karimnosseir [lite] Handle case when src and dst is same tensor during tensor ... ... ✓ History

11 contributors 

276 lines (239 sloc) | 7.6 KB ...

```
1  /* Copyright 2019 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/c/common.h"
17 #include "tensorflow/lite/c/c_api_types.h"
18
19 #ifndef TF_LITE_STATIC_MEMORY
20 #include <stdlib.h>
21 #include <string.h>
22 #endif  // TF_LITE_STATIC_MEMORY
23
24 int TfLiteIntArrayGetSizeInBytes(int size) {
25   static TfLiteIntArray dummy;
26
27   int computed_size = sizeof(dummy) + sizeof(dummy.data[0]) * size;
28 #if defined(_MSC_VER)
29   // Context for why this is needed is in http://b/189926408#comment21
```

```
  computed_size -= sizeof(dummy.data[0]);
#endif
  return computed_size;
}

int TfLiteIntArrayEqual(const TfLiteIntArray* a, const TfLiteIntArray* b) {
  if (a == b) return 1;
  if (a == NULL || b == NULL) return 0;
  return TfLiteIntArrayEqualsArray(a, b->size, b->data);
}

int TfLiteIntArrayEqualsArray(const TfLiteIntArray* a, int b_size,
                              const int b_data[]) {
  if (a == NULL) return (b_size == 0);
  if (a->size != b_size) return 0;
  int i = 0;
  for (; i < a->size; i++)
    if (a->data[i] != b_data[i]) return 0;
  return 1;
}

#ifndef TF_LITE_STATIC_MEMORY

TfLiteIntArray* TfLiteIntArrayCreate(int size) {
  int alloc_size = TfLiteIntArrayGetSizeInBytes(size);
  if (alloc_size <= 0) return NULL;
  TfLiteIntArray* ret = (TfLiteIntArray*)malloc(alloc_size);
  if (!ret) return ret;
  ret->size = size;
  return ret;
}

TfLiteIntArray* TfLiteIntArrayCopy(const TfLiteIntArray* src) {
  if (!src) return NULL;
  TfLiteIntArray* ret = TfLiteIntArrayCreate(src->size);
  if (ret) {
    memcpy(ret->data, src->data, src->size * sizeof(int));
  }
  return ret;
}

void TfLiteIntArrayFree(TfLiteIntArray* a) { free(a); }

#endif  // TF_LITE_STATIC_MEMORY

int TfLiteFloatArrayGetSizeInBytes(int size) {
  static TfLiteFloatArray dummy;

  int computed_size = sizeof(dummy) + sizeof(dummy.data[0]) * size;
```

```
#if defined(_MSC_VER)
  // Context for why this is needed is in http://b/189926408#comment21
  computed_size -= sizeof(dummy.data[0]);
#endif
  return computed_size;
}

#ifndef TF_LITE_STATIC_MEMORY

TfLiteFloatArray* TfLiteFloatArrayCreate(int size) {
  TfLiteFloatArray* ret =
      (TfLiteFloatArray*)malloc(TfLiteFloatArrayGetSizeInBytes(size));
  ret->size = size;
  return ret;
}

void TfLiteFloatArrayFree(TfLiteFloatArray* a) { free(a); }

void TfLiteTensorDataFree(TfLiteTensor* t) {
  if (t->allocation_type == kTfLiteDynamic ||
      t->allocation_type == kTfLitePersistentRo) {
    free(t->data.raw);
  }
  t->data.raw = NULL;
}

void TfLiteQuantizationFree(TfLiteQuantization* quantization) {
  if (quantization->type == kTfLiteAffineQuantization) {
    TfLiteAffineQuantization* q_params =
        (TfLiteAffineQuantization*)(quantization->params);
    if (q_params->scale) {
      TfLiteFloatArrayFree(q_params->scale);
      q_params->scale = NULL;
    }
    if (q_params->zero_point) {
      TfLiteIntArrayFree(q_params->zero_point);
      q_params->zero_point = NULL;
    }
    free(q_params);
  }
  quantization->params = NULL;
  quantization->type = kTfLiteNoQuantization;
}

void TfLiteSparsityFree(TfLiteSparsity* sparsity) {
  if (sparsity == NULL) {
    return;
  }

```

```
  if (sparsity->traversal_order) {
    TfLiteIntArrayFree(sparsity->traversal_order);
    sparsity->traversal_order = NULL;
  }

  if (sparsity->block_map) {
    TfLiteIntArrayFree(sparsity->block_map);
    sparsity->block_map = NULL;
  }

  if (sparsity->dim_metadata) {
    int i = 0;
    for (; i < sparsity->dim_metadata_size; i++) {
      TfLiteDimensionMetadata metadata = sparsity->dim_metadata[i];
      if (metadata.format == kTfLiteDimSparseCSR) {
        TfLiteIntArrayFree(metadata.array_segments);
        metadata.array_segments = NULL;
        TfLiteIntArrayFree(metadata.array_indices);
        metadata.array_indices = NULL;
      }
    }
    free(sparsity->dim_metadata);
    sparsity->dim_metadata = NULL;
  }

  free(sparsity);
}

void TfLiteTensorFree(TfLiteTensor* t) {
  TfLiteTensorDataFree(t);
  if (t->dims) TfLiteIntArrayFree(t->dims);
  t->dims = NULL;

  if (t->dims_signature) {
    TfLiteIntArrayFree((TfLiteIntArray *) t->dims_signature);
  }
  t->dims_signature = NULL;

  TfLiteQuantizationFree(&t->quantization);
  TfLiteSparsityFree(t->sparsity);
  t->sparsity = NULL;
}

void TfLiteTensorReset(TfLiteType type, const char* name, TfLiteIntArray* dims,
                       TfLiteQuantizationParams quantization, char* buffer,
                       size_t size, TfLiteAllocationType allocation_type,
                       const void* allocation, bool is_variable,
                       TfLiteTensor* tensor) {
  TfLiteTensorFree(tensor);
```

```
  tensor->type = type;
  tensor->name = name;
  tensor->dims = dims;
  tensor->params = quantization;
  tensor->data.raw = buffer;
  tensor->bytes = size;
  tensor->allocation_type = allocation_type;
  tensor->allocation = allocation;
  tensor->is_variable = is_variable;

  tensor->quantization.type = kTfLiteNoQuantization;
  tensor->quantization.params = NULL;
}

TfLiteStatus TfLiteTensorCopy(const TfLiteTensor* src, TfLiteTensor* dst) {
  if (!src || !dst)
    return kTfLiteOk;
  if (src->bytes != dst->bytes)
    return kTfLiteError;
  if (src == dst)
    return kTfLiteOk;

  dst->type = src->type;
  if (dst->dims)
    TfLiteIntArrayFree(dst->dims);
  dst->dims = TfLiteIntArrayCopy(src->dims);
  memcpy(dst->data.raw, src->data.raw, src->bytes);
  dst->buffer_handle = src->buffer_handle;
  dst->data_is_stale = src->data_is_stale;
  dst->delegate = src->delegate;

  return kTfLiteOk;
}

void TfLiteTensorRealloc(size_t num_bytes, TfLiteTensor* tensor) {
  if (tensor->allocation_type != kTfLiteDynamic &&
      tensor->allocation_type != kTfLitePersistentRo) {
    return;
  }
  // TODO(b/145340303): Tensor data should be aligned.
  if (!tensor->data.raw) {
    tensor->data.raw = (char*)malloc(num_bytes);
  } else if (num_bytes > tensor->bytes) {
    tensor->data.raw = (char*)realloc(tensor->data.raw, num_bytes);
  }
  tensor->bytes = num_bytes;
}
#endif  // TF_LITE_STATIC_MEMORY

```

```
const char* TfLiteTypeGetName(TfLiteType type) {
  switch (type) {
    case kTfLiteNoType:
      return "NOTYPE";
    case kTfLiteFloat32:
      return "FLOAT32";
    case kTfLiteInt16:
      return "INT16";
    case kTfLiteInt32:
      return "INT32";
    case kTfLiteUInt32:
      return "UINT32";
    case kTfLiteUInt8:
      return "UINT8";
    case kTfLiteInt8:
      return "INT8";
    case kTfLiteInt64:
      return "INT64";
    case kTfLiteUInt64:
      return "UINT64";
    case kTfLiteBool:
      return "BOOL";
    case kTfLiteComplex64:
      return "COMPLEX64";
    case kTfLiteComplex128:
      return "COMPLEX128";
    case kTfLiteString:
      return "STRING";
    case kTfLiteFloat16:
      return "FLOAT16";
    case kTfLiteFloat64:
      return "FLOAT64";
    case kTfLiteResource:
      return "RESOURCE";
    case kTfLiteVariant:
      return "VARIANT";
  }
  return "Unknown type";
}

TfLiteDelegate TfLiteDelegateCreate(void) {
  TfLiteDelegate d = {
      .data_ = NULL,
      .Prepare = NULL,
      .CopyFromBufferHandle = NULL,
      .CopyToBufferHandle = NULL,
      .FreeBufferHandle = NULL,
      .flags = kTfLiteDelegateFlagsNone,
  };
```

```
275     return d;
276 }
```