293 lines (265 sloc) | 11.8 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.
==============================================================================*/

// See docs in ../ops/math_ops.cc.

#define EIGEN_USE_THREADS

#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/type_traits.h"
#include "tensorflow/core/framework/types.h"
#include "tensorflow/core/kernels/meta_support.h"
#include "tensorflow/core/kernels/quantization_utils.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/platform/bfloat16.h"

namespace {
```

```
enum {
  QUANTIZE_MODE_MIN_COMBINED,
  QUANTIZE_MODE_MIN_FIRST,
  QUANTIZE_MODE_SCALED,
};
}  // namespace

namespace tensorflow {

typedef Eigen::ThreadPoolDevice CPUDevice;

template <typename T>
T Cast(float v) {
  return v;
}

template <>
bfloat16 Cast<bfloat16>(float v) {
  return bfloat16(v);
}

template <typename Device, typename T, typename S>
class DequantizeOp : public OpKernel {
 public:
  explicit DequantizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    string mode_string;
    OP_REQUIRES_OK(ctx, ctx->GetAttr("mode", &mode_string));
    OP_REQUIRES(
        ctx,
        (ctx->output_type(0) == DT_FLOAT || ctx->output_type(0) == DT_BFLOAT16),
        errors::InvalidArgument("Output type must be bfloat16 or float,"
                                " is '" +
                                DataTypeString(ctx->output_type(0)) + "'"));

    need_cast_ = true;
    if (ctx->output_type(0) == DT_FLOAT) {
      need_cast_ = false;
      OP_REQUIRES(ctx,
                  (mode_string == "MIN_COMBINED" ||
                   mode_string == "MIN_FIRST" || mode_string == "SCALED"),
                  errors::InvalidArgument("Mode string must be 'MIN_COMBINED',"
                                          " 'MIN_FIRST', or 'SCALED', is '" +
                                          mode_string + "'"));
    } else {
      OP_REQUIRES(
          ctx, (mode_string == "MIN_COMBINED"),
          errors::InvalidArgument("When output type is bfloat16, Mode"
                                  " string must be 'MIN_COMBINED', is '" +
                                  mode_string + "'"));
```

```
    }

    if (mode_string == "MIN_COMBINED") {
      mode_ = QUANTIZE_MODE_MIN_COMBINED;
    } else if (mode_string == "MIN_FIRST") {
      mode_ = QUANTIZE_MODE_MIN_FIRST;
    } else if (mode_string == "SCALED") {
      mode_ = QUANTIZE_MODE_SCALED;
    }
    OP_REQUIRES_OK(ctx, ctx->GetAttr("narrow_range", &narrow_range_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("axis", &axis_));
  }

  void Compute(OpKernelContext* ctx) override {
    const Tensor& input = ctx->input(0);
    const Tensor& input_min_tensor = ctx->input(1);
    const Tensor& input_max_tensor = ctx->input(2);

    int num_slices = 1;
    if (axis_ > -1) {
      num_slices = input.dim_size(axis_);
    }
    OP_REQUIRES(ctx, input_min_tensor.NumElements() == num_slices,
                errors::InvalidArgument(
                    "input_min_tensor must have as many elements as input on "
                    "the dequantization axis (",
                    axis_, "), got ", input_min_tensor.NumElements(),
                    ", expected ", num_slices));
    OP_REQUIRES(ctx, input_max_tensor.NumElements() == num_slices,
                errors::InvalidArgument(
                    "input_max_tensor must have as many elements as input on "
                    "the dequantization axis (",
                    axis_, "), got ", input_max_tensor.NumElements(),
                    ", expected ", num_slices));

    Tensor* output = nullptr;
    OP_REQUIRES_OK(ctx, ctx->allocate_output(0, input.shape(), &output));
    Tensor float_output =
        need_cast_ ? tensorflow::Tensor(DT_FLOAT, input.shape()) : *output;
    if (num_slices == 1) {
      const float min_range = input_min_tensor.flat<float>()(0);
      const float max_range = input_max_tensor.flat<float>()(0);
      DequantizeTensor(ctx, input, min_range, max_range, &float_output);
    } else {
      OP_REQUIRES(ctx, mode_ != QUANTIZE_MODE_MIN_FIRST,
                  errors::Unimplemented("MIN_FIRST mode is not implemented for "
                                        "Dequantize with axis != -1."));

      int64_t pre_dim = 1, post_dim = 1;
```

```
128       for (int i = 0; i < axis_; ++i) {
129         pre_dim *= float_output.dim_size(i);
130       }
131       for (int i = axis_ + 1; i < float_output.dims(); ++i) {
132         post_dim *= float_output.dim_size(i);
133       }
134       auto input_tensor = input.template bit_casted_shaped<T, 3>(
135           {pre_dim, num_slices, post_dim});
136       auto output_tensor =
137           float_output.flat_inner_outer_dims<float, 3>(axis_ - 1);
138       auto min_ranges = input_min_tensor.vec<float>();
139       auto max_ranges = input_max_tensor.vec<float>();
140       for (int i = 0; i < num_slices; ++i) {
141         DequantizeSlice(ctx->eigen_device<Device>(), ctx,
142                         input_tensor.template chip<1>(i), min_ranges(i),
143                         max_ranges(i), output_tensor.template chip<1>(i));
144       }
145     }
146     if (need_cast_) {
147       S* out_ptr = output->flat<S>().data();
148       float* in_ptr = float_output.flat<float>().data();
149       for (int64_t i = 0; i < float_output.NumElements(); ++i) {
150         out_ptr[i] = static_cast<S>(in_ptr[i]);
151       }
152     }
153   }
154
155   void DequantizeTensor(OpKernelContext* ctx, const Tensor& input,
156                         const float min_range, const float max_range,
157                         Tensor* output) {
158     const float half_range =
159         !std::is_signed<T>::value
160             ? 0.0f
161             : (static_cast<float>(std::numeric_limits<T>::max()) -
162                std::numeric_limits<T>::min() + 1) /
163                   2.0f;
164
165     if (mode_ == QUANTIZE_MODE_MIN_COMBINED) {
166       const float scale_factor =
167           (max_range - min_range) /
168           (static_cast<float>(std::numeric_limits<T>::max()) -
169            std::numeric_limits<T>::min());
170
171       const auto& input_tensor = input.flat<T>();
172       output->flat<float>() =
173           ((input_tensor.template cast<float>() + half_range) * scale_factor) +
174           min_range;
175
176     } else if (mode_ == QUANTIZE_MODE_MIN_FIRST) {
```

```
177       if (meta::IsSupportedAndEnabled() && std::is_same<T, quint8>()) {
178         auto input_ui8_array = input.flat<quint8>();
179         meta::Dequantize(ctx, input_ui8_array.data(), input_ui8_array.size(),
180                          min_range, max_range, output->flat<float>().data());
181       } else {
182         QuantizedTensorToFloatInPlaceUsingEigen<T>(
183             ctx->template eigen_device<Device>(), input, min_range, max_range,
184             output);
185       }
186     } else if (mode_ == QUANTIZE_MODE_SCALED) {
187       const int min_output_value =
188           std::numeric_limits<T>::min() + (narrow_range_ ? 1 : 0);
189       const float scale_factor =
190           std::numeric_limits<T>::min() == 0
191               ? (max_range / std::numeric_limits<T>::max())
192               : std::max(min_range / min_output_value,
193                          max_range / std::numeric_limits<T>::max());
194       const auto& input_tensor = input.flat<T>();
195       output->flat<float>() =
196           input_tensor.template cast<int>().template cast<float>() *
197           scale_factor;
198     }
199   }
200
201   template <typename ConstVec, typename Vec>
202   void DequantizeSlice(const Device& d, OpKernelContext* ctx,
203                        const ConstVec& input, float min_range, float max_range,
204                        Vec output) {
205     // TODO(pauldonnelly): Factor out the similar calculations in quantize,
206     //   dequantize and quantize_and_dequantize ops.
207     const float half_range =
208         !std::is_signed<T>::value
209             ? 0.0f
210             : (static_cast<float>(std::numeric_limits<T>::max()) -
211                std::numeric_limits<T>::min() + 1) /
212                   2.0f;
213
214     if (mode_ == QUANTIZE_MODE_MIN_COMBINED) {
215       const float scale_factor =
216           (max_range - min_range) /
217           (static_cast<float>(std::numeric_limits<T>::max()) -
218            std::numeric_limits<T>::min());
219
220       output.device(d) =
221           ((input.template cast<float>() + half_range) * scale_factor) +
222           min_range;
223     } else if (mode_ == QUANTIZE_MODE_SCALED) {
224       const int min_output_value =
225           std::numeric_limits<T>::min() + (narrow_range_ ? 1 : 0);
```

```
      const float scale_factor =
          std::numeric_limits<T>::min() == 0
              ? (max_range / std::numeric_limits<T>::max())
              : std::max(min_range / min_output_value,
                         max_range / std::numeric_limits<T>::max());
      output.device(d) = input.template cast<float>() * scale_factor;
    }
  }

 private:
  int mode_;
  int axis_;
  bool narrow_range_;
  bool need_cast_;
};

REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<quint8>("T")
                            .TypeConstraint<float>("dtype"),
                        DequantizeOp<CPUDevice, quint8, float>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<qint8>("T")
                            .TypeConstraint<float>("dtype"),
                        DequantizeOp<CPUDevice, qint8, float>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<quint16>("T")
                            .TypeConstraint<float>("dtype"),
                        DequantizeOp<CPUDevice, quint16, float>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<qint16>("T")
                            .TypeConstraint<float>("dtype"),
                        DequantizeOp<CPUDevice, qint16, float>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<qint32>("T")
                            .TypeConstraint<float>("dtype"),
                        DequantizeOp<CPUDevice, qint32, float>);

REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<quint8>("T")
                            .TypeConstraint<bfloat16>("dtype"),
                        DequantizeOp<CPUDevice, quint8, bfloat16>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
```

```
                            .TypeConstraint<qint8>("T")
                            .TypeConstraint<bfloat16>("dtype"),
                        DequantizeOp<CPUDevice, qint8, bfloat16>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<quint16>("T")
                            .TypeConstraint<bfloat16>("dtype"),
                        DequantizeOp<CPUDevice, quint16, bfloat16>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<qint16>("T")
                            .TypeConstraint<bfloat16>("dtype"),
                        DequantizeOp<CPUDevice, qint16, bfloat16>);
REGISTER_KERNEL_BUILDER(Name("Dequantize")
                            .Device(DEVICE_CPU)
                            .TypeConstraint<qint32>("T")
                            .TypeConstraint<bfloat16>("dtype"),
                        DequantizeOp<CPUDevice, qint32, bfloat16>);
}  // namespace tensorflow
```