

a1320ec1ea ▾

...

tensorflow / tensorflow / core / kernels / image / decode_image_op.cc



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

History

3 contributors



737 lines (668 sloc) | 31.6 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 // See docs in ../ops/image_ops.cc
17
18 #include <stdint>
19 #include <memory>
20
21 #define EIGEN_USE_THREADS
22
23 #include "absl/strings/escaping.h"
24 #include "tensorflow/core/framework/bounds_check.h"
25 #include "tensorflow/core/framework/op_kernel.h"
26 #include "tensorflow/core/framework/register_types.h"
27 #include "tensorflow/core/framework/tensor.h"
28 #include "tensorflow/core/framework/tensor_shape.h"
29 #include "tensorflow/core/framework/types.h"

```

```

30 #include "tensorflow/core/lib/core/status.h"
31 #include "tensorflow/core/lib/gif/gif_io.h"
32 #include "tensorflow/core/lib/jpeg/jpeg_mem.h"
33 #include "tensorflow/core/lib/png/png_io.h"
34 #include "tensorflow/core/lib/strings/str_util.h"
35 #include "tensorflow/core/platform/byte_order.h"
36 #include "tensorflow/core/platform/logging.h"
37 #include "tensorflow/core/util/tensor_bundle/byte_swap.h"
38
39 namespace tensorflow {
40 namespace {
41
42 // Magic bytes (hex) for each image format.
43 // https://en.wikipedia.org/wiki/List_of_file_signatures
44 // WARNING: Changing `static const` to `constexpr` requires first checking that
45 // it works with supported MSVC version.
46 // https://docs.microsoft.com/en-us/cpp/cpp/constexpr-cpp?redirectedfrom=MSDN&view=vs-2019
47 static const char kPngMagicBytes[] = "\x89\x50\x4E\x47\x0D\x0A\x1A\x0A";
48 static const char kGifMagicBytes[] = "\x47\x49\x46\x38";
49 static const char kBmpMagicBytes[] = "\x42\x4d";
50 // The 4th byte of JPEG is '\xe0' or '\xe1', so check just the first three.
51 static const char kJpegMagicBytes[] = "\xff\xd8\xff";
52
53 enum FileFormat {
54     kUnknownFormat = 0,
55     kPngFormat = 1,
56     kJpgFormat = 2,
57     kGifFormat = 3,
58     kBmpFormat = 4,
59 };
60
61 // Classify the contents of a file based on starting bytes (the magic number).
62 FileFormat ClassifyFileFormat(StringPiece data) {
63     if (absl::StartsWith(data, kJpegMagicBytes)) return kJpgFormat;
64     if (absl::StartsWith(data, kPngMagicBytes)) return kPngFormat;
65     if (absl::StartsWith(data, kGifMagicBytes)) return kGifFormat;
66     if (absl::StartsWith(data, kBmpMagicBytes)) return kBmpFormat;
67     return kUnknownFormat;
68 }
69
70 // Decode an image. Supported image formats are JPEG, PNG, GIF and BMP. This is
71 // a newer version of `DecodeImageOp` for enabling image data parsing to take
72 // place in kernels only, reducing security vulnerabilities and redundancy.
73 class DecodeImageV2Op : public OpKernel {
74 public:
75     explicit DecodeImageV2Op(OpKernelConstruction* context) : OpKernel(context) {
76         // Keep track of op string information because:
77         // [1] Currently by the API, PNG, JPEG and GIF can decode each other and
78         //     depending on the op type, we need to return either 3-D or 4-D shapes.

```

```

79 // [2] Different ops have different attributes. e.g. `DecodeImage` op has
80 //     `expand_animations` attribute that other ops don't.
81 //     `DecodeAndDropJpeg` also has additional attributes.
82 op_type_ = type_string();
83
84 // Validate op type.
85 OP_REQUIRES(context,
86             op_type_ == "DecodeJpeg" || op_type_ == "DecodeAndCropJpeg" ||
87             op_type_ == "DecodePng" || op_type_ == "DecodeGif" ||
88             op_type_ == "DecodeBmp" || op_type_ == "DecodeImage",
89             errors::InvalidArgument("Bad op type ", op_type_));
90
91 // Get attributes from `DecodeJpeg` and `DecodeAndCropJpeg` op
92 // invocations. For `DecodeImage` op, set JPEG decoding setting to TF
93 // default.
94 if (op_type_ == "DecodeJpeg" || op_type_ == "DecodeAndCropJpeg") {
95     OP_REQUIRES_OK(context, context->GetAttr("ratio", &flags_.ratio));
96     OP_REQUIRES(context,
97                 flags_.ratio == 1 || flags_.ratio == 2 || flags_.ratio == 4 ||
98                 flags_.ratio == 8,
99                 errors::InvalidArgument("ratio must be 1, 2, 4, or 8, got ",
100                                         flags_.ratio));
101     OP_REQUIRES_OK(context, context->GetAttr("fancy_upscaling",
102                                             &flags_.fancy_upscaling));
103     OP_REQUIRES_OK(context,
104                     context->GetAttr("try_recover_truncated",
105                                     &flags_.try_recover_truncated_jpeg));
106     OP_REQUIRES_OK(context,
107                     context->GetAttr("acceptable_fraction",
108                                     &flags_.min_acceptable_fraction));
109     string dct_method;
110     OP_REQUIRES_OK(context, context->GetAttr("dct_method", &dct_method));
111     OP_REQUIRES(
112         context,
113         (dct_method.empty() || dct_method == "INTEGER_FAST" ||
114          dct_method == "INTEGER_ACCURATE"),
115         errors::InvalidArgument("dct_method must be one of "
116                                 "{', 'INTEGER_FAST', 'INTEGER_ACCURATE'}"));
117     // The TensorFlow-chosen default for JPEG decoding is IFAST, sacrificing
118     // image quality for speed.
119     if (dct_method.empty() || dct_method == "INTEGER_FAST") {
120         flags_.dct_method = JDCT_IFAST;
121     } else if (dct_method == "INTEGER_ACCURATE") {
122         flags_.dct_method = JDCT_ISLOW;
123     }
124 } else {
125     flags_ = jpeg::UncompressFlags();
126     flags_.dct_method = JDCT_IFAST;
127 }

```

```

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

```

```

177 // Helper for decoding BMP.
178 inline int16 ByteSwapInt16ForBigEndian(int16_t x) {
179     if (!port::kLittleEndian) {
180         return BYTE_SWAP_16(x);
181     } else {
182         return x;
183     }
184 }
185
186 void Compute(OpKernelContext* context) override {
187     const Tensor& contents = context->input(0);
188     OP_REQUIRES(
189         context, TensorShapeUtils::IsScalar(contents.shape()),
190         errors::InvalidArgument("`contents` must be scalar but got shape",
191             contents.shape().DebugString()));
192     const StringPiece input = contents.scalar<tstring>();
193     OP_REQUIRES(context, !input.empty(),
194         errors::InvalidArgument("Input is empty.));
195     OP_REQUIRES(context, input.size() <= std::numeric_limits<int>::max(),
196         errors::InvalidArgument(
197             "Input contents are too large for int: ", input.size()));
198
199     // Parse magic bytes to determine file format.
200     switch (ClassifyFileFormat(input)) {
201     case kJpgFormat:
202         DecodeJpegV2(context, input);
203         break;
204     case kPngFormat:
205         DecodePngV2(context, input);
206         break;
207     case kGifFormat:
208         DecodeGifV2(context, input);
209         break;
210     case kBmpFormat:
211         DecodeBmpV2(context, input);
212         break;
213     case kUnknownFormat:
214         OP_REQUIRES(context, false,
215             errors::InvalidArgument("Unknown image file format. One of "
216                 "JPEG, PNG, GIF, BMP required.));
217         break;
218     }
219 }
220
221 void DecodeJpegV2(OpKernelContext* context, StringPiece input) {
222     OP_REQUIRES(context, channels_ == 0 || channels_ == 1 || channels_ == 3,
223         errors::InvalidArgument("JPEG does not support 4 channels"));
224
225     // Use local copy of flags to avoid race condition as the class member is

```

```

226 // shared among different invocations.
227 jpeg::UncompressFlags flags = flags_;
228 flags.components = channels_;
229
230 if (op_type_ == "DecodeAndCropJpeg") {
231     flags.crop = true;
232     // Update flags to include crop window.
233     const Tensor& crop_window = context->input(1);
234     OP_REQUIRES(context, crop_window.dims() == 1,
235                 errors::InvalidArgument("crop_window must be 1-D, got shape ",
236                                         crop_window.shape().DebugString()));
237     OP_REQUIRES(context, crop_window.dim_size(0) == 4,
238                 errors::InvalidArgument("crop_size must have four elements ",
239                                         crop_window.shape().DebugString()));
240     auto crop_window_vec = crop_window.vec<int32>();
241     flags.crop_y = crop_window_vec(0);
242     flags.crop_x = crop_window_vec(1);
243     flags.crop_height = crop_window_vec(2);
244     flags.crop_width = crop_window_vec(3);
245 } else if (op_type_ == "DecodeBmp") {
246     // TODO(b/171060723): Only DecodeBmp as op_type_ is not acceptable here
247     // because currently `decode_(jpeg|png|gif)` ops can decode any one of
248     // jpeg, png or gif but not bmp. Similarly, `decode_bmp` cannot decode
249     // anything but bmp formats. This behavior needs to be revisited. For more
250     // details, please refer to the bug.
251     OP_REQUIRES(context, false,
252                 errors::InvalidArgument(
253                     "Trying to decode JPEG format using DecodeBmp op. Use "
254                     "`decode_jpeg` or `decode_image` instead."));
255 }
256
257 // Output tensor and the image buffer size.
258 Tensor* output = nullptr;
259 int buffer_size = 0;
260
261 // Decode JPEG. Directly allocate to the output buffer if data type is
262 // uint8 (to save extra copying). Otherwise, allocate a new uint8 buffer
263 // with buffer size. `jpeg::Uncompress` supports uint8 only.
264 uint8* buffer = jpeg::Uncompress(
265     input.data(), input.size(), flags, nullptr /* nwarn */,
266     [&](int width, int height, int channels) -> uint8* {
267         buffer_size = height * width * channels;
268         Status status;
269         // By the existing API, we support decoding JPEG with `DecodeGif`
270         // op. We need to make sure to return 4-D shapes when using
271         // `DecodeGif`.
272         if (op_type_ == "DecodeGif") {
273             status = context->allocate_output(
274                 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 is uint8, 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;

```

```

324     png::DecodeContext decode;
325     OP_REQUIRES(
326         context, png::CommonInitDecode(input, channels_, channel_bits, &decode),
327         errors::InvalidArgument("Invalid PNG. Failed to initialize decoder.));
328
329     // Verify that width and height are not too large:
330     // - verify width and height don't overflow int.
331     // - width can later be multiplied by channels_ and sizeof(uint16), so
332     //   verify single dimension is not too large.
333     // - verify when width and height are multiplied together, there are a few
334     //   bits to spare as well.
335     const int width = static_cast<int>(decode.width);
336     const int height = static_cast<int>(decode.height);
337     const int64_t total_size =
338         static_cast<int64_t>(width) * static_cast<int64_t>(height);
339     if (width != static_cast<int64_t>(decode.width) || width <= 0 ||
340         width >= (1LL << 27) || height != static_cast<int64_t>(decode.height) ||
341         height <= 0 || height >= (1LL << 27) || total_size >= (1LL << 29)) {
342         png::CommonFreeDecode(&decode);
343         OP_REQUIRES(context, false,
344             errors::InvalidArgument("PNG size too large for int: ",
345                                     decode.width, " by ", decode.height));
346     }
347
348     Tensor* output = nullptr;
349     Status status;
350     // By the existing API, we support decoding PNG with `DecodeGif` op.
351     // We need to make sure to return 4-D shapes when using `DecodeGif`.
352     if (op_type_ == "DecodeGif") {
353         status = context->allocate_output(
354             0, TensorShape({1, height, width, decode.channels}), &output);
355     } else {
356         status = context->allocate_output(
357             0, TensorShape({height, width, decode.channels}), &output);
358     }
359
360     if (op_type_ == "DecodeBmp") {
361         // TODO(b/171060723): Only DecodeBmp as op_type_ is not acceptable here
362         // because currently `decode_(jpeg|png|gif)` ops can decode any one of
363         // jpeg, png or gif but not bmp. Similarly, `decode_bmp` cannot decode
364         // anything but bmp formats. This behavior needs to be revisited. For more
365         // details, please refer to the bug.
366         OP_REQUIRES(context, false,
367             errors::InvalidArgument(
368                 "Trying to decode PNG format using DecodeBmp op. Use "
369                 "`decode_png` or `decode_image` instead.));
370     } else if (op_type_ == "DecodeAndCropJpeg") {
371         OP_REQUIRES(context, false,
372             errors::InvalidArgument(

```



```

373         "DecodeAndCropJpeg operation can run on JPEG only, but "
374         "detected PNG."));
375     }
376
377     if (!status.ok()) png::CommonFreeDecode(&decode);
378     OP_REQUIRES_OK(context, status);
379
380     if (data_type_ == DataType::DT_UINT8) {
381         OP_REQUIRES(
382             context,
383             png::CommonFinishDecode(
384                 reinterpret_cast<png_bytep>(output->flat<uint8>().data()),
385                 decode.channels * width * sizeof(uint8), &decode),
386             errors::InvalidArgument("Invalid PNG data, size ", input.size()));
387     } else if (data_type_ == DataType::DT_UINT16) {
388         OP_REQUIRES(
389             context,
390             png::CommonFinishDecode(
391                 reinterpret_cast<png_bytep>(output->flat<uint16>().data()),
392                 decode.channels * width * sizeof(uint16), &decode),
393             errors::InvalidArgument("Invalid PNG data, size ", input.size()));
394     } else if (data_type_ == DataType::DT_FLOAT) {
395         // `png::CommonFinishDecode` does not support `float`. First allocate
396         // uint16 buffer for the image and decode in uint16 (lossless). Wrap the
397         // buffer in `unique_ptr` so that we don't forget to delete the buffer.
398         std::unique_ptr<uint16[]> buffer(
399             new uint16[height * width * decode.channels]);
400         OP_REQUIRES(
401             context,
402             png::CommonFinishDecode(reinterpret_cast<png_bytep>(buffer.get()),
403                                     decode.channels * width * sizeof(uint16),
404                                     &decode),
405             errors::InvalidArgument("Invalid PNG data, size ", input.size()));
406
407         // Convert uint16 image data to desired data type.
408         // Use eigen threadpooling to speed up the copy operation.
409         const auto& device = context->eigen_device<Eigen::ThreadPoolDevice>();
410         TTypes<uint16, 3>::UnalignedConstTensor buf(buffer.get(), height, width,
411                                                     decode.channels);
412         float scale = 1. / std::numeric_limits<uint16>::max();
413         // Fill output tensor with desired dtype.
414         output->tensor<float, 3>().device(device) = buf.cast<float>() * scale;
415     }
416 }
417
418 void DecodeGifV2(OpKernelContext* context, StringPiece input) {
419     // GIF has 3 channels.
420     OP_REQUIRES(context, channels_ == 0 || channels_ == 3,
421                 errors::InvalidArgument("channels must be 0 or 3 for GIF, got ",

```

```

422         channels_));
423
424     if (op_type_ == "DecodeBmp") {
425         // TODO(b/171060723): Only DecodeBmp as op_type_ is not acceptable here
426         // because currently `decode_(jpeg|png|gif)` ops can decode any one of
427         // jpeg, png or gif but not bmp. Similarly, `decode_bmp` cannot decode
428         // anything but bmp formats. This behavior needs to be revisited. For more
429         // details, please refer to the bug.
430         OP_REQUIRES(context, false,
431             errors::InvalidArgument(
432                 "Trying to decode GIF format using DecodeBmp op. Use "
433                 "`decode_gif` or `decode_image` instead."));
434     } else if (op_type_ == "DecodeAndCropJpeg") {
435         OP_REQUIRES(context, false,
436             errors::InvalidArgument(
437                 "DecodeAndCropJpeg operation can run on JPEG only, but "
438                 "detected GIF."));
439     }
440
441     // Decode GIF, allocating tensor if dtype is uint8, otherwise defer tensor
442     // allocation til after dtype conversion is done. `gif::Decode` supports
443     // uint8 only.
444     Tensor* output = nullptr;
445     int buffer_size = 0;
446     string error_string;
447     uint8* buffer = gif::Decode(
448         input.data(), input.size(),
449         [&](int num_frames, int width, int height, int channels) -> uint8* {
450             buffer_size = num_frames * height * width * channels;
451
452             Status status;
453             // By the existing API, we support decoding GIF with `decode_jpeg` or
454             // with `decode_png` if the GIF is a single-frame GIF (non-animated).
455             // We need to make sure to return 3-D shapes when using in this case.
456             if (op_type_ == "DecodePng" || op_type_ == "DecodeJpeg") {
457                 if (num_frames == 1) {
458                     status = context->allocate_output(
459                         0, TensorShape({height, width, channels}), &output);
460                 } else {
461                     status = errors::InvalidArgument(
462                         "Got ", num_frames, " frames, but animated gifs ",
463                         "can only be decoded by tf.io.decode_gif or ",
464                         "tf.io.decode_image");
465                 }
466             } else if (op_type_ == "DecodeGif" ||
467                 (op_type_ == "DecodeImage" && expand_animations_)) {
468                 status = context->allocate_output(
469                     0, TensorShape({num_frames, height, width, channels}), &output);
470             } 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 uint8, 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
```