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

15 contributors  +3

687 lines (620 sloc) | 26.5 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 #ifndef TENSORFLOW_CORE_KERNELS_CWISE_OPS_COMMON_H_
17 #define TENSORFLOW_CORE_KERNELS_CWISE_OPS_COMMON_H_
18
19 // See docs in ../ops/math_ops.cc.
20 #define _USE_MATH_DEFINES
21 #include <cmath>
22
23 #define EIGEN_USE_THREADS
24
25 #include "tensorflow/core/platform/bfloat16.h"
26
27
28 #include "tensorflow/core/framework/op.h"
29 #include "tensorflow/core/framework/op_kernel.h"
```

```
#include "tensorflow/core/framework/tensor_types.h"
#include "tensorflow/core/framework/variant_op_registry.h"
#include "tensorflow/core/kernels/cwise_ops.h"
#include "tensorflow/core/kernels/cwise_ops_gradients.h"
#include "tensorflow/core/kernels/fill_functor.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/util/bcast.h"

namespace tensorflow {

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

class BinaryOpShared : public OpKernel {
 public:
  explicit BinaryOpShared(OpKernelConstruction* ctx, DataType out, DataType in);

 protected:
  struct BinaryOpState {
    // Sets up bcast with the shape of in0 and in1, ensures that the bcast
    // is valid, and if so, set out, either by allocating a new buffer using
    // ctx->output(...) or by creating an alias for an owned input buffer for
    // in-place computation.
    // Caller must check ctx->status() upon return for non-ok status.
    // If ctx->status().ok() is true, then out is guaranteed to be allocated.
    explicit BinaryOpState(OpKernelContext* ctx);

    const Tensor& in0;
    const Tensor& in1;

    BCast bcast;
    Tensor* out = nullptr;
    int64_t out_num_elements;

    int64_t in0_num_elements;
    int64_t in1_num_elements;

    int ndims;
    bool result;
  };

  void SetUnimplementedError(OpKernelContext* ctx);
  void SetComputeError(OpKernelContext* ctx);
};

// Coefficient-wise binary operations:
//   Device: E.g., CPUDevice, GPUDevice.
//   Functor: defined in cwise_ops.h. E.g., functor::add.
template <typename Device, typename Functor>
```

```
class BinaryOp : public BinaryOpShared {
 public:
  typedef typename Functor::in_type Tin;    // Input scalar data type.
  typedef typename Functor::out_type Tout;  // Output scalar data type.

  explicit BinaryOp(OpKernelConstruction* ctx)
      : BinaryOpShared(ctx, DataTypeToEnum<Tout>::v(),
                       DataTypeToEnum<Tin>::v()) {}

  void Compute(OpKernelContext* ctx) override {
    const Tensor& input_0 = ctx->input(0);
    const Tensor& input_1 = ctx->input(1);
    const Device& eigen_device = ctx->eigen_device<Device>();
    bool error = false;
    bool* const error_ptr = Functor::has_errors ? &error : nullptr;

    // NOTE: Handle three simple cases before building the BinaryOpState, which
    // is relatively expensive for small operations.
    if (input_0.shape() == input_1.shape()) {
      // tensor op tensor with no broadcasting.
      Tensor* out;
      OP_REQUIRES_OK(ctx, ctx->forward_input_or_allocate_output(
                              {0, 1}, 0, input_0.shape(), &out));
      functor::BinaryFunctor<Device, Functor, 1>()(
          eigen_device, out->template flat<Tout>(),
          input_0.template flat<Tin>(), input_1.template flat<Tin>(),
          error_ptr);
      if (Functor::has_errors && error) {
        SetComputeError(ctx);
      }
      return;
    } else if (input_0.shape().dims() == 0) {
      // scalar op tensor.
      Tensor* out;
      OP_REQUIRES_OK(ctx, ctx->forward_input_or_allocate_output(
                              {1}, 0, input_1.shape(), &out));

      functor::BinaryFunctor<Device, Functor, 1>().Left(
          eigen_device, out->template flat<Tout>(),
          input_0.template scalar<Tin>(), input_1.template flat<Tin>(),
          error_ptr);
      if (Functor::has_errors && error) {
        SetComputeError(ctx);
      }
      return;
    } else if (input_1.shape().dims() == 0) {
      // tensor op scalar.
      Tensor* out;
      OP_REQUIRES_OK(ctx, ctx->forward_input_or_allocate_output(
```

```
                                {0}, 0, input_0.shape(), &out));
      functor::BinaryFunctor<Device, Functor, 1>().Right(
          eigen_device, out->template flat<Tout>(),
          input_0.template flat<Tin>(), input_1.template scalar<Tin>(),
          error_ptr);
      if (Functor::has_errors && error) {
        SetComputeError(ctx);
      }
      return;
    }

    // 'state': Shared helper not dependent on T to reduce code size
    BinaryOpState state(ctx);
    if (ctx->status().code() == error::RESOURCE_EXHAUSTED) {
      // Stop when BinaryOpState's constructor failed due to OOM.
      return;
    }
    auto& bcast = state.bcast;
    Tensor* out = state.out;
    if (!bcast.IsValid()) {
      if (ctx->status().ok()) {
        if (state.result) {
          functor::SetOneFunctor<Device, bool>()(eigen_device,
                                                 out->flat<bool>());
        } else {
          functor::SetZeroFunctor<Device, bool>()(eigen_device,
                                                  out->flat<bool>());
        }
      }
      return;
    }

    auto& in0 = state.in0;
    auto& in1 = state.in1;
    if (state.out_num_elements == 0) {
      return;
    }

    const int ndims = state.ndims;
    if (ndims <= 1) {
      auto out_flat = out->flat<Tout>();
      if (state.in1_num_elements == 1) {
        // tensor op scalar
        functor::BinaryFunctor<Device, Functor, 1>().Right(
            eigen_device, out_flat, in0.template flat<Tin>(),
            in1.template scalar<Tin>(), error_ptr);
      } else if (state.in0_num_elements == 1) {
        // scalar op tensor
        functor::BinaryFunctor<Device, Functor, 1>().Left(
```

```
177             eigen_device, out_flat, in0.template scalar<Tin>(),
178             in1.template flat<Tin>(), error_ptr);
179       } else {
180         functor::BinaryFunctor<Device, Functor, 1>()(
181             eigen_device, out_flat, in0.template flat<Tin>(),
182             in1.template flat<Tin>(), error_ptr);
183       }
184     } else if (ndims == 2) {
185       functor::BinaryFunctor<Device, Functor, 2>().BCast(
186           eigen_device, out->shaped<Tout, 2>(bcast.result_shape()),
187           in0.template shaped<Tin, 2>(bcast.x_reshape()),
188           BCast::ToIndexArray<2>(bcast.x_bcast()),
189           in1.template shaped<Tin, 2>(bcast.y_reshape()),
190           BCast::ToIndexArray<2>(bcast.y_bcast()), error_ptr);
191     } else if (ndims == 3) {
192       functor::BinaryFunctor<Device, Functor, 3>().BCast(
193           eigen_device, out->shaped<Tout, 3>(bcast.result_shape()),
194           in0.template shaped<Tin, 3>(bcast.x_reshape()),
195           BCast::ToIndexArray<3>(bcast.x_bcast()),
196           in1.template shaped<Tin, 3>(bcast.y_reshape()),
197           BCast::ToIndexArray<3>(bcast.y_bcast()), error_ptr);
198     } else if (ndims == 4) {
199       functor::BinaryFunctor<Device, Functor, 4>().BCast(
200           eigen_device, out->shaped<Tout, 4>(bcast.result_shape()),
201           in0.template shaped<Tin, 4>(bcast.x_reshape()),
202           BCast::ToIndexArray<4>(bcast.x_bcast()),
203           in1.template shaped<Tin, 4>(bcast.y_reshape()),
204           BCast::ToIndexArray<4>(bcast.y_bcast()), error_ptr);
205     } else if (ndims == 5) {
206       functor::BinaryFunctor<Device, Functor, 5>().BCast(
207           eigen_device, out->shaped<Tout, 5>(bcast.result_shape()),
208           in0.template shaped<Tin, 5>(bcast.x_reshape()),
209           BCast::ToIndexArray<5>(bcast.x_bcast()),
210           in1.template shaped<Tin, 5>(bcast.y_reshape()),
211           BCast::ToIndexArray<5>(bcast.y_bcast()), error_ptr);
212     } else {
213       SetUnimplementedError(ctx);
214     }
215     if (Functor::has_errors && error) {
216       SetComputeError(ctx);
217     }
218   }
219 };
220
221 template <typename Device, typename T>
222 class ApproximateEqualOp : public OpKernel {
223  public:
224   explicit ApproximateEqualOp(OpKernelConstruction* context)
225       : OpKernel(context) {
```

```
    float tolerance;
    OP_REQUIRES_OK(context, context->GetAttr("tolerance", &tolerance));
    tolerance_ = T(tolerance);
  }
  void Compute(OpKernelContext* context) override {
    const Tensor& x_input = context->input(0);
    const Tensor& y_input = context->input(1);
    OP_REQUIRES(
        context, x_input.shape() == y_input.shape(),
        errors::InvalidArgument("x and y must be of the same shape. ",
                                "x shape: ", x_input.shape().DebugString(),
                                ". y shape: ", y_input.shape().DebugString()));
    Tensor* z_output = nullptr;
    OP_REQUIRES_OK(context,
                   context->allocate_output(0, x_input.shape(), &z_output));
    const Device& d = context->eigen_device<Device>();
    typename TTypes<T>::ConstFlat x(x_input.flat<T>());
    typename TTypes<T>::ConstFlat y(y_input.flat<T>());
    typename TTypes<bool>::Flat z(z_output->flat<bool>());
    functor::ApproximateEqual<Device, T>()(d, x, y, tolerance_, z);
  }

 private:
  T tolerance_;
};

// Basic coefficient-wise binary operations that are known to not require
// any broadcasting. This is the case for example of the gradients of
// unary operations.
//   Device: E.g., CPUDevice, GPUDevice.
//   Functor: defined above. E.g., functor::tanh_grad.
template <typename Device, typename Functor>
class SimpleBinaryOp : public OpKernel {
 public:
  typedef typename Functor::in_type Tin;    // Input scalar data type.
  typedef typename Functor::out_type Tout;  // Output scalar data type.

  explicit SimpleBinaryOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}

  void Compute(OpKernelContext* ctx) override {
    const Tensor& in0 = ctx->input(0);
    const Tensor& in1 = ctx->input(1);
    OP_REQUIRES(
        ctx, in0.NumElements() == in1.NumElements(),
        errors::InvalidArgument("The two arguments to a cwise op must have "
                                "same number of elements, got ",
                                in0.NumElements(), " and ", in1.NumElements()));
    auto in0_flat = in0.flat<Tin>();
    auto in1_flat = in1.flat<Tin>();
```

```
275     const Device& eigen_device = ctx->eigen_device<Device>();
276
277     Tensor* out = nullptr;
278     if (std::is_same<Tin, Tout>::value) {
279       OP_REQUIRES_OK(ctx, ctx->forward_input_or_allocate_output(
280                               {0, 1}, 0, in0.shape(), &out));
281     } else {
282       OP_REQUIRES_OK(ctx, ctx->allocate_output(0, in0.shape(), &out));
283     }
284     auto out_flat = out->flat<Tout>();
285     functor::SimpleBinaryFunctor<Device, Functor>()(eigen_device, out_flat,
286                                                     in0_flat, in1_flat);
287   }
288 };
289
290 // Coefficient-wise unary operations:
291 //   Device: E.g., CPUDevice, GPUDevice.
292 //   Functor: defined in cwise_ops.h. E.g., functor::sqrt.
293 template <typename Device, typename Functor>
294 class UnaryOp : public OpKernel {
295  public:
296   typedef typename Functor::in_type Tin;    // Input scalar data type.
297   typedef typename Functor::out_type Tout;  // Output scalar data type.
298   // Tin may be different from Tout. E.g., abs: complex64 -> float
299
300   explicit UnaryOp(OpKernelConstruction* ctx) : OpKernel(ctx) {
301     auto in = DataTypeToEnum<Tin>::v();
302     auto out = DataTypeToEnum<Tout>::v();
303     OP_REQUIRES_OK(ctx, ctx->MatchSignature({in}, {out}));
304   }
305
306   void Compute(OpKernelContext* ctx) override {
307     const Tensor& inp = ctx->input(0);
308     Tensor* out = nullptr;
309     if (std::is_same<Tin, Tout>::value) {
310       OP_REQUIRES_OK(ctx, ctx->forward_input_or_allocate_output(
311                               {0}, 0, inp.shape(), &out));
312     } else {
313       OP_REQUIRES_OK(ctx, ctx->allocate_output(0, inp.shape(), &out));
314     }
315     functor::UnaryFunctor<Device, Functor>()(
316         ctx->eigen_device<Device>(), out->flat<Tout>(), inp.flat<Tin>());
317   }
318 };
319
320 template <typename Device, VariantUnaryOp OpEnum>
321 class UnaryVariantOp : public OpKernel {
322  public:
323   explicit UnaryVariantOp(OpKernelConstruction* ctx) : OpKernel(ctx) {}
```

```
324
325   void Compute(OpKernelContext* ctx) override {
326     const Tensor& inp = ctx->input(0);
327     OP_REQUIRES(
328         ctx, TensorShapeUtils::IsScalar(inp.shape()),
329         errors::InvalidArgument("Non-scalar variants are not supported."));
330     const Variant& v = inp.scalar<Variant>()();
331     Variant v_out;
332     OP_REQUIRES_OK(ctx, UnaryOpVariant<Device>(ctx, OpEnum, v, &v_out));
333     int numa_node = ctx->device()->NumaNode();
334     Tensor out(cpu_allocator(numa_node), DT_VARIANT, TensorShape());
335     out.scalar<Variant>()() = std::move(v_out);
336     ctx->set_output(0, std::move(out));
337   }
338 };
339
340 namespace functor {
341
342 template <typename D, typename Out, typename Rhs>
343 void Assign(const D& d, Out out, Rhs rhs) {
344   out.device(d) = rhs;
345 }
346
347 // Partial specialization of BinaryFunctor<Device=CPUDevice, Functor, NDIMS>
348 // for functors with no error checking.
349 template <typename Functor, int NDIMS>
350 struct BinaryFunctor<CPUDevice, Functor, NDIMS, false> {
351   void operator()(const CPUDevice& d, typename Functor::tout_type out,
352                   typename Functor::tin_type in0,
353                   typename Functor::tin_type in1, bool* error) {
354     Assign(d, out, in0.binaryExpr(in1, typename Functor::func()));
355   }
356
357   void Left(const CPUDevice& d, typename Functor::tout_type out,
358             typename Functor::tscalar_type scalar,
359             typename Functor::tin_type in, bool* error) {
360     typedef typename Functor::out_type Tout;
361     typedef typename Functor::in_type Tin;
362     typedef typename Functor::func Binary;
363     typedef
364         typename Eigen::internal::scalar_left<Tout, Tin, Binary,
365                                               /*is_scalar_in_host_memory=*/true>
366             Unary;
367     Assign(d, out, in.unaryExpr(Unary(scalar.data())));
368   }
369
370   void Right(const CPUDevice& d, typename Functor::tout_type out,
371              typename Functor::tin_type in,
372              typename Functor::tscalar_type scalar, bool* error) {
```

```
373     typedef typename Functor::out_type Tout;
374     typedef typename Functor::in_type Tin;
375     typedef typename Functor::func Binary;
376     typedef typename Eigen::internal::scalar_right<
377         Tout, Tin, Binary, /*is_scalar_in_host_memory=*/true>
378         Unary;
379     Assign(d, out, in.unaryExpr(Unary(scalar.data())));
380   }
381
382   void BCast(const CPUDevice& dev,
383              typename TTypes<typename Functor::out_type, NDIMS>::Tensor out,
384              typename TTypes<typename Functor::in_type, NDIMS>::ConstTensor in0,
385              typename Eigen::array<Eigen::DenseIndex, NDIMS> bcast0,
386              typename TTypes<typename Functor::in_type, NDIMS>::ConstTensor in1,
387              typename Eigen::array<Eigen::DenseIndex, NDIMS> bcast1,
388              bool* error) {
389     typename Functor::func func;
390     if (AllOne<NDIMS>(bcast0) && AllOne<NDIMS>(bcast1)) {
391       Assign(dev, out, in0.binaryExpr(in1, func));
392     } else if (AllOne<NDIMS>(bcast0)) {
393       auto rhs = in1.broadcast(bcast1);
394       Assign(dev, out, in0.binaryExpr(rhs, func));
395     } else if (AllOne<NDIMS>(bcast1)) {
396       auto lhs = in0.broadcast(bcast0);
397       Assign(dev, out, lhs.binaryExpr(in1, func));
398     } else {
399       auto lhs = in0.broadcast(bcast0);
400       auto rhs = in1.broadcast(bcast1);
401       Assign(dev, out, lhs.binaryExpr(rhs, func));
402     }
403   }
404 };
405
406 // Partial specialization of BinaryFunctor<Device=CPUDevice, Functor, 2>
407 // for functors with no error checking.
408 template <typename Functor>
409 struct BinaryFunctor<CPUDevice, Functor, 2, false> {
410   enum { NDIMS = 2 };
411
412   void operator()(const CPUDevice& d, typename Functor::tout_type out,
413                   typename Functor::tin_type in0,
414                   typename Functor::tin_type in1, bool* error) {
415     Assign(d, out, in0.binaryExpr(in1, typename Functor::func()));
416   }
417
418   void Left(const CPUDevice& d, typename Functor::tout_type out,
419             typename Functor::tscalar_type scalar,
420             typename Functor::tin_type in, bool* error) {
421     typedef typename Functor::out_type Tout;
```

```
422     typedef typename Functor::in_type Tin;
423     typedef typename Functor::func Binary;
424     typedef
425         typename Eigen::internal::scalar_left<Tout, Tin, Binary,
426                                               /*is_scalar_in_host_memory=*/true>
427             Unary;
428     Assign(d, out, in.unaryExpr(Unary(scalar.data())));
429   }
430
431   void Right(const CPUDevice& d, typename Functor::tout_type out,
432              typename Functor::tin_type in,
433              typename Functor::tscalar_type scalar, bool* error) {
434     typedef typename Functor::out_type Tout;
435     typedef typename Functor::in_type Tin;
436     typedef typename Functor::func Binary;
437     typedef typename Eigen::internal::scalar_right<
438         Tout, Tin, Binary, /*is_scalar_in_host_memory=*/true>
439         Unary;
440     Assign(d, out, in.unaryExpr(Unary(scalar.data())));
441   }
442
443 #if !defined(EIGEN_HAS_INDEX_LIST)
444   inline Eigen::DSizes<int, 2> NByOne(int n) {
445     return Eigen::DSizes<int, 2>(n, 1);
446   }
447   inline Eigen::DSizes<int, 2> OneByM(int m) {
448     return Eigen::DSizes<int, 2>(1, m);
449   }
450 #else
451   inline Eigen::IndexList<int, Eigen::type2index<1>> NByOne(int n) {
452     Eigen::IndexList<int, Eigen::type2index<1>> ret;
453     ret.set(0, n);
454     return ret;
455   }
456   inline Eigen::IndexList<Eigen::type2index<1>, int> OneByM(int m) {
457     Eigen::IndexList<Eigen::type2index<1>, int> ret;
458     ret.set(1, m);
459     return ret;
460   }
461 #endif
462
463   void BCast(const CPUDevice& dev,
464              typename TTypes<typename Functor::out_type, NDIMS>::Tensor out,
465              typename TTypes<typename Functor::in_type, NDIMS>::ConstTensor in0,
466              typename Eigen::array<Eigen::DenseIndex, NDIMS> bcast0,
467              typename TTypes<typename Functor::in_type, NDIMS>::ConstTensor in1,
468              typename Eigen::array<Eigen::DenseIndex, NDIMS> bcast1,
469              bool* error) {
470     typedef typename Functor::in_type T;
```

```
471     typename Functor::func func;
472     if (Functor::use_bcast_optimization && use_bcast_optimization<T>::value) {
473       // Optimize for speed by using Eigen::type2index and avoid
474       // .broadcast() when we know it's a no-op.
475       //
476       // Here, we need to handle 6 cases depending on how many "1"
477       // exist in in0 and in1's shapes (4 numbers in total). It's not
478       // possible that two shapes have more than 2 1s because those
479       // are simplified to NDIMS==1 case.
480       //
481       // Because this optimization increases the binary size for each
482       // Functor (+, -, *, /, <, <=, etc.), type and ndim combination.
483       // we only apply such optimization for selected ops/types/ndims.
484       //
485       // Because NDIMS, Functor::use_broadcast_optimization and
486       // use_broadcast_optimization<T> are compile-time constant, gcc
487       // does a decent job avoiding generating code when conditions
488       // are not met.
489       const int a = in0.dimension(0);  // in0 is shape [a, b]
490       const int b = in0.dimension(1);
491       const int c = in1.dimension(0);  // in1 is shape [c, d]
492       const int d = in1.dimension(1);
493       if ((a == 1) && (d == 1)) {
494         auto lhs = in0.reshape(OneByM(b)).broadcast(NByOne(c));
495         auto rhs = in1.reshape(NByOne(c)).broadcast(OneByM(b));
496         Assign(dev, out, lhs.binaryExpr(rhs, func));
497         return;
498       }
499       if ((b == 1) && (c == 1)) {
500         auto lhs = in0.reshape(NByOne(a)).broadcast(OneByM(d));
501         auto rhs = in1.reshape(OneByM(d)).broadcast(NByOne(a));
502         Assign(dev, out, lhs.binaryExpr(rhs, func));
503         return;
504       }
505       if (a == 1) {
506         auto lhs = in0.reshape(OneByM(b)).broadcast(NByOne(c));
507         auto rhs = in1;
508         Assign(dev, out, lhs.binaryExpr(rhs, func));
509         return;
510       }
511       if (b == 1) {
512         auto lhs = in0.reshape(NByOne(a)).broadcast(OneByM(d));
513         auto rhs = in1;
514         Assign(dev, out, lhs.binaryExpr(rhs, func));
515         return;
516       }
517       if (c == 1) {
518         auto lhs = in0;
519         auto rhs = in1.reshape(OneByM(d)).broadcast(NByOne(a));
```

```
520         Assign(dev, out, lhs.binaryExpr(rhs, func));
521         return;
522       }
523       if (d == 1) {
524         auto lhs = in0;
525         auto rhs = in1.reshape(NByOne(c)).broadcast(OneByM(b));
526         Assign(dev, out, lhs.binaryExpr(rhs, func));
527         return;
528       }
529
530       const bool bcast0_all_one = AllOne<NDIMS>(bcast0);
531       const bool bcast1_all_one = AllOne<NDIMS>(bcast1);
532       if (bcast0_all_one && !bcast1_all_one) {
533         auto lhs = in0;  // No need to do broadcast for in0
534         auto rhs = in1.broadcast(bcast1);
535         Assign(dev, out, lhs.binaryExpr(rhs, func));
536         return;
537       }
538
539       if (!bcast0_all_one && bcast1_all_one) {
540         auto lhs = in0.broadcast(bcast0);
541         auto rhs = in1;  // No need to do broadcast for in1
542         Assign(dev, out, lhs.binaryExpr(rhs, func));
543         return;
544       }
545     }
546
547     // Fallback path. Always works and probably slower.
548     auto lhs = in0.broadcast(bcast0);
549     auto rhs = in1.broadcast(bcast1);
550     Assign(dev, out, lhs.binaryExpr(rhs, func));
551   }
552 };
553
554 // Version of BinaryFunctor with error handling.
555 template <typename Functor, int NDIMS>
556 struct BinaryFunctor<CPUDevice, Functor, NDIMS, true> {
557   void operator()(const CPUDevice& d, typename Functor::tout_type out,
558                   typename Functor::tin_type in0,
559                   typename Functor::tin_type in1, bool* error) {
560     Assign(d, out, in0.binaryExpr(in1, typename Functor::func(error)));
561   }
562
563   void Left(const CPUDevice& d, typename Functor::tout_type out,
564             typename Functor::tscalar_type scalar,
565             typename Functor::tin_type in, bool* error) {
566     typedef typename Functor::out_type Tout;
567     typedef typename Functor::in_type Tin;
568     typedef typename Functor::func Binary;
```

```
569       typedef
570           typename Eigen::internal::scalar_left<Tout, Tin, Binary,
571                                                 /*is_scalar_in_host_memory=*/true>
572               Unary;
573       Assign(d, out, in.unaryExpr(Unary(scalar.data(), error)));
574     }
575
576     void Right(const CPUDevice& d, typename Functor::tout_type out,
577                typename Functor::tin_type in,
578                typename Functor::tscalar_type scalar, bool* error) {
579       typedef typename Functor::out_type Tout;
580       typedef typename Functor::in_type Tin;
581       typedef typename Functor::func Binary;
582       typedef typename Eigen::internal::scalar_right<
583           Tout, Tin, Binary, /*is_scalar_in_host_memory=*/true>
584           Unary;
585       Assign(d, out, in.unaryExpr(Unary(scalar.data(), error)));
586     }
587
588     void BCast(const CPUDevice& dev,
589                typename TTypes<typename Functor::out_type, NDIMS>::Tensor out,
590                typename TTypes<typename Functor::in_type, NDIMS>::ConstTensor in0,
591                typename Eigen::array<Eigen::DenseIndex, NDIMS> bcast0,
592                typename TTypes<typename Functor::in_type, NDIMS>::ConstTensor in1,
593                typename Eigen::array<Eigen::DenseIndex, NDIMS> bcast1,
594                bool* error) {
595       typename Functor::func func(error);
596       auto lhs = in0.broadcast(bcast0);
597       auto rhs = in1.broadcast(bcast1);
598       Assign(dev, out, lhs.binaryExpr(rhs, func));
599     }
600   };
601
602   // Partial specialization of UnaryFunctor<Device=CPUDevice, Functor>.
603   template <typename Functor>
604   struct UnaryFunctor<CPUDevice, Functor> {
605     void operator()(const CPUDevice& d, typename Functor::tout_type out,
606                     typename Functor::tin_type in) {
607       Assign(d, out, in.unaryExpr(typename Functor::func()));
608     }
609   };
610
611   // Partial specialization of ApproximateEqual<Device=CPUDevice, T>.
612   template <typename T>
613   struct ApproximateEqual<CPUDevice, T> {
614     void operator()(const CPUDevice& d, typename TTypes<T>::ConstFlat x,
615                     typename TTypes<T>::ConstFlat y, T tolerance,
616                     typename TTypes<bool>::Flat z) {
617       auto diff = x - y;
```

```
    z.device(d) = diff.abs() <= tolerance;
  }
};

}  // end namespace functor

#define REGISTER(OP, D, N, F, T)                                               \
  REGISTER_KERNEL_BUILDER(Name(N).Device(DEVICE_##D).TypeConstraint<T>("T"), \
                          OP<D##Device, F<T>>);

#define REGISTER_VARIANT(OP, D, N, ENUM)                       \
  REGISTER_KERNEL_BUILDER(                                     \
      Name(N).Device(DEVICE_##D).TypeConstraint<Variant>("T"), \
      OP<D##Device, ENUM>);

// Macros to register kernels for multiple types (T0, T1, etc.)  on
// device type "D" (CPU or GPU) for operation "N" (e.g., sqrt) using
// the functor "F" (e.g., functor::sqrt).

#if defined(__ANDROID_TYPES_SLIM__)
// Note that __ANDROID_TYPES_SLIM__ is also checked in the cwise_ops*.cc files.
// Normally Android TensorFlow is built with a reduced number of types (float).
// Override on the command-line using "--copt=-D__ANDROID_TYPES_FULL__"
// to generate a library with full type support with a consequent increase in
// code size.
#define REGISTER2(OP, D, N, F, T0, T1) REGISTER(OP, D, N, F, T0)
#define REGISTER3(OP, D, N, F, T0, T1, T2) REGISTER(OP, D, N, F, T0)
#define REGISTER4(OP, D, N, F, T0, T1, T2, T3) REGISTER(OP, D, N, F, T0)
#define REGISTER5(OP, D, N, F, T0, T1, T2, T3, T4) REGISTER(OP, D, N, F, T0)
#define REGISTER6(OP, D, N, F, T0, T1, T2, T3, T4, T5) REGISTER(OP, D, N, F, T0)
#define REGISTER7(OP, D, N, F, T0, T1, T2, T3, T4, T5, T6) \
  REGISTER(OP, D, N, F, T0)
#define REGISTER8(OP, D, N, F, T0, T1, T2, T3, T4, T5, T6, T7) \
  REGISTER(OP, D, N, F, T0)
#define REGISTER9(OP, D, N, F, T0, T1, T2, T3, T4, T5, T6, T7, T8) \
  REGISTER(OP, D, N, F, T0)
#else  // !defined(__ANDROID_TYPES_SLIM__)
#define REGISTER2(OP, D, N, F, T0, T1) \
  REGISTER(OP, D, N, F, T0)            \
  REGISTER(OP, D, N, F, T1)
#define REGISTER3(OP, D, N, F, T0, T1, T2) \
  REGISTER2(OP, D, N, F, T0, T1)           \
  REGISTER(OP, D, N, F, T2)
#define REGISTER4(OP, D, N, F, T0, T1, T2, T3) \
  REGISTER2(OP, D, N, F, T0, T1)               \
  REGISTER2(OP, D, N, F, T2, T3)
#define REGISTER5(OP, D, N, F, T0, T1, T2, T3, T4) \
  REGISTER3(OP, D, N, F, T0, T1, T2)               \
  REGISTER2(OP, D, N, F, T3, T4)
```

```
#define REGISTER6(OP, D, N, F, T0, T1, T2, T3, T4, T5) \
  REGISTER3(OP, D, N, F, T0, T1, T2)                   \
  REGISTER3(OP, D, N, F, T3, T4, T5)
#define REGISTER7(OP, D, N, F, T0, T1, T2, T3, T4, T5, T6) \
  REGISTER4(OP, D, N, F, T0, T1, T2, T3)                   \
  REGISTER3(OP, D, N, F, T4, T5, T6)
#define REGISTER8(OP, D, N, F, T0, T1, T2, T3, T4, T5, T6, T7) \
  REGISTER4(OP, D, N, F, T0, T1, T2, T3)                       \
  REGISTER4(OP, D, N, F, T4, T5, T6, T7)
#define REGISTER9(OP, D, N, F, T0, T1, T2, T3, T4, T5, T6, T7, T8) \
  REGISTER5(OP, D, N, F, T0, T1, T2, T3, T4)                       \
  REGISTER4(OP, D, N, F, T5, T6, T7, T8)

// Instead of adding REGISTER10, etc., shard the .cc files - see
// cwise_op_equal_to_*.cc for an example.

#endif  // defined(__ANDROID_TYPES_SLIM__)

}  // end namespace tensorflow

#endif  // TENSORFLOW_CORE_KERNELS_CWISE_OPS_COMMON_H_
```