OpKernelContext* ctx) override {
    Buffer* buf = nullptr;
    OP_REQUIRES_OK(ctx, GetBuffer(ctx, def(), &buf));
    core::ScopedUnref scope(buf);
    Buffer::Tuple tuple;

    buf->Get(&tuple);

    OP_REQUIRES(
        ctx, tuple.size() == (size_t)ctx->num_outputs(),
        errors::InvalidArgument("Mismatch stage/unstage: ", tuple.size(),
                                " vs. ", ctx->num_outputs()));

    for (size_t i = 0; i < tuple.size(); ++i) {
      ctx->set_output(i, tuple[i]);
    }
  }
};

REGISTER_KERNEL_BUILDER(Name("Unstage").Device(DEVICE_CPU), UnstageOp);
REGISTER_KERNEL_BUILDER(Name("Unstage").Device(DEVICE_DEFAULT), UnstageOp);

class StagePeekOp : public OpKernel {
 public:
  explicit StagePeekOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  // Using this op in such a way that it blocks forever
  // is an error.  As such cancellation is not handled.
  void Compute(OpKernelContext* ctx) override {
    Buffer* buf = nullptr;
    OP_REQUIRES_OK(ctx, GetBuffer(ctx, def(), &buf));
    core::ScopedUnref scope(buf);
    Buffer::Tuple tuple;

    std::size_t index = ctx->input(0).scalar<int>()();

    OP_REQUIRES_OK(ctx, buf->Peek(index, &tuple));

    OP_REQUIRES(
        ctx, tuple.size() == (size_t)ctx->num_outputs(),
        errors::InvalidArgument("Mismatch stage/unstage: ", tuple.size(),
                                " vs. ", ctx->num_outputs()));

    for (size_t i = 0; i < tuple.size(); ++i) {
      ctx->set_output(i, tuple[i]);
    }
  }
};
```

```
275
276 REGISTER_KERNEL_BUILDER(Name("StagePeek").Device(DEVICE_CPU), StagePeekOp);
277 REGISTER_KERNEL_BUILDER(
278     Name("StagePeek").HostMemory("index").Device(DEVICE_DEFAULT), StagePeekOp);
279
280 class StageSizeOp : public OpKernel {
281  public:
282   explicit StageSizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
283
284   // Using this op in such a way that it blocks forever
285   // is an error.  As such cancellation is not handled.
286   void Compute(OpKernelContext* ctx) override {
287     Buffer* buf = nullptr;
288     OP_REQUIRES_OK(ctx, GetBuffer(ctx, def(), &buf));
289     core::ScopedUnref scope(buf);
290
291     // Allocate size output tensor
292     Tensor* size = nullptr;
293     OP_REQUIRES_OK(ctx, ctx->allocate_output(0, TensorShape({}), &size));
294
295     // Set it to the actual size
296     size->scalar<int32>().setConstant(buf->Size());
297   }
298 };
299
300 REGISTER_KERNEL_BUILDER(Name("StageSize").Device(DEVICE_CPU), StageSizeOp);
301 REGISTER_KERNEL_BUILDER(
302     Name("StageSize").HostMemory("size").Device(DEVICE_DEFAULT), StageSizeOp);
303
304 class StageClearOp : public OpKernel {
305  public:
306   explicit StageClearOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
307
308   // Using this op in such a way that it blocks forever
309   // is an error.  As such cancellation is not handled.
310   void Compute(OpKernelContext* ctx) override {
311     Buffer* buf = nullptr;
312     OP_REQUIRES_OK(ctx, GetBuffer(ctx, def(), &buf));
313     core::ScopedUnref scope(buf);
314
315     buf->Clear();
316   }
317 };
318
319 REGISTER_KERNEL_BUILDER(Name("StageClear").Device(DEVICE_CPU), StageClearOp);
320 REGISTER_KERNEL_BUILDER(Name("StageClear").Device(DEVICE_DEFAULT),
321                         StageClearOp);
322
323 }  // namespace tensorflow
```