SOFTWARE METAPAPER

The Ocean Tensor Package

Ewout van den Berg

IBM T.J. Watson Research Center, Yorktown Heights, NY, US
evandenberg@us.ibm.com

The Ocean Tensor Package is an open-source package for matrix and tensor operations on CPU and GPU. The package aims to serve as a foundational layer for applications that require dense tensor operations on a variety of device types. All operations are available through a unified interface that is carefully designed to be powerful, extensible, and at the same time easy to use. The package has a modular implementation in C and provides a light-weight Python interface. Modularity of the package facilitates the addition of new operations as well as new device types.

(1) Overview

Introduction

Over the last decade or so, general-purpose GPUs have been successfully used in fields such as medical imaging [13, 17], molecular dynamics simulations [8], radio astronomy [2], data mining [3], graph processing [14], and many others [9]. More recently, GPUs have found widespread use in deep learning. Perhaps more so than in other fields, there has been a proliferation of software packages for deep learning, such as Caffe [7], Torch [6], and Tensorflow [1]. One reason for this is the flexibility required in deep learning to try out different network architectures and data transformations to obtain the best possible model. Given that the data, node parameters, and intermediate results are conveniently expressed in the form of multi-dimensional arrays, or tensors, these packages have seen a gradual shift towards general-purpose compute environments. Despite these advances there is still a lot of room for improvement: existing packages tend to be monolithic, require a large number of external dependencies, and tend to lack in tensor layout flexibility, supported data types, or extensions to new device types. Most importantly, a stand-alone tensor-support package designed to serve as the foundation for a wide range of other applications is still missing. To fill this gap and address some of the shortcomings of existing packages, we propose the Ocean Tensor Package, a modular open-source[1] foundation library for dense tensor operations.

Functionality

The Ocean Tensor Package [16] provides a comprehensive set of tensor operations for CPU and GPU. The functions are available directly as a C library, or through an easy-to-use Python interface. In this section we explore some of the features of the package, illustrated with code excerpts based on the Python interface.

Object types

The user interface to the Ocean Tensor Package exposes several object types, illustrated in **Figure 1**. At the top of the object hierarchy are Tensors and Scalars, each of which has a given data type. Tensors are views on contiguous chunks of memory, stored as Storage objects. Associated with each Storage object is a Stream, which is used to schedule asynchronous operations as well as to maintain inter-stream dependencies. Stream objects themselves are associated with a Device instance (such as CPU or GPU #0) of a certain Device type (such as CPU or GPU).

Devices

Device objects enable the specification of the device to be used when instantiating a Tensor or Storage object. In addition, they provide generic information of the given device, such as the support for byte-swapped data or a list of all currently loaded modules. Depending on the device type, addition information may be available. For instance, on GPU devices it is possible to query numerous properties, including the multiprocessor count, or the currently available amount of free memory. Advanced functions include the instantiation of a new stream, and the specification of the number of intermediate tensor buffers for the device and their maximum size. Ocean maintains a list of available devices, including the `ocean.cpu` device as well as the `ocean.gpu` list of GPU devices, which can be indexed to obtain the desired device instance.

Storage

Storage objects encapsulate a contiguous piece of memory that is either allocated dynamically or provided by an external source. The data type associated with the storage has two main purposes: first as the default data type when instantiating a tensor from the storage without
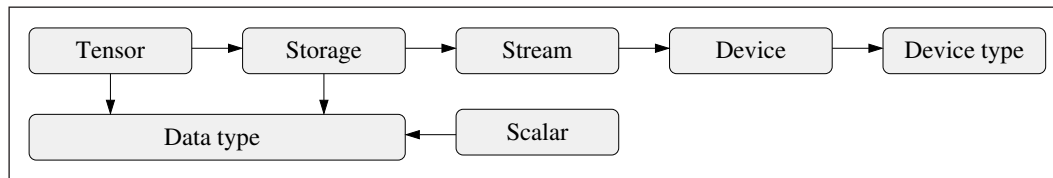
Figure 1: Connection between the main object types in Ocean.

providing a type; and second, for formatting the storage elements for display. The data type of storage objects can be changed freely without affecting the tensors that use it. It is possible to superimpose tensors of different data types on the same storage. One typical example where this happens is when querying the imaginary part of a complex double-precision tensor, which results in an additional tensor view on the storage of type double. There are no restrictions on the number of different tensor types that can share the same storage. Tensor operations use the storage stream for synchronization to avoid race conditions and inconsistencies in the data. Storage data can be marked as read only, which prevents any updates to the data, either directly or through tensors operations (marking storage as read-only is reflected in all derived tensors).

Tensors

Ocean tensors are easy to instantiate on any of the available devices. For example, creation of a 3×4 tensor with single-precision floating-point format on device `gpu[0]` is done simply by writing: `tensor = ocean. tensor([3,4], ocean.float, ocean.gpu[0])`. When the data type or device is omitted, user-specified default values are used. In contrast to some of the existing packages, there is no notion of a currently active device, and no explicit device changes need to be done in order to instantiate new tensors, or perform operations on them. By default, tensors follow a column-major memory layout, but general strides (given in bytes) are supported, thereby allowing compatibility with most Numpy tensors. Two differences with Numpy [11] are (1) the support of the complex half-precision data type in Ocean, and additional data types in Numpy (such as string and datetime), and (2) the maximum number of tensors dimensions. The maximum number of tensor dimensions in Ocean is currently set to eight, but this restriction is easy relaxed or removed; Numpy has a hard-coded maximum of 32 tensor dimensions. Similar to Numpy, Ocean allows tensors on the CPU to have either little and big endian byte order, and tensor operations can be applied in either byte order. The byte order can be easily changed, if needed, either by byte-swapping the elements, or, in case of a mismatch between the flag and the actual byte ordering, by simply specifying the appropriate byte order.

Tensor operations

Tensor operations in Ocean are provided through modules. The Core module forms the basis of Ocean, and includes the definition of the basic object classes as well as the device instances. As the most elementary operation, the Core module supports tensor creation; from storage,

from data in the form of nested lists, sequences, and other tensor types, or without initialization. Aside from tensor creation, the Core module provides an extensive set of elementary functions, including functions for shape and axis manipulation, indexing, copy functions, type and device casting, basic arithmetic operations, trigonometric operations (supported on all real and complex floating-point types), as well as tensor reductions along one or more axes. (A complete list of functions can be found on the Ocean Tensor Package repository [16].) Below we highlight some of the functionalities.

Type casting

The type of a tensor can be seen as the combination of the data type and the device associated with the tensor. Tensors in Ocean have an associated type, and type casting may therefore be necessary at times. Explicit type casting can be using the `ocean.cast` function, which returns a copy of the tensor with the desired type, and the `ocean.ensure` function, which returns a type-cast tensor only if the requested type differs from that of the input. Type casting a tensor `T` with a data type (`ocean.float(T)`) or device instance (`ocean.gpu[1] (T)`) is equivalent to calling the ensure function with only a data type or device update.

Implicit type casting is used in Ocean to ensure that the input arguments to tensor operations have appropriate types and byte order. Consider for instance the tensor addition: `C = A+B`. To avoid implementing addition for all possible type combinations we need to determine the type of `C` based on those of `A` and `B`, and normalize the data types and device accordingly. One way of resolving the device type is to impose a device ordering and choose the device with the highest priority. This requires specification of the order and may result in unexpected results, certainly when the device order could be changed from other parts of the code. Another way is to always use the device from the left-hand side of the operator, or the first parameter in the argument list. In `A+=B` it is clear that `B` should be coerced to the device of `A`, and we therefore do the same for `A+B`. In case it is desirable to use the device of `B` (i.e., `B.device`), it possible in this case to write `B+A`, or to use explicit casting: `ocean.ensure(A,B.device) + B`, or simply `B.device(A) + B`.

For implicit casting of the data types we follow Numpy and use the smallest available data type that can keep both data types. For instance, addition of signed and unsigned 8-bit integers would give a 16-bit signed integer. Since no standard data type is available for quadruple-precision floats, an exception is made for 64-bit integers and floating-point numbers, which result in double-precision floats. Automatic type casting in Ocean is switched on by

default, but can be disabled by the user in case strict type checks are needed. When switched off, an exception is raised whenever a type mismatch is encountered.

Type casting based on the contents of the tensor is desirable, for example, when taking the square root of negative real numbers or the arccosine of scalars with magnitude larger than one. Should such operations result in a not-a-number (NaN) value, return a complex-valued result, or generate an error? The approach taken in Ocean is to add parameters to such operators that indicate the compute mode. In the standard mode no checks on the tensor elements are done and NaN values are generated whenever needed. Checks are made in the warning and error modes, giving respectively a warning or an error when elements with values outside of the operator domain are encountered. Finally, in the complex mode, checks are made to determine whether the resulting data type should be real or complex. If needed, explicit types casting can always be used.

Indexing

Ocean supports a variety of indexing modes along one or more dimensions that can be combined to index a tensor. The basic single-dimension indexing modes are (1) scalars, to index a single element along an axis; (2) ranges, to index a regularly spaced set of elements; and (3) the colon ':' operator to indicate the entire dimension. In addition to the basic modes it is possible to use one or two-dimensional index arrays to select particular elements, by specifying the indices along a single dimension, or tuples of indices along several dimensions. As is customary in Python, negative indices can be used to indicate the index relative to the end of the dimension. Finally, boolean tensors can be used as masks for indexing, with the non-zero elements indicating the elements to be selected. Dimensions that are omitted in the indexing are implicitly indexed with the colon operator, and the ellipsis object '…' can appear once to indicate application of zero or more colon operator in that location to complete the index. When indexing a tensor using only basic indexing modes (either explicitly or implicitly), a view of that tensor is returned in the form of a new tensor that shares the original storage. In all other cases a new tensor is created by copying the indexed elements.

Special preprocessing is needed for index arrays and boolean masks: for index arrays the indices need to be checked for validity; whereas for boolean masks it is necessary to count the number of selected elements, in order to determine the size of the output tensor; and to convert the selected indices into relative offsets into the data buffer of tensor being indexed. When such indices are used repeatedly, computational efforts are wasted in applying the same preprocessing steps for each use. To avoid this situation, Ocean introduces index objects, which are constructed by indexing the `ocean.index` object (ranges and colons are not allowed as parameters in regular function calls). Once an index object is created it can be bound to a tensor size to convert negative indices, check the validity of indices, determine the overlap of index ranges with the given dimensions, and

convert boolean masks to explicit indices. Index objects can subsequently (or directly) be bound to tensor strides, which converts all index arrays and boolean masks to relative offsets within the tensor data. Both bound and unbound index object can be used in exactly the same way as the index modes used to construct it. As such, index objects can be used to create yet other index objects, if needed. When index objects are bound to size, the relevant tensor dimensions must match, and likewise for strides.

Interoperability

The Python interface of Ocean provides for plug-in modules to define external object types for tensors and scalars, and the conversion between these and the corresponding Ocean types. All extended object types provided by the plug-ins are compared against when parsing the tensor operation parameters. This allows them to be used in essentially the same way as Ocean tensors and scalars. As an example, we can declare the Numpy tensor and scalar types by importing `pyOceanNumpy`. Once this is done, it is possible to write expressions such as `A + np.asarray([4,5,6])`, where `A` is an Ocean tensor. Conversion to Numpy can be done using `A.convertTo('numpy')`, where the 'numpy' string is registered by the plug-in. When supported by the external tensor type, a shallow copy of the tensor is made, unless otherwise requested by the user.

Deallocation

Automatic garbage collection in Python can delay the deletion of tensor objects and cause devices to run out of free memory despite careful management by the user. In order to force tensor deletion, it is possible to call the `dealloc` tensor function, which maintains the Python tensor object, but replaces the content by an empty tensor. This frees up any dynamically allocated tensor data, while avoiding problems with accidental usage of the tensor after deallocation.

Existing packages

We now compare some of the features in Ocean with those in other packages. Since most of the packages are in active development, we only discuss the features available at the time of writing.

Numpy [11] is the *de facto* Python package for dense tensor operations. Numpy was written for tensors on CPUs and does not support tensors on any other device types. The more recent CuPy [10] package implements tensors for GPU devices with an interface that closely mirrors that of Numpy, but is otherwise largely independent. Both packages are written as a Python-C API and directly extend Python classes, which limits their usage as stand-alone packages. Moreover, each of these two package supports only a single device type.

A package that supports multiple device types and is written as a general library with separate language bindings is ArrayFire [18]. The same applies to most deep-learning packages. As mentioned in the introduction, tensor operations form the foundation of deep-learning

packages, and we therefore consider some of the most popular ones: Caffe [7], PyTorch [5, 12], TensorFlow [1], and MXNet [4]. All of these packages support tensor operations on multiple device types, and expose at least part of the available tensor operations to the user. Nevertheless, given the focus on deep learning, these packages are not written or intended to serve as stand-alone tensor support packages. In particular, many of the packages define tensor classes with highly domain-specific member functions and variables. For example, the class may provide gradient information with each tensor, or include references to a symbolic compute graph node that contains the tensor.

In **Table 1** we list several properties that we consider to be important in a general-purpose tensor package, and are therefore implemented in Ocean. One of these properties is the support for automatic type casting, as discussed earlier. This feature is supported in Numpy, CuPy, and ArrayFire, but is missing from all the deep-learning packages under consideration. From a developer's point of view it is convenient to have a unified tensor type or class. This is supported by all packages except Caffe, which provides a template class (several of the other packages use templated classes behind the scenes, but provide a unified type in the API). Support for a comprehensive set of data types is clearly important for a tensor package.[2] Coverage between packages differs substantially, and we therefore focus on support for complex data types, which required additional functionality, such as conjugation and access to real and imaginary parts of the tensor. Of the four deep-learning packages considered, only TensorFlow supports complex tensor types (based on single and double-precision floats). These types are also supported by Numpy, CuPy, and ArrayFire. Ocean is the only package that additionally provides a complex data type based on half-precision floats.

The layout of tensors in memory is given by the strides, or distance between successive elements along each of the dimensions. Flexibility in the tensor strides enables features such as broadcasting along dimensions, easy manipulation of axes, and creation of views on regularly indexed subtensors. In addition, it ensures compatibility with a wide range of existing data types for tensors and matrices. Most of the deep-learning packages, as well as ArrayFire, adhere to a contiguous row-major data order, with implicit strides that can be inferred based on the tensor dimensions and the element size of the given data type. PyTorch also uses this data order by default, but

allows users to override the standard layout by specifying tensor strides as nonnegative multiples of the element size. Numpy and CuPy support arbitrary strides. Each of these packages, along with Ocean, implements sorting of axes and merging of consecutive axes, when possible, to increase memory locality and reduce the overhead of iterating over dimensions, both of which help increase the computational efficiency of tensor operations on strided data. Packages that use a contiguous tensor layout can flatten tensors to a single dimension for many operations, such as unary and binary elementwise operations; other operations may require optimizations similar to those mentioned above. ArrayFire limits the number of tensor dimensions to four and often uses explicit nested for-loops with index computation at the innermost loop to traverse the data.

Some of the difficulties that come with the provision for arbitrary strides is that tensors may self overlap in memory. Hence, overlap detection between pairs of tensors becomes non-trivial. For consistent results in computations, such as $A[[1, 2]] = A[[2, 1]]$, good support for overlap detection is essential. Ocean checks for self-overlapping tensors and treats them as read-only for most operations (the semantics of writing different values to the same memory address is not well defined). Overlap detection between pairs of tensors and allocation of intermediate tensors to resolve overlaps is also included. Similar checks are present in PyTorch and Numpy. Overlap detection in TensorFlow is restricted to tensor views.

Aside from Ocean, none of the packages considered in this section defines a clear separation between tensor types and the low-level implementation of the tensor operations. As a result, none of the tensor operations other than those already provided by existing libraries, such as BLAS and cuBLAS, are easily transferable for use in other packages.

Illustrative example

We now illustrate some of the features of the Ocean Tensor Package based on an example QR factorization (see for instance [15]). This is of course only an example; QR factorization should be an integral part of the package, and direct support is planned in a future linear-algebra module. A comprehensive list of functions provided by the core module, as well as a large number of examples can be found in the documentation of the Ocean Tensor Package repository [16].

Table 1: Comparison between different packages providing tensor functionality.

	Numpy	CuPy	Caffe	PyTorch	TensorFlow	MXNet	ArrayFire	Ocean
Multiple device types	✗	✗	✓	✓	✓	✓	✓	✓
Automatic type casting	✓	✓	✗	✗	✗	✗	✓	✓
Unified tensor type	✓	✓	✗	✓	✓	✓	✓	✓
Complex data types	✓	✓	✗	✗	✓	✗	✓	✓
Flexible tensor strides	✓	✓	✗	✓	✗	✗	✗	✓
Tensor overlap detection	✓	✗	✗	✓	✓	✗	✗	✓

The functions shown in **Figure 2(a)** and **(b)** implement two variations of an in-place QR factorization $A = QR$. The original matrix A is stored in the first parameter, Q, and will be overwritten by the orthonormal part of the factorization. The second parameter, R, is assumed to be pre-allocated and will be used to store the upper-diagonal part of the factorization. Normally the only input to the function would be A, but here we allow the user to specify Q and R with any data type and device. In line 2 of the code we determine the number of columns in Q. The QR factorization then proceeds by repeating the following process on each of the columns i in Q. First we get the current column $q = Q[:,i]$ in line 4. Because the indexing is regular this tensor will be a view of the original Q. In line 5 we determine the two-norm of q by taking the square root of the inner-product with itself. Note that,

```
1   def InplaceQR(Q,R) :
2       n = Q.size[1]
3       for i in range(n) :
4           q = Q[:,i]
5           r = ocean.sqrt(q.T * q)
6           q /= r
7           R[i,i] = r
8           for j in range(i+1,n) :
9               r = q.T * Q[:,j]
10              Q[:,j] -= q * r
11              R[i,j] = r
```

(a)

```
1   def InplaceQR(Q,R) :
2       n = Q.size[1]
3       for i in range(n) :
4           q = Q[:,i]
5           r = ocean.sqrt(q.T * q)
6           q /= r
7           R[i,i] = r
8           if (i+1 < n) :
9               r = Q[:,i+1:].T * q
10              Q[:,i+1:] -= q * r.T
11              R[i,i+1:] = r
```

(b)

```
1   import ocean
2
3   # Create an example matrix A with one added to the diagonal
4   # entries to make if full rank.
5   A = ocean.arange(25, ocean.double).reshape(5,5)
6   d = A.diag(); d += 1;
7
8   # As an example, create a byte-swapped copy Q on the cpu and a
9   # single-precision result tensor R on gpu[0].
10  Q = A.clone(); Q.byteswap()
11  R = ocean.zeros(A.size, ocean.float, ocean.gpu[0])
12
13  # Call the in-place QR factorization code (see code above)
14  InplaceQR(Q,R)
15
16  # Display matrices, verify orthogonality, and check factorization
17  print(Q); print(R)
18  print(ocean.norm(Q.T * Q - ocean.eye(Q.size[0])))
19  print(ocean.norm(Q*R - A))
```

(c)

```
# Matrix Q
(:,:)
    0.17961    0.41037    0.58318    0.51237    0.44353
    0.17961    0.77910   -0.59087   -0.07841    0.07392
    0.35921    0.26763    0.51389   -0.66920   -0.29569
    0.53882   -0.05947   -0.06544    0.50915   -0.66530
    0.71842   -0.38658   -0.20594   -0.15575    0.51745
<tensor.double of size 5x5 on cpu (byteswapped)>

# Matrix R
(:,:)
    5.56776   15.44606   25.50395   35.56185   45.61975
    0.00000    5.42396    9.96772   14.69584   19.42397
    0.00000    0.00000    2.27881    2.87354    3.90708
    0.00000    0.00000    0.00000    1.76914    1.69502
    0.00000    0.00000    0.00000    0.00000    1.55236
<tensor.float of size 5x5 on gpu0>

# Orthogonality and factorization using single-precision R
8.81018e-15
2.1845e-6
```

(d)

Figure 2: In-place QR factorization functions with **(a)** individual column updates in Q, and **(b)** multi-column rank-one matrix updates of Q; along with **(c)** the calling script, and **(d)** the output annotated with comments. Note that Ocean supports vector inner and outer products of the form `u.T*v` and `u*v.T`.

unlike many other packages, we define the transpose `q.T` as a $1 \times n$ matrix rather than the original one-dimensional vector, to enable natural notation for inner and outer products. In line 6 we normalize q, and therefore the corresponding column in Q, and update the diagonal entry in R. The for-loop in lines 8–11 in **Figure 2(a)** projects out the q component in each of the remaining columns of Q, proceeding one column at a time. The same lines in **Figure 2(b)** achieve the same, operating on all remaining columns at once using indexing and matrix operations instead of vector operations. In the second case we take the outer product of vectors q and r, written as qr^T.

Figure 2(c) gives an example script calling either one of the two factorization functions (algorithmically they are equivalent). After importing the Ocean Tensor Package in line 1 we create a square matrix A on line 5, by first creating a vector of type double on the default device (CPU) containing the numbers 0 through 24 of type double, and then reshaping the vector to a 5×5 matrix. The resulting matrix has rank 2 instead of 5, and in line 6 we therefore add the identity matrix by creating a view d on the diagonal and adding 1 to each element. On lines 10–11 we prepare Q by creating a deep copy of matrix A and byteswapping its data, and allocate a float matrix R on device `gpu[0]` with the same dimensions as A. We choose Q and R to have different data type, device, and byte order to illustrate the seamless type casting provided by Ocean. After calling the in-place QR factorization function in line 14, we print out the two matrices along with the Frobenius norms $\|Q^TQ - I\|_F$ and $\|QR - A\|_F$ in lines 17–19. The output of the script given in **Figure 2(d)** shows the elements and type of matrices Q and R, and the values of the two norms. The seemingly inaccurate factorization of A is due to the use of single-precision floats in R.

Implementation and architecture

The Ocean Tensor Package is designed to serve as a foundational layer for applications that require dense tensor operations on one or more device types and instances. Given the wide range of potential applications and domains, it is important that the tensor operations are grouped in coherent modules, rather than be provided through a huge monolithic package. This way, functionality can be installed by the user when needed, which helps reduce the effective number of dependencies. Another advantage is that interfaces and compatibility with external libraries is localized to independent modules, thus making the package easier to manage. Another design principle used in Ocean, and discussed later in this section, is the use of well-defined layers.

Modularity

Modules in Ocean consist of an interface along with independent implementations for each of the supported device types. The module interface takes care of device-independent parameter checks including validity of the tensors and compatibility of tensor dimensions. It then determines the data type and device to use, and queries a function look-up table for the module associated with the device type. When available, the function is called after performing all necessary type conversions, broadcasting of dimensions, and allocation of result and intermediate tensors (for instance when tensors overlap in memory). In case the function is not available, or the module implementation for the device type has not been loaded, an error is raised. Functions at the device level typically only need to check for support of tensors of the given data type and either implement the tensor operation or, more typically, call a lower-level library function that provides the desired operation. If needed, functions can access the module-specific context information associated with each device instance.

Module interfaces and device implementations can be loaded separately, except for the core module interface, which includes the CPU implementation. The separation between the interface and the implementation makes it possible to replace the module implementation with alternatives, such as a highly-tuned or specialized proprietary version. The use of function tables also makes it possible to replace individual functions for performance comparisons or debugging, or to insert functions that record runtime or accumulate call statistics. The separation between module interfaces and device implementation also make it easy to extend Ocean with new device types. In particular, modules and functions within each module can be added and tested one at a time, thus avoiding a huge development effort to get started.

The Core module forms the basis of the Ocean Tensor Package. It provides all elementary tensor operations and instantiates and exposes available device instances. Many of the standard functions, such as printing and tensor copies between different device types require tensor support on the CPU, and the Core module interface is therefore combined with the CPU implementation (`pyOcean_cpu`). The GPU implementation can be loaded separately by importing `pyOcean_gpu`. For convenience both packages are loaded by importing `ocean`. Dependencies between modules are allowed, and instantiation of one module can cause other modules to be loaded. The instantiation order of modules is carefully registered to ensure that modules are finalized only when no other modules depend on them.

Layered implementation

In the implementation of the Ocean Tensor Package care is taken to maintain a clean separation between the different abstraction levels, as illustrated in **Figure 3**. The libraries at the bottom level provide low-level tensor operations that are independent on the tensor representation. This includes existing libraries such as BLAS and cuBLAS, as well as the custom developed Solid foundation library, which provides elementary tensor functions for CPU and GPU. Libraries at this level are not specific to Ocean and can be used independently by other applications that require low-level tensor operations. The Ocean tensor API defines a unified tensor type along with a variety of tensor operations, organized as modules. As described above, modules themselves can be classified in two layers, namely interfaces and device implementations. The Ocean API is implemented in C
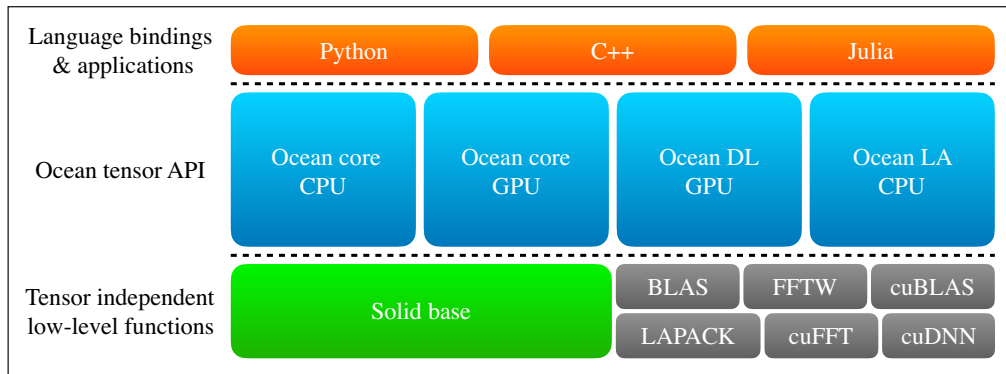
Figure 3: Layered design of the Ocean Tensor Package. The current version implements the core modules for CPU and GPU, based on BLAS, cuBLAS, and the Solid foundation library, along with the Python language binding.

to maximize the accessibility from other languages. The top level in **Figure 3** shows possible language bindings to Ocean (the current version only implements support for Python). However, usage of the API is not restricted to language bindings. For instance, applications that use tensor operations, or libraries that support symbolic tensor compute graphs, too could be build on top of the Ocean tensor API.

Quality control

The package comes with an extensive collection of unit tests in the `python/examples` directory. The `verify` scripts in the `cpu` and `gpu` directories can be used to run the unit tests and check whether the output matches the reference.

(2) Availability

Operating system

The system has been tested on Red Had Enterprise Linux version 7.4 running on Power8 and Intel Xeon, as well as on MacOS High Sierra, version 10.13.4, running on a MacBook Pro with Intel Core i7.

Programming language

The Ocean Tensor Package is written in C based on the C99 standard, with GPU functionality implemented using CUDA. The Python-C API is also written in C, along with several Python scripts. The package has been tested with Python 2.7.5 and 3.5.2.

Additional system requirements

When available, the Ocean Tensor Package supports CUDA-enabled GPUs. The total total disk space required after compilation is approximately 300Mb.

Dependencies

The CPU part of the code has optional dependencies on BLAS or CBLAS. The user is encouraged to provide these, otherwise compilation is done using a non-optimized implementation provided with the package. The implementation of multi-threaded tensor operations is done using OpenMP, when available, otherwise all operations are single-threaded. Compilation on Linux was done using GCC 4.8.5; on MacOS compilation was tested using Clang versions 7.0.0 through 9.1.0.

The GPU part of the code uses CUDA as well as the cuBLAS library. The package has been tested with CUDA versions 7.5, 8.0 (GA1, GA2), 9.0, 9.1, and 9.2. Compilation on MacOS using NVCC requires the appropriate combination of Clang and CUDA versions. A table of compatible versions is provided in the `INSTALL.txt` file provided with the package.

The Python interface of the package provides an optional interface to Numpy, when available on the system.

List of contributors

Ewout van den Berg designed and implemented the package, and currently maintains the GitHub repository.

Code repository

Name: GitHub
Persistent identifier: https://github.com/IBM/ocean-tensor-package
Licence: Apache-2.0
Date published: 16/08/2018

Language

English

(3) Reuse potential

The Ocean Tensor Package provides support for tensor operations on CPU and GPU. Given the common usage of these operations in various fields, there is a large reuse potential of the package. The package can be used directly as the user level, or can serve as the foundation for other packages. Tensor functionality is organized in modules to enable addition of separate modules with operations specific to other fields. Additional modules can be provided as third-party extensions, or incorporated in the package itself, depending on the level of specialization of the functions. The package was designed to allow support for devices other than CPU and GPU, although no such extensions are currently planned. Please contact the author if you are interested in extending the package, or if you have questions or feedback regarding the installation and usage of the package.

Notes

[1] Available at https://github.com/ibm/ocean-tensor-package.

² Ocean supports booleans; 8, 16, 32, and 64-bit signed and unsigned integers; as well as real and complex half, single and double-precision floating-point data types.

Competing Interests

The author has no competing interests to declare.

References

1. **Abadi, M, Agarwal, A, Barham, P, Brevdo, E, Chen, Z, Citro, C, Corrado, G S, Davis, A, Dean, J, Devin, M, Ghemawat, S, Goodfellow, I, Harp, A, Irving, G, Isard, M, Jia, Y, Jozefowicz, R, Kaiser, L, Kudlur, M, Levenberg, J, Mané, D, Monga, R, Moore, S, Murray, D, Olah, C, Schuster, M, Shlens, J, Steiner, B, Sutskever, I, Talwar, K, Tucker, P, Vanhoucke, V, Vasudevan, V, Viégas, F, Vinyals, O, Warden, P, Wattenberg, M, Wicke, M, Yu, Y** and **Zheng, X** 2015 TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
2. **Broekema, P C, Mol, J J D, Nijboer, R, van Amesfoort, A S, Brentjens, M A, Loose, G M, Klijn, W F A** and **Romein, J W** 2018 A GPU-based correlator and beamformer for LOFAR. arXiv: 1801.04834. DOI: https://doi.org/10.1016/j.ascom.2018.04.006
3. **Cano, A** 2018 A survey on graphic processing unit computating for large-scale data mining. *WIREs Data Mining Knowledge discovery*, 8: e1232. DOI https://doi.org/10.1002/widm.1232
4. **Chen, T, Li, M, Li, Y, Lin, M, Wang, N, Wang, M, Xiao, T, Xu, B, Zhang, C** and **Zhang, Z** 2015 MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*.
5. **Collobert, R, Bengio, S** and **Marithoz, J** 2002 Torch: A modular machine learning software library.
6. **Collobert, R, Kavukcuoglu, K** and **Farabet, C** 2011 Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop, number EPFLCONF-192376*.
7. **Jia, Y, Shelhamer, E, Donahue, J, Karayev, S, Long, J, Girshick, R, Guadarrama, S** and **Darrell, T** 2014 Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, 675–678. DOI: https://doi.org/10.1145/2647868.2654889
8. **Kutzner, C, Páll, S, Fechner, M, Esztermann, A, de Groot, B L** and **Grubmüller, H** 2015 Best bang for your buck: GPU nodes for GROMACS biomolecular simulations. *Journal of Computational Chemistry*, 36(26): 1990–2008. DOI: https://doi.org/10.1002/jcc.24030
9. **NVIDIA.** GPU-accelerated applications. https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/.
10. **Okuta, R, Unno, Y, Nishino, D, Hido, S** and **Loomis, C** 2017 CuPy: A NumPycompatible library for NVIDIA GPU calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*.
11. **Oliphant, T E** 2006 *A guide to Numpy*. Trelgol Publishing.
12. **Paszke, A, Gross, S, Chintala, S, Chanan, G, Yang, E, DeVito, Z, Lin, Z, Desmaison, A, Antiga, L** and **Lerer, A** 2017 Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop*.
13. **Shi, L, Liu, W, Zhang, H, Xie, Y** and **Wang, D** 2012 A survey of GPU-based medical image computing techniques. *Quantitative Imaging in Medicine and Surgery*, 2(3): 188–206.
14. **Shi, X, Zheng, Z, Zhou, Y, Jin, H, He, L, Liu, B** and **Hua, Q-S** 2018 Graph processing on GPUs: A survey. *ACM Computing Surveys*, 50(6): 81: 1–35. DOI: https://doi.org/10.1145/3128571
15. **Trefethen, L N** and **Bau, D, III** 1997 *Numerical Linear Algebra*. SIAM. DOI: https://doi.org/10.1137/1.9780898719574
16. **van den Berg, E** 2018 The Ocean Tensor Package. Available at https://github.com/ibm/ocean-tensor-package.
17. **Wang, H, Peng, H, Chang, Y** and **Liang, D** 2018 A survey of GPU-based acceleration techniques in mri reconstructions. *Quantitative Imaging in Medicine and Surgery*, 8(2): 196–208. DOI: https://doi.org/10.21037/qims.2018.03.07
18. **Yalamanchili, P, Arshad, U, Mohammed, Z, Garigipati, P, Entschev, P, Kloppenborg, B, James, J** and **Melonakos, J** 2015 ArrayFire – a high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire.