pak-laura Validate axis input in tf.raw_ops.QuantizeAndDequantizeV4Grad ... ✓ History

9 contributors

494 lines (451 sloc) | 23 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.
==============================================================================*/

#include "tensorflow/core/framework/op_requires.h"
#define EIGEN_USE_THREADS

#if (defined(GOOGLE_CUDA) && GOOGLE_CUDA) || \
    (defined(TENSORFLOW_USE_ROCM) && TENSORFLOW_USE_ROCM)
#define EIGEN_USE_GPU
#endif  // GOOGLE_CUDA || TENSORFLOW_USE_ROCM

#include "tensorflow/core/kernels/quantize_and_dequantize_op.h"

#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/register_types.h"
#include "tensorflow/core/framework/type_traits.h"
```

```
#include "tensorflow/core/framework/types.h"
#include "tensorflow/core/lib/core/errors.h"

namespace tensorflow {

typedef Eigen::ThreadPoolDevice CPUDevice;
typedef Eigen::GpuDevice GPUDevice;

// Simulate quantization precision loss in a float tensor by:
// 1. Quantize the tensor to fixed point numbers, which should match the target
//    quantization method when it is used in inference.
// 2. Dequantize it back to floating point numbers for the following ops, most
//    likely matmul.
template <typename Device, typename T>
class QuantizeAndDequantizeV2Op : public OpKernel {
 public:
  explicit QuantizeAndDequantizeV2Op(OpKernelConstruction* ctx)
      : OpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("signed_input", &signed_input_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("axis", &axis_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("num_bits", &num_bits_));
    OP_REQUIRES(ctx, num_bits_ > 0 && num_bits_ < (signed_input_ ? 62 : 63),
                errors::InvalidArgument("num_bits is out of range: ", num_bits_,
                                        " with signed_input_ ", signed_input_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("range_given", &range_given_));

    string round_mode_string;
    OP_REQUIRES_OK(ctx, ctx->GetAttr("round_mode", &round_mode_string));
    OP_REQUIRES(
        ctx,
        (round_mode_string == "HALF_UP" || round_mode_string == "HALF_TO_EVEN"),
        errors::InvalidArgument("Round mode string must be "
                                "'HALF_UP' or "
                                "'HALF_TO_EVEN', is '" +
                                round_mode_string + "'"));
    if (round_mode_string == "HALF_UP") {
      round_mode_ = ROUND_HALF_UP;
    } else if (round_mode_string == "HALF_TO_EVEN") {
      round_mode_ = ROUND_HALF_TO_EVEN;
    }
    OP_REQUIRES_OK(ctx, ctx->GetAttr("narrow_range", &narrow_range_));
  }

  void Compute(OpKernelContext* ctx) override {
    const Tensor& input = ctx->input(0);
    OP_REQUIRES(
        ctx, axis_ >= -1,
        errors::InvalidArgument("Axis must be at least -1. Found ", axis_));
    OP_REQUIRES(
```

```
        ctx, (axis_ == -1 || axis_ < input.shape().dims()),
        errors::InvalidArgument("Shape must be at least rank ", axis_ + 1,
                                " but is rank ", input.shape().dims()));
    const int depth = (axis_ == -1) ? 1 : input.dim_size(axis_);
    Tensor input_min_tensor;
    Tensor input_max_tensor;
    Tensor* output = nullptr;
    OP_REQUIRES_OK(ctx, ctx->allocate_output(0, input.shape(), &output));
    if (range_given_) {
      input_min_tensor = ctx->input(1);
      input_max_tensor = ctx->input(2);
      if (axis_ == -1) {
        auto min_val = input_min_tensor.scalar<T>()();
        auto max_val = input_max_tensor.scalar<T>()();
        OP_REQUIRES(ctx, min_val <= max_val,
                    errors::InvalidArgument("Invalid range: input_min ",
                                            min_val, " > input_max ", max_val));
      } else {
        OP_REQUIRES(ctx, input_min_tensor.dim_size(0) == depth,
                    errors::InvalidArgument(
                        "input_min_tensor has incorrect size, was ",
                        input_min_tensor.dim_size(0), " expected ", depth,
                        " to match dim ", axis_, " of the input ",
                        input_min_tensor.shape()));
        OP_REQUIRES(ctx, input_max_tensor.dim_size(0) == depth,
                    errors::InvalidArgument(
                        "input_max_tensor has incorrect size, was ",
                        input_max_tensor.dim_size(0), " expected ", depth,
                        " to match dim ", axis_, " of the input ",
                        input_max_tensor.shape()));
      }
    } else {
      auto range_shape = (axis_ == -1) ? TensorShape({}) : TensorShape({depth});
      OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
                                             range_shape, &input_min_tensor));
      OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
                                             range_shape, &input_max_tensor));
    }

    if (axis_ == -1) {
      functor::QuantizeAndDequantizeOneScaleFunctor<Device, T> f;
      f(ctx->eigen_device<Device>(), input.flat<T>(), signed_input_, num_bits_,
        range_given_, &input_min_tensor, &input_max_tensor, round_mode_,
        narrow_range_, output->flat<T>());
    } else {
      functor::QuantizeAndDequantizePerChannelFunctor<Device, T> f;
      f(ctx->eigen_device<Device>(),
        input.template flat_inner_outer_dims<T, 3>(axis_ - 1), signed_input_,
        num_bits_, range_given_, &input_min_tensor, &input_max_tensor,
```

```
128         round_mode_, narrow_range_,
129         output->template flat_inner_outer_dims<T, 3>(axis_ - 1));
130     }
131   }
132
133  private:
134   int num_bits_;
135   int axis_;
136   QuantizerRoundMode round_mode_;
137   bool signed_input_;
138   bool range_given_;
139   bool narrow_range_;
140 };
141
142 // Implementation of QuantizeAndDequantizeV4GradientOp.
143 // When back-propagating the error through a quantized layer, the following
144 // paper gives evidence that clipped-ReLU is better than non-clipped:
145 // "Deep Learning with Low Precision by Half-wave Gaussian Quantization"
146 // http://zpascal.net/cvpr2017/Cai_Deep_Learning_With_CVPR_2017_paper.pdf
147 template <typename Device, typename T>
148 class QuantizeAndDequantizeV4GradientOp : public OpKernel {
149  public:
150   explicit QuantizeAndDequantizeV4GradientOp(OpKernelConstruction* ctx)
151       : OpKernel::OpKernel(ctx) {
152     OP_REQUIRES_OK(ctx, ctx->GetAttr("axis", &axis_));
153   }
154
155   void Compute(OpKernelContext* ctx) override {
156     const Tensor& gradient = ctx->input(0);
157     const Tensor& input = ctx->input(1);
158     Tensor* input_backprop = nullptr;
159     OP_REQUIRES_OK(ctx,
160                    ctx->allocate_output(0, input.shape(), &input_backprop));
161     OP_REQUIRES(
162         ctx, axis_ >= -1,
163         errors::InvalidArgument("Axis must be at least -1. Found ", axis_));
164     OP_REQUIRES(ctx, (axis_ == -1 || axis_ < input.shape().dims()),
165                 errors::InvalidArgument(
166                     "Axis should be -1 or 0 or a positive value less than ",
167                     input.shape().dims(), "but given axis value was ", axis_));
168
169     OP_REQUIRES(
170         ctx, input.IsSameSize(gradient),
171         errors::InvalidArgument("gradient and input must be the same size"));
172     const int depth = (axis_ == -1) ? 1 : input.dim_size(axis_);
173     const Tensor& input_min_tensor = ctx->input(2);
174     OP_REQUIRES(ctx,
175                 input_min_tensor.dims() == 0 || input_min_tensor.dims() == 1,
176                 errors::InvalidArgument(
```

```
                    "Input min tensor must have dimension 1. Recieved ",
                    input_min_tensor.dims(), "."));
    const Tensor& input_max_tensor = ctx->input(3);
    OP_REQUIRES(ctx,
                input_max_tensor.dims() == 0 || input_max_tensor.dims() == 1,
                errors::InvalidArgument(
                    "Input max tensor must have dimension 1. Recieved ",
                    input_max_tensor.dims(), "."));
    if (axis_ != -1) {
      OP_REQUIRES(
          ctx, input_min_tensor.dim_size(0) == depth,
          errors::InvalidArgument("min has incorrect size, expected ", depth,
                                  " was ", input_min_tensor.dim_size(0)));
      OP_REQUIRES(
          ctx, input_max_tensor.dim_size(0) == depth,
          errors::InvalidArgument("max has incorrect size, expected ", depth,
                                  " was ", input_max_tensor.dim_size(0)));
    }

    TensorShape min_max_shape(input_min_tensor.shape());
    Tensor* input_min_backprop;
    OP_REQUIRES_OK(ctx,
                   ctx->allocate_output(1, min_max_shape, &input_min_backprop));

    Tensor* input_max_backprop;
    OP_REQUIRES_OK(ctx,
                   ctx->allocate_output(2, min_max_shape, &input_max_backprop));

    if (axis_ == -1) {
      functor::QuantizeAndDequantizeOneScaleGradientFunctor<Device, T> f;
      f(ctx->eigen_device<Device>(), gradient.template flat<T>(),
        input.template flat<T>(), input_min_tensor.scalar<T>(),
        input_max_tensor.scalar<T>(), input_backprop->template flat<T>(),
        input_min_backprop->template scalar<T>(),
        input_max_backprop->template scalar<T>());
    } else {
      functor::QuantizeAndDequantizePerChannelGradientFunctor<Device, T> f;
      f(ctx->eigen_device<Device>(),
        gradient.template flat_inner_outer_dims<T, 3>(axis_ - 1),
        input.template flat_inner_outer_dims<T, 3>(axis_ - 1),
        &input_min_tensor, &input_max_tensor,
        input_backprop->template flat_inner_outer_dims<T, 3>(axis_ - 1),
        input_min_backprop->template flat<T>(),
        input_max_backprop->template flat<T>());
    }
  }

 private:
  int axis_;
```

```
};

// Simulate quantization precision loss in a float tensor by:
// 1. Quantize the tensor to fixed point numbers, which should match the target
//    quantization method when it is used in inference.
// 2. Dequantize it back to floating point numbers for the following ops, most
//    likely matmul.
// Almost identical to QuantizeAndDequantizeV2Op, except that num_bits is a
// tensor.
template <typename Device, typename T>
class QuantizeAndDequantizeV3Op : public OpKernel {
 public:
  explicit QuantizeAndDequantizeV3Op(OpKernelConstruction* ctx)
      : OpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("signed_input", &signed_input_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("range_given", &range_given_));
    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);
    OP_REQUIRES(ctx, axis_ < input.dims(),
                errors::InvalidArgument(
                    "Axis requested is larger than input dimensions. Axis: ",
                    axis_, " Input Dimensions: ", input.dims()));
    const int depth = (axis_ == -1) ? 1 : input.dim_size(axis_);
    Tensor* output = nullptr;
    OP_REQUIRES_OK(ctx, ctx->allocate_output(0, input.shape(), &output));

    Tensor num_bits_tensor;
    num_bits_tensor = ctx->input(3);
    int num_bits_val = num_bits_tensor.scalar<int32>()();

    OP_REQUIRES(
        ctx, num_bits_val > 0 && num_bits_val < (signed_input_ ? 62 : 63),
        errors::InvalidArgument("num_bits is out of range: ", num_bits_val,
                                " with signed_input_ ", signed_input_));

    Tensor input_min_tensor;
    Tensor input_max_tensor;
    if (range_given_) {
      input_min_tensor = ctx->input(1);
      input_max_tensor = ctx->input(2);
      if (axis_ == -1) {
        auto min_val = input_min_tensor.scalar<T>()();
        auto max_val = input_max_tensor.scalar<T>()();
        OP_REQUIRES(ctx, min_val <= max_val,
                    errors::InvalidArgument("Invalid range: input_min ",
```

```
                                          min_val, " > input_max ", max_val));
      } else {
        OP_REQUIRES(ctx, input_min_tensor.dim_size(0) == depth,
                    errors::InvalidArgument(
                        "input_min_tensor has incorrect size, was ",
                        input_min_tensor.dim_size(0), " expected ", depth,
                        " to match dim ", axis_, " of the input ",
                        input_min_tensor.shape()));
        OP_REQUIRES(ctx, input_max_tensor.dim_size(0) == depth,
                    errors::InvalidArgument(
                        "input_max_tensor has incorrect size, was ",
                        input_max_tensor.dim_size(0), " expected ", depth,
                        " to match dim ", axis_, " of the input ",
                        input_max_tensor.shape()));
      }
    } else {
      auto range_shape = (axis_ == -1) ? TensorShape({}) : TensorShape({depth});
      OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
                                             range_shape, &input_min_tensor));
      OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
                                             range_shape, &input_max_tensor));
    }

    if (axis_ == -1) {
      functor::QuantizeAndDequantizeOneScaleFunctor<Device, T> f;
      f(ctx->eigen_device<Device>(), input.flat<T>(), signed_input_,
        num_bits_val, range_given_, &input_min_tensor, &input_max_tensor,
        ROUND_HALF_TO_EVEN, narrow_range_, output->flat<T>());
    } else {
      functor::QuantizeAndDequantizePerChannelFunctor<Device, T> f;
      f(ctx->eigen_device<Device>(),
        input.template flat_inner_outer_dims<T, 3>(axis_ - 1), signed_input_,
        num_bits_val, range_given_, &input_min_tensor, &input_max_tensor,
        ROUND_HALF_TO_EVEN, narrow_range_,
        output->template flat_inner_outer_dims<T, 3>(axis_ - 1));
    }
  }

 private:
  int axis_;
  bool signed_input_;
  bool range_given_;
  bool narrow_range_;
};

// DEPRECATED: Use QuantizeAndDequantizeV2Op.
template <typename Device, typename T>
class QuantizeAndDequantizeOp : public OpKernel {
 public:
```

```
  explicit QuantizeAndDequantizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("signed_input", &signed_input_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("num_bits", &num_bits_));
    OP_REQUIRES(ctx, num_bits_ > 0 && num_bits_ < (signed_input_ ? 62 : 63),
                errors::InvalidArgument("num_bits is out of range: ", num_bits_,
                                        " with signed_input_ ", signed_input_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("range_given", &range_given_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("input_min", &input_min_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("input_max", &input_max_));
    if (range_given_) {
      OP_REQUIRES(
          ctx, input_min_ <= input_max_,
          errors::InvalidArgument("Invalid range: input_min ", input_min_,
                                  " > input_max ", input_max_));
    }
  }

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

    Tensor* output = nullptr;
    OP_REQUIRES_OK(ctx, ctx->allocate_output(0, input.shape(), &output));

    // One global scale.
    Tensor input_min_tensor(DataTypeToEnum<T>::value, TensorShape());
    Tensor input_max_tensor(DataTypeToEnum<T>::value, TensorShape());
    // Initialize the tensors with the values in the Attrs.
    input_min_tensor.template scalar<T>()() = static_cast<T>(input_min_);
    input_max_tensor.template scalar<T>()() = static_cast<T>(input_max_);

    functor::QuantizeAndDequantizeOneScaleFunctor<Device, T> functor;
    functor(ctx->eigen_device<Device>(), input.flat<T>(), signed_input_,
            num_bits_, range_given_, &input_min_tensor, &input_max_tensor,
            ROUND_HALF_TO_EVEN, /*narrow_range=*/false, output->flat<T>());
  }

 private:
  bool signed_input_;
  int num_bits_;
  bool range_given_;
  float input_min_;
  float input_max_;
};

// Specializations for CPUDevice.

namespace functor {
template <typename T>
struct QuantizeAndDequantizeOneScaleFunctor<CPUDevice, T> {
```

```
  void operator()(const CPUDevice& d, typename TTypes<T>::ConstVec input,
                  const bool signed_input, const int num_bits,
                  const bool range_given, Tensor* input_min_tensor,
                  Tensor* input_max_tensor, QuantizerRoundMode round_mode,
                  bool narrow_range, typename TTypes<T>::Vec out) {
    QuantizeAndDequantizeOneScaleImpl<CPUDevice, T>::Compute(
        d, input, signed_input, num_bits, range_given, input_min_tensor,
        input_max_tensor, round_mode, narrow_range, out);
  }
};

template <typename T>
struct QuantizeAndDequantizePerChannelFunctor<CPUDevice, T> {
  void operator()(const CPUDevice& d, typename TTypes<T, 3>::ConstTensor input,
                  bool signed_input, int num_bits, bool range_given,
                  Tensor* input_min_tensor, Tensor* input_max_tensor,
                  QuantizerRoundMode round_mode, bool narrow_range,
                  typename TTypes<T, 3>::Tensor out) {
    QuantizeAndDequantizePerChannelImpl<CPUDevice, T>::Compute(
        d, input, signed_input, num_bits, range_given, input_min_tensor,
        input_max_tensor, round_mode, narrow_range, out);
  }
};

template <typename T>
struct QuantizeAndDequantizeOneScaleGradientFunctor<CPUDevice, T> {
  void operator()(const CPUDevice& d, typename TTypes<T>::ConstFlat gradient,
                  typename TTypes<T>::ConstFlat input,
                  typename TTypes<T>::ConstScalar input_min_tensor,
                  typename TTypes<T>::ConstScalar input_max_tensor,
                  typename TTypes<T>::Flat input_backprop,
                  typename TTypes<T>::Scalar input_min_backprop,
                  typename TTypes<T>::Scalar input_max_backprop) {
    QuantizeAndDequantizeOneScaleGradientImpl<CPUDevice, T>::Compute(
        d, gradient, input, input_min_tensor, input_max_tensor, input_backprop,
        input_min_backprop, input_max_backprop);
  }
};

template <typename T>
struct QuantizeAndDequantizePerChannelGradientFunctor<CPUDevice, T> {
  void operator()(const CPUDevice& d,
                  typename TTypes<T, 3>::ConstTensor gradient,
                  typename TTypes<T, 3>::ConstTensor input,
                  const Tensor* input_min_tensor,
                  const Tensor* input_max_tensor,
                  typename TTypes<T, 3>::Tensor input_backprop,
                  typename TTypes<T>::Flat input_min_backprop,
                  typename TTypes<T>::Flat input_max_backprop) {
```

```
    QuantizeAndDequantizePerChannelGradientImpl<CPUDevice, T>::Compute(
        d, gradient, input, input_min_tensor, input_max_tensor, input_backprop,
        input_min_backprop, input_max_backprop);
  }
};

template struct functor::QuantizeAndDequantizeOneScaleGradientFunctor<CPUDevice,
                                                                      float>;
template struct functor::QuantizeAndDequantizePerChannelGradientFunctor<
    CPUDevice, double>;

}  // namespace functor

#define REGISTER_CPU_KERNEL(T)                                         \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV2")              \
                              .Device(DEVICE_CPU)                      \
                              .TypeConstraint<T>("T"),                 \
                          QuantizeAndDequantizeV2Op<CPUDevice, T>);    \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV3")              \
                              .Device(DEVICE_CPU)                      \
                              .TypeConstraint<T>("T"),                 \
                          QuantizeAndDequantizeV3Op<CPUDevice, T>);    \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4")              \
                              .Device(DEVICE_CPU)                      \
                              .TypeConstraint<T>("T"),                 \
                          QuantizeAndDequantizeV2Op<CPUDevice, T>);    \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4Grad")          \
                              .Device(DEVICE_CPU)                      \
                              .TypeConstraint<T>("T"),                 \
                          QuantizeAndDequantizeV4GradientOp<CPUDevice, T>);  \
  REGISTER_KERNEL_BUILDER(                                             \
      Name("QuantizeAndDequantize").Device(DEVICE_CPU).TypeConstraint<T>("T"), \
      QuantizeAndDequantizeOp<CPUDevice, T>);
TF_CALL_float(REGISTER_CPU_KERNEL);
TF_CALL_double(REGISTER_CPU_KERNEL);
#undef REGISTER_CPU_KERNEL

#if (defined(GOOGLE_CUDA) && GOOGLE_CUDA) || \
    (defined(TENSORFLOW_USE_ROCM) && TENSORFLOW_USE_ROCM)
#define REGISTER_GPU_KERNEL(T)                                         \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV2")              \
                              .Device(DEVICE_GPU)                      \
                              .HostMemory("input_min")                 \
                              .HostMemory("input_max")                 \
                              .TypeConstraint<T>("T"),                 \
                          QuantizeAndDequantizeV2Op<GPUDevice, T>);    \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV3")              \
                              .Device(DEVICE_GPU)                      \
                              .HostMemory("input_min")                 \
```

```
                              .HostMemory("input_max")                          \
                              .HostMemory("num_bits")                           \
                              .TypeConstraint<T>("T"),                          \
                          QuantizeAndDequantizeV3Op<GPUDevice, T>);             \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4")                       \
                              .Device(DEVICE_GPU)                               \
                              .HostMemory("input_min")                          \
                              .HostMemory("input_max")                          \
                              .TypeConstraint<T>("T"),                          \
                          QuantizeAndDequantizeV2Op<GPUDevice, T>);             \
  REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4Grad")                   \
                              .Device(DEVICE_GPU)                               \
                              .HostMemory("input_min")                          \
                              .HostMemory("input_max")                          \
                              .TypeConstraint<T>("T"),                          \
                          QuantizeAndDequantizeV4GradientOp<GPUDevice, T>);     \
  REGISTER_KERNEL_BUILDER(                                                      \
      Name("QuantizeAndDequantize").Device(DEVICE_GPU).TypeConstraint<T>("T"), \
      QuantizeAndDequantizeOp<GPUDevice, T>);
TF_CALL_float(REGISTER_GPU_KERNEL);
TF_CALL_double(REGISTER_GPU_KERNEL);
#undef REGISTER_GPU_KERNEL
#endif  // GOOGLE_CUDA || TENSORFLOW_USE_ROCM
}  // namespace tensorflow
```