rino20 Fix dependency from tf/lite/kernels to tf/lite/tools ... History

1 contributor

392 lines (348 sloc) | 14 KB ...

```
/* Copyright 2020 The TensorFlow Authors. All Rights Reserved.

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

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

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

#include <cstdint>
#include <utility>
#include <vector>

namespace tflite {
namespace internal {
namespace sparsity {

namespace {
uint64_t GetFlattenedIndex(const std::vector<int>& indices,
                           const std::vector<int>& shape) {
  uint64_t index = 0;
  int sub_elements = 1;
```

```
  for (int i = shape.size() - 1; i >= 0; i--) {
    index += indices[i] * sub_elements;
    sub_elements *= shape[i];
  }
  return index;
}

std::vector<int> TfLiteIntArrayToVector(const TfLiteIntArray* int_array) {
  std::vector<int> values;
  if (!int_array) {
    return values;
  }

  values.resize(int_array->size);
  for (size_t i = 0; i < int_array->size; i++) {
    values[i] = int_array->data[i];
  }

  return values;
}

}  // namespace

template <typename T>
FormatConverter<T>::FormatConverter(
    const std::vector<int>& shape, const std::vector<int>& traversal_order,
    const std::vector<TfLiteDimensionType>& format,
    const std::vector<int>& block_size, const std::vector<int>& block_map)
    : dense_shape_(shape),
      traversal_order_(traversal_order),
      block_size_(block_size),
      block_map_(block_map) {
  dense_size_ = 1;
  int block_dim = 0;
  blocked_shape_.resize(shape.size());
  format_.resize(shape.size() + block_map.size());
  for (int i = 0; i < shape.size(); i++) {
    format_[i] = format[traversal_order[i]];
    dense_size_ *= shape[i];
    if (block_dim < block_map.size() && block_map[block_dim] == i) {
      blocked_shape_[i] = shape[i] / block_size[block_dim];
      block_dim++;
    } else {
      blocked_shape_[i] = shape[i];
    }
  }

  // Only dense blocks are supported.
  for (int i = 0; i < block_map.size(); i++) {
```

```
    format_[i + shape.size()] = kTfLiteDimDense;
  }
}

template <typename T>
TfLiteStatus FormatConverter<T>::DenseToSparse(const T* src_data) {
  int num_original_dims = dense_shape_.size();
  int num_block_dims = block_map_.size();
  int num_expanded_dims = num_original_dims + num_block_dims;
  std::vector<int> expanded_shape(num_expanded_dims);
  for (int i = 0; i < num_expanded_dims; i++) {
    if (i < num_original_dims) {
      expanded_shape[i] = blocked_shape_[i];
    } else {
      expanded_shape[i] = block_size_[i - num_original_dims];
    }
  }

  std::vector<int> shape_offset(num_original_dims);
  shape_offset[shape_offset.size() - 1] = 1;
  for (int i = num_original_dims - 1; i > 0; --i) {
    shape_offset[i - 1] = shape_offset[i] * dense_shape_[i];
  }

  std::vector<int> expanded_shape_offset(num_expanded_dims);
  for (int i = 0; i < num_original_dims; ++i) {
    expanded_shape_offset[i] = shape_offset[i];
  }
  for (int i = 0; i < num_block_dims; ++i) {
    int mapped_dim = block_map_[i];
    expanded_shape_offset[num_original_dims + i] = shape_offset[mapped_dim];
    expanded_shape_offset[mapped_dim] *= block_size_[i];
  }

  std::vector<int> dst_ordered_offset(num_expanded_dims);
  for (int i = 0; i < num_expanded_dims; ++i) {
    dst_ordered_offset[i] = expanded_shape_offset[traversal_order_[i]];
  }

  std::vector<bool> dst_dim_has_nonzeroes(num_expanded_dims);
  std::fill(dst_dim_has_nonzeroes.begin(), dst_dim_has_nonzeroes.end(), false);
  std::vector<int> inner_compressed_dim(num_expanded_dims);
  int most_recent_compressed_dim = -1;
  std::vector<int> num_segments_of_next_compressed_dim(num_expanded_dims);
  int segment_count = 1;
  for (int i = num_expanded_dims - 1; i >= 0; --i) {
    inner_compressed_dim[i] = most_recent_compressed_dim;
    if (format_[i] == kTfLiteDimSparseCSR) {
      most_recent_compressed_dim = i;
```

```
      num_segments_of_next_compressed_dim[i] = segment_count;
      segment_count = 1;
    } else {
      num_segments_of_next_compressed_dim[i] = -1;
      segment_count *= expanded_shape[traversal_order_[i]];
    }
  }

  dim_metadata_.resize(num_expanded_dims * 2);
  std::vector<int> dst_sparse_dims;
  dst_sparse_dims.reserve(num_expanded_dims);
  for (int i = 0; i < num_expanded_dims; ++i) {
    dim_metadata_[i * 2].clear();
    dim_metadata_[i * 2 + 1].clear();
    if (format_[i] == kTfLiteDimDense) {
      // If dimension is dense, just store the shape.
      dim_metadata_[i * 2].push_back(expanded_shape[traversal_order_[i]]);
    } else {
      dim_metadata_[i * 2].push_back(0);  // Segment array always begins with 0.
      dst_sparse_dims.push_back(i);       // Add dimension to the sparse list.
    }
  }

  // This algorithm assumes that the block size is small enough for all the
  // elements to fit in cache, so the strided accesses from different traversal
  // order and the write-first-erase-later strategy shouldn't be too slow
  int dst_dim_idx = num_expanded_dims;
  std::vector<int> coordinate(num_expanded_dims, 0);
  int dense_tensor_idx = 0;
  while (dst_dim_idx >= 0) {
    if (dst_dim_idx == num_expanded_dims) {
      // We have a complete coordinate. Add the element to the value array if it
      // is not zero, or if the last dimension is dense.
      if (!IsZero(src_data[dense_tensor_idx])) {
        data_.push_back(src_data[dense_tensor_idx]);
        // Mark all sparse dimensions that their current indices have nonzeroes.
        for (auto dst_dim : dst_sparse_dims) {
          if (!dst_dim_has_nonzeroes[dst_dim]) {
            // Only add the index to the indices array if the current nonzero
            // is the first nonzero of the block.
            dim_metadata_[2 * dst_dim + 1].push_back(coordinate[dst_dim]);
            dst_dim_has_nonzeroes[dst_dim] = true;
          }
        }
      } else if (format_[num_expanded_dims - 1] == kTfLiteDimDense) {
        data_.push_back(src_data[dense_tensor_idx]);
      }
      --dst_dim_idx;
    } else {
```

```
      int original_dim_idx = traversal_order_[dst_dim_idx];
      int dim_size = expanded_shape[original_dim_idx];
      if (dst_dim_has_nonzeroes[dst_dim_idx]) {
        // If the previous block has nonzeroes, reset the flag to false since
        // we have just moved to a new block.
        dst_dim_has_nonzeroes[dst_dim_idx] = false;
      } else if (format_[dst_dim_idx] == kTfLiteDimSparseCSR) {
        // This block is empty. Delete unnecessary values if compressed.
        int next_compressed_dim = inner_compressed_dim[dst_dim_idx];
        int erase_offset = dim_metadata_[2 * dst_dim_idx + 1].size() *
                           num_segments_of_next_compressed_dim[dst_dim_idx];
        if (next_compressed_dim >= 0) {
          auto& segments = dim_metadata_[2 * inner_compressed_dim[dst_dim_idx]];
          segments.erase(segments.begin() + 1 + erase_offset, segments.end());
        } else {
          data_.erase(data_.begin() + erase_offset, data_.end());
        }
      }
      if (++coordinate[dst_dim_idx] < dim_size) {
        // The current dst_dim_idx is valid (not out of bound).
        dense_tensor_idx += dst_ordered_offset[dst_dim_idx];
        ++dst_dim_idx;
      } else {
        // dst_dim_idx has reached its dim size. Update segment array and go
        // back to incrementing the previous dimension (dst_dim_idx - 1).
        if (format_[dst_dim_idx] == kTfLiteDimSparseCSR) {
          dim_metadata_[2 * dst_dim_idx].push_back(
              dim_metadata_[2 * dst_dim_idx + 1].size());
        }
        coordinate[dst_dim_idx] = -1;
        dense_tensor_idx -= dst_ordered_offset[dst_dim_idx] * dim_size;
        --dst_dim_idx;
      }
    }
  }

  return kTfLiteOk;
}

template <typename T>
FormatConverter<T>::FormatConverter(
    const std::vector<int>& shape, const std::vector<int>& traversal_order,
    const std::vector<TfLiteDimensionType>& format,
    const std::vector<int>& dense_size,
    const std::vector<std::vector<int>>& segments,
    const std::vector<std::vector<int>>& indices,
    const std::vector<int>& block_map) {
  InitSparseToDenseConverter(shape, traversal_order, format, dense_size,
                             segments, indices, block_map);
```

```
}

template <typename T>
FormatConverter<T>::FormatConverter(const std::vector<int>& shape,
                                    const TfLiteSparsity& sparsity) {
  auto traversal_order = TfLiteIntArrayToVector(sparsity.traversal_order);
  auto block_map = TfLiteIntArrayToVector(sparsity.block_map);

  std::vector<TfLiteDimensionType> format(sparsity.dim_metadata_size);
  std::vector<int> dense_size(sparsity.dim_metadata_size);
  std::vector<std::vector<int>> segments(sparsity.dim_metadata_size);
  std::vector<std::vector<int>> indices(sparsity.dim_metadata_size);
  for (int i = 0; i < sparsity.dim_metadata_size; i++) {
    format[i] = sparsity.dim_metadata[i].format;
    dense_size[i] = sparsity.dim_metadata[i].dense_size;
    segments[i] =
        TfLiteIntArrayToVector(sparsity.dim_metadata[i].array_segments);
    indices[i] = TfLiteIntArrayToVector(sparsity.dim_metadata[i].array_indices);
  }

  InitSparseToDenseConverter(shape, std::move(traversal_order),
                             std::move(format), std::move(dense_size),
                             std::move(segments), std::move(indices),
                             std::move(block_map));
}

template <typename T>
void FormatConverter<T>::InitSparseToDenseConverter(
    std::vector<int> shape, std::vector<int> traversal_order,
    std::vector<TfLiteDimensionType> format, std::vector<int> dense_size,
    std::vector<std::vector<int>> segments,
    std::vector<std::vector<int>> indices, std::vector<int> block_map) {
  dense_shape_ = std::move(shape);
  traversal_order_ = std::move(traversal_order);
  block_map_ = std::move(block_map);
  format_ = std::move(format);

  dense_size_ = 1;
  for (int i = 0; i < dense_shape_.size(); i++) {
    dense_size_ *= dense_shape_[i];
  }

  dim_metadata_.resize(2 * format_.size());
  for (int i = 0; i < format_.size(); i++) {
    if (format_[i] == kTfLiteDimDense) {
      dim_metadata_[2 * i] = {dense_size[i]};
    } else {
      dim_metadata_[2 * i] = std::move(segments[i]);
      dim_metadata_[2 * i + 1] = std::move(indices[i]);
```

```
    }
  }

  int original_rank = dense_shape_.size();
  int block_dim = 0;

  blocked_shape_.resize(original_rank);
  block_size_.resize(block_map_.size());
  for (int i = 0; i < original_rank; i++) {
    if (block_dim < block_map_.size() && block_map_[block_dim] == i) {
      int orig_dim = traversal_order_[original_rank + block_dim];
      block_size_[block_dim] = dense_size[orig_dim];
      blocked_shape_[i] = dense_shape_[i] / dense_size[orig_dim];
      block_dim++;
    } else {
      blocked_shape_[i] = dense_shape_[i];
    }
  }
}

template <typename T>
void FormatConverter<T>::Populate(const T* src_data, std::vector<int> indices,
                                  int level, int prev_idx, int* src_data_ptr,
                                  T* dest_data) {
  if (level == indices.size()) {
    int orig_rank = dense_shape_.size();
    std::vector<int> orig_idx;
    orig_idx.resize(orig_rank);
    int i = 0;
    for (; i < orig_idx.size(); i++) {
      int orig_dim = traversal_order_[i];
      orig_idx[orig_dim] = indices[i];
    }

    for (; i < indices.size(); i++) {
      const int block_idx = traversal_order_[i] - orig_rank;
      const int orig_dim = block_map_[block_idx];
      orig_idx[orig_dim] =
          orig_idx[orig_dim] * block_size_[block_idx] + indices[i];
    }

    dest_data[GetFlattenedIndex(orig_idx, dense_shape_)] =
        src_data[*src_data_ptr];

    *src_data_ptr = *src_data_ptr + 1;
    return;
  }

  const int metadata_idx = 2 * level;
```

```
  const int shape_of_level = dim_metadata_[metadata_idx][0];
  if (format_[level] == kTfLiteDimDense) {
    for (int i = 0; i < shape_of_level; i++) {
      indices[level] = i;
      Populate(src_data, indices, level + 1, prev_idx * shape_of_level + i,
               src_data_ptr, dest_data);
    }
  } else {
    const auto& array_segments = dim_metadata_[metadata_idx];
    const auto& array_indices = dim_metadata_[metadata_idx + 1];
    for (int i = array_segments[prev_idx]; i < array_segments[prev_idx + 1];
         i++) {
      indices[level] = array_indices[i];
      Populate(src_data, indices, level + 1, i, src_data_ptr, dest_data);
    }
  }
}

template <typename T>
TfLiteStatus FormatConverter<T>::SparseToDense(const T* src_data) {
  data_.resize(dense_size_);
  std::fill(data_.begin(), data_.end(), T(0));

  int total_rank = traversal_order_.size();
  int src_data_ptr = 0;
  std::vector<int> indices(total_rank);
  Populate(src_data, indices, 0, 0, &src_data_ptr, data_.data());

  return kTfLiteOk;
}

template <typename T>
TfLiteStatus FormatConverter<T>::SparseToDense(const T* src_data,
                                               const size_t dest_size,
                                               T* dest_data,
                                               TfLiteContext* context) {
  if (dest_size != dense_size_) {
    TF_LITE_MAYBE_KERNEL_LOG(
        context, "unexpected buffer size for densified data, expected %lld.\n",
        dense_size_);
    return kTfLiteError;
  }

  // For types like Eigen::half, we cannot do a simple memset() with 0 values.
  for (auto i = 0; i < dest_size; i++) {
    dest_data[i] = T(0);
  }

  const int total_rank = traversal_order_.size();
```

```
  int src_data_ptr = 0;
  std::vector<int> indices(total_rank);
  Populate(src_data, indices, 0, 0, &src_data_ptr, dest_data);

  return kTfLiteOk;
}

template <typename T>
bool FormatConverter<T>::IsZero(const T val) {
  return (val == static_cast<T>(0));
}

template class FormatConverter<int32_t>;
template class FormatConverter<int8_t>;
template class FormatConverter<float>;
template class FormatConverter<Eigen::half>;

}  // namespace sparsity
}  // namespace internal
}  // namespace tflite
```