

ca6f96b62a ▾

...

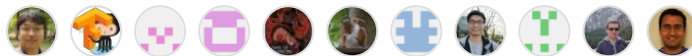
tensorflow / tensorflow / lite / c / common.c



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

```

...

```

30     computed_size -= sizeof(dummy.data[0]);
31 #endif
32     return computed_size;
33 }
34
35 int TfLiteIntArrayEqual(const TfLiteIntArray* a, const TfLiteIntArray* b) {
36     if (a == b) return 1;
37     if (a == NULL || b == NULL) return 0;
38     return TfLiteIntArrayEqualsArray(a, b->size, b->data);
39 }
40
41 int TfLiteIntArrayEqualsArray(const TfLiteIntArray* a, int b_size,
42                               const int b_data[]) {
43     if (a == NULL) return (b_size == 0);
44     if (a->size != b_size) return 0;
45     int i = 0;
46     for (; i < a->size; i++)
47         if (a->data[i] != b_data[i]) return 0;
48     return 1;
49 }
50
51 #ifndef TF_LITE_STATIC_MEMORY
52
53 TfLiteIntArray* TfLiteIntArrayCreate(int size) {
54     int alloc_size = TfLiteIntArrayGetSizeInBytes(size);
55     if (alloc_size <= 0) return NULL;
56     TfLiteIntArray* ret = (TfLiteIntArray*)malloc(alloc_size);
57     if (!ret) return ret;
58     ret->size = size;
59     return ret;
60 }
61
62 TfLiteIntArray* TfLiteIntArrayCopy(const TfLiteIntArray* src) {
63     if (!src) return NULL;
64     TfLiteIntArray* ret = TfLiteIntArrayCreate(src->size);
65     if (ret) {
66         memcpy(ret->data, src->data, src->size * sizeof(int));
67     }
68     return ret;
69 }
70
71 void TfLiteIntArrayFree(TfLiteIntArray* a) { free(a); }
72
73 #endif // TF_LITE_STATIC_MEMORY
74
75 int TfLiteFloatArrayGetSizeInBytes(int size) {
76     static TfLiteFloatArray dummy;
77
78     int computed_size = sizeof(dummy) + sizeof(dummy.data[0]) * size;

```

```

79  #if defined(_MSC_VER)
80      // Context for why this is needed is in http://b/189926408#comment21
81      computed_size -= sizeof(dummy.data[0]);
82  #endif
83      return computed_size;
84  }
85
86  #ifndef TF_LITE_STATIC_MEMORY
87
88  TfLiteFloatArray* TfLiteFloatArrayCreate(int size) {
89      TfLiteFloatArray* ret =
90          (TfLiteFloatArray*)malloc(TfLiteFloatArrayGetSizeInBytes(size));
91      ret->size = size;
92      return ret;
93  }
94
95  void TfLiteFloatArrayFree(TfLiteFloatArray* a) { free(a); }
96
97  void TfLiteTensorDataFree(TfLiteTensor* t) {
98      if (t->allocation_type == kTfLiteDynamic ||
99          t->allocation_type == kTfLitePersistentRo) {
100          free(t->data.raw);
101      }
102      t->data.raw = NULL;
103  }
104
105  void TfLiteQuantizationFree(TfLiteQuantization* quantization) {
106      if (quantization->type == kTfLiteAffineQuantization) {
107          TfLiteAffineQuantization* q_params =
108              (TfLiteAffineQuantization*)(quantization->params);
109          if (q_params->scale) {
110              TfLiteFloatArrayFree(q_params->scale);
111              q_params->scale = NULL;
112          }
113          if (q_params->zero_point) {
114              TfLiteIntArrayFree(q_params->zero_point);
115              q_params->zero_point = NULL;
116          }
117          free(q_params);
118      }
119      quantization->params = NULL;
120      quantization->type = kTfLiteNoQuantization;
121  }
122
123  void TfLiteSparsityFree(TfLiteSparsity* sparsity) {
124      if (sparsity == NULL) {
125          return;
126      }
127

```

```

128     if (sparsity->traversal_order) {
129         TfLiteIntArrayFree(sparsity->traversal_order);
130         sparsity->traversal_order = NULL;
131     }
132
133     if (sparsity->block_map) {
134         TfLiteIntArrayFree(sparsity->block_map);
135         sparsity->block_map = NULL;
136     }
137
138     if (sparsity->dim_metadata) {
139         int i = 0;
140         for (; i < sparsity->dim_metadata_size; i++) {
141             TfLiteDimensionMetadata metadata = sparsity->dim_metadata[i];
142             if (metadata.format == kTfLiteDimSparseCSR) {
143                 TfLiteIntArrayFree(metadata.array_segments);
144                 metadata.array_segments = NULL;
145                 TfLiteIntArrayFree(metadata.array_indices);
146                 metadata.array_indices = NULL;
147             }
148         }
149         free(sparsity->dim_metadata);
150         sparsity->dim_metadata = NULL;
151     }
152
153     free(sparsity);
154 }
155
156 void TfLiteTensorFree(TfLiteTensor* t) {
157     TfLiteTensorDataFree(t);
158     if (t->dims) TfLiteIntArrayFree(t->dims);
159     t->dims = NULL;
160
161     if (t->dims_signature) {
162         TfLiteIntArrayFree((TfLiteIntArray *) t->dims_signature);
163     }
164     t->dims_signature = NULL;
165
166     TfLiteQuantizationFree(&t->quantization);
167     TfLiteSparsityFree(t->sparsity);
168     t->sparsity = NULL;
169 }
170
171 void TfLiteTensorReset(TfLiteType type, const char* name, TfLiteIntArray* dims,
172                       TfLiteQuantizationParams quantization, char* buffer,
173                       size_t size, TfLiteAllocationType allocation_type,
174                       const void* allocation, bool is_variable,
175                       TfLiteTensor* tensor) {
176     TfLiteTensorFree(tensor);

```

```

177     tensor->type = type;
178     tensor->name = name;
179     tensor->dims = dims;
180     tensor->params = quantization;
181     tensor->data.raw = buffer;
182     tensor->bytes = size;
183     tensor->allocation_type = allocation_type;
184     tensor->allocation = allocation;
185     tensor->is_variable = is_variable;
186
187     tensor->quantization.type = kTfLiteNoQuantization;
188     tensor->quantization.params = NULL;
189 }
190
191 TfLiteStatus TfLiteTensorCopy(const TfLiteTensor* src, TfLiteTensor* dst) {
192     if (!src || !dst)
193         return kTfLiteOk;
194     if (src->bytes != dst->bytes)
195         return kTfLiteError;
196     if (src == dst)
197         return kTfLiteOk;
198
199     dst->type = src->type;
200     if (dst->dims)
201         TfLiteIntArrayFree(dst->dims);
202     dst->dims = TfLiteIntArrayCopy(src->dims);
203     memcpy(dst->data.raw, src->data.raw, src->bytes);
204     dst->buffer_handle = src->buffer_handle;
205     dst->data_is_stale = src->data_is_stale;
206     dst->delegate = src->delegate;
207
208     return kTfLiteOk;
209 }
210
211 void TfLiteTensorRealloc(size_t num_bytes, TfLiteTensor* tensor) {
212     if (tensor->allocation_type != kTfLiteDynamic &&
213         tensor->allocation_type != kTfLitePersistentRo) {
214         return;
215     }
216     // TODO(b/145340303): Tensor data should be aligned.
217     if (!tensor->data.raw) {
218         tensor->data.raw = (char*)malloc(num_bytes);
219     } else if (num_bytes > tensor->bytes) {
220         tensor->data.raw = (char*)realloc(tensor->data.raw, num_bytes);
221     }
222     tensor->bytes = num_bytes;
223 }
224 #endif // TF_LITE_STATIC_MEMORY
225

```

```

226 const char* TfLiteTypeGetName(TfLiteType type) {
227     switch (type) {
228         case kTfLiteNoType:
229             return "NOTYPE";
230         case kTfLiteFloat32:
231             return "FLOAT32";
232         case kTfLiteInt16:
233             return "INT16";
234         case kTfLiteInt32:
235             return "INT32";
236         case kTfLiteUInt32:
237             return "UINT32";
238         case kTfLiteUInt8:
239             return "UINT8";
240         case kTfLiteInt8:
241             return "INT8";
242         case kTfLiteInt64:
243             return "INT64";
244         case kTfLiteUInt64:
245             return "UINT64";
246         case kTfLiteBool:
247             return "BOOL";
248         case kTfLiteComplex64:
249             return "COMPLEX64";
250         case kTfLiteComplex128:
251             return "COMPLEX128";
252         case kTfLiteString:
253             return "STRING";
254         case kTfLiteFloat16:
255             return "FLOAT16";
256         case kTfLiteFloat64:
257             return "FLOAT64";
258         case kTfLiteResource:
259             return "RESOURCE";
260         case kTfLiteVariant:
261             return "VARIANT";
262     }
263     return "Unknown type";
264 }
265
266 TfLiteDelegate TfLiteDelegateCreate(void) {
267     TfLiteDelegate d = {
268         .data_ = NULL,
269         .Prepare = NULL,
270         .CopyFromBufferHandle = NULL,
271         .CopyToBufferHandle = NULL,
272         .FreeBufferHandle = NULL,
273         .flags = kTfLiteDelegateFlagsNone,
274     };

```

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