cnine

tensor Library, v0.0

Last updated: August 2021

Risi Kondor
Department of Computer Science, Statistics,
and the Computational and Applied Math Initiative
The University of Chicago

Contents

Overview

cnine is a simple tensor library written in C++ with a CUDA backend. cnine is designed to make it easier to write high performance GPU code for machine learning without reliance on complex proprietary numerical libraries.

cnine provides the following features:

- Support for real and complex valued scalars, tensors and tensor-arrays.
- Data objects can be freely moved between the host (CPU) and the GPU. Built-in operators work in both contexts.
- The memory layout of tensors and tensor-arrays is optimized for GPU performance.
- Multiple mechanisms are provided for parallelizing operations over tensors and tensor-arrays, including mapping operations over the cells of tensor-arrays in both regular and irregular patterns.

Some of the present limitations of cnine are the following:

- Single precision arithmetic only;
- No tensor core support;
- Single GPU support only;
- Only NVIDIA GPUs supported.

cnine is under continuous development. Some features described in this document may not be fully implemented yet. cnine is authored by Risi Kondor and released under the Mozilla Public License, version 2.0, `https://www.mozilla.org/en-US/MPL/2.0/`.

Installation and usage

Compiling code with cnine requires the following:

- An appropriate `C++11` compiler together with the STL standard template library.
- CUDA and CUBLAS if the library is to be used with GPU functionality.

Most of cnine is structured as a header library, which can be called directly from user code without pre-compilation. The exception to this are the CUDA object files corresponding to built-in cell operators, which are compiled by running `make all` in the `cuda` directory.

Compilation options

Some global compilation options can be set in the file `options.txt`:

Variable	Default value	Description
`CC`	`clang`	Name of C++ compiler.
`WITH_CUDA`	`t`	If this variable is defined, the library will link to CUDA. If not defined, GPU functionality is disabled.
`CUDA_DIR`	`/usr/local/cuda`	Root directory of the CUDA installation on your system.
`NVCC`	`nvcc`	Name of CUDA compiler.
`NVCCFLAGS`	omitted	Various flags to be passed to the NVCC compiler.
`WITH_CUBLAS`	`t`	If this variable is defined, the library will link to CUBLAS for low level linear algebra functionality on the GPU.

Usage

To use cnine from your own code you must do the following:

- `#include` the relevant header files in your soure files.
- `#include` the file `include/cnine_base.cpp` in your top level source file (the one that contains your `main` function).
- Define a `cnine_session` object in your code before calling any `cnine` objects or functions.
- Link in the appropriate CUDA object files from the `cuda` directory, as required.

Classes

The following pages describe the APIs of most user callable classes and functions in cnine.

As a general rule, every non-trivial class in cnine defines

1. A copy constructor `CLASS(const CLASS& x)`.
2. An assignment operator `CLASS& operator=(const CLASS& x)`.
3. A move-constructor `CLASS(const CLASS&& x)`.
4. A move-assignment operator `CLASS& operator=(const CLASS&& x)`.
5. The destructor `~CLASS()`.

These methods are not listed separately.

Devices

The `device` helper class is used to select whether a given object is created on (or moved to) the CPU or the GPU. Initializing a `device` variable with the integer 0 corresponds to the CPU and 1 corresponds to the GPU. Alternatively, the following static variable can be used:

`device deviceid::CPU`	The `device` object corresponding to the host (CPU).
`device deviceid::GPU0`	The `device` object corresponding to first graphics device (GPU).

Fill types

cnine uses dummy types to specify how new data objects are initialized.

`fill_noalloc`	No new memory is allocated to store the object's data.
`fill_raw`	Memory is allocated to store the object, but not initialized.
`fill_zero`	The object if filled with zeros.
`fill_ones`	Each element of the object is set to "1".
`fill_sequential`	The elements of the object are set to $1, 2, \ldots$.
`fill_identity`	The object is initialized to an identity matrix.
`fill_unform`	Each element of the object is drawn from the uniform distribution on $[0, 1]$.
`fill_gaussian`	Each element of the object is drawn from the normal distribution $\mathcal{N}(0, 1)$.

To simplify syntax, each of these types has a corresponding static object.

fill::noalloc	A dummy object of type fill_noalloc
fill::raw	A dummy object of type fill_raw.
fill::zero	A dummy object of type fill_zero.
fill::ones	A dummy object of type fill_ones.
fill::sequential	A dummy object of type fill_sequential.
fill::identity	A dummy object of type fill_identity.
fill::unform	A dummy object of type fill_uniform.
fill::gaussian	A dummy object of type fill_gaussian.

Some classes also provide specialized named constuctors corresponding to these fill types, for example, Ctensor::zero(...), Ctensor::ones(...), etc..

Expression templates

cnine uses the following expression templates:

	shorthand	description
Conjugate<OBJ>	conj(x)	The conjugate of x.
Transpose<OBJ>	transp(x)	The transpose of x.
Hermitian<OBJ>	herm(x)	The Hermitian conjugate (conjugate transpose) of x.

Bundling

There are multiple mechanisms in cnine for parallelization. One of the simplest ones is adding a "bundle dimension" to variables, which effectively amounts to multiplexing the variable n_{bu} times. This is similar to what is called the "batch dimension" in e.g., PyTorch. Applying an operation OP to two or more bundled objects is equivalent to applying OP independtly to each of the n_{bu} parts of the bundle and then bundling the results into a new object with the same bundle dimension. Most operators are written in such a way that when performed on the GPU this process is automatically parallelized over the bundle dimension.

Omitting the bundle dimension in the constructor or setting it to -1 generally turns bundling off. The bundling functionality is currently experimental and not fully implemented in all cnine classes.

Notation

The following notations and shorthands are used in the class descriptions.

☜	This function produces a temporary that is assignable and therefore may be used as an lvalue.

NBU	Bundle dimension. If bundling is off, should be set to -1.
DEVICE	device object specifying the device on which a given variable is created or moved to. Currently the two choices are deviceid::CPU and deviceid::GPU0 (can also be initilized with 0 and 1).
FILLTYPE	One of the fill type dummy objects listed above.

Scalar and tensor classes

The individual elements of cnine's tensor and tensor-array classes can be accessed directly as `float`s or `complex<float>`s. The library also provides its own scalar classes, `rscalar` and `scalar`, which provide some additional functionality.

An `Rtensor` is a k'th order real tensor $\mathbb{R}^{d_1 \times d_2 \times \ldots \times d_k}$ and a `Ctensor` is a k'th order complex tensor $\mathbb{C}^{d_1 \times d_2 \times \ldots \times d_k}$. Both real and complex tensors are stored in single precision arithmetic, similar to the `float` type. Matrices are a special case corresponding to $k = 2$. cnine's default backend tensor classes are `RtensorA` and `CtensorA`, but these do not need to be accessed directly except when adding new CUDA cell operators.

Tensor objects can be placed either on the CPU or the GPU and moved back and forth between them with the `move_to` method. Almost all tensor operations can be performed on either the CPU or the GPU. The general rule is that all tensor arguments of a given operation must reside on the same device, and the result of the operation will be placed on the same device as where the arguments were.

Scalar objects cannot be placed on the GPU because they are too small for that to be economical. The results of operations involving a mixture of scalar and tensor arguments are placed on the same device as the tensor arguments.

rscalar

An `rscalar` object represents a single real number x, or, when bundling is enabled, n_{bu} separate real numbers. `rscalar` objects are always allocated on the host (CPU).

CONSTRUCTORS

`rscalar(float x)`
    Create a new `rscalar` object initialized to x.

`rscalar([NBU], FILLTYPE)`
    Create a new `rscalar` of budle dimension `NBU`. `FILLTYPE` can be `fill::raw`, `fill::zero` or `fill::gaussian`.

STATIC CONSTRUCTORS

`rscalar rscalar::zero([NBU])`
`rscalar rscalar::gaussian([NBU])`
    Create a zero or Gaussian `rscalar` of bundle dimension `NBU`.

ELEMENT ACCESS

`float get_value() const`
`explicit operator float() const`
    Return the value of `x` as a `float`. `value` may be used as a shorthand for `get_value`.

`Cscalar& set_value(float y)`
`Cscalar& operator=(float y)`
    Set the value of `x` to y. `set` may be used as a shorthand for `set_value`.

`rscalar apply(std::function<float(float x)> fn)`
    Apply the function `fn` to x.

ARITHMETIC

`x1+x2` (rscalar,rscalar) -> rscalar
`x1-x2` (rscalar,rscalar) -> rscalar
`x1*x2` (rscalar,rscalar) -> rscalar
`x1/x2` (rscalar,rscalar) -> rscalar

`abs(x)` rscalar -> rscalar
    Return absolute vale of x

`pow(x,p)` (rscalar,float) -> rscalar
    Return x^p.

`exp(x)` (rscalar,float) -> rscalar
    Return e^x.

IN-PLACE OPERATIONS

`+=y` rscalar -> rscalar
`-=y` rscalar -> rscalar

Increment/decrement x by y.

I/O

`string str(const indent="") const`

Print x to string with the optional indentation `indent`.

cscalar

A `cscalar` object represents a single complex number z, or, when bundling is enabled, $n_{\rm bu}$ separate complex numbers. `cscalar` objects are always allocated on the host (CPU).

CONSTRUCTORS

`cscalar(complex z)`
  Create a new `cscalar` object initialized to z.

`cscalar(float x, float y)`
  Create a new `cscalar` object initialized to $z = x + iy$.

`cscalar([NBU],FILLTYPE)`
  Create a new `cscalar` object. `FILLTYPE` can be `fill::raw`, `fill::zero` or `fill::gaussian`.

STATIC CONSTRUCTORS

`cscalar cscalar::zero([NBU])`
`cscalar cscalar::gaussian([NBU])`
  Create a zero or Gaussian `cscalar` of bundle size `NBU`.

ELEMENT ACCESS

`complex<float> get_value()`
`explicit operator complex<float>()`
  Return the value of `z` as a `complex<float>`. `value` may be used as a shorthand for `get_value`.

`cscalar& set_value(complex<float> x)`
`cscalar& operator=(complex<float> x)`
  Set the value of `z` to x. `set` may be used as a shorthand for `set_value`.

`cscalar apply(std::function<complex<float>(complex<float> z)> fn)`
  Apply the function `fn` to z.

ARITHMETIC

`z1+z2`	`(cscalar,cscalar) -> cscalar`
`z1-z2`	`(cscalar,cscalar) -> cscalar`
`z1*z2`	`(cscalar,cscalar) -> cscalar`
`z1/z2`	`(cscalar,cscalar) -> cscalar`

`pow(z,p)`   `(cscalar,TYPE) -> cscalar`
  Return z^p.

IN-PLACE OPERATIONS

`+=y` — cscalar
`-=y` — cscalar

Increment/decrement z by y.

OTHER FUNCTIONS

`real(z)` — cscalar -> rscalar
`imag(z)` — cscalar -> rscalar

The real resp. imaginary part of z.

`abs(z)` — cscalar -> cscalar
`conj(z)` — cscalar -> cscalar

The modulus resp. complex conjugate of z.

`ReLU(z)` — cscalar -> cscalar
`ReLU(z,alpha)` — (cscalar,float) -> cscalar

Apply the ReLU or leaky ReLU operator to z.

I/O

`string str(const indent="") const`

Print the value of the scalar to string with the optional indentation `indent`.

rtensor

An `rtensor` object stores a k'th order real tensor T. The $k = 2$ case is just a matrix.

Implemented by: `RtensorA`

CONSTRUCTORS

```
rtensor(const Gdims& dims, [NBU], [FILLTYPE], [DEVICE])
```

Create a new `rtensor` object of dimensions `dims`. `FILLTYPE` can be `fill::raw`, `fill::zero`, `fill::ones`, `fill::identity`, `fill::sequential` or `fill::gaussian`.

```
rtensor(const Gdims& dims,[NBU],std::function<float>(const Gindex& ix) fn,[DEVICE])
rtensor(const Gdims& dims,[NBU],std::function<complex<float>>(const int i1,...,
        const int ik) fn,[DEVICE])
```

Create a new `rtensor` of dimensions `dims` initialized by calling the function `fn` for each element.

STATIC CONSTRUCTORS

```
rtensor rtensor::zero(const Gdims& dims, [NBU], [DEVICE])
rtensor rtensor::ones(const Gdims& dims, [NBU], [DEVICE])
rtensor rtensor::identity(const Gdims& dims, [NBU], [DEVICE])
rtensor rtensor::sequential(const Gdims& dims, [NBU], [DEVICE])
rtensor rtensor::gaussian(const Gdims& dims, [NBU], [DEVICE])
```

Create a new `rtensor` of dimensions `dims`, bundle size `NBU` on device `DEVICE`.

CONVERSIONS

```
rtensor(const rtensor& T, const device& dev)
```

Return a copy of `T` on device `dev`.

```
Gtensor<complex<float>> gtensor() const
rtensor(const Gtensor<float>& S)
```

Return T as a `Gtensor` or initialize T from a `Gtensor`.

ELEMENT ACCESS

```
operator()(const Gindex& ix) ☜
operator()(int i1,...,int ik) ☜
```

Return a temporary object referencing $T_{i_1,\ldots,i_k}$.

```
rscalar get(const Gindex& ix) const
rscalar get(int i1,...,int ik) const
```

Return the tensor element $T_{i_1,\ldots,i_k}$ as an `rscalar`.

```
rtensor& set(const Gindex& ix, const rscalar& x)
rtensor& set(int i1,...,int ik, const rscalar& x)
```

Set $T_{i_1,\dots,i_k} = x$.

```
float get_value(const Gindex& ix) const
float get_value(int i1,...,int ik) const
```

Return the value of $T_{i_1,\dots,i_k}$.

```
rtensor& set_value(const Gindex& ix, const float x)
rtensor& set_value(int i1,...,int ik, const float x)
```

Set $T_{i_1,\dots,i_k} = x$.

```
rtensor apply(std::function<float>(float x)> fn)
rtensor apply(std::function<float>(const int i, const int j, float x)> fn)
```

Map the function `fn` over each element of T. In the second form, `i` is the first index and j is the second index of T.

```
rtensor slice(int d, int i) const
```

Return the i'th slice of T along dimension d.

```
rtensor chunk(int d, int i, int n) const
```

Return the chunk of T corresponding to the d'th index being in the range $[i,\, i+n-1]$.

```
reshape(const Gdims& dims)
```

Reinterpret T as `rtensor` of dimensions `dims`.

ARITHMETIC

`c*T`	`(rscalar,rtensor) -> rtensor`
`S+T`	`(rtensor,rtensor) -> rtensor`
`S-T`	`(rtensor,rtensor) -> rtensor`
`S*T`	`(rtensor,rtensor) -> rtensor`

The contracted product $R_{i_1,\dots i_{k_1+k_2-2}} = \sum_j S_{i_1,\dots i_{k_1-1},j}\, T_{j,i_{k_1},\dots i_{k_1+k_2-2}}$. In the matrix/matrix case this reduces to the ordinary matrix product.

IN-PLACE OPERATIONS

`+=S`	`rtensor`
`-=S`	`rtensor`

Increment/decrement T by S.

OTHER FUNCTIONS

`transp(T)` — `rtensor -> rtensor`

The transpose of `T`.

`stack(d,T1,...,Tm)` — `(rtensor,...,rtensor) -> rtensor`

Stack the k'th order tensors $T_1,\dots,T_m$ along dimension d into a $(k+1)$'th order tensor.

`cat(d,T1,...,Tm)` — `(rtensor,...,rtensor,int) -> rtensor`

Concatenate $T_1,\dots,T_m$ along dimension d.

`norm2(T)` (rtensor,rtensor) -> rscalar

The squared Frobenius norm $\|T\|^2_{\mathrm{Frob}} = \sum_{i_1,\ldots i_k} |T_{i_1,\ldots,i_k}|^2$.

`inp(S,T)` (rtensor,rtensor) -> rscalar

`odot(S,T)` (rtensor,rtensor) -> rscalar

The inner product $\langle S,T\rangle = \sum_{i_1,\ldots i_k} S_{i_1,\ldots,i_k} T^*_{i_1,\ldots,i_k}$, and pointwise product $R_{i_1,\ldots i_k} = S_{i_1,\ldots,i_k} T_{i_1,\ldots,i_k}$.

`ReLU(z)` rtensor -> rtensor

`ReLU(z,alpha)` (rtensor,float) > rtensor

Apply the ReLU or leaky ReLU operator to each element of T.

I/O

`string str(const indent="") const`

Print the tensor to string with the optional indentation `indent`.

ctensor

An `ctensor` object stores a k'th order complex tensor T. The $k=2$ case is just a matrix.

Implemented by: `CtensorA`

CONSTRUCTORS

```
ctensor(const Gdims& dims, [NBU], [FILLTYPE], [DEVICE])
```

Create a new `ctensor` object of dimensions `dims`. `FILLTYPE` can be `fill::raw`, `fill::zero`, `fill::ones`, `fill::identity`, `fill::sequential` or `fill::gaussian`.

```
ctensor(const Gdims& dims,[NBU],std::function<complex<float>>(const Gindex& ix) fn,[DEVICE])
ctensor(const Gdims& dims,[NBU],std::function<complex<float>>(const int i1,...,
        const int ik) fn,[DEVICE])
```

Create a new `ctensor` of dimensions `dims` initialized by calling the function `fn` for each element.

STATIC CONSTRUCTORS

```
ctensor ctensor::zero(const Gdims& dims, [NBU], [DEVICE])
ctensor ctensor::ones(const Gdims& dims, [NBU], [DEVICE])
ctensor ctensor::identity(const Gdims& dims, [NBU], [DEVICE])
ctensor ctensor::sequential(const Gdims& dims, [NBU], [DEVICE])
ctensor ctensor::gaussian(const Gdims& dims, [NBU], [DEVICE])
```

Create a new `ctensor` of dimensions `dims`, bundle size `NBU` on device `DEVICE`.

CONVERSIONS

```
ctensor(const ctensor& T, const device& dev)
```

Return a copy of `T` on device `dev`.

```
Gtensor<complex<float>> gtensor() const
ctensor(const Gtensor<complex<float>>& S)
```

Return T as a `Gtensor` or initialize T from a `Gtensor`.

ELEMENT ACCESS

```
operator()(const Gindex& ix)
operator()(int i1,...,int ik)
```

Return a temporary object referencing $T_{i_1,\ldots,i_k}$.

```
cscalar get(const Gindex& ix) const
cscalar get(int i1,...,int ik) const
```

Return the tensor element $T_{i_1,\ldots,i_k}$ as a `cscalar`.

`ctensor& set(const Gindex& ix, const cscalar& x)`
`ctensor& set(int i1,...,int ik, const cscalar& x)`

Set $T_{i_1,\ldots,i_k} = x$.

`complex<float> get_value(const Gindex& ix) const`
`complex<float> get_value(int i1,...,int ik) const`

Return the value of $T_{i_1,\ldots,i_k}$.

`ctensor& set_value(const Gindex& ix, const complex<float> x)`
`ctensor& set_value(int i1,...,int ik, const complex<float> x)`

Set $T_{i_1,\ldots,i_k} = x$.

`ctensor apply(std::function<complex<float>(complex<float> x)> fn)`
`ctensor apply(std::function<complex<float>(const int i, const int j, complex<float> x)> fn)`

Map the function `fn` over each element of T. In the second form, `i` is the first index and j is the second index of T.

`ctensor slice(int d, int i) const` ☜

Return the i'th slice of T along dimension d.

`ctensor chunk(int d, int i, int n) const` ☜

Return the chunk of T corresponding to the d'th index being in the range $[i,\, i+n-1]$.

`reshape(const Gdims& dims)`

Reinterpret T as `ctensor` of dimensions `dims`.

ARITHMETIC

`c*T` `(cscalar,ctensor) -> ctensor`
`S+T` `(ctensor,ctensor) -> ctensor`
`S-T` `(ctensor,ctensor) -> ctensor`

`S*T` `(ctensor,ctensor) -> ctensor`

The contracted product $R_{i_1,\ldots i_{k_1+k_2-2}} = \sum_j S_{i_1,\ldots i_{k_1-1},j}\, T_{j,i_{k_1},\ldots i_{k_1+k_2-2}}$. In the matrix/matrix case this reduces to the ordinary matrix product.

IN-PLACE OPERATIONS

`+=S` `ctensor`
`-=S` `ctensor`

Increment/decrement T by S.

OTHER FUNCTIONS

`conj(T)` `ctensor -> ctensor`
`transp(T)` `ctensor -> ctensor`
`herm(T)` `ctensor -> ctensor`

The conjugate, transpose, and Hermitian conjugate of `T`.

`stack(d,T1,...,Tm)` `(ctensor,...,ctensor) -> ctensor`

Stack the k'th order tensors $T_1,\ldots,T_m$ along dimension d into a $(k+1)$'th order tensor.

`cat(d,T1,...,Tm)` `(ctensor,...,ctensor,int) -> ctensor`

Concatenate $T_1, \ldots, T_m$ along dimension d.

`norm2(T)` `(ctensor,ctensor) -> cscalar`

The squared Frobenius norm $\|T\|^2_{\mathrm{Frob}} = \sum_{i_1,\ldots i_k} |T_{i_1,\ldots,i_k}|^2$.

`inp(S,T)` `(ctensor,ctensor) -> cscalar`
`odot(S,T)` `(ctensor,ctensor) -> cscalar`
`odotC(S,T)` `(ctensor,ctensor) -> cscalar`

The inner product $\langle S,T\rangle = \sum_{i_1,\ldots i_k} S_{i_1,\ldots,i_k} T^*_{i_1,\ldots,i_k}$, and pointwise products $R_{i_1,\ldots i_k} = S_{i_1,\ldots,i_k} T_{i_1,\ldots,i_k}$ resp. $R_{i_1,\ldots i_k} = S_{i_1,\ldots,i_k} T^*_{i_1,\ldots,i_k}$

`ReLU(z)` `ctensor -> ctensor`
`ReLU(z,alpha)` `(ctensor,float) > ctensor`

Apply the ReLU or leaky ReLU operator to each element of T.

I/O

`string str(const indent="") const`

Print the tensor to string with the optional indentation `indent`.

Tensor-array classes

A tensor-array is an array of tensors of the same type and dimensions. Tensor-arrays are the main vehicle in cnine for parallelizing operations on the GPU, allowing the same operation to be executed concurrently across many cells in different patterns. For example, given two `CtensorArray` objects `A` and `B` with the same array dimensions,

```
CtensorArray C=cellwise<Ctensor_Mprod>(A,B);
```

computes a new array in which each cell is the matrix product of the corresponding cells of `A` and `B`. On the other hand, assuming that `A` is an array of n tensors and `B` is an array of m tensors,

```
CtensorArray D=outer<Ctensor_Mprod>(A,B);
```

creates an $n \times m$ array, in which the cell $[\![D]\!]_{i,j}$ is the matrix product of $[\![A]\!]_i$ and $[\![B]\!]_j$.

The default backend class of `CtensorArray` is `CtensorArrayA`, which stores the real part of each tensor in a single C++ array and the imaginary part of each tensor in a second C++ array. Each cell in the two arrays is padded to a multiple of 128 bytes. This facilitates efficient parallelization on GPUs, but also means that a tensor-array with k array dimensions and ℓ cell dimensions cannot easily be reinterpreted as a single $k+\ell$'th order tensor in standard storage format.

`RtensorArray`

A `RtensorArray` stores a multidimensional array $[\![T]\!]$ of `Rtensor` objects.

Implemented by: `RtensorArrayA` (derived from `RtensorA`)

CONSTRUCTORS

`RtensorArray(const Gdims& adims, const Gdims& dims, [NBU], [FILLTYPE], [DEVICE])`
: Construct an `adims` sized array of `Rtensor`s, each of dimensions `dims`. `FILLTYPE` can be `fill::raw`, `fill::zero`, `fill::identity`, `fill::sequential` or `fill::gaussian`.

`RtensorArray(const Rtensor& T)`
`RtensorArray(const Gdims& adims, const Rtensor& T)`
: Construct a single cell array from T or an array of size `adims` in which each cell is T.

`RtensorArray(const RtensorArray& T, const view_flag& dummy)`
: Construct a view of `T`.

`RtensorArray(const RtensorArray& T, const device& dev)`
: Construct a copy of `T` on device `dev`.

`RtensorArray(const Gdims& adims, const Gdims& dims, [NBU], fill::view,float* arr, [DEVICE])`
: Construct a `RtensorArray` that is a view of the data at `arr`.

`RtensorArray(const Gdims& adims, const Gdims& dims, [NBU], std::function<float>`
`(const Gindex& aix,const Gindex& ix) fn, [DEVICE])`
: Construct an array of `adims` tensors of dimensions `dims` initialized by calling the function `fn` for each element of each cell. Here `aix` is the index of the cell in the array and `ix` is the index of the element in the cell.

STATIC CONSTRUCTORS

```
RtensorArray RtensorArray::zero(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
RtensorArray RtensorArray::ones(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
RtensorArray RtensorArray::identity(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
RtensorArray RtensorArray::sequential(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
RtensorArray RtensorArray::gaussian(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
```
: Create a new `RtensorArray` of array dimensions `adims` and cell dimensions `cdims` on device `DEVICE` with the given fill pattern and bundle size `NBU`.

TRANSPORT

`RtensorArray to(const device& dev) const`
`RtensorArray to_device(const int dev) const`
: Return a copy of the tensorarray on device `dev`.

`RtensorArray& move_to(const device& dev)`
`RtensorArray& move_to_device(const int dev)`
: Move the tensorarray to device `dev`.

ACCESS

`const Gdims& get_adims() const`
`int get_adim(const int i)`
Return the array dimensions of $[\![T]\!]$ or just the i'th array dimension.

`const Gdims& get_cdims() const`
`int get_cdim(const int i)`
Return the dimensions of each cell in $[\![T]\!]$ or just the i'th cell dimension.

`Rtensor get_cell(Gindex& aix) const`
`Rtensor get_cell(int i1,...,int ik) const`
Return the cell $[\![T]\!](i_1,\ldots,i_k)$.

`RtensorArray& set_cell(Gindex& aix, const Rtensor& A)`
`RtensorArray& set_cell(int i1,...,int ik, const Rtensor& A)`
Set $[\![T]\!](i_1,\ldots,i_k) = A$.

`Rtensor cell(Gindex& aix)`
`Rtensor cell(int i1,...,int ik)`
Return a view of cell $[\![T]\!](i_1,\ldots,i_k)$.

`const Rtensor cell(Gindex& aix) const`
`const Rtensor cell(int i1,...,int ik) const`
Return a `const` view of cell $[\![T]\!](i_1,\ldots,i_k)$.

VIEWS

`RtensorArray view()`
Return a view of `T`.

`RtensorArray view_of_slice(const int j)`
If `T` has array dimensions $(a_1,\ldots,a_k)$, this function returns a tensor-array with array dimensions $(a_2,\ldots,a_k)$ providing a view of the slice of `T` corresponding to its first index being set to j.

`RtensorArray view_of_slice(const Gdims& ix)`
If `T` has array dimensions $(a_1,\ldots,a_k)$ and `ix` is $(j_1,\ldots,j_p)$, this function returns a tensor-array with array dimensions $(a_{p+1},\ldots,a_k)$ providing a view of the slice of `T` corresponding to its first p indicies being set to $(j_1,\ldots,j_p)$.

RESHAPING

`RtensorArray ashape(const Gdims& adims) const`
Return a copy of $[\![T]\!]$ but with array shape changed to `adims`.

`RtensorArray& change_ashape(const Gdims& adims)`
Reinterpret $[\![T]\!]$ as an tensor-array of shape `adims`.

`RtensorArray& as_ashape(const Gdims& adims)`
Return a temporary object that reinterprets $[\![T]\!]$ as an tensor-array of shape `adims`.

ARRAY OPERATIONS

`RtensorArray broaden(int ix, int n) const`

Create a $d+1$ dimensional array by stacking n copies of $[\![T]\!]$ along dimension ix.

`RtensorArray reduce(int ix) const`

Create a $d-1$ dimensional array by summing out dimension `ix`.

CELLWISE ARITHMETIC

`c*T` `(RscalarArray,RtensorArray) -> RtensorArray`
`S+T` `(RtensorArray,RtensorArray) -> RtensorArray`
`S-T` `(RtensorArray,RtensorArray) -> RtensorArray`

`S*T` `(RtensorArray,RtensorArray) -> RtensorArray`

Compute the contracted product $R_{i_1,\ldots i_{k_1+k_2-2}} = \sum_j S_{i_1,\ldots i_{k_1-1},j}\, T_{j,i_{k_1},\ldots i_{k_1+k_2-2}}$ between each cell of $[\![S]\!]$ and the corresponding cell of $[\![T]\!]$.

CELL/ELEMENT ARITHMETIC

`S*T` `(Rtensor,RtensorArray) -> RtensorArray`
`T*S` `(Rtensor,RtensorArray) -> RtensorArray`

Multiply each cell of $[\![T]\!]$ by the same tensor S.

IN-PLACE OPERATIONS

`+=S` `RtensorArray -> RtensorArray`
`-=S` `RtensorArray -> RtensorArray`

Increment/decrement T by S.

OTHER FUNCTIONS

`transp(T)` `RtensorArray -> RtensorArray`

The transpose of `T`.

`stack(d,T1,...,Tm)` `(RtensorArray,...,RtensorArray) -> RtensorArray`

Stack the k'th order tensors $T_1, \ldots, T_m$ along dimension d into a $(k+1)$'th order tensor.

`cat(d,T1,...,Tm)` `(RtensorArray,...,RtensorArray,int) -> RtensorArray`

Concatenate $T_1, \ldots, T_m$ along dimension d.

`norm2(T)` `(RtensorArray,RtensorArray) -> Rscalar`

The squared Frobenius norm $\|T\|^2_{\mathrm{Frob}} = \sum_{i_1,\ldots i_k} |T_{i_1,\ldots,i_k}|^2$.

`inp(S,T)` `(Rtensor,Rtensor) -> Rscalar`
`odot(S,T)` `(Rtensor,Rtensor) -> Rscalar`

The inner product $\langle S,T\rangle = \sum_{i_1,\ldots i_k} S_{i_1,\ldots,i_k} T^*_{i_1,\ldots,i_k}$, and pointwise products $R_{i_1,\ldots i_k} = S_{i_1,\ldots,i_k} T_{i_1,\ldots,i_k}$.

`ReLU(z)` `Rtensor -> Rtensor`
`ReLU(z,alpha)` `(Rtensor,float) > Rtensor`

Apply the ReLU or leaky ReLU operator to each element of T.

I/O

`string str(const indent="") const`

Print the tensor to string with the optional indentation `indent`.

CtensorArray

A `CtensorArray` stores a multidimensional array $[\![T]\!]$ of `Ctensor` objects.

Implemented by: `CtensorArrayA` (derived from `CtensorA`)

CONSTRUCTORS

```
CtensorArray(const Gdims& adims, const Gdims& dims, [NBU], [FILLTYPE], [DEVICE])
```

Construct an `adims` sized array of `Ctensor`s, each of dimensions `dims`. `FILLTYPE` can be `fill::raw`, `fill::zero`, `fill::identity`, `fill::sequential` or `fill::gaussian`.

```
CtensorArray(const Ctensor& T)
CtensorArray(const Gdims& adims, const Ctensor& T)
```

Construct a single cell array from T or an array of size `adims` in which each cell is T.

```
CtensorArray(const CtensorArray& T, const view_flag& dummy)
```

Construct a view of `T`.

```
CtensorArray(const CtensorArray& T, const device& dev)
```

Construct a copy of `T` on device `dev`.

```
CtensorArray(const Gdims& adims, const Gdims& dims, [NBU], fill::view,
       float* arr, float* arrc, [DEVICE])
```

Construct a `CtensorArray` that is a view of the data at `arr` (real part) and `arrc` (imaginary part).

```
CtensorArray(const Gdims& adims, const Gdims& dims, [NBU], std::function<complex<float>>
         (const Gindex& aix,const Gindex& ix) fn, [DEVICE])
```

Construct an array of `adims` tensors of dimensions `dims` initialized by calling the function `fn` for each element of each cell. Here `aix` is the index of the cell in the array and `ix` is the index of the element in the cell.

STATIC CONSTRUCTORS

```
CtensorArray CtensorArray::zero(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
CtensorArray CtensorArray::ones(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
CtensorArray CtensorArray::identity(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
CtensorArray CtensorArray::sequential(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
CtensorArray CtensorArray::gaussian(const Gdims& adims, const Gdims& cdims, [NBU], [DEVICE])
```

Create a new `CtensorArray` of array dimensions `adims` and cell dimensions `cdims` on device `DEVICE` with the given fill pattern and bundle size `NBU`.

TRANSPORT

```
CtensorArray to(const device& dev) const
CtensorArray to_device(const int dev) const
```

Return a copy of the tensorarray on device `dev`.

`CtensorArray& move_to(const device& dev)`
`CtensorArray& move_to_device(const int dev)`
Move the tensorarray to device `dev`.

ACCESS

`const Gdims& get_adims() const`
`int get_adim(const int i)`
Return the array dimensions of $[\![T]\!]$ or just the i'th array dimension.

`const Gdims& get_cdims() const`
`int get_cdim(const int i)`
Return the dimensions of each cell in $[\![T]\!]$ or just the i'th cell dimension.

`Ctensor get_cell(Gindex& aix) const`
`Ctensor get_cell(int i1,...,int ik) const`
Return the cell $[\![T]\!](i_1,\ldots,i_k)$.

`CtensorArray& set_cell(Gindex& aix, const Ctensor& A)`
`CtensorArray& set_cell(int i1,...,int ik, const Ctensor& A)`
Set $[\![T]\!](i_1,\ldots,i_k)=A$.

`Ctensor cell(Gindex& aix)`
`Ctensor cell(int i1,...,int ik)`
Return a view of cell $[\![T]\!](i_1,\ldots,i_k)$.

`const Ctensor cell(Gindex& aix) const`
`const Ctensor cell(int i1,...,int ik) const`
Return a `const` view of cell $[\![T]\!](i_1,\ldots,i_k)$.

VIEWS

`CtensorArray view()`
Return a view of `T`.

`CtensorArray view_of_slice(const int j)`
If `T` has array dimensions $(a_1,\ldots,a_k)$, this function returns a tensor-array with array dimensions $(a_2,\ldots,a_k)$ providing a view of the slice of `T` corresponding to its first index being set to j.

`CtensorArray view_of_slice(const Gdims& ix)`
If `T` has array dimensions $(a_1,\ldots,a_k)$ and `ix` is $(j_1,\ldots,j_p)$, this function returns a tensor-array with array dimensions $(a_{p+1},\ldots,a_k)$ providing a view of the slice of `T` corresponding to its first p indicies being set to $(j_1,\ldots,j_p)$.

RESHAPING

`CtensorArray ashape(const Gdims& adims) const`
Return a copy of $[\![T]\!]$ but with array shape changed to `adims`.

`CtensorArray& change_ashape(const Gdims& adims)`
Reinterpret $[\![T]\!]$ as an tensor-array of shape `adims`.

`CtensorArray& as_ashape(const Gdims& adims)`
Return a temporary object that reinterprets $[\![T]\!]$ as an tensor-array of shape `adims`.

ARRAY OPERATIONS

`CtensorArray broaden(int ix, int n) const`
Create a $d+1$ dimensional array by stacking n copies of $[\![T]\!]$ along dimension ix.

`CtensorArray reduce(int ix) const`
Create a $d-1$ dimensional array by summing out dimension `ix`.

CELLWISE ARITHMETIC

`c*T` `(CscalarArray,CtensorArray) -> CtensorArray`
`S+T` `(CtensorArray,CtensorArray) -> CtensorArray`
`S-T` `(CtensorArray,CtensorArray) -> CtensorArray`

`S*T` `(CtensorArray,CtensorArray) -> CtensorArray`
Compute the contracted product $R_{i_1,\ldots i_{k_1+k_2-2}} = \sum_j S_{i_1,\ldots i_{k_1-1},j}\, T_{j,i_{k_1},\ldots i_{k_1+k_2-2}}$ between each cell of $[\![S]\!]$ and the corresponding cell of $[\![T]\!]$.

CELL/ELEMENT ARITHMETIC

`S*T` `(Ctensor,CtensorArray) -> CtensorArray`
`T*S` `(Ctensor,CtensorArray) -> CtensorArray`
Multiply each cell of $[\![T]\!]$ by the same tensor S.

IN-PLACE OPERATIONS

`+=S` `CtensorArray -> CtensorArray`
`-=S` `CtensorArray -> CtensorArray`
Increment/decrement T by S.

OTHER FUNCTIONS

`conj(T)` `CtensorArray -> CtensorArray`
`transp(T)` `CtensorArray -> CtensorArray`
`herm(T)` `CtensorArray -> CtensorArray`
The conjugate, transpose, and Hermitian conjugate of `T`.

`stack(d,T1,...,Tm)` `(CtensorArray,...,CtensorArray) -> CtensorArray`
Stack the k'th order tensors $T_1,\ldots,T_m$ along dimension d into a $(k+1)$'th order tensor.

`cat(d,T1,...,Tm)` `(CtensorArray,...,CtensorArray,int) -> CtensorArray`
Concatenate $T_1,\ldots,T_m$ along dimension d.

`norm2(T)` (CtensorArray,CtensorArray) -> Cscalar

The squared Frobenius norm $\|T\|^2_{\mathrm{Frob}} = \sum_{i_1,\ldots i_k} |T_{i_1,\ldots,i_k}|^2$.

`inp(S,T)` (Ctensor,Ctensor) -> Cscalar
`odot(S,T)` (Ctensor,Ctensor) -> Cscalar
`odotC(S,T)` (Ctensor,Ctensor) -> Cscalar

The inner product $\langle S,T\rangle = \sum_{i_1,\ldots i_k} S_{i_1,\ldots,i_k} T^*_{i_1,\ldots,i_k}$, and pointwise products $R_{i_1,\ldots i_k} = S_{i_1,\ldots,i_k} T_{i_1,\ldots,i_k}$ resp. $R_{i_1,\ldots i_k} = S_{i_1,\ldots,i_k} T^*_{i_1,\ldots,i_k}$

`ReLU(z)` Ctensor -> Ctensor
`ReLU(z,alpha)` (Ctensor,float) > Ctensor

Apply the ReLU or leaky ReLU operator to each element of T.

I/O

`string str(const indent="") const`

Print the tensor to string with the optional indentation `indent`.

Cell maps

Cellwise cell maps

Cellwise cell maps map an operation `OP` over all cells in a tensor array or all corresponding cells in multiple tensor arrays. Cellwise cell maps are implemented via `CellwiseUnaryCmap`, `CellwiseBinaryCmap` and `CellwiseTernaryCmap` classes, but usually invoked through one of the template functions below.

CREATOR TEMPLATE FUNCTIONS

`ARR cellwise<OP>(const ARR& A)`
: Return a new array of the same size as `A`, with $[\![R]\!]_{i_1,\ldots,i_k} = \mathrm{OP}([\![A]\!]_{i_1,\ldots,i_k})$.

`ARR cellwise<OP>(const ARR& A, const ARR& B)`
: Return a new array `R` of the same size as `A`, with $[\![R]\!]_{i_1,\ldots,i_k} = \mathrm{OP}([\![A]\!]_{i_1,\ldots,i_k}, [\![B]\!]_{i_1,\ldots,i_k})$.

CUMULATIVE TEMPLATE FUNCTIONS

`void add_cellwise<OP>(ARR& R, const ARR& A)`
: For each cell of `R`, set $[\![R]\!]_{i_1,\ldots,i_k} \leftarrow [\![R]\!]_{i_1,\ldots,i_k} + \mathrm{OP}([\![A]\!]_{i_1,\ldots,i_k})$.

`void add_cellwise<OP>(ARR& R, const ARR& A, const ARR& B)`
: For each cell of `R`, set $[\![R]\!]_{i_1,\ldots,i_k} \leftarrow [\![R]\!]_{i_1,\ldots,i_k} + \mathrm{OP}([\![A]\!]_{i_1,\ldots,i_k}, [\![B]\!]_{i_1,\ldots,i_k})$.

Broadcast cell maps

Broadcast cell maps map an operation over the cells of one or more arrays, but one or more arguments is a fixed individual cell object rather than an entire array. Broadcast cell maps are implemented via the `BroadcastUnaryCmap`, `BroadcastBinaryCmap`, but usually invoked through one of the template functions below.

CREATOR TEMPLATE FUNCTIONS

`broadcast<OP>(const OBJ& M, const ARR& A)`
: Return a new array of the same size as `A` with $[\![R]\!]_{i_1,\ldots,i_k} = \mathrm{OP}(M, A_{i_1,\ldots,i_k})$.

`broadcast<OP>(const ARR& A, const OBJ& M)`
: Return a new array of the same size as `A` with $[\![R]\!]_{i_1,\ldots,i_k} = \mathrm{OP}(A_{i_1,\ldots,i_k}, M)$.

CUMULATIVE TEMPLATE FUNCTIONS

`void add_broadcast<OP>(ARR& R, const OBJ& M)`
: For each cell of `R`, set $[\![R]\!]_{i_1,\ldots,i_k} \leftarrow [\![R]\!]_{i_1,\ldots,i_k} + \mathrm{OP}(M)$.

`void add_broadcast<OP>(ARR& R, const OBJ& M, const ARR& A)`
: $[\![R]\!]_{i_1,\ldots,i_k} \leftarrow [\![R]\!]_{i_1,\ldots,i_k} + \mathrm{OP}(M, A_{i_1,\ldots,i_k})$.

`void add_broadcast<OP>(ARR& R, const ARR& A, const OBJ& M)`
: $[\![R]\!]_{i_1,\ldots,i_k} \leftarrow [\![R]\!]_{i_1,\ldots,i_k} + \mathrm{OP}(A_{i_1,\ldots,i_k}, M)$.

Inner cell maps

The inner cell map multiplies applies an operator `OP` to corresponding cells of two tensor arrays and sums the result. It is implemented by the class `InnerCmap`, but usually invoked via the following template functions.

CREATOR TEMPLATE FUNCTIONS

`ARR inner<OP>(const ARR& X, const ARR& Y)`
Return a tensor-array consisting of the single cell $[\![R]\!]_0 = \sum_i \mathrm{OP}([\![A]\!]_i, [\![B]\!]_i)$

CUMULATIVE TEMPLATE FUNCTIONS

`add_inner<OP>(ARR& R, const ARR& A, const ARR& B)`
Set $[\![R]\!]_0 \leftarrow [\![R]\!]_0 + \sum_i \mathrm{OP}([\![A]\!]_i, [\![B]\!]_i)$

Outer cell maps

CREATOR TEMPLATE FUNCTIONS

```
ARR outer<OP>(const ARR& X, const ARR& Y)
```

Return a two dimensional tensor-array with $[\![R]\!]_{i,j} = \mathrm{OP}([\![A]\!]_i, [\![B]\!]_j)$.

CUMULATIVE TEMPLATE FUNCTIONS

```
void add_outer<OP>(ARR& R, const ARR& A, const ARR& B)
```

Set $[\![R]\!]_{i,j} \leftarrow [\![R]\!]_{i,j} + \mathrm{OP}([\![A]\!]_i, [\![B]\!]_j)$.

Matrix–vector product cell maps

CREATOR TEMPLATE FUNCTIONS

`ARR MVprod<OP>(const ARR& X, const ARR& Y)`

Return a two dimensional tensor-array with $[\![R]\!]_i = \sum_j \mathrm{OP}([\![A]\!]_{i,j}, [\![B]\!]_j)$.

CUMULATIVE TEMPLATE FUNCTIONS

`add_MVprod<OP>(ARR& R, const ARR& A, const ARR& B)`

Set $[\![R]\!]_i \leftarrow [\![R]\!]_i + \sum_j \mathrm{OP}([\![A]\!]_{i,j}, [\![B]\!]_j)$.

Vector–matrix product cell maps

CREATOR TEMPLATE FUNCTIONS

`ARR VMprod<OP>(const ARR& X, const ARR& Y)`

Return a two dimensional tensor-array with $[\![R]\!]_i = \sum_j \mathrm{OP}([\![A]\!]_i, [\![B]\!]_{i,j})$.

CUMULATIVE TEMPLATE FUNCTIONS

`void add_VMprod<OP>(ARR& R, const ARR& A, const ARR& B)`

Set $[\![R]\!]_i \leftarrow [\![R]\!]_i + \sum_j \mathrm{OP}([\![A]\!]_i, [\![B]\!]_{i,j})$.

1D convolutional cell maps

CREATOR TEMPLATE FUNCTIONS

`ARR convolve1<OP>(const ARR& A, const ARR& B)`
Return a new array `R` of the dimensions (I) with $[\![R]\!]_i = \sum_{j=0}^{J-1} \mathrm{OP}([\![A]\!]_{i+j}, [\![B]\!]_j)$.

CUMULATIVE TEMPLATE FUNCTIONS

`void add_convolve1<OP>(ARR& R, const ARR& A, const ARR& B)`
For each cell of `R`, set $[\![R]\!]_i \leftarrow [\![R]\!]_i + \sum_{j=0}^{J-1} \mathrm{OP}([\![A]\!]_{i+j}, [\![B]\!]_j)$.

2D convolutional cell maps

CREATOR TEMPLATE FUNCTIONS

`ARR convolve2<OP>(const ARR& A, const ARR& B)`

Return a new array `R` of the dimensions (I_1, I_2) with $[\![R]\!]_{i,j} = \sum_{j_1=0}^{J_1-1} \sum_{j_2=0}^{J_2-1} \mathrm{OP}([\![A]\!]_{i_1+j_1,\, i_2+j_2}, [\![B]\!]_{j_1,j_2})$.

CUMULATIVE TEMPLATE FUNCTIONS

`void add_convolve2<OP>(ARR& R, const ARR& A, const ARR& B)`

For each cell of `R`, set $[\![R]\!]_{i,j} \leftarrow [\![R]\!]_{i,j} + \sum_{j_1=0}^{J_1-1} \sum_{j_2=0}^{J_2-1} \mathrm{OP}([\![A]\!]_{i_1+j_1,\, i_2+j_2}, [\![B]\!]_{j_1,j_2})$.

Cell operators

Cell operators specify how to perform a given operation on individual cells of a tensor-array, but are written in such as way that cell maps can execute the operation in parallel on all cells in an array or different combinations of cells, both on the CPU and the GPU.

BinaryCop<OBJ,ARR>

`BinaryCop` is the abstract base class of binary cell operators in cnine. The `OBJ` template argument is the type of the cells, while `ARR` is the type of the corresponding array object.

METHODS

`void apply(OBJ& r, const OBJ& x, const OBJ& y, int add_flag=0) const`
: Compute $\mathrm{OP}(x, y)$ and place the result in r. If `add_flag=1`, the result will be added to current value of `r`, otherwise it will overwrite it.

`void apply(const CMAP& map, ARR& R, const ARR& X, const ARR& Y, int add_flag=0) const`
: Use the cell map `CMAP` to map `OP` over the cells of `x` and `y` and place the result in `R`. If `add_flag=1`, the result will be added to current value of `R`, otherwise it will overwrite it.

`void accumulate(const CMAP& map, ARR& R, const ARR& X, const ARR& Y, int add_flag=0) const`
: Use the cell map `CMAP` to map `OP` over the cells of `x` and `y` and place the result in `R`. If `add_flag=1`, the result will be added to current value of `R`, otherwise it will overwrite it.

Helper classes

cnine_session

Any program using cnine must define a single cnine_session object, which is initialized before any other calls are made to the library, and is not destroyed until essentially the end of the program. There purpose of this object is to initialize various internal variables, the CUBLAS context, and various other global variables/resources.

CONSTRUCTOR AND DESTRUCTOR

`cnine_session()`
: Create the cnine_session object and initialize the necessary internal resources.

`~cnine_session()`
: Shut down the cnine system.

Gdims

The `Gdims` class is used to store the dimensions of tensors and matrices.

Derived from: `vector<int>`

CONSTRUCTORS

`Gdims(d0,...)`
: Create a new `Gdims` object initialized to $(d_0, d_1, \ldots, d_{k-1})$.

`Gdims(const int k, const fill_raw& fill)`
: Create a new k'th order `Gdims` object, but with the dimensions uninitialized.

`Gdims(const Gdims& a, const Gdims& b)`
: Concatenate `a` and `b` into a single `Gdims` object.

ACCESS

`int k() const`
: Return the number of dimensions, k.

`operator(int i) const`
: Return the i'th dimension, d_i.

`int asize() const`
: Return the the total number of elements, $\prod_{i=0}^{k-1} d_i$.

METHODS

`Gdims prepend(const int d) const`
: Prepend d to the list of dimensions.

`Gdims append(const int d) const`
: Append d to the list of dimensions.

`Gdims insert(const int i, const int d) const`
: Insert d in position i.

`Gdims remove(const int i) const`
: Remove the i'th dimension.

`Gdims chunk(const int i, const int k) const`
: Return $(d_i, d_{i+1}, \ldots, d_{i+k-1})$.

`Gdims Mprod(const Gdims& D1, const Gdims& D2) const`
: Return the dimensions of the tensor that results from taking the matrix-like product of a tensor of dimensions D_1 with a tensor of dimensions D_2.

I/O

`string str() const`

Print $(d_1, \ldots, d_k)$ to string.

RELATED FUNCTIONS

`Gdims dims(d0,...)`

Return a new `Gdims` object initialized to $(d_0, d_1, \ldots, d_{k-1})$.

Gindex

`Gindex` stores the indices of a given element in a tensor or of a given cell in an object array.

Derived from: `vector<int>`

CONSTRUCTORS

`Gindex(i0,...i`$_k$`))`
: Construct the index vector $(i_1, \ldots, i_k)$.

`Gindex(const int k, const fill_zero& dummy)`
: Construct a k element index vector with all indices initialized to zero.

ACCESS

`int k() const`
: Return the number of indices, k.

`operator(int j) const`
: Return the j'th index, i_j.

INTERACTING WITH TENSORS/ARRAYS

`int operator()(const Gdims& dims) const`
`int operator()(cost vector<int> strides) const`
: Return the linear index of the element corresponding to this vector in a tensor/array of dimensions `dims` or strides `strides`.

`Gindex(const int p, const Gdims& dims) const`
`Gindex(const int p, const vector<int>& strides) const`
: Return the index vector of the element with linear index p in an array/tensor of dimensions `dims` or strides `strides`.

I/O

`string str() const`
: Print to string.

Gtensor<TYPE>

`Gtensor<TYPE>` is a simple helper tensor class, mainly intended to be used with data loading and preprocessing. `Gtensor<TYPE>` does have the multithreading and batching capabilities of the library's main tensor class, `Ctensor`, but has the advantage of offering a range of convenient member functions.

CONSTRUCTORS

`Gtensor(const Gdims& dims, const device_id& dev=0)`
: Create a new `Gtensor` A of size `dims` on device `dev`.

`Gtensor(const Gdims& _dims, fill::raw, [DEVICE])`
`Gtensor(const Gdims& dims, fill::zero, [DEVICE])`
`Gtensor(const Gdims& dims, fill::identity, [DEVICE])`
`Gtensor(const Gdims& dims, fill::sequential, [DEVICE])`
`Gtensor(const Gdims& dims, fill::gaussian, [DEVICE])`
: Create a new `Gtensor` of size `dims` on device `DEVICE` with entries that are (a) uninitialized; (b) initialized to zero; (c) initialized to the identity matrix; (d) initialized with the numbers $0, 1, 2, \ldots$; (e) initialized with IID normal distributed numbers.

ACCESS

`TYPE operator()(i1,...,ik) const`
`get(i1,...,ik) const`
: Return the $(i_1, \ldots, i_k)$ element of the tensor.

`TYPE& operator()(i1,...,ik)`
: Return a reference to the $(i_1, \ldots, i_k)$ element of the tensor.

`TYPE& set(i1,...,ik, const TYPE v)`
: Set the $(i_1, \ldots, i_k)$ element to v.

IN-PLACE OPERATORS

`operator+=(const Gtensor<TYPE>& B)`
: Add B to A.

`operator-=(const Gtensor<TYPE>& B)`
: Subtract B from A.

`operator*=(const TYPE c)`
: Multiply A by c.

`operator/=(const TYPE c)`
: Divide each element of A by c.

OPERATORS

`Gtensor<TYPE> operator+(const Gtensor<TYPE>& B)`
Return $A+B$.

`Gtensor<TYPE> operator-(const Gtensor<TYPE>& B)`
Return $A-B$.

`Gtensor<TYPE> operator*(const TYPE c)`
Return cA.

`Gtensor<TYPE> abs()`
Apply `abs` to each element.

`Gtensor<TYPE> conj()`
Return the conjugate tensor.

`Gtensor<float> real()`
Return the real part of the tensor.

I/O

`string str() const`
Print the tensor to string.