tensorflower-gardener [tf.data] Change Cardinality() implementation to read c... ... History

9 contributors

582 lines (498 sloc) | 21.2 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 #include "tensorflow/core/kernels/data/experimental/threadpool_dataset_op.h"
16
17 #include <memory>
18
19 #include "tensorflow/core/data/dataset_utils.h"
20 #include "tensorflow/core/framework/dataset.h"
21 #include "tensorflow/core/framework/op_kernel.h"
22 #include "tensorflow/core/framework/resource_mgr.h"
23 #include "tensorflow/core/lib/core/refcount.h"
24 #include "tensorflow/core/lib/core/threadpool.h"
25 #include "tensorflow/core/platform/cpu_info.h"
26 #include "tensorflow/core/platform/stringprintf.h"
27 #include "tensorflow/core/platform/thread_annotations.h"
28 #include "tensorflow/core/util/work_sharder.h"
29
```

```
namespace tensorflow {
namespace data {
namespace experimental {

/* static */ constexpr const char* const
    MaxIntraOpParallelismDatasetOp::kDatasetType;
/* static */ constexpr const char* const
    MaxIntraOpParallelismDatasetOp::kDatasetOp;
/* static */ constexpr const char* const
    PrivateThreadPoolDatasetOp::kDatasetType;
/* static */ constexpr const char* const PrivateThreadPoolDatasetOp::kDatasetOp;

class ThreadPoolResource : public ResourceBase {
 public:
  ThreadPoolResource(Env* env, const ThreadOptions& thread_options,
                     const string& name, int num_threads, bool low_latency_hint,
                     int max_intra_op_parallelism)
      : thread_pool_(env, thread_options, name, num_threads, low_latency_hint),
        max_intra_op_parallelism_(max_intra_op_parallelism) {}

  // Schedules fn() for execution in the pool of threads.
  void Schedule(std::function<void()> fn) {
    if (max_intra_op_parallelism_ < 0) {
      thread_pool_.Schedule(std::move(fn));
    } else {
      thread_pool_.Schedule(std::bind(
          [this](std::function<void()> bound_fn) {
            // TODO(mrry): Consider moving this thread-local configuration to
            // the threads themselves.
            ScopedPerThreadMaxParallelism scope(max_intra_op_parallelism_);
            bound_fn();
          },
          std::move(fn)));
    }
  }

  int32 NumThreads() { return thread_pool_.NumThreads(); }

  string DebugString() const override { return "ThreadPoolResource"; }

 private:
  thread::ThreadPool thread_pool_;
  const int max_intra_op_parallelism_;
};

// Creates a handle to a ThreadPool resource. Note that we don't use
// ResourceOpKernel here because the ThreadPoolResource constructor requires
// access to `OpKernelContext::env()`, which isn't provided by
// `ResourceOpKernel<T>::CreateResource()`.
```

```
class ThreadPoolHandleOp : public OpKernel {
 public:
  explicit ThreadPoolHandleOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("display_name", &display_name_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("num_threads", &num_threads_));
    OP_REQUIRES_OK(ctx, ctx->GetAttr("max_intra_op_parallelism",
                                     &max_intra_op_parallelism_));
    OP_REQUIRES(
        ctx, num_threads_ > 0,
        errors::InvalidArgument("`num_threads` must be greater than zero."));
  }

  // The resource is deleted from the resource manager only when it is private
  // to kernel. Ideally the resource should be deleted when it is no longer held
  // by anyone, but it would break backward compatibility.
  ~ThreadPoolHandleOp() override {
    if (cinfo_.resource_is_private_to_kernel()) {
      if (!cinfo_.resource_manager()
               ->Delete<ThreadPoolResource>(cinfo_.container(), cinfo_.name())
               .ok()) {
        // Do nothing; the resource can have been deleted by session resets.
      }
    }
  }

  void Compute(OpKernelContext* ctx) override TF_LOCKS_EXCLUDED(mu_) {
    mutex_lock l(mu_);
    if (!initialized_) {
      ResourceMgr* mgr = ctx->resource_manager();
      OP_REQUIRES_OK(ctx, cinfo_.Init(mgr, def()));
      ThreadPoolResource* resource;
      OP_REQUIRES_OK(ctx, mgr->LookupOrCreate<ThreadPoolResource>(
                              cinfo_.container(), cinfo_.name(), &resource,
                              [this, ctx](ThreadPoolResource** ret)
                                  TF_EXCLUSIVE_LOCKS_REQUIRED(mu_) {
                                    *ret = new ThreadPoolResource(
                                        ctx->env(), {}, display_name_,
                                        num_threads_,
                                        /*low_latency_hint=*/false,
                                        max_intra_op_parallelism_);
                                    return Status::OK();
                                  }));
      initialized_ = true;
    }
    OP_REQUIRES_OK(ctx, MakeResourceHandleToOutput(
                            ctx, 0, cinfo_.container(), cinfo_.name(),
                            TypeIndex::Make<ThreadPoolResource>()));
  }

```

```
 private:
  mutex mu_;
  ContainerInfo cinfo_ TF_GUARDED_BY(mu_);
  bool initialized_ TF_GUARDED_BY(mu_) = false;
  string display_name_;
  int num_threads_;
  int max_intra_op_parallelism_;
};

class ThreadPoolDatasetOp : public UnaryDatasetOpKernel {
 public:
  explicit ThreadPoolDatasetOp(OpKernelConstruction* ctx)
      : UnaryDatasetOpKernel(ctx) {}

  void MakeDataset(OpKernelContext* ctx, DatasetBase* input,
                   DatasetBase** output) override {
    core::RefCountPtr<ThreadPoolResource> threadpool_resource;
    OP_REQUIRES_OK(ctx, LookupResource(ctx, HandleFromInput(ctx, 1),
                                       &threadpool_resource));
    *output = new Dataset(ctx, input, ctx->input(1), threadpool_resource.get());
  }

 private:
  class Dataset : public DatasetBase {
   public:
    Dataset(OpKernelContext* ctx, const DatasetBase* input,
            const Tensor& resource_handle, ThreadPoolResource* threadpool)
        : DatasetBase(DatasetContext(ctx)),
          input_(input),
          resource_handle_(resource_handle),
          threadpool_(threadpool) {
      input_->Ref();
      threadpool_->Ref();
    }

    ~Dataset() override {
      input_->Unref();
      threadpool_->Unref();
    }

    std::unique_ptr<IteratorBase> MakeIteratorInternal(
        const string& prefix) const override {
      return absl::make_unique<Iterator>(
          Iterator::Params{this, strings::StrCat(prefix, "::ThreadPool")});
    }

    const DataTypeVector& output_dtypes() const override {
      return input_->output_dtypes();
    }
```

```
177     const std::vector<PartialTensorShape>& output_shapes() const override {
178       return input_->output_shapes();
179     }
180
181     string DebugString() const override {
182       return "ThreadPoolDatasetOp::Dataset";
183     }
184
185     int64_t CardinalityInternal() const override {
186       return input_->Cardinality();
187     }
188
189     Status InputDatasets(
190         std::vector<const DatasetBase*>* inputs) const override {
191       inputs->push_back(input_);
192       return Status::OK();
193     }
194
195     Status CheckExternalState() const override {
196       return input_->CheckExternalState();
197     }
198
199    protected:
200     Status AsGraphDefInternal(SerializationContext* ctx,
201                               DatasetGraphDefBuilder* b,
202                               Node** output) const override {
203       Node* input_graph_node = nullptr;
204       TF_RETURN_IF_ERROR(b->AddInputDataset(ctx, input_, &input_graph_node));
205       Node* resource_handle_node = nullptr;
206       TF_RETURN_IF_ERROR(b->AddTensor(resource_handle_, &resource_handle_node));
207       TF_RETURN_IF_ERROR(b->AddDataset(
208           this, {input_graph_node, resource_handle_node}, output));
209       return Status::OK();
210     }
211
212    private:
213     class Iterator : public DatasetIterator<Dataset> {
214      public:
215       explicit Iterator(const Params& params)
216           : DatasetIterator<Dataset>(params) {}
217
218       Status Initialize(IteratorContext* ctx) override {
219         return dataset()->input_->MakeIterator(
220             IteratorContext(CreateParams(ctx)), this, prefix(), &input_impl_);
221       }
222
223       Status GetNextInternal(IteratorContext* ctx,
224                              std::vector<Tensor>* out_tensors,
225                              bool* end_of_sequence) override {
```

```
226         return input_impl_->GetNext(IteratorContext(CreateParams(ctx)),
227                                     out_tensors, end_of_sequence);
228       }
229
230      protected:
231       std::shared_ptr<model::Node> CreateNode(
232           IteratorContext* ctx, model::Node::Args args) const override {
233         return model::MakeKnownRatioNode(std::move(args),
234                                          /*ratio=*/1);
235       }
236
237       Status SaveInternal(SerializationContext* ctx,
238                           IteratorStateWriter* writer) override {
239         DCHECK(input_impl_ != nullptr);
240         TF_RETURN_IF_ERROR(SaveInput(ctx, writer, input_impl_));
241         return Status::OK();
242       }
243
244       Status RestoreInternal(IteratorContext* ctx,
245                              IteratorStateReader* reader) override {
246         TF_RETURN_IF_ERROR(RestoreInput(ctx, reader, input_impl_));
247         return Status::OK();
248       }
249
250      private:
251       IteratorContext::Params CreateParams(IteratorContext* ctx) {
252         ThreadPoolResource* pool = dataset()->threadpool_;
253         IteratorContext::Params params(ctx);
254         params.runner = [pool](std::function<void()> c) {
255           pool->Schedule(std::move(c));
256         };
257         params.runner_threadpool_size = pool->NumThreads();
258         return params;
259       }
260
261       std::unique_ptr<IteratorBase> input_impl_;
262     };
263
264     const DatasetBase* const input_;
265     const Tensor resource_handle_;
266     ThreadPoolResource* const threadpool_;
267   };
268 };
269
270 class MaxIntraOpParallelismDatasetOp::Dataset : public DatasetBase {
271  public:
272   Dataset(OpKernelContext* ctx, const DatasetBase* input,
273           int64_t max_intra_op_parallelism)
274       : Dataset(DatasetContext(ctx), input, max_intra_op_parallelism) {}
```

```
275
276   Dataset(DatasetContext&& ctx, const DatasetBase* input,
277           int64_t max_intra_op_parallelism)
278       : DatasetBase(std::move(ctx)),
279         input_(input),
280         max_intra_op_parallelism_(max_intra_op_parallelism),
281         traceme_metadata_(
282             {{"parallelism",
283               strings::Printf("%lld", static_cast<long long>(
284                                           max_intra_op_parallelism_))}}) {
285     input_->Ref();
286   }
287
288   ~Dataset() override { input_->Unref(); }
289
290   std::unique_ptr<IteratorBase> MakeIteratorInternal(
291       const string& prefix) const override {
292     return absl::make_unique<Iterator>(Iterator::Params{
293         this, strings::StrCat(prefix, "::MaxIntraOpParallelism")});
294   }
295
296   const DataTypeVector& output_dtypes() const override {
297     return input_->output_dtypes();
298   }
299   const std::vector<PartialTensorShape>& output_shapes() const override {
300     return input_->output_shapes();
301   }
302
303   string DebugString() const override {
304     return "MaxIntraOpParallelismDatasetOp::Dataset";
305   }
306
307   int64_t CardinalityInternal() const override { return input_->Cardinality(); }
308
309   Status InputDatasets(std::vector<const DatasetBase*>* inputs) const override {
310     inputs->clear();
311     inputs->push_back(input_);
312     return Status::OK();
313   }
314
315   Status CheckExternalState() const override {
316     return input_->CheckExternalState();
317   }
318
319  protected:
320   Status AsGraphDefInternal(SerializationContext* ctx,
321                             DatasetGraphDefBuilder* b,
322                             Node** output) const override {
323     Node* input_graph_node = nullptr;
```

```
    TF_RETURN_IF_ERROR(b->AddInputDataset(ctx, input_, &input_graph_node));
    Node* max_intra_op_parallelism_node = nullptr;
    TF_RETURN_IF_ERROR(b->AddScalar(max_intra_op_parallelism_,
                                    &max_intra_op_parallelism_node));
    TF_RETURN_IF_ERROR(b->AddDataset(
        this, {input_graph_node, max_intra_op_parallelism_node}, output));
    return Status::OK();
  }

 private:
  class Iterator : public DatasetIterator<Dataset> {
   public:
    explicit Iterator(const Params& params)
        : DatasetIterator<Dataset>(params) {}

    Status Initialize(IteratorContext* ctx) override {
      return dataset()->input_->MakeIterator(ctx, this, prefix(), &input_impl_);
    }

    Status GetNextInternal(IteratorContext* ctx,
                           std::vector<Tensor>* out_tensors,
                           bool* end_of_sequence) override {
      IteratorContext::Params params(ctx);
      auto max_parallelism = dataset()->max_intra_op_parallelism_;
      params.runner = RunnerWithMaxParallelism(*ctx->runner(), max_parallelism);
      return input_impl_->GetNext(IteratorContext{std::move(params)},
                                  out_tensors, end_of_sequence);
    }

   protected:
    std::shared_ptr<model::Node> CreateNode(
        IteratorContext* ctx, model::Node::Args args) const override {
      return model::MakeKnownRatioNode(std::move(args), /*ratio=*/1);
    }

    Status SaveInternal(SerializationContext* ctx,
                        IteratorStateWriter* writer) override {
      DCHECK(input_impl_ != nullptr);
      TF_RETURN_IF_ERROR(SaveInput(ctx, writer, input_impl_));
      return Status::OK();
    }

    Status RestoreInternal(IteratorContext* ctx,
                           IteratorStateReader* reader) override {
      TF_RETURN_IF_ERROR(RestoreInput(ctx, reader, input_impl_));
      return Status::OK();
    }

    TraceMeMetadata GetTraceMeMetadata() const override {
```

```
373       return dataset()->traceme_metadata_;
374     }
375
376    private:
377     std::unique_ptr<IteratorBase> input_impl_;
378   };
379
380   const DatasetBase* const input_;
381   const int64_t max_intra_op_parallelism_;
382   const TraceMeMetadata traceme_metadata_;
383 };
384
385 /* static */
386 void MaxIntraOpParallelismDatasetOp::MakeDatasetFromOptions(
387     OpKernelContext* ctx, DatasetBase* input, int32_t max_intra_op_parallelism,
388     DatasetBase** output) {
389   OP_REQUIRES(
390       ctx, max_intra_op_parallelism >= 0,
391       errors::InvalidArgument("`max_intra_op_parallelism` must be >= 0"));
392   *output = new Dataset(DatasetContext(DatasetContext::Params(
393                             {MaxIntraOpParallelismDatasetOp::kDatasetType,
394                              MaxIntraOpParallelismDatasetOp::kDatasetOp})),
395                         input, max_intra_op_parallelism);
396 }
397
398 void MaxIntraOpParallelismDatasetOp::MakeDataset(OpKernelContext* ctx,
399                                                  DatasetBase* input,
400                                                  DatasetBase** output) {
401   int64_t max_intra_op_parallelism;
402   OP_REQUIRES_OK(ctx,
403                  ParseScalarArgument<int64_t>(ctx, "max_intra_op_parallelism",
404                                               &max_intra_op_parallelism));
405   OP_REQUIRES(
406       ctx, max_intra_op_parallelism >= 0,
407       errors::InvalidArgument("`max_intra_op_parallelism` must be >= 0"));
408   *output = new Dataset(ctx, input, max_intra_op_parallelism);
409 }
410
411 class PrivateThreadPoolDatasetOp::Dataset : public DatasetBase {
412  public:
413   Dataset(OpKernelContext* ctx, const DatasetBase* input, int num_threads)
414       : Dataset(ctx, DatasetContext(ctx), input, num_threads) {}
415
416   Dataset(OpKernelContext* ctx, DatasetContext&& dataset_ctx,
417           const DatasetBase* input, int num_threads)
418       : DatasetBase(std::move(dataset_ctx)),
419         input_(input),
420         num_threads_(num_threads == 0 ? port::MaxParallelism() : num_threads),
421         traceme_metadata_(
```

```
422             {{"num_threads",
423               strings::Printf("%lld", static_cast<long long>(num_threads_))}}) {
424     thread_pool_ = absl::make_unique<thread::ThreadPool>(
425         ctx->env(), ThreadOptions{}, "data_private_threadpool", num_threads_);
426     input_->Ref();
427   }
428
429   ~Dataset() override { input_->Unref(); }
430
431   std::unique_ptr<IteratorBase> MakeIteratorInternal(
432       const string& prefix) const override {
433     return absl::make_unique<Iterator>(
434         Iterator::Params{this, strings::StrCat(prefix, "::PrivateThreadPool")});
435   }
436
437   const DataTypeVector& output_dtypes() const override {
438     return input_->output_dtypes();
439   }
440   const std::vector<PartialTensorShape>& output_shapes() const override {
441     return input_->output_shapes();
442   }
443
444   string DebugString() const override {
445     return "PrivateThreadPoolDatasetOp::Dataset";
446   }
447
448   int64_t CardinalityInternal() const override { return input_->Cardinality(); }
449
450   Status InputDatasets(std::vector<const DatasetBase*>* inputs) const override {
451     inputs->clear();
452     inputs->push_back(input_);
453     return Status::OK();
454   }
455
456   Status CheckExternalState() const override {
457     return input_->CheckExternalState();
458   }
459
460  protected:
461   Status AsGraphDefInternal(SerializationContext* ctx,
462                             DatasetGraphDefBuilder* b,
463                             Node** output) const override {
464     Node* input_graph_node = nullptr;
465     TF_RETURN_IF_ERROR(b->AddInputDataset(ctx, input_, &input_graph_node));
466     Node* num_threads_node = nullptr;
467     TF_RETURN_IF_ERROR(b->AddScalar(num_threads_, &num_threads_node));
468     TF_RETURN_IF_ERROR(
469         b->AddDataset(this, {input_graph_node, num_threads_node}, output));
470     return Status::OK();
```

```
471   }
472
473  private:
474   class Iterator : public DatasetIterator<Dataset> {
475    public:
476     explicit Iterator(const Params& params)
477         : DatasetIterator<Dataset>(params) {}
478
479     Status Initialize(IteratorContext* ctx) override {
480       return dataset()->input_->MakeIterator(ctx, this, prefix(), &input_impl_);
481     }
482
483     Status GetNextInternal(IteratorContext* ctx,
484                            std::vector<Tensor>* out_tensors,
485                            bool* end_of_sequence) override {
486       thread::ThreadPool* pool = dataset()->thread_pool_.get();
487       IteratorContext::Params params(ctx);
488       params.runner = [pool](std::function<void()> c) {
489         pool->Schedule(std::move(c));
490       };
491       params.runner_threadpool_size = dataset()->num_threads_;
492       return input_impl_->GetNext(IteratorContext{std::move(params)},
493                                   out_tensors, end_of_sequence);
494     }
495
496    protected:
497     std::shared_ptr<model::Node> CreateNode(
498         IteratorContext* ctx, model::Node::Args args) const override {
499       return model::MakeKnownRatioNode(std::move(args), /*ratio=*/1);
500     }
501
502     Status SaveInternal(SerializationContext* ctx,
503                         IteratorStateWriter* writer) override {
504       DCHECK(input_impl_ != nullptr);
505       TF_RETURN_IF_ERROR(SaveInput(ctx, writer, input_impl_));
506       return Status::OK();
507     }
508
509     Status RestoreInternal(IteratorContext* ctx,
510                            IteratorStateReader* reader) override {
511       TF_RETURN_IF_ERROR(RestoreInput(ctx, reader, input_impl_));
512       return Status::OK();
513     }
514
515     TraceMeMetadata GetTraceMeMetadata() const override {
516       return dataset()->traceme_metadata_;
517     }
518
519    private:
```

```
520       std::unique_ptr<IteratorBase> input_impl_;
521     };
522
523     const DatasetBase* const input_;
524     const int64_t num_threads_;
525     const TraceMeMetadata traceme_metadata_;
526     std::unique_ptr<thread::ThreadPool> thread_pool_;
527   };
528
529   /* static */
530   void PrivateThreadPoolDatasetOp::MakeDatasetFromOptions(OpKernelContext* ctx,
531                                                           DatasetBase* input,
532                                                           int32_t num_threads,
533                                                           DatasetBase** output) {
534     OP_REQUIRES(ctx, num_threads >= 0,
535                 errors::InvalidArgument("`num_threads` must be >= 0"));
536     *output = new Dataset(ctx,
537                           DatasetContext(DatasetContext::Params(
538                               {PrivateThreadPoolDatasetOp::kDatasetType,
539                                PrivateThreadPoolDatasetOp::kDatasetOp})),
540                           input, num_threads);
541   }
542
543   void PrivateThreadPoolDatasetOp::MakeDataset(OpKernelContext* ctx,
544                                                DatasetBase* input,
545                                                DatasetBase** output) {
546     int64_t num_threads = 0;
547     OP_REQUIRES_OK(
548         ctx, ParseScalarArgument<int64_t>(ctx, "num_threads", &num_threads));
549     OP_REQUIRES(ctx, num_threads >= 0,
550                 errors::InvalidArgument("`num_threads` must be >= 0"));
551     *output = new Dataset(ctx, input, num_threads);
552   }
553
554   namespace {
555
556   REGISTER_KERNEL_BUILDER(Name("MaxIntraOpParallelismDataset").Device(DEVICE_CPU),
557                           MaxIntraOpParallelismDatasetOp);
558   REGISTER_KERNEL_BUILDER(
559       Name("ExperimentalMaxIntraOpParallelismDataset").Device(DEVICE_CPU),
560       MaxIntraOpParallelismDatasetOp);
561
562   REGISTER_KERNEL_BUILDER(Name("PrivateThreadPoolDataset").Device(DEVICE_CPU),
563                           PrivateThreadPoolDatasetOp);
564   REGISTER_KERNEL_BUILDER(
565       Name("ExperimentalPrivateThreadPoolDataset").Device(DEVICE_CPU),
566       PrivateThreadPoolDatasetOp);
567
568   REGISTER_KERNEL_BUILDER(Name("ThreadPoolHandle").Device(DEVICE_CPU),
```

```
                        ThreadPoolHandleOp);
REGISTER_KERNEL_BUILDER(Name("ExperimentalThreadPoolHandle").Device(DEVICE_CPU),
                        ThreadPoolHandleOp);

REGISTER_KERNEL_BUILDER(Name("ThreadPoolDataset").Device(DEVICE_CPU),
                        ThreadPoolDatasetOp);
REGISTER_KERNEL_BUILDER(
    Name("ExperimentalThreadPoolDataset").Device(DEVICE_CPU),
    ThreadPoolDatasetOp);

}  // namespace
}  // namespace experimental
}  // namespace data
}  // namespace tensorflow
```