jpienaar Rename to underlying type rather than alias ... ✓ History

7 contributors

808 lines (644 sloc) | 25.7 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 <functional>
#include <map>
#include <mutex>
#include <numeric>
#include <unordered_map>
#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/gtl/optional.h"
#include "tensorflow/core/lib/strings/strcat.h"
```

```cpp
#include "tensorflow/core/platform/env.h"
#include "tensorflow/core/platform/mutex.h"
#include "tensorflow/core/platform/thread_annotations.h"

namespace tensorflow {
namespace {

// Partial Ordering Comparator for Tensor keys containing scalar int64's
struct KeyTensorLess {
  bool operator()(const Tensor& lhs, const Tensor& rhs) const {
    return std::less<int64_t>{}(lhs.scalar<int64_t>()(),
                                rhs.scalar<int64_t>()());
  }
};

// Key Equality operator for Tensor keys containing scalar int64's
struct KeyTensorEqual {
  bool operator()(const Tensor& lhs, const Tensor& rhs) const {
    return std::equal_to<int64_t>{}(lhs.scalar<int64_t>()(),
                                    rhs.scalar<int64_t>()());
  }
};

// Hash for Tensor keys containing scalar int64's
struct KeyTensorHash {
  std::size_t operator()(const Tensor& key) const {
    return std::hash<int64_t>{}(key.scalar<int64_t>()());
  }
};

// Primary template.
template <bool Ordered, typename Data>
struct MapTraits;

// Partial specialization for ordered.
template <typename Data>
struct MapTraits<true, Data> {
  using KeyType = Tensor;
  using DataType = Data;
  using MapType = std::map<KeyType, Data, KeyTensorLess>;
};

// Partial specialization for unordered.
template <typename Data>
struct MapTraits<false, Data> {
  using KeyType = Tensor;
  using DataType = Data;
  using MapType =
      std::unordered_map<KeyType, Data, KeyTensorHash, KeyTensorEqual>;
```

```
};

// Wrapper around map/unordered_map.
template <bool Ordered>
class StagingMap : public ResourceBase {
 public:
  // Public typedefs
  using Tuple = std::vector<Tensor>;
  using OptionalTensor = gtl::optional<Tensor>;
  using OptionalTuple = std::vector<OptionalTensor>;

  using MapType = typename MapTraits<Ordered, OptionalTuple>::MapType;
  using KeyType = typename MapTraits<Ordered, OptionalTuple>::KeyType;

  using IncompleteType = typename MapTraits<false, OptionalTuple>::MapType;

 private:
  // Private variables
  DataTypeVector dtypes_ TF_GUARDED_BY(mu_);
  std::size_t capacity_ TF_GUARDED_BY(mu_);
  std::size_t memory_limit_ TF_GUARDED_BY(mu_);
  std::size_t current_bytes_ TF_GUARDED_BY(mu_);
  tensorflow::mutex mu_;
  tensorflow::condition_variable not_empty_;
  tensorflow::condition_variable full_;
  IncompleteType incomplete_ TF_GUARDED_BY(mu_);
  MapType map_ TF_GUARDED_BY(mu_);

 private:
  // private methods

  // If map is configured for bounded capacity, notify
  // waiting inserters that space is now available
  void notify_inserters_if_bounded() TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    if (has_capacity() || has_memory_limit()) {
      // Notify all inserters. The removal of an element
      // may make memory available for many inserters
      // to insert new elements
      full_.notify_all();
    }
  }

  // Notify all removers waiting to extract values
  // that data is now available
  void notify_removers() {
    // Notify all removers. This is because they are
    // waiting for specific keys to appear in the map
    // so we don't know which one to wake up.
    not_empty_.notify_all();
```

```
  }

  bool has_capacity() const TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    return capacity_ > 0;
  }

  bool has_memory_limit() const TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    return memory_limit_ > 0;
  }

  bool would_exceed_memory_limit(std::size_t bytes) const
      TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    return has_memory_limit() && bytes + current_bytes_ > memory_limit_;
  }

  bool is_capacity_full() const TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    return has_capacity() && map_.size() >= capacity_;
  }

  // Get number of bytes in the tuple
  std::size_t get_tuple_bytes(const Tuple& tuple) {
    return std::accumulate(tuple.begin(), tuple.end(),
                           static_cast<std::size_t>(0),
                           [](const std::size_t& lhs, const Tensor& rhs) {
                             return lhs + rhs.TotalBytes();
                           });
  }

  // Get number of bytes in the incomplete tuple
  std::size_t get_tuple_bytes(const OptionalTuple& tuple) {
    return std::accumulate(
        tuple.begin(), tuple.end(), static_cast<std::size_t>(0),
        [](const std::size_t& lhs, const OptionalTensor& rhs) {
          return (lhs + rhs.has_value()) ? rhs.value().TotalBytes() : 0;
        });
  }

  // Check that the index is within bounds
  Status check_index(const Tensor& key, std::size_t index)
      TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    if (index >= dtypes_.size()) {
      return Status(errors::InvalidArgument(
          "Index '", index, "' for key '", key.scalar<int64_t>()(),
          "' was out of bounds '", dtypes_.size(), "'."));
    }

    return Status::OK();
  }

```

```
177   Status copy_or_move_tensors(OptionalTuple* map_tuple, const Tensor& key,
178                               const Tensor& indices, Tuple* output,
179                               bool copy = false)
180       TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
181     auto findices = indices.flat<int>();
182
183     // Return values at specified indices
184     for (std::size_t i = 0; i < findices.dimension(0); ++i) {
185       std::size_t index = findices(i);
186
187       TF_RETURN_IF_ERROR(check_index(key, index));
188
189       // Insist on a value present at the specified index
190       if (!(*map_tuple)[index].has_value()) {
191         return Status(errors::InvalidArgument(
192             "Tensor at index '", index, "' for key '", key.scalar<int64_t>()(),
193             "' has already been removed."));
194       }
195
196       // Copy the contained tensor and
197       // remove from the OptionalTuple
198       output->push_back((*map_tuple)[index].value());
199
200       // Clear out the entry if we're not copying (moving)
201       if (!copy) {
202         (*map_tuple)[index].reset();
203       }
204     }
205
206     return Status::OK();
207   }
208
209   // Check that the optional value at the specified index
210   // is uninitialized
211   Status check_index_uninitialized(const Tensor& key, std::size_t index,
212                                    const OptionalTuple& tuple)
213       TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
214     if (tuple[index].has_value()) {
215       return errors::InvalidArgument("The tensor for index '", index,
216                                      "' for key '", key.scalar<int64_t>()(),
217                                      "' was already initialized '",
218                                      dtypes_.size(), "'.");
219     }
220
221     return Status::OK();
222   }
223
224   // Check that the indices are strictly ordered
225   Status check_index_ordering(const Tensor& indices) {
```

```
    if (indices.NumElements() == 0) {
      return errors::InvalidArgument("Indices are empty");
    }

    auto findices = indices.flat<int>();

    for (std::size_t i = 0; i < findices.dimension(0) - 1; ++i) {
      if (findices(i) < findices(i + 1)) {
        continue;
      }

      return errors::InvalidArgument("Indices are not strictly ordered");
    }

    return Status::OK();
  }

  // Check bytes are within memory limits memory limits
  Status check_memory_limit(std::size_t bytes)
      TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    if (has_memory_limit() && bytes > memory_limit_) {
      return errors::ResourceExhausted(
          "Attempted to insert tensors with combined size of '", bytes,
          "' bytes into Staging Area with a memory limit of '", memory_limit_,
          "'.");
    }

    return Status::OK();
  }

  // Insert incomplete data into the Barrier
  Status put_incomplete(const KeyType& key, const Tensor& indices,
                        OptionalTuple* tuple, tensorflow::mutex_lock* lock)
      TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    auto findices = indices.flat<int>();

    // Search for the key in our incomplete set
    auto it = incomplete_.find(key);

    // Check that the tuple fits within the memory limit
    std::size_t tuple_bytes = get_tuple_bytes(*tuple);
    TF_RETURN_IF_ERROR(check_memory_limit(tuple_bytes));

    // Wait until we don't exceed the memory limit
    while (would_exceed_memory_limit(tuple_bytes)) {
      full_.wait(*lock);
    }

    // This key isn't present in the incomplete set
```

```
    // Create OptionalTuple and insert
    if (it == incomplete_.end()) {
      OptionalTuple empty(dtypes_.size());

      // Initialize empty tuple with given dta
      for (std::size_t i = 0; i < findices.dimension(0); ++i) {
        std::size_t index = findices(i);
        TF_RETURN_IF_ERROR(check_index(key, index));

        // Assign tuple at this index
        empty[index] = std::move((*tuple)[i]);
      }

      // Insert into incomplete map
      incomplete_.insert({key, std::move(empty)});

      // Increment size
      current_bytes_ += tuple_bytes;
    }
    // Found an entry in the incomplete index
    // Update with given data and insert complete entries
    // into the main map
    else {
      // Reference existing incomplete tuple
      OptionalTuple& present = it->second;

      // Assign given data
      for (std::size_t i = 0; i < findices.dimension(0); ++i) {
        std::size_t index = findices(i);
        TF_RETURN_IF_ERROR(check_index(key, index));
        TF_RETURN_IF_ERROR(check_index_uninitialized(key, index, present));

        // Assign tuple at this index
        present[index] = std::move((*tuple)[i]);
      }

      // Increment size
      current_bytes_ += tuple_bytes;

      // Do we have values at all tuple elements?
      bool complete =
          std::all_of(present.begin(), present.end(),
                      [](const OptionalTensor& v) { return v.has_value(); });

      // If so, put the tuple in the actual map
      if (complete) {
        OptionalTuple insert_tuple = std::move(it->second);

        // Remove from incomplete
```

```
        incomplete_.erase(it);

        TF_RETURN_IF_ERROR(put_complete(key, &insert_tuple));
      }
    }

    return Status::OK();
  }

  // Does the insertion into the actual staging area
  Status put_complete(const KeyType& key, OptionalTuple* tuple)
      TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
    // Insert key and tuples into the map
    map_.insert({key, std::move(*tuple)});

    notify_removers();

    return Status::OK();
  }

 public:
  // public methods
  explicit StagingMap(const DataTypeVector& dtypes, std::size_t capacity,
                      std::size_t memory_limit)
      : dtypes_(dtypes),
        capacity_(capacity),
        memory_limit_(memory_limit),
        current_bytes_(0) {}

  Status put(KeyType* key, const Tensor* indices, OptionalTuple* tuple) {
    tensorflow::mutex_lock lock(mu_);

    // Sanity check the indices
    TF_RETURN_IF_ERROR(check_index_ordering(*indices));

    // Handle incomplete inserts
    if (indices->NumElements() != dtypes_.size()) {
      return put_incomplete(*key, *indices, tuple, &lock);
    }

    std::size_t tuple_bytes = get_tuple_bytes(*tuple);
    // Check that tuple_bytes fits within the memory limit
    TF_RETURN_IF_ERROR(check_memory_limit(tuple_bytes));

    // Wait until there's space for insertion.
    while (would_exceed_memory_limit(tuple_bytes) || is_capacity_full()) {
      full_.wait(lock);
    }

```

```
373       // Do the put operation
374       TF_RETURN_IF_ERROR(put_complete(*key, tuple));
375
376       // Update the current size
377       current_bytes_ += tuple_bytes;
378
379       return Status::OK();
380     }
381
382     Status get(const KeyType* key, const Tensor* indices, Tuple* tuple) {
383       tensorflow::mutex_lock lock(mu_);
384
385       // Sanity check the indices
386       TF_RETURN_IF_ERROR(check_index_ordering(*indices));
387
388       typename MapType::iterator it;
389
390       // Wait until the element with the requested key is present
391       while ((it = map_.find(*key)) == map_.end()) {
392         not_empty_.wait(lock);
393       }
394
395       TF_RETURN_IF_ERROR(
396           copy_or_move_tensors(&it->second, *key, *indices, tuple, true));
397
398       // Update bytes in the Staging Area
399       current_bytes_ -= get_tuple_bytes(*tuple);
400
401       return Status::OK();
402     }
403
404     Status pop(const KeyType* key, const Tensor* indices, Tuple* tuple) {
405       tensorflow::mutex_lock lock(mu_);
406
407       // Sanity check the indices
408       TF_RETURN_IF_ERROR(check_index_ordering(*indices));
409
410       typename MapType::iterator it;
411
412       // Wait until the element with the requested key is present
413       while ((it = map_.find(*key)) == map_.end()) {
414         not_empty_.wait(lock);
415       }
416
417       TF_RETURN_IF_ERROR(
418           copy_or_move_tensors(&it->second, *key, *indices, tuple));
419
420       // Remove entry if all the values have been consumed
421       if (!std::any_of(
```

```
422             it->second.begin(), it->second.end(),
423             [](const OptionalTensor& tensor) { return tensor.has_value(); })) {
424       map_.erase(it);
425     }
426
427     // Update bytes in the Staging Area
428     current_bytes_ -= get_tuple_bytes(*tuple);
429
430     notify_inserters_if_bounded();
431
432     return Status::OK();
433   }
434
435   Status popitem(KeyType* key, const Tensor* indices, Tuple* tuple) {
436     tensorflow::mutex_lock lock(mu_);
437
438     // Sanity check the indices
439     TF_RETURN_IF_ERROR(check_index_ordering(*indices));
440
441     // Wait until map is not empty
442     while (this->map_.empty()) {
443       not_empty_.wait(lock);
444     }
445
446     // Move from the first element and erase it
447
448     auto it = map_.begin();
449
450     TF_RETURN_IF_ERROR(
451         copy_or_move_tensors(&it->second, *key, *indices, tuple));
452
453     *key = it->first;
454
455     // Remove entry if all the values have been consumed
456     if (!std::any_of(
457             it->second.begin(), it->second.end(),
458             [](const OptionalTensor& tensor) { return tensor.has_value(); })) {
459       map_.erase(it);
460     }
461
462     // Update bytes in the Staging Area
463     current_bytes_ -= get_tuple_bytes(*tuple);
464
465     notify_inserters_if_bounded();
466
467     return Status::OK();
468   }
469
470   Status clear() {
```

```
471       tensorflow::mutex_lock lock(mu_);
472       map_.clear();
473       incomplete_.clear();
474       current_bytes_ = 0;
475
476       notify_inserters_if_bounded();
477
478       return Status::OK();
479     }
480
481     std::size_t incomplete_size() {
482       tensorflow::mutex_lock lock(mu_);
483       return incomplete_.size();
484     }
485
486     std::size_t size() {
487       tensorflow::mutex_lock lock(mu_);
488       return map_.size();
489     }
490
491     string DebugString() const override { return "StagingMap"; }
492   };
493
494   template <bool Ordered>
495   Status GetStagingMap(OpKernelContext* ctx, const NodeDef& ndef,
496                        StagingMap<Ordered>** map) {
497     auto rm = ctx->resource_manager();
498     ContainerInfo cinfo;
499
500     // Lambda for creating the Staging Area
501     auto create_fn = [&ndef](StagingMap<Ordered>** ret) -> Status {
502       DataTypeVector dtypes;
503       int64_t capacity;
504       int64_t memory_limit;
505       TF_RETURN_IF_ERROR(GetNodeAttr(ndef, "dtypes", &dtypes));
506       TF_RETURN_IF_ERROR(GetNodeAttr(ndef, "capacity", &capacity));
507       TF_RETURN_IF_ERROR(GetNodeAttr(ndef, "memory_limit", &memory_limit));
508       *ret = new StagingMap<Ordered>(dtypes, capacity, memory_limit);
509       return Status::OK();
510     };
511
512     TF_RETURN_IF_ERROR(cinfo.Init(rm, ndef, true /* use name() */));
513     TF_RETURN_IF_ERROR(rm->LookupOrCreate<StagingMap<Ordered>>(
514         cinfo.container(), cinfo.name(), map, create_fn));
515     return Status::OK();
516   }
517
518   template <bool Ordered>
519   class MapStageOp : public OpKernel {
```

```
 public:
  explicit MapStageOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  void Compute(OpKernelContext* ctx) override {
    StagingMap<Ordered>* map = nullptr;
    OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
    core::ScopedUnref scope(map);
    typename StagingMap<Ordered>::OptionalTuple tuple;

    const Tensor* key_tensor;
    const Tensor* indices_tensor;
    OpInputList values_tensor;

    OP_REQUIRES_OK(ctx, ctx->input("key", &key_tensor));
    OP_REQUIRES_OK(ctx, ctx->input("indices", &indices_tensor));
    OP_REQUIRES_OK(ctx, ctx->input_list("values", &values_tensor));
    OP_REQUIRES(ctx, key_tensor->NumElements() > 0,
                errors::InvalidArgument("key must not be empty"));

    // Create copy for insertion into Staging Area
    Tensor key(*key_tensor);

    // Create the tuple to store
    for (std::size_t i = 0; i < values_tensor.size(); ++i) {
      tuple.push_back(values_tensor[i]);
    }

    // Store the tuple in the map
    OP_REQUIRES_OK(ctx, map->put(&key, indices_tensor, &tuple));
  }
};

REGISTER_KERNEL_BUILDER(Name("MapStage").Device(DEVICE_CPU), MapStageOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapStage").Device(DEVICE_CPU),
                        MapStageOp<true>);

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
REGISTER_KERNEL_BUILDER(
    Name("MapStage").HostMemory("key").HostMemory("indices").Device(DEVICE_GPU),
    MapStageOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapStage")
                            .HostMemory("key")
                            .HostMemory("indices")
                            .Device(DEVICE_GPU),
                        MapStageOp<true>);
#endif  // GOOGLE_CUDA || TENSORFLOW_USE_ROCM


template <bool Ordered>
```

```
class MapUnstageOp : public OpKernel {
 public:
  explicit MapUnstageOp(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 {
    StagingMap<Ordered>* map = nullptr;
    OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
    core::ScopedUnref scope(map);
    typename StagingMap<Ordered>::Tuple tuple;

    const Tensor* key_tensor;
    const Tensor* indices_tensor;

    OP_REQUIRES_OK(ctx, ctx->input("key", &key_tensor));
    OP_REQUIRES_OK(ctx, ctx->input("indices", &indices_tensor));
    OP_REQUIRES_OK(ctx, map->pop(key_tensor, indices_tensor, &tuple));

    OP_REQUIRES(
        ctx, tuple.size() == indices_tensor->NumElements(),
        errors::InvalidArgument("output/indices size mismatch: ", tuple.size(),
                                " vs. ", indices_tensor->NumElements()));

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

REGISTER_KERNEL_BUILDER(Name("MapUnstage").Device(DEVICE_CPU),
                        MapUnstageOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstage").Device(DEVICE_CPU),
                        MapUnstageOp<true>);

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
REGISTER_KERNEL_BUILDER(Name("MapUnstage")
                            .HostMemory("key")
                            .HostMemory("indices")
                            .Device(DEVICE_GPU),
                        MapUnstageOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstage")
                            .HostMemory("key")
                            .HostMemory("indices")
                            .Device(DEVICE_GPU),
                        MapUnstageOp<true>);
#endif

template <bool Ordered>
```

```
618 class MapPeekOp : public OpKernel {
619  public:
620   explicit MapPeekOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
621
622   // Using this op in such a way that it blocks forever
623   // is an error.  As such cancellation is not handled.
624   void Compute(OpKernelContext* ctx) override {
625     StagingMap<Ordered>* map = nullptr;
626     OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
627     core::ScopedUnref scope(map);
628     typename StagingMap<Ordered>::Tuple tuple;
629
630     const Tensor* key_tensor;
631     const Tensor* indices_tensor;
632
633     OP_REQUIRES_OK(ctx, ctx->input("key", &key_tensor));
634     OP_REQUIRES_OK(ctx, ctx->input("indices", &indices_tensor));
635     OP_REQUIRES_OK(ctx, map->get(key_tensor, indices_tensor, &tuple));
636
637     OP_REQUIRES(
638         ctx, tuple.size() == indices_tensor->NumElements(),
639         errors::InvalidArgument("output/indices size mismatch: ", tuple.size(),
640                                 " vs. ", indices_tensor->NumElements()));
641
642     for (std::size_t i = 0; i < tuple.size(); ++i) {
643       ctx->set_output(i, tuple[i]);
644     }
645   }
646 };
647
648 REGISTER_KERNEL_BUILDER(Name("MapPeek").Device(DEVICE_CPU), MapPeekOp<false>);
649 REGISTER_KERNEL_BUILDER(Name("OrderedMapPeek").Device(DEVICE_CPU),
650                         MapPeekOp<true>);
651
652 #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
653 REGISTER_KERNEL_BUILDER(
654     Name("MapPeek").HostMemory("key").HostMemory("indices").Device(DEVICE_GPU),
655     MapPeekOp<false>);
656 REGISTER_KERNEL_BUILDER(Name("OrderedMapPeek")
657                             .HostMemory("key")
658                             .HostMemory("indices")
659                             .Device(DEVICE_GPU),
660                         MapPeekOp<true>);
661 #endif
662
663
664 template <bool Ordered>
665 class MapUnstageNoKeyOp : public OpKernel {
666  public:
```

```
  explicit MapUnstageNoKeyOp(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 {
    StagingMap<Ordered>* map = nullptr;
    OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
    core::ScopedUnref scope(map);

    // Pop a random (key, value) off the map
    typename StagingMap<Ordered>::KeyType key;
    typename StagingMap<Ordered>::Tuple tuple;

    const Tensor* indices_tensor;

    OP_REQUIRES_OK(ctx, ctx->input("indices", &indices_tensor));
    OP_REQUIRES_OK(ctx, map->popitem(&key, indices_tensor, &tuple));

    // Allocate a key tensor and assign the key as the first output
    ctx->set_output(0, key);

    // Set the rest of the outputs to the tuple Tensors
    OP_REQUIRES(
        ctx, tuple.size() == indices_tensor->NumElements(),
        errors::InvalidArgument("output/indices size mismatch: ", tuple.size(),
                                " vs. ", indices_tensor->NumElements()));

    for (std::size_t i = 0; i < tuple.size(); ++i) {
      ctx->set_output(i + 1, tuple[i]);
    }
  }
};

REGISTER_KERNEL_BUILDER(Name("MapUnstageNoKey").Device(DEVICE_CPU),
                        MapUnstageNoKeyOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstageNoKey").Device(DEVICE_CPU),
                        MapUnstageNoKeyOp<true>);

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
REGISTER_KERNEL_BUILDER(Name("MapUnstageNoKey")
                            .HostMemory("key")
                            .HostMemory("indices")
                            .Device(DEVICE_GPU),
                        MapUnstageNoKeyOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstageNoKey")
                            .HostMemory("key")
                            .HostMemory("indices")
                            .Device(DEVICE_GPU),
                        MapUnstageNoKeyOp<true>);
```

```
#endif


template <bool Ordered>
class MapSizeOp : public OpKernel {
 public:
  explicit MapSizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  void Compute(OpKernelContext* ctx) override {
    StagingMap<Ordered>* map = nullptr;
    OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
    core::ScopedUnref scope(map);

    // Allocate size output tensor
    Tensor* size = nullptr;
    OP_REQUIRES_OK(ctx, ctx->allocate_output(0, TensorShape({}), &size));

    // Set it to the actual size
    size->scalar<int32>().setConstant(map->size());
  }
};

REGISTER_KERNEL_BUILDER(Name("MapSize").Device(DEVICE_CPU), MapSizeOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapSize").Device(DEVICE_CPU),
                        MapSizeOp<true>);

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
REGISTER_KERNEL_BUILDER(Name("MapSize").Device(DEVICE_GPU).HostMemory("size"),
                        MapSizeOp<false>);
REGISTER_KERNEL_BUILDER(
    Name("OrderedMapSize").Device(DEVICE_GPU).HostMemory("size"),
    MapSizeOp<true>);
#endif

template <bool Ordered>
class MapIncompleteSizeOp : public OpKernel {
 public:
  explicit MapIncompleteSizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  void Compute(OpKernelContext* ctx) override {
    StagingMap<Ordered>* map = nullptr;
    OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
    core::ScopedUnref scope(map);

    // Allocate size output tensor
    Tensor* size = nullptr;
    OP_REQUIRES_OK(ctx, ctx->allocate_output(0, TensorShape({}), &size));

    // Set it to the actual size
```

```
    size->scalar<int32>().setConstant(map->incomplete_size());
  }
};

REGISTER_KERNEL_BUILDER(Name("MapIncompleteSize").Device(DEVICE_CPU),
                        MapIncompleteSizeOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapIncompleteSize").Device(DEVICE_CPU),
                        MapIncompleteSizeOp<true>);

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
REGISTER_KERNEL_BUILDER(
    Name("MapIncompleteSize").Device(DEVICE_GPU).HostMemory("size"),
    MapIncompleteSizeOp<false>);
REGISTER_KERNEL_BUILDER(
    Name("OrderedMapIncompleteSize").Device(DEVICE_GPU).HostMemory("size"),
    MapIncompleteSizeOp<true>);
#endif

template <bool Ordered>
class MapClearOp : public OpKernel {
 public:
  explicit MapClearOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  void Compute(OpKernelContext* ctx) override {
    StagingMap<Ordered>* map = nullptr;
    OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
    core::ScopedUnref scope(map);

    OP_REQUIRES_OK(ctx, map->clear());
  }
};

REGISTER_KERNEL_BUILDER(Name("MapClear").Device(DEVICE_CPU), MapClearOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapClear").Device(DEVICE_CPU),
                        MapClearOp<true>);

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
REGISTER_KERNEL_BUILDER(Name("MapClear").Device(DEVICE_GPU), MapClearOp<false>);
REGISTER_KERNEL_BUILDER(Name("OrderedMapClear").Device(DEVICE_GPU),
                        MapClearOp<true>);
#endif

}  // namespace
}  // namespace tensorflow
```