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

5 contributors

519 lines (427 sloc) | 19.6 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.
==============================================================================*/

#define EIGEN_USE_THREADS

#include <algorithm>
#include <numeric>
#include <unordered_map>
#include <utility>
#include <vector>

#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/register_types.h"
#include "tensorflow/core/framework/resource_mgr.h"
#include "tensorflow/core/framework/tensor.h"
#include "tensorflow/core/framework/tensor_util.h"
#include "tensorflow/core/framework/types.h"
```

```
#include "tensorflow/core/lib/gtl/inlined_vector.h"
#include "tensorflow/core/util/overflow.h"
#include "tensorflow/core/util/sparse/sparse_tensor.h"

namespace tensorflow {

typedef Eigen::ThreadPoolDevice CPUDevice;

using sparse::SparseTensor;

class SparseTensorsMap : public ResourceBase {
 public:
  explicit SparseTensorsMap(const string& name) : name_(name), counter_(0) {}

  string DebugString() const override { return "A SparseTensorsMap"; }

  typedef struct {
    Tensor indices;
    Tensor values;
    gtl::InlinedVector<int64_t, 8> shape;
  } PersistentSparseTensor;

  Status AddSparseTensor(OpKernelContext* ctx, const SparseTensor& sp,
                         int64_t* handle) {
    Tensor ix;
    TF_RETURN_IF_ERROR(
        ctx->allocate_temp(sp.indices().dtype(), sp.indices().shape(), &ix));
    ix = sp.indices();

    Tensor values;
    TF_RETURN_IF_ERROR(ctx->allocate_temp(sp.indices().dtype(),
                                          sp.indices().shape(), &values));
    values = sp.values();
    {
      mutex_lock l(mu_);
      int64_t unique_st_handle = counter_++;  // increment is guarded on purpose
      sp_tensors_[unique_st_handle] = PersistentSparseTensor{
          ix, values,
          gtl::InlinedVector<int64_t, 8>(sp.shape().begin(), sp.shape().end())};
      *handle = unique_st_handle;
    }
    return Status::OK();
  }

  Status RetrieveAndClearSparseTensors(
      OpKernelContext* ctx, const TTypes<int64_t>::ConstVec& handles,
      std::vector<SparseTensor>* sparse_tensors) {
    sparse_tensors->clear();
    sparse_tensors->reserve(handles.size());
```

```
    {
      mutex_lock l(mu_);
      for (size_t i = 0; i < handles.size(); ++i) {
        const int64_t handle = handles(i);
        auto sp_iter = sp_tensors_.find(handle);
        if (sp_iter == sp_tensors_.end()) {
          return errors::InvalidArgument(
              "Unable to find SparseTensor: ", handle, " in map: ", name_);
        }
        const Tensor* ix = &sp_iter->second.indices;
        const Tensor* values = &sp_iter->second.values;
        const auto& shape = sp_iter->second.shape;
        SparseTensor tensor;
        TF_RETURN_IF_ERROR(SparseTensor::Create(*ix, *values, shape, &tensor));
        sparse_tensors->push_back(std::move(tensor));
        sp_tensors_.erase(sp_iter);
      }
    }

    return Status::OK();
  }

 protected:
  ~SparseTensorsMap() override {}

 private:
  string name_;

  mutex mu_;
  int64_t counter_ TF_GUARDED_BY(mu_);
  std::unordered_map<int64_t, PersistentSparseTensor> sp_tensors_
      TF_GUARDED_BY(mu_);
};

class SparseTensorAccessingOp : public OpKernel {
 public:
  typedef std::function<Status(SparseTensorsMap**)> CreatorCallback;

  explicit SparseTensorAccessingOp(OpKernelConstruction* context)
      : OpKernel(context), sparse_tensors_map_(nullptr) {}

 protected:
  ~SparseTensorAccessingOp() override {
    if (sparse_tensors_map_) sparse_tensors_map_->Unref();
  }

  Status GetMap(OpKernelContext* ctx, bool is_writing,
                SparseTensorsMap** sparse_tensors_map) {
    mutex_lock l(mu_);
```

```
128
129     if (sparse_tensors_map_) {
130       *sparse_tensors_map = sparse_tensors_map_;
131       return Status::OK();
132     }
133
134     TF_RETURN_IF_ERROR(cinfo_.Init(ctx->resource_manager(), def(),
135                                    is_writing /* use_node_name_as_default */));
136
137     CreatorCallback sparse_tensors_map_creator = [this](SparseTensorsMap** c) {
138       SparseTensorsMap* map = new SparseTensorsMap(cinfo_.name());
139       *c = map;
140       return Status::OK();
141     };
142
143     TF_RETURN_IF_ERROR(
144         cinfo_.resource_manager()->LookupOrCreate<SparseTensorsMap>(
145             cinfo_.container(), cinfo_.name(), &sparse_tensors_map_,
146             sparse_tensors_map_creator));
147
148     *sparse_tensors_map = sparse_tensors_map_;
149     return Status::OK();
150   }
151
152  private:
153   ContainerInfo cinfo_;
154
155   mutex mu_;
156   SparseTensorsMap* sparse_tensors_map_ TF_PT_GUARDED_BY(mu_);
157 };
158
159 class AddSparseToTensorsMapOp : public SparseTensorAccessingOp {
160  public:
161   explicit AddSparseToTensorsMapOp(OpKernelConstruction* context)
162       : SparseTensorAccessingOp(context) {}
163
164   void Compute(OpKernelContext* context) override {
165     const Tensor* input_indices;
166     const Tensor* input_values;
167     const Tensor* input_shape;
168     SparseTensorsMap* map;
169
170     OP_REQUIRES_OK(context, context->input("sparse_indices", &input_indices));
171     OP_REQUIRES_OK(context, context->input("sparse_values", &input_values));
172     OP_REQUIRES_OK(context, context->input("sparse_shape", &input_shape));
173     OP_REQUIRES_OK(context, GetMap(context, true /* is_writing */, &map));
174
175     OP_REQUIRES(context, TensorShapeUtils::IsMatrix(input_indices->shape()),
176                 errors::InvalidArgument(
```

```
                    "Input indices should be a matrix but received shape ",
                    input_indices->shape().DebugString()));

    OP_REQUIRES(context, TensorShapeUtils::IsVector(input_values->shape()),
                errors::InvalidArgument(
                    "Input values should be a vector but received shape ",
                    input_values->shape().DebugString()));

    OP_REQUIRES(context, TensorShapeUtils::IsVector(input_shape->shape()),
                errors::InvalidArgument(
                    "Input shape should be a vector but received shape ",
                    input_shape->shape().DebugString()));

    TensorShape input_shape_object;
    OP_REQUIRES_OK(
        context, TensorShapeUtils::MakeShape(input_shape->vec<int64_t>().data(),
                                             input_shape->NumElements(),
                                             &input_shape_object));
    SparseTensor st;
    OP_REQUIRES_OK(context, SparseTensor::Create(*input_indices, *input_values,
                                                 input_shape_object, &st));
    int64_t handle;
    OP_REQUIRES_OK(context, map->AddSparseTensor(context, st, &handle));

    Tensor sparse_handle(DT_INT64, TensorShape({}));
    auto sparse_handle_t = sparse_handle.scalar<int64_t>();

    sparse_handle_t() = handle;

    context->set_output(0, sparse_handle);
  }
};

REGISTER_KERNEL_BUILDER(Name("AddSparseToTensorsMap").Device(DEVICE_CPU),
                        AddSparseToTensorsMapOp);

template <typename T>
class AddManySparseToTensorsMapOp : public SparseTensorAccessingOp {
 public:
  explicit AddManySparseToTensorsMapOp(OpKernelConstruction* context)
      : SparseTensorAccessingOp(context) {}

  void Compute(OpKernelContext* context) override {
    const Tensor* input_indices;
    const Tensor* input_values;
    const Tensor* input_shape;
    SparseTensorsMap* map;

    OP_REQUIRES_OK(context, context->input("sparse_indices", &input_indices));
```

```
226     OP_REQUIRES_OK(context, context->input("sparse_values", &input_values));
227     OP_REQUIRES_OK(context, context->input("sparse_shape", &input_shape));
228     OP_REQUIRES_OK(context, GetMap(context, true /* is_writing */, &map));
229
230     OP_REQUIRES(context, TensorShapeUtils::IsMatrix(input_indices->shape()),
231                 errors::InvalidArgument(
232                     "Input indices should be a matrix but received shape ",
233                     input_indices->shape().DebugString()));
234
235     OP_REQUIRES(context, TensorShapeUtils::IsVector(input_values->shape()),
236                 errors::InvalidArgument(
237                     "Input values should be a vector but received shape ",
238                     input_values->shape().DebugString()));
239
240     OP_REQUIRES(context, TensorShapeUtils::IsVector(input_shape->shape()),
241                 errors::InvalidArgument(
242                     "Input shape should be a vector but received shape ",
243                     input_shape->shape().DebugString()));
244
245     int rank = input_shape->NumElements();
246
247     OP_REQUIRES(
248         context, rank > 1,
249         errors::InvalidArgument(
250             "Rank of input SparseTensor should be > 1, but saw rank: ", rank));
251
252     auto input_shape_vec = input_shape->vec<int64_t>();
253     int new_num_elements = 1;
254     bool overflow_ocurred = false;
255     for (int i = 0; i < input_shape_vec.size(); i++) {
256       new_num_elements =
257           MultiplyWithoutOverflow(new_num_elements, input_shape_vec(i));
258       if (new_num_elements < 0) {
259         overflow_ocurred = true;
260         break;
261       }
262     }
263
264     OP_REQUIRES(
265         context, !overflow_ocurred,
266         errors::Internal("Encountered overflow from large input shape."));
267
268     TensorShape tensor_input_shape(input_shape_vec);
269     gtl::InlinedVector<int64_t, 8> std_order(rank);
270     std::iota(std_order.begin(), std_order.end(), 0);
271     SparseTensor input_st;
272     OP_REQUIRES_OK(context, SparseTensor::Create(*input_indices, *input_values,
273                                                  tensor_input_shape, std_order,
274                                                  &input_st));
```

```
275
276     const int64_t N = input_shape_vec(0);
277
278     Tensor sparse_handles(DT_INT64, TensorShape({N}));
279     auto sparse_handles_t = sparse_handles.vec<int64_t>();
280
281     OP_REQUIRES_OK(context, input_st.IndicesValid());
282
283     // We can generate the output shape proto string now, for all
284     // minibatch entries.
285     TensorShape output_shape;
286     OP_REQUIRES_OK(context, TensorShapeUtils::MakeShape(
287                                 input_shape_vec.data() + 1,
288                                 input_shape->NumElements() - 1, &output_shape));
289
290     // Get groups by minibatch dimension
291     std::unordered_set<int64_t> visited;
292     sparse::GroupIterable minibatch = input_st.group({0});
293     for (const auto& subset : minibatch) {
294       const int64_t b = subset.group()[0];
295       visited.insert(b);
296       OP_REQUIRES(
297           context, b > -1 && b < N,
298           errors::InvalidArgument(
299               "Received unexpected column 0 value in input SparseTensor: ", b,
300               " < 0 or >= N (= ", N, ")"));
301
302       const auto indices = subset.indices();
303       const auto values = subset.values<T>();
304       const int64_t num_entries = values.size();
305
306       Tensor output_indices = Tensor(DT_INT64, {num_entries, rank - 1});
307       Tensor output_values = Tensor(DataTypeToEnum<T>::value, {num_entries});
308
309       auto output_indices_t = output_indices.matrix<int64_t>();
310       auto output_values_t = output_values.vec<T>();
311
312       for (int i = 0; i < num_entries; ++i) {
313         for (int d = 1; d < rank; ++d) {
314           output_indices_t(i, d - 1) = indices(i, d);
315         }
316         output_values_t(i) = values(i);
317       }
318
319       SparseTensor st_i;
320       OP_REQUIRES_OK(context,
321                      SparseTensor::Create(output_indices, output_values,
322                                           output_shape, &st_i));
323       int64_t handle;
```

```
      OP_REQUIRES_OK(context, map->AddSparseTensor(context, st_i, &handle));
      sparse_handles_t(b) = handle;
    }

    // Fill in any gaps; we must provide an empty ST for batch entries
    // the grouper didn't find.
    if (visited.size() < N) {
      Tensor empty_indices(DT_INT64, {0, rank - 1});
      Tensor empty_values(DataTypeToEnum<T>::value, {0});
      SparseTensor empty_st;
      OP_REQUIRES_OK(context, SparseTensor::Create(empty_indices, empty_values,
                                                   output_shape, &empty_st));

      for (int64_t b = 0; b < N; ++b) {
        // We skipped this batch entry.
        if (visited.find(b) == visited.end()) {
          int64_t handle;
          OP_REQUIRES_OK(context,
                         map->AddSparseTensor(context, empty_st, &handle));
          sparse_handles_t(b) = handle;
        }
      }
    }

    context->set_output(0, sparse_handles);
  }
};

#define REGISTER_KERNELS(type)                                \
  REGISTER_KERNEL_BUILDER(Name("AddManySparseToTensorsMap") \
                              .Device(DEVICE_CPU)           \
                              .TypeConstraint<type>("T"),   \
                          AddManySparseToTensorsMapOp<type>)

TF_CALL_ALL_TYPES(REGISTER_KERNELS);
#undef REGISTER_KERNELS

template <typename T>
class TakeManySparseFromTensorsMapOp : public SparseTensorAccessingOp {
 public:
  explicit TakeManySparseFromTensorsMapOp(OpKernelConstruction* context)
      : SparseTensorAccessingOp(context) {}

  void Compute(OpKernelContext* context) override {
    SparseTensorsMap* map = nullptr;
    OP_REQUIRES_OK(context, GetMap(context, false /* is_writing */, &map));

    const Tensor& sparse_handles = context->input(0);

```

```
373     OP_REQUIRES(context, TensorShapeUtils::IsVector(sparse_handles.shape()),
374                 errors::InvalidArgument(
375                     "sparse_handles should be a vector but received shape ",
376                     sparse_handles.shape().DebugString()));
377
378     int64_t N = sparse_handles.shape().dim_size(0);
379
380     OP_REQUIRES(
381         context, N > 0,
382         errors::InvalidArgument("Must have at least 1 serialized SparseTensor, "
383                                 "but input matrix has 0 rows"));
384
385     std::vector<Tensor> indices_to_concat;
386     std::vector<Tensor> values_to_concat;
387     std::vector<TensorShape> shapes_to_concat;
388
389     const auto& sparse_handles_t = sparse_handles.vec<int64_t>();
390
391     std::vector<SparseTensor> sparse_tensors;
392
393     OP_REQUIRES_OK(context, map->RetrieveAndClearSparseTensors(
394                                 context, sparse_handles_t, &sparse_tensors));
395
396     for (int64_t i = 0; i < N; ++i) {
397       const SparseTensor& st = sparse_tensors[i];
398       const Tensor& output_indices = st.indices();
399       const Tensor& output_values = st.values();
400       const auto output_shape = st.shape();
401
402       OP_REQUIRES(context, TensorShapeUtils::IsMatrix(output_indices.shape()),
403                   errors::InvalidArgument(
404                       "Expected sparse_handles[", i,
405                       "] to represent an index matrix but received shape ",
406                       output_indices.shape().DebugString()));
407       OP_REQUIRES(context, TensorShapeUtils::IsVector(output_values.shape()),
408                   errors::InvalidArgument(
409                       "Expected sparse_handles[", i,
410                       "] to represent a values vector but received shape ",
411                       output_values.shape().DebugString()));
412       OP_REQUIRES(
413           context, DataTypeToEnum<T>::value == output_values.dtype(),
414           errors::InvalidArgument(
415               "Requested SparseTensor of type ",
416               DataTypeString(DataTypeToEnum<T>::value), " but SparseTensor[", i,
417               "].values.dtype() == ", DataTypeString(output_values.dtype())));
418
419       int64_t num_entries = output_indices.dim_size(0);
420       OP_REQUIRES(context, num_entries == output_values.dim_size(0),
421                   errors::InvalidArgument(
```

```
422                     "Expected row counts of SparseTensor[", i,
423                     "].indices and SparseTensor[", i,
424                     "].values to match but they do not: ", num_entries,
425                     " vs. ", output_values.dim_size(0)));
426       int rank = output_indices.dim_size(1);
427       OP_REQUIRES(
428           context, rank == output_shape.size(),
429           errors::InvalidArgument("Expected column counts of SparseTensor[", i,
430                                   "].indices to match size of SparseTensor[", i,
431                                   "].shape "
432                                   "but they do not: ",
433                                   rank, " vs. ", output_shape.size()));
434
435       // Now we expand each SparseTensors' indices and shape by
436       // prefixing a dimension
437       Tensor expanded_indices(
438           DT_INT64, TensorShape({num_entries, 1 + output_indices.dim_size(1)}));
439       Tensor expanded_shape(DT_INT64, TensorShape({1 + rank}));
440       const auto& output_indices_t = output_indices.matrix<int64_t>();
441       auto expanded_indices_t = expanded_indices.matrix<int64_t>();
442       auto expanded_shape_t = expanded_shape.vec<int64_t>();
443       expanded_indices_t.chip<1>(0).setZero();
444       Eigen::DSizes<Eigen::DenseIndex, 2> indices_start(0, 1);
445       Eigen::DSizes<Eigen::DenseIndex, 2> indices_sizes(num_entries, rank);
446       expanded_indices_t.slice(indices_start, indices_sizes) = output_indices_t;
447       expanded_shape_t(0) = 1;
448       // TODO: copy shape from TensorShape to &expanded_shape_t(1)
449       // std::copy_n(&output_shape_t(0), rank, &expanded_shape_t(1));
450       for (int i = 0; i < rank; ++i) {
451         expanded_shape_t(i + 1) = output_shape[i];
452       }
453       TensorShape expanded_tensor_shape(expanded_shape_t);
454
455       indices_to_concat.push_back(std::move(expanded_indices));
456       values_to_concat.push_back(output_values);
457       shapes_to_concat.push_back(std::move(expanded_tensor_shape));
458     }
459
460     int rank = -1;
461     for (int i = 0; i < N; ++i) {
462       if (rank < 0) rank = shapes_to_concat[i].dims();
463       OP_REQUIRES(context, rank == shapes_to_concat[i].dims(),
464                   errors::InvalidArgument(
465                       "Inconsistent rank across SparseTensors: rank prior to "
466                       "SparseTensor[",
467                       i, "] was: ", rank, " but rank of SparseTensor[", i,
468                       "] is: ", shapes_to_concat[i].dims()));
469     }
470
```

```
    // SparseTensor::Concat requires consistent shape for all but the
    // primary order dimension (dimension 0 in this case).  So we get
    // the maximum value across all the input SparseTensors for each
    // dimension and use that.
    TensorShape preconcat_shape(shapes_to_concat[0]);
    for (int i = 0; i < N; ++i) {
      for (int d = 0; d < rank; ++d) {
        preconcat_shape.set_dim(d, std::max(preconcat_shape.dim_size(d),
                                            shapes_to_concat[i].dim_size(d)));
      }
    }

    // Dimension 0 is the primary dimension.
    gtl::InlinedVector<int64_t, 8> std_order(rank);
    std::iota(std_order.begin(), std_order.end(), 0);

    std::vector<SparseTensor> tensors_to_concat;
    tensors_to_concat.reserve(N);
    for (int i = 0; i < N; ++i) {
      SparseTensor tensor;
      OP_REQUIRES_OK(context,
                     SparseTensor::Create(std::move(indices_to_concat[i]),
                                          std::move(values_to_concat[i]),
                                          preconcat_shape, std_order, &tensor));
      tensors_to_concat.push_back(std::move(tensor));
    }

    auto output = SparseTensor::Concat<T>(tensors_to_concat);
    Tensor final_output_shape(DT_INT64, TensorShape({output.dims()}));

    std::copy_n(output.shape().data(), output.dims(),
                final_output_shape.vec<int64_t>().data());

    context->set_output(0, output.indices());
    context->set_output(1, output.values());
    context->set_output(2, final_output_shape);
  }
};

#define REGISTER_KERNELS(type)                                \
  REGISTER_KERNEL_BUILDER(Name("TakeManySparseFromTensorsMap") \
                              .Device(DEVICE_CPU)             \
                              .TypeConstraint<type>("dtype"),  \
                          TakeManySparseFromTensorsMapOp<type>)

TF_CALL_ALL_TYPES(REGISTER_KERNELS);
#undef REGISTER_KERNELS

}  // namespace tensorflow
```