

a1320ec1ea ▾

...

tensorflow / tensorflow / core / framework / tensor.cc



tensorflow-gardener Prevent crashes when loading tensor slices with unsup... ... ✕

History

38 contributors



1316 lines (1194 sloc) | 45.1 KB

...

```

1  /* Copyright 2015 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 // Implementation notes:
17 //
18 // Tensor.cc uses a few templated classes and structs to facilitate
19 // implementation of the Tensor class.
20 //
21 // * Buffer<T>: provides the implementation for a typed array T[n].
22 //   The array is allocated by the given allocator. It runs T's
23 //   default constructors and destructors when T is not a simple type
24 //   (e.g., string.), and skips them otherwise.
25 //
26 // * Helper<T>: provides various routines given type T. The routines
27 //   includes running the constructor and destructor of T[], encoding
28 //   an decoding T[] into/from a Cord, etc.
29

```

```

30 #include "tensorflow/core/framework/tensor.h"
31
32 #include <utility>
33
34 #include "absl/strings/escaping.h"
35 #include "tensorflow/core/framework/allocation_description.pb.h"
36 #include "tensorflow/core/framework/log_memory.h"
37 #include "tensorflow/core/framework/resource_handle.h"
38 #include "tensorflow/core/framework/resource_handle.pb.h"
39 #include "tensorflow/core/framework/tensor.pb.h"
40 #include "tensorflow/core/framework/tensor_description.pb.h"
41 #include "tensorflow/core/framework/type_traits.h"
42 #include "tensorflow/core/framework/typed_allocator.h"
43 #include "tensorflow/core/framework/types.h"
44 #include "tensorflow/core/framework/types.pb.h"
45 #include "tensorflow/core/framework/variant.h"
46 #include "tensorflow/core/framework/variant_encode_decode.h"
47 #include "tensorflow/core/framework/variant_op_registry.h"
48 #include "tensorflow/core/framework/variant_tensor_data.h"
49 #include "tensorflow/core/lib/core/coding.h"
50 #include "tensorflow/core/lib/core/errors.h"
51 #include "tensorflow/core/lib/core/status.h"
52 #include "tensorflow/core/lib/gtl/inlined_vector.h"
53 #include "tensorflow/core/lib/strings/str_util.h"
54 #include "tensorflow/core/lib/strings/strcat.h"
55 #include "tensorflow/core/platform/errors.h"
56 #include "tensorflow/core/platform/logging.h"
57 #include "tensorflow/core/platform/macros.h"
58 #include "tensorflow/core/platform/protobuf.h"
59 #include "tensorflow/core/platform/tensor_coding.h"
60 #include "tensorflow/core/platform/types.h"
61
62 namespace tensorflow {
63
64 // Allow Tensors to be stored inside Variants with automatic
65 // encoding/decoding when those Variants are themselves being decoded
66 // in a Tensor's FromProto.
67 //
68 // NOTE(mrry): The corresponding "copy function" registrations can be found in
69 // ../common_runtime/copy_tensor.cc (due to dependencies on other common_runtime
70 // code).
71 REGISTER_UNARY_VARIANT_DECODE_FUNCTION(Tensor, "tensorflow::Tensor");
72
73 bool TensorBuffer::GetAllocatedBytes(size_t* out_bytes) const {
74     AllocationDescription allocation_description;
75     FillAllocationDescription(&allocation_description);
76     if (allocation_description.allocated_bytes() > 0) {
77         *out_bytes = allocation_description.allocated_bytes();
78         return true;

```

[illegible]

```

128     Allocator* const alloc_;
129 };
130
131 // Typed ref-counted buffer: T[n].
132 template <typename T>
133 class Buffer : public BufferBase {
134 public:
135     Buffer(Allocator* a, int64_t n);
136     Buffer(Allocator* a, int64_t n, const AllocationAttributes& allocation_attr);
137
138     size_t size() const override { return sizeof(T) * elem_; }
139
140 private:
141     int64_t elem_;
142
143     ~Buffer() override;
144
145     TF_DISALLOW_COPY_AND_ASSIGN(Buffer);
146 };
147
148 void LogUnexpectedSize(int64_t actual, int64_t expected) {
149     LOG(ERROR) << "Input size was " << actual << " and expected " << expected;
150 }
151
152 bool MemoryLoggingEnabled() {
153     static bool memory_logging_enabled = LogMemory::IsEnabled();
154     return memory_logging_enabled;
155 }
156
157 // A set of helper functions depending on T.
158 template <typename T>
159 struct Helper {
160     // By default, we assume T is a simple type (float, int32, etc.)
161     static_assert(is_simple_type<T>::value, "T is not a simple type.");
162     typedef protobuf::RepeatedField<T> RepeatedFieldType;
163
164     // Encoder of simple type T to a string. We do a copy.
165     template <typename Destination>
166     static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
167         DCHECK_EQ(in->size(), sizeof(T) * n);
168         port::AssignRefCounted(StringPiece(in->base<const char>(), in->size()), in,
169                                 out);
170     }
171
172     // Decoder of simple type T. Copy the bytes from "in" into the
173     // tensor buffer.
174     template <typename Source>
175     static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
176         if (in.size() != sizeof(T) * n) {

```

```

177     LogUnexpectedSize(in.size(), sizeof(T) * n);
178     return nullptr;
179 }
180 Buffer<T>* buf = new Buffer<T>(a, n);
181 char* data = buf->template base<char>();
182 if (data == nullptr) {
183     buf->Unref();
184     return nullptr;
185 }
186 port::CopyToArray(in, data);
187 return buf;
188 }
189
190 // Memory usage.
191 static int64_t TotalBytes(TensorBuffer* in, int64_t n) {
192     DCHECK_EQ(in->size(), sizeof(T) * n);
193     return in->size();
194 }
195 };
196
197 // Helper specialization for string (the only non-simple type we
198 // support).
199 template <>
200 struct Helper<tstring> {
201     // Proto message uses RepeatedFieldType to hold repeated T.
202     typedef protobuf::RepeatedPtrField<string> RepeatedFieldType;
203
204     // Encodes "n" elements of type string stored in "in" into Cord
205     // "out", which is usually the TensorProto::tensor_content.
206     template <typename Destination>
207     static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
208         port::EncodeStringList(in->base<const tstring>(), n, out);
209     }
210
211     // Decodes "n" elements of type string from "in" and constructs a
212     // buffer out of it. Returns nullptr if the decoding fails. "in" is
213     // usually the TensorProto::tensor_content.
214     template <typename Source>
215     static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
216         Buffer<tstring>* buf = new Buffer<tstring>(a, n);
217         tstring* strings = buf->template base<tstring>();
218         if (strings == nullptr || !port::DecodeStringList(in, strings, n)) {
219             buf->Unref();
220             return nullptr;
221         }
222         return buf;
223     }
224
225     // Returns the estimated memory usage of "n" elements of type T

```

```

226 // stored in buffer "in".
227 static int64_t TotalBytes(TensorBuffer* in, int n) {
228     int64_t tot = in->size();
229     DCHECK_EQ(tot, sizeof(tstring) * n);
230     const tstring* p = in->base<const tstring>();
231     for (int i = 0; i < n; ++i, ++p) tot += p->size();
232     return tot;
233 }
234 };
235
236 template <>
237 struct Helper<ResourceHandle> {
238     // Proto message uses RepeatedFieldType to hold repeated T.
239     typedef protobuf::RepeatedPtrField<string> RepeatedFieldType;
240
241     // Encodes "n" elements of type ResourceHandle stored in "in" into destination
242     // "out", which is usually the TensorProto::tensor_content.
243     template <typename Destination>
244     static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
245         EncodeResourceHandleList(in->base<const ResourceHandle>(), n,
246                                 port::NewStringListEncoder(out));
247     }
248
249     // Decodes "n" elements of type string from "in" and constructs a
250     // buffer out of it. Returns nullptr if the decoding fails. "in" is
251     // usually the TensorProto::tensor_content.
252     template <typename Source>
253     static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
254         auto* buf = new Buffer<ResourceHandle>(a, n);
255         ResourceHandle* ps = buf->template base<ResourceHandle>();
256         if (ps == nullptr ||
257             !DecodeResourceHandleList(port::NewStringListDecoder(in), ps, n)) {
258             buf->Unref();
259             return nullptr;
260         }
261         return buf;
262     }
263
264     // Returns the estimated memory usage of "n" elements of type T
265     // stored in buffer "in".
266     static int64_t TotalBytes(TensorBuffer* in, int n) {
267         return n * sizeof(ResourceHandle);
268     }
269 };
270
271 template <>
272 struct Helper<Variant> {
273     // Encodes "n" elements of type Variant stored in "in" into destination
274     // "out", which is usually the TensorProto::tensor_content.

```

```

275     template <typename Destination>
276     static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
277         EncodeVariantList(in->base<const Variant>(), n,
278             port::NewStringListEncoder(out));
279     }
280
281     // Decodes "n" elements of type Variant from "in" and constructs a
282     // buffer out of it. Returns nullptr if the decoding fails. "in" is
283     // usually the TensorProto::tensor_content.
284     template <typename Source>
285     static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
286         auto* buf = new Buffer<Variant>(a, n);
287         Variant* ps = buf->template base<Variant>();
288         if (ps == nullptr ||
289             !DecodeVariantList(port::NewStringListDecoder(in), ps, n)) {
290             buf->Unref();
291             return nullptr;
292         }
293         return buf;
294     }
295
296     // Returns the estimated memory usage of "n" elements of type T
297     // stored in buffer "in".
298     static int64_t TotalBytes(TensorBuffer* in, int n) {
299         return n * sizeof(Variant);
300     }
301 };
302
303     template <typename T>
304     struct ProtoHelper {};
305
306     // For a C++ type "T" (float, double, int32, etc.), the repeated field
307     // "N"_val (float_val, int_val, label_val, etc.) of type "F" (float,
308     // int32, string, etc) in the TensorProto is used for serializing the
309     // tensor of type "T".
310     #define PROTO_TRAITS(T, F, N) \
311     template <> \
312     struct ProtoHelper<T> { \
313         typedef Helper<F>::RepeatedFieldType FieldType; \
314         static FieldType::const_iterator Begin(const TensorProto& proto) { \
315             return proto.N##_val().begin(); \
316         } \
317         static size_t NumElements(const TensorProto& proto) { \
318             return proto.N##_val().size(); \
319         } \
320         static void Fill(const T* data, size_t n, TensorProto* proto) { \
321             typename ProtoHelper<T>::FieldType copy(data, data + n); \
322             proto->mutable_##N##_val()->Swap(&copy); \
323         } \

```

```

324     };
325     PROTO_TRAITS(float, float, float);
326     PROTO_TRAITS(double, double, double);
327     PROTO_TRAITS(int32, int32, int);
328     PROTO_TRAITS(uint8, int32, int);
329     PROTO_TRAITS(uint16, int32, int);
330     PROTO_TRAITS(uint32, uint32, uint32);
331     PROTO_TRAITS(int16, int32, int);
332     PROTO_TRAITS(int8, int32, int);
333     PROTO_TRAITS(bool, bool, bool);
334     PROTO_TRAITS(tstring, tstring, string);
335     PROTO_TRAITS(qint8, int32, int);
336     PROTO_TRAITS(quint8, int32, int);
337     PROTO_TRAITS(qint16, int32, int);
338     PROTO_TRAITS(quint16, int32, int);
339     #undef PROTO_TRAITS
340
341     template <>
342     struct ProtoHelper<int64_t> {
343         static const int64_t* Begin(const TensorProto& proto) {
344             return reinterpret_cast<const int64_t*>(proto.int64_val().begin());
345         }
346         static size_t NumElements(const TensorProto& proto) {
347             return proto.int64_val().size();
348         }
349         static void Fill(const int64_t* data, size_t n, TensorProto* proto) {
350             protobuf::RepeatedField<protobuf_int64> copy(data, data + n);
351             proto->mutable_int64_val()->Swap(&copy);
352         }
353     };
354
355     template <>
356     struct ProtoHelper<uint64_t> {
357         static const uint64_t* Begin(const TensorProto& proto) {
358             return reinterpret_cast<const uint64_t*>(proto.uint64_val().begin());
359         }
360         static size_t NumElements(const TensorProto& proto) {
361             return proto.uint64_val().size();
362         }
363         static void Fill(const uint64_t* data, size_t n, TensorProto* proto) {
364             protobuf::RepeatedField<protobuf_uint64> copy(data, data + n);
365             proto->mutable_uint64_val()->Swap(&copy);
366         }
367     };
368
369     template <>
370     struct ProtoHelper<ResourceHandle> {
371         static protobuf::RepeatedPtrField<ResourceHandleProto>::const_iterator Begin(
372             const TensorProto& proto) {

```



```

373     return proto.resource_handle_val().begin();
374 }
375 static size_t NumElements(const TensorProto& proto) {
376     return proto.resource_handle_val().size();
377 }
378 static void Fill(const ResourceHandle* data, size_t n, TensorProto* proto) {
379     auto* handles = proto->mutable_resource_handle_val();
380     handles->Clear();
381     for (size_t i = 0; i < n; i++) {
382         data[i].AsProto(handles->Add());
383     }
384 }
385 };
386
387 template <>
388 struct ProtoHelper<Variant> {
389     static protobuf::RepeatedPtrField<VariantTensorDataProto>::const_iterator
390     Begin(const TensorProto& proto) {
391         return proto.variant_val().begin();
392     }
393     static size_t NumElements(const TensorProto& proto) {
394         return proto.variant_val().size();
395     }
396     static void Fill(const Variant* data, size_t n, TensorProto* proto) {
397         auto* variant_values = proto->mutable_variant_val();
398         variant_values->Clear();
399         for (size_t i = 0; i < n; ++i) {
400             VariantTensorData tmp;
401             data[i].Encode(&tmp);
402             tmp.ToProto(variant_values->Add());
403         }
404     }
405 };
406
407 template <>
408 struct ProtoHelper<complex64> {
409     typedef Helper<float>::RepeatedFieldType FieldType;
410     static const complex64* Begin(const TensorProto& proto) {
411         return reinterpret_cast<const complex64*>(proto.scomplex_val().data());
412     }
413     static size_t NumElements(const TensorProto& proto) {
414         return proto.scomplex_val().size() / 2;
415     }
416     static void Fill(const complex64* data, size_t n, TensorProto* proto) {
417         const float* p = reinterpret_cast<const float*>(data);
418         FieldType copy(p, p + n * 2);
419         proto->mutable_scomplex_val()->Swap(&copy);
420     }
421 };

```

```

422
423 template <>
424 struct ProtoHelper<complex128> {
425     typedef Helper<double>::RepeatedFieldType FieldType;
426     static const complex128* Begin(const TensorProto& proto) {
427         return reinterpret_cast<const complex128*>(proto.dcomplex_val().data());
428     }
429     static size_t NumElements(const TensorProto& proto) {
430         return proto.dcomplex_val().size() / 2;
431     }
432     static void Fill(const complex128* data, size_t n, TensorProto* proto) {
433         const double* p = reinterpret_cast<const double*>(data);
434         FieldType copy(p, p + n * 2);
435         proto->mutable_dcomplex_val()->Swap(&copy);
436     }
437 };
438
439 template <>
440 struct ProtoHelper<qint32> {
441     typedef Helper<int32>::RepeatedFieldType FieldType;
442     static const qint32* Begin(const TensorProto& proto) {
443         return reinterpret_cast<const qint32*>(proto.int_val().data());
444     }
445     static size_t NumElements(const TensorProto& proto) {
446         return proto.int_val().size();
447     }
448     static void Fill(const qint32* data, size_t n, TensorProto* proto) {
449         const int32* p = reinterpret_cast<const int32*>(data);
450         FieldType copy(p, p + n);
451         proto->mutable_int_val()->Swap(&copy);
452     }
453 };
454
455 template <>
456 struct ProtoHelper<bfloat16> {
457     static void Fill(const bfloat16* data, size_t n, TensorProto* proto) {
458         proto->mutable_half_val()->Reserve(n);
459         for (size_t i = 0; i < n; ++i) {
460             proto->mutable_half_val()->AddAlreadyReserved(
461                 Eigen::numext::bit_cast<uint16>(data[i]));
462         }
463     }
464 };
465
466 template <>
467 struct ProtoHelper<Eigen::half> {
468     static void Fill(const Eigen::half* data, size_t n, TensorProto* proto) {
469         proto->mutable_half_val()->Reserve(n);
470         for (size_t i = 0; i < n; ++i) {

```

```

471         proto->mutable_half_val()->AddAlreadyReserved(
472             Eigen::numext::bit_cast<uint16>(data[i]));
473     }
474 }
475 };
476
477 template <typename T>
478 Buffer<T>::Buffer(Allocator* a, int64_t n)
479     : BufferBase(a, TypedAllocator::Allocate<T>(a, n, AllocationAttributes()),
480         elem_(n) {})
481
482 template <typename T>
483 Buffer<T>::Buffer(Allocator* a, int64_t n,
484     const AllocationAttributes& allocation_attr)
485     : BufferBase(a, TypedAllocator::Allocate<T>(a, n, allocation_attr),
486         elem_(n) {})
487
488 template <typename T>
489 Buffer<T>::~~Buffer() {
490     if (data()) {
491         if (MemoryLoggingEnabled()) {
492             RecordDeallocation();
493         }
494         TypedAllocator::Deallocate<T>(alloc_, static_cast<T*>(data()), elem_);
495     }
496 }
497
498 // Allocates a T[n] buffer. Fills in the buffer with repeated values
499 // in "in". If "in" has less values than "n", fills the rest of T[n]
500 // with the last value. If "in" has no values, fills T[n] with the
501 // default value for T.
502 //
503 // This routine is using the typed fields (float_val, etc.) in the
504 // tensor proto as opposed to the untyped binary representation
505 // (tensor_content). This is used when we expect the TensorProto is
506 // used by a client program which may not know how to encode a tensor
507 // in the compact binary representation.
508 template <typename T>
509 TensorBuffer* FromProtoField(Allocator* a, const TensorProto& in, int64_t n) {
510     CHECK_GT(n, 0);
511     Buffer<T>* buf = new Buffer<T>(a, n);
512     T* data = buf->template base<T>();
513     if (data == nullptr) {
514         buf->Unref();
515         return nullptr;
516     }
517
518     const int64_t in_n = ProtoHelper<T>::NumElements(in);
519     if (in_n <= 0) {

```

```

520     std::fill_n(data, n, T());
521 } else {
522     auto begin = ProtoHelper<T>::Begin(in);
523     if (n <= in_n) {
524         std::copy_n(begin, n, data);
525     } else {
526         std::copy_n(begin, in_n, data);
527         if (std::is_trivially_copyable<T>::value) {
528             const T last = *(data + in_n - 1);
529             std::fill_n(data + in_n, n - in_n, last);
530         } else {
531             const T& last = *(data + in_n - 1);
532             std::fill_n(data + in_n, n - in_n, last);
533         }
534     }
535 }
536
537 return buf;
538 }
539
540 template <>
541 TensorBuffer* FromProtoField<Variant>(Allocator* a, const TensorProto& in,
542                                       int64_t n) {
543     CHECK_GT(n, 0);
544     Buffer<Variant>* buf = new Buffer<Variant>(a, n);
545     Variant* data = buf->template base<Variant>();
546     if (data == nullptr) {
547         buf->Unref();
548         return nullptr;
549     }
550     const int64_t in_n = ProtoHelper<Variant>::NumElements(in);
551     if (in_n <= 0) {
552         std::fill_n(data, n, Variant());
553     } else {
554         // If tensor shape says we have n < in_n elements in the output tensor
555         // then make sure to only decode the first n out of the in_n elements in the
556         // in tensors. In all other cases, we decode all in_n elements of in and set
557         // the remaining elements up to n to be the default Variant() value.
558         const int64_t real_n = n < in_n ? n : in_n;
559         for (int64_t i = 0; i < real_n; ++i) {
560             data[i] = in.variant_val(i);
561             if (!DecodeUnaryVariant(&data[i])) {
562                 LOG(ERROR) << "Could not decode variant with type_name: \""
563                     << data[i].TypeName()
564                     << "\". Perhaps you forgot to register a "
565                     << "decoder via REGISTER_UNARY_VARIANT_DECODE_FUNCTION?";
566                 buf->Unref();
567                 return nullptr;
568             }

```

```

569     }
570     for (int64_t i = in_n; i < n; ++i) {
571         data[i] = Variant();
572     }
573 }
574 return buf;
575 }
576
577 // fp16 and bfloat16 are opaque to the protobuf, so we deserialize these
578 // identical to uint16 but with data stored in half_val instead of int_val (ie.,
579 // we don't use ProtoHelper<uint16>).
580 template <>
581 TensorBuffer* FromProtoField<Eigen::half>(Allocator* a, const TensorProto& in,
582                                           int64_t n) {
583     CHECK_GT(n, 0);
584     Buffer<Eigen::half>* buf = new Buffer<Eigen::half>(a, n);
585     uint16* data = buf->template base<uint16>();
586     if (data == nullptr) {
587         buf->Unref();
588         return nullptr;
589     }
590     const int64_t in_n = in.half_val().size();
591     auto begin = in.half_val().begin();
592     if (n <= in_n) {
593         std::copy_n(begin, n, data);
594     } else if (in_n > 0) {
595         std::copy_n(begin, in_n, data);
596         const uint16 last = *(data + in_n - 1);
597         std::fill_n(data + in_n, n - in_n, last);
598     } else {
599         std::fill_n(data, n, 0);
600     }
601     return buf;
602 }
603
604 template <>
605 TensorBuffer* FromProtoField<bfloat16>(Allocator* a, const TensorProto& in,
606                                         int64_t n) {
607     CHECK_GT(n, 0);
608     Buffer<bfloat16>* buf = new Buffer<bfloat16>(a, n);
609     uint16* data = buf->template base<uint16>();
610     if (data == nullptr) {
611         buf->Unref();
612         return nullptr;
613     }
614     const int64_t in_n = in.half_val().size();
615     auto begin = in.half_val().begin();
616     if (n <= in_n) {
617         std::copy_n(begin, n, data);

```

```

618     } else if (in_n > 0) {
619         std::copy_n(begin, in_n, data);
620         const uint16 last = *(data + in_n - 1);
621         std::fill_n(data + in_n, n - in_n, last);
622     } else {
623         std::fill_n(data, n, 0);
624     }
625     return buf;
626 }
627
628 // Copies T[n] stored in the buffer "in" into the repeated field in
629 // "out" corresponding to type T.
630 template <typename T>
631 void ToProtoField(const TensorBuffer& in, int64_t n, TensorProto* out) {
632     const T* data = in.base<const T>();
633     // NOTE: T may not be the same as
634     // ProtoHelper<T>::FieldType::value_type. E.g., T==int16,
635     // ProtoHelper<T>::FieldType::value_type==int32. If performance is
636     // critical, we can specialize T=float and do memcpy directly.
637     ProtoHelper<T>::Fill(data, n, out);
638 }
639
640 void RefIfNonNull(core::RefCounted* buf) {
641     if (buf) buf->Ref();
642 }
643
644 void UnrefIfNonNull(core::RefCounted* buf) {
645     if (buf) buf->Unref();
646 }
647
648 } // end namespace
649
650 Tensor::Tensor() : Tensor(DT_FLOAT) {}
651
652 Tensor::Tensor(DataType type) : shape_(type), buf_(nullptr) {}
653
654 Tensor::Tensor(DataType type, const TensorShape& shape, TensorBuffer* buf)
655     : shape_(shape), buf_(buf) {
656     set_dtype(type);
657     RefIfNonNull(buf);
658 }
659
660 Tensor::Tensor(DataType type, TensorShape shape,
661     core::RefCountPtr<TensorBuffer> buf)
662     : shape_(std::move(shape)), buf_(buf.release()) {
663     set_dtype(type);
664 }
665
666 bool Tensor::IsInitialized() const {

```

```

667     return (buf_ != nullptr && buf_->data() != nullptr) ||
668         shape_.num_elements() == 0;
669 }
670
671 void Tensor::CheckType(DataType expected_dtype) const {
672     CHECK_EQ(dtype(), expected_dtype)
673         << " " << DataTypeString(expected_dtype) << " expected, got "
674         << DataTypeString(dtype());
675 }
676
677 void Tensor::CheckTypeAndIsAligned(DataType expected_dtype) const {
678     CHECK_EQ(dtype(), expected_dtype)
679         << " " << DataTypeString(expected_dtype) << " expected, got "
680         << DataTypeString(dtype());
681     CHECK(IsAligned()) << "ptr = " << base<void>();
682 }
683
684 void Tensor::CheckIsAlignedAndSingleElement() const {
685     CHECK(IsAligned()) << "Aligned and single element";
686     CHECK_EQ(1, NumElements()) << "Must have a one element tensor";
687 }
688
689 Tensor::~Tensor() { UnrefIfNonNull(buf_); }
690
691 Status Tensor::BitcastFrom(const Tensor& other, DataType dtype,
692                           const TensorShape& shape) {
693     int in_size = DataTypeSize(other.dtype());
694     int out_size = DataTypeSize(dtype);
695     if (in_size == 0) {
696         return errors::InvalidArgument("other tensor has zero-sized data type");
697     }
698     if (out_size == 0) {
699         return errors::InvalidArgument("specified output type is zero-sized");
700     }
701     if (shape.num_elements() * out_size !=
702         other.shape().num_elements() * in_size) {
703         return errors::InvalidArgument(
704             "input and output shapes/data type sizes are not compatible");
705     }
706     shape_ = shape;
707     shape_.set_data_type(dtype);
708     if (buf_ != other.buf_) {
709         UnrefIfNonNull(buf_);
710         buf_ = other.buf_;
711         RefIfNonNull(buf_);
712     }
713     return Status::OK();
714 }
715

```

```

716 // Notice that buf_ either points to a regular TensorBuffer or a SubBuffer.
717 // For the latter case, we have to make sure that the refcount is
718 // one both for the SubBuffer _and_ the underlying TensorBuffer.
719 bool Tensor::RefCountIsOne() const {
720     return buf_ != nullptr && buf_->RefCountIsOne() &&
721         buf_->root_buffer()->RefCountIsOne() && buf_->OwnsMemory();
722 }
723
724 // The macro CASES() expands to a switch statement conditioned on
725 // TYPE_ENUM. Each case expands the STMTS after a typedef for T.
726 #define SINGLE_ARG(...) __VA_ARGS__
727 #define CASE(TYPE, STMTS) \
728     case DataTypeToEnum<TYPE>::value: { \
729         typedef TF_ATTRIBUTE_UNUSED TYPE T; \
730         STMTS; \
731         break; \
732     }
733 #define CASES_WITH_DEFAULT(TYPE_ENUM, STMTS, INVALID, DEFAULT) \
734     switch (TYPE_ENUM) { \
735         CASE(float, SINGLE_ARG(STMTS)) \
736         CASE(double, SINGLE_ARG(STMTS)) \
737         CASE(int32, SINGLE_ARG(STMTS)) \
738         CASE(uint8, SINGLE_ARG(STMTS)) \
739         CASE(uint16, SINGLE_ARG(STMTS)) \
740         CASE(uint32, SINGLE_ARG(STMTS)) \
741         CASE(uint64, SINGLE_ARG(STMTS)) \
742         CASE(int16, SINGLE_ARG(STMTS)) \
743         CASE(int8, SINGLE_ARG(STMTS)) \
744         CASE(tstring, SINGLE_ARG(STMTS)) \
745         CASE(complex64, SINGLE_ARG(STMTS)) \
746         CASE(complex128, SINGLE_ARG(STMTS)) \
747         CASE(int64_t, SINGLE_ARG(STMTS)) \
748         CASE(bool, SINGLE_ARG(STMTS)) \
749         CASE(qint32, SINGLE_ARG(STMTS)) \
750         CASE(quint8, SINGLE_ARG(STMTS)) \
751         CASE(qint8, SINGLE_ARG(STMTS)) \
752         CASE(quint16, SINGLE_ARG(STMTS)) \
753         CASE(qint16, SINGLE_ARG(STMTS)) \
754         CASE(bfloat16, SINGLE_ARG(STMTS)) \
755         CASE(Eigen::half, SINGLE_ARG(STMTS)) \
756         CASE(ResourceHandle, SINGLE_ARG(STMTS)) \
757         CASE(Variant, SINGLE_ARG(STMTS)) \
758         case DT_INVALID: \
759             INVALID; \
760             break; \
761         default: \
762             DEFAULT; \
763             break; \
764     }

```



```

765
766 #define CASES(TYPE_ENUM, STMTS) \
767     CASES_WITH_DEFAULT(TYPE_ENUM, STMTS, LOG(FATAL) << "Type not set"; \
768         , LOG(FATAL) << "Unexpected type: " << TYPE_ENUM;)
769
770 Tensor::Tensor(Allocator* a, DataType type, const TensorShape& shape)
771     : shape_(shape), buf_(nullptr) {
772     set_dtype(type);
773     CHECK_NOTNULL(a);
774     if (shape_.num_elements() > 0 || a->AllocatesOpaqueHandle()) {
775         CASES(type, buf_ = new Buffer<T>(a, shape.num_elements()));
776     }
777     if (MemoryLoggingEnabled() && buf_ != nullptr && buf_->data() != nullptr) {
778         LogMemory::RecordTensorAllocation("Unknown", LogMemory::UNKNOWN_STEP_ID,
779             *this);
780     }
781 }
782
783 Tensor::Tensor(Allocator* a, DataType type, const TensorShape& shape,
784     const AllocationAttributes& allocation_attr)
785     : shape_(shape), buf_(nullptr) {
786     set_dtype(type);
787     CHECK_NOTNULL(a);
788     if (shape_.num_elements() > 0 || a->AllocatesOpaqueHandle()) {
789         CASES(type, buf_ = new Buffer<T>(a, shape.num_elements(), allocation_attr));
790     }
791     if (MemoryLoggingEnabled() && !allocation_attr.allocation_will_be_logged &&
792         buf_ != nullptr && buf_->data() != nullptr) {
793         LogMemory::RecordTensorAllocation("Unknown (with attributes)",
794             LogMemory::UNKNOWN_STEP_ID, *this);
795     }
796 }
797
798 Status Tensor::BuildTensor(DataType type, const TensorShape& shape,
799     Tensor* out_tensor) {
800     // Avoid crashes due to invalid or unsupported types.
801     CASES_WITH_DEFAULT(
802         type, {}, return errors::InvalidArgument("Type not set"),
803         return errors::InvalidArgument("Unexpected type: ", DataType_Name(type)));
804     *out_tensor = Tensor(type, shape);
805     return Status::OK();
806 }
807
808 // NOTE(mrry): The default allocator for a Tensor (when none is specified) is
809 // the default CPU allocator for NUMA zone 0. Accessing that currently involves
810 // acquiring a lock, which guards initialization of the per-NUMA zone
811 // allocators, and becomes highly contended.
812 //
813 // Note also that it would be better if all Tensor allocations required the user

```

```

814 // to specify an allocator, for purposes of accounting, etc. However, the
815 // default allocator is widely used throughout the codebase and in client code.
816 static Allocator* get_default_cpu_allocator() {
817     static Allocator* default_cpu_allocator =
818         cpu_allocator(port::kNUMANoAffinity);
819     return default_cpu_allocator;
820 }
821
822 Tensor::Tensor(DataType type, const TensorShape& shape)
823     : Tensor(get_default_cpu_allocator(), type, shape) {}
824
825 bool Tensor::HostScalarTensorBufferBase::GetAllocatedBytes(
826     size_t* out_bytes) const {
827     // `this->FillAllocationDescription()` never sets allocated bytes information,
828     // so we can short-circuit the construction of an `AllocationDescription`.
829     return false;
830 }
831
832 void Tensor::HostScalarTensorBufferBase::FillAllocationDescription(
833     AllocationDescription* proto) const {
834     proto->set_requested_bytes(size());
835     proto->set_allocator_name("HostScalarTensorBuffer");
836     proto->set_ptr(reinterpret_cast<uintptr_t>(data()));
837 }
838
839 template <typename T>
840 class SubBuffer : public TensorBuffer {
841 public:
842     // This buffer is an alias to buf[delta, delta + n).
843     SubBuffer(TensorBuffer* buf, int64_t delta, int64_t n)
844         : TensorBuffer(buf->base<T>() + delta),
845           root_(buf->root_buffer()),
846           elem_(n) {
847         // Sanity check. The caller should ensure the sub buffer is valid.
848         CHECK_LE(root_->base<T>(), this->base<T>());
849         T* root_limit = root_->base<T>() + root_->size() / sizeof(T);
850         CHECK_LE(this->base<T>(), root_limit);
851         CHECK_LE(this->base<T>() + n, root_limit);
852         // Hold a ref of the underlying root buffer.
853         // NOTE: 'buf' is a sub-buffer inside the 'root_' buffer.
854         root_->Ref();
855     }
856
857     size_t size() const override { return sizeof(T) * elem_; }
858     TensorBuffer* root_buffer() override { return root_; }
859     bool GetAllocatedBytes(size_t* out_bytes) const override {
860         return root_->GetAllocatedBytes(out_bytes);
861     }
862     void FillAllocationDescription(AllocationDescription* proto) const override {

```

```

863     root_->FillAllocationDescription(proto);
864 }
865
866 private:
867     TensorBuffer* root_;
868     int64_t elem_;
869
870     ~SubBuffer() override { root_->Unref(); }
871
872     TF_DISALLOW_COPY_AND_ASSIGN(SubBuffer);
873 };
874
875 Tensor Tensor::Slice(int64_t start, int64_t limit) const {
876     CHECK_GE(dims(), 1);
877     CHECK_LE(0, start);
878     CHECK_LE(start, limit);
879     int64_t dim0_size = shape_.dim_size(0);
880     CHECK_LE(limit, dim0_size);
881     if ((start == 0) && (limit == dim0_size)) {
882         return *this;
883     }
884     Tensor ret;
885     ret.shape_ = shape_;
886     ret.set_dtype(dtype());
887     ret.buf_ = nullptr;
888     if (dim0_size > 0) {
889         const int64_t elems_per_dim0 = NumElements() / dim0_size;
890         const int64_t delta = start * elems_per_dim0;
891         dim0_size = limit - start;
892         ret.shape_.set_dim(0, dim0_size);
893         const int64_t num_elems = dim0_size * elems_per_dim0;
894         if (buf_) {
895             DataType dt = dtype();
896             CASES(dt, ret.buf_ = new SubBuffer<T>(buf_, delta, num_elems));
897         }
898     }
899     return ret;
900 }
901
902 Tensor Tensor::SubSlice(int64_t index) const {
903     CHECK_GE(dims(), 1); // Crash ok.
904     CHECK_LE(0, index); // Crash ok.
905     int64_t dim0_size = shape_.dim_size(0);
906     CHECK_LE(index, dim0_size); // Crash ok.
907     Tensor ret;
908     ret.shape_ = shape_;
909     ret.shape_.RemoveDim(0);
910     ret.set_dtype(dtype());
911     ret.buf_ = nullptr;

```

```

912     if (dim0_size > 0) {
913         const int64_t elems_per_dim0 = NumElements() / dim0_size;
914         const int64_t delta = index * elems_per_dim0;
915         const int64_t num_elems = elems_per_dim0;
916         if (buf_) {
917             DataType dt = dtype();
918             CASES(dt, ret.buf_ = new SubBuffer<T>(buf_, delta, num_elems));
919         }
920     }
921     return ret;
922 }
923
924 bool Tensor::FromProto(const TensorProto& proto) {
925     return FromProto(get_default_cpu_allocator(), proto);
926 }
927
928 bool Tensor::FromProto(Allocator* a, const TensorProto& proto) {
929     CHECK_NOTNULL(a);
930     TensorBuffer* p = nullptr;
931     if (!TensorShape::IsValid(proto.tensor_shape())) return false;
932     if (proto.dtype() == DT_INVALID) return false;
933     TensorShape shape(proto.tensor_shape());
934     const int64_t N = shape.num_elements();
935     if (N > 0 && proto.dtype()) {
936         bool dtype_error = false;
937         if (!proto.tensor_content().empty()) {
938             const auto& content = proto.tensor_content();
939             CASES_WITH_DEFAULT(proto.dtype(), p = Helper<T>::Decode(a, content, N),
940                               dtype_error = true, dtype_error = true);
941         } else {
942             CASES_WITH_DEFAULT(proto.dtype(), p = FromProtoField<T>(a, proto, N),
943                               dtype_error = true, dtype_error = true);
944         }
945         if (dtype_error || p == nullptr) return false;
946     }
947     shape_ = shape;
948     set_dtype(proto.dtype());
949     UnrefIfNonNull(buf_);
950     buf_ = p;
951     // TODO(misard) add tracking of which kernels and steps are calling
952     // FromProto.
953     if (MemoryLoggingEnabled() && buf_ != nullptr && buf_>data() != nullptr) {
954         LogMemory::RecordTensorAllocation("Unknown (from Proto)",
955                                           LogMemory::UNKNOWN_STEP_ID, *this);
956     }
957     return true;
958 }
959
960 void Tensor::AsProtoField(TensorProto* proto) const {

```

```

961     proto->Clear();
962     shape_.AsProto(proto->mutable_tensor_shape());
963     proto->set_dtype(dtype());
964     if (buf_) {
965         CASES(dtype(), ToProtoField<T>(*buf_, shape_.num_elements(), proto));
966     }
967 }
968
969 void Tensor::AsProtoTensorContent(TensorProto* proto) const {
970     proto->Clear();
971     proto->set_dtype(dtype());
972     shape_.AsProto(proto->mutable_tensor_shape());
973     if (buf_) {
974         CASES(dtype(), Helper<T>::Encode(buf_, shape_.num_elements(),
975                                         proto->mutable_tensor_content()));
976     }
977 }
978
979 size_t Tensor::TotalBytes() const {
980     if (shape_.num_elements() == 0) return 0;
981     CHECK(buf_) << "null buf_ with non-zero shape size " << shape_.num_elements();
982     CASES(dtype(), return Helper<T>::TotalBytes(buf_, shape_.num_elements()));
983     return 0; // Makes compiler happy.
984 }
985
986 size_t Tensor::AllocatedBytes() const {
987     if (buf_) {
988         size_t ret;
989         if (buf_->GetAllocatedBytes(&ret)) {
990             return ret;
991         }
992     }
993     return TotalBytes();
994 }
995
996 bool Tensor::CanUseDMA() const {
997     CASES(dtype(), return is_simple_type<T>::value);
998     return false; // Makes compiler happy.
999 }
1000
1001 #undef CASES
1002 #undef CASE
1003
1004 namespace {
1005
1006 // StrCat and StrAppend don't support Eigen::half directly at the moment, and
1007 // we would like to keep them compatible with their absl counterparts, for ease
1008 // of migration. We could rely on errors::internal::PrepareForStrCat() but the
1009 // logic is so simple we can just replicate it here, where it is close to its

```

```

1010 // usage and easy to change later. And there's the extra benefit of not
1011 // accessing an 'internal' namespace.
1012 inline const strings::AlphaNum& PrintOneElement(const strings::AlphaNum& a,
1013                                                  bool print_v2) {
1014     return a;
1015 }
1016 inline string PrintOneElement(const tstring& a, bool print_v2) {
1017     if (print_v2) {
1018         return "\"" + absl::Utf8SafeCEscape(a) + "\"";
1019     } else {
1020         return absl::Utf8SafeCEscape(a);
1021     }
1022 }
1023 inline float PrintOneElement(const Eigen::half& h, bool print_v2) {
1024     return static_cast<float>(h);
1025 }
1026
1027 inline float PrintOneElement(bfloat16 f, bool print_v2) {
1028     return static_cast<float>(f);
1029 }
1030
1031 // Print from left dim to right dim recursively.
1032 template <typename T>
1033 void PrintOneDim(int dim_index, const gtl::InlinedVector<int64, 4>& shape,
1034                 int64_t limit, int shape_size, const T* data,
1035                 int64_t* data_index, string* result) {
1036     if (*data_index >= limit) return;
1037     int64_t element_count = shape[dim_index];
1038     // We have reached the right-most dimension of the tensor.
1039     if (dim_index == shape_size - 1) {
1040         for (int64_t i = 0; i < element_count; i++) {
1041             if (*data_index >= limit) {
1042                 // If not enough elements has been printed, append "...".
1043                 if (dim_index != 0) {
1044                     strings::StrAppend(result, "...");
1045                 }
1046                 return;
1047             }
1048             if (i > 0) strings::StrAppend(result, " ");
1049             strings::StrAppend(result, PrintOneElement(data[(*data_index)++], false));
1050         }
1051         return;
1052     }
1053     // Loop every element of one dim.
1054     for (int64_t i = 0; i < element_count; i++) {
1055         bool flag = false;
1056         if (*data_index < limit) {
1057             strings::StrAppend(result, "[" );
1058             flag = true;

```

```

1059     }
1060     // As for each element, print the sub-dim.
1061     PrintOneDim(dim_index + 1, shape, limit, shape_size, data, data_index,
1062                 result);
1063     if (*data_index < limit || flag) {
1064         strings::StrAppend(result, "];");
1065         flag = false;
1066     }
1067 }
1068 }
1069
1070 // Appends the spacing between elements for a given dim onto a result string
1071 void PrintDimSpacing(int dim_index, int num_dims, string* result) {
1072     if (dim_index == num_dims - 1) {
1073         strings::StrAppend(result, " ");
1074         return;
1075     }
1076     for (int j = 0; j < num_dims - dim_index - 1; j++) {
1077         strings::StrAppend(result, "\n");
1078     }
1079     for (int j = 0; j <= dim_index; j++) {
1080         strings::StrAppend(result, " ");
1081     }
1082 }
1083
1084 // Print from left dim to right dim recursively.
1085 template <typename T>
1086 void PrintOneDimV2(int dim_index, const gtl::InlinedVector<int64, 4>& shape,
1087                   int64_t num_elts_at_ends, int num_dims, const T* data,
1088                   int64_t data_index, string* result) {
1089     // We have recursed beyond all the dimensions into a single element
1090     // of the tensor.
1091     if (dim_index == num_dims) {
1092         strings::StrAppend(result, PrintOneElement(data[data_index], true));
1093         return;
1094     }
1095
1096     strings::StrAppend(result, "[");
1097     int64_t element_count = shape[dim_index];
1098     int64_t start_of_end =
1099         std::max(num_elts_at_ends, element_count - num_elts_at_ends);
1100
1101     // Loop every element of one dim.
1102     int64_t elements_per_iter = 1;
1103     for (int i = dim_index + 1; i < num_dims; i++) {
1104         elements_per_iter *= shape[i];
1105     }
1106     for (int64_t i = 0; (i < num_elts_at_ends) && (i < element_count); i++) {
1107         if (i > 0) {

```

```

1108     PrintDimSpacing(dim_index, num_dims, result);
1109 }
1110
1111 // As for each element, print the sub-dim.
1112 PrintOneDimV2(dim_index + 1, shape, num_elts_at_ends, num_dims, data,
1113             data_index + elements_per_iter * i, result);
1114 }
1115 if (element_count > 2 * num_elts_at_ends) {
1116     PrintDimSpacing(dim_index, num_dims, result);
1117     strings::StrAppend(result, "...");
1118 }
1119 for (int64_t i = start_of_end; i < element_count; i++) {
1120     // As for each element, print the sub-dim.
1121     PrintDimSpacing(dim_index, num_dims, result);
1122     PrintOneDimV2(dim_index + 1, shape, num_elts_at_ends, num_dims, data,
1123                 data_index + elements_per_iter * i, result);
1124 }
1125
1126 strings::StrAppend(result, "]");
1127 }
1128
1129 template <typename T>
1130 string SummarizeArray(int64_t limit, int64_t num_elts,
1131                     const TensorShape& tensor_shape, const char* data,
1132                     const bool print_v2) {
1133     string ret;
1134     const T* array = reinterpret_cast<const T*>(data);
1135
1136     const gtl::InlinedVector<int64_t, 4> shape = tensor_shape.dim_sizes();
1137     if (shape.empty()) {
1138         for (int64_t i = 0; i < limit; ++i) {
1139             if (i > 0) strings::StrAppend(&ret, " ");
1140             strings::StrAppend(&ret, PrintOneElement(array[i], print_v2));
1141         }
1142         if (num_elts > limit) strings::StrAppend(&ret, "...");
1143         return ret;
1144     }
1145     if (print_v2) {
1146         const int num_dims = tensor_shape.dims();
1147         PrintOneDimV2(0, shape, limit, num_dims, array, 0, &ret);
1148     } else {
1149         int64_t data_index = 0;
1150         const int shape_size = tensor_shape.dims();
1151         PrintOneDim(0, shape, limit, shape_size, array, &data_index, &ret);
1152
1153         if (num_elts > limit) strings::StrAppend(&ret, "...");
1154     }
1155
1156     return ret;

```



```

1157 }
1158 } // namespace
1159
1160 string Tensor::SummarizeValue(int64_t max_entries, bool print_v2) const {
1161     const int64_t num_elts = NumElements();
1162     if (max_entries < 0) {
1163         max_entries = num_elts;
1164     }
1165     size_t limit = std::min(max_entries, num_elts);
1166     if ((limit > 0) && (buf_ == nullptr)) {
1167         return strings::StrCat("uninitialized Tensor of ", num_elts,
1168                                " elements of type ", dtype());
1169     }
1170     const char* data = limit > 0 ? tensor_data().data() : nullptr;
1171     switch (dtype()) {
1172     case DT_BFLOAT16:
1173         return SummarizeArray<bfloat16>(limit, num_elts, shape_, data, print_v2);
1174         break;
1175     case DT_HALF:
1176         return SummarizeArray<Eigen::half>(limit, num_elts, shape_, data,
1177                                             print_v2);
1178         break;
1179     case DT_FLOAT:
1180         return SummarizeArray<float>(limit, num_elts, shape_, data, print_v2);
1181         break;
1182     case DT_DOUBLE:
1183         return SummarizeArray<double>(limit, num_elts, shape_, data, print_v2);
1184         break;
1185     case DT_UINT32:
1186         return SummarizeArray<uint32>(limit, num_elts, shape_, data, print_v2);
1187         break;
1188     case DT_INT32:
1189         return SummarizeArray<int32>(limit, num_elts, shape_, data, print_v2);
1190         break;
1191     case DT_UINT8:
1192     case DT_QUINT8:
1193         return SummarizeArray<uint8>(limit, num_elts, shape_, data, print_v2);
1194         break;
1195     case DT_UINT16:
1196     case DT_QUINT16:
1197         return SummarizeArray<uint16>(limit, num_elts, shape_, data, print_v2);
1198         break;
1199     case DT_INT16:
1200     case DT_QINT16:
1201         return SummarizeArray<int16>(limit, num_elts, shape_, data, print_v2);
1202         break;
1203     case DT_INT8:
1204     case DT_QINT8:
1205         return SummarizeArray<int8>(limit, num_elts, shape_, data, print_v2);

```

```

1206         break;
1207     case DT_UINT64:
1208         return SummarizeArray<uint64>(limit, num_elts, shape_, data, print_v2);
1209         break;
1210     case DT_INT64:
1211         return SummarizeArray<int64_t>(limit, num_elts, shape_, data, print_v2);
1212         break;
1213     case DT_BOOL:
1214         // TODO(tucker): Is it better to emit "True False..."? This
1215         // will emit "1 0..." which is more compact.
1216         return SummarizeArray<bool>(limit, num_elts, shape_, data, print_v2);
1217         break;
1218     case DT_STRING:
1219         return SummarizeArray<tstring>(limit, num_elts, shape_, data, print_v2);
1220         break;
1221     default: {
1222         // All irregular cases
1223         string ret;
1224         if (print_v2 && (dims() > 0)) {
1225             strings::StrAppend(&ret, "[");
1226         }
1227         // TODO(irving): Don't call flat every time around this
1228         // loop.
1229         for (size_t i = 0; i < limit; ++i) {
1230             if (i > 0) strings::StrAppend(&ret, " ");
1231             switch (dtype()) {
1232                 case DT_VARIANT: {
1233                     const Variant& v = flat<Variant>()(i);
1234                     strings::StrAppend(&ret, "<", v.SummarizeValue(), ">");
1235                 } break;
1236                 case DT_RESOURCE: {
1237                     const ResourceHandle& r = flat<ResourceHandle>()(i);
1238                     strings::StrAppend(&ret, "<", r.SummarizeValue(), ">");
1239                 } break;
1240                 default:
1241                     // TODO(zhifengc, josh11b): Pretty-print other types (bool,
1242                     // complex64, quantized).
1243                     strings::StrAppend(&ret, "?");
1244             }
1245         }
1246         if (max_entries < num_elts) strings::StrAppend(&ret, "...");
1247         if (print_v2 && (dims() > 0)) {
1248             strings::StrAppend(&ret, "]");
1249         }
1250         return ret;
1251     }
1252 }
1253 }
1254

```

```

1255 StringPiece Tensor::tensor_data() const {
1256     if (buf_ == nullptr) return StringPiece(); // Don't die for empty tensors
1257     return StringPiece(static_cast<char*>(buf_->data()), TotalBytes());
1258 }
1259
1260 void* Tensor::data() const {
1261     if (buf_ == nullptr) return nullptr; // Don't die for empty tensors
1262     return static_cast<void*>(buf_->data());
1263 }
1264
1265 bool Tensor::SharesBufferWith(const Tensor& b) const {
1266     return buf_ != nullptr && b.buf_ != nullptr &&
1267         buf_->root_buffer() == b.buf_->root_buffer();
1268 }
1269
1270 string Tensor::DebugString(int num_values) const {
1271     return strings::StrCat("Tensor<type: ", DataTypeString(dtype()),
1272         " shape: ", shape().DebugString(),
1273         " values: ", SummarizeValue(num_values), ">");
1274 }
1275
1276 string Tensor::DeviceSafeDebugString() const {
1277     return strings::StrCat("Tensor<type: ", DataTypeString(dtype()),
1278         " shape: ", shape().DebugString(), ">");
1279 }
1280
1281 void Tensor::FillDescription(TensorDescription* description) const {
1282     description->set_dtype(dtype());
1283     shape().AsProto(description->mutable_shape());
1284     if (buf_ != nullptr && buf_->data() != nullptr) {
1285         buf_->FillAllocationDescription(
1286             description->mutable_allocation_description());
1287     }
1288 }
1289
1290 gtl::InlinedVector<int64_t, 4> Tensor::ComputeFlatInnerDims(
1291     gtl::ArraySlice<int64_t> orig, int64_t num_out_dims) {
1292     gtl::InlinedVector<int64_t, 4> out_dims(num_out_dims, 0);
1293     int64_t offset = orig.size() - num_out_dims;
1294     for (int64_t out_dim = num_out_dims - 1; out_dim >= 0; --out_dim) {
1295         const int64_t in_dim = out_dim + offset;
1296         out_dims[out_dim] = in_dim < 0 ? 1 : orig[in_dim];
1297     }
1298     for (int64_t in_dim = 0; in_dim < offset; ++in_dim) {
1299         out_dims[0] *= orig[in_dim];
1300     }
1301     return out_dims;
1302 }
1303

```

```
1304 gtl::InlinedVector<int64_t, 4> Tensor::ComputeFlatOuterDims(  
1305     gtl::ArraySlice<int64_t> orig, int64_t num_out_dims) {  
1306     gtl::InlinedVector<int64_t, 4> out_dims(num_out_dims, 0);  
1307     for (int64_t out_dim = 0; out_dim <= num_out_dims - 1; ++out_dim) {  
1308         out_dims[out_dim] = out_dim >= orig.size() ? 1 : orig[out_dim];  
1309     }  
1310     for (int64_t in_dim = num_out_dims; in_dim < orig.size(); ++in_dim) {  
1311         out_dims[num_out_dims - 1] *= orig[in_dim];  
1312     }  
1313     return out_dims;  
1314 }  
1315  
1316 } // namespace tensorflow
```