penpornk Prevent out-of-bound accesses in SparseBincount. ... ✓ History

9 contributors

526 lines (467 sloc) | 19.3 KB ...

```
1  /* Copyright 2017 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 // See docs in ../ops/math_ops.cc.
17
18 #include "tensorflow/core/platform/errors.h"
19 #define EIGEN_USE_THREADS
20
21 #include "tensorflow/core/framework/op_kernel.h"
22 #include "tensorflow/core/framework/register_types.h"
23 #include "tensorflow/core/framework/types.h"
24 #include "tensorflow/core/kernels/bincount_op.h"
25 #include "tensorflow/core/kernels/fill_functor.h"
26 #include "tensorflow/core/lib/core/threadpool.h"
27 #include "tensorflow/core/platform/types.h"
28 #include "tensorflow/core/util/determinism.h"
29
```

```
namespace tensorflow {

using thread::ThreadPool;

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

namespace functor {

template <typename Tidx, typename T>
struct BincountFunctor<CPUDevice, Tidx, T, true> {
  static Status Compute(OpKernelContext* context,
                        const typename TTypes<Tidx, 1>::ConstTensor& arr,
                        const typename TTypes<T, 1>::ConstTensor& weights,
                        typename TTypes<T, 1>::Tensor& output,
                        const Tidx num_bins) {
    Tensor all_nonneg_t;
    TF_RETURN_IF_ERROR(context->allocate_temp(
        DT_BOOL, TensorShape({}), &all_nonneg_t, AllocatorAttributes()));
    all_nonneg_t.scalar<bool>().device(context->eigen_cpu_device()) =
        (arr >= Tidx(0)).all();
    if (!all_nonneg_t.scalar<bool>()()) {
      return errors::InvalidArgument("Input arr must be non-negative!");
    }

    // Allocate partial output bin sums for each worker thread. Worker ids in
    // ParallelForWithWorkerId range from 0 to NumThreads() inclusive.
    ThreadPool* thread_pool =
        context->device()->tensorflow_cpu_worker_threads()->workers;
    const int64_t num_threads = thread_pool->NumThreads() + 1;
    Tensor partial_bins_t;
    TF_RETURN_IF_ERROR(context->allocate_temp(
        DT_BOOL, TensorShape({num_threads, num_bins}), &partial_bins_t));
    auto partial_bins = partial_bins_t.matrix<bool>();
    partial_bins.setZero();
    thread_pool->ParallelForWithWorkerId(
        arr.size(), 8 /* cost */,
        [&](int64_t start_ind, int64_t limit_ind, int64_t worker_id) {
          for (int64_t i = start_ind; i < limit_ind; i++) {
            Tidx value = arr(i);
            if (value < num_bins) {
              partial_bins(worker_id, value) = true;
            }
          }
        });

    // Sum the partial bins along the 0th axis.
    Eigen::array<int, 1> reduce_dim({0});
    output.device(context->eigen_cpu_device()) =
```

```
        partial_bins.any(reduce_dim).cast<T>();
    return Status::OK();
  }
};

template <typename Tidx, typename T>
struct BincountFunctor<CPUDevice, Tidx, T, false> {
  static Status Compute(OpKernelContext* context,
                        const typename TTypes<Tidx, 1>::ConstTensor& arr,
                        const typename TTypes<T, 1>::ConstTensor& weights,
                        typename TTypes<T, 1>::Tensor& output,
                        const Tidx num_bins) {
    Tensor all_nonneg_t;
    TF_RETURN_IF_ERROR(context->allocate_temp(
        DT_BOOL, TensorShape({}), &all_nonneg_t, AllocatorAttributes()));
    all_nonneg_t.scalar<bool>().device(context->eigen_cpu_device()) =
        (arr >= Tidx(0)).all();
    if (!all_nonneg_t.scalar<bool>()()) {
      return errors::InvalidArgument("Input arr must be non-negative!");
    }

    // Allocate partial output bin sums for each worker thread. Worker ids in
    // ParallelForWithWorkerId range from 0 to NumThreads() inclusive.
    ThreadPool* thread_pool =
        context->device()->tensorflow_cpu_worker_threads()->workers;
    const int64_t num_threads = thread_pool->NumThreads() + 1;
    const Tidx* arr_data = arr.data();
    const std::ptrdiff_t arr_size = arr.size();
    const T* weight_data = weights.data();
    if (weights.size() && weights.size() != arr_size) {
      return errors::InvalidArgument(
          "Input indices and weights must have the same size.");
    }
    if (num_threads == 1) {
      output.setZero();
      T* output_data = output.data();
      if (weights.size()) {
        for (int64_t i = 0; i < arr_size; i++) {
          const Tidx value = arr_data[i];
          if (value < num_bins) {
            output_data[value] += weight_data[i];
          }
        }
      } else {
        for (int64_t i = 0; i < arr_size; i++) {
          const Tidx value = arr_data[i];
          if (value < num_bins) {
            // Complex numbers don't support "++".
            output_data[value] += T(1);
```

```
          }
        }
      }
    } else {
      Tensor partial_bins_t;
      TF_RETURN_IF_ERROR(context->allocate_temp(
          DataTypeToEnum<T>::value, TensorShape({num_threads, num_bins}),
          &partial_bins_t));
      auto partial_bins = partial_bins_t.matrix<T>();
      partial_bins.setZero();
      thread_pool->ParallelForWithWorkerId(
          arr_size, 8 /* cost */,
          [&](int64_t start_ind, int64_t limit_ind, int64_t worker_id) {
            if (weights.size()) {
              for (int64_t i = start_ind; i < limit_ind; i++) {
                Tidx value = arr_data[i];
                if (value < num_bins) {
                  partial_bins(worker_id, value) += weight_data[i];
                }
              }
            } else {
              for (int64_t i = start_ind; i < limit_ind; i++) {
                Tidx value = arr_data[i];
                if (value < num_bins) {
                  // Complex numbers don't support "++".
                  partial_bins(worker_id, value) += T(1);
                }
              }
            }
          });

      // Sum the partial bins along the 0th axis.
      Eigen::array<int, 1> reduce_dim({0});
      output.device(context->eigen_cpu_device()) = partial_bins.sum(reduce_dim);
    }
    return Status::OK();
  }
};

template <typename Tidx, typename T, bool binary_output>
struct BincountReduceFunctor<CPUDevice, Tidx, T, binary_output> {
  static Status Compute(OpKernelContext* context,
                        const typename TTypes<Tidx, 2>::ConstTensor& in,
                        const typename TTypes<T, 2>::ConstTensor& weights,
                        typename TTypes<T, 2>::Tensor& out,
                        const Tidx num_bins) {
    const int num_rows = out.dimension(0);
    const int num_cols = in.dimension(1);
    ThreadPool* thread_pool =
```

```
        context->device()->tensorflow_cpu_worker_threads()->workers;
    thread_pool->ParallelForWithWorkerId(
        num_rows, 8 /* cost */,
        [&](int64_t start_row, int64_t end_row, int64_t worker_id) {
          for (int64_t i = start_row; i < end_row; ++i) {
            for (int64_t j = 0; j < num_cols; ++j) {
              Tidx value = in(i, j);
              if (value < num_bins) {
                if (binary_output) {
                  out(i, value) = T(1);
                } else {
                  if (weights.size()) {
                    out(i, value) += weights(i, j);
                  } else {
                    out(i, value) += T(1);
                  }
                }
              }
            }
          }
        });
    return Status::OK();
  }
};

}  // namespace functor

template <typename Device, typename T>
class BincountOp : public OpKernel {
 public:
  explicit BincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  void Compute(OpKernelContext* ctx) override {
    const Tensor& arr_t = ctx->input(0);
    const Tensor& size_tensor = ctx->input(1);
    OP_REQUIRES(ctx, size_tensor.dims() == 0,
                errors::InvalidArgument("Shape must be rank 0 but is rank ",
                                        size_tensor.dims()));
    int32_t size = size_tensor.scalar<int32_t>()();
    OP_REQUIRES(
        ctx, size >= 0,
        errors::InvalidArgument("size (", size, ") must be non-negative"));

    const Tensor& weights_t = ctx->input(2);
    const auto arr = arr_t.flat<int32_t>();
    const auto weights = weights_t.flat<T>();
    Tensor* output_t;
    OP_REQUIRES_OK(ctx,
                   ctx->allocate_output(0, TensorShape({size}), &output_t));
```

```
    auto output = output_t->flat<T>();
    OP_REQUIRES_OK(ctx,
                   functor::BincountFunctor<Device, int32_t, T, false>::Compute(
                       ctx, arr, weights, output, size));
  }
};

#define REGISTER_KERNELS(type)                                         \
  REGISTER_KERNEL_BUILDER(                                             \
      Name("Bincount").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
      BincountOp<CPUDevice, type>)

TF_CALL_NUMBER_TYPES(REGISTER_KERNELS);
#undef REGISTER_KERNELS

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM

#define REGISTER_KERNELS(type)                           \
  REGISTER_KERNEL_BUILDER(Name("Bincount")               \
                              .Device(DEVICE_GPU)        \
                              .HostMemory("size")        \
                              .TypeConstraint<type>("T"), \
                          BincountOp<GPUDevice, type>)

TF_CALL_int32(REGISTER_KERNELS);
TF_CALL_float(REGISTER_KERNELS);
#undef REGISTER_KERNELS

#endif  // GOOGLE_CUDA || TENSORFLOW_USE_ROCM

template <typename Device, typename Tidx, typename T>
class DenseBincountOp : public OpKernel {
 public:
  explicit DenseBincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("binary_output", &binary_output_));
    if (std::is_same<Device, GPUDevice>::value) {
      OP_REQUIRES(
          ctx, !OpDeterminismRequired(),
          errors::Unimplemented(
              "Determinism is not yet supported in GPU implementation of "
              "DenseBincount."));
    }
  }

  void Compute(OpKernelContext* ctx) override {
    const Tensor& data = ctx->input(0);
    OP_REQUIRES(ctx, data.dims() <= 2,
                errors::InvalidArgument(
                    "Shape must be at most rank 2 but is rank ", data.dims()));
```

```
275
276     const Tensor& size_t = ctx->input(1);
277     const Tensor& weights = ctx->input(2);
278
279     Tidx size = size_t.scalar<Tidx>()();
280     OP_REQUIRES(
281         ctx, size >= 0,
282         errors::InvalidArgument("size (", size, ") must be non-negative"));
283
284     Tensor* out_t;
285     functor::SetZeroFunctor<Device, T> fill;
286     if (data.dims() == 1) {
287       OP_REQUIRES_OK(ctx, ctx->allocate_output(0, TensorShape({size}), &out_t));
288       auto out = out_t->flat<T>();
289       fill(ctx->eigen_device<Device>(), out);
290       if (binary_output_) {
291         OP_REQUIRES_OK(
292             ctx, functor::BincountFunctor<Device, Tidx, T, true>::Compute(
293                      ctx, data.flat<Tidx>(), weights.flat<T>(), out, size));
294       } else {
295         OP_REQUIRES_OK(
296             ctx, functor::BincountFunctor<Device, Tidx, T, false>::Compute(
297                      ctx, data.flat<Tidx>(), weights.flat<T>(), out, size));
298       }
299     } else if (data.dims() == 2) {
300       const int64_t num_rows = data.dim_size(0);
301       auto weight_matrix =
302           (weights.NumElements() == 0)
303               ? weights.shaped<T, 2>(gtl::InlinedVector<int64_t, 2>(2, 0))
304               : weights.matrix<T>();
305       OP_REQUIRES_OK(
306           ctx, ctx->allocate_output(0, TensorShape({num_rows, size}), &out_t));
307       auto out = out_t->matrix<T>();
308       fill(ctx->eigen_device<Device>(), out_t->flat<T>());
309       if (binary_output_) {
310         OP_REQUIRES_OK(
311             ctx, functor::BincountReduceFunctor<Device, Tidx, T, true>::Compute(
312                      ctx, data.matrix<Tidx>(), weight_matrix, out, size));
313       } else {
314         OP_REQUIRES_OK(
315             ctx,
316             functor::BincountReduceFunctor<Device, Tidx, T, false>::Compute(
317                 ctx, data.matrix<Tidx>(), weight_matrix, out, size));
318       }
319     }
320   }
321
322  private:
323   bool binary_output_;
```

```
};

#define REGISTER_KERNELS(Tidx, T)                              \
  REGISTER_KERNEL_BUILDER(Name("DenseBincount")                \
                              .Device(DEVICE_CPU)              \
                              .TypeConstraint<T>("T")          \
                              .TypeConstraint<Tidx>("Tidx"), \
                          DenseBincountOp<CPUDevice, Tidx, T>);
#define REGISTER_CPU_KERNELS(T) \
  REGISTER_KERNELS(int32, T);   \
  REGISTER_KERNELS(int64_t, T);

TF_CALL_NUMBER_TYPES(REGISTER_CPU_KERNELS);
#undef REGISTER_CPU_KERNELS
#undef REGISTER_KERNELS

#if GOOGLE_CUDA || TENSORFLOW_USE_ROCM

#define REGISTER_KERNELS(Tidx, T)                              \
  REGISTER_KERNEL_BUILDER(Name("DenseBincount")                \
                              .Device(DEVICE_GPU)              \
                              .HostMemory("size")              \
                              .TypeConstraint<T>("T")          \
                              .TypeConstraint<Tidx>("Tidx"), \
                          DenseBincountOp<GPUDevice, Tidx, T>);
#define REGISTER_GPU_KERNELS(T) \
  REGISTER_KERNELS(int32, T);   \
  REGISTER_KERNELS(int64_t, T);

TF_CALL_int32(REGISTER_GPU_KERNELS);
TF_CALL_float(REGISTER_GPU_KERNELS);
#undef REGISTER_GPU_KERNELS
#undef REGISTER_KERNELS

#endif  // GOOGLE_CUDA || TENSORFLOW_USE_ROCM

template <typename Device, typename Tidx, typename T>
class SparseBincountOp : public OpKernel {
 public:
  explicit SparseBincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("binary_output", &binary_output_));
  }

  void Compute(OpKernelContext* ctx) override {
    const Tensor& indices = ctx->input(0);
    const auto values = ctx->input(1).flat<Tidx>();
    const Tensor& dense_shape = ctx->input(2);
    const Tensor& size_t = ctx->input(3);
    const auto weights = ctx->input(4).flat<T>();
```

```
373     const int64_t weights_size = weights.size();
374
375     Tidx size = size_t.scalar<Tidx>()();
376     OP_REQUIRES(
377         ctx, size >= 0,
378         errors::InvalidArgument("size (", size, ") must be non-negative"));
379
380     bool is_1d = dense_shape.NumElements() == 1;
381
382     Tensor* out_t;
383     functor::SetZeroFunctor<Device, T> fill;
384     if (is_1d) {
385       OP_REQUIRES_OK(ctx, ctx->allocate_output(0, TensorShape({size}), &out_t));
386       auto out = out_t->flat<T>();
387       fill(ctx->eigen_device<Device>(), out);
388       if (binary_output_) {
389         OP_REQUIRES_OK(ctx,
390                        functor::BincountFunctor<Device, Tidx, T, true>::Compute(
391                            ctx, values, weights, out, size));
392       } else {
393         OP_REQUIRES_OK(
394             ctx, functor::BincountFunctor<Device, Tidx, T, false>::Compute(
395                      ctx, values, weights, out, size));
396       }
397     } else {
398       const auto shape = dense_shape.flat<int64_t>();
399       const int64_t num_rows = shape(0);
400       OP_REQUIRES_OK(
401           ctx, ctx->allocate_output(0, TensorShape({num_rows, size}), &out_t));
402       const auto out = out_t->matrix<T>();
403       fill(ctx->eigen_device<Device>(), out_t->flat<T>());
404       const auto indices_mat = indices.matrix<int64_t>();
405       for (int64_t i = 0; i < indices_mat.dimension(0); ++i) {
406         const int64_t batch = indices_mat(i, 0);
407         const Tidx bin = values(i);
408         OP_REQUIRES(
409             ctx, batch < out.dimension(0),
410             errors::InvalidArgument("Index out of bound. `batch` (", batch,
411                                     ") must be less than the dimension size (",
412                                     out.dimension(0), ")."));
413         OP_REQUIRES(
414             ctx, bin < out.dimension(1),
415             errors::InvalidArgument("Index out ouf bound. `bin` (", bin,
416                                     ") must be less then the dimension size (",
417                                     out.dimension(1), ")."));
418         if (bin < size) {
419           if (binary_output_) {
420             out(batch, bin) = T(1);
421           } else {
```

```
            if (weights_size) {
              out(batch, bin) += weights(i);
            } else {
              out(batch, bin) += T(1);
            }
          }
        }
      }
    }

 private:
  bool binary_output_;
};

#define REGISTER_KERNELS(Tidx, T)                             \
  REGISTER_KERNEL_BUILDER(Name("SparseBincount")              \
                              .Device(DEVICE_CPU)             \
                              .TypeConstraint<T>("T")         \
                              .TypeConstraint<Tidx>("Tidx"), \
                          SparseBincountOp<CPUDevice, Tidx, T>);
#define REGISTER_CPU_KERNELS(T) \
  REGISTER_KERNELS(int32, T);   \
  REGISTER_KERNELS(int64_t, T);

TF_CALL_NUMBER_TYPES(REGISTER_CPU_KERNELS);
#undef REGISTER_CPU_KERNELS
#undef REGISTER_KERNELS

template <typename Device, typename Tidx, typename T>
class RaggedBincountOp : public OpKernel {
 public:
  explicit RaggedBincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("binary_output", &binary_output_));
  }

  void Compute(OpKernelContext* ctx) override {
    const auto splits = ctx->input(0).flat<int64_t>();
    const auto values = ctx->input(1).flat<Tidx>();
    const Tensor& size_t = ctx->input(2);
    const auto weights = ctx->input(3).flat<T>();
    const int64_t weights_size = weights.size();

    Tidx size = size_t.scalar<Tidx>()();
    OP_REQUIRES(
        ctx, size >= 0,
        errors::InvalidArgument("size (", size, ") must be non-negative"));

    int num_rows = splits.size() - 1;
```

```
    int num_values = values.size();
    int batch_idx = 0;

    OP_REQUIRES(ctx, splits(0) == 0,
                errors::InvalidArgument("Splits must start with 0, not with ",
                                        splits(0)));

    OP_REQUIRES(ctx, splits(num_rows) == num_values,
                errors::InvalidArgument(
                    "Splits must end with the number of values, got ",
                    splits(num_rows), " instead of ", num_values));

    Tensor* out_t;
    OP_REQUIRES_OK(
        ctx, ctx->allocate_output(0, TensorShape({num_rows, size}), &out_t));
    functor::SetZeroFunctor<Device, T> fill;
    fill(ctx->eigen_device<Device>(), out_t->flat<T>());
    const auto out = out_t->matrix<T>();

    for (int idx = 0; idx < num_values; ++idx) {
      while (idx >= splits(batch_idx)) {
        batch_idx++;
      }
      Tidx bin = values(idx);
      OP_REQUIRES(ctx, bin >= 0,
                  errors::InvalidArgument("Input must be non-negative"));
      if (bin < size) {
        if (binary_output_) {
          out(batch_idx - 1, bin) = T(1);
        } else {
          T value = (weights_size > 0) ? weights(idx) : T(1);
          out(batch_idx - 1, bin) += value;
        }
      }
    }
  }

 private:
  bool binary_output_;
};

#define REGISTER_KERNELS(Tidx, T)                              \
  REGISTER_KERNEL_BUILDER(Name("RaggedBincount")               \
                              .Device(DEVICE_CPU)              \
                              .TypeConstraint<T>("T")          \
                              .TypeConstraint<Tidx>("Tidx"), \
                          RaggedBincountOp<CPUDevice, Tidx, T>);
#define REGISTER_CPU_KERNELS(T) \
  REGISTER_KERNELS(int32, T);   \
```

```
520   REGISTER_KERNELS(int64_t, T);
521
522 TF_CALL_NUMBER_TYPES(REGISTER_CPU_KERNELS);
523 #undef REGISTER_CPU_KERNELS
524 #undef REGISTER_KERNELS
525
526 }  // end namespace tensorflow
```