tensorflower-gardener Prevent crashes when loading tensor slices with unsup... History

38 contributors +21

1316 lines (1194 sloc) | 45.1 KB

```
/* Copyright 2015 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.
==============================================================================*/

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

```

```
#include "tensorflow/core/framework/tensor.h"

#include <utility>

#include "absl/strings/escaping.h"
#include "tensorflow/core/framework/allocation_description.pb.h"
#include "tensorflow/core/framework/log_memory.h"
#include "tensorflow/core/framework/resource_handle.h"
#include "tensorflow/core/framework/resource_handle.pb.h"
#include "tensorflow/core/framework/tensor.pb.h"
#include "tensorflow/core/framework/tensor_description.pb.h"
#include "tensorflow/core/framework/type_traits.h"
#include "tensorflow/core/framework/typed_allocator.h"
#include "tensorflow/core/framework/types.h"
#include "tensorflow/core/framework/types.pb.h"
#include "tensorflow/core/framework/variant.h"
#include "tensorflow/core/framework/variant_encode_decode.h"
#include "tensorflow/core/framework/variant_op_registry.h"
#include "tensorflow/core/framework/variant_tensor_data.h"
#include "tensorflow/core/lib/core/coding.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/core/status.h"
#include "tensorflow/core/lib/gtl/inlined_vector.h"
#include "tensorflow/core/lib/strings/str_util.h"
#include "tensorflow/core/lib/strings/strcat.h"
#include "tensorflow/core/platform/errors.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/platform/macros.h"
#include "tensorflow/core/platform/protobuf.h"
#include "tensorflow/core/platform/tensor_coding.h"
#include "tensorflow/core/platform/types.h"

namespace tensorflow {

// Allow Tensors to be stored inside Variants with automatic
// encoding/decoding when those Variants are themselves being decoded
// in a Tensor's FromProto.
//
// NOTE(mrry): The corresponding "copy function" registrations can be found in
// ../common_runtime/copy_tensor.cc (due to dependencies on other common_runtime
// code).
REGISTER_UNARY_VARIANT_DECODE_FUNCTION(Tensor, "tensorflow::Tensor");

bool TensorBuffer::GetAllocatedBytes(size_t* out_bytes) const {
  AllocationDescription allocation_description;
  FillAllocationDescription(&allocation_description);
  if (allocation_description.allocated_bytes() > 0) {
    *out_bytes = allocation_description.allocated_bytes();
    return true;
```

```
  } else {
    return false;
  }
}

namespace {

// An un-templated base class for Buffer.
class BufferBase : public TensorBuffer {
 public:
  explicit BufferBase(Allocator* alloc, void* data_ptr)
      : TensorBuffer(data_ptr), alloc_(alloc) {}

  TensorBuffer* root_buffer() override { return this; }

  bool GetAllocatedBytes(size_t* out_bytes) const override {
    if (alloc_->TracksAllocationSizes()) {
      *out_bytes = alloc_->AllocatedSize(data());
      return *out_bytes > 0;
    } else {
      return false;
    }
  }

  void FillAllocationDescription(AllocationDescription* proto) const override {
    void* data_ptr = data();
    int64_t rb = size();
    proto->set_requested_bytes(rb);
    proto->set_allocator_name(alloc_->Name());
    proto->set_ptr(reinterpret_cast<uintptr_t>(data_ptr));
    if (alloc_->TracksAllocationSizes()) {
      int64_t ab = alloc_->AllocatedSize(data_ptr);
      proto->set_allocated_bytes(ab);
      int64_t id = alloc_->AllocationId(data_ptr);
      if (id > 0) {
        proto->set_allocation_id(id);
      }
      if (RefCountIsOne()) {
        proto->set_has_single_reference(true);
      }
    }
  }

 protected:
  void RecordDeallocation() {
    LogMemory::RecordTensorDeallocation(alloc_->AllocationId(data()),
                                        alloc_->Name());
  }

```

```
  Allocator* const alloc_;
};

// Typed ref-counted buffer: T[n].
template <typename T>
class Buffer : public BufferBase {
 public:
  Buffer(Allocator* a, int64_t n);
  Buffer(Allocator* a, int64_t n, const AllocationAttributes& allocation_attr);

  size_t size() const override { return sizeof(T) * elem_; }

 private:
  int64_t elem_;

  ~Buffer() override;

  TF_DISALLOW_COPY_AND_ASSIGN(Buffer);
};

void LogUnexpectedSize(int64_t actual, int64_t expected) {
  LOG(ERROR) << "Input size was " << actual << " and expected " << expected;
}

bool MemoryLoggingEnabled() {
  static bool memory_logging_enabled = LogMemory::IsEnabled();
  return memory_logging_enabled;
}

// A set of helper functions depending on T.
template <typename T>
struct Helper {
  // By default, we assume T is a simple type (float, int32, etc.)
  static_assert(is_simple_type<T>::value, "T is not a simple type.");
  typedef protobuf::RepeatedField<T> RepeatedFieldType;

  // Encoder of simple type T to a string.  We do a copy.
  template <typename Destination>
  static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
    DCHECK_EQ(in->size(), sizeof(T) * n);
    port::AssignRefCounted(StringPiece(in->base<const char>(), in->size()), in,
                           out);
  }

  // Decoder of simple type T. Copy the bytes from "in" into the
  // tensor buffer.
  template <typename Source>
  static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
    if (in.size() != sizeof(T) * n) {
```

```
      LogUnexpectedSize(in.size(), sizeof(T) * n);
      return nullptr;
    }
    Buffer<T>* buf = new Buffer<T>(a, n);
    char* data = buf->template base<char>();
    if (data == nullptr) {
      buf->Unref();
      return nullptr;
    }
    port::CopyToArray(in, data);
    return buf;
  }

  // Memory usage.
  static int64_t TotalBytes(TensorBuffer* in, int64_t n) {
    DCHECK_EQ(in->size(), sizeof(T) * n);
    return in->size();
  }
};

// Helper specialization for string (the only non-simple type we
// support).
template <>
struct Helper<tstring> {
  // Proto message uses RepeatedFieldType to hold repeated T.
  typedef protobuf::RepeatedPtrField<string> RepeatedFieldType;

  // Encodes "n" elements of type string stored in "in" into Cord
  // "out", which is usually the TensorProto::tensor_content.
  template <typename Destination>
  static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
    port::EncodeStringList(in->base<const tstring>(), n, out);
  }

  // Decodes "n" elements of type string from "in" and constructs a
  // buffer out of it. Returns nullptr if the decoding fails. "in" is
  // usually the TensorProto::tensor_content.
  template <typename Source>
  static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
    Buffer<tstring>* buf = new Buffer<tstring>(a, n);
    tstring* strings = buf->template base<tstring>();
    if (strings == nullptr || !port::DecodeStringList(in, strings, n)) {
      buf->Unref();
      return nullptr;
    }
    return buf;
  }

  // Returns the estimated memory usage of "n" elements of type T
```

```
  // stored in buffer "in".
  static int64_t TotalBytes(TensorBuffer* in, int n) {
    int64_t tot = in->size();
    DCHECK_EQ(tot, sizeof(tstring) * n);
    const tstring* p = in->base<const tstring>();
    for (int i = 0; i < n; ++i, ++p) tot += p->size();
    return tot;
  }
};

template <>
struct Helper<ResourceHandle> {
  // Proto message uses RepeatedFieldType to hold repeated T.
  typedef protobuf::RepeatedPtrField<string> RepeatedFieldType;

  // Encodes "n" elements of type ResourceHandle stored in "in" into destination
  // "out", which is usually the TensorProto::tensor_content.
  template <typename Destination>
  static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
    EncodeResourceHandleList(in->base<const ResourceHandle>(), n,
                             port::NewStringListEncoder(out));
  }

  // Decodes "n" elements of type string from "in" and constructs a
  // buffer out of it. Returns nullptr if the decoding fails. "in" is
  // usually the TensorProto::tensor_content.
  template <typename Source>
  static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
    auto* buf = new Buffer<ResourceHandle>(a, n);
    ResourceHandle* ps = buf->template base<ResourceHandle>();
    if (ps == nullptr ||
        !DecodeResourceHandleList(port::NewStringListDecoder(in), ps, n)) {
      buf->Unref();
      return nullptr;
    }
    return buf;
  }

  // Returns the estimated memory usage of "n" elements of type T
  // stored in buffer "in".
  static int64_t TotalBytes(TensorBuffer* in, int n) {
    return n * sizeof(ResourceHandle);
  }
};

template <>
struct Helper<Variant> {
  // Encodes "n" elements of type Variant stored in "in" into destination
  // "out", which is usually the TensorProto::tensor_content.
```

```
  template <typename Destination>
  static void Encode(TensorBuffer* in, int64_t n, Destination* out) {
    EncodeVariantList(in->base<const Variant>(), n,
                      port::NewStringListEncoder(out));
  }

  // Decodes "n" elements of type Variant from "in" and constructs a
  // buffer out of it. Returns nullptr if the decoding fails. "in" is
  // usually the TensorProto::tensor_content.
  template <typename Source>
  static TensorBuffer* Decode(Allocator* a, const Source& in, int64_t n) {
    auto* buf = new Buffer<Variant>(a, n);
    Variant* ps = buf->template base<Variant>();
    if (ps == nullptr ||
        !DecodeVariantList(port::NewStringListDecoder(in), ps, n)) {
      buf->Unref();
      return nullptr;
    }
    return buf;
  }

  // Returns the estimated memory usage of "n" elements of type T
  // stored in buffer "in".
  static int64_t TotalBytes(TensorBuffer* in, int n) {
    return n * sizeof(Variant);
  }
};

template <typename T>
struct ProtoHelper {};

// For a C++ type "T" (float, double, int32, etc.), the repeated field
// "N"_val (float_val, int_val, label_val, etc.) of type "F" (float,
// int32, string, etc) in the TensorProto is used for serializing the
// tensor of type "T".
#define PROTO_TRAITS(T, F, N)                                          \
  template <>                                                          \
  struct ProtoHelper<T> {                                              \
    typedef Helper<F>::RepeatedFieldType FieldType;                    \
    static FieldType::const_iterator Begin(const TensorProto& proto) { \
      return proto.N##_val().begin();                                  \
    }                                                                  \
    static size_t NumElements(const TensorProto& proto) {              \
      return proto.N##_val().size();                                   \
    }                                                                  \
    static void Fill(const T* data, size_t n, TensorProto* proto) {    \
      typename ProtoHelper<T>::FieldType copy(data, data + n);         \
      proto->mutable_##N##_val()->Swap(&copy);                         \
    }                                                                  \
```

```
  };
PROTO_TRAITS(float, float, float);
PROTO_TRAITS(double, double, double);
PROTO_TRAITS(int32, int32, int);
PROTO_TRAITS(uint8, int32, int);
PROTO_TRAITS(uint16, int32, int);
PROTO_TRAITS(uint32, uint32, uint32);
PROTO_TRAITS(int16, int32, int);
PROTO_TRAITS(int8, int32, int);
PROTO_TRAITS(bool, bool, bool);
PROTO_TRAITS(tstring, tstring, string);
PROTO_TRAITS(qint8, int32, int);
PROTO_TRAITS(quint8, int32, int);
PROTO_TRAITS(qint16, int32, int);
PROTO_TRAITS(quint16, int32, int);
#undef PROTO_TRAITS

template <>
struct ProtoHelper<int64_t> {
  static const int64_t* Begin(const TensorProto& proto) {
    return reinterpret_cast<const int64_t*>(proto.int64_val().begin());
  }
  static size_t NumElements(const TensorProto& proto) {
    return proto.int64_val().size();
  }
  static void Fill(const int64_t* data, size_t n, TensorProto* proto) {
    protobuf::RepeatedField<protobuf_int64> copy(data, data + n);
    proto->mutable_int64_val()->Swap(&copy);
  }
};

template <>
struct ProtoHelper<uint64> {
  static const uint64* Begin(const TensorProto& proto) {
    return reinterpret_cast<const uint64*>(proto.uint64_val().begin());
  }
  static size_t NumElements(const TensorProto& proto) {
    return proto.uint64_val().size();
  }
  static void Fill(const uint64* data, size_t n, TensorProto* proto) {
    protobuf::RepeatedField<protobuf_uint64> copy(data, data + n);
    proto->mutable_uint64_val()->Swap(&copy);
  }
};

template <>
struct ProtoHelper<ResourceHandle> {
  static protobuf::RepeatedPtrField<ResourceHandleProto>::const_iterator Begin(
      const TensorProto& proto) {
```

```cpp
    return proto.resource_handle_val().begin();
  }
  static size_t NumElements(const TensorProto& proto) {
    return proto.resource_handle_val().size();
  }
  static void Fill(const ResourceHandle* data, size_t n, TensorProto* proto) {
    auto* handles = proto->mutable_resource_handle_val();
    handles->Clear();
    for (size_t i = 0; i < n; i++) {
      data[i].AsProto(handles->Add());
    }
  }
};

template <>
struct ProtoHelper<Variant> {
  static protobuf::RepeatedPtrField<VariantTensorDataProto>::const_iterator
  Begin(const TensorProto& proto) {
    return proto.variant_val().begin();
  }
  static size_t NumElements(const TensorProto& proto) {
    return proto.variant_val().size();
  }
  static void Fill(const Variant* data, size_t n, TensorProto* proto) {
    auto* variant_values = proto->mutable_variant_val();
    variant_values->Clear();
    for (size_t i = 0; i < n; ++i) {
      VariantTensorData tmp;
      data[i].Encode(&tmp);
      tmp.ToProto(variant_values->Add());
    }
  }
};

template <>
struct ProtoHelper<complex64> {
  typedef Helper<float>::RepeatedFieldType FieldType;
  static const complex64* Begin(const TensorProto& proto) {
    return reinterpret_cast<const complex64*>(proto.scomplex_val().data());
  }
  static size_t NumElements(const TensorProto& proto) {
    return proto.scomplex_val().size() / 2;
  }
  static void Fill(const complex64* data, size_t n, TensorProto* proto) {
    const float* p = reinterpret_cast<const float*>(data);
    FieldType copy(p, p + n * 2);
    proto->mutable_scomplex_val()->Swap(&copy);
  }
};
```

```
template <>
struct ProtoHelper<complex128> {
  typedef Helper<double>::RepeatedFieldType FieldType;
  static const complex128* Begin(const TensorProto& proto) {
    return reinterpret_cast<const complex128*>(proto.dcomplex_val().data());
  }
  static size_t NumElements(const TensorProto& proto) {
    return proto.dcomplex_val().size() / 2;
  }
  static void Fill(const complex128* data, size_t n, TensorProto* proto) {
    const double* p = reinterpret_cast<const double*>(data);
    FieldType copy(p, p + n * 2);
    proto->mutable_dcomplex_val()->Swap(&copy);
  }
};

template <>
struct ProtoHelper<qint32> {
  typedef Helper<int32>::RepeatedFieldType FieldType;
  static const qint32* Begin(const TensorProto& proto) {
    return reinterpret_cast<const qint32*>(proto.int_val().data());
  }
  static size_t NumElements(const TensorProto& proto) {
    return proto.int_val().size();
  }
  static void Fill(const qint32* data, size_t n, TensorProto* proto) {
    const int32* p = reinterpret_cast<const int32*>(data);
    FieldType copy(p, p + n);
    proto->mutable_int_val()->Swap(&copy);
  }
};

template <>
struct ProtoHelper<bfloat16> {
  static void Fill(const bfloat16* data, size_t n, TensorProto* proto) {
    proto->mutable_half_val()->Reserve(n);
    for (size_t i = 0; i < n; ++i) {
      proto->mutable_half_val()->AddAlreadyReserved(
          Eigen::numext::bit_cast<uint16>(data[i]));
    }
  }
};

template <>
struct ProtoHelper<Eigen::half> {
  static void Fill(const Eigen::half* data, size_t n, TensorProto* proto) {
    proto->mutable_half_val()->Reserve(n);
    for (size_t i = 0; i < n; ++i) {
```

```
      proto->mutable_half_val()->AddAlreadyReserved(
          Eigen::numext::bit_cast<uint16>(data[i]));
    }
  }
};

template <typename T>
Buffer<T>::Buffer(Allocator* a, int64_t n)
    : BufferBase(a, TypedAllocator::Allocate<T>(a, n, AllocationAttributes())),
      elem_(n) {}

template <typename T>
Buffer<T>::Buffer(Allocator* a, int64_t n,
                  const AllocationAttributes& allocation_attr)
    : BufferBase(a, TypedAllocator::Allocate<T>(a, n, allocation_attr)),
      elem_(n) {}

template <typename T>
Buffer<T>::~Buffer() {
  if (data()) {
    if (MemoryLoggingEnabled()) {
      RecordDeallocation();
    }
    TypedAllocator::Deallocate<T>(alloc_, static_cast<T*>(data()), elem_);
  }
}

// Allocates a T[n] buffer. Fills in the buffer with repeated values
// in "in".  If "in" has less values than "n", fills the rest of T[n]
// with the last value. If "in" has no values, fills T[n] with the
// default value for T.
//
// This routine is using the typed fields (float_val, etc.) in the
// tensor proto as opposed to the untyped binary representation
// (tensor_content). This is used when we expect the TensorProto is
// used by a client program which may not know how to encode a tensor
// in the compact binary representation.
template <typename T>
TensorBuffer* FromProtoField(Allocator* a, const TensorProto& in, int64_t n) {
  CHECK_GT(n, 0);
  Buffer<T>* buf = new Buffer<T>(a, n);
  T* data = buf->template base<T>();
  if (data == nullptr) {
    buf->Unref();
    return nullptr;
  }

  const int64_t in_n = ProtoHelper<T>::NumElements(in);
  if (in_n <= 0) {
```

```
      std::fill_n(data, n, T());
    } else {
      auto begin = ProtoHelper<T>::Begin(in);
      if (n <= in_n) {
        std::copy_n(begin, n, data);
      } else {
        std::copy_n(begin, in_n, data);
        if (std::is_trivially_copyable<T>::value) {
          const T last = *(data + in_n - 1);
          std::fill_n(data + in_n, n - in_n, last);
        } else {
          const T& last = *(data + in_n - 1);
          std::fill_n(data + in_n, n - in_n, last);
        }
      }
    }

    return buf;
}

template <>
TensorBuffer* FromProtoField<Variant>(Allocator* a, const TensorProto& in,
                                      int64_t n) {
  CHECK_GT(n, 0);
  Buffer<Variant>* buf = new Buffer<Variant>(a, n);
  Variant* data = buf->template base<Variant>();
  if (data == nullptr) {
    buf->Unref();
    return nullptr;
  }
  const int64_t in_n = ProtoHelper<Variant>::NumElements(in);
  if (in_n <= 0) {
    std::fill_n(data, n, Variant());
  } else {
    // If tensor shape says we have n < in_n elements in the output tensor
    // then make sure to only decode the first n out of the in_n elements in the
    // in tensors. In all other cases, we decode all in_n elements of in and set
    // the remaining elements up to n to be the default Variant() value.
    const int64_t real_n = n < in_n ? n : in_n;
    for (int64_t i = 0; i < real_n; ++i) {
      data[i] = in.variant_val(i);
      if (!DecodeUnaryVariant(&data[i])) {
        LOG(ERROR) << "Could not decode variant with type_name: \""
                   << data[i].TypeName()
                   << "\".  Perhaps you forgot to register a "
                      "decoder via REGISTER_UNARY_VARIANT_DECODE_FUNCTION?";
        buf->Unref();
        return nullptr;
      }
```

```cpp
    }
    for (int64_t i = in_n; i < n; ++i) {
      data[i] = Variant();
    }
  }
  return buf;
}

// fp16 and bfloat16 are opaque to the protobuf, so we deserialize these
// identical to uint16 but with data stored in half_val instead of int_val (ie.,
// we don't use ProtoHelper<uint16>).
template <>
TensorBuffer* FromProtoField<Eigen::half>(Allocator* a, const TensorProto& in,
                                          int64_t n) {
  CHECK_GT(n, 0);
  Buffer<Eigen::half>* buf = new Buffer<Eigen::half>(a, n);
  uint16* data = buf->template base<uint16>();
  if (data == nullptr) {
    buf->Unref();
    return nullptr;
  }
  const int64_t in_n = in.half_val().size();
  auto begin = in.half_val().begin();
  if (n <= in_n) {
    std::copy_n(begin, n, data);
  } else if (in_n > 0) {
    std::copy_n(begin, in_n, data);
    const uint16 last = *(data + in_n - 1);
    std::fill_n(data + in_n, n - in_n, last);
  } else {
    std::fill_n(data, n, 0);
  }
  return buf;
}

template <>
TensorBuffer* FromProtoField<bfloat16>(Allocator* a, const TensorProto& in,
                                       int64_t n) {
  CHECK_GT(n, 0);
  Buffer<bfloat16>* buf = new Buffer<bfloat16>(a, n);
  uint16* data = buf->template base<uint16>();
  if (data == nullptr) {
    buf->Unref();
    return nullptr;
  }
  const int64_t in_n = in.half_val().size();
  auto begin = in.half_val().begin();
  if (n <= in_n) {
    std::copy_n(begin, n, data);
```

```
  } else if (in_n > 0) {
    std::copy_n(begin, in_n, data);
    const uint16 last = *(data + in_n - 1);
    std::fill_n(data + in_n, n - in_n, last);
  } else {
    std::fill_n(data, n, 0);
  }
  return buf;
}

// Copies T[n] stored in the buffer "in" into the repeated field in
// "out" corresponding to type T.
template <typename T>
void ToProtoField(const TensorBuffer& in, int64_t n, TensorProto* out) {
  const T* data = in.base<const T>();
  // NOTE: T may not the same as
  // ProtoHelper<T>::FieldType::value_type.  E.g., T==int16,
  // ProtoHelper<T>::FieldType::value_type==int32.  If performance is
  // critical, we can specialize T=float and do memcpy directly.
  ProtoHelper<T>::Fill(data, n, out);
}

void RefIfNonNull(core::RefCounted* buf) {
  if (buf) buf->Ref();
}

void UnrefIfNonNull(core::RefCounted* buf) {
  if (buf) buf->Unref();
}

}  // end namespace

Tensor::Tensor() : Tensor(DT_FLOAT) {}

Tensor::Tensor(DataType type) : shape_(type), buf_(nullptr) {}

Tensor::Tensor(DataType type, const TensorShape& shape, TensorBuffer* buf)
    : shape_(shape), buf_(buf) {
  set_dtype(type);
  RefIfNonNull(buf);
}

Tensor::Tensor(DataType type, TensorShape shape,
               core::RefCountPtr<TensorBuffer> buf)
    : shape_(std::move(shape)), buf_(buf.release()) {
  set_dtype(type);
}

bool Tensor::IsInitialized() const {
```

```
  return (buf_ != nullptr && buf_->data() != nullptr) ||
         shape_.num_elements() == 0;
}

void Tensor::CheckType(DataType expected_dtype) const {
  CHECK_EQ(dtype(), expected_dtype)
      << " " << DataTypeString(expected_dtype) << " expected, got "
      << DataTypeString(dtype());
}

void Tensor::CheckTypeAndIsAligned(DataType expected_dtype) const {
  CHECK_EQ(dtype(), expected_dtype)
      << " " << DataTypeString(expected_dtype) << " expected, got "
      << DataTypeString(dtype());
  CHECK(IsAligned()) << "ptr = " << base<void>();
}

void Tensor::CheckIsAlignedAndSingleElement() const {
  CHECK(IsAligned()) << "Aligned and single element";
  CHECK_EQ(1, NumElements()) << "Must have a one element tensor";
}

Tensor::~Tensor() { UnrefIfNonNull(buf_); }

Status Tensor::BitcastFrom(const Tensor& other, DataType dtype,
                           const TensorShape& shape) {
  int in_size = DataTypeSize(other.dtype());
  int out_size = DataTypeSize(dtype);
  if (in_size == 0) {
    return errors::InvalidArgument("other tensor has zero-sized data type");
  }
  if (out_size == 0) {
    return errors::InvalidArgument("specified output type is zero-sized");
  }
  if (shape.num_elements() * out_size !=
      other.shape().num_elements() * in_size) {
    return errors::InvalidArgument(
        "input and output shapes/data type sizes are not compatible");
  }
  shape_ = shape;
  shape_.set_data_type(dtype);
  if (buf_ != other.buf_) {
    UnrefIfNonNull(buf_);
    buf_ = other.buf_;
    RefIfNonNull(buf_);
  }
  return Status::OK();
}

```

```
// Notice that buf_ either points to a regular TensorBuffer or a SubBuffer.
// For the latter case, we have to make sure that the refcount is
// one both for the SubBuffer _and_ the underlying TensorBuffer.
bool Tensor::RefCountIsOne() const {
  return buf_ != nullptr && buf_->RefCountIsOne() &&
         buf_->root_buffer()->RefCountIsOne() && buf_->OwnsMemory();
}

// The macro CASES() expands to a switch statement conditioned on
// TYPE_ENUM. Each case expands the STMTS after a typedef for T.
#define SINGLE_ARG(...) __VA_ARGS__
#define CASE(TYPE, STMTS)               \
  case DataTypeToEnum<TYPE>::value: {   \
    typedef TF_ATTRIBUTE_UNUSED TYPE T; \
    STMTS;                              \
    break;                              \
  }
#define CASES_WITH_DEFAULT(TYPE_ENUM, STMTS, INVALID, DEFAULT) \
  switch (TYPE_ENUM) {                                         \
    CASE(float, SINGLE_ARG(STMTS))                             \
    CASE(double, SINGLE_ARG(STMTS))                            \
    CASE(int32, SINGLE_ARG(STMTS))                             \
    CASE(uint8, SINGLE_ARG(STMTS))                             \
    CASE(uint16, SINGLE_ARG(STMTS))                            \
    CASE(uint32, SINGLE_ARG(STMTS))                            \
    CASE(uint64, SINGLE_ARG(STMTS))                            \
    CASE(int16, SINGLE_ARG(STMTS))                             \
    CASE(int8, SINGLE_ARG(STMTS))                              \
    CASE(tstring, SINGLE_ARG(STMTS))                           \
    CASE(complex64, SINGLE_ARG(STMTS))                         \
    CASE(complex128, SINGLE_ARG(STMTS))                        \
    CASE(int64_t, SINGLE_ARG(STMTS))                           \
    CASE(bool, SINGLE_ARG(STMTS))                              \
    CASE(qint32, SINGLE_ARG(STMTS))                            \
    CASE(quint8, SINGLE_ARG(STMTS))                            \
    CASE(qint8, SINGLE_ARG(STMTS))                             \
    CASE(quint16, SINGLE_ARG(STMTS))                           \
    CASE(qint16, SINGLE_ARG(STMTS))                            \
    CASE(bfloat16, SINGLE_ARG(STMTS))                          \
    CASE(Eigen::half, SINGLE_ARG(STMTS))                       \
    CASE(ResourceHandle, SINGLE_ARG(STMTS))                    \
    CASE(Variant, SINGLE_ARG(STMTS))                           \
    case DT_INVALID:                                           \
      INVALID;                                                 \
      break;                                                   \
    default:                                                   \
      DEFAULT;                                                 \
      break;                                                   \
  }
```

```
#define CASES(TYPE_ENUM, STMTS)                                      \
  CASES_WITH_DEFAULT(TYPE_ENUM, STMTS, LOG(FATAL) << "Type not set"; \
                     , LOG(FATAL) << "Unexpected type: " << TYPE_ENUM;)

Tensor::Tensor(Allocator* a, DataType type, const TensorShape& shape)
    : shape_(shape), buf_(nullptr) {
  set_dtype(type);
  CHECK_NOTNULL(a);
  if (shape_.num_elements() > 0 || a->AllocatesOpaqueHandle()) {
    CASES(type, buf_ = new Buffer<T>(a, shape.num_elements()));
  }
  if (MemoryLoggingEnabled() && buf_ != nullptr && buf_->data() != nullptr) {
    LogMemory::RecordTensorAllocation("Unknown", LogMemory::UNKNOWN_STEP_ID,
                                      *this);
  }
}

Tensor::Tensor(Allocator* a, DataType type, const TensorShape& shape,
               const AllocationAttributes& allocation_attr)
    : shape_(shape), buf_(nullptr) {
  set_dtype(type);
  CHECK_NOTNULL(a);
  if (shape_.num_elements() > 0 || a->AllocatesOpaqueHandle()) {
    CASES(type, buf_ = new Buffer<T>(a, shape.num_elements(), allocation_attr));
  }
  if (MemoryLoggingEnabled() && !allocation_attr.allocation_will_be_logged &&
      buf_ != nullptr && buf_->data() != nullptr) {
    LogMemory::RecordTensorAllocation("Unknown (with attributes)",
                                      LogMemory::UNKNOWN_STEP_ID, *this);
  }
}

Status Tensor::BuildTensor(DataType type, const TensorShape& shape,
                           Tensor* out_tensor) {
  // Avoid crashes due to invalid or unsupported types.
  CASES_WITH_DEFAULT(
      type, {}, return errors::InvalidArgument("Type not set"),
      return errors::InvalidArgument("Unexpected type: ", DataType_Name(type)));
  *out_tensor = Tensor(type, shape);
  return Status::OK();
}

// NOTE(mrry): The default allocator for a Tensor (when none is specified) is
// the default CPU allocator for NUMA zone 0. Accessing that currently involves
// acquiring a lock, which guards initialization of the per-NUMA zone
// allocators, and becomes highly contended.
//
// Note also that it would be better if all Tensor allocations required the user
```

```
// to specify an allocator, for purposes of accounting, etc. However, the
// default allocator is widely used throughout the codebase and in client code.
static Allocator* get_default_cpu_allocator() {
  static Allocator* default_cpu_allocator =
      cpu_allocator(port::kNUMANoAffinity);
  return default_cpu_allocator;
}

Tensor::Tensor(DataType type, const TensorShape& shape)
    : Tensor(get_default_cpu_allocator(), type, shape) {}

bool Tensor::HostScalarTensorBufferBase::GetAllocatedBytes(
    size_t* out_bytes) const {
  // `this->FillAllocationDescription()` never sets allocated bytes information,
  // so we can short-circuit the construction of an `AllocationDescription`.
  return false;
}

void Tensor::HostScalarTensorBufferBase::FillAllocationDescription(
    AllocationDescription* proto) const {
  proto->set_requested_bytes(size());
  proto->set_allocator_name("HostScalarTensorBuffer");
  proto->set_ptr(reinterpret_cast<uintptr_t>(data()));
}

template <typename T>
class SubBuffer : public TensorBuffer {
 public:
  // This buffer is an alias to buf[delta, delta + n).
  SubBuffer(TensorBuffer* buf, int64_t delta, int64_t n)
      : TensorBuffer(buf->base<T>() + delta),
        root_(buf->root_buffer()),
        elem_(n) {
    // Sanity check. The caller should ensure the sub buffer is valid.
    CHECK_LE(root_->base<T>(), this->base<T>());
    T* root_limit = root_->base<T>() + root_->size() / sizeof(T);
    CHECK_LE(this->base<T>(), root_limit);
    CHECK_LE(this->base<T>() + n, root_limit);
    // Hold a ref of the underlying root buffer.
    // NOTE: 'buf' is a sub-buffer inside the 'root_' buffer.
    root_->Ref();
  }

  size_t size() const override { return sizeof(T) * elem_; }
  TensorBuffer* root_buffer() override { return root_; }
  bool GetAllocatedBytes(size_t* out_bytes) const override {
    return root_->GetAllocatedBytes(out_bytes);
  }
  void FillAllocationDescription(AllocationDescription* proto) const override {
```

```
    root_->FillAllocationDescription(proto);
  }

 private:
  TensorBuffer* root_;
  int64_t elem_;

  ~SubBuffer() override { root_->Unref(); }

  TF_DISALLOW_COPY_AND_ASSIGN(SubBuffer);
};

Tensor Tensor::Slice(int64_t start, int64_t limit) const {
  CHECK_GE(dims(), 1);
  CHECK_LE(0, start);
  CHECK_LE(start, limit);
  int64_t dim0_size = shape_.dim_size(0);
  CHECK_LE(limit, dim0_size);
  if ((start == 0) && (limit == dim0_size)) {
    return *this;
  }
  Tensor ret;
  ret.shape_ = shape_;
  ret.set_dtype(dtype());
  ret.buf_ = nullptr;
  if (dim0_size > 0) {
    const int64_t elems_per_dim0 = NumElements() / dim0_size;
    const int64_t delta = start * elems_per_dim0;
    dim0_size = limit - start;
    ret.shape_.set_dim(0, dim0_size);
    const int64_t num_elems = dim0_size * elems_per_dim0;
    if (buf_) {
      DataType dt = dtype();
      CASES(dt, ret.buf_ = new SubBuffer<T>(buf_, delta, num_elems));
    }
  }
  return ret;
}

Tensor Tensor::SubSlice(int64_t index) const {
  CHECK_GE(dims(), 1);  // Crash ok.
  CHECK_LE(0, index);   // Crash ok.
  int64_t dim0_size = shape_.dim_size(0);
  CHECK_LE(index, dim0_size);  // Crash ok.
  Tensor ret;
  ret.shape_ = shape_;
  ret.shape_.RemoveDim(0);
  ret.set_dtype(dtype());
  ret.buf_ = nullptr;
```

```
  if (dim0_size > 0) {
    const int64_t elems_per_dim0 = NumElements() / dim0_size;
    const int64_t delta = index * elems_per_dim0;
    const int64_t num_elems = elems_per_dim0;
    if (buf_) {
      DataType dt = dtype();
      CASES(dt, ret.buf_ = new SubBuffer<T>(buf_, delta, num_elems));
    }
  }
  return ret;
}

bool Tensor::FromProto(const TensorProto& proto) {
  return FromProto(get_default_cpu_allocator(), proto);
}

bool Tensor::FromProto(Allocator* a, const TensorProto& proto) {
  CHECK_NOTNULL(a);
  TensorBuffer* p = nullptr;
  if (!TensorShape::IsValid(proto.tensor_shape())) return false;
  if (proto.dtype() == DT_INVALID) return false;
  TensorShape shape(proto.tensor_shape());
  const int64_t N = shape.num_elements();
  if (N > 0 && proto.dtype()) {
    bool dtype_error = false;
    if (!proto.tensor_content().empty()) {
      const auto& content = proto.tensor_content();
      CASES_WITH_DEFAULT(proto.dtype(), p = Helper<T>::Decode(a, content, N),
                         dtype_error = true, dtype_error = true);
    } else {
      CASES_WITH_DEFAULT(proto.dtype(), p = FromProtoField<T>(a, proto, N),
                         dtype_error = true, dtype_error = true);
    }
    if (dtype_error || p == nullptr) return false;
  }
  shape_ = shape;
  set_dtype(proto.dtype());
  UnrefIfNonNull(buf_);
  buf_ = p;
  // TODO(misard) add tracking of which kernels and steps are calling
  // FromProto.
  if (MemoryLoggingEnabled() && buf_ != nullptr && buf_->data() != nullptr) {
    LogMemory::RecordTensorAllocation("Unknown (from Proto)",
                                      LogMemory::UNKNOWN_STEP_ID, *this);
  }
  return true;
}

void Tensor::AsProtoField(TensorProto* proto) const {
```

```
  proto->Clear();
  shape_.AsProto(proto->mutable_tensor_shape());
  proto->set_dtype(dtype());
  if (buf_) {
    CASES(dtype(), ToProtoField<T>(*buf_, shape_.num_elements(), proto));
  }
}

void Tensor::AsProtoTensorContent(TensorProto* proto) const {
  proto->Clear();
  proto->set_dtype(dtype());
  shape_.AsProto(proto->mutable_tensor_shape());
  if (buf_) {
    CASES(dtype(), Helper<T>::Encode(buf_, shape_.num_elements(),
                                     proto->mutable_tensor_content()));
  }
}

size_t Tensor::TotalBytes() const {
  if (shape_.num_elements() == 0) return 0;
  CHECK(buf_) << "null buf_ with non-zero shape size " << shape_.num_elements();
  CASES(dtype(), return Helper<T>::TotalBytes(buf_, shape_.num_elements()));
  return 0;  // Makes compiler happy.
}

size_t Tensor::AllocatedBytes() const {
  if (buf_) {
    size_t ret;
    if (buf_->GetAllocatedBytes(&ret)) {
      return ret;
    }
  }
  return TotalBytes();
}

bool Tensor::CanUseDMA() const {
  CASES(dtype(), return is_simple_type<T>::value);
  return false;  // Makes compiler happy.
}

#undef CASES
#undef CASE

namespace {

// StrCat and StrAppend don't support Eigen::half directly at the moment, and
// we would like to keep them compatible with their absl counterparts, for ease
// of migration. We could rely on errors::internal::PrepareForStrCat() but the
// logic is so simple we can just replicate it here, where it is close to its
```

```
// usage and easy to change later. And there's the extra benefit of not
// accessing an 'internal' namespace.
inline const strings::AlphaNum& PrintOneElement(const strings::AlphaNum& a,
                                                bool print_v2) {
  return a;
}
inline string PrintOneElement(const tstring& a, bool print_v2) {
  if (print_v2) {
    return "\"" + absl::Utf8SafeCEscape(a) + "\"";
  } else {
    return absl::Utf8SafeCEscape(a);
  }
}
inline float PrintOneElement(const Eigen::half& h, bool print_v2) {
  return static_cast<float>(h);
}

inline float PrintOneElement(bfloat16 f, bool print_v2) {
  return static_cast<float>(f);
}

// Print from left dim to right dim recursively.
template <typename T>
void PrintOneDim(int dim_index, const gtl::InlinedVector<int64, 4>& shape,
                 int64_t limit, int shape_size, const T* data,
                 int64_t* data_index, string* result) {
  if (*data_index >= limit) return;
  int64_t element_count = shape[dim_index];
  // We have reached the right-most dimension of the tensor.
  if (dim_index == shape_size - 1) {
    for (int64_t i = 0; i < element_count; i++) {
      if (*data_index >= limit) {
        // If not enough elements has been printed, append "...".
        if (dim_index != 0) {
          strings::StrAppend(result, "...");
        }
        return;
      }
      if (i > 0) strings::StrAppend(result, " ");
      strings::StrAppend(result, PrintOneElement(data[(*data_index)++], false));
    }
    return;
  }
  // Loop every element of one dim.
  for (int64_t i = 0; i < element_count; i++) {
    bool flag = false;
    if (*data_index < limit) {
      strings::StrAppend(result, "[");
      flag = true;
```

```
    }
    // As for each element, print the sub-dim.
    PrintOneDim(dim_index + 1, shape, limit, shape_size, data, data_index,
                result);
    if (*data_index < limit || flag) {
      strings::StrAppend(result, "]");
      flag = false;
    }
  }
}

// Appends the spacing between elements for a given dim onto a result string
void PrintDimSpacing(int dim_index, int num_dims, string* result) {
  if (dim_index == num_dims - 1) {
    strings::StrAppend(result, " ");
    return;
  }
  for (int j = 0; j < num_dims - dim_index - 1; j++) {
    strings::StrAppend(result, "\n");
  }
  for (int j = 0; j <= dim_index; j++) {
    strings::StrAppend(result, " ");
  }
}

// Print from left dim to right dim recursively.
template <typename T>
void PrintOneDimV2(int dim_index, const gtl::InlinedVector<int64, 4>& shape,
                   int64_t num_elts_at_ends, int num_dims, const T* data,
                   int64_t data_index, string* result) {
  // We have recursed beyond all the dimensions into a single element
  // of the tensor.
  if (dim_index == num_dims) {
    strings::StrAppend(result, PrintOneElement(data[data_index], true));
    return;
  }

  strings::StrAppend(result, "[");
  int64_t element_count = shape[dim_index];
  int64_t start_of_end =
      std::max(num_elts_at_ends, element_count - num_elts_at_ends);

  // Loop every element of one dim.
  int64_t elements_per_iter = 1;
  for (int i = dim_index + 1; i < num_dims; i++) {
    elements_per_iter *= shape[i];
  }
  for (int64_t i = 0; (i < num_elts_at_ends) && (i < element_count); i++) {
    if (i > 0) {
```

```
      PrintDimSpacing(dim_index, num_dims, result);
    }

    // As for each element, print the sub-dim.
    PrintOneDimV2(dim_index + 1, shape, num_elts_at_ends, num_dims, data,
                  data_index + elements_per_iter * i, result);
  }
  if (element_count > 2 * num_elts_at_ends) {
    PrintDimSpacing(dim_index, num_dims, result);
    strings::StrAppend(result, "...");
  }
  for (int64_t i = start_of_end; i < element_count; i++) {
    // As for each element, print the sub-dim.
    PrintDimSpacing(dim_index, num_dims, result);
    PrintOneDimV2(dim_index + 1, shape, num_elts_at_ends, num_dims, data,
                  data_index + elements_per_iter * i, result);
  }

  strings::StrAppend(result, "]");
}

template <typename T>
string SummarizeArray(int64_t limit, int64_t num_elts,
                      const TensorShape& tensor_shape, const char* data,
                      const bool print_v2) {
  string ret;
  const T* array = reinterpret_cast<const T*>(data);

  const gtl::InlinedVector<int64_t, 4> shape = tensor_shape.dim_sizes();
  if (shape.empty()) {
    for (int64_t i = 0; i < limit; ++i) {
      if (i > 0) strings::StrAppend(&ret, " ");
      strings::StrAppend(&ret, PrintOneElement(array[i], print_v2));
    }
    if (num_elts > limit) strings::StrAppend(&ret, "...");
    return ret;
  }
  if (print_v2) {
    const int num_dims = tensor_shape.dims();
    PrintOneDimV2(0, shape, limit, num_dims, array, 0, &ret);
  } else {
    int64_t data_index = 0;
    const int shape_size = tensor_shape.dims();
    PrintOneDim(0, shape, limit, shape_size, array, &data_index, &ret);

    if (num_elts > limit) strings::StrAppend(&ret, "...");
  }

  return ret;
```

```
}
}  // namespace

string Tensor::SummarizeValue(int64_t max_entries, bool print_v2) const {
  const int64_t num_elts = NumElements();
  if (max_entries < 0) {
    max_entries = num_elts;
  }
  size_t limit = std::min(max_entries, num_elts);
  if ((limit > 0) && (buf_ == nullptr)) {
    return strings::StrCat("uninitialized Tensor of ", num_elts,
                           " elements of type ", dtype());
  }
  const char* data = limit > 0 ? tensor_data().data() : nullptr;
  switch (dtype()) {
    case DT_BFLOAT16:
      return SummarizeArray<bfloat16>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_HALF:
      return SummarizeArray<Eigen::half>(limit, num_elts, shape_, data,
                                         print_v2);
      break;
    case DT_FLOAT:
      return SummarizeArray<float>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_DOUBLE:
      return SummarizeArray<double>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_UINT32:
      return SummarizeArray<uint32>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_INT32:
      return SummarizeArray<int32>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_UINT8:
    case DT_QUINT8:
      return SummarizeArray<uint8>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_UINT16:
    case DT_QUINT16:
      return SummarizeArray<uint16>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_INT16:
    case DT_QINT16:
      return SummarizeArray<int16>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_INT8:
    case DT_QINT8:
      return SummarizeArray<int8>(limit, num_elts, shape_, data, print_v2);
```

```
      break;
    case DT_UINT64:
      return SummarizeArray<uint64>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_INT64:
      return SummarizeArray<int64_t>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_BOOL:
      // TODO(tucker): Is it better to emit "True False..."?  This
      // will emit "1 0..." which is more compact.
      return SummarizeArray<bool>(limit, num_elts, shape_, data, print_v2);
      break;
    case DT_STRING:
      return SummarizeArray<tstring>(limit, num_elts, shape_, data, print_v2);
      break;
    default: {
      // All irregular cases
      string ret;
      if (print_v2 && (dims() > 0)) {
        strings::StrAppend(&ret, "[");
      }
      // TODO(irving): Don't call flat every time around this
      // loop.
      for (size_t i = 0; i < limit; ++i) {
        if (i > 0) strings::StrAppend(&ret, " ");
        switch (dtype()) {
          case DT_VARIANT: {
            const Variant& v = flat<Variant>()(i);
            strings::StrAppend(&ret, "<", v.SummarizeValue(), ">");
          } break;
          case DT_RESOURCE: {
            const ResourceHandle& r = flat<ResourceHandle>()(i);
            strings::StrAppend(&ret, "<", r.SummarizeValue(), ">");
          } break;
          default:
            // TODO(zhifengc, josh11b): Pretty-print other types (bool,
            // complex64, quantized).
            strings::StrAppend(&ret, "?");
        }
      }
      if (max_entries < num_elts) strings::StrAppend(&ret, "...");
      if (print_v2 && (dims() > 0)) {
        strings::StrAppend(&ret, "]");
      }
      return ret;
    }
  }
}

```

```cpp
StringPiece Tensor::tensor_data() const {
  if (buf_ == nullptr) return StringPiece();  // Don't die for empty tensors
  return StringPiece(static_cast<char*>(buf_->data()), TotalBytes());
}

void* Tensor::data() const {
  if (buf_ == nullptr) return nullptr;  // Don't die for empty tensors
  return static_cast<void*>(buf_->data());
}

bool Tensor::SharesBufferWith(const Tensor& b) const {
  return buf_ != nullptr && b.buf_ != nullptr &&
         buf_->root_buffer() == b.buf_->root_buffer();
}

string Tensor::DebugString(int num_values) const {
  return strings::StrCat("Tensor<type: ", DataTypeString(dtype()),
                         " shape: ", shape().DebugString(),
                         " values: ", SummarizeValue(num_values), ">");
}

string Tensor::DeviceSafeDebugString() const {
  return strings::StrCat("Tensor<type: ", DataTypeString(dtype()),
                         " shape: ", shape().DebugString(), ">");
}

void Tensor::FillDescription(TensorDescription* description) const {
  description->set_dtype(dtype());
  shape().AsProto(description->mutable_shape());
  if (buf_ != nullptr && buf_->data() != nullptr) {
    buf_->FillAllocationDescription(
        description->mutable_allocation_description());
  }
}

gtl::InlinedVector<int64_t, 4> Tensor::ComputeFlatInnerDims(
    gtl::ArraySlice<int64_t> orig, int64_t num_out_dims) {
  gtl::InlinedVector<int64_t, 4> out_dims(num_out_dims, 0);
  int64_t offset = orig.size() - num_out_dims;
  for (int64_t out_dim = num_out_dims - 1; out_dim >= 0; --out_dim) {
    const int64_t in_dim = out_dim + offset;
    out_dims[out_dim] = in_dim < 0 ? 1 : orig[in_dim];
  }
  for (int64_t in_dim = 0; in_dim < offset; ++in_dim) {
    out_dims[0] *= orig[in_dim];
  }
  return out_dims;
}

```

```
gtl::InlinedVector<int64_t, 4> Tensor::ComputeFlatOuterDims(
    gtl::ArraySlice<int64_t> orig, int64_t num_out_dims) {
  gtl::InlinedVector<int64_t, 4> out_dims(num_out_dims, 0);
  for (int64_t out_dim = 0; out_dim <= num_out_dims - 1; ++out_dim) {
    out_dims[out_dim] = out_dim >= orig.size() ? 1 : orig[out_dim];
  }
  for (int64_t in_dim = num_out_dims; in_dim < orig.size(); ++in_dim) {
    out_dims[num_out_dims - 1] *= orig[in_dim];
  }
  return out_dims;
}

}  // namespace tensorflow
```