

5100e359ae ▾

...

tensorflow / tensorflow / core / kernels / bincount\_op.cc



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

```

```

30 namespace tensorflow {
31
32 using thread::ThreadPool;
33
34 typedef Eigen::ThreadPoolDevice CPUDevice;
35 typedef Eigen::GpuDevice GPUDevice;
36
37 namespace functor {
38
39 template <typename Tidx, typename T>
40 struct BincountFunctor<CPUDevice, Tidx, T, true> {
41     static Status Compute(OpKernelContext* context,
42                          const typename TTypes<Tidx, 1>::ConstTensor& arr,
43                          const typename TTypes<T, 1>::ConstTensor& weights,
44                          typename TTypes<T, 1>::Tensor& output,
45                          const Tidx num_bins) {
46         Tensor all_nonneg_t;
47         TF_RETURN_IF_ERROR(context->allocate_temp(
48             DT_BOOL, TensorShape({}), &all_nonneg_t, AllocatorAttributes()));
49         all_nonneg_t.scalar<bool>().device(context->eigen_cpu_device()) =
50             (arr >= Tidx(0)).all();
51         if (!all_nonneg_t.scalar<bool>().val()) {
52             return errors::InvalidArgument("Input arr must be non-negative!");
53         }
54
55         // Allocate partial output bin sums for each worker thread. Worker ids in
56         // ParallelForWithWorkerId range from 0 to NumThreads() inclusive.
57         ThreadPool* thread_pool =
58             context->device()->tensorflow_cpu_worker_threads()->workers;
59         const int64_t num_threads = thread_pool->NumThreads() + 1;
60         Tensor partial_bins_t;
61         TF_RETURN_IF_ERROR(context->allocate_temp(
62             DT_BOOL, TensorShape({num_threads, num_bins}), &partial_bins_t));
63         auto partial_bins = partial_bins_t.matrix<bool>();
64         partial_bins.setZero();
65         thread_pool->ParallelForWithWorkerId(
66             arr.size(), 8 /* cost */,
67             [&](int64_t start_ind, int64_t limit_ind, int64_t worker_id) {
68                 for (int64_t i = start_ind; i < limit_ind; i++) {
69                     Tidx value = arr(i);
70                     if (value < num_bins) {
71                         partial_bins(worker_id, value) = true;
72                     }
73                 }
74             });
75
76         // Sum the partial bins along the 0th axis.
77         Eigen::array<int, 1> reduce_dim({0});
78         output.device(context->eigen_cpu_device()) =

```

```

79     partial_bins.any(reduce_dim).cast<T>();
80     return Status::OK();
81 }
82 };
83
84 template <typename Tidx, typename T>
85 struct BincountFunctor<CPUDevice, Tidx, T, false> {
86     static Status Compute(OpKernelContext* context,
87         const typename TTypes<Tidx, 1>::ConstTensor& arr,
88         const typename TTypes<T, 1>::ConstTensor& weights,
89         typename TTypes<T, 1>::Tensor& output,
90         const Tidx num_bins) {
91         Tensor all_nonneg_t;
92         TF_RETURN_IF_ERROR(context->allocate_temp(
93             DT_BOOL, TensorShape({}), &all_nonneg_t, AllocatorAttributes()));
94         all_nonneg_t.scalar<bool>().device(context->eigen_cpu_device()) =
95             (arr >= Tidx(0)).all();
96         if (!all_nonneg_t.scalar<bool>().()) {
97             return errors::InvalidArgument("Input arr must be non-negative!");
98         }
99
100         // Allocate partial output bin sums for each worker thread. Worker ids in
101         // ParallelForWithWorkerId range from 0 to NumThreads() inclusive.
102         ThreadPool* thread_pool =
103             context->device()->tensorflow_cpu_worker_threads()->workers;
104         const int64_t num_threads = thread_pool->NumThreads() + 1;
105         const Tidx* arr_data = arr.data();
106         const std::ptrdiff_t arr_size = arr.size();
107         const T* weight_data = weights.data();
108         if (weights.size() && weights.size() != arr_size) {
109             return errors::InvalidArgument(
110                 "Input indices and weights must have the same size.");
111         }
112         if (num_threads == 1) {
113             output.setZero();
114             T* output_data = output.data();
115             if (weights.size()) {
116                 for (int64_t i = 0; i < arr_size; i++) {
117                     const Tidx value = arr_data[i];
118                     if (value < num_bins) {
119                         output_data[value] += weight_data[i];
120                     }
121                 }
122             } else {
123                 for (int64_t i = 0; i < arr_size; i++) {
124                     const Tidx value = arr_data[i];
125                     if (value < num_bins) {
126                         // Complex numbers don't support "+=".
127                         output_data[value] += T(1);

```

```

128     }
129 }
130 }
131 } else {
132     Tensor partial_bins_t;
133     TF_RETURN_IF_ERROR(context->allocate_temp(
134         DataTypeToEnum<T>::value, TensorShape({num_threads, num_bins}),
135         &partial_bins_t));
136     auto partial_bins = partial_bins_t.matrix<T>();
137     partial_bins.setZero();
138     thread_pool->ParallelForWithWorkerId(
139         arr_size, 8 /* cost */,
140         [&](int64_t start_ind, int64_t limit_ind, int64_t worker_id) {
141             if (weights.size()) {
142                 for (int64_t i = start_ind; i < limit_ind; i++) {
143                     Tidx value = arr_data[i];
144                     if (value < num_bins) {
145                         partial_bins(worker_id, value) += weight_data[i];
146                     }
147                 }
148             } else {
149                 for (int64_t i = start_ind; i < limit_ind; i++) {
150                     Tidx value = arr_data[i];
151                     if (value < num_bins) {
152                         // Complex numbers don't support "+=".
153                         partial_bins(worker_id, value) += T(1);
154                     }
155                 }
156             }
157         });
158
159     // Sum the partial bins along the 0th axis.
160     Eigen::array<int, 1> reduce_dim({0});
161     output.device(context->eigen_cpu_device()) = partial_bins.sum(reduce_dim);
162 }
163 return Status::OK();
164 }
165 };
166
167 template <typename Tidx, typename T, bool binary_output>
168 struct BincountReduceFunctor<CPUDevice, Tidx, T, binary_output> {
169     static Status Compute(OpKernelContext* context,
170                          const typename TTypes<Tidx, 2>::ConstTensor& in,
171                          const typename TTypes<T, 2>::ConstTensor& weights,
172                          typename TTypes<T, 2>::Tensor& out,
173                          const Tidx num_bins) {
174         const int num_rows = out.dimension(0);
175         const int num_cols = in.dimension(1);
176         ThreadPool* thread_pool =

```

```

177     context->device()->tensorflow_cpu_worker_threads()->workers;
178     thread_pool->ParallelForWithWorkerId(
179         num_rows, 8 /* cost */,
180         [&](int64_t start_row, int64_t end_row, int64_t worker_id) {
181             for (int64_t i = start_row; i < end_row; ++i) {
182                 for (int64_t j = 0; j < num_cols; ++j) {
183                     Tidx value = in(i, j);
184                     if (value < num_bins) {
185                         if (binary_output) {
186                             out(i, value) = T(1);
187                         } else {
188                             if (weights.size()) {
189                                 out(i, value) += weights(i, j);
190                             } else {
191                                 out(i, value) += T(1);
192                             }
193                         }
194                     }
195                 }
196             }
197         });
198     return Status::OK();
199 }
200 };
201
202 } // namespace functor
203
204 template <typename Device, typename T>
205 class BincountOp : public OpKernel {
206 public:
207     explicit BincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
208
209     void Compute(OpKernelContext* ctx) override {
210         const Tensor& arr_t = ctx->input(0);
211         const Tensor& size_tensor = ctx->input(1);
212         OP_REQUIRES(ctx, size_tensor.dims() == 0,
213             errors::InvalidArgument("Shape must be rank 0 but is rank ",
214                                     size_tensor.dims()));
215         int32_t size = size_tensor.scalar<int32_t>();
216         OP_REQUIRES(
217             ctx, size >= 0,
218             errors::InvalidArgument("size (", size, ") must be non-negative"));
219
220         const Tensor& weights_t = ctx->input(2);
221         const auto arr = arr_t.flat<int32_t>();
222         const auto weights = weights_t.flat<T>();
223         Tensor* output_t;
224         OP_REQUIRES_OK(ctx,
225             ctx->allocate_output(0, TensorShape({size}), &output_t));

```

```

226     auto output = output_t->flat<T>();
227     OP_REQUIRES_OK(ctx,
228         functor::BincountFunctor<Device, int32_t, T, false>::Compute(
229             ctx, arr, weights, output, size));
230 }
231 };
232
233 #define REGISTER_KERNELS(type) \
234     REGISTER_KERNEL_BUILDER( \
235         Name("Bincount").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
236         BincountOp<CPUDevice, type>)
237
238 TF_CALL_NUMBER_TYPES(REGISTER_KERNELS);
239 #undef REGISTER_KERNELS
240
241 #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
242
243 #define REGISTER_KERNELS(type) \
244     REGISTER_KERNEL_BUILDER(Name("Bincount") \
245         .Device(DEVICE_GPU) \
246         .HostMemory("size") \
247         .TypeConstraint<type>("T"), \
248         BincountOp<GPUDevice, type>)
249
250 TF_CALL_int32(REGISTER_KERNELS);
251 TF_CALL_float(REGISTER_KERNELS);
252 #undef REGISTER_KERNELS
253
254 #endif // GOOGLE_CUDA || TENSORFLOW_USE_ROCM
255
256 template <typename Device, typename Tidx, typename T>
257 class DenseBincountOp : public OpKernel {
258 public:
259     explicit DenseBincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
260         OP_REQUIRES_OK(ctx, ctx->GetAttr("binary_output", &binary_output_));
261         if (std::is_same<Device, GPUDevice>::value) {
262             OP_REQUIRES(
263                 ctx, !OpDeterminismRequired(),
264                 errors::Unimplemented(
265                     "Determinism is not yet supported in GPU implementation of "
266                     "DenseBincount."));
267         }
268     }
269
270     void Compute(OpKernelContext* ctx) override {
271         const Tensor& data = ctx->input(0);
272         OP_REQUIRES(ctx, data.dims() <= 2,
273             errors::InvalidArgument(
274                 "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_;

```

```

324 };
325
326 #define REGISTER_KERNELS(Tidx, T) \
327     REGISTER_KERNEL_BUILDER(Name("DenseBincount") \
328         .Device(DEVICE_CPU) \
329         .TypeConstraint<T>("T") \
330         .TypeConstraint<Tidx>("Tidx"), \
331         DenseBincountOp<CPUDevice, Tidx, T>);
332 #define REGISTER_CPU_KERNELS(T) \
333     REGISTER_KERNELS(int32, T); \
334     REGISTER_KERNELS(int64_t, T);
335
336 TF_CALL_NUMBER_TYPES(REGISTER_CPU_KERNELS);
337 #undef REGISTER_CPU_KERNELS
338 #undef REGISTER_KERNELS
339
340 #if GOOGLE_CUDA || TENSORFLOW_USE_ROCM
341
342 #define REGISTER_KERNELS(Tidx, T) \
343     REGISTER_KERNEL_BUILDER(Name("DenseBincount") \
344         .Device(DEVICE_GPU) \
345         .HostMemory("size") \
346         .TypeConstraint<T>("T") \
347         .TypeConstraint<Tidx>("Tidx"), \
348         DenseBincountOp<GPUDevice, Tidx, T>);
349 #define REGISTER_GPU_KERNELS(T) \
350     REGISTER_KERNELS(int32, T); \
351     REGISTER_KERNELS(int64_t, T);
352
353 TF_CALL_int32(REGISTER_GPU_KERNELS);
354 TF_CALL_float(REGISTER_GPU_KERNELS);
355 #undef REGISTER_GPU_KERNELS
356 #undef REGISTER_KERNELS
357
358 #endif // GOOGLE_CUDA || TENSORFLOW_USE_ROCM
359
360 template <typename Device, typename Tidx, typename T>
361 class SparseBincountOp : public OpKernel {
362 public:
363     explicit SparseBincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
364         OP_REQUIRES_OK(ctx, ctx->GetAttr("binary_output", &binary_output_));
365     }
366
367     void Compute(OpKernelContext* ctx) override {
368         const Tensor& indices = ctx->input(0);
369         const auto values = ctx->input(1).flat<Tidx>();
370         const Tensor& dense_shape = ctx->input(2);
371         const Tensor& size_t = ctx->input(3);
372         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 of bound. `bin` (", bin,
416                                         ") must be less than the dimension size (",
417                                         out.dimension(1), ")."));
418             if (bin < size) {
419                 if (binary_output_) {
420                     out(batch, bin) = T(1);
421                 } else {

```

```

422         if (weights_size) {
423             out(batch, bin) += weights(i);
424         } else {
425             out(batch, bin) += T(1);
426         }
427     }
428 }
429 }
430 }
431 }
432
433 private:
434     bool binary_output_;
435 };
436
437 #define REGISTER_KERNELS(Tidx, T) \
438     REGISTER_KERNEL_BUILDER(Name("SparseBincount") \
439                             .Device(DEVICE_CPU) \
440                             .TypeConstraint<T>("T") \
441                             .TypeConstraint<Tidx>("Tidx"), \
442                             SparseBincountOp<CPUDevice, Tidx, T>);
443 #define REGISTER_CPU_KERNELS(T) \
444     REGISTER_KERNELS(int32, T); \
445     REGISTER_KERNELS(int64_t, T);
446
447 TF_CALL_NUMBER_TYPES(REGISTER_CPU_KERNELS);
448 #undef REGISTER_CPU_KERNELS
449 #undef REGISTER_KERNELS
450
451 template <typename Device, typename Tidx, typename T>
452 class RaggedBincountOp : public OpKernel {
453 public:
454     explicit RaggedBincountOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
455         OP_REQUIRES_OK(ctx, ctx->GetAttr("binary_output", &binary_output_));
456     }
457
458     void Compute(OpKernelContext* ctx) override {
459         const auto splits = ctx->input(0).flat<int64_t>();
460         const auto values = ctx->input(1).flat<Tidx>();
461         const Tensor& size_t = ctx->input(2);
462         const auto weights = ctx->input(3).flat<T>();
463         const int64_t weights_size = weights.size();
464
465         Tidx size = size_t.scalar<Tidx>();
466         OP_REQUIRES(
467             ctx, size >= 0,
468             errors::InvalidArgument("size (", size, ") must be non-negative"));
469
470         int num_rows = splits.size() - 1;

```

```

471     int num_values = values.size();
472     int batch_idx = 0;
473
474     OP_REQUIRES(ctx, splits(0) == 0,
475                 errors::InvalidArgument("Splits must start with 0, not with ",
476                                         splits(0)));
477
478     OP_REQUIRES(ctx, splits(num_rows) == num_values,
479                 errors::InvalidArgument(
480                     "Splits must end with the number of values, got ",
481                     splits(num_rows), " instead of ", num_values));
482
483     Tensor* out_t;
484     OP_REQUIRES_OK(
485         ctx, ctx->allocate_output(0, TensorShape({num_rows, size}), &out_t));
486     functor::SetZeroFunctor<Device, T> fill;
487     fill(ctx->eigen_device<Device>(), out_t->flat<T>());
488     const auto out = out_t->matrix<T>();
489
490     for (int idx = 0; idx < num_values; ++idx) {
491         while (idx >= splits(batch_idx)) {
492             batch_idx++;
493         }
494         Tidx bin = values(idx);
495         OP_REQUIRES(ctx, bin >= 0,
496                     errors::InvalidArgument("Input must be non-negative"));
497         if (bin < size) {
498             if (binary_output_) {
499                 out(batch_idx - 1, bin) = T(1);
500             } else {
501                 T value = (weights_size > 0) ? weights(idx) : T(1);
502                 out(batch_idx - 1, bin) += value;
503             }
504         }
505     }
506 }
507
508 private:
509     bool binary_output_;
510 };
511
512 #define REGISTER_KERNELS(Tidx, T) \
513     REGISTER_KERNEL_BUILDER(Name("RaggedBincount") \
514                             .Device(DEVICE_CPU) \
515                             .TypeConstraint<T>("T") \
516                             .TypeConstraint<Tidx>("Tidx"), \
517                             RaggedBincountOp<CPUDevice, Tidx, T>);
518 #define REGISTER_CPU_KERNELS(T) \
519     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
```