tensorflow / **tensorflow** Public

Code  Issues 2.1k  Pull requests 283  Actions  Projects 1  ...

5100e359ae

tensorflow / tensorflow / lite / kernels / **depthwise_conv.cc**

**mihaimaruseac** Fix a null pointer exception caused by branching on uninitialize... ... ✗ History

18 contributors +6

648 lines (584 sloc) | 27.7 KB

```
#include "tensorflow/lite/kernels/internal/optimized/integer_ops/depthwise_conv.h"

#include <stddef.h>
#include <stdint.h>

#include <vector>

#include "tensorflow/lite/c/builtin_op_data.h"
#include "tensorflow/lite/c/common.h"
#include "tensorflow/lite/kernels/cpu_backend_context.h"
#include "tensorflow/lite/kernels/internal/compatibility.h"
#include "tensorflow/lite/kernels/internal/optimized/cpu_check.h"
#include "tensorflow/lite/kernels/internal/optimized/depthwiseconv_multithread.h"
#include "tensorflow/lite/kernels/internal/optimized/integer_ops/depthwise_conv_hybrid.h"
```

```
#include "tensorflow/lite/kernels/internal/optimized/neon_check.h"
#include "tensorflow/lite/kernels/internal/quantization_util.h"
#include "tensorflow/lite/kernels/internal/reference/depthwiseconv_float.h"
#include "tensorflow/lite/kernels/internal/reference/depthwiseconv_uint8.h"
#include "tensorflow/lite/kernels/internal/reference/integer_ops/depthwise_conv.h"
#include "tensorflow/lite/kernels/internal/tensor.h"
#include "tensorflow/lite/kernels/internal/tensor_ctypes.h"
#include "tensorflow/lite/kernels/internal/tensor_utils.h"
#include "tensorflow/lite/kernels/internal/types.h"
#include "tensorflow/lite/kernels/kernel_util.h"
#include "tensorflow/lite/kernels/padding.h"

namespace tflite {
namespace ops {
namespace builtin {
namespace depthwise_conv {

constexpr int kInputTensor = 0;
constexpr int kFilterTensor = 1;
constexpr int kBiasTensor = 2;
constexpr int kOutputTensor = 0;

// This file has three implementation of DepthwiseConv.
enum KernelType {
  kReference,
  kGenericOptimized,  // Neon-free
  kNeonOptimized,
};

const int kTensorNotAllocated = -1;

struct OpData {
  TfLitePaddingValues padding;
  // The scaling factor from input to output (aka the 'real multiplier') can
  // be represented as a fixed point multiplier plus a left shift.
  int32_t output_multiplier;
  int output_shift;
  // The range of the fused activation layer. For example for kNone and
  // uint8_t these would be 0 and 255.
  int32_t output_activation_min;
  int32_t output_activation_max;

  // Per channel output multiplier and shift.
  std::vector<int32_t> per_channel_output_multiplier;
  std::vector<int> per_channel_output_shift;

  // Hybrid per channel temporary tensors.
  int input_quantized_id = kTensorNotAllocated;
  int scaling_factors_id = kTensorNotAllocated;
```

```cpp
  int input_offset_id = kTensorNotAllocated;
  int32_t input_quantized_index;
  int32_t scaling_factors_index;
  int32_t input_offset_index;
};

void* Init(TfLiteContext* context, const char* buffer, size_t length) {
  // This is a builtin op, so we don't use the contents in 'buffer', if any.
  // Instead, we allocate a new object to carry information from Prepare() to
  // Eval().
  return new OpData;
}

void Free(TfLiteContext* context, void* buffer) {
  delete reinterpret_cast<OpData*>(buffer);
}

TfLiteStatus Prepare(TfLiteContext* context, TfLiteNode* node) {
  auto* params =
      reinterpret_cast<TfLiteDepthwiseConvParams*>(node->builtin_data);
  OpData* data = reinterpret_cast<OpData*>(node->user_data);

  bool has_bias = NumInputs(node) == 3;

  TF_LITE_ENSURE(context, has_bias || NumInputs(node) == 2);
  const TfLiteTensor* input;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kInputTensor, &input));
  const TfLiteTensor* filter;
  TF_LITE_ENSURE_OK(context,
                    GetInputSafe(context, node, kFilterTensor, &filter));
  const TfLiteTensor* bias = nullptr;

  TF_LITE_ENSURE_EQ(context, NumOutputs(node), 1);
  TfLiteTensor* output;
  TF_LITE_ENSURE_OK(context,
                    GetOutputSafe(context, node, kOutputTensor, &output));

  TF_LITE_ENSURE_EQ(context, NumDimensions(input), 4);
  TF_LITE_ENSURE_EQ(context, NumDimensions(filter), 4);

  const TfLiteType data_type = input->type;

  const TfLiteType filter_type = filter->type;
  const bool is_hybrid =
      data_type == kTfLiteFloat32 && filter_type == kTfLiteInt8;
  TF_LITE_ENSURE(context,
                 data_type == kTfLiteFloat32 || data_type == kTfLiteUInt8 ||
                     data_type == kTfLiteInt8 || data_type == kTfLiteInt16);
  TF_LITE_ENSURE_TYPES_EQ(context, output->type, data_type);
  if (!is_hybrid) {
    TF_LITE_ENSURE(context,
                   filter->type == data_type || data_type == kTfLiteInt16);
  }

  if (data_type == kTfLiteInt16) {
    TF_LITE_ENSURE_EQ(context, input->params.zero_point, 0);
    TF_LITE_ENSURE_EQ(context, output->params.zero_point, 0);
  }

  // Filter in DepthwiseConv is expected to be [1, H, W, O].
  TF_LITE_ENSURE_EQ(context, SizeOfDimension(filter, 0), 1);

  if (has_bias) {
    TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kBiasTensor, &bias));
    if (data_type == kTfLiteUInt8 || data_type == kTfLiteInt8) {
      TF_LITE_ENSURE_TYPES_EQ(context, bias->type, kTfLiteInt32);
      TF_LITE_ENSURE_EQ(context, bias->params.zero_point, 0);
    } else if (data_type == kTfLiteInt16) {
      TF_LITE_ENSURE_TYPES_EQ(context, bias->type, kTfLiteInt64);
      TF_LITE_ENSURE_EQ(context, bias->params.zero_point, 0);
    } else {
      TF_LITE_ENSURE_TYPES_EQ(context, bias->type, data_type);
    }
    TF_LITE_ENSURE_EQ(context, NumDimensions(bias), 1);
    TF_LITE_ENSURE_EQ(context, SizeOfDimension(filter, 3),
                      SizeOfDimension(bias, 0));
  }

  int channels_out = SizeOfDimension(filter, 3);
  int width = SizeOfDimension(input, 2);
  int height = SizeOfDimension(input, 1);
  int filter_width = SizeOfDimension(filter, 2);
  int filter_height = SizeOfDimension(filter, 1);
  int batches = SizeOfDimension(input, 0);

  // Matching GetWindowedOutputSize in TensorFlow.
  auto padding = params->padding;
  int out_width, out_height;

  data->padding = ComputePaddingHeightWidth(
      params->stride_height, params->stride_width,
      params->dilation_height_factor, params->dilation_width_factor, height,
      width, filter_height, filter_width, padding, &out_height, &out_width);

  // Note that quantized inference requires that all tensors have their
  // parameters set. This is usually done during quantized training or
  // calibration.
  if (data_type != kTfLiteFloat32) {
    TF_LITE_ENSURE_EQ(context, filter->quantization.type,
                      kTfLiteAffineQuantization);
    TF_LITE_ENSURE(context, filter->quantization.type != kTfLiteNoQuantization);
    const auto* affine_quantization =
        reinterpret_cast<TfLiteAffineQuantization*>(
            filter->quantization.params);
    TF_LITE_ENSURE(context, affine_quantization);
    TF_LITE_ENSURE(context, affine_quantization->scale);
    TF_LITE_ENSURE(context, (affine_quantization->scale->size == 1 ||
                             affine_quantization->scale->size == channels_out));

    data->per_channel_output_multiplier.resize(channels_out);
    data->per_channel_output_shift.resize(channels_out);
    TF_LITE_ENSURE_STATUS(tflite::PopulateConvolutionQuantizationParams(
        context, input, filter, bias, output, params->activation,
        &data->output_multiplier, &data->output_shift,
        &data->output_activation_min, &data->output_activation_max,
        data->per_channel_output_multiplier.data(),
        data->per_channel_output_shift.data(), channels_out));
  }

  if (is_hybrid) {
    TF_LITE_ENSURE(context, filter->quantization.type != kTfLiteNoQuantization);
    const auto* affine_quantization =
        reinterpret_cast<TfLiteAffineQuantization*>(
            filter->quantization.params);
    TF_LITE_ENSURE(context, affine_quantization);
    TF_LITE_ENSURE(context, affine_quantization->scale);
    TF_LITE_ENSURE_EQ(
        context, affine_quantization->scale->size,
        filter->dims->data[affine_quantization->quantized_dimension]);

    int temporaries_count = 0;
    data->input_quantized_index = temporaries_count;
    if (data->input_quantized_id == kTensorNotAllocated) {
      TF_LITE_ENSURE_OK(
          context, context->AddTensors(context, 1, &data->input_quantized_id));
    }
    ++temporaries_count;
    data->scaling_factors_index = temporaries_count;
    if (data->scaling_factors_id == kTensorNotAllocated) {
      TF_LITE_ENSURE_OK(
          context, context->AddTensors(context, 1, &data->scaling_factors_id));
    }
    ++temporaries_count;
    data->input_offset_index = temporaries_count;
    if (data->input_offset_id == kTensorNotAllocated) {
      TF_LITE_ENSURE_OK(
          context, context->AddTensors(context, 1, &data->input_offset_id));
    }
    ++temporaries_count;

    TfLiteIntArrayFree(node->temporaries);
    node->temporaries = TfLiteIntArrayCreate(temporaries_count);

    node->temporaries->data[data->input_quantized_index] =
        data->input_quantized_id;
    TfLiteTensor* input_quantized;
    TF_LITE_ENSURE_OK(
        context, GetTemporarySafe(context, node, data->input_quantized_index,
                                  &input_quantized));
    input_quantized->type = kTfLiteInt8;
    input_quantized->allocation_type = kTfLiteArenaRw;
    if (!TfLiteIntArrayEqual(input_quantized->dims, input->dims)) {
      TfLiteIntArray* input_quantized_size = TfLiteIntArrayCopy(input->dims);
      TF_LITE_ENSURE_OK(context, context->ResizeTensor(context, input_quantized,
                                                       input_quantized_size));
    }
    node->temporaries->data[data->scaling_factors_index] =
        data->scaling_factors_id;
    TfLiteTensor* scaling_factors;
    TF_LITE_ENSURE_OK(
        context, GetTemporarySafe(context, node, data->scaling_factors_index,
                                  &scaling_factors));
    scaling_factors->type = kTfLiteFloat32;
    scaling_factors->allocation_type = kTfLiteArenaRw;
    const int batch_size = SizeOfDimension(input, 0);
    int scaling_dims[1] = {batch_size};
    if (!TfLiteIntArrayEqualsArray(scaling_factors->dims, 1, scaling_dims)) {
      TfLiteIntArray* scaling_factors_size = TfLiteIntArrayCreate(1);
      scaling_factors_size->data[0] = batch_size;
      TF_LITE_ENSURE_OK(context, context->ResizeTensor(context, scaling_factors,
                                                       scaling_factors_size));
    }
    node->temporaries->data[data->input_offset_index] = data->input_offset_id;
    TfLiteTensor* input_offsets;
    TF_LITE_ENSURE_OK(context,
                      GetTemporarySafe(context, node, data->input_offset_index,
                                       &input_offsets));
    input_offsets->type = kTfLiteInt32;
    input_offsets->allocation_type = kTfLiteArenaRw;
    if (!TfLiteIntArrayEqualsArray(input_offsets->dims, 1, scaling_dims)) {
      TfLiteIntArray* input_offsets_size = TfLiteIntArrayCreate(1);
      input_offsets_size->data[0] = batch_size;
      TF_LITE_ENSURE_OK(context, context->ResizeTensor(context, input_offsets,
                                                       input_offsets_size));
    }
  }

  TfLiteIntArray* outputSize = TfLiteIntArrayCreate(4);
  outputSize->data[0] = batches;
  outputSize->data[1] = out_height;
  outputSize->data[2] = out_width;
  outputSize->data[3] = channels_out;
  return context->ResizeTensor(context, output, outputSize);
}

TfLiteStatus ComputeDepthMultiplier(TfLiteContext* context,
                                    const TfLiteTensor* input,
                                    const TfLiteTensor* filter,
                                    int16* depth_multiplier) {
  int num_filter_channels = SizeOfDimension(filter, 3);
  int num_input_channels = SizeOfDimension(input, 3);
  TF_LITE_ENSURE(context, num_input_channels != 0);
  TF_LITE_ENSURE_EQ(context, num_filter_channels % num_input_channels, 0);
  *depth_multiplier = num_filter_channels / num_input_channels;
  return kTfLiteOk;
}

template <KernelType kernel_type>
TfLiteStatus EvalFloat(TfLiteContext* context, TfLiteNode* node,
                       TfLiteDepthwiseConvParams* params, OpData* data,
                       const TfLiteTensor* input, const TfLiteTensor* filter,
                       const TfLiteTensor* bias, TfLiteTensor* output) {
  float output_activation_min, output_activation_max;
  CalculateActivationRange(params->activation, &output_activation_min,
                           &output_activation_max);

  DepthwiseParams op_params;
  op_params.padding_type = PaddingType::kSame;
  op_params.padding_values.width = data->padding.width;
  op_params.padding_values.height = data->padding.height;
  op_params.stride_width = params->stride_width;
  op_params.stride_height = params->stride_height;
  op_params.dilation_width_factor = params->dilation_width_factor;
  op_params.dilation_height_factor = params->dilation_height_factor;
  op_params.float_activation_min = output_activation_min;
  op_params.float_activation_max = output_activation_max;
  TF_LITE_ENSURE_STATUS(ComputeDepthMultiplier(context, input, filter,
                                               &op_params.depth_multiplier));
  if (kernel_type == kReference) {
    reference_ops::DepthwiseConv(
        op_params, GetTensorShape(input), GetTensorData<float>(input),
        GetTensorShape(filter), GetTensorData<float>(filter),
        GetTensorShape(bias), GetTensorData<float>(bias),
        GetTensorShape(output), GetTensorData<float>(output));
  } else {
    optimized_ops::DepthwiseConv<float, float>(
        op_params, GetTensorShape(input), GetTensorData<float>(input),
        GetTensorShape(filter), GetTensorData<float>(filter),
        GetTensorShape(bias), GetTensorData<float>(bias),
        GetTensorShape(output), GetTensorData<float>(output),
        CpuBackendContext::GetFromContext(context));
  }
  return kTfLiteOk;
}

template <KernelType kernel_type>
TfLiteStatus EvalQuantized(TfLiteContext* context, TfLiteNode* node,
                           TfLiteDepthwiseConvParams* params, OpData* data,
                           const TfLiteTensor* input,
                           const TfLiteTensor* filter, const TfLiteTensor* bias,
                           TfLiteTensor* output) {
  auto input_offset = -input->params.zero_point;
  auto filter_offset = -filter->params.zero_point;
  auto output_offset = output->params.zero_point;

  DepthwiseParams op_params;
  op_params.padding_type = PaddingType::kSame;
  op_params.padding_values.width = data->padding.width;
  op_params.padding_values.height = data->padding.height;
  op_params.stride_width = params->stride_width;
  op_params.stride_height = params->stride_height;
  op_params.dilation_width_factor = params->dilation_width_factor;
  op_params.dilation_height_factor = params->dilation_height_factor;
  op_params.input_offset = input_offset;
  op_params.weights_offset = filter_offset;
  op_params.output_offset = output_offset;
  op_params.output_multiplier = data->output_multiplier;
  op_params.output_shift = -data->output_shift;
  op_params.quantized_activation_min = data->output_activation_min;
  op_params.quantized_activation_max = data->output_activation_max;
  TF_LITE_ENSURE_STATUS(ComputeDepthMultiplier(context, input, filter,
                                               &op_params.depth_multiplier));
  if (kernel_type == kReference) {
    reference_ops::DepthwiseConv(
        op_params, GetTensorShape(input), GetTensorData<uint8_t>(input),
        GetTensorShape(filter), GetTensorData<uint8_t>(filter),
        GetTensorShape(bias), GetTensorData<int32_t>(bias),
        GetTensorShape(output), GetTensorData<uint8_t>(output));
  } else {
    optimized_ops::DepthwiseConv<uint8, int32>(
        op_params, GetTensorShape(input), GetTensorData<uint8_t>(input),
        GetTensorShape(filter), GetTensorData<uint8_t>(filter),
        GetTensorShape(bias), GetTensorData<int32_t>(bias),
        GetTensorShape(output), GetTensorData<uint8_t>(output),
        CpuBackendContext::GetFromContext(context));
  }
  return kTfLiteOk;
}

template <KernelType kernel_type>
TfLiteStatus EvalQuantizedPerChannel(TfLiteContext* context, TfLiteNode* node,
                                     TfLiteDepthwiseConvParams* params,
                                     OpData* data, const TfLiteTensor* input,
                                     const TfLiteTensor* filter,
                                     const TfLiteTensor* bias,
                                     TfLiteTensor* output) {
  DepthwiseParams op_params;
  op_params.padding_type = PaddingType::kSame;
  op_params.padding_values.width = data->padding.width;
  op_params.padding_values.height = data->padding.height;
  op_params.stride_width = params->stride_width;
  op_params.stride_height = params->stride_height;
  op_params.dilation_width_factor = params->dilation_width_factor;
  op_params.dilation_height_factor = params->dilation_height_factor;
  op_params.input_offset = -input->params.zero_point;
  op_params.weights_offset = 0;
  op_params.output_offset = output->params.zero_point;
  op_params.quantized_activation_min = data->output_activation_min;
  op_params.quantized_activation_max = data->output_activation_max;
  TF_LITE_ENSURE_STATUS(ComputeDepthMultiplier(context, input, filter,
                                                &op_params.depth_multiplier));

  if (kernel_type == kReference) {
    reference_integer_ops::DepthwiseConvPerChannel(
        op_params, data->per_channel_output_multiplier.data(),
        data->per_channel_output_shift.data(), GetTensorShape(input),
        GetTensorData<int8>(input), GetTensorShape(filter),
        GetTensorData<int8>(filter), GetTensorShape(bias),
        GetTensorData<int32>(bias), GetTensorShape(output),
        GetTensorData<int8>(output));
  } else {
    optimized_integer_ops::DepthwiseConvPerChannel(
        op_params, data->per_channel_output_multiplier.data(),
        data->per_channel_output_shift.data(), GetTensorShape(input),
        GetTensorData<int8>(input), GetTensorShape(filter),
        GetTensorData<int8>(filter), GetTensorShape(bias),
        GetTensorData<int32>(bias), GetTensorShape(output),
        GetTensorData<int8>(output),
        CpuBackendContext::GetFromContext(context));
  }
  return kTfLiteOk;
}

```

```
TfLiteStatus EvalQuantizedPerChannel16x8(
    const TfLiteDepthwiseConvParams* params, const OpData* data,
    const TfLiteTensor* input, const TfLiteTensor* filter,
    const TfLiteTensor* bias, TfLiteTensor* output) {
  DepthwiseParams op_params;
  op_params.padding_type = PaddingType::kSame;
  op_params.padding_values.width = data->padding.width;
  op_params.padding_values.height = data->padding.height;
  op_params.stride_width = params->stride_width;
  op_params.stride_height = params->stride_height;
  op_params.dilation_width_factor = params->dilation_width_factor;
  op_params.dilation_height_factor = params->dilation_height_factor;
  op_params.depth_multiplier = params->depth_multiplier;
  op_params.weights_offset = 0;
  op_params.quantized_activation_min = data->output_activation_min;
  op_params.quantized_activation_max = data->output_activation_max;

  reference_integer_ops::DepthwiseConvPerChannel(
      op_params, data->per_channel_output_multiplier.data(),
      data->per_channel_output_shift.data(), GetTensorShape(input),
      GetTensorData<int16>(input), GetTensorShape(filter),
      GetTensorData<int8>(filter), GetTensorShape(bias),
      GetTensorData<std::int64_t>(bias), GetTensorShape(output),
      GetTensorData<int16>(output));

  return kTfLiteOk;
}

template <KernelType kernel_type>
TfLiteStatus EvalHybridPerChannel(TfLiteContext* context, TfLiteNode* node,
                                  TfLiteDepthwiseConvParams* params,
                                  OpData* data, const TfLiteTensor* input,
                                  const TfLiteTensor* filter,
                                  const TfLiteTensor* bias,
                                  TfLiteTensor* output) {
  float output_activation_min, output_activation_max;
  CalculateActivationRange(params->activation, &output_activation_min,
                           &output_activation_max);
  const int batch_size = SizeOfDimension(input, 0);
  TF_LITE_ENSURE(context, batch_size != 0);
  const int input_size = NumElements(input) / batch_size;
  TfLiteTensor* input_quantized;
  TF_LITE_ENSURE_OK(context,
                    GetTemporarySafe(context, node, data->input_quantized_index,
                                     &input_quantized));
  int8_t* quantized_input_ptr_batch = input_quantized->data.int8;
  TfLiteTensor* scaling_factors_tensor;
  TF_LITE_ENSURE_OK(context,
                    GetTemporarySafe(context, node, data->scaling_factors_index,
                                             &scaling_factors_tensor));
  float* scaling_factors_ptr = GetTensorData<float>(scaling_factors_tensor);
  TfLiteTensor* input_offset_tensor;
  TF_LITE_ENSURE_OK(context,
                    GetTemporarySafe(context, node, data->input_offset_index,
                                     &input_offset_tensor));
  int32_t* input_offset_ptr = GetTensorData<int32_t>(input_offset_tensor);

  for (int b = 0; b < batch_size; ++b) {
    const int offset = b * input_size;
    tensor_utils::AsymmetricQuantizeFloats(
        GetTensorData<float>(input) + offset, input_size,
        quantized_input_ptr_batch + offset, &scaling_factors_ptr[b],
        &input_offset_ptr[b]);
  }

  DepthwiseParams op_params;
  op_params.padding_type = PaddingType::kSame;
  op_params.padding_values.width = data->padding.width;
  op_params.padding_values.height = data->padding.height;
  op_params.stride_width = params->stride_width;
  op_params.stride_height = params->stride_height;
  op_params.dilation_width_factor = params->dilation_width_factor;
  op_params.dilation_height_factor = params->dilation_height_factor;
  op_params.depth_multiplier = params->depth_multiplier;

  op_params.weights_offset = 0;
  op_params.float_activation_min = output_activation_min;
  op_params.float_activation_max = output_activation_max;
  TF_LITE_ENSURE(context, filter->quantization.type != kTfLiteNoQuantization);
  const auto* affine_quantization =
      reinterpret_cast<TfLiteAffineQuantization*>(filter->quantization.params);
  if (kernel_type == kReference) {
    reference_integer_ops::DepthwiseConvHybridPerChannel(
        op_params, scaling_factors_ptr, GetTensorShape(input),
        quantized_input_ptr_batch, GetTensorShape(filter),
        GetTensorData<int8>(filter), GetTensorShape(bias),
        GetTensorData<float>(bias), GetTensorShape(output),
        GetTensorData<float>(output), affine_quantization->scale->data,
        input_offset_ptr);
  } else {
    optimized_integer_ops::DepthwiseConvHybridPerChannel(
        op_params, scaling_factors_ptr, GetTensorShape(input),
        quantized_input_ptr_batch, GetTensorShape(filter),
        GetTensorData<int8>(filter), GetTensorShape(bias),
        GetTensorData<float>(bias), GetTensorShape(output),
        GetTensorData<float>(output), affine_quantization->scale->data,
        input_offset_ptr, CpuBackendContext::GetFromContext(context));
  }

  return kTfLiteOk;
}

template <KernelType kernel_type, TfLiteType input_type>
TfLiteStatus EvalImpl(TfLiteContext* context, TfLiteNode* node) {
  auto* params =
      reinterpret_cast<TfLiteDepthwiseConvParams*>(node->builtin_data);
  OpData* data = reinterpret_cast<OpData*>(node->user_data);

  TfLiteTensor* output;
  TF_LITE_ENSURE_OK(context,
                    GetOutputSafe(context, node, kOutputTensor, &output));
  const TfLiteTensor* input;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kInputTensor, &input));
  const TfLiteTensor* filter;
  TF_LITE_ENSURE_OK(context,
                    GetInputSafe(context, node, kFilterTensor, &filter));
  const TfLiteTensor* bias =
      (NumInputs(node) == 3) ? GetInput(context, node, kBiasTensor) : nullptr;
  TFLITE_DCHECK_EQ(input_type, input->type);

  switch (input_type) {  // Already know in/out types are same.
    case kTfLiteFloat32:
      if (filter->type == kTfLiteFloat32) {
        return EvalFloat<kernel_type>(context, node, params, data, input,
                                      filter, bias, output);
      } else if (filter->type == kTfLiteInt8) {
        return EvalHybridPerChannel<kernel_type>(context, node, params, data,
                                                 input, filter, bias, output);
      } else {
        TF_LITE_KERNEL_LOG(
            context, "Type %s with filter type %s not currently supported.",
            TfLiteTypeGetName(input->type), TfLiteTypeGetName(filter->type));
        return kTfLiteError;
      }
      break;
    case kTfLiteUInt8:
      return EvalQuantized<kernel_type>(context, node, params, data, input,
                                        filter, bias, output);
      break;
    case kTfLiteInt8:
      return EvalQuantizedPerChannel<kernel_type>(context, node, params, data,
                                                  input, filter, bias, output);
      break;
    case kTfLiteInt16:
      return EvalQuantizedPerChannel16x8(params, data, input, filter, bias,
                                         output);
      break;
    default:
      context->ReportError(context, "Type %d not currently supported.",
                           input->type);
      return kTfLiteError;
  }
}

template <KernelType kernel_type>
TfLiteStatus Eval(TfLiteContext* context, TfLiteNode* node) {
  const TfLiteTensor* input;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, kInputTensor, &input));

  switch (input->type) {  // Already know in/out types are same.
    case kTfLiteFloat32:
      return EvalImpl<kernel_type, kTfLiteFloat32>(context, node);
    case kTfLiteUInt8:
      return EvalImpl<kernel_type, kTfLiteUInt8>(context, node);
    case kTfLiteInt8:
      return EvalImpl<kernel_type, kTfLiteInt8>(context, node);
    case kTfLiteInt16:
      return EvalImpl<kernel_type, kTfLiteInt16>(context, node);
    default:
      context->ReportError(context, "Type %d not currently supported.",
                           input->type);
      return kTfLiteError;
  }
}

}  // namespace depthwise_conv

TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_REF() {
  static TfLiteRegistration r = {
      depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
      depthwise_conv::Eval<depthwise_conv::kReference>};
  return &r;
}

TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_GENERIC_OPT() {
  static TfLiteRegistration r = {
      depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
      depthwise_conv::Eval<depthwise_conv::kGenericOptimized>};
  return &r;
}

TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_NEON_OPT() {
  static TfLiteRegistration r = {
      depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
      depthwise_conv::Eval<depthwise_conv::kNeonOptimized>};
  return &r;
```

```
}

TfLiteRegistration* Register_DEPTHWISE_CONVOLUTION_NEON_OPT_UINT8() {
  static TfLiteRegistration r = {
      depthwise_conv::Init, depthwise_conv::Free, depthwise_conv::Prepare,
      depthwise_conv::EvalImpl<depthwise_conv::kNeonOptimized, kTfLiteUInt8>};
  return &r;
}

TfLiteRegistration* Register_DEPTHWISE_CONV_2D() {
#ifdef USE_NEON
  return Register_DEPTHWISE_CONVOLUTION_NEON_OPT();
#else
  return Register_DEPTHWISE_CONVOLUTION_GENERIC_OPT();
#endif
}

// Warning: Clients using this variant are responsible for ensuring that their
// models only need the UINT8 type. TFLite's op registration mechanism doesn't
// yet allow for more nuanced registration mechanisms.
TfLiteRegistration* Register_DEPTHWISE_CONV_2D_UINT8() {
#ifdef USE_NEON
  return Register_DEPTHWISE_CONVOLUTION_NEON_OPT_UINT8();
#else
  return Register_DEPTHWISE_CONV_2D();
#endif
}

}  // namespace builtin
}  // namespace ops
}  // namespace tflite
```