tensorflow / **tensorflow** Public

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

5100e359ae

**tensorflow** / tensorflow / lite / kernels / internal / **common.h**

**tensorflower-gardener** Merge pull request #45342 from Tessil:toupstream/tabl... ... History

23 contributors +11

1083 lines (974 sloc) | 42.8 KB

```
/* Copyright 2017 The TensorFlow Authors. All Rights Reserved.

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

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

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

#ifndef ALLOW_SLOW_GENERIC_DEPTHWISECONV_FALLBACK
#ifdef GEMMLOWP_ALLOW_SLOW_SCALAR_FALLBACK
#define ALLOW_SLOW_GENERIC_DEPTHWISECONV_FALLBACK
#endif
#endif

#include <functional>

#include "fixedpoint/fixedpoint.h"
#include "tensorflow/lite/kernels/internal/cppmath.h"
#include "tensorflow/lite/kernels/internal/optimized/neon_check.h"
#include "tensorflow/lite/kernels/internal/types.h"
```

```

namespace tflite {

constexpr int kReverseShift = -1;

inline void GetActivationMinMax(FusedActivationFunctionType ac,
                                float* output_activation_min,
                                float* output_activation_max) {
  switch (ac) {
    case FusedActivationFunctionType::kNone:
      *output_activation_min = std::numeric_limits<float>::lowest();
      *output_activation_max = std::numeric_limits<float>::max();
      break;
    case FusedActivationFunctionType::kRelu:
      *output_activation_min = 0.f;
      *output_activation_max = std::numeric_limits<float>::max();
      break;
    case FusedActivationFunctionType::kRelu1:
      *output_activation_min = -1.f;
      *output_activation_max = 1.f;
      break;
    case FusedActivationFunctionType::kRelu6:
      *output_activation_min = 0.f;
      *output_activation_max = 6.f;
      break;
  }
}

template <typename T>
inline T ActivationFunctionWithMinMax(T x, T output_activation_min,
                                      T output_activation_max) {
  using std::max;
  using std::min;
  return min(max(x, output_activation_min), output_activation_max);
}

// Legacy function, left for compatibility only.
template <FusedActivationFunctionType Ac>
float ActivationFunction(float x) {
  float output_activation_min, output_activation_max;
  GetActivationMinMax(Ac, &output_activation_min, &output_activation_max);
  return ActivationFunctionWithMinMax(x, output_activation_min,
                                      output_activation_max);
}

inline void BiasAndClamp(float clamp_min, float clamp_max, int bias_size,
                         const float* bias_data, int array_size,
                         float* array_data) {
  // Note: see b/132215220: in May 2019 we thought it would be OK to replace
```

```
  // this with the Eigen one-liner:
  //   return (array.colwise() + bias).cwiseMin(clamp_max).cwiseMin(clamp_max).
  // This turned out to severely regress performance: +4ms (i.e. 8%) on
  // MobileNet v2 / 1.0 / 224. So we keep custom NEON code for now.
  TFLITE_DCHECK_EQ((array_size % bias_size), 0);
#ifdef USE_NEON
  float* array_ptr = array_data;
  float* array_end_ptr = array_ptr + array_size;
  const auto clamp_min_vec = vdupq_n_f32(clamp_min);
  const auto clamp_max_vec = vdupq_n_f32(clamp_max);
  for (; array_ptr != array_end_ptr; array_ptr += bias_size) {
    int i = 0;
    for (; i <= bias_size - 16; i += 16) {
      auto b0 = vld1q_f32(bias_data + i);
      auto b1 = vld1q_f32(bias_data + i + 4);
      auto b2 = vld1q_f32(bias_data + i + 8);
      auto b3 = vld1q_f32(bias_data + i + 12);
      auto a0 = vld1q_f32(array_ptr + i);
      auto a1 = vld1q_f32(array_ptr + i + 4);
      auto a2 = vld1q_f32(array_ptr + i + 8);
      auto a3 = vld1q_f32(array_ptr + i + 12);
      auto x0 = vaddq_f32(a0, b0);
      auto x1 = vaddq_f32(a1, b1);
      auto x2 = vaddq_f32(a2, b2);
      auto x3 = vaddq_f32(a3, b3);
      x0 = vmaxq_f32(clamp_min_vec, x0);
      x1 = vmaxq_f32(clamp_min_vec, x1);
      x2 = vmaxq_f32(clamp_min_vec, x2);
      x3 = vmaxq_f32(clamp_min_vec, x3);
      x0 = vminq_f32(clamp_max_vec, x0);
      x1 = vminq_f32(clamp_max_vec, x1);
      x2 = vminq_f32(clamp_max_vec, x2);
      x3 = vminq_f32(clamp_max_vec, x3);
      vst1q_f32(array_ptr + i, x0);
      vst1q_f32(array_ptr + i + 4, x1);
      vst1q_f32(array_ptr + i + 8, x2);
      vst1q_f32(array_ptr + i + 12, x3);
    }
    for (; i <= bias_size - 4; i += 4) {
      auto b = vld1q_f32(bias_data + i);
      auto a = vld1q_f32(array_ptr + i);
      auto x = vaddq_f32(a, b);
      x = vmaxq_f32(clamp_min_vec, x);
      x = vminq_f32(clamp_max_vec, x);
      vst1q_f32(array_ptr + i, x);
    }
    for (; i < bias_size; i++) {
      array_ptr[i] = ActivationFunctionWithMinMax(array_ptr[i] + bias_data[i],
                                                  clamp_min, clamp_max);
```

```
    }
  }
#else  // not NEON
  for (int array_offset = 0; array_offset < array_size;
       array_offset += bias_size) {
    for (int i = 0; i < bias_size; i++) {
      array_data[array_offset + i] = ActivationFunctionWithMinMax(
          array_data[array_offset + i] + bias_data[i], clamp_min, clamp_max);
    }
  }
#endif
}

inline int32_t MultiplyByQuantizedMultiplierSmallerThanOneExp(
    int32_t x, int32_t quantized_multiplier, int left_shift) {
  using gemmlowp::RoundingDivideByPOT;
  using gemmlowp::SaturatingRoundingDoublingHighMul;
  return RoundingDivideByPOT(
      SaturatingRoundingDoublingHighMul(x, quantized_multiplier), -left_shift);
}

inline int32_t MultiplyByQuantizedMultiplierGreaterThanOne(
    int32_t x, int32_t quantized_multiplier, int left_shift) {
  using gemmlowp::SaturatingRoundingDoublingHighMul;
  return SaturatingRoundingDoublingHighMul(x * (1 << left_shift),
                                           quantized_multiplier);
}

inline int32_t MultiplyByQuantizedMultiplier(int32_t x,
                                             int32_t quantized_multiplier,
                                             int shift) {
  using gemmlowp::RoundingDivideByPOT;
  using gemmlowp::SaturatingRoundingDoublingHighMul;
  int left_shift = shift > 0 ? shift : 0;
  int right_shift = shift > 0 ? 0 : -shift;
  return RoundingDivideByPOT(SaturatingRoundingDoublingHighMul(
                                 x * (1 << left_shift), quantized_multiplier),
                             right_shift);
}

inline int32_t MultiplyByQuantizedMultiplier(int64_t x,
                                             int32_t quantized_multiplier,
                                             int shift) {
  // Inputs:
  // - quantized_multiplier has fixed point at bit 31
  // - shift is -31 to +7 (negative for right shift)
  //
  // Assumptions: The following input ranges are assumed
  // - quantize_scale>=0  (the usual range is (1<<30) to (1>>31)-1)
```

```
  // - scaling is chosen so final scaled result fits in int32_t
  // - input x is in the range -(1<<47) <= x < (1<<47)
  assert(quantized_multiplier >= 0);
  assert(shift >= -31 && shift < 8);
  assert(x >= -(static_cast<int64_t>(1) << 47) &&
         x < (static_cast<int64_t>(1) << 47));

  int32_t reduced_multiplier = (quantized_multiplier < 0x7FFF0000)
                                   ? ((quantized_multiplier + (1 << 15)) >> 16)
                                   : 0x7FFF;
  int total_shift = 15 - shift;
  x = (x * (int64_t)reduced_multiplier) + ((int64_t)1 << (total_shift - 1));
  int32_t result = x >> total_shift;
  return result;
}

#ifdef USE_NEON
// Round uses ARM's rounding shift right.
inline int32x4x4_t MultiplyByQuantizedMultiplier4Rows(
    int32x4x4_t input_val, int32_t quantized_multiplier, int shift) {
  const int left_shift = std::max(shift, 0);
  const int right_shift = std::min(shift, 0);
  int32x4x4_t result;

  int32x4_t multiplier_dup = vdupq_n_s32(quantized_multiplier);
  int32x4_t left_shift_dup = vdupq_n_s32(left_shift);
  int32x4_t right_shift_dup = vdupq_n_s32(right_shift);

  result.val[0] =
      vrshlq_s32(vqrdmulhq_s32(vshlq_s32(input_val.val[0], left_shift_dup),
                               multiplier_dup),
                 right_shift_dup);

  result.val[1] =
      vrshlq_s32(vqrdmulhq_s32(vshlq_s32(input_val.val[1], left_shift_dup),
                               multiplier_dup),
                 right_shift_dup);

  result.val[2] =
      vrshlq_s32(vqrdmulhq_s32(vshlq_s32(input_val.val[2], left_shift_dup),
                               multiplier_dup),
                 right_shift_dup);

  result.val[3] =
      vrshlq_s32(vqrdmulhq_s32(vshlq_s32(input_val.val[3], left_shift_dup),
                               multiplier_dup),
                 right_shift_dup);

  return result;
```

```
}
#endif

template <typename T>
int CountLeadingZeros(T integer_input) {
  static_assert(std::is_unsigned<T>::value,
                "Only unsigned integer types handled.");
#if defined(__GNUC__)
  return integer_input ? __builtin_clz(integer_input)
                       : std::numeric_limits<T>::digits;
#else
  if (integer_input == 0) {
    return std::numeric_limits<T>::digits;
  }

  const T one_in_leading_positive = static_cast<T>(1)
                                    << (std::numeric_limits<T>::digits - 1);
  int leading_zeros = 0;
  while (integer_input < one_in_leading_positive) {
    integer_input <<= 1;
    ++leading_zeros;
  }
  return leading_zeros;
#endif
}

template <typename T>
inline int CountLeadingSignBits(T integer_input) {
  static_assert(std::is_signed<T>::value, "Only signed integer types handled.");
#if defined(__GNUC__) && !defined(__clang__)
  return integer_input ? __builtin_clrsb(integer_input)
                       : std::numeric_limits<T>::digits;
#else
  using U = typename std::make_unsigned<T>::type;
  return integer_input >= 0
             ? CountLeadingZeros(static_cast<U>(integer_input)) - 1
         : integer_input != std::numeric_limits<T>::min()
             ? CountLeadingZeros(2 * static_cast<U>(-integer_input) - 1)
             : 0;
#endif
}

// Use "count leading zeros" helper functions to do a fast Floor(log_2(x)).
template <typename Integer>
inline Integer FloorLog2(Integer n) {
  static_assert(std::is_integral<Integer>::value, "");
  static_assert(std::is_signed<Integer>::value, "");
  static_assert(sizeof(Integer) == 4 || sizeof(Integer) == 8, "");
  TFLITE_CHECK_GT(n, 0);
```

```
  if (sizeof(Integer) == 4) {
    return 30 - CountLeadingSignBits(n);
  } else {
    return 62 - CountLeadingSignBits(n);
  }
}

// The size of the LUT depends on the type of input. For int8 inputs a simple
// 256 entries LUT is used. For int16 inputs the high 9 bits are used for
// indexing and the 7 remaining bits are used for interpolation. We thus use a
// 513-entries LUT for int16 cases, 512 for the 9-bit indexing and 1 extra entry
// to interpolate the last value.
template <typename LutInT>
constexpr int lut_size() {
  static_assert(std::is_same<LutInT, int8_t>::value ||
                    std::is_same<LutInT, int16_t>::value,
                "Only LUTs with int8 or int16 inputs are supported.");
  return std::is_same<LutInT, int8_t>::value ? 256 : 513;
}

// Generate a LUT for 'func' which can be used to approximate functions like
// exp, log, ...
//
// - func: the function to build the LUT for (e.g exp(x))
// - input_min, input_max: range of the func inputs
// - output_min, output_max: range of the func outputs
// - lut: pointer to the LUT table to fill, the table must be of size
// lut_size<LutInT>()
template <typename FloatT, typename LutInT, typename LutOutT>
inline void gen_lut(FloatT (*func)(FloatT), FloatT input_min, FloatT input_max,
                    FloatT output_min, FloatT output_max, LutOutT* lut) {
  static_assert(std::is_same<LutInT, int8_t>::value ||
                    std::is_same<LutInT, int16_t>::value,
                "Only LUTs with int8 or int16 inputs are supported.");
  static_assert(std::is_same<LutOutT, int8_t>::value ||
                    std::is_same<LutOutT, int16_t>::value,
                "Only LUTs with int8 or int16 outputs are supported.");
  static_assert(std::is_floating_point<FloatT>::value,
                "FloatT must be a floating-point type.");

  const int nb_steps = std::is_same<LutInT, int8_t>::value ? 256 : 512;
  const FloatT step = (input_max - input_min) / nb_steps;
  const FloatT half_step = step / 2;
  const FloatT output_scaling_inv =
      static_cast<FloatT>(std::numeric_limits<LutOutT>::max() -
                          std::numeric_limits<LutOutT>::min() + 1) /
      (output_max - output_min);
  const FloatT table_min =
      static_cast<FloatT>(std::numeric_limits<LutOutT>::min());
```

```
  const FloatT table_max =
      static_cast<FloatT>(std::numeric_limits<LutOutT>::max());

  for (int i = 0; i < nb_steps; i++) {
    const FloatT val = func(input_min + i * step);
    const FloatT val_midpoint = func(input_min + i * step + half_step);
    const FloatT val_next = func(input_min + (i + 1) * step);

    const FloatT sample_val = TfLiteRound(val * output_scaling_inv);
    const FloatT midpoint_interp_val =
        TfLiteRound((val_next * output_scaling_inv +
                     TfLiteRound(val * output_scaling_inv)) /
                    2);
    const FloatT midpoint_val = TfLiteRound(val_midpoint * output_scaling_inv);
    const FloatT midpoint_err = midpoint_interp_val - midpoint_val;
    const FloatT bias = TfLiteRound(midpoint_err / 2);

    lut[i] = static_cast<LutOutT>(std::min<FloatT>(
        std::max<FloatT>(sample_val - bias, table_min), table_max));
  }

  const bool with_extra_interpolation_value =
      std::is_same<LutInT, int16_t>::value;
  if (with_extra_interpolation_value) {
    lut[nb_steps] = static_cast<LutOutT>(std::min<FloatT>(
        std::max<FloatT>(TfLiteRound(func(input_max) * output_scaling_inv),
                         table_min),
        table_max));
  }
}

// LUT must have 513 values
template <typename LutOutT>
inline LutOutT lut_lookup_with_interpolation(int16_t value,
                                             const LutOutT* lut) {
  static_assert(std::is_same<LutOutT, int8_t>::value ||
                    std::is_same<LutOutT, int16_t>::value,
                "Only LUTs with int8 or int16 outputs are supported.");
  // 512 base values, lut[513] is only used to calculate the slope
  const uint16_t index = static_cast<uint16_t>(256 + (value >> 7));
  assert(index < 512 && "LUT index out of range.");
  const int16_t offset = value & 0x7f;

  // Base and slope are Q0.x
  const LutOutT base = lut[index];
  const LutOutT slope = lut[index + 1] - lut[index];

  // Q0.x * Q0.7 = Q0.(x + 7)
  // Round and convert from Q0.(x + 7) to Q0.x
```

```
  const int delta = (slope * offset + 64) >> 7;

  // Q0.15 + Q0.15
  return static_cast<LutOutT>(base + delta);
}

// int16_t -> int16_t table lookup with interpolation
// LUT must have 513 values
inline int16_t lut_lookup(int16_t value, const int16_t* lut) {
  return lut_lookup_with_interpolation(value, lut);
}

// int16_t -> int8_t table lookup with interpolation
// LUT must have 513 values
inline int8_t lut_lookup(int16_t value, const int8_t* lut) {
  return lut_lookup_with_interpolation(value, lut);
}

// int8_t -> int8_t table lookup without interpolation
// LUT must have 256 values
inline int8_t lut_lookup(int8_t value, const int8_t* lut) {
  return lut[128 + value];
}

// int8_t -> int16_t table lookup without interpolation
// LUT must have 256 values
inline int16_t lut_lookup(int8_t value, const int16_t* lut) {
  return lut[128 + value];
}

// Table of sigmoid(i/24) at 0.16 format - 256 elements.

// We use combined sigmoid and tanh look-up table, since
// tanh(x) = 2*sigmoid(2*x) -1.
// Both functions are symmetric, so the LUT table is only needed
// for the absolute value of the input.
static const uint16_t sigmoid_table_uint16[256] = {
    32768, 33451, 34133, 34813, 35493, 36169, 36843, 37513, 38180, 38841, 39498,
    40149, 40794, 41432, 42064, 42688, 43304, 43912, 44511, 45102, 45683, 46255,
    46817, 47369, 47911, 48443, 48964, 49475, 49975, 50464, 50942, 51409, 51865,
    52311, 52745, 53169, 53581, 53983, 54374, 54755, 55125, 55485, 55834, 56174,
    56503, 56823, 57133, 57433, 57724, 58007, 58280, 58544, 58800, 59048, 59288,
    59519, 59743, 59959, 60168, 60370, 60565, 60753, 60935, 61110, 61279, 61441,
    61599, 61750, 61896, 62036, 62172, 62302, 62428, 62549, 62666, 62778, 62886,
    62990, 63090, 63186, 63279, 63368, 63454, 63536, 63615, 63691, 63765, 63835,
    63903, 63968, 64030, 64090, 64148, 64204, 64257, 64308, 64357, 64405, 64450,
    64494, 64536, 64576, 64614, 64652, 64687, 64721, 64754, 64786, 64816, 64845,
    64873, 64900, 64926, 64950, 64974, 64997, 65019, 65039, 65060, 65079, 65097,
    65115, 65132, 65149, 65164, 65179, 65194, 65208, 65221, 65234, 65246, 65258,
```

```
    65269, 65280, 65291, 65301, 65310, 65319, 65328, 65337, 65345, 65352, 65360,
    65367, 65374, 65381, 65387, 65393, 65399, 65404, 65410, 65415, 65420, 65425,
    65429, 65433, 65438, 65442, 65445, 65449, 65453, 65456, 65459, 65462, 65465,
    65468, 65471, 65474, 65476, 65479, 65481, 65483, 65485, 65488, 65489, 65491,
    65493, 65495, 65497, 65498, 65500, 65501, 65503, 65504, 65505, 65507, 65508,
    65509, 65510, 65511, 65512, 65513, 65514, 65515, 65516, 65517, 65517, 65518,
    65519, 65520, 65520, 65521, 65522, 65522, 65523, 65523, 65524, 65524, 65525,
    65525, 65526, 65526, 65526, 65527, 65527, 65528, 65528, 65528, 65529, 65529,
    65529, 65529, 65530, 65530, 65530, 65530, 65531, 65531, 65531, 65531, 65531,
    65532, 65532, 65532, 65532, 65532, 65532, 65533, 65533, 65533, 65533, 65533,
    65533, 65533, 65533, 65534, 65534, 65534, 65534, 65534, 65534, 65534, 65534,
    65534, 65534, 65535};

// TODO(b/77858996): Add these to gemmlowp.
template <typename IntegerType>
IntegerType SaturatingAddNonGemmlowp(IntegerType a, IntegerType b) {
  static_assert(std::is_same<IntegerType, void>::value, "unimplemented");
  return a;
}

template <>
inline std::int32_t SaturatingAddNonGemmlowp(std::int32_t a, std::int32_t b) {
  std::int64_t a64 = a;
  std::int64_t b64 = b;
  std::int64_t sum = a64 + b64;
  return static_cast<std::int32_t>(std::min(
      static_cast<std::int64_t>(std::numeric_limits<std::int32_t>::max()),
      std::max(
          static_cast<std::int64_t>(std::numeric_limits<std::int32_t>::min()),
          sum)));
}

template <typename tRawType, int tIntegerBits>
gemmlowp::FixedPoint<tRawType, tIntegerBits> SaturatingAddNonGemmlowp(
    gemmlowp::FixedPoint<tRawType, tIntegerBits> a,
    gemmlowp::FixedPoint<tRawType, tIntegerBits> b) {
  return gemmlowp::FixedPoint<tRawType, tIntegerBits>::FromRaw(
      SaturatingAddNonGemmlowp(a.raw(), b.raw()));
}

template <typename IntegerType>
IntegerType SaturatingSub(IntegerType a, IntegerType b) {
  static_assert(std::is_same<IntegerType, void>::value, "unimplemented");
  return a;
}

template <>
inline std::int16_t SaturatingSub(std::int16_t a, std::int16_t b) {
  std::int32_t a32 = a;
```

```
  std::int32_t b32 = b;
  std::int32_t diff = a32 - b32;
  return static_cast<std::int16_t>(
      std::min(static_cast<int32_t>(32767),
               std::max(static_cast<int32_t>(-32768), diff)));
}

template <>
inline std::int32_t SaturatingSub(std::int32_t a, std::int32_t b) {
  std::int64_t a64 = a;
  std::int64_t b64 = b;
  std::int64_t diff = a64 - b64;
  return static_cast<std::int32_t>(std::min(
      static_cast<std::int64_t>(std::numeric_limits<std::int32_t>::max()),
      std::max(
          static_cast<std::int64_t>(std::numeric_limits<std::int32_t>::min()),
          diff)));
}

template <typename tRawType, int tIntegerBits>
gemmlowp::FixedPoint<tRawType, tIntegerBits> SaturatingSub(
    gemmlowp::FixedPoint<tRawType, tIntegerBits> a,
    gemmlowp::FixedPoint<tRawType, tIntegerBits> b) {
  return gemmlowp::FixedPoint<tRawType, tIntegerBits>::FromRaw(
      SaturatingSub(a.raw(), b.raw()));
}
// End section to be moved to gemmlowp.

template <typename IntegerType>
IntegerType SaturatingRoundingMultiplyByPOTParam(IntegerType x, int exponent) {
  if (exponent == 0) {
    return x;
  }
  using ScalarIntegerType =
      typename gemmlowp::FixedPointRawTypeTraits<IntegerType>::ScalarRawType;
  const IntegerType min =
      gemmlowp::Dup<IntegerType>(std::numeric_limits<ScalarIntegerType>::min());
  const IntegerType max =
      gemmlowp::Dup<IntegerType>(std::numeric_limits<ScalarIntegerType>::max());
  const int ScalarIntegerTypeBits = 8 * sizeof(ScalarIntegerType);

  const std::int32_t threshold =
      ((1 << (ScalarIntegerTypeBits - 1 - exponent)) - 1);
  const IntegerType positive_mask =
      gemmlowp::MaskIfGreaterThan(x, gemmlowp::Dup<IntegerType>(threshold));
  const IntegerType negative_mask =
      gemmlowp::MaskIfLessThan(x, gemmlowp::Dup<IntegerType>(-threshold));

  IntegerType result = gemmlowp::ShiftLeft(x, exponent);
```

```
  result = gemmlowp::SelectUsingMask(positive_mask, max, result);
  result = gemmlowp::SelectUsingMask(negative_mask, min, result);
  return result;
}

// If we want to leave IntegerBits fixed, then multiplication
// by a power of two has to be saturating/rounding, not exact anymore.
template <typename tRawType, int tIntegerBits>
gemmlowp::FixedPoint<tRawType, tIntegerBits>
SaturatingRoundingMultiplyByPOTParam(
    gemmlowp::FixedPoint<tRawType, tIntegerBits> a, int exponent) {
  return gemmlowp::FixedPoint<tRawType, tIntegerBits>::FromRaw(
      SaturatingRoundingMultiplyByPOTParam(a.raw(), exponent));
}

// Convert int32_t multiplier to int16_t with rounding.
inline void DownScaleInt32ToInt16Multiplier(int32_t multiplier_int32_t,
                                            int16_t* multiplier_int16_t) {
  TFLITE_DCHECK_GE(multiplier_int32_t, 0);
  static constexpr int32_t kRoundingOffset = 1 << 15;
  if (multiplier_int32_t >=
      std::numeric_limits<int32_t>::max() - kRoundingOffset) {
    *multiplier_int16_t = std::numeric_limits<int16_t>::max();
    return;
  }
  const int32_t result = (multiplier_int32_t + kRoundingOffset) >> 16;
  TFLITE_DCHECK_LE(result << 16, multiplier_int32_t + kRoundingOffset);
  TFLITE_DCHECK_GT(result << 16, multiplier_int32_t - kRoundingOffset);
  *multiplier_int16_t = result;
  TFLITE_DCHECK_EQ(*multiplier_int16_t, result);
}

// Minimum output bits to accommodate log of maximum input range.  It actually
// does not matter if one considers, say, [-64,64] or [-64,64).
//
// For example, run this through Octave:
// [0:127; ...
//  ceil(log(abs( log(2.^(0:127))+1 ))/log(2)); ...
//  ceil(log(abs( log(2.^(0:127))+1 ))/log(2))]
constexpr int min_log_x_output_bits(int input_bits) {
  return input_bits > 90   ? 7
         : input_bits > 44 ? 6
         : input_bits > 21 ? 5
         : input_bits > 10 ? 4
         : input_bits > 4  ? 3
         : input_bits > 1  ? 2
                           : 1;
}

```

```
// Although currently the name of this function says that it cannot handle
// values less than 1, in practice it can handle as low as 1/x_max, where
// x_max is the largest representable input.  In other words, the output range
// is symmetric.
template <int OutputIntegerBits, int InputIntegerBits>
inline gemmlowp::FixedPoint<int32_t, OutputIntegerBits>
log_x_for_x_greater_than_or_equal_to_1_impl(
    gemmlowp::FixedPoint<int32_t, InputIntegerBits> input_val) {
  // assert(__builtin_clz(0u) >= std::numeric_limits<uint32_t>::digits - 1);
  // assert(__builtin_clz(0u) <= std::numeric_limits<uint32_t>::digits);
  using FixedPoint0 = gemmlowp::FixedPoint<int32_t, 0>;
  // The reason for accumulating the result with an extra bit of headroom is
  // that z_pow_2_adj * log_2 might be saturated, and adding num_scaled *
  // recip_denom will otherwise introduce an error.
  static constexpr int kAccumIntegerBits = OutputIntegerBits + 1;
  using FixedPointAccum = gemmlowp::FixedPoint<int32_t, kAccumIntegerBits>;

  const FixedPoint0 log_2 = GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(
      FixedPoint0, 1488522236, std::log(2.0));
  const FixedPoint0 sqrt_sqrt_half = GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(
      FixedPoint0, 1805811301, std::sqrt(std::sqrt(0.5)));
  const FixedPoint0 sqrt_half = GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(
      FixedPoint0, 1518500250, std::sqrt(0.5));
  const FixedPoint0 one_quarter =
      GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(FixedPoint0, 536870912, 1.0 / 4.0);

  const FixedPoint0 alpha_n = GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(
      FixedPoint0, 117049297, 11.0 / 240.0 * std::sqrt(std::sqrt(2.0)));
  const FixedPoint0 alpha_d = GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(
      FixedPoint0, 127690142, 1.0 / 20.0 * std::sqrt(std::sqrt(2.0)));
  const FixedPoint0 alpha_i = GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(
      FixedPoint0, 1057819769,
      2.0 / std::sqrt(std::sqrt(2.0)) - std::sqrt(std::sqrt(2.0)));
  const FixedPoint0 alpha_f = GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(
      FixedPoint0, 638450708, 1.0 / 4.0 * std::sqrt(std::sqrt(2.0)));

  const FixedPointAccum shifted_quarter =
      gemmlowp::Rescale<kAccumIntegerBits>(one_quarter);

  // Reinterpret the input value as Q0.31, because we will figure out the
  // required shift "ourselves" instead of using, say, Rescale.
  FixedPoint0 z_a = FixedPoint0::FromRaw(input_val.raw());
  // z_a_pow_2 = input_integer_bits - z_a_headroom;
  int z_a_headroom_plus_1 = CountLeadingZeros(static_cast<uint32_t>(z_a.raw()));
  FixedPoint0 r_a_tmp =
      SaturatingRoundingMultiplyByPOTParam(z_a, (z_a_headroom_plus_1 - 1));
  const int32_t r_a_raw =
      SaturatingRoundingMultiplyByPOTParam((r_a_tmp * sqrt_half).raw(), 1);
  // z_pow_2_adj = max(z_pow_2_a - 0.75, z_pow_2_b - 0.25);
```

```
  // z_pow_2_adj = max(InputIntegerBits - z_a_headroom_plus_1 + 0.25,
  //                   InputIntegerBits - z_b_headroom - 0.25);
  const FixedPointAccum z_a_pow_2_adj = SaturatingAddNonGemmlowp(
      FixedPointAccum::FromRaw(SaturatingRoundingMultiplyByPOTParam(
          static_cast<int32_t>(InputIntegerBits - z_a_headroom_plus_1),
          31 - kAccumIntegerBits)),
      shifted_quarter);

  // z_b is treated like z_a, but premultiplying by sqrt(0.5).
  FixedPoint0 z_b = z_a * sqrt_half;
  int z_b_headroom = CountLeadingZeros(static_cast<uint32_t>(z_b.raw())) - 1;
  const int32_t r_b_raw =
      SaturatingRoundingMultiplyByPOTParam(z_a.raw(), z_b_headroom);
  const FixedPointAccum z_b_pow_2_adj = SaturatingSub(
      FixedPointAccum::FromRaw(SaturatingRoundingMultiplyByPOTParam(
          static_cast<int32_t>(InputIntegerBits - z_b_headroom),
          31 - kAccumIntegerBits)),
      shifted_quarter);

  const FixedPoint0 r = FixedPoint0::FromRaw(std::min(r_a_raw, r_b_raw));
  const FixedPointAccum z_pow_2_adj = FixedPointAccum::FromRaw(
      std::max(z_a_pow_2_adj.raw(), z_b_pow_2_adj.raw()));

  const FixedPoint0 p = gemmlowp::RoundingHalfSum(r, sqrt_sqrt_half);
  FixedPoint0 q = r - sqrt_sqrt_half;
  q = q + q;

  const FixedPoint0 common_sq = q * q;
  const FixedPoint0 num = q * r + q * common_sq * alpha_n;
  const FixedPoint0 denom_minus_one_0 =
      p * (alpha_i + q + alpha_d * common_sq) + alpha_f * q;
  const FixedPoint0 recip_denom =
      one_over_one_plus_x_for_x_in_0_1(denom_minus_one_0);

  const FixedPointAccum num_scaled = gemmlowp::Rescale<kAccumIntegerBits>(num);
  return gemmlowp::Rescale<OutputIntegerBits>(z_pow_2_adj * log_2 +
                                              num_scaled * recip_denom);
}

template <int OutputIntegerBits, int InputIntegerBits>
inline gemmlowp::FixedPoint<int32_t, OutputIntegerBits>
log_x_for_x_greater_than_or_equal_to_1(
    gemmlowp::FixedPoint<int32_t, InputIntegerBits> input_val) {
  static_assert(
      OutputIntegerBits >= min_log_x_output_bits(InputIntegerBits),
      "Output integer bits must be sufficient to accommodate logs of inputs.");
  return log_x_for_x_greater_than_or_equal_to_1_impl<OutputIntegerBits,
                                                     InputIntegerBits>(
      input_val);
```

```
}

inline int32_t GetReciprocal(int32_t x, int x_integer_digits,
                             int* num_bits_over_unit) {
  int headroom_plus_one = CountLeadingZeros(static_cast<uint32_t>(x));
  // This is the number of bits to the left of the binary point above 1.0.
  // Consider x=1.25.  In that case shifted_scale=0.8 and
  // no later adjustment will be needed.
  *num_bits_over_unit = x_integer_digits - headroom_plus_one;
  const int32_t shifted_sum_minus_one =
      static_cast<int32_t>((static_cast<uint32_t>(x) << headroom_plus_one) -
                           (static_cast<uint32_t>(1) << 31));

  gemmlowp::FixedPoint<int32_t, 0> shifted_scale =
      gemmlowp::one_over_one_plus_x_for_x_in_0_1(
          gemmlowp::FixedPoint<int32_t, 0>::FromRaw(shifted_sum_minus_one));
  return shifted_scale.raw();
}

inline void GetInvSqrtQuantizedMultiplierExp(int32_t input, int reverse_shift,
                                             int32_t* output_inv_sqrt,
                                             int* output_shift) {
  TFLITE_DCHECK_GE(input, 0);
  if (input <= 1) {
    // Handle the input value 1 separately to avoid overflow in that case
    // in the general computation below (b/143972021). Also handle 0 as if it
    // were a 1. 0 is an invalid input here (divide by zero) and 1 is a valid
    // but rare/unrealistic input value. We can expect both to occur in some
    // incompletely trained models, but probably not in fully trained models.
    *output_inv_sqrt = std::numeric_limits<std::int32_t>::max();
    *output_shift = 0;
    return;
  }
  TFLITE_DCHECK_GT(input, 1);
  *output_shift = 11;
  while (input >= (1 << 29)) {
    input /= 4;
    ++*output_shift;
  }
  const unsigned max_left_shift_bits =
      CountLeadingZeros(static_cast<uint32_t>(input)) - 1;
  const unsigned max_left_shift_bit_pairs = max_left_shift_bits / 2;
  const unsigned left_shift_bit_pairs = max_left_shift_bit_pairs - 1;
  *output_shift -= left_shift_bit_pairs;
  input <<= 2 * left_shift_bit_pairs;
  TFLITE_DCHECK_GE(input, (1 << 27));
  TFLITE_DCHECK_LT(input, (1 << 29));
  using gemmlowp::FixedPoint;
  using gemmlowp::Rescale;
```

```
  using gemmlowp::SaturatingRoundingMultiplyByPOT;
  // Using 3 integer bits gives us enough room for the internal arithmetic in
  // this Newton-Raphson iteration.
  using F3 = FixedPoint<int32_t, 3>;
  using F0 = FixedPoint<int32_t, 0>;
  const F3 fixedpoint_input = F3::FromRaw(input >> 1);
  const F3 fixedpoint_half_input =
      SaturatingRoundingMultiplyByPOT<-1>(fixedpoint_input);
  const F3 fixedpoint_half_three =
      GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(F3, (1 << 28) + (1 << 27), 1.5);
  // Newton-Raphson iteration
  // Naive unoptimized starting guess: x = 1
  F3 x = F3::One();
  // Naive unoptimized number of iterations: 5
  for (int i = 0; i < 5; i++) {
    const F3 x3 = Rescale<3>(x * x * x);
    x = Rescale<3>(fixedpoint_half_three * x - fixedpoint_half_input * x3);
  }
  const F0 fixedpoint_half_sqrt_2 =
      GEMMLOWP_CHECKED_FIXEDPOINT_CONSTANT(F0, 1518500250, std::sqrt(2.) / 2.);
  x = x * fixedpoint_half_sqrt_2;
  *output_inv_sqrt = x.raw();
  if (*output_shift < 0) {
    *output_inv_sqrt <<= -*output_shift;
    *output_shift = 0;
  }
  // Convert right shift (right is positive) to left shift.
  *output_shift *= reverse_shift;
}

// DO NOT USE THIS STRUCT FOR NEW FUNCTIONALITY BEYOND IMPLEMENTING
// BROADCASTING.
//
// NdArrayDesc<N> describes the shape and memory layout of an N-dimensional
// rectangular array of numbers.
//
// NdArrayDesc<N> is basically identical to Dims<N> defined in types.h.
// However, as Dims<N> is to be deprecated, this class exists as an adaptor
// to enable simple unoptimized implementations of element-wise broadcasting
// operations.
template <int N>
struct NdArrayDesc {
  // The "extent" of each dimension. Indices along dimension d must be in the
  // half-open interval [0, extents[d]).
  int extents[N];

  // The number of *elements* (not bytes) between consecutive indices of each
  // dimension.
  int strides[N];
```

```
};

// DO NOT USE THIS FUNCTION FOR NEW FUNCTIONALITY BEYOND IMPLEMENTING
// BROADCASTING.
//
// Same as Offset(), except takes as NdArrayDesc<N> instead of Dims<N>.
inline int SubscriptToIndex(const NdArrayDesc<4>& desc, int i0, int i1, int i2,
                            int i3) {
  TFLITE_DCHECK(i0 >= 0 && i0 < desc.extents[0]);
  TFLITE_DCHECK(i1 >= 0 && i1 < desc.extents[1]);
  TFLITE_DCHECK(i2 >= 0 && i2 < desc.extents[2]);
  TFLITE_DCHECK(i3 >= 0 && i3 < desc.extents[3]);
  return i0 * desc.strides[0] + i1 * desc.strides[1] + i2 * desc.strides[2] +
         i3 * desc.strides[3];
}

inline int SubscriptToIndex(const NdArrayDesc<5>& desc, int indexes[5]) {
  return indexes[0] * desc.strides[0] + indexes[1] * desc.strides[1] +
         indexes[2] * desc.strides[2] + indexes[3] * desc.strides[3] +
         indexes[4] * desc.strides[4];
}

inline int SubscriptToIndex(const NdArrayDesc<8>& desc, int indexes[8]) {
  return indexes[0] * desc.strides[0] + indexes[1] * desc.strides[1] +
         indexes[2] * desc.strides[2] + indexes[3] * desc.strides[3] +
         indexes[4] * desc.strides[4] + indexes[5] * desc.strides[5] +
         indexes[6] * desc.strides[6] + indexes[7] * desc.strides[7];
}

// Given the dimensions of the operands for an element-wise binary broadcast,
// adjusts them so that they can be directly iterated over with simple loops.
// Returns the adjusted dims as instances of NdArrayDesc in 'desc0_out' and
// 'desc1_out'. 'desc0_out' and 'desc1_out' cannot be nullptr.
//
// This function assumes that the two input shapes are compatible up to
// broadcasting and the shorter one has already been prepended with 1s to be the
// same length. E.g., if shape0 is (1, 16, 16, 64) and shape1 is (1, 64),
// shape1 must already have been prepended to be (1, 1, 1, 64). Recall that
// Dims<N> refer to shapes in reverse order. In this case, input0_dims will be
// (64, 16, 16, 1) and input1_dims will be (64, 1, 1, 1).
//
// When two shapes are compatible up to broadcasting, for each dimension d,
// the input extents are either equal, or one of them is 1.
//
// This function performs the following for each dimension d:
// - If the extents are equal, then do nothing since the loop that walks over
//   both of the input arrays is correct.
// - Otherwise, one (and only one) of the extents must be 1. Say extent0 is 1
//   and extent1 is e1. Then set extent0 to e1 and stride0 *to 0*. This allows
```

```
//   array0 to be referenced *at any index* in dimension d and still access the
//   same slice.
template <int N>
inline void NdArrayDescsForElementwiseBroadcast(const Dims<N>& input0_dims,
                                                const Dims<N>& input1_dims,
                                                NdArrayDesc<N>* desc0_out,
                                                NdArrayDesc<N>* desc1_out) {
  TFLITE_DCHECK(desc0_out != nullptr);
  TFLITE_DCHECK(desc1_out != nullptr);

  // Copy dims to desc.
  for (int i = 0; i < N; ++i) {
    desc0_out->extents[i] = input0_dims.sizes[i];
    desc0_out->strides[i] = input0_dims.strides[i];
    desc1_out->extents[i] = input1_dims.sizes[i];
    desc1_out->strides[i] = input1_dims.strides[i];
  }

  // Walk over each dimension. If the extents are equal do nothing.
  // Otherwise, set the desc with extent 1 to have extent equal to the other and
  // stride 0.
  for (int i = 0; i < N; ++i) {
    const int extent0 = ArraySize(input0_dims, i);
    const int extent1 = ArraySize(input1_dims, i);
    if (extent0 != extent1) {
      if (extent0 == 1) {
        desc0_out->strides[i] = 0;
        desc0_out->extents[i] = extent1;
      } else {
        TFLITE_DCHECK_EQ(extent1, 1);
        desc1_out->strides[i] = 0;
        desc1_out->extents[i] = extent0;
      }
    }
  }
}

// Copies dims to desc, calculating strides.
template <int N>
inline void CopyDimsToDesc(const RuntimeShape& input_shape,
                           NdArrayDesc<N>* desc_out) {
  int desc_stride = 1;
  for (int i = N - 1; i >= 0; --i) {
    desc_out->extents[i] = input_shape.Dims(i);
    desc_out->strides[i] = desc_stride;
    desc_stride *= input_shape.Dims(i);
  }
}

```

```
template <int N>
inline void NdArrayDescsForElementwiseBroadcast(
    const RuntimeShape& input0_shape, const RuntimeShape& input1_shape,
    NdArrayDesc<N>* desc0_out, NdArrayDesc<N>* desc1_out) {
  TFLITE_DCHECK(desc0_out != nullptr);
  TFLITE_DCHECK(desc1_out != nullptr);

  auto extended_input0_shape = RuntimeShape::ExtendedShape(N, input0_shape);
  auto extended_input1_shape = RuntimeShape::ExtendedShape(N, input1_shape);

  // Copy dims to desc, calculating strides.
  CopyDimsToDesc<N>(extended_input0_shape, desc0_out);
  CopyDimsToDesc<N>(extended_input1_shape, desc1_out);

  // Walk over each dimension. If the extents are equal do nothing.
  // Otherwise, set the desc with extent 1 to have extent equal to the other and
  // stride 0.
  for (int i = 0; i < N; ++i) {
    const int extent0 = extended_input0_shape.Dims(i);
    const int extent1 = extended_input1_shape.Dims(i);
    if (extent0 != extent1) {
      if (extent0 == 1) {
        desc0_out->strides[i] = 0;
        desc0_out->extents[i] = extent1;
      } else {
        TFLITE_DCHECK_EQ(extent1, 1);
        desc1_out->strides[i] = 0;
        desc1_out->extents[i] = extent0;
      }
    }
  }
}

template <int N>
inline void NdArrayDescsForElementwiseBroadcast(
    const RuntimeShape& input0_shape, const RuntimeShape& input1_shape,
    const RuntimeShape& input2_shape, NdArrayDesc<N>* desc0_out,
    NdArrayDesc<N>* desc1_out, NdArrayDesc<N>* desc2_out) {
  TFLITE_DCHECK(desc0_out != nullptr);
  TFLITE_DCHECK(desc1_out != nullptr);
  TFLITE_DCHECK(desc2_out != nullptr);

  auto extended_input0_shape = RuntimeShape::ExtendedShape(N, input0_shape);
  auto extended_input1_shape = RuntimeShape::ExtendedShape(N, input1_shape);
  auto extended_input2_shape = RuntimeShape::ExtendedShape(N, input2_shape);

  // Copy dims to desc, calculating strides.
  CopyDimsToDesc<N>(extended_input0_shape, desc0_out);
  CopyDimsToDesc<N>(extended_input1_shape, desc1_out);
```

```
  CopyDimsToDesc<N>(extended_input2_shape, desc2_out);

  // Walk over each dimension. If the extents are equal do nothing.
  // Otherwise, set the desc with extent 1 to have extent equal to the other and
  // stride 0.
  for (int i = 0; i < N; ++i) {
    const int extent0 = extended_input0_shape.Dims(i);
    const int extent1 = extended_input1_shape.Dims(i);
    const int extent2 = extended_input2_shape.Dims(i);

    int extent = extent0;
    if (extent1 != 1) extent = extent1;
    if (extent2 != 1) extent = extent2;

    TFLITE_DCHECK(extent0 == 1 || extent0 == extent);
    TFLITE_DCHECK(extent1 == 1 || extent1 == extent);
    TFLITE_DCHECK(extent2 == 1 || extent2 == extent);

    if (!(extent0 == extent1 && extent1 == extent2)) {
      if (extent0 == 1) {
        desc0_out->strides[i] = 0;
        desc0_out->extents[i] = extent;
      }
      if (extent1 == 1) {
        desc1_out->strides[i] = 0;
        desc1_out->extents[i] = extent;
      }
      if (extent2 == 1) {
        desc2_out->strides[i] = 0;
        desc2_out->extents[i] = extent;
      }
    }
  }
}

// Detailed implementation of NDOpsHelper, the indexes must be a zero array.
// This implementation is equivalent to N nested loops. Ex, if N=4, it can be
// re-writen as:
// for (int b = 0; b < output.extents[0]; ++b) {
//   for (int y = 0; y < output.extents[1]; ++y) {
//     for (int x = 0; x < output.extents[2]; ++x) {
//       for (int c = 0; c < output.extents[3]; ++c) {
//           calc({b,y,x,c});
//       }
//     }
//   }
// }
template <int N, int DIM, typename Calc>
typename std::enable_if<DIM != N - 1, void>::type NDOpsHelperImpl(
```

```
    const NdArrayDesc<N>& output, const Calc& calc, int indexes[N]) {
  for (indexes[DIM] = 0; indexes[DIM] < output.extents[DIM]; ++indexes[DIM]) {
    NDOpsHelperImpl<N, DIM + 1, Calc>(output, calc, indexes);
  }
}

template <int N, int DIM, typename Calc>
typename std::enable_if<DIM == N - 1, void>::type NDOpsHelperImpl(
    const NdArrayDesc<N>& output, const Calc& calc, int indexes[N]) {
  for (indexes[DIM] = 0; indexes[DIM] < output.extents[DIM]; ++indexes[DIM]) {
    calc(indexes);
  }
}

// Execute the calc function in the innermost iteration based on the shape of
// the output. The calc function should take a single argument of type int[N].
template <int N, typename Calc>
inline void NDOpsHelper(const NdArrayDesc<N>& output, const Calc& calc) {
  int indexes[N] = {0};
  NDOpsHelperImpl<N, 0, Calc>(output, calc, indexes);
}
// Copied from gemmlowp::RoundDown when we dropped direct dependency on
// gemmlowp.
//
// Returns the runtime argument rounded down to the nearest multiple of
// the fixed Modulus.
template <unsigned Modulus, typename Integer>
Integer RoundDown(Integer i) {
  return i - (i % Modulus);
}

// Copied from gemmlowp::RoundUp when we dropped direct dependency on
// gemmlowp.
//
// Returns the runtime argument rounded up to the nearest multiple of
// the fixed Modulus.
template <unsigned Modulus, typename Integer>
Integer RoundUp(Integer i) {
  return RoundDown<Modulus>(i + Modulus - 1);
}

// Copied from gemmlowp::CeilQuotient when we dropped direct dependency on
// gemmlowp.
//
// Returns the quotient a / b rounded up ('ceil') to the nearest integer.
template <typename Integer>
Integer CeilQuotient(Integer a, Integer b) {
  return (a + b - 1) / b;
}
```

```cpp

// This function is a copy of gemmlowp::HowManyThreads, copied when we dropped
// the direct dependency of internal/optimized/ on gemmlowp.
//
// It computes a reasonable number of threads to use for a GEMM of shape
// (rows, cols, depth).
//
// TODO(b/131910176): get rid of this function by switching each call site
// to its own more sensible logic for its own workload.
template <int KernelRows>
inline int LegacyHowManyThreads(int max_num_threads, int rows, int cols,
                                int depth) {
  // Early-exit in the default case where multi-threading is disabled.
  if (max_num_threads == 1) {
    return 1;
  }

  // Ensure that each thread has KernelRows rows to process, if at all possible.
  int thread_count = std::min(max_num_threads, rows / KernelRows);

  // Limit the number of threads according to the overall size of the problem.
  if (thread_count > 1) {
    // Empirically determined value.
    static constexpr std::uint64_t min_cubic_size_per_thread = 64 * 1024;

    // We can only multiply two out of three sizes without risking overflow
    const std::uint64_t cubic_size =
        std::uint64_t(rows) * std::uint64_t(cols) * std::uint64_t(depth);

    thread_count = std::min(
        thread_count, static_cast<int>(cubic_size / min_cubic_size_per_thread));
  }

  if (thread_count < 1) {
    thread_count = 1;
  }

  assert(thread_count > 0 && thread_count <= max_num_threads);
  return thread_count;
}

template <typename T>
void optimized_ops_preload_l1_stream(const T* ptr) {
#ifdef __GNUC__
  // builtin offered by GCC-compatible compilers including clang
  __builtin_prefetch(ptr, /* 0 means read */ 0, /* 0 means no locality */ 0);
#else
  (void)ptr;
#endif
```

```
}

template <typename T>
void optimized_ops_preload_l1_keep(const T* ptr) {
#ifdef __GNUC__
  // builtin offered by GCC-compatible compilers including clang
  __builtin_prefetch(ptr, /* 0 means read */ 0, /* 3 means high locality */ 3);
#else
  (void)ptr;
#endif
}

template <typename T>
void optimized_ops_prefetch_write_l1_keep(const T* ptr) {
#ifdef __GNUC__
  // builtin offered by GCC-compatible compilers including clang
  __builtin_prefetch(ptr, /* 1 means write */ 1, /* 3 means high locality */ 3);
#else
  (void)ptr;
#endif
}

}  // namespace tflite

#endif  // TENSORFLOW_LITE_KERNELS_INTERNAL_COMMON_H_
```