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

3 contributors 

390 lines (343 sloc) | 15.9 KB ...

```
1  /* Copyright 2016 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 #define EIGEN_USE_THREADS
16
17 #include <algorithm>
18 #include <cmath>
19 #include <random>
20 #include <vector>
21
22 #include "tensorflow/core/kernels/fractional_pool_common.h"
23
24 #include "third_party/eigen3/unsupported/Eigen/CXX11/Tensor"
25 #include "tensorflow/core/framework/numeric_op.h"
26 #include "tensorflow/core/framework/op_kernel.h"
27 #include "tensorflow/core/lib/random/random.h"
28 #include "tensorflow/core/platform/logging.h"
29 #include "tensorflow/core/platform/mutex.h"
```

```
#include "tensorflow/core/util/guarded_philox_random.h"

namespace tensorflow {
typedef Eigen::ThreadPoolDevice CPUDevice;

template <typename T>
class FractionalMaxPoolOp : public OpKernel {
 public:
  explicit FractionalMaxPoolOp(OpKernelConstruction* context)
      : OpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("pooling_ratio", &pooling_ratio_));
    OP_REQUIRES_OK(context, context->GetAttr("pseudo_random", &pseudo_random_));
    OP_REQUIRES_OK(context, context->GetAttr("overlapping", &overlapping_));

    OP_REQUIRES(context, pooling_ratio_.size() == 4,
                errors::InvalidArgument("pooling_ratio field must "
                                        "specify 4 dimensions"));

    OP_REQUIRES(
        context, pooling_ratio_[0] == 1 || pooling_ratio_[3] == 1,
        errors::Unimplemented("Fractional max pooling is not yet "
                              "supported on the batch nor channel dimension."));

    OP_REQUIRES_OK(context, context->GetAttr("deterministic", &deterministic_));
    OP_REQUIRES_OK(context, context->GetAttr("seed", &seed_));
    OP_REQUIRES_OK(context, context->GetAttr("seed2", &seed2_));
    if (deterministic_) {
      // If both seeds are not set when deterministic_ is true, force set seeds.
      if ((seed_ == 0) && (seed2_ == 0)) {
        seed_ = random::New64();
        seed2_ = random::New64();
      }
    } else {
      OP_REQUIRES(
          context, (seed_ == 0) && (seed2_ == 0),
          errors::InvalidArgument(
              "Both seed and seed2 should be 0 if deterministic is false."));
    }
  }

  void Compute(OpKernelContext* context) override {
    typedef Eigen::Map<const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>>
        ConstEigenMatrixMap;
    typedef Eigen::Map<Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>>
        EigenMatrixMap;

    constexpr int tensor_in_and_out_dims = 4;

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

```
    OP_REQUIRES(context, tensor_in.dims() == tensor_in_and_out_dims,
                errors::InvalidArgument("tensor_in must be 4-dimensional"));

    std::vector<int> input_size(tensor_in_and_out_dims);
    std::vector<int> output_size(tensor_in_and_out_dims);
    for (int i = 0; i < tensor_in_and_out_dims; ++i) {
      input_size[i] = tensor_in.dim_size(i);
    }
    // Output size.
    for (int i = 0; i < tensor_in_and_out_dims; ++i) {
      // This must match the same logic in the shape function in
      // core/ops/nn_ops.cc.
      output_size[i] =
          static_cast<int>(std::floor(input_size[i] / pooling_ratio_[i]));
      DCHECK_GT(output_size[i], 0);
    }

    // Generate pooling sequence.
    std::vector<int64_t> height_cum_seq;
    std::vector<int64_t> width_cum_seq;
    GuardedPhiloxRandom generator;
    generator.Init(seed_, seed2_);
    height_cum_seq = GeneratePoolingSequence(input_size[1], output_size[1],
                                             &generator, pseudo_random_);
    width_cum_seq = GeneratePoolingSequence(input_size[2], output_size[2],
                                            &generator, pseudo_random_);

    // Prepare output.
    Tensor* output_tensor = nullptr;
    OP_REQUIRES_OK(context, context->allocate_output(
                                0,
                                TensorShape({output_size[0], output_size[1],
                                             output_size[2], output_size[3]}),
                                &output_tensor));
    Tensor* output_height_seq_tensor = nullptr;
    OP_REQUIRES_OK(
        context,
        context->allocate_output(
            1, TensorShape({static_cast<int64_t>(height_cum_seq.size())}),
            &output_height_seq_tensor));
    Tensor* output_width_seq_tensor = nullptr;
    OP_REQUIRES_OK(
        context,
        context->allocate_output(
            2, TensorShape({static_cast<int64_t>(width_cum_seq.size())}),
            &output_width_seq_tensor));

    ConstEigenMatrixMap in_mat(tensor_in.flat<T>().data(), input_size[3],
                               input_size[2] * input_size[1] * input_size[0]);
```

```
128
129     EigenMatrixMap out_mat(output_tensor->flat<T>().data(), output_size[3],
130                            output_size[2] * output_size[1] * output_size[0]);
131
132     // Initializes the output tensor with MIN<T>.
133     output_tensor->flat<T>().setConstant(Eigen::NumTraits<T>::lowest());
134
135     auto output_height_seq_flat = output_height_seq_tensor->flat<int64_t>();
136     auto output_width_seq_flat = output_width_seq_tensor->flat<int64_t>();
137
138     // Set output tensors.
139     for (int i = 0; i < height_cum_seq.size(); ++i) {
140       output_height_seq_flat(i) = height_cum_seq[i];
141     }
142
143     for (int i = 0; i < width_cum_seq.size(); ++i) {
144       output_width_seq_flat(i) = width_cum_seq[i];
145     }
146
147     // For both input and output,
148     // 0: batch
149     // 1: height / row
150     // 2: width / col
151     // 3: depth / channel
152     const int64_t height_max = input_size[1] - 1;
153     const int64_t width_max = input_size[2] - 1;
154     for (int64_t b = 0; b < input_size[0]; ++b) {
155       // height sequence.
156       for (int64_t hs = 0; hs < height_cum_seq.size() - 1; ++hs) {
157         // height start and end.
158         const int64_t height_start = height_cum_seq[hs];
159         int64_t height_end =
160             overlapping_ ? height_cum_seq[hs + 1] : height_cum_seq[hs + 1] - 1;
161         height_end = std::min(height_end, height_max);
162
163         // width sequence.
164         for (int64_t ws = 0; ws < width_cum_seq.size() - 1; ++ws) {
165           const int64_t out_offset =
166               (b * output_size[1] + hs) * output_size[2] + ws;
167           // width start and end.
168           const int64_t width_start = width_cum_seq[ws];
169           int64_t width_end =
170               overlapping_ ? width_cum_seq[ws + 1] : width_cum_seq[ws + 1] - 1;
171           width_end = std::min(width_end, width_max);
172           for (int64_t h = height_start; h <= height_end; ++h) {
173             for (int64_t w = width_start; w <= width_end; ++w) {
174               const int64_t in_offset =
175                   (b * input_size[1] + h) * input_size[2] + w;
176               out_mat.col(out_offset) =
```

```
                out_mat.col(out_offset).cwiseMax(in_mat.col(in_offset));
            }
          }
        }
      }
    }

 private:
  bool deterministic_;
  int64_t seed_;
  int64_t seed2_;
  std::vector<float> pooling_ratio_;
  bool pseudo_random_;
  bool overlapping_;
};

#define REGISTER_FRACTIONALMAXPOOL(type)                                \
  REGISTER_KERNEL_BUILDER(                                              \
      Name("FractionalMaxPool").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
      FractionalMaxPoolOp<type>)

REGISTER_FRACTIONALMAXPOOL(int32);
REGISTER_FRACTIONALMAXPOOL(int64_t);
REGISTER_FRACTIONALMAXPOOL(float);
REGISTER_FRACTIONALMAXPOOL(double);

#undef REGISTER_FRACTIONALMAXPOOL

static const int kInvalidMaxPoolingIndex = -1;

template <class T>
class FractionalMaxPoolGradOp : public OpKernel {
 public:
  explicit FractionalMaxPoolGradOp(OpKernelConstruction* context)
      : OpKernel(context) {
    OP_REQUIRES_OK(context, context->GetAttr("overlapping", &overlapping_));
  }

  void Compute(OpKernelContext* context) override {
    // There are two steps when calculating gradient for FractionalMaxPool.
    // 1) Walk through the process of calculating fractional pooling given
    //    pooling region; however, in the process, keep track of where the max
    //    element comes from. (arg_max)
    // 2) Populate the value of out_backprop to where arg_max indicates. If
    //    we support overlapping, it is likely to have multiple out_backprop[i]
    //    propagates back to the same arg_max value.
    typedef Eigen::Map<const Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>>
        ConstEigenMatrixMap;
```

```
226     typedef Eigen::Map<Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>>
227         EigenMatrixMap;
228     typedef Eigen::Map<Eigen::Matrix<int64, Eigen::Dynamic, Eigen::Dynamic>>
229         EigenIndexMatrixMap;
230
231     const Tensor& tensor_in = context->input(0);
232     const Tensor& tensor_out = context->input(1);
233     const Tensor& out_backprop = context->input(2);
234     const Tensor& height_seq_tensor = context->input(3);
235     const Tensor& width_seq_tensor = context->input(4);
236
237     // Just to make it similar to FractionalMaxPoolOp.
238     constexpr int tensor_in_and_out_dims = 4;
239     OP_REQUIRES(
240         context, tensor_in.dims() == tensor_in_and_out_dims,
241         errors::InvalidArgument("orig_input should be a tensor of rank 4, got ",
242                                 tensor_in.DebugString()));
243     OP_REQUIRES(context, tensor_in.NumElements() > 0,
244                 errors::InvalidArgument("orig_input must not be empty, got ",
245                                         tensor_in.DebugString()));
246     OP_REQUIRES(context, tensor_out.dims() == tensor_in_and_out_dims,
247                 errors::InvalidArgument(
248                     "orig_output should be a tensor of rank 4, got ",
249                     tensor_out.DebugString()));
250     OP_REQUIRES(context, tensor_out.NumElements() > 0,
251                 errors::InvalidArgument("orig_output must not be empty, got ",
252                                         tensor_out.DebugString()));
253     std::vector<int64_t> input_size(tensor_in_and_out_dims);
254     std::vector<int64_t> output_size(tensor_in_and_out_dims);
255     for (int i = 0; i < tensor_in_and_out_dims; ++i) {
256       input_size[i] = tensor_in.dim_size(i);
257     }
258     for (int i = 0; i < tensor_in_and_out_dims; ++i) {
259       output_size[i] = tensor_out.dim_size(i);
260     }
261
262     // ---------
263     // Step 1
264     // ---------
265     Tensor tensor_out_dup;
266     OP_REQUIRES_OK(context, context->forward_input_or_allocate_temp(
267                                 {1}, DataTypeToEnum<T>::v(), tensor_out.shape(),
268                                 &tensor_out_dup));
269     Tensor tensor_out_arg_max;
270     OP_REQUIRES_OK(context, context->allocate_temp(DataTypeToEnum<int64_t>::v(),
271                                                    tensor_out.shape(),
272                                                    &tensor_out_arg_max));
273     // Find arg_max for each tensor_out
274     ConstEigenMatrixMap tensor_in_mat(
```

```
275         tensor_in.flat<T>().data(), input_size[3],
276         input_size[2] * input_size[1] * input_size[0]);
277     EigenMatrixMap tensor_out_dup_mat(
278         tensor_out_dup.flat<T>().data(), output_size[3],
279         output_size[2] * output_size[1] * output_size[0]);
280     EigenIndexMatrixMap tensor_out_arg_max_mat(
281         tensor_out_arg_max.flat<int64_t>().data(), output_size[3],
282         output_size[2] * output_size[1] * output_size[0]);
283 
284     tensor_out_arg_max.flat<int64_t>().setConstant(kInvalidMaxPoolingIndex);
285     // Initializes the duplicate output tensor with MIN<T>.
286     tensor_out_dup.flat<T>().setConstant(Eigen::NumTraits<T>::lowest());
287 
288     auto height_seq_tensor_flat = height_seq_tensor.flat<int64_t>();
289     auto width_seq_tensor_flat = width_seq_tensor.flat<int64_t>();
290 
291     // Now walk through the process of fractional max pooling again.
292     // For both input and output,
293     // 0: batch
294     // 1: height / row
295     // 2: width / col
296     // 3: depth / channel
297     const int64_t height_max = input_size[1] - 1;
298     const int64_t width_max = input_size[2] - 1;
299     for (int64_t b = 0; b < input_size[0]; ++b) {
300       // height sequence.
301       for (int64_t hs = 0; hs < height_seq_tensor.dim_size(0) - 1; ++hs) {
302         // height start and end.
303         const int64_t height_start = height_seq_tensor_flat(hs);
304         int64_t height_end = overlapping_ ? height_seq_tensor_flat(hs + 1)
305                                           : height_seq_tensor_flat(hs + 1) - 1;
306         height_end = std::min(height_end, height_max);
307 
308         // width sequence.
309         for (int64_t ws = 0; ws < width_seq_tensor.dim_size(0) - 1; ++ws) {
310           const int64_t out_index =
311               (b * output_size[1] + hs) * output_size[2] + ws;
312           // width start and end.
313           const int64_t width_start = width_seq_tensor_flat(ws);
314           int64_t width_end = overlapping_ ? width_seq_tensor_flat(ws + 1)
315                                            : width_seq_tensor_flat(ws + 1) - 1;
316           width_end = std::min(width_end, width_max);
317           for (int64_t h = height_start; h <= height_end; ++h) {
318             for (int64_t w = width_start; w <= width_end; ++w) {
319               const int64_t in_index =
320                   (b * input_size[1] + h) * input_size[2] + w;
321               // Walk through each channel (depth).
322               for (int64_t d = 0; d < input_size[3]; ++d) {
323                 const T& input_ref = tensor_in_mat.coeffRef(d, in_index);
```

```
                T& output_ref = tensor_out_dup_mat.coeffRef(d, out_index);
                int64_t& out_arg_max_ref =
                    tensor_out_arg_max_mat.coeffRef(d, out_index);
                if (output_ref < input_ref ||
                    out_arg_max_ref == kInvalidMaxPoolingIndex) {
                  output_ref = input_ref;
                  int input_offset = in_index * input_size[3] + d;
                  out_arg_max_ref = input_offset;
                }
              }
            }
          }
        }
      }

    // Check tensor_out_dup is the same as tensor_out.
    ConstEigenMatrixMap tensor_out_mat(
        tensor_out.flat<T>().data(), output_size[3],
        output_size[2] * output_size[1] * output_size[0]);
    const int64_t num_reshaped_cols =
        output_size[2] * output_size[1] * output_size[0];
    for (int64_t i = 0; i < num_reshaped_cols; ++i) {
      for (int64_t j = 0; j < output_size[3]; ++j) {
        DCHECK_EQ(tensor_out_dup_mat(j, i), tensor_out_mat(j, i));
      }
    }

    Tensor* output = nullptr;
    OP_REQUIRES_OK(context, context->forward_input_or_allocate_output(
                                {0}, 0, tensor_in.shape(), &output));
    output->flat<T>().setZero();

    auto out_backprop_flat = out_backprop.flat<T>();
    auto input_backprop_flat = output->flat<T>();
    auto out_arg_max_flat = tensor_out_arg_max.flat<int64_t>();
    int num_total_outputs = out_backprop_flat.size();
    int num_total_inputs = input_backprop_flat.size();

    for (int index = 0; index < num_total_outputs; ++index) {
      int input_backprop_index = out_arg_max_flat(index);
      // According to maxpooling_op.cc, the performance impact below is small.
      CHECK(input_backprop_index >= 0 &&
            input_backprop_index < num_total_inputs)
          << "Invalid input backprop index: " << input_backprop_index << ", "
          << num_total_inputs;
      input_backprop_flat(input_backprop_index) += out_backprop_flat(index);
    }
  }
```

```
373
374  private:
375   bool overlapping_;
376 };
377
378 #define REGISTER_FRACTIONALMAXPOOLGRAD(type)              \
379   REGISTER_KERNEL_BUILDER(Name("FractionalMaxPoolGrad")   \
380                               .Device(DEVICE_CPU)         \
381                               .TypeConstraint<type>("T"), \
382                           FractionalMaxPoolGradOp<type>)
383
384 REGISTER_FRACTIONALMAXPOOLGRAD(int32);
385 REGISTER_FRACTIONALMAXPOOLGRAD(int64_t);
386 REGISTER_FRACTIONALMAXPOOLGRAD(float);
387 REGISTER_FRACTIONALMAXPOOLGRAD(double);
388
389 #undef REGISTER_FRACTIONALMAXPOOLGRAD
390 }  // namespace tensorflow
```