```
☐ tensorflow / tensorflow (Public)
<> Code
            Issues 2.1k  Pull requests 283
                                                      Actions Projects 1
  ጕ 5100e359ae ▼
tensorflow / tensorflow / core / kernels / map_stage_op.cc
      jpienaar Rename to underlying type rather than alias ... \checkmark
                                                                                        ( History
 At 7 contributors 😭 🌑 🚇 😥 <equation-block>
  808 lines (644 sloc) | 25.7 KB
        /* Copyright 2017 The TensorFlow Authors. All Rights Reserved.
    2
        Licensed under the Apache License, Version 2.0 (the "License");
    3
        you may not use this file except in compliance with the License.
        You may obtain a copy of the License at
    5
    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,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   11
        See the License for the specific language governing permissions and
   12
        limitations under the License.
   14
        15
        #include <cstddef>
   16
        #include <functional>
   17
   18
        #include <map>
        #include <mutex>
   19
   20
        #include <numeric>
        #include <unordered_map>
   21
        #include <vector>
   22
   23
        #include "tensorflow/core/framework/op_kernel.h"
   24
        #include "tensorflow/core/framework/resource_mgr.h"
   25
        #include "tensorflow/core/framework/tensor.h"
        #include "tensorflow/core/framework/tensor_shape.h"
   27
        #include "tensorflow/core/lib/gtl/optional.h"
   28
   29
        #include "tensorflow/core/lib/strings/strcat.h"
```

```
30
     #include "tensorflow/core/platform/env.h"
     #include "tensorflow/core/platform/mutex.h"
31
32
     #include "tensorflow/core/platform/thread annotations.h"
33
34
     namespace tensorflow {
35
     namespace {
36
37
     // Partial Ordering Comparator for Tensor keys containing scalar int64's
38
     struct KeyTensorLess {
       bool operator()(const Tensor& lhs, const Tensor& rhs) const {
39
40
         return std::less<int64_t>{}(lhs.scalar<int64_t>()(),
                                      rhs.scalar<int64 t>()());
41
       }
42
43
     };
44
     // Key Equality operator for Tensor keys containing scalar int64's
45
     struct KeyTensorEqual {
46
       bool operator()(const Tensor& lhs, const Tensor& rhs) const {
47
         return std::equal_to<int64_t>{}(lhs.scalar<int64_t>()(),
48
                                          rhs.scalar<int64_t>()());
49
50
       }
51
     };
52
     // Hash for Tensor keys containing scalar int64's
53
54
     struct KeyTensorHash {
       std::size_t operator()(const Tensor& key) const {
55
         return std::hash<int64_t>{}(key.scalar<int64_t>()());
56
57
       }
58
     };
59
60
     // Primary template.
     template <bool Ordered, typename Data>
61
     struct MapTraits;
62
63
64
     // Partial specialization for ordered.
65
     template <typename Data>
     struct MapTraits<true, Data> {
66
67
       using KeyType = Tensor;
68
       using DataType = Data;
       using MapType = std::map<KeyType, Data, KeyTensorLess>;
69
70
     };
71
72
     // Partial specialization for unordered.
73
     template <typename Data>
74
     struct MapTraits<false, Data> {
75
       using KeyType = Tensor;
       using DataType = Data;
76
77
       using MapType =
78
           std::unordered_map<KeyType, Data, KeyTensorHash, KeyTensorEqual>;
```

```
79
      };
80
81
      // Wrapper around map/unordered map.
82
      template <bool Ordered>
83
      class StagingMap : public ResourceBase {
84
       public:
85
        // Public typedefs
        using Tuple = std::vector<Tensor>;
86
87
        using OptionalTensor = gtl::optional<Tensor>;
        using OptionalTuple = std::vector<OptionalTensor>;
88
89
90
        using MapType = typename MapTraits<Ordered, OptionalTuple>::MapType;
        using KeyType = typename MapTraits<Ordered, OptionalTuple>::KeyType;
91
92
93
        using IncompleteType = typename MapTraits<false, OptionalTuple>::MapType;
94
       private:
95
        // Private variables
96
        DataTypeVector dtypes_ TF_GUARDED_BY(mu_);
97
        std::size t capacity TF GUARDED BY(mu );
98
99
        std::size_t memory_limit_ TF_GUARDED_BY(mu_);
100
        std::size t current bytes TF GUARDED BY(mu );
101
        tensorflow::mutex mu;
102
        tensorflow::condition_variable not_empty_;
103
        tensorflow::condition_variable full_;
        IncompleteType incomplete_ TF_GUARDED_BY(mu_);
104
        MapType map_ TF_GUARDED_BY(mu_);
105
106
107
       private:
108
        // private methods
109
        // If map is configured for bounded capacity, notify
110
111
        // waiting inserters that space is now available
        void notify_inserters_if_bounded() TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
112
          if (has capacity() || has memory limit()) {
113
            // Notify all inserters. The removal of an element
114
115
            // may make memory available for many inserters
116
            // to insert new elements
            full_.notify_all();
117
118
          }
119
        }
120
121
        // Notify all removers waiting to extract values
122
        // that data is now available
        void notify_removers() {
123
          // Notify all removers. This is because they are
124
125
          // waiting for specific keys to appear in the map
          // so we don't know which one to wake up.
126
          not_empty_.notify_all();
127
```

```
128
        }
129
130
        bool has_capacity() const TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
131
          return capacity_ > 0;
132
        }
133
134
        bool has_memory_limit() const TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
          return memory_limit_ > 0;
135
136
        }
137
138
        bool would_exceed_memory_limit(std::size_t bytes) const
139
            TF EXCLUSIVE LOCKS REQUIRED(mu ) {
          return has memory limit() && bytes + current bytes > memory limit;
140
        }
141
142
        bool is capacity full() const TF EXCLUSIVE LOCKS REQUIRED(mu ) {
143
144
          return has_capacity() && map_.size() >= capacity_;
145
        }
146
147
        // Get number of bytes in the tuple
        std::size_t get_tuple_bytes(const Tuple& tuple) {
148
149
          return std::accumulate(tuple.begin(), tuple.end(),
                                  static_cast<std::size_t>(0),
150
                                  [](const std::size_t& lhs, const Tensor& rhs) {
151
152
                                    return lhs + rhs.TotalBytes();
153
                                  });
154
        }
155
        // Get number of bytes in the incomplete tuple
156
        std::size_t get_tuple_bytes(const OptionalTuple& tuple) {
157
          return std::accumulate(
158
              tuple.begin(), tuple.end(), static_cast<std::size_t>(0),
159
              [](const std::size_t& lhs, const OptionalTensor& rhs) {
160
                return (lhs + rhs.has_value()) ? rhs.value().TotalBytes() : 0;
161
162
              });
        }
163
164
165
        // Check that the index is within bounds
        Status check_index(const Tensor& key, std::size_t index)
166
167
            TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
168
          if (index >= dtypes .size()) {
            return Status(errors::InvalidArgument(
169
                "Index '", index, "' for key '", key.scalar<int64_t>()(),
170
171
                "' was out of bounds '", dtypes_.size(), "'."));
172
          }
173
174
          return Status::OK();
175
        }
176
```

```
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
          // Return values at specified indices
183
          for (std::size_t i = 0; i < findices.dimension(0); ++i) {</pre>
184
            std::size t index = findices(i);
185
186
187
            TF_RETURN_IF_ERROR(check_index(key, index));
188
            // Insist on a value present at the specified index
189
190
            if (!(*map_tuple)[index].has_value()) {
191
              return Status(errors::InvalidArgument(
                   "Tensor at index '", index, "' for key '", key.scalar<int64 t>()(),
192
                   "' has already been removed."));
193
194
            }
195
            // Copy the contained tensor and
196
197
            // remove from the OptionalTuple
198
            output->push_back((*map_tuple)[index].value());
199
            // Clear out the entry if we're not copying (moving)
200
201
            if (!copy) {
              (*map_tuple)[index].reset();
202
            }
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,
                                          const OptionalTuple& tuple)
212
213
            TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
214
          if (tuple[index].has value()) {
            return errors::InvalidArgument("The tensor for index '", index,
215
                                             "' for key '", key.scalar<int64_t>()(),
216
                                            "' was already initialized '",
217
                                            dtypes_.size(), "'.");
218
          }
219
220
221
          return Status::OK();
222
        }
223
224
        // Check that the indices are strictly ordered
225
        Status check_index_ordering(const Tensor& indices) {
```

```
226
          if (indices.NumElements() == 0) {
227
            return errors::InvalidArgument("Indices are empty");
228
          }
229
          auto findices = indices.flat<int>();
230
231
          for (std::size_t i = 0; i < findices.dimension(0) - 1; ++i) {</pre>
232
            if (findices(i) < findices(i + 1)) {</pre>
233
              continue:
234
            }
235
236
            return errors::InvalidArgument("Indices are not strictly ordered");
237
238
          }
239
240
          return Status::OK();
241
        }
242
243
        // Check bytes are within memory limits memory limits
        Status check memory limit(std::size t bytes)
244
            TF EXCLUSIVE LOCKS REQUIRED(mu ) {
245
246
          if (has_memory_limit() && bytes > memory_limit_) {
247
            return errors::ResourceExhausted(
                 "Attempted to insert tensors with combined size of '", bytes,
248
                 "' bytes into Staging Area with a memory limit of '", memory_limit_,
249
250
251
          }
252
253
          return Status::OK();
254
        }
255
        // Insert incomplete data into the Barrier
256
        Status put_incomplete(const KeyType& key, const Tensor& indices,
257
258
                               OptionalTuple* tuple, tensorflow::mutex_lock* lock)
            TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
259
260
          auto findices = indices.flat<int>();
261
262
          // Search for the key in our incomplete set
263
          auto it = incomplete .find(key);
264
265
          // Check that the tuple fits within the memory limit
266
          std::size_t tuple_bytes = get_tuple_bytes(*tuple);
          TF_RETURN_IF_ERROR(check_memory_limit(tuple_bytes));
267
268
269
          // Wait until we don't exceed the memory limit
          while (would_exceed_memory_limit(tuple_bytes)) {
270
271
            full_.wait(*lock);
          }
272
273
274
          // This key isn't present in the incomplete set
```

```
275
          // Create OptionalTuple and insert
276
          if (it == incomplete_.end()) {
277
            OptionalTuple empty(dtypes_.size());
278
279
            // Initialize empty tuple with given dta
            for (std::size_t i = 0; i < findices.dimension(0); ++i) {</pre>
280
               std::size_t index = findices(i);
281
              TF_RETURN_IF_ERROR(check_index(key, index));
282
283
284
              // Assign tuple at this index
285
              empty[index] = std::move((*tuple)[i]);
            }
286
287
            // Insert into incomplete map
288
289
            incomplete_.insert({key, std::move(empty)});
290
291
            // Increment size
292
            current_bytes_ += tuple_bytes;
293
          }
294
          // Found an entry in the incomplete index
295
          // Update with given data and insert complete entries
296
          // into the main map
          else {
297
            // Reference existing incomplete tuple
298
299
            OptionalTuple& present = it->second;
300
301
            // Assign given data
            for (std::size t i = 0; i < findices.dimension(0); ++i) {</pre>
302
              std::size_t index = findices(i);
303
304
              TF_RETURN_IF_ERROR(check_index(key, index));
              TF_RETURN_IF_ERROR(check_index_uninitialized(key, index, present));
305
306
307
              // Assign tuple at this index
308
              present[index] = std::move((*tuple)[i]);
309
            }
310
311
            // Increment size
312
            current_bytes_ += tuple_bytes;
313
            // Do we have values at all tuple elements?
314
315
            bool complete =
316
                std::all_of(present.begin(), present.end(),
                             [](const OptionalTensor& v) { return v.has_value(); });
317
318
319
            // If so, put the tuple in the actual map
            if (complete) {
320
              OptionalTuple insert_tuple = std::move(it->second);
321
322
323
              // Remove from incomplete
```

```
324
               incomplete_.erase(it);
325
326
              TF_RETURN_IF_ERROR(put_complete(key, &insert_tuple));
327
            }
328
          }
329
          return Status::OK();
330
        }
331
332
333
        // Does the insertion into the actual staging area
334
        Status put_complete(const KeyType& key, OptionalTuple* tuple)
335
            TF EXCLUSIVE LOCKS REQUIRED(mu ) {
          // Insert key and tuples into the map
336
          map_.insert({key, std::move(*tuple)});
337
338
339
          notify removers();
340
341
          return Status::OK();
342
        }
343
       public:
344
345
        // public methods
        explicit StagingMap(const DataTypeVector& dtypes, std::size_t capacity,
346
                             std::size_t memory_limit)
347
348
             : dtypes_(dtypes),
               capacity_(capacity),
349
              memory_limit_(memory_limit),
350
351
              current_bytes_(0) {}
352
        Status put(KeyType* key, const Tensor* indices, OptionalTuple* tuple) {
353
          tensorflow::mutex_lock lock(mu_);
354
355
356
          // Sanity check the indices
          TF_RETURN_IF_ERROR(check_index_ordering(*indices));
357
358
359
          // Handle incomplete inserts
360
          if (indices->NumElements() != dtypes_.size()) {
            return put_incomplete(*key, *indices, tuple, &lock);
361
          }
362
363
364
          std::size_t tuple_bytes = get_tuple_bytes(*tuple);
          // Check that tuple_bytes fits within the memory limit
365
366
          TF_RETURN_IF_ERROR(check_memory_limit(tuple_bytes));
367
          // Wait until there's space for insertion.
368
369
          while (would_exceed_memory_limit(tuple_bytes) || is_capacity_full()) {
370
            full_.wait(lock);
371
          }
372
```

```
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
          // Wait until the element with the requested key is present
390
          while ((it = map .find(*key)) == map .end()) {
391
            not_empty_.wait(lock);
392
393
          }
394
395
          TF RETURN IF ERROR(
396
              copy_or_move_tensors(&it->second, *key, *indices, tuple, true));
397
          // Update bytes in the Staging Area
398
          current_bytes_ -= get_tuple_bytes(*tuple);
399
400
401
          return Status::OK();
402
        }
403
        Status pop(const KeyType* key, const Tensor* indices, Tuple* tuple) {
404
          tensorflow::mutex_lock lock(mu_);
405
406
407
          // Sanity check the indices
          TF_RETURN_IF_ERROR(check_index_ordering(*indices));
408
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
          // Remove entry if all the values have been consumed
420
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
          current_bytes_ -= get_tuple_bytes(*tuple);
428
429
          notify inserters if bounded();
430
431
432
          return Status::OK();
        }
433
434
        Status popitem(KeyType* key, const Tensor* indices, Tuple* tuple) {
435
436
          tensorflow::mutex_lock lock(mu_);
437
          // Sanity check the indices
438
          TF_RETURN_IF_ERROR(check_index_ordering(*indices));
439
440
441
          // Wait until map is not empty
442
          while (this->map_.empty()) {
443
            not_empty_.wait(lock);
444
          }
445
          // Move from the first element and erase it
446
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(
                  it->second.begin(), it->second.end(),
457
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
          notify_inserters_if_bounded();
465
466
          return Status::OK();
467
        }
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() {
          tensorflow::mutex lock lock(mu );
482
          return incomplete .size();
483
        }
484
485
486
        std::size t size() {
          tensorflow::mutex_lock lock(mu_);
487
488
          return map_.size();
489
        }
490
491
        string DebugString() const override { return "StagingMap"; }
492
      };
493
      template <bool Ordered>
494
495
      Status GetStagingMap(OpKernelContext* ctx, const NodeDef& ndef,
                            StagingMap<Ordered>** map) {
496
497
        auto rm = ctx->resource_manager();
        ContainerInfo cinfo;
498
499
        // Lambda for creating the Staging Area
500
        auto create_fn = [&ndef](StagingMap<Ordered>** ret) -> Status {
501
502
          DataTypeVector dtypes;
          int64_t capacity;
503
504
          int64_t memory_limit;
          TF RETURN IF ERROR(GetNodeAttr(ndef, "dtypes", &dtypes));
505
          TF_RETURN_IF_ERROR(GetNodeAttr(ndef, "capacity", &capacity));
506
507
          TF_RETURN_IF_ERROR(GetNodeAttr(ndef, "memory_limit", &memory_limit));
508
          *ret = new StagingMap<Ordered>(dtypes, capacity, memory_limit);
          return Status::OK();
509
510
        };
511
512
        TF_RETURN_IF_ERROR(cinfo.Init(rm, ndef, true /* use name() */));
        TF_RETURN_IF_ERROR(rm->LookupOrCreate<StagingMap<Ordered>>(
513
514
            cinfo.container(), cinfo.name(), map, create_fn));
515
        return Status::OK();
516
      }
517
518
      template <bool Ordered>
519
      class MapStageOp : public OpKernel {
```

```
520
       public:
521
        explicit MapStageOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
522
523
        void Compute(OpKernelContext* ctx) override {
524
          StagingMap<Ordered>* map = nullptr;
525
          OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
          core::ScopedUnref scope(map);
526
          typename StagingMap<Ordered>::OptionalTuple tuple;
527
528
          const Tensor* key_tensor;
529
530
          const Tensor* indices_tensor;
          OpInputList values tensor;
531
532
          OP_REQUIRES_OK(ctx, ctx->input("key", &key_tensor));
533
534
          OP REQUIRES OK(ctx, ctx->input("indices", &indices tensor));
535
          OP REQUIRES OK(ctx, ctx->input list("values", &values tensor));
          OP_REQUIRES(ctx, key_tensor->NumElements() > 0,
536
537
                       errors::InvalidArgument("key must not be empty"));
538
539
          // Create copy for insertion into Staging Area
540
          Tensor key(*key_tensor);
541
          // Create the tuple to store
542
          for (std::size_t i = 0; i < values_tensor.size(); ++i) {</pre>
543
            tuple.push_back(values_tensor[i]);
544
          }
545
546
          // Store the tuple in the map
547
          OP_REQUIRES_OK(ctx, map->put(&key, indices_tensor, &tuple));
548
549
        }
550
      };
551
552
      REGISTER_KERNEL_BUILDER(Name("MapStage").Device(DEVICE_CPU), MapStageOp<false>);
      REGISTER_KERNEL_BUILDER(Name("OrderedMapStage").Device(DEVICE_CPU),
553
554
                               MapStageOp<true>);
555
556
      #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
557
      REGISTER KERNEL BUILDER(
          Name("MapStage").HostMemory("key").HostMemory("indices").Device(DEVICE GPU),
558
559
          MapStageOp<false>);
560
      REGISTER_KERNEL_BUILDER(Name("OrderedMapStage")
561
                                   .HostMemory("key")
562
                                   .HostMemory("indices")
563
                                   .Device(DEVICE_GPU),
564
                               MapStageOp<true>);
      #endif // GOOGLE_CUDA || TENSORFLOW_USE_ROCM
565
566
567
568
      template <bool Ordered>
```

```
569
      class MapUnstageOp : public OpKernel {
570
       public:
571
        explicit MapUnstageOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
572
573
        // Using this op in such a way that it blocks forever
        // is an error. As such cancellation is not handled.
574
575
        void Compute(OpKernelContext* ctx) override {
          StagingMap<Ordered>* map = nullptr;
576
          OP REQUIRES OK(ctx, GetStagingMap(ctx, def(), &map));
577
          core::ScopedUnref scope(map);
578
          typename StagingMap<Ordered>::Tuple tuple;
579
580
          const Tensor* key tensor;
581
582
          const Tensor* indices_tensor;
583
          OP REQUIRES OK(ctx, ctx->input("key", &key tensor));
584
          OP_REQUIRES_OK(ctx, ctx->input("indices", &indices_tensor));
585
          OP_REQUIRES_OK(ctx, map->pop(key_tensor, indices_tensor, &tuple));
586
587
          OP REQUIRES(
588
589
              ctx, tuple.size() == indices_tensor->NumElements(),
590
              errors::InvalidArgument("output/indices size mismatch: ", tuple.size(),
                                       " vs. ", indices_tensor->NumElements()));
591
592
593
          for (std::size_t i = 0; i < tuple.size(); ++i) {</pre>
            ctx->set_output(i, tuple[i]);
594
          }
595
596
        }
597
      };
598
599
      REGISTER_KERNEL_BUILDER(Name("MapUnstage").Device(DEVICE_CPU),
                               MapUnstageOp<false>);
600
601
      REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstage").Device(DEVICE_CPU),
602
                               MapUnstageOp<true>);
603
      #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
604
605
      REGISTER_KERNEL_BUILDER(Name("MapUnstage")
606
                                   .HostMemory("key")
                                   .HostMemory("indices")
607
608
                                   .Device(DEVICE_GPU),
609
                               MapUnstageOp<false>);
      REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstage")
610
                                   .HostMemory("key")
611
612
                                   .HostMemory("indices")
613
                                   .Device(DEVICE_GPU),
                               MapUnstageOp<true>);
614
615
      #endif
616
617
      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
        // is an error. As such cancellation is not handled.
623
624
        void Compute(OpKernelContext* ctx) override {
          StagingMap<Ordered>* map = nullptr;
625
          OP REQUIRES OK(ctx, GetStagingMap(ctx, def(), &map));
626
          core::ScopedUnref scope(map);
627
          typename StagingMap<Ordered>::Tuple tuple;
628
629
          const Tensor* key tensor;
630
631
          const Tensor* indices_tensor;
632
          OP REQUIRES OK(ctx, ctx->input("key", &key tensor));
633
          OP_REQUIRES_OK(ctx, ctx->input("indices", &indices_tensor));
634
          OP_REQUIRES_OK(ctx, map->get(key_tensor, indices_tensor, &tuple));
635
636
          OP REQUIRES(
637
638
              ctx, tuple.size() == indices_tensor->NumElements(),
639
              errors::InvalidArgument("output/indices size mismatch: ", tuple.size(),
                                       " vs. ", indices_tensor->NumElements()));
640
641
642
          for (std::size_t i = 0; i < tuple.size(); ++i) {</pre>
            ctx->set_output(i, tuple[i]);
643
          }
644
645
        }
646
      };
647
      REGISTER KERNEL BUILDER(Name("MapPeek").Device(DEVICE CPU), MapPeekOp<false>);
648
      REGISTER_KERNEL_BUILDER(Name("OrderedMapPeek").Device(DEVICE_CPU),
649
650
                               MapPeekOp<true>);
651
      #if GOOGLE CUDA || TENSORFLOW USE ROCM
652
653
      REGISTER_KERNEL_BUILDER(
654
          Name("MapPeek").HostMemory("key").HostMemory("indices").Device(DEVICE_GPU),
655
          MapPeekOp<false>);
      REGISTER_KERNEL_BUILDER(Name("OrderedMapPeek")
656
657
                                   .HostMemory("key")
658
                                   .HostMemory("indices")
659
                                   .Device(DEVICE_GPU),
                               MapPeekOp<true>);
660
661
      #endif
662
663
664
      template <bool Ordered>
665
      class MapUnstageNoKeyOp : public OpKernel {
       public:
666
```

```
667
        explicit MapUnstageNoKeyOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
668
        // Using this op in such a way that it blocks forever
669
670
        // is an error. As such cancellation is not handled.
        void Compute(OpKernelContext* ctx) override {
671
          StagingMap<Ordered>* map = nullptr;
672
          OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
673
          core::ScopedUnref scope(map);
674
675
          // Pop a random (key, value) off the map
676
          typename StagingMap<Ordered>::KeyType key;
677
          typename StagingMap<Ordered>::Tuple tuple;
678
679
          const Tensor* indices_tensor;
680
681
          OP REQUIRES OK(ctx, ctx->input("indices", &indices tensor));
682
          OP_REQUIRES_OK(ctx, map->popitem(&key, indices_tensor, &tuple));
683
684
685
          // Allocate a key tensor and assign the key as the first output
          ctx->set output(0, key);
686
687
          // Set the rest of the outputs to the tuple Tensors
688
          OP REQUIRES(
689
              ctx, tuple.size() == indices_tensor->NumElements(),
690
              errors::InvalidArgument("output/indices size mismatch: ", tuple.size(),
691
                                       " vs. ", indices_tensor->NumElements()));
692
693
          for (std::size_t i = 0; i < tuple.size(); ++i) {</pre>
694
695
            ctx->set_output(i + 1, tuple[i]);
696
          }
697
        }
698
      };
699
      REGISTER_KERNEL_BUILDER(Name("MapUnstageNoKey").Device(DEVICE_CPU),
700
701
                               MapUnstageNoKeyOp<false>);
702
      REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstageNoKey").Device(DEVICE_CPU),
703
                               MapUnstageNoKeyOp<true>);
704
705
      #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
706
      REGISTER_KERNEL_BUILDER(Name("MapUnstageNoKey")
707
                                   .HostMemory("key")
                                   .HostMemory("indices")
708
709
                                   .Device(DEVICE_GPU),
710
                               MapUnstageNoKeyOp<false>);
711
      REGISTER_KERNEL_BUILDER(Name("OrderedMapUnstageNoKey")
712
                                   .HostMemory("key")
713
                                   .HostMemory("indices")
                                   .Device(DEVICE_GPU),
714
715
                               MapUnstageNoKeyOp<true>);
```

```
716
      #endif
717
718
719
      template <bool Ordered>
720
      class MapSizeOp : public OpKernel {
721
       public:
722
        explicit MapSizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
723
724
        void Compute(OpKernelContext* ctx) override {
          StagingMap<Ordered>* map = nullptr;
725
          OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
726
727
          core::ScopedUnref scope(map);
728
729
          // Allocate size output tensor
730
          Tensor* size = nullptr;
          OP REQUIRES OK(ctx, ctx->allocate output(0, TensorShape({}), &size));
731
732
          // Set it to the actual size
733
734
          size->scalar<int32>().setConstant(map->size());
735
       }
736
      };
737
738
      REGISTER KERNEL BUILDER(Name("MapSize").Device(DEVICE CPU), MapSizeOp<false>);
739
      REGISTER_KERNEL_BUILDER(Name("OrderedMapSize").Device(DEVICE_CPU),
740
                              MapSizeOp<true>);
741
742
      #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
743
      REGISTER_KERNEL_BUILDER(Name("MapSize").Device(DEVICE_GPU).HostMemory("size"),
744
                              MapSizeOp<false>);
745
      REGISTER_KERNEL_BUILDER(
          Name("OrderedMapSize").Device(DEVICE GPU).HostMemory("size"),
746
747
          MapSizeOp<true>);
748
      #endif
749
750
      template <bool Ordered>
751
      class MapIncompleteSizeOp : public OpKernel {
752
       public:
753
        explicit MapIncompleteSizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
754
755
        void Compute(OpKernelContext* ctx) override {
756
          StagingMap<Ordered>* map = nullptr;
757
          OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
758
          core::ScopedUnref scope(map);
759
760
          // Allocate size output tensor
761
          Tensor* size = nullptr;
762
          OP_REQUIRES_OK(ctx, ctx->allocate_output(0, TensorShape({}), &size));
763
764
          // Set it to the actual size
```

```
765
          size->scalar<int32>().setConstant(map->incomplete size());
        }
766
767
      };
768
      REGISTER KERNEL BUILDER(Name("MapIncompleteSize").Device(DEVICE CPU),
769
770
                              MapIncompleteSizeOp<false>);
771
      REGISTER_KERNEL_BUILDER(Name("OrderedMapIncompleteSize").Device(DEVICE_CPU),
772
                              MapIncompleteSizeOp<true>);
773
774
      #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
775
      REGISTER_KERNEL_BUILDER(
776
          Name("MapIncompleteSize").Device(DEVICE GPU).HostMemory("size"),
777
          MapIncompleteSizeOp<false>);
778
      REGISTER_KERNEL_BUILDER(
779
          Name("OrderedMapIncompleteSize").Device(DEVICE GPU).HostMemory("size"),
780
          MapIncompleteSizeOp<true>);
      #endif
781
782
      template <bool Ordered>
783
784
      class MapClearOp : public OpKernel {
785
       public:
786
        explicit MapClearOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
787
788
        void Compute(OpKernelContext* ctx) override {
789
          StagingMap<Ordered>* map = nullptr;
790
          OP_REQUIRES_OK(ctx, GetStagingMap(ctx, def(), &map));
          core::ScopedUnref scope(map);
791
792
793
          OP_REQUIRES_OK(ctx, map->clear());
794
        }
795
      };
796
797
      REGISTER_KERNEL_BUILDER(Name("MapClear").Device(DEVICE_CPU), MapClearOp<false>);
798
      REGISTER_KERNEL_BUILDER(Name("OrderedMapClear").Device(DEVICE_CPU),
799
                              MapClearOp<true>);
800
801
      #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
      REGISTER_KERNEL_BUILDER(Name("MapClear").Device(DEVICE_GPU), MapClearOp<false>);
802
      REGISTER_KERNEL_BUILDER(Name("OrderedMapClear").Device(DEVICE_GPU),
803
804
                              MapClearOp<true>);
805
      #endif
806
807
      } // namespace
808
      } // namespace tensorflow
```