

5100e359ae ▾

...

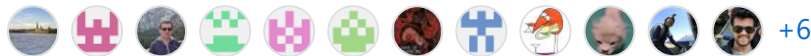
tensorflow / tensorflow / lite / kernels / depthwise\_conv.cc



mihaimaruseac Fix a null pointer exception caused by branching on uninitialized... ... ✖

History

18 contributors



648 lines (584 sloc) | 27.7 KB

...

```

1  /* Copyright 2017 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
6
7      http://www.apache.org/licenses/LICENSE-2.0
8
9  Unless required by applicable law or agreed to in writing, software
10 distributed under the License is distributed on an "AS IS" BASIS,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 =====*/
15
16 #include "tensorflow/lite/kernels/internal/optimized/integer_ops/depthwise_conv.h"
17
18 #include <stddef.h>
19 #include <stdint.h>
20
21 #include <vector>
22
23 #include "tensorflow/lite/c/builtin_op_data.h"
24 #include "tensorflow/lite/c/common.h"
25 #include "tensorflow/lite/kernels/cpu_backend_context.h"
26 #include "tensorflow/lite/kernels/internal/compatibility.h"
27 #include "tensorflow/lite/kernels/internal/optimized/cpu_check.h"
28 #include "tensorflow/lite/kernels/internal/optimized/depthwiseconv_multithread.h"
29 #include "tensorflow/lite/kernels/internal/optimized/integer_ops/depthwise_conv_hybrid.h"

```

```

30 #include "tensorflow/lite/kernels/internal/optimized/neon_check.h"
31 #include "tensorflow/lite/kernels/internal/quantization_util.h"
32 #include "tensorflow/lite/kernels/internal/reference/depthwiseconv_float.h"
33 #include "tensorflow/lite/kernels/internal/reference/depthwiseconv_uint8.h"
34 #include "tensorflow/lite/kernels/internal/reference/integer_ops/depthwise_conv.h"
35 #include "tensorflow/lite/kernels/internal/tensor.h"
36 #include "tensorflow/lite/kernels/internal/tensor_ctypes.h"
37 #include "tensorflow/lite/kernels/internal/tensor_utils.h"
38 #include "tensorflow/lite/kernels/internal/types.h"
39 #include "tensorflow/lite/kernels/kernel_util.h"
40 #include "tensorflow/lite/kernels/padding.h"
41
42 namespace tflite {
43 namespace ops {
44 namespace builtin {
45 namespace depthwise_conv {
46
47 constexpr int kInputTensor = 0;
48 constexpr int kFilterTensor = 1;
49 constexpr int kBiasTensor = 2;
50 constexpr int kOutputTensor = 0;
51
52 // This file has three implementation of DepthwiseConv.
53 enum KernelType {
54     kReference,
55     kGenericOptimized, // Neon-free
56     kNeonOptimized,
57 };
58
59 const int kTensorNotAllocated = -1;
60
61 struct OpData {
62     TfLitePaddingValues padding;
63     // The scaling factor from input to output (aka the 'real multiplier') can
64     // be represented as a fixed point multiplier plus a left shift.
65     int32_t output_multiplier;
66     int output_shift;
67     // The range of the fused activation layer. For example for kNone and
68     // uint8_t these would be 0 and 255.
69     int32_t output_activation_min;
70     int32_t output_activation_max;
71
72     // Per channel output multiplier and shift.
73     std::vector<int32_t> per_channel_output_multiplier;
74     std::vector<int> per_channel_output_shift;
75
76     // Hybrid per channel temporary tensors.
77     int input_quantized_id = kTensorNotAllocated;
78     int scaling_factors_id = kTensorNotAllocated;

```

```

79     int input_offset_id = kTensorNotAllocated;
80     int32_t input_quantized_index;
81     int32_t scaling_factors_index;
82     int32_t input_offset_index;
83 };
84
85 void* Init(TfLiteContext* context, const char* buffer, size_t length) {
86     // This is a builtin op, so we don't use the contents in 'buffer', if any.
87     // Instead, we allocate a new object to carry information from Prepare() to
88     // Eval().
89     return new OpData;
90 }
91
92 void Free(TfLiteContext* context, void* buffer) {
93     delete reinterpret_cast<OpData*>(buffer);
94 }
95
96 TfLiteStatus Prepare(TfLiteContext* context, TfLiteNode* node) {
97     auto* params =
98         reinterpret_cast<TfLiteDepthwiseConvParams*>(node->builtin_data);
99     OpData* data = reinterpret_cast<OpData*>(node->user_data);
100
101     bool has_bias = NumInputs(node) == 3;
102
103     TF_LITE_ENSURE(context, has_bias || NumInputs(node) == 2);
104     const TfLiteTensor* input;
105     TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kInputTensor, &input));
106     const TfLiteTensor* filter;
107     TF_LITE_ENSURE_OK(context,
108         GetInputSafe(context, node, kFilterTensor, &filter));
109     const TfLiteTensor* bias = nullptr;
110
111     TF_LITE_ENSURE_EQ(context, NumOutputs(node), 1);
112     TfLiteTensor* output;
113     TF_LITE_ENSURE_OK(context,
114         GetOutputSafe(context, node, kOutputTensor, &output));
115
116     TF_LITE_ENSURE_EQ(context, NumDimensions(input), 4);
117     TF_LITE_ENSURE_EQ(context, NumDimensions(filter), 4);
118
119     const TfLiteType data_type = input->type;
120
121     const TfLiteType filter_type = filter->type;
122     const bool is_hybrid =
123         data_type == kTfLiteFloat32 && filter_type == kTfLiteInt8;
124     TF_LITE_ENSURE(context,
125         data_type == kTfLiteFloat32 || data_type == kTfLiteUInt8 ||
126         data_type == kTfLiteInt8 || data_type == kTfLiteInt16);
127     TF_LITE_ENSURE_TYPES_EQ(context, output->type, data_type);

```

```

128     if (!is_hybrid) {
129         TF_LITE_ENSURE(context,
130             filter->type == data_type || data_type == kTfLiteInt16);
131     }
132
133     if (data_type == kTfLiteInt16) {
134         TF_LITE_ENSURE_EQ(context, input->params.zero_point, 0);
135         TF_LITE_ENSURE_EQ(context, output->params.zero_point, 0);
136     }
137
138     // Filter in DepthwiseConv is expected to be [1, H, W, 0].
139     TF_LITE_ENSURE_EQ(context, SizeOfDimension(filter, 0), 1);
140
141     if (has_bias) {
142         TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kBiasTensor, &bias));
143         if (data_type == kTfLiteUInt8 || data_type == kTfLiteInt8) {
144             TF_LITE_ENSURE_TYPES_EQ(context, bias->type, kTfLiteInt32);
145             TF_LITE_ENSURE_EQ(context, bias->params.zero_point, 0);
146         } else if (data_type == kTfLiteInt16) {
147             TF_LITE_ENSURE_TYPES_EQ(context, bias->type, kTfLiteInt64);
148             TF_LITE_ENSURE_EQ(context, bias->params.zero_point, 0);
149         } else {
150             TF_LITE_ENSURE_TYPES_EQ(context, bias->type, data_type);
151         }
152         TF_LITE_ENSURE_EQ(context, NumDimensions(bias), 1);
153         TF_LITE_ENSURE_EQ(context, SizeOfDimension(filter, 3),
154             SizeOfDimension(bias, 0));
155     }
156
157     int channels_out = SizeOfDimension(filter, 3);
158     int width = SizeOfDimension(input, 2);
159     int height = SizeOfDimension(input, 1);
160     int filter_width = SizeOfDimension(filter, 2);
161     int filter_height = SizeOfDimension(filter, 1);
162     int batches = SizeOfDimension(input, 0);
163
164     // Matching GetWindowedOutputSize in TensorFlow.
165     auto padding = params->padding;
166     int out_width, out_height;
167
168     data->padding = ComputePaddingHeightWidth(
169         params->stride_height, params->stride_width,
170         params->dilation_height_factor, params->dilation_width_factor, height,
171         width, filter_height, filter_width, padding, &out_height, &out_width);
172
173     // Note that quantized inference requires that all tensors have their
174     // parameters set. This is usually done during quantized training or
175     // calibration.
176     if (data_type != kTfLiteFloat32) {

```

```

177     TF_LITE_ENSURE_EQ(context, filter->quantization.type,
178                        kTfLiteAffineQuantization);
179     TF_LITE_ENSURE(context, filter->quantization.type != kTfLiteNoQuantization);
180     const auto* affine_quantization =
181         reinterpret_cast<TfLiteAffineQuantization*>(
182             filter->quantization.params);
183     TF_LITE_ENSURE(context, affine_quantization);
184     TF_LITE_ENSURE(context, affine_quantization->scale);
185     TF_LITE_ENSURE(context, (affine_quantization->scale->size == 1 ||
186                             affine_quantization->scale->size == channels_out));
187
188     data->per_channel_output_multiplier.resize(channels_out);
189     data->per_channel_output_shift.resize(channels_out);
190     TF_LITE_ENSURE_STATUS(tflite::PopulateConvolutionQuantizationParams(
191         context, input, filter, bias, output, params->activation,
192         &data->output_multiplier, &data->output_shift,
193         &data->output_activation_min, &data->output_activation_max,
194         data->per_channel_output_multiplier.data(),
195         data->per_channel_output_shift.data(), channels_out));
196 }
197
198 if (is_hybrid) {
199     TF_LITE_ENSURE(context, filter->quantization.type != kTfLiteNoQuantization);
200     const auto* affine_quantization =
201         reinterpret_cast<TfLiteAffineQuantization*>(
202             filter->quantization.params);
203     TF_LITE_ENSURE(context, affine_quantization);
204     TF_LITE_ENSURE(context, affine_quantization->scale);
205     TF_LITE_ENSURE_EQ(
206         context, affine_quantization->scale->size,
207         filter->dims->data[affine_quantization->quantized_dimension]);
208
209     int temporaries_count = 0;
210     data->input_quantized_index = temporaries_count;
211     if (data->input_quantized_id == kTensorNotAllocated) {
212         TF_LITE_ENSURE_OK(
213             context, context->AddTensors(context, 1, &data->input_quantized_id));
214     }
215     ++temporaries_count;
216     data->scaling_factors_index = temporaries_count;
217     if (data->scaling_factors_id == kTensorNotAllocated) {
218         TF_LITE_ENSURE_OK(
219             context, context->AddTensors(context, 1, &data->scaling_factors_id));
220     }
221     ++temporaries_count;
222     data->input_offset_index = temporaries_count;
223     if (data->input_offset_id == kTensorNotAllocated) {
224         TF_LITE_ENSURE_OK(
225             context, context->AddTensors(context, 1, &data->input_offset_id));

```

```

226     }
227     ++temporaries_count;
228
229     TfLiteIntArrayFree(node->temporaries);
230     node->temporaries = TfLiteIntArrayCreate(temporaries_count);
231
232     node->temporaries->data[data->input_quantized_index] =
233         data->input_quantized_id;
234     TfLiteTensor* input_quantized;
235     TF_LITE_ENSURE_OK(
236         context, GetTemporarySafe(context, node, data->input_quantized_index,
237                                   &input_quantized));
238     input_quantized->type = kTfLiteInt8;
239     input_quantized->allocation_type = kTfLiteArenaRw;
240     if (!TfLiteIntArrayEqual(input_quantized->dims, input->dims)) {
241         TfLiteIntArray* input_quantized_size = TfLiteIntArrayCopy(input->dims);
242         TF_LITE_ENSURE_OK(context, context->ResizeTensor(context, input_quantized,
243                                                         input_quantized_size));
244     }
245     node->temporaries->data[data->scaling_factors_index] =
246         data->scaling_factors_id;
247     TfLiteTensor* scaling_factors;
248     TF_LITE_ENSURE_OK(
249         context, GetTemporarySafe(context, node, data->scaling_factors_index,
250                                   &scaling_factors));
251     scaling_factors->type = kTfLiteFloat32;
252     scaling_factors->allocation_type = kTfLiteArenaRw;
253     const int batch_size = SizeOfDimension(input, 0);
254     int scaling_dims[1] = {batch_size};
255     if (!TfLiteIntArrayEqualsArray(scaling_factors->dims, 1, scaling_dims)) {
256         TfLiteIntArray* scaling_factors_size = TfLiteIntArrayCreate(1);
257         scaling_factors_size->data[0] = batch_size;
258         TF_LITE_ENSURE_OK(context, context->ResizeTensor(context, scaling_factors,
259                                                         scaling_factors_size));
260     }
261     node->temporaries->data[data->input_offset_index] = data->input_offset_id;
262     TfLiteTensor* input_offsets;
263     TF_LITE_ENSURE_OK(context,
264         GetTemporarySafe(context, node, data->input_offset_index,
265                           &input_offsets));
266     input_offsets->type = kTfLiteInt32;
267     input_offsets->allocation_type = kTfLiteArenaRw;
268     if (!TfLiteIntArrayEqualsArray(input_offsets->dims, 1, scaling_dims)) {
269         TfLiteIntArray* input_offsets_size = TfLiteIntArrayCreate(1);
270         input_offsets_size->data[0] = batch_size;
271         TF_LITE_ENSURE_OK(context, context->ResizeTensor(context, input_offsets,
272                                                         input_offsets_size));
273     }
274 }

```

```

275
276 TfliteIntArray* outputSize = TfliteIntArrayCreate(4);
277 outputSize->data[0] = batches;
278 outputSize->data[1] = out_height;
279 outputSize->data[2] = out_width;
280 outputSize->data[3] = channels_out;
281 return context->ResizeTensor(context, output, outputSize);
282 }
283
284 TfliteStatus ComputeDepthMultiplier(TfliteContext* context,
285                                     const TfliteTensor* input,
286                                     const TfliteTensor* filter,
287                                     int16* depth_multiplier) {
288     int num_filter_channels = SizeOfDimension(filter, 3);
289     int num_input_channels = SizeOfDimension(input, 3);
290     TF_LITE_ENSURE(context, num_input_channels != 0);
291     TF_LITE_ENSURE_EQ(context, num_filter_channels % num_input_channels, 0);
292     *depth_multiplier = num_filter_channels / num_input_channels;
293     return kTfLiteOk;
294 }
295
296 template <KernelType kernel_type>
297 TfliteStatus EvalFloat(TfliteContext* context, TfliteNode* node,
298                       TfliteDepthwiseConvParams* params, OpData* data,
299                       const TfliteTensor* input, const TfliteTensor* filter,
300                       const TfliteTensor* bias, TfliteTensor* output) {
301     float output_activation_min, output_activation_max;
302     CalculateActivationRange(params->activation, &output_activation_min,
303                             &output_activation_max);
304
305     DepthwiseParams op_params;
306     op_params.padding_type = PaddingType::kSame;
307     op_params.padding_values.width = data->padding.width;
308     op_params.padding_values.height = data->padding.height;
309     op_params.stride_width = params->stride_width;
310     op_params.stride_height = params->stride_height;
311     op_params.dilation_width_factor = params->dilation_width_factor;
312     op_params.dilation_height_factor = params->dilation_height_factor;
313     op_params.float_activation_min = output_activation_min;
314     op_params.float_activation_max = output_activation_max;
315     TF_LITE_ENSURE_STATUS(ComputeDepthMultiplier(context, input, filter,
316                                                  &op_params.depth_multiplier));
317     if (kernel_type == kReference) {
318         reference_ops::DepthwiseConv(
319             op_params, GetTensorShape(input), GetTensorData<float>(input),
320             GetTensorShape(filter), GetTensorData<float>(filter),
321             GetTensorShape(bias), GetTensorData<float>(bias),
322             GetTensorShape(output), GetTensorData<float>(output));
323     } else {

```

```

324     optimized_ops::DepthwiseConv<float, float>(
325         op_params, GetTensorShape(input), GetTensorData<float>(input),
326         GetTensorShape(filter), GetTensorData<float>(filter),
327         GetTensorShape(bias), GetTensorData<float>(bias),
328         GetTensorShape(output), GetTensorData<float>(output),
329         CpuBackendContext::GetFromContext(context));
330 }
331 return kTfLiteOk;
332 }
333
334 template <KernelType kernel_type>
335 TfLiteStatus EvalQuantized(TfLiteContext* context, TfLiteNode* node,
336                             TfLiteDepthwiseConvParams* params, OpData* data,
337                             const TfLiteTensor* input,
338                             const TfLiteTensor* filter, const TfLiteTensor* bias,
339                             TfLiteTensor* output) {
340     auto input_offset = -input->params.zero_point;
341     auto filter_offset = -filter->params.zero_point;
342     auto output_offset = output->params.zero_point;
343
344     DepthwiseParams op_params;
345     op_params.padding_type = PaddingType::kSame;
346     op_params.padding_values.width = data->padding.width;
347     op_params.padding_values.height = data->padding.height;
348     op_params.stride_width = params->stride_width;
349     op_params.stride_height = params->stride_height;
350     op_params.dilation_width_factor = params->dilation_width_factor;
351     op_params.dilation_height_factor = params->dilation_height_factor;
352     op_params.input_offset = input_offset;
353     op_params.weights_offset = filter_offset;
354     op_params.output_offset = output_offset;
355     op_params.output_multiplier = data->output_multiplier;
356     op_params.output_shift = -data->output_shift;
357     op_params.quantized_activation_min = data->output_activation_min;
358     op_params.quantized_activation_max = data->output_activation_max;
359     TF_LITE_ENSURE_STATUS(ComputeDepthMultiplier(context, input, filter,
360                                                    &op_params.depth_multiplier));
361     if (kernel_type == kReference) {
362         reference_ops::DepthwiseConv(
363             op_params, GetTensorShape(input), GetTensorData<uint8_t>(input),
364             GetTensorShape(filter), GetTensorData<uint8_t>(filter),
365             GetTensorShape(bias), GetTensorData<int32_t>(bias),
366             GetTensorShape(output), GetTensorData<uint8_t>(output));
367     } else {
368         optimized_ops::DepthwiseConv<uint8, int32>(
369             op_params, GetTensorShape(input), GetTensorData<uint8_t>(input),
370             GetTensorShape(filter), GetTensorData<uint8_t>(filter),
371             GetTensorShape(bias), GetTensorData<int32_t>(bias),
372             GetTensorShape(output), GetTensorData<uint8_t>(output),

```



```

373         CpuBackendContext::GetFromContext(context));
374     }
375     return kTfLiteOk;
376 }
377
378 template <KernelType kernel_type>
379 TfLiteStatus EvalQuantizedPerChannel(TfLiteContext* context, TfLiteNode* node,
380                                     TfLiteDepthwiseConvParams* params,
381                                     OpData* data, const TfLiteTensor* input,
382                                     const TfLiteTensor* filter,
383                                     const TfLiteTensor* bias,
384                                     TfLiteTensor* output) {
385     DepthwiseParams op_params;
386     op_params.padding_type = PaddingType::kSame;
387     op_params.padding_values.width = data->padding.width;
388     op_params.padding_values.height = data->padding.height;
389     op_params.stride_width = params->stride_width;
390     op_params.stride_height = params->stride_height;
391     op_params.dilation_width_factor = params->dilation_width_factor;
392     op_params.dilation_height_factor = params->dilation_height_factor;
393     op_params.input_offset = -input->params.zero_point;
394     op_params.weights_offset = 0;
395     op_params.output_offset = output->params.zero_point;
396     op_params.quantized_activation_min = data->output_activation_min;
397     op_params.quantized_activation_max = data->output_activation_max;
398     TF_LITE_ENSURE_STATUS(ComputeDepthMultiplier(context, input, filter,
399                                                  &op_params.depth_multiplier));
400
401     if (kernel_type == kReference) {
402         reference_integer_ops::DepthwiseConvPerChannel(
403             op_params, data->per_channel_output_multiplier.data(),
404             data->per_channel_output_shift.data(), GetTensorShape(input),
405             GetTensorData<int8>(input), GetTensorShape(filter),
406             GetTensorData<int8>(filter), GetTensorShape(bias),
407             GetTensorData<int32>(bias), GetTensorShape(output),
408             GetTensorData<int8>(output));
409     } else {
410         optimized_integer_ops::DepthwiseConvPerChannel(
411             op_params, data->per_channel_output_multiplier.data(),
412             data->per_channel_output_shift.data(), GetTensorShape(input),
413             GetTensorData<int8>(input), GetTensorShape(filter),
414             GetTensorData<int8>(filter), GetTensorShape(bias),
415             GetTensorData<int32>(bias), GetTensorShape(output),
416             GetTensorData<int8>(output),
417             CpuBackendContext::GetFromContext(context));
418     }
419     return kTfLiteOk;
420 }
421

```

```

422 TfLiteStatus EvalQuantizedPerChannel16x8(
423     const TfLiteDepthwiseConvParams* params, const OpData* data,
424     const TfLiteTensor* input, const TfLiteTensor* filter,
425     const TfLiteTensor* bias, TfLiteTensor* output) {
426     DepthwiseParams op_params;
427     op_params.padding_type = PaddingType::kSame;
428     op_params.padding_values.width = data->padding.width;
429     op_params.padding_values.height = data->padding.height;
430     op_params.stride_width = params->stride_width;
431     op_params.stride_height = params->stride_height;
432     op_params.dilation_width_factor = params->dilation_width_factor;
433     op_params.dilation_height_factor = params->dilation_height_factor;
434     op_params.depth_multiplier = params->depth_multiplier;
435     op_params.weights_offset = 0;
436     op_params.quantized_activation_min = data->output_activation_min;
437     op_params.quantized_activation_max = data->output_activation_max;
438
439     reference_integer_ops::DepthwiseConvPerChannel(
440         op_params, data->per_channel_output_multiplier.data(),
441         data->per_channel_output_shift.data(), GetTensorShape(input),
442         GetTensorData<int16>(input), GetTensorShape(filter),
443         GetTensorData<int8>(filter), GetTensorShape(bias),
444         GetTensorData<std::int64_t>(bias), GetTensorShape(output),
445         GetTensorData<int16>(output));
446
447     return kTfLiteOk;
448 }
449
450 template <KernelType kernel_type>
451 TfLiteStatus EvalHybridPerChannel(TfLiteContext* context, TfLiteNode* node,
452     TfLiteDepthwiseConvParams* params,
453     OpData* data, const TfLiteTensor* input,
454     const TfLiteTensor* filter,
455     const TfLiteTensor* bias,
456     TfLiteTensor* output) {
457     float output_activation_min, output_activation_max;
458     CalculateActivationRange(params->activation, &output_activation_min,
459         &output_activation_max);
460     const int batch_size = SizeOfDimension(input, 0);
461     TF_LITE_ENSURE(context, batch_size != 0);
462     const int input_size = NumElements(input) / batch_size;
463     TfLiteTensor* input_quantized;
464     TF_LITE_ENSURE_OK(context,
465         GetTemporarySafe(context, node, data->input_quantized_index,
466             &input_quantized));
467     int8_t* quantized_input_ptr_batch = input_quantized->data.int8;
468     TfLiteTensor* scaling_factors_tensor;
469     TF_LITE_ENSURE_OK(context,
470         GetTemporarySafe(context, node, data->scaling_factors_index,

```

```

471         &scaling_factors_tensor));
472 float* scaling_factors_ptr = GetTensorData<float>(scaling_factors_tensor);
473 TfLiteTensor* input_offset_tensor;
474 TF_LITE_ENSURE_OK(context,
475     GetTemporarySafe(context, node, data->input_offset_index,
476         &input_offset_tensor));
477 int32_t* input_offset_ptr = GetTensorData<int32_t>(input_offset_tensor);
478
479 for (int b = 0; b < batch_size; ++b) {
480     const int offset = b * input_size;
481     tensor_utils::AsymmetricQuantizeFloats(
482         GetTensorData<float>(input) + offset, input_size,
483         quantized_input_ptr_batch + offset, &scaling_factors_ptr[b],
484         &input_offset_ptr[b]);
485 }
486
487 DepthwiseParams op_params;
488 op_params.padding_type = PaddingType::kSame;
489 op_params.padding_values.width = data->padding.width;
490 op_params.padding_values.height = data->padding.height;
491 op_params.stride_width = params->stride_width;
492 op_params.stride_height = params->stride_height;
493 op_params.dilation_width_factor = params->dilation_width_factor;
494 op_params.dilation_height_factor = params->dilation_height_factor;
495 op_params.depth_multiplier = params->depth_multiplier;
496
497 op_params.weights_offset = 0;
498 op_params.float_activation_min = output_activation_min;
499 op_params.float_activation_max = output_activation_max;
500 TF_LITE_ENSURE(context, filter->quantization.type != kTfLiteNoQuantization);
501 const auto* affine_quantization =
502     reinterpret_cast<TfLiteAffineQuantization*>(filter->quantization.params);
503 if (kernel_type == kReference) {
504     reference_integer_ops::DepthwiseConvHybridPerChannel(
505         op_params, scaling_factors_ptr, GetTensorShape(input),
506         quantized_input_ptr_batch, GetTensorShape(filter),
507         GetTensorData<int8>(filter), GetTensorShape(bias),
508         GetTensorData<float>(bias), GetTensorShape(output),
509         GetTensorData<float>(output), affine_quantization->scale->data,
510         input_offset_ptr);
511 } else {
512     optimized_integer_ops::DepthwiseConvHybridPerChannel(
513         op_params, scaling_factors_ptr, GetTensorShape(input),
514         quantized_input_ptr_batch, GetTensorShape(filter),
515         GetTensorData<int8>(filter), GetTensorShape(bias),
516         GetTensorData<float>(bias), GetTensorShape(output),
517         GetTensorData<float>(output), affine_quantization->scale->data,
518         input_offset_ptr, CpuBackendContext::GetFromContext(context));
519 }

```

```

520
521     return kTfLiteOk;
522 }
523
524 template <KernelType kernel_type, TfLiteType input_type>
525 TfLiteStatus EvalImpl(TfLiteContext* context, TfLiteNode* node) {
526     auto* params =
527         reinterpret_cast<TfLiteDepthwiseConvParams*>(node->builtin_data);
528     OpData* data = reinterpret_cast<OpData*>(node->user_data);
529
530     TfLiteTensor* output;
531     TF_LITE_ENSURE_OK(context,
532         GetOutputSafe(context, node, kOutputTensor, &output));
533     const TfLiteTensor* input;
534     TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kInputTensor, &input));
535     const TfLiteTensor* filter;
536     TF_LITE_ENSURE_OK(context,
537         GetInputSafe(context, node, kFilterTensor, &filter));
538     const TfLiteTensor* bias =
539         (NumInputs(node) == 3) ? GetInput(context, node, kBiasTensor) : nullptr;
540     TFLITE_DCHECK_EQ(input_type, input->type);
541
542     switch (input_type) { // Already know in/out types are same.
543     case kTfLiteFloat32:
544         if (filter->type == kTfLiteFloat32) {
545             return EvalFloat<kernel_type>(context, node, params, data, input,
546                 filter, bias, output);
547         } else if (filter->type == kTfLiteInt8) {
548             return EvalHybridPerChannel<kernel_type>(context, node, params, data,
549                 input, filter, bias, output);
550         } else {
551             TF_LITE_KERNEL_LOG(
552                 context, "Type %s with filter type %s not currently supported.",
553                 TfLiteTypeGetName(input->type), TfLiteTypeGetName(filter->type));
554             return kTfLiteError;
555         }
556         break;
557     case kTfLiteUInt8:
558         return EvalQuantized<kernel_type>(context, node, params, data, input,
559             filter, bias, output);
560         break;
561     case kTfLiteInt8:
562         return EvalQuantizedPerChannel<kernel_type>(context, node, params, data,
563             input, filter, bias, output);
564         break;
565     case kTfLiteInt16:
566         return EvalQuantizedPerChannel16x8(params, data, input, filter, bias,
567             output);
568         break;

```

```

569     default:
570         context->ReportError(context, "Type %d not currently supported.",
571                               input->type);
572     return kTfLiteError;
573 }
574 }
575
576 template <KernelType kernel_type>
577 TfLiteStatus Eval(TfLiteContext* context, TfLiteNode* node) {
578     const TfLiteTensor* input;
579     TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kInputTensor, &input));
580
581     switch (input->type) { // Already know in/out types are same.
582     case kTfLiteFloat32:
583         return EvalImpl<kernel_type, kTfLiteFloat32>(context, node);
584     case kTfLiteUInt8:
585         return EvalImpl<kernel_type, kTfLiteUInt8>(context, node);
586     case kTfLiteInt8:
587         return EvalImpl<kernel_type, kTfLiteInt8>(context, node);
588     case kTfLiteInt16:
589         return EvalImpl<kernel_type, kTfLiteInt16>(context, node);
590     default:
591         context->ReportError(context, "Type %d not currently supported.",
592                               input->type);
593         return kTfLiteError;
594     }
595 }
596
597 } // namespace depthwise_conv
598
599 TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_REF() {
600     static TfLiteRegistration r = {
601         depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
602         depthwise_conv::Eval<depthwise_conv::kReference>};
603     return &r;
604 }
605
606 TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_GENERIC_OPT() {
607     static TfLiteRegistration r = {
608         depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
609         depthwise_conv::Eval<depthwise_conv::kGenericOptimized>};
610     return &r;
611 }
612
613 TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_NEON_OPT() {
614     static TfLiteRegistration r = {
615         depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
616         depthwise_conv::Eval<depthwise_conv::kNeonOptimized>};
617     return &r;

```

```

618 }
619
620 TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_NEON_OPT_UINT8() {
621     static TfLiteRegistration r = {
622         depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
623         depthwise_conv::EvalImpl<depthwise_conv::kNeonOptimized, kTfLiteUInt8>;
624     return &r;
625 }
626
627 TfLiteRegistration* Register_DEPTHWISE_CONV_2D() {
628 #ifdef USE_NEON
629     return Register_DEPTHWISE_CONVOLUTION_NEON_OPT();
630 #else
631     return Register_DEPTHWISE_CONVOLUTION_GENERIC_OPT();
632 #endif
633 }
634
635 // Warning: Clients using this variant are responsible for ensuring that their
636 // models only need the UINT8 type. TfLite's op registration mechanism doesn't
637 // yet allow for more nuanced registration mechanisms.
638 TfLiteRegistration* Register_DEPTHWISE_CONV_2D_UINT8() {
639 #ifdef USE_NEON
640     return Register_DEPTHWISE_CONVOLUTION_NEON_OPT_UINT8();
641 #else
642     return Register_DEPTHWISE_CONV_2D();
643 #endif
644 }
645
646 } // namespace builtin
647 } // namespace ops
648 } // namespace tfLite

```