

f3b9bf4c3c ▾

...

tensorflow / tensorflow / core / kernels / quantize_and_dequantize_op.cc



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

History

9 contributors



494 lines (451 sloc) | 23 KB

...

```

1  /* Copyright 2015 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
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,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 =====*/
15
16 #include "tensorflow/core/framework/op_requires.h"
17 #define EIGEN_USE_THREADS
18
19 #if (defined(GOOGLE_CUDA) && GOOGLE_CUDA) || \
20     (defined(TENSORFLOW_USE_ROCM) && TENSORFLOW_USE_ROCM)
21 #define EIGEN_USE_GPU
22 #endif // GOOGLE_CUDA || TENSORFLOW_USE_ROCM
23
24 #include "tensorflow/core/kernels/quantize_and_dequantize_op.h"
25
26 #include "tensorflow/core/framework/op.h"
27 #include "tensorflow/core/framework/op_kernel.h"
28 #include "tensorflow/core/framework/register_types.h"
29 #include "tensorflow/core/framework/type_traits.h"

```

```

30 #include "tensorflow/core/framework/types.h"
31 #include "tensorflow/core/lib/core/errors.h"
32
33 namespace tensorflow {
34
35 typedef Eigen::ThreadPoolDevice CPUDevice;
36 typedef Eigen::GpuDevice GPUDevice;
37
38 // Simulate quantization precision loss in a float tensor by:
39 // 1. Quantize the tensor to fixed point numbers, which should match the target
40 //    quantization method when it is used in inference.
41 // 2. Dequantize it back to floating point numbers for the following ops, most
42 //    likely matmul.
43 template <typename Device, typename T>
44 class QuantizeAndDequantizeV2Op : public OpKernel {
45 public:
46     explicit QuantizeAndDequantizeV2Op(OpKernelConstruction* ctx)
47         : OpKernel(ctx) {
48         OP_REQUIRES_OK(ctx, ctx->GetAttr("signed_input", &signed_input_));
49         OP_REQUIRES_OK(ctx, ctx->GetAttr("axis", &axis_));
50         OP_REQUIRES_OK(ctx, ctx->GetAttr("num_bits", &num_bits_));
51         OP_REQUIRES(ctx, num_bits_ > 0 && num_bits_ < (signed_input_ ? 62 : 63),
52                     errors::InvalidArgument("num_bits is out of range: ", num_bits_,
53                                             " with signed_input_ ", signed_input_));
54         OP_REQUIRES_OK(ctx, ctx->GetAttr("range_given", &range_given_));
55
56         string round_mode_string;
57         OP_REQUIRES_OK(ctx, ctx->GetAttr("round_mode", &round_mode_string));
58         OP_REQUIRES(
59             ctx,
60             (round_mode_string == "HALF_UP" || round_mode_string == "HALF_TO_EVEN"),
61             errors::InvalidArgument("Round mode string must be "
62                                     "'HALF_UP' or "
63                                     "'HALF_TO_EVEN', is '" +
64                                     round_mode_string + "'"));
65         if (round_mode_string == "HALF_UP") {
66             round_mode_ = ROUND_HALF_UP;
67         } else if (round_mode_string == "HALF_TO_EVEN") {
68             round_mode_ = ROUND_HALF_TO_EVEN;
69         }
70         OP_REQUIRES_OK(ctx, ctx->GetAttr("narrow_range", &narrow_range_));
71     }
72
73     void Compute(OpKernelContext* ctx) override {
74         const Tensor& input = ctx->input(0);
75         OP_REQUIRES(
76             ctx, axis_ >= -1,
77             errors::InvalidArgument("Axis must be at least -1. Found ", axis_));
78         OP_REQUIRES(

```

```

79     ctx, (axis_ == -1 || axis_ < input.shape().dims()),
80     errors::InvalidArgument("Shape must be at least rank ", axis_ + 1,
81                             " but is rank ", input.shape().dims()));
82 const int depth = (axis_ == -1) ? 1 : input.dim_size(axis_);
83 Tensor input_min_tensor;
84 Tensor input_max_tensor;
85 Tensor* output = nullptr;
86 OP_REQUIRES_OK(ctx, ctx->allocate_output(0, input.shape(), &output));
87 if (range_given_) {
88     input_min_tensor = ctx->input(1);
89     input_max_tensor = ctx->input(2);
90     if (axis_ == -1) {
91         auto min_val = input_min_tensor.scalar<T>();
92         auto max_val = input_max_tensor.scalar<T>();
93         OP_REQUIRES(ctx, min_val <= max_val,
94                     errors::InvalidArgument("Invalid range: input_min ",
95                                             min_val, " > input_max ", max_val));
96     } else {
97         OP_REQUIRES(ctx, input_min_tensor.dim_size(0) == depth,
98                     errors::InvalidArgument(
99                         "input_min_tensor has incorrect size, was ",
100                        input_min_tensor.dim_size(0), " expected ", depth,
101                        " to match dim ", axis_, " of the input ",
102                        input_min_tensor.shape()));
103         OP_REQUIRES(ctx, input_max_tensor.dim_size(0) == depth,
104                     errors::InvalidArgument(
105                         "input_max_tensor has incorrect size, was ",
106                        input_max_tensor.dim_size(0), " expected ", depth,
107                        " to match dim ", axis_, " of the input ",
108                        input_max_tensor.shape()));
109     }
110 } else {
111     auto range_shape = (axis_ == -1) ? TensorShape({}) : TensorShape({depth});
112     OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
113                                           range_shape, &input_min_tensor));
114     OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
115                                           range_shape, &input_max_tensor));
116 }
117
118 if (axis_ == -1) {
119     functor::QuantizeAndDequantizeOneScaleFunctor<Device, T> f;
120     f(ctx->eigen_device<Device>(), input.flat<T>(), signed_input_, num_bits_,
121     range_given_, &input_min_tensor, &input_max_tensor, round_mode_,
122     narrow_range_, output->flat<T>());
123 } else {
124     functor::QuantizeAndDequantizePerChannelFunctor<Device, T> f;
125     f(ctx->eigen_device<Device>(),
126     input.template flat_inner_outer_dims<T, 3>(axis_ - 1), signed_input_,
127     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://zfpascal.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(

```

```

177         "Input min tensor must have dimension 1. Recieved ",
178         input_min_tensor.dims(), ".");
179     const Tensor& input_max_tensor = ctx->input(3);
180     OP_REQUIRES(ctx,
181         input_max_tensor.dims() == 0 || input_max_tensor.dims() == 1,
182         errors::InvalidArgument(
183             "Input max tensor must have dimension 1. Recieved ",
184             input_max_tensor.dims(), "."));
185     if (axis_ != -1) {
186         OP_REQUIRES(
187             ctx, input_min_tensor.dim_size(0) == depth,
188             errors::InvalidArgument("min has incorrect size, expected ", depth,
189                                     " was ", input_min_tensor.dim_size(0)));
190         OP_REQUIRES(
191             ctx, input_max_tensor.dim_size(0) == depth,
192             errors::InvalidArgument("max has incorrect size, expected ", depth,
193                                     " was ", input_max_tensor.dim_size(0)));
194     }
195
196     TensorShape min_max_shape(input_min_tensor.shape());
197     Tensor* input_min_backprop;
198     OP_REQUIRES_OK(ctx,
199         ctx->allocate_output(1, min_max_shape, &input_min_backprop));
200
201     Tensor* input_max_backprop;
202     OP_REQUIRES_OK(ctx,
203         ctx->allocate_output(2, min_max_shape, &input_max_backprop));
204
205     if (axis_ == -1) {
206         functor::QuantizeAndDequantizeOneScaleGradientFunctor<Device, T> f;
207         f(ctx->eigen_device<Device>(), gradient.template flat<T>(),
208           input.template flat<T>(), input_min_tensor.scalar<T>(),
209           input_max_tensor.scalar<T>(), input_backprop->template flat<T>(),
210           input_min_backprop->template scalar<T>(),
211           input_max_backprop->template scalar<T>());
212     } else {
213         functor::QuantizeAndDequantizePerChannelGradientFunctor<Device, T> f;
214         f(ctx->eigen_device<Device>(),
215           gradient.template flat_inner_outer_dims<T, 3>(axis_ - 1),
216           input.template flat_inner_outer_dims<T, 3>(axis_ - 1),
217           &input_min_tensor, &input_max_tensor,
218           input_backprop->template flat_inner_outer_dims<T, 3>(axis_ - 1),
219           input_min_backprop->template flat<T>(),
220           input_max_backprop->template flat<T>());
221     }
222 }
223
224 private:
225     int axis_;

```

```

226 };
227
228 // Simulate quantization precision loss in a float tensor by:
229 // 1. Quantize the tensor to fixed point numbers, which should match the target
230 //    quantization method when it is used in inference.
231 // 2. Dequantize it back to floating point numbers for the following ops, most
232 //    likely matmul.
233 // Almost identical to QuantizeAndDequantizeV2Op, except that num_bits is a
234 // tensor.
235 template <typename Device, typename T>
236 class QuantizeAndDequantizeV3Op : public OpKernel {
237 public:
238     explicit QuantizeAndDequantizeV3Op(OpKernelConstruction* ctx)
239         : OpKernel(ctx) {
240         OP_REQUIRES_OK(ctx, ctx->GetAttr("signed_input", &signed_input_));
241         OP_REQUIRES_OK(ctx, ctx->GetAttr("range_given", &range_given_));
242         OP_REQUIRES_OK(ctx, ctx->GetAttr("narrow_range", &narrow_range_));
243         OP_REQUIRES_OK(ctx, ctx->GetAttr("axis", &axis_));
244     }
245
246     void Compute(OpKernelContext* ctx) override {
247         const Tensor& input = ctx->input(0);
248         OP_REQUIRES(ctx, axis_ < input.dims(),
249                     errors::InvalidArgument(
250                         "Axis requested is larger than input dimensions. Axis: ",
251                         axis_, " Input Dimensions: ", input.dims()));
252         const int depth = (axis_ == -1) ? 1 : input.dim_size(axis_);
253         Tensor* output = nullptr;
254         OP_REQUIRES_OK(ctx, ctx->allocate_output(0, input.shape(), &output));
255
256         Tensor num_bits_tensor;
257         num_bits_tensor = ctx->input(3);
258         int num_bits_val = num_bits_tensor.scalar<int32>()();
259
260         OP_REQUIRES(
261             ctx, num_bits_val > 0 && num_bits_val < (signed_input_ ? 62 : 63),
262             errors::InvalidArgument("num_bits is out of range: ", num_bits_val,
263                                     " with signed_input_ ", signed_input_));
264
265         Tensor input_min_tensor;
266         Tensor input_max_tensor;
267         if (range_given_) {
268             input_min_tensor = ctx->input(1);
269             input_max_tensor = ctx->input(2);
270             if (axis_ == -1) {
271                 auto min_val = input_min_tensor.scalar<T>()();
272                 auto max_val = input_max_tensor.scalar<T>()();
273                 OP_REQUIRES(ctx, min_val <= max_val,
274                             errors::InvalidArgument("Invalid range: input_min ",

```

```

275         min_val, " > input_max ", max_val));
276     } else {
277         OP_REQUIRES(ctx, input_min_tensor.dim_size(0) == depth,
278             errors::InvalidArgument(
279                 "input_min_tensor has incorrect size, was ",
280                 input_min_tensor.dim_size(0), " expected ", depth,
281                 " to match dim ", axis_, " of the input ",
282                 input_min_tensor.shape()));
283         OP_REQUIRES(ctx, input_max_tensor.dim_size(0) == depth,
284             errors::InvalidArgument(
285                 "input_max_tensor has incorrect size, was ",
286                 input_max_tensor.dim_size(0), " expected ", depth,
287                 " to match dim ", axis_, " of the input ",
288                 input_max_tensor.shape()));
289     }
290 } else {
291     auto range_shape = (axis_ == -1) ? TensorShape({}) : TensorShape({depth});
292     OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
293         range_shape, &input_min_tensor));
294     OP_REQUIRES_OK(ctx, ctx->allocate_temp(DataTypeToEnum<T>::value,
295         range_shape, &input_max_tensor));
296 }
297
298 if (axis_ == -1) {
299     functor::QuantizeAndDequantizeOneScaleFunctor<Device, T> f;
300     f(ctx->eigen_device<Device>(), input.flat<T>(), signed_input_,
301         num_bits_val, range_given_, &input_min_tensor, &input_max_tensor,
302         ROUND_HALF_TO_EVEN, narrow_range_, output->flat<T>());
303 } else {
304     functor::QuantizeAndDequantizePerChannelFunctor<Device, T> f;
305     f(ctx->eigen_device<Device>(),
306         input.template flat_inner_outer_dims<T, 3>(axis_ - 1), signed_input_,
307         num_bits_val, range_given_, &input_min_tensor, &input_max_tensor,
308         ROUND_HALF_TO_EVEN, narrow_range_,
309         output->template flat_inner_outer_dims<T, 3>(axis_ - 1));
310 }
311 }
312
313 private:
314     int axis_;
315     bool signed_input_;
316     bool range_given_;
317     bool narrow_range_;
318 };
319
320 // DEPRECATED: Use QuantizeAndDequantizeV2Op.
321 template <typename Device, typename T>
322 class QuantizeAndDequantizeOp : public OpKernel {
323 public:

```

```

324 explicit QuantizeAndDequantizeOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
325     OP_REQUIRES_OK(ctx, ctx->GetAttr("signed_input", &signed_input_));
326     OP_REQUIRES_OK(ctx, ctx->GetAttr("num_bits", &num_bits_));
327     OP_REQUIRES(ctx, num_bits_ > 0 && num_bits_ < (signed_input_ ? 62 : 63),
328         errors::InvalidArgument("num_bits is out of range: ", num_bits_,
329             " with signed_input_ ", signed_input_));
330     OP_REQUIRES_OK(ctx, ctx->GetAttr("range_given", &range_given_));
331     OP_REQUIRES_OK(ctx, ctx->GetAttr("input_min", &input_min_));
332     OP_REQUIRES_OK(ctx, ctx->GetAttr("input_max", &input_max_));
333     if (range_given_) {
334         OP_REQUIRES(
335             ctx, input_min_ <= input_max_,
336             errors::InvalidArgument("Invalid range: input_min ", input_min_,
337                 " > input_max ", input_max_));
338     }
339 }
340
341 void Compute(OpKernelContext* ctx) override {
342     const Tensor& input = ctx->input(0);
343
344     Tensor* output = nullptr;
345     OP_REQUIRES_OK(ctx, ctx->allocate_output(0, input.shape(), &output));
346
347     // One global scale.
348     Tensor input_min_tensor(DataTypeToEnum<T>::value, TensorShape());
349     Tensor input_max_tensor(DataTypeToEnum<T>::value, TensorShape());
350     // Initialize the tensors with the values in the Attrs.
351     input_min_tensor.template scalar<T>() = static_cast<T>(input_min_);
352     input_max_tensor.template scalar<T>() = static_cast<T>(input_max_);
353
354     functor::QuantizeAndDequantizeOneScaleFunctor<Device, T> functor;
355     functor(ctx->eigen_device<Device>(), input.flat<T>(), signed_input_,
356         num_bits_, range_given_, &input_min_tensor, &input_max_tensor,
357         ROUND_HALF_TO_EVEN, /*narrow_range=*/false, output->flat<T>());
358 }
359
360 private:
361     bool signed_input_;
362     int num_bits_;
363     bool range_given_;
364     float input_min_;
365     float input_max_;
366 };
367
368 // Specializations for CPUDevice.
369
370 namespace functor {
371     template <typename T>
372     struct QuantizeAndDequantizeOneScaleFunctor<CPUDevice, T> {

```



```

373 void operator()(const CPUDevice& d, typename TTypes<T>::ConstVec input,
374               const bool signed_input, const int num_bits,
375               const bool range_given, Tensor* input_min_tensor,
376               Tensor* input_max_tensor, QuantizerRoundMode round_mode,
377               bool narrow_range, typename TTypes<T>::Vec out) {
378     QuantizeAndDequantizeOneScaleImpl<CPUDevice, T>::Compute(
379         d, input, signed_input, num_bits, range_given, input_min_tensor,
380         input_max_tensor, round_mode, narrow_range, out);
381 }
382 };
383
384 template <typename T>
385 struct QuantizeAndDequantizePerChannelFunctor<CPUDevice, T> {
386     void operator()(const CPUDevice& d, typename TTypes<T, 3>::ConstTensor input,
387                   bool signed_input, int num_bits, bool range_given,
388                   Tensor* input_min_tensor, Tensor* input_max_tensor,
389                   QuantizerRoundMode round_mode, bool narrow_range,
390                   typename TTypes<T, 3>::Tensor out) {
391         QuantizeAndDequantizePerChannelImpl<CPUDevice, T>::Compute(
392             d, input, signed_input, num_bits, range_given, input_min_tensor,
393             input_max_tensor, round_mode, narrow_range, out);
394     }
395 };
396
397 template <typename T>
398 struct QuantizeAndDequantizeOneScaleGradientFunctor<CPUDevice, T> {
399     void operator()(const CPUDevice& d, typename TTypes<T>::ConstFlat gradient,
400                   typename TTypes<T>::ConstFlat input,
401                   typename TTypes<T>::ConstScalar input_min_tensor,
402                   typename TTypes<T>::ConstScalar input_max_tensor,
403                   typename TTypes<T>::Flat input_backprop,
404                   typename TTypes<T>::Scalar input_min_backprop,
405                   typename TTypes<T>::Scalar input_max_backprop) {
406         QuantizeAndDequantizeOneScaleGradientImpl<CPUDevice, T>::Compute(
407             d, gradient, input, input_min_tensor, input_max_tensor, input_backprop,
408             input_min_backprop, input_max_backprop);
409     }
410 };
411
412 template <typename T>
413 struct QuantizeAndDequantizePerChannelGradientFunctor<CPUDevice, T> {
414     void operator()(const CPUDevice& d,
415                   typename TTypes<T, 3>::ConstTensor gradient,
416                   typename TTypes<T, 3>::ConstTensor input,
417                   const Tensor* input_min_tensor,
418                   const Tensor* input_max_tensor,
419                   typename TTypes<T, 3>::Tensor input_backprop,
420                   typename TTypes<T>::Flat input_min_backprop,
421                   typename TTypes<T>::Flat input_max_backprop) {

```

```

422     QuantizeAndDequantizePerChannelGradientImpl<CPUDevice, T>::Compute(
423         d, gradient, input, input_min_tensor, input_max_tensor, input_backprop,
424         input_min_backprop, input_max_backprop);
425     }
426 };
427
428 template struct functor::QuantizeAndDequantizeOneScaleGradientFunctor<CPUDevice,
429                                     float>;
430 template struct functor::QuantizeAndDequantizePerChannelGradientFunctor<
431     CPUDevice, double>;
432
433 } // namespace functor
434
435 #define REGISTER_CPU_KERNEL(T) \
436     REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV2") \
437         .Device(DEVICE_CPU) \
438         .TypeConstraint<T>("T"), \
439         QuantizeAndDequantizeV2Op<CPUDevice, T>); \
440     REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV3") \
441         .Device(DEVICE_CPU) \
442         .TypeConstraint<T>("T"), \
443         QuantizeAndDequantizeV3Op<CPUDevice, T>); \
444     REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4") \
445         .Device(DEVICE_CPU) \
446         .TypeConstraint<T>("T"), \
447         QuantizeAndDequantizeV2Op<CPUDevice, T>); \
448     REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4Grad") \
449         .Device(DEVICE_CPU) \
450         .TypeConstraint<T>("T"), \
451         QuantizeAndDequantizeV4GradientOp<CPUDevice, T>); \
452     REGISTER_KERNEL_BUILDER( \
453         Name("QuantizeAndDequantize").Device(DEVICE_CPU).TypeConstraint<T>("T"), \
454         QuantizeAndDequantizeOp<CPUDevice, T>);
455 TF_CALL_float(REGISTER_CPU_KERNEL);
456 TF_CALL_double(REGISTER_CPU_KERNEL);
457 #undef REGISTER_CPU_KERNEL
458
459 #if (defined(GOOGLE_CUDA) && GOOGLE_CUDA) || \
460     (defined(TENSORFLOW_USE_ROCM) && TENSORFLOW_USE_ROCM)
461 #define REGISTER_GPU_KERNEL(T) \
462     REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV2") \
463         .Device(DEVICE_GPU) \
464         .HostMemory("input_min") \
465         .HostMemory("input_max") \
466         .TypeConstraint<T>("T"), \
467         QuantizeAndDequantizeV2Op<GPUDevice, T>); \
468     REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV3") \
469         .Device(DEVICE_GPU) \
470         .HostMemory("input_min") \

```

```

471         .HostMemory("input_max")           \
472         .HostMemory("num_bits")             \
473         .TypeConstraint<T>("T"),            \
474         QuantizeAndDequantizeV3Op<GPUDevice, T>); \
475 REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4") \
476         .Device(DEVICE_GPU)                   \
477         .HostMemory("input_min")               \
478         .HostMemory("input_max")               \
479         .TypeConstraint<T>("T"),              \
480         QuantizeAndDequantizeV2Op<GPUDevice, T>); \
481 REGISTER_KERNEL_BUILDER(Name("QuantizeAndDequantizeV4Grad") \
482         .Device(DEVICE_GPU)                   \
483         .HostMemory("input_min")               \
484         .HostMemory("input_max")               \
485         .TypeConstraint<T>("T"),              \
486         QuantizeAndDequantizeV4GradientOp<GPUDevice, T>); \
487 REGISTER_KERNEL_BUILDER( \
488     Name("QuantizeAndDequantize").Device(DEVICE_GPU).TypeConstraint<T>("T"), \
489     QuantizeAndDequantizeOp<GPUDevice, T>);
490 TF_CALL_float(REGISTER_GPU_KERNEL);
491 TF_CALL_double(REGISTER_GPU_KERNEL);
492 #undef REGISTER_GPU_KERNEL
493 #endif // GOOGLE_CUDA || TENSORFLOW_USE_ROCM
494 } // namespace tensorflow

```