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

3 contributors   

737 lines (668 sloc) | 31.6 KB ...

```
/* Copyright 2015 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
==============================================================================*/

// See docs in ../ops/image_ops.cc

#include <cstdint>
#include <memory>

#define EIGEN_USE_THREADS

#include "absl/strings/escaping.h"
#include "tensorflow/core/framework/bounds_check.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/register_types.h"
#include "tensorflow/core/framework/tensor.h"
#include "tensorflow/core/framework/tensor_shape.h"
#include "tensorflow/core/framework/types.h"
```

```cpp
#include "tensorflow/core/lib/core/status.h"
#include "tensorflow/core/lib/gif/gif_io.h"
#include "tensorflow/core/lib/jpeg/jpeg_mem.h"
#include "tensorflow/core/lib/png/png_io.h"
#include "tensorflow/core/lib/strings/str_util.h"
#include "tensorflow/core/platform/byte_order.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/util/tensor_bundle/byte_swap.h"

namespace tensorflow {
namespace {

// Magic bytes (hex) for each image format.
// https://en.wikipedia.org/wiki/List_of_file_signatures
// WARNING: Changing `static const` to `constexpr` requires first checking that
// it works with supported MSVC version.
// https://docs.microsoft.com/en-us/cpp/cpp/constexpr-cpp?redirectedfrom=MSDN&view=vs-2019
static const char kPngMagicBytes[] = "\x89\x50\x4E\x47\x0D\x0A\x1A\x0A";
static const char kGifMagicBytes[] = "\x47\x49\x46\x38";
static const char kBmpMagicBytes[] = "\x42\x4d";
// The 4th byte of JPEG is '\xe0' or '\xe1', so check just the first three.
static const char kJpegMagicBytes[] = "\xff\xd8\xff";

enum FileFormat {
  kUnknownFormat = 0,
  kPngFormat = 1,
  kJpgFormat = 2,
  kGifFormat = 3,
  kBmpFormat = 4,
};

// Classify the contents of a file based on starting bytes (the magic number).
FileFormat ClassifyFileFormat(StringPiece data) {
  if (absl::StartsWith(data, kJpegMagicBytes)) return kJpgFormat;
  if (absl::StartsWith(data, kPngMagicBytes)) return kPngFormat;
  if (absl::StartsWith(data, kGifMagicBytes)) return kGifFormat;
  if (absl::StartsWith(data, kBmpMagicBytes)) return kBmpFormat;
  return kUnknownFormat;
}

// Decode an image. Supported image formats are JPEG, PNG, GIF and BMP. This is
// a newer version of `DecodeImageOp` for enabling image data parsing to take
// place in kernels only, reducing security vulnerabilities and redundancy.
class DecodeImageV2Op : public OpKernel {
 public:
  explicit DecodeImageV2Op(OpKernelConstruction* context) : OpKernel(context) {
    // Keep track of op string information because:
    // [1] Currently by the API, PNG, JPEG and GIF can decode each other and
    //     depending on the op type, we need to return either 3-D or 4-D shapes.
```

```
    // [2] Different ops have different attributes. e.g. `DecodeImage` op has
    //     `expand_animations` attribute that other ops don't.
    //     `DecodeAndDropJpeg` also has additional attributes.
    op_type_ = type_string();

    // Validate op type.
    OP_REQUIRES(context,
                op_type_ == "DecodeJpeg" || op_type_ == "DecodeAndCropJpeg" ||
                    op_type_ == "DecodePng" || op_type_ == "DecodeGif" ||
                    op_type_ == "DecodeBmp" || op_type_ == "DecodeImage",
                errors::InvalidArgument("Bad op type ", op_type_));

    // Get attributes from `DecodeJpeg` and `DecodeAndCropJpeg` op
    // invocations. For `DecodeImage` op, set JPEG decoding setting to TF
    // default.
    if (op_type_ == "DecodeJpeg" || op_type_ == "DecodeAndCropJpeg") {
      OP_REQUIRES_OK(context, context->GetAttr("ratio", &flags_.ratio));
      OP_REQUIRES(context,
                  flags_.ratio == 1 || flags_.ratio == 2 || flags_.ratio == 4 ||
                      flags_.ratio == 8,
                  errors::InvalidArgument("ratio must be 1, 2, 4, or 8, got ",
                                          flags_.ratio));
      OP_REQUIRES_OK(context, context->GetAttr("fancy_upscaling",
                                               &flags_.fancy_upscaling));
      OP_REQUIRES_OK(context,
                     context->GetAttr("try_recover_truncated",
                                      &flags_.try_recover_truncated_jpeg));
      OP_REQUIRES_OK(context,
                     context->GetAttr("acceptable_fraction",
                                      &flags_.min_acceptable_fraction));
      string dct_method;
      OP_REQUIRES_OK(context, context->GetAttr("dct_method", &dct_method));
      OP_REQUIRES(
          context,
          (dct_method.empty() || dct_method == "INTEGER_FAST" ||
           dct_method == "INTEGER_ACCURATE"),
          errors::InvalidArgument("dct_method must be one of "
                                  "{'', 'INTEGER_FAST', 'INTEGER_ACCURATE'}"));
      // The TensorFlow-chosen default for JPEG decoding is IFAST, sacrificing
      // image quality for speed.
      if (dct_method.empty() || dct_method == "INTEGER_FAST") {
        flags_.dct_method = JDCT_IFAST;
      } else if (dct_method == "INTEGER_ACCURATE") {
        flags_.dct_method = JDCT_ISLOW;
      }
    } else {
      flags_ = jpeg::UncompressFlags();
      flags_.dct_method = JDCT_IFAST;
    }
```

```
128
129     // Get `dtype` attribute from `DecodePng` or `DecodeImage` op invocations.
130     if (op_type_ == "DecodePng" || op_type_ == "DecodeImage") {
131       OP_REQUIRES_OK(context, context->GetAttr("dtype", &data_type_));
132       if (op_type_ == "DecodePng") {
133         OP_REQUIRES(
134             context,
135             data_type_ == DataType::DT_UINT8 ||
136                 data_type_ == DataType::DT_UINT16,
137             errors::InvalidArgument(
138                 "`dtype` for `DecodePng` must be unit8, unit16 but got: ",
139                 data_type_));
140       } else {
141         OP_REQUIRES(context,
142                     data_type_ == DataType::DT_UINT8 ||
143                         data_type_ == DataType::DT_UINT16 ||
144                         data_type_ == DataType::DT_FLOAT,
145                     errors::InvalidArgument("`dtype` for `DecodeImage` must be "
146                                             "unit8, unit16, float but got: ",
147                                             data_type_));
148         OP_REQUIRES_OK(context, context->GetAttr("expand_animations",
149                                                  &expand_animations_));
150       }
151     }
152
153     // Get `channels` attribute for all ops except `DecodeGif` op.
154     // `DecodeGif` doesn't have `channels` attribute but it supports 3
155     // channels by default.
156     if (op_type_ != "DecodeGif") {
157       OP_REQUIRES_OK(context, context->GetAttr("channels", &channels_));
158       OP_REQUIRES(
159           context,
160           channels_ == 0 || channels_ == 1 || channels_ == 3 || channels_ == 4,
161           errors::InvalidArgument("`channels` must be 0, 1, 3 or 4 but got ",
162                                   channels_));
163     } else {
164       channels_ = 3;
165     }
166   }
167
168   // Helper for decoding BMP.
169   inline int32 ByteSwapInt32ForBigEndian(int32_t x) {
170     if (!port::kLittleEndian) {
171       return BYTE_SWAP_32(x);
172     } else {
173       return x;
174     }
175   }
176
```

```
  // Helper for decoding BMP.
  inline int16 ByteSwapInt16ForBigEndian(int16_t x) {
    if (!port::kLittleEndian) {
      return BYTE_SWAP_16(x);
    } else {
      return x;
    }
  }

  void Compute(OpKernelContext* context) override {
    const Tensor& contents = context->input(0);
    OP_REQUIRES(
        context, TensorShapeUtils::IsScalar(contents.shape()),
        errors::InvalidArgument("`contents` must be scalar but got shape",
                                contents.shape().DebugString()));
    const StringPiece input = contents.scalar<tstring>()();
    OP_REQUIRES(context, !input.empty(),
                errors::InvalidArgument("Input is empty."));
    OP_REQUIRES(context, input.size() <= std::numeric_limits<int>::max(),
                errors::InvalidArgument(
                    "Input contents are too large for int: ", input.size()));

    // Parse magic bytes to determine file format.
    switch (ClassifyFileFormat(input)) {
      case kJpgFormat:
        DecodeJpegV2(context, input);
        break;
      case kPngFormat:
        DecodePngV2(context, input);
        break;
      case kGifFormat:
        DecodeGifV2(context, input);
        break;
      case kBmpFormat:
        DecodeBmpV2(context, input);
        break;
      case kUnknownFormat:
        OP_REQUIRES(context, false,
                    errors::InvalidArgument("Unknown image file format. One of "
                                            "JPEG, PNG, GIF, BMP required."));
        break;
    }
  }

  void DecodeJpegV2(OpKernelContext* context, StringPiece input) {
    OP_REQUIRES(context, channels_ == 0 || channels_ == 1 || channels_ == 3,
                errors::InvalidArgument("JPEG does not support 4 channels"));

    // Use local copy of flags to avoid race condition as the class member is
```

```
    // shared among different invocations.
    jpeg::UncompressFlags flags = flags_;
    flags.components = channels_;

    if (op_type_ == "DecodeAndCropJpeg") {
      flags.crop = true;
      // Update flags to include crop window.
      const Tensor& crop_window = context->input(1);
      OP_REQUIRES(context, crop_window.dims() == 1,
                  errors::InvalidArgument("crop_window must be 1-D, got shape ",
                                          crop_window.shape().DebugString()));
      OP_REQUIRES(context, crop_window.dim_size(0) == 4,
                  errors::InvalidArgument("crop_size must have four elements ",
                                          crop_window.shape().DebugString()));
      auto crop_window_vec = crop_window.vec<int32>();
      flags.crop_y = crop_window_vec(0);
      flags.crop_x = crop_window_vec(1);
      flags.crop_height = crop_window_vec(2);
      flags.crop_width = crop_window_vec(3);
    } else if (op_type_ == "DecodeBmp") {
      // TODO(b/171060723): Only DecodeBmp as op_type_ is not acceptable here
      // because currently `decode_(jpeg|png|gif)` ops can decode any one of
      // jpeg, png or gif but not bmp. Similarly, `decode_bmp` cannot decode
      // anything but bmp formats. This behavior needs to be revisited. For more
      // details, please refer to the bug.
      OP_REQUIRES(context, false,
                  errors::InvalidArgument(
                      "Trying to decode JPEG format using DecodeBmp op. Use "
                      "`decode_jpeg` or `decode_image` instead."));
    }

    // Output tensor and the image buffer size.
    Tensor* output = nullptr;
    int buffer_size = 0;

    // Decode JPEG. Directly allocate to the output buffer if data type is
    // uint8 (to save extra copying). Otherwise, allocate a new uint8 buffer
    // with buffer size. `jpeg::Uncompress` supports unit8 only.
    uint8* buffer = jpeg::Uncompress(
        input.data(), input.size(), flags, nullptr /* nwarn */,
        [&](int width, int height, int channels) -> uint8* {
          buffer_size = height * width * channels;
          Status status;
          // By the existing API, we support decoding JPEG with `DecodeGif`
          // op. We need to make sure to return 4-D shapes when using
          // `DecodeGif`.
          if (op_type_ == "DecodeGif") {
            status = context->allocate_output(
                0, TensorShape({1, height, width, channels}), &output);
```

```
275           } else {
276             status = context->allocate_output(
277                 0, TensorShape({height, width, channels}), &output);
278           }
279           if (!status.ok()) {
280             VLOG(1) << status;
281             context->SetStatus(status);
282             return nullptr;
283           }
284
285           if (data_type_ == DataType::DT_UINT8) {
286             return output->flat<uint8>().data();
287           } else {
288             return new uint8[buffer_size];
289           }
290         });
291
292     OP_REQUIRES(
293         context, buffer,
294         errors::InvalidArgument(
295             "jpeg::Uncompress failed. Invalid JPEG data or crop window."));
296
297     // For when desired data type if unit8, the output buffer is already
298     // allocated during the `jpeg::Uncompress` call above; return.
299     if (data_type_ == DataType::DT_UINT8) {
300       return;
301     }
302     // Make sure we don't forget to deallocate `buffer`.
303     std::unique_ptr<uint8[]> buffer_unique_ptr(buffer);
304
305     // Convert uint8 image data to desired data type.
306     // Use eigen threadpooling to speed up the copy operation.
307     const auto& device = context->eigen_device<Eigen::ThreadPoolDevice>();
308     TTypes<uint8>::UnalignedConstFlat buffer_view(buffer, buffer_size);
309     if (data_type_ == DataType::DT_UINT16) {
310       uint16 scale = floor((std::numeric_limits<uint16>::max() + 1) /
311                            (std::numeric_limits<uint8>::max() + 1));
312       // Fill output tensor with desired dtype.
313       output->flat<uint16>().device(device) =
314           buffer_view.cast<uint16>() * scale;
315     } else if (data_type_ == DataType::DT_FLOAT) {
316       float scale = 1. / std::numeric_limits<uint8>::max();
317       // Fill output tensor with desired dtype.
318       output->flat<float>().device(device) = buffer_view.cast<float>() * scale;
319     }
320   }
321
322   void DecodePngV2(OpKernelContext* context, StringPiece input) {
323     int channel_bits = (data_type_ == DataType::DT_UINT8) ? 8 : 16;
```

```
    png::DecodeContext decode;
    OP_REQUIRES(
        context, png::CommonInitDecode(input, channels_, channel_bits, &decode),
        errors::InvalidArgument("Invalid PNG. Failed to initialize decoder."));

    // Verify that width and height are not too large:
    // - verify width and height don't overflow int.
    // - width can later be multiplied by channels_ and sizeof(uint16), so
    //   verify single dimension is not too large.
    // - verify when width and height are multiplied together, there are a few
    //   bits to spare as well.
    const int width = static_cast<int>(decode.width);
    const int height = static_cast<int>(decode.height);
    const int64_t total_size =
        static_cast<int64_t>(width) * static_cast<int64_t>(height);
    if (width != static_cast<int64_t>(decode.width) || width <= 0 ||
        width >= (1LL << 27) || height != static_cast<int64_t>(decode.height) ||
        height <= 0 || height >= (1LL << 27) || total_size >= (1LL << 29)) {
      png::CommonFreeDecode(&decode);
      OP_REQUIRES(context, false,
                  errors::InvalidArgument("PNG size too large for int: ",
                                          decode.width, " by ", decode.height));
    }

    Tensor* output = nullptr;
    Status status;
    // By the existing API, we support decoding PNG with `DecodeGif` op.
    // We need to make sure to return 4-D shapes when using `DecodeGif`.
    if (op_type_ == "DecodeGif") {
      status = context->allocate_output(
          0, TensorShape({1, height, width, decode.channels}), &output);
    } else {
      status = context->allocate_output(
          0, TensorShape({height, width, decode.channels}), &output);
    }

    if (op_type_ == "DecodeBmp") {
      // TODO(b/171060723): Only DecodeBmp as op_type_ is not acceptable here
      // because currently `decode_(jpeg|png|gif)` ops can decode any one of
      // jpeg, png or gif but not bmp. Similarly, `decode_bmp` cannot decode
      // anything but bmp formats. This behavior needs to be revisited. For more
      // details, please refer to the bug.
      OP_REQUIRES(context, false,
                  errors::InvalidArgument(
                      "Trying to decode PNG format using DecodeBmp op. Use "
                      "`decode_png` or `decode_image` instead."));
    } else if (op_type_ == "DecodeAndCropJpeg") {
      OP_REQUIRES(context, false,
                  errors::InvalidArgument(
```

```
                    "DecodeAndCropJpeg operation can run on JPEG only, but "
                    "detected PNG."));
  }

  if (!status.ok()) png::CommonFreeDecode(&decode);
  OP_REQUIRES_OK(context, status);

  if (data_type_ == DataType::DT_UINT8) {
    OP_REQUIRES(
        context,
        png::CommonFinishDecode(
            reinterpret_cast<png_bytep>(output->flat<uint8>().data()),
            decode.channels * width * sizeof(uint8), &decode),
        errors::InvalidArgument("Invalid PNG data, size ", input.size()));
  } else if (data_type_ == DataType::DT_UINT16) {
    OP_REQUIRES(
        context,
        png::CommonFinishDecode(
            reinterpret_cast<png_bytep>(output->flat<uint16>().data()),
            decode.channels * width * sizeof(uint16), &decode),
        errors::InvalidArgument("Invalid PNG data, size ", input.size()));
  } else if (data_type_ == DataType::DT_FLOAT) {
    // `png::CommonFinishDecode` does not support `float`. First allocate
    // uint16 buffer for the image and decode in uint16 (lossless). Wrap the
    // buffer in `unique_ptr` so that we don't forget to delete the buffer.
    std::unique_ptr<uint16[]> buffer(
        new uint16[height * width * decode.channels]);
    OP_REQUIRES(
        context,
        png::CommonFinishDecode(reinterpret_cast<png_bytep>(buffer.get()),
                                decode.channels * width * sizeof(uint16),
                                &decode),
        errors::InvalidArgument("Invalid PNG data, size ", input.size()));

    // Convert uint16 image data to desired data type.
    // Use eigen threadpooling to speed up the copy operation.
    const auto& device = context->eigen_device<Eigen::ThreadPoolDevice>();
    TTypes<uint16, 3>::UnalignedConstTensor buf(buffer.get(), height, width,
                                                decode.channels);
    float scale = 1. / std::numeric_limits<uint16>::max();
    // Fill output tensor with desired dtype.
    output->tensor<float, 3>().device(device) = buf.cast<float>() * scale;
  }
}

void DecodeGifV2(OpKernelContext* context, StringPiece input) {
  // GIF has 3 channels.
  OP_REQUIRES(context, channels_ == 0 || channels_ == 3,
              errors::InvalidArgument("channels must be 0 or 3 for GIF, got ",
```

```
                                      channels_));

    if (op_type_ == "DecodeBmp") {
      // TODO(b/171060723): Only DecodeBmp as op_type_ is not acceptable here
      // because currently `decode_(jpeg|png|gif)` ops can decode any one of
      // jpeg, png or gif but not bmp. Similarly, `decode_bmp` cannot decode
      // anything but bmp formats. This behavior needs to be revisited. For more
      // details, please refer to the bug.
      OP_REQUIRES(context, false,
                  errors::InvalidArgument(
                      "Trying to decode GIF format using DecodeBmp op. Use "
                      "`decode_gif` or `decode_image` instead."));
    } else if (op_type_ == "DecodeAndCropJpeg") {
      OP_REQUIRES(context, false,
                  errors::InvalidArgument(
                      "DecodeAndCropJpeg operation can run on JPEG only, but "
                      "detected GIF."));
    }

    // Decode GIF, allocating tensor if dtype is uint8, otherwise defer tensor
    // allocation til after dtype conversion is done. `gif`::Decode` supports
    // uint8 only.
    Tensor* output = nullptr;
    int buffer_size = 0;
    string error_string;
    uint8* buffer = gif::Decode(
        input.data(), input.size(),
        [&](int num_frames, int width, int height, int channels) -> uint8* {
          buffer_size = num_frames * height * width * channels;

          Status status;
          // By the existing API, we support decoding GIF with `decode_jpeg` or
          // with `decode_png` if the GIF is a single-frame GIF (non-animated).
          // We need to make sure to return 3-D shapes when using in this case.
          if (op_type_ == "DecodePng" || op_type_ == "DecodeJpeg") {
            if (num_frames == 1) {
              status = context->allocate_output(
                  0, TensorShape({height, width, channels}), &output);
            } else {
              status = errors::InvalidArgument(
                  "Got ", num_frames, " frames, but animated gifs ",
                  "can only be decoded by tf.io.decode_gif or ",
                  "tf.io.decode_image");
            }
          } else if (op_type_ == "DecodeGif" ||
                     (op_type_ == "DecodeImage" && expand_animations_)) {
            status = context->allocate_output(
                0, TensorShape({num_frames, height, width, channels}), &output);
          } else if (op_type_ == "DecodeImage" && !expand_animations_) {
```

```
471             status = context->allocate_output(
472                 0, TensorShape({height, width, channels}), &output);
473           } else {
474             status = errors::InvalidArgument("Bad op type ", op_type_);
475           }
476           if (!status.ok()) {
477             VLOG(1) << status;
478             context->SetStatus(status);
479             return nullptr;
480           }
481
482           if (data_type_ == DataType::DT_UINT8) {
483             return output->flat<uint8>().data();
484           } else {
485             return new uint8[buffer_size];
486           }
487         },
488         &error_string, expand_animations_);
489
490     OP_REQUIRES(context, buffer,
491                 errors::InvalidArgument("Invalid GIF data (size ", input.size(),
492                                         "), ", error_string));
493
494     // For when desired data type is unit8, the output buffer is already
495     // allocated during the `gif::Decode` call above; return.
496     if (data_type_ == DataType::DT_UINT8) {
497       return;
498     }
499     // Make sure we don't forget to deallocate `buffer`.
500     std::unique_ptr<uint8[]> buffer_unique_ptr(buffer);
501
502     // Convert the raw uint8 buffer to desired dtype.
503     // Use eigen threadpooling to speed up the copy operation.
504     TTypes<uint8>::UnalignedConstFlat buffer_view(buffer, buffer_size);
505     const auto& device = context->eigen_device<Eigen::ThreadPoolDevice>();
506     if (data_type_ == DataType::DT_UINT16) {
507       uint16 scale = floor((std::numeric_limits<uint16>::max() + 1) /
508                            (std::numeric_limits<uint8>::max() + 1));
509       // Fill output tensor with desired dtype.
510       output->flat<uint16>().device(device) =
511           buffer_view.cast<uint16>() * scale;
512     } else if (data_type_ == DataType::DT_FLOAT) {
513       float scale = 1. / std::numeric_limits<uint8>::max();
514       // Fill output tensor with desired dtype.
515       output->flat<float>().device(device) = buffer_view.cast<float>() * scale;
516     }
517   }
518
519   void DecodeBmpV2(OpKernelContext* context, StringPiece input) {
```

```
520     OP_REQUIRES(
521         context, channels_ != 1,
522         errors::InvalidArgument(
523             "`channels` must be 0, 3 or 4 for BMP, but got ", channels_));
524
525     if (op_type_ != "DecodeBmp" && op_type_ != "DecodeImage") {
526       if (op_type_ == "DecodeAndCropJpeg") {
527         OP_REQUIRES(context, false,
528                     errors::InvalidArgument(
529                         "DecodeAndCropJpeg operation can run on JPEG only, but "
530                         "detected BMP."));
531       } else {
532         OP_REQUIRES(context, false,
533                     errors::InvalidArgument(
534                         "Trying to decode BMP format using a wrong op. Use "
535                         "`decode_bmp` or `decode_image` instead. Op used: ",
536                         op_type_));
537       }
538     }
539
540     OP_REQUIRES(context, (32 <= input.size()),
541                 errors::InvalidArgument("Incomplete bmp content, requires at "
542                                         "least 32 bytes to find the header "
543                                         "size, width, height, and bpp, got ",
544                                         input.size(), " bytes"));
545
546     const uint8* img_bytes = reinterpret_cast<const uint8*>(input.data());
547     int32_t header_size_ = internal::SubtleMustCopy(
548         *(reinterpret_cast<const int32*>(img_bytes + 10)));
549     const int32_t header_size = ByteSwapInt32ForBigEndian(header_size_);
550     int32_t width_ = internal::SubtleMustCopy(
551         *(reinterpret_cast<const int32*>(img_bytes + 18)));
552     const int32_t width = ByteSwapInt32ForBigEndian(width_);
553     int32_t height_ = internal::SubtleMustCopy(
554         *(reinterpret_cast<const int32*>(img_bytes + 22)));
555     const int32_t height = ByteSwapInt32ForBigEndian(height_);
556     int16_t bpp_ = internal::SubtleMustCopy(
557         *(reinterpret_cast<const int16*>(img_bytes + 28)));
558     const int16_t bpp = ByteSwapInt16ForBigEndian(bpp_);
559
560     // `channels_` is desired number of channels. `img_channels` is number of
561     // channels inherent in the image.
562     int img_channels = bpp / 8;
563     OP_REQUIRES(
564         context, (img_channels == 1 || img_channels == 3 || img_channels == 4),
565         errors::InvalidArgument(
566             "Number of channels inherent in the image must be 1, 3 or 4, was ",
567             img_channels));
568     const int requested_channels = channels_ ? channels_ : img_channels;
```

```
569
570     OP_REQUIRES(context, width > 0,
571                 errors::InvalidArgument("Width must be positive"));
572     OP_REQUIRES(context, height != 0,
573                 errors::InvalidArgument("Height must be nonzero"));
574     OP_REQUIRES(context, header_size >= 0,
575                 errors::InvalidArgument("header size must be nonnegative"));
576
577     // The real requirement is < 2^31 minus some headers and channel data,
578     // so rounding down to something that's still ridiculously big.
579     OP_REQUIRES(
580         context,
581         (static_cast<int64_t>(width) * std::abs(static_cast<int64_t>(height))) <
582             static_cast<int64_t>(std::numeric_limits<int32_t>::max() / 8),
583         errors::InvalidArgument(
584             "Total possible pixel bytes must be less than 2^30"));
585
586     const int32_t abs_height = abs(height);
587
588     // there may be padding bytes when the width is not a multiple of 4 bytes
589     const int row_size = (img_channels * width + 3) / 4 * 4;
590
591     // Make sure the size of input data matches up with the total size of
592     // headers plus height * row_size.
593     int size_diff = input.size() - header_size - (row_size * abs_height);
594     OP_REQUIRES(
595         context, size_diff == 0,
596         errors::InvalidArgument(
597             "Input size should match (header_size + row_size * abs_height) but "
598             "they differ by ",
599             size_diff));
600
601     const int64_t last_pixel_offset = static_cast<int64_t>(header_size) +
602                                       (abs_height - 1) * row_size +
603                                       (width - 1) * img_channels;
604
605     // [expected file size] = [last pixel offset] + [last pixel size=channels]
606     const int64_t expected_file_size = last_pixel_offset + img_channels;
607
608     OP_REQUIRES(
609         context, (expected_file_size <= input.size()),
610         errors::InvalidArgument("Incomplete bmp content, requires at least ",
611                                 expected_file_size, " bytes, got ",
612                                 input.size(), " bytes"));
613
614     // if height is negative, data layout is top down
615     // otherwise, it's bottom up.
616     bool top_down = (height < 0);
617
```

```
618      // Decode image, allocating tensor once the image size is known.
619      Tensor* output = nullptr;
620      OP_REQUIRES_OK(
621          context,
622          context->allocate_output(
623              0, TensorShape({abs_height, width, requested_channels}), &output));
624
625      const uint8* bmp_pixels = &img_bytes[header_size];
626
627      if (data_type_ == DataType::DT_UINT8) {
628        DecodeBMP(bmp_pixels, row_size, output->flat<uint8>().data(), width,
629                  abs_height, requested_channels, img_channels, top_down);
630      } else {
631        std::unique_ptr<uint8[]> buffer(
632            new uint8[height * width * requested_channels]);
633        DecodeBMP(bmp_pixels, row_size, buffer.get(), width, abs_height,
634                  requested_channels, img_channels, top_down);
635        TTypes<uint8, 3>::UnalignedConstTensor buf(buffer.get(), height, width,
636                                                   requested_channels);
637        // Convert the raw uint8 buffer to desired dtype.
638        // Use eigen threadpooling to speed up the copy operation.
639        const auto& device = context->eigen_device<Eigen::ThreadPoolDevice>();
640        if (data_type_ == DataType::DT_UINT16) {
641          uint16 scale = floor((std::numeric_limits<uint16>::max() + 1) /
642                               (std::numeric_limits<uint8>::max() + 1));
643          // Fill output tensor with desired dtype.
644          output->tensor<uint16, 3>().device(device) = buf.cast<uint16>() * scale;
645        } else if (data_type_ == DataType::DT_FLOAT) {
646          float scale = 1. / std::numeric_limits<uint8>::max();
647          // Fill output tensor with desired dtype.
648          output->tensor<float, 3>().device(device) = buf.cast<float>() * scale;
649        }
650      }
651    }
652
653   private:
654    void DecodeBMP(const uint8* input, const int row_size, uint8* const output,
655                   const int width, const int height, const int output_channels,
656                   const int input_channels, bool top_down);
657
658    int channels_ = 0;
659    DataType data_type_ = DataType::DT_UINT8;
660    bool expand_animations_ = true;
661    jpeg::UncompressFlags flags_;
662    string op_type_;
663  };
664
665  REGISTER_KERNEL_BUILDER(Name("DecodeJpeg").Device(DEVICE_CPU), DecodeImageV2Op);
666  REGISTER_KERNEL_BUILDER(Name("DecodePng").Device(DEVICE_CPU), DecodeImageV2Op);
```

```
667 REGISTER_KERNEL_BUILDER(Name("DecodeGif").Device(DEVICE_CPU), DecodeImageV2Op);
668 REGISTER_KERNEL_BUILDER(Name("DecodeAndCropJpeg").Device(DEVICE_CPU),
669                         DecodeImageV2Op);
670 REGISTER_KERNEL_BUILDER(Name("DecodeImage").Device(DEVICE_CPU),
671                         DecodeImageV2Op);
672 REGISTER_KERNEL_BUILDER(Name("DecodeBmp").Device(DEVICE_CPU), DecodeImageV2Op);
673
674 void DecodeImageV2Op::DecodeBMP(const uint8* input, const int row_size,
675                                 uint8* const output, const int width,
676                                 const int height, const int output_channels,
677                                 const int input_channels, bool top_down) {
678   for (int i = 0; i < height; i++) {
679     int src_pos;
680     int dst_pos;
681
682     for (int j = 0; j < width; j++) {
683       if (!top_down) {
684         src_pos = ((height - 1 - i) * row_size) + j * input_channels;
685       } else {
686         src_pos = i * row_size + j * input_channels;
687       }
688
689       dst_pos = (i * width + j) * output_channels;
690
691       switch (input_channels) {
692         case 1:
693           output[dst_pos] = input[src_pos];
694           // Set 2nd and 3rd channels if user requested for 3 or 4 channels.
695           // Repeat 1st channel's value.
696           if (output_channels == 3 || output_channels == 4) {
697             output[dst_pos + 1] = input[src_pos];
698             output[dst_pos + 2] = input[src_pos];
699           }
700           // Set 4th channel (alpha) to maximum value if user requested for
701           // 4 channels.
702           if (output_channels == 4) {
703             output[dst_pos + 3] = UINT8_MAX;
704           }
705           break;
706         case 3:
707           // BGR -> RGB
708           output[dst_pos] = input[src_pos + 2];
709           output[dst_pos + 1] = input[src_pos + 1];
710           output[dst_pos + 2] = input[src_pos];
711           // Set 4th channel (alpha) to maximum value if the user requested for
712           // 4 channels and the input image has 3 channels only.
713           if (output_channels == 4) {
714             output[dst_pos + 3] = UINT8_MAX;
715           }
```

```
716           break;
717         case 4:
718           // BGRA -> RGBA
719           output[dst_pos] = input[src_pos + 2];
720           output[dst_pos + 1] = input[src_pos + 1];
721           output[dst_pos + 2] = input[src_pos];
722           // Set 4th channel only if the user requested for 4 channels. If not,
723           // then user requested 3 channels; skip this step.
724           if (output_channels == 4) {
725             output[dst_pos + 3] = input[src_pos + 3];
726           }
727           break;
728         default:
729           LOG(FATAL) << "Unexpected number of channels: " << input_channels;
730           break;
731       }
732     }
733   }
734 }
735
736 }  // namespace
737 }  // namespace tensorflow
```