Cengine

Asynchronous C++ CPU/GPU compute engine, v0.0

Risi Kondor
Flatiron Institute NYC, and
The University of Chicago

Contents

Abstract

Cengine is a lightweight compute engine designed to parallelize numerical computations at run time by

(a) distributing computation across multiple CPU threads and/or
(b) batching together operations of the same kind for parallel execution on the GPU.

Cengine employs the delayed execution model of computation to decouple user code from worker threads, and maintains an internal dependency graph to ensure correctness.

From the user code side, the engine expects a sequence of simple instructions. For example,

```
c=engine.push<ctensor_add_op>(a,b)
```

tells the engine to add tensors `a` and `b` and store the result in `c`. Instead of executing this instruction directly, Cengine queues the corresponding `ctensor_add_op` internally, and executes it later, when either a CPU threads becomes available or a sufficient number of operations of the same type have accumulated to make executing them on the GPU in batch economical.

Cengine offers a small collection of built-in data types to represent real/complex valued scalar and tensor objects, and a corresponding complement of basic arithmetic and linear algebra operators. However, the engine is primarily designed to be used with user defined classes and operators.

Cengine is written in standard C++11 and requires no other libraries besides the standard template library and CUDA/CUBLAS for GPU functionality.

Overview

The most important class in Cengine is the compute engine itself, `Cengine::Cengine`. Normally a single instance of this class is initialized at startup and keeps running until the program terminates. For example,

```
Cengine::Cengine engine(4)
```

initializes a compute engine with 4 CPU threads.

User level code does not have direct access to the data objects managed by the engine. Rather, it issues commands to the engine via the engine's `push` method. The `push` method returns a pointer to a handle which can subsequently be used to reference the resulting object. For example,

```
Chandle* A=engine.push<new_ctensor_gaussian>({3,3});
```

instructs the engine to create a new 3×3 complex matrix filled with random numbers drawn IID from the standard normal distribution. Issuing the command

```
Chandle* B=engine.push<ctensor_add>(A,A);
```

adds `A` to itself and stores the result in an object referenced by `B`.

Asynchronous execution

Cengine follows the asynchronous, delayed execution model of computation. This means that most commands are not executed when they are issued, but at some later point in time, when, depending on context, either a CPU thread becomes available, or a sufficient number of operations of the same type have accumulated for execution on the GPU. The order in which the operations are executed need not be the same as the order that they were issued to the engine. To ensure correctness, the engine keeps track of dependencies between operations internally in the form of a directed acyclic graph (DAG).

Delayed execution implies that the `Chandle` objects returned by the engine do not point to the actual result of the computation, but only to where the result will eventually appear. To correctly manage this, user level code will typically encapsulate `Cengine` calls in a separate set of classes. For example, the user might define a `ComplexMatrix` class which has a member variable `hdl` to store the handle returned by engine. To implement in-place matrix addition, the user will add a member function

```
ComplexMatrix& ComplexMatrix::operator+(const ComplexMatrix& B){
   Chandle* t=engine.push<ctensor_add>(hdl,B.hdl);
   delete hdl;
   hdl=t;
}
```

Of course eventually the result of any given sequence of computations does have to be extracted from the engine. For this we use commands that are *blocking*, meaning that the calling function will wait until the result has actually been computed. For example, the command

```
Gtensor M=engine.push<ctensor_get>(B);
```

returns the value of `B` in a user side tensor object `M`. This command requires explicitly materializing `B`, therefore it waits until all computations leading up to `B` are complete and `B` has been computed as well.

Calling

```
cengine::flush(B);
```

has a similar effect, while

```
cengine::flush();
```

flushes all pending operations.

Cengine automatically takes care of memory management. For any given backend object `xobj`, when there are no operations pending that take `xobj` as an argument and no user side handles pointing to `xobj`, the object is scheduled for deletion. When the `Cengine` is deleted or shut down, all pending operations are flushed and all backend objects are destroyed.

Operators

Commands in Cengine are implemented as operators, and each command must have a corresponding class. The template argument of `Cengine::push` command is the name of this operator class. The abstract base class of all operator classes is `Cengine::Coperator`. For example, the internal definition of the `ctensor_add_op` operator (in slightly abbreviated form) is

```
class ctensor_add_op:  public Coperator, public CumulativeOperator{
public:

   using Coperator::Coperator;

   void exec(){
      owner->obj=inputs[0]->obj;
      asCtensorB(owner).add(asCtensorB(inputs[1]));
   }

   string str() const{
      return "ctensor_add"+inp_str();
   }

};
```

The `exec` method is responsible for carrying out the actual operation on the operator's arguments. In the case of `ctensor_add_op` this just amounts to calling the backend object's method for tensor summation. However, in user-defined operators the `exec` method can often be significantly more involved. Adding a new operator to Cengine just requires defining the corresponding operator class.

The concrete data object that `ctensor_add_op` operates on is a `Cengine::CtensorB`, which is the backend container for `ctensor` objects. The built-in `rscalar, scalar` and `ctensor` classes, extended by user defined operators are sufficient for many purposes. However, there is nothing stopping the user from adding new backend classes as well, simply by subclassing the `Cengine::Cobject` abstract class. Cengine can manage any type of user defined backend object, as long as it is derived from `Cengine::Cobject`, and any type of user defined operator, as long as it is derived from `Cengine::Coperator`. Adding new objects and operators does not require making any changes to Cengine itself.

Batched operators

Batching refers to accumulating multiple instances of the same operation and executing them together, in parallel. Batching is particularly important efficiently utilizing graphics processor units (GPUs), since GPU threads are generally tied: on NVIDIA architectures, for example, all threads running on the same streaming multiprocessor must essentially be executing the the exact same machine level instruction at any given time. Some types of computations, such as solving a systems of partial differential equations on a regular grid are well suited to this paradigm, since the operations that need to be performed at each gridpoint are the same.

Other types of computations, however, are much less structured. In a graph neural network, for example, the operation performed at each node depends on the number of neighbors. In principle, it is possible to write code that separately parallelizes over all nodes with just one neighbor, all nodes with two neighbors, and so on, but in practice such low level multithreading is laborious and highly error prone.

One solution that has emerged is *dynamic batching*, which refers to accumulating operations of each type and executing them together as a batch. Taking dynamic batching too far can lead to situations where a large number of batched operations are mutually waiting on each other and none of the batches are actually run. Therefore, as a general principle, it is best to use dynamic batching sparingly, on a small set of frequent operations that are expensive enough to be performance critical, yet small enough that executing the operations individually (without batching) would waste much of the GPU's parallel processing power. Basic matrix operations, such as matrix/scalar and matrix/matrix products are good candidates for batching. Accessing individual components of matrices, however, is not an operation that would likely benefit from dynamic batching.

Cengine will attempt to dynamically batch any operator derived from the `BatchedOperator` class. In addition to the `exec()` method, batched operators must also have a `batched_exec` method, which takes a *vector* of pointers to compute graph nodes as its argument, and executes each node in parallel. For each batched operator class `UserOp1`, the engine will internally create a separate `BatcherA<UserOp1>` object to manage the batching process. To keep track of the correspondence between operators and batchers, each batched operator class must provide a static integer `batcher_id` variable.

Meta-batchers

Many batched operators require separate batchers for different settings of their parameters. For example, in order to batch matrix multiplication, we need separate batchers for each combination of input matrix dimensions. Cengine makes it easy to implement such *multi-batched* operators by introducing batcher signatures and the `MetaBatcher` class.

Any multi-batched class must have a corresponding signature type. For example the signature class of the matrix multiplication operator is `Mprod_signature`, which stores the dimensions of the two matrices to be multiplied and some flags to signify if either matrix is transposed. The matrix multiplication operator `ctensor_Mprod_op` must have a `signature()` method that returns the signature object corresponding to the given pair of matrices to be multiplied. The engine will then create a separate batcher object for each distinct signature.

The purpose of the `MetaBatcher` class (for matrix multiplication) is to route individual matrix products to their corresponding batcher. All that the operator class needs to do enable this process is provide a `spawn_batcher()` method that creates the appropriate templated `MetaBatcher` object. In the case of our example the type of this (in slightly abbreviated form) would be

```
MetaBatcher< ctensor_add_Mprod_op, Mprod_signature, BatcherA<ctensor_add_Mprod_op> >.
```

Built-in types

While Cengine is primarily designed to be used with user-defined data classes and operators, it does provides three built-in “starter” types corresponding to real and complex scalars/tensors:

```
rscalar
cscalar
ctensor .
```

To enable fast GPU computation, each of these classes is implemented in single precision arithmetic (`float`). The corresponding back-end classes are `RscalarB, CscalarB` and `CtensorB`. Each of these types is equipped with a minimal set of arithmetic and linear algebra operators.

Data layout GPU functionality, and bundles

Similarly to deep learning frameworks such as `PyTorch` and `TensorFlow`, Cengine’s built in objects can be flexibly moved back and forth between the host and the GPUs. This is done by the `to_device(d)` command, where `d` is the identifier of the GPU, or `0` in case that the object is to be moved back to the host.

In general, every backend operation must have two separate implementations: one for execution on the CPU, and once for execution on the GPU written in CUDA or CUBLAS. Whether a given operation is executed on the CPU or GPU depends on where its arguments reside: in general, Cengine will move all input objects to the same device as where the first input argument resides and perform the operation on that device.

The storage layout of the built in classes is optimized for GPU computation. In particular, matrix/tensor objects are padded to multiples of 32 `float`s, and complex tensor are stored with their real and imaginary parts separate. Cengine uses a row-major matrix/tensor storage format.

The simplest type of parallelism is multiplexing a single operation over n “channels”. In systems such as `PyTorch` this is done by adding an outer “batch” dimension to each operand. To avoid confusing this with “dynamic batching”, in Cengine the corresponding concept is called *bundles*. Any `rscalar`, `scalar` or `ctensor` object is thus allowed to have a “bundle dimension” n_{bu}. Thus, a single `rscalar` object for example can actually store not just on, but n_{bu} different scalars, operated on independently. The CUDA/CUBLAS backend efficiently parallelizes over the bundle dimension.

Installation

Cengine is a header-only library that does not require separately compiling any object files. Assuming that the library's root directory is `$(CENGINE_ROOT)`, the core header files are located in `$(CENGINE_ROOT)/include` and `$(CENGINE_ROOT)/engine`. In addition, if the user wishes to use the built in scalar and tensor classes, they must also include `$(CENGINE_ROOT)/backend/scalar` and `$(CENGINE_ROOT)/backend/tensor`. Thus, to compile your own code `foo.cpp` with Cengine the compiler might be invoked as for example

```
c++ -o foo foo.cpp -std=c++11 -lstdc++11 -lm -I$(CENGINE_ROOT)/include
-I$(CENGINE_ROOT)/engine $(CENGINE_ROOT)/backend/scalar $(CENGINE_ROOT)/backend/tensor.
```

Some systems require also linking the pthreads library as `-lpthreads`.

The top level executable of any code compiled with Cengine must include `Cengine.cpp`, which defines certain global variables and starts the compute engine.

Compile time flags and variables

Use `#define` to set any of the following compile time flags.

Flag	default	
DEBUG_ENGINE_FLAG	off	When set, the engine will print detailed diagnostic information as it runs.

Use `#define` to set the value of any of the following compile time variables.

Flag	default	
CENGINE_NWORKERS	4	Default number of CPU worker threads.

Reference

Core classes

Cengine

Cengine is the library's central class, responsible for scheduling operations demanded by the user code. Typically a single Cengine instance is initialized at startup and remains running until the program terminates.

CONSTRUCTORS

`Cengine()`
`Cengine(const int n)`
Start an Cengine with n CPU worker threads. If n is not specified, it is set to CENGINE_NWORKERS.

METHODS

`push<OPERATOR>(Chandle* x0, ...Chandle* xk, const PARAM0 p0, ...const PARAMM pm)`
Enqueue the operation OPERATOR with arguments $x_0, \ldots, x_k$ and parameters $p_0, \ldots, p_m$ on the engine. Calls made through this method are the main mechanism for communicating with the engine.

`flush(const Chandle& x)`
Expedite operations leading up to computing x and block until x has been computed.

`flush()`
Flush all operations curretnly enqueued on the engine, including batched operations. Control blocks until all computations are complete.

Chandle

`Chandle` objects are used in user code to refer to objects managed by the engine. In particular, most `push` commands return a `Chandle` to the object created, and, in turn, command arguments are passed to the engine in the form of `Chandle`s.

`Chandle`s are also critical for the engine's memory management. As long as there is at least one `Chandle` pointing to a given backend object, the object is not deleted. On the other hand, when the last `Chandle` referencing a given object is deleted and there are no operations enqued in the engine dependging on the object as an argument either, the object is automatically scheduled for deletion.

Coperator

`Coperator` is the virtual base class of all operator objects that the `Cengine` can accept in its `push` method. New operators are defined simply by subclassing `Coperator`.

MEMBER VARIABLES

`Cnode* owner`
: Pointer to the compute graph node associated with this operation.

`vector<Cnode*> inputs`
: Vector of pointes to the compute graph nodes corresponding to the inputs of this operation.

METHODS

`virtual void exec()=0`
: The `exec` member function carries out the actual operation. This function is run by one of the CPU worker threads after all the inputs of the operator have already been computed. Therefore the body of the function can assume `inputs[0]->obj,...,inputs[k]->obj` point to all the correct input objects.

`virtual string str() const=0`
: Return a human readable representation of the operator for debugging purposes.

BatchedOperator

`BatchedOperator` is the virtual base class of all `Coperator`s that are batchable. New batchable operators are defined by subclassing `BtachedOperator`.

Derived from: `Coperator`

MEMBER VARIABLES

`Cnode* owner`
Pointer to the compute graph node associated with this operation.

`vector<Cnode*> inputs`
Vector of pointes to the compute graph nodes corresponding to the inputs of this operation.

METHODS

`virtual int batcher_id() const=0`
Return the index of this operator class. The index is a static variable of the operator class that is set by the engine itself.

`virtual void set_batcher_id(const int i=0`
Set the index of this operator class to i. This function is used by the engine to set the index of the operator class, the first time the operator is encountered.

`virtual void exec()=0`
As in the `Coperator` base class, this function defines the operator's operation when executed on data objects individually (not batched).

`virtual void batched`$_{e}xec() = 0$
This function defines the operator's operation when executed on a batch of inputs.

`SIGNATURE signature() const=0`
Return a `SIGNATURE` object that captures the given operator's signature.

`Batcher* spawn_batcher() const=0`
Create a new `Batcher` object for this operator.

I/O

`virtual string str() const=0`
Return a human readable representation of the operator for debugging purposes.

Built-in types and operators

Cengine provides three built-in "virtual" data types:

- `rscalar` to represent real valued scalars;
- `cscalar` to represent complex valued scalars;
- `ctensor` to represent complex valued tensors and matrices.

All three use single precision arithmetic and support parallelism through bundles. The types are "virtual" in the sense that corresponding can only be created and accessed by pushing the appropriate operators to the `Cengine`. The actual backend classes corresponding to the three types are `RscalarB`, `CscalarB` and `CtensorB`, but these are not directly accessible to the user.

The following pages list the corresponding operators in function format. For example, the command issued to the engine to add a real scalar y to another real scalar x and store the resulting handle in hdl is

```
Chandle* hdl=engine.push<rscalar_add_op>(xhdl,yhdl).
```

In the documentation this would appear simply as

```
rscalar_add_op(const rscalar& x, const rscalar& y),
```

since `xhdl` and `yhdl` are Chandle objects pointing to `rscalar`s.

rscalar

The `rscalar` virtual type is used to represent single precision real valued scalars. An `rscalar` object can store a single real number or a bundle of n_{bu} real numbers. User level code can access `rscalar` objects by using the following operators.

CONSTRUCTORS

`new_rscalar_op(const int nbu=-1, const int dev=0)`
`new_rscalar_zero_op(const int nbu=-1, const int dev=0)`
`new_rscalar_set_op(const int nbu=-1, const float x, const int dev=0)`
`new_rscalar_gaussian_op(const int nbu=-1, const int dev=0)`

Construct a new `rscalar` object with bundle size `nbu` on device `dev`. The four cases correspond to the object being uninitialized, initialized to zero, initialized to x, or initialized with random standard normal entries. `nbu=-1` signifies that the object is not bundled and `dev=0` is the host.

`rscalar_copy_op(const rscalar& x)`

Create a new `rscalar` by copying `x`.

IN-PLACE OPERATORS

`rscalar_set_zero_op(const rscalar& x)`

Set `x` to zero.

CUMULATIVE OPERATORS

`rscalar_add_op(const rscalar& r, const rscalar& x)`

Set $r \leftarrow r + x$.

`rscalar_subtract_op(const rscalar& r, const rscalar& x)`

Set $r \leftarrow r - x$.

`rscalar_add_prod_op(const rscalar& r, const rscalar& x, const rscalar& y)`

Set $r \leftarrow r + xy$.

`rscalar_add_div_op(const rscalar& r, const rscalar& x, const rscalar& y)`

Set $r \leftarrow r + x/y$.

`rscalar_add_abs_op(const rscalar& r, const rscalar& x)`

Set $r \leftarrow r + |\,x\,|$.

`rscalar_add_exp_op(const rscalar& r, const rscalar& x)`

Set $r \leftarrow r + e^x$.

`rscalar_add_pow_op(const rscalar& r, const rscalar& x, const float p, const float c)`

Set $r \leftarrow r + c\,x^p$.

`rscalar_add_ReLU_op(const rscalar& r, const rscalar& x, const float c)`

Set $r \leftarrow r + x$ if $x \geq 0$, otherwise set $r \leftarrow r + cx$.

`rscalar_add_sigmoid_op(const rscalar& r, const rscalar& x)`

Set $r \leftarrow r + 1/(1+e^{-x})$.

BACKWARD OPERATORS

The following "backward" operators are for use in automatic differentiation.

```
rscalar_add_div_back1_op(const rscalar& r, const rscalar& g, const rscalar& x,
                                                                   const rscalar& y)
```

Set $r \leftarrow r - g\,x/y^2$.

```
rscalar_add_pow_back_op(const rscalar& r, const rscalar& x, const rscalar& g, const float p,
                                                                        const float c)
```

Set $r \leftarrow r + c\,g\,x^p$.

```
rscalar_add_abs_back_op(const rscalar& r, const rscalar& g, const rscalar& x)
```

Set $r \leftarrow r + g$ if $x \geq 0$ and $r \leftarrow r - g$ otherwise.

```
rscalar_add_ReLU_back_op(const rscalar& r, const rscalar& g, const rscalar& x, const float c)
```

Set $r \leftarrow r + g$ if $x \geq 0$, otherwise $r \leftarrow r + cg$.

```
rscalar_add_sigmoid_back_op(const rscalar& r, const rscalar& g, const rscalar& x)
```

Set $r \leftarrow r + g\,x/(1-x)$.

BLOCKING FUNCTIONS

The following functions are called directly (as opposed to being pushed to the engine as an operator).

```
vector<float> rscalar_get(const rscalar& x)
```

Flush `x` and return its value(s) in an `std::vector`.

cscalar

The cscalar virtual type is used to represent single precision complex valued scalars. A cscalar object may either store a single complex number or a bundle of n_{bu} complex numbers. User level code can access cscalar objects using the following operators.

CONSTRUCTORS

new_cscalar_op(const int nbu=-1, const int dev =0)
new_cscalar_zero_op(const int nbu=-1, const int dev=0)
new_cscalar_set_op(const int nbu=-1, const complex<float> z, const int dev=0)
new_cscalar_gaussian_op(const int nbu=-1, const int dev=0)

Construct a new cscalar object with nbu bundles on dev. The four cases correspond to the object being uninitialized, initialized to zero, initialized to z, or initialized with random standard normal entries. nbu=-1 signifies that the object is not bundled and device=0 is the host.

cscalar_copy_op(const cscalar& x)

Create a new cscalar by copying x.

IN-PLACE OPERATORS

cscalar_set_zero_op(const cscalar& r)

Set r to zero.

NOT IN-PLACE OPERATORS

cscalar_conj_op(const cscalar& z)

Return $\overline{z}$.

cscalar_get_real_op(const cscalar& z)

Return the real part of z.

cscalar_get_imag_op(const cscalar& z)

Return the imaginary part of z.

CUMULATIVE OPERATORS

cscalar_add_op(const cscalar& r, const cscalar& z)

Set $r \leftarrow r + z$.

cscalar_subtract_op(const cscalar& r, const cscalar& z)

Set $r \leftarrow r - z$.

cscalar_add_prod_r_op(const cscalar& r, const cscalar& x, const rscalar& y)

Set $r \leftarrow r + xy$.

cscalar_add_prod_r_op(const cscalar& r, const cscalar& x, const rscalar& y)

Set $r \leftarrow r + xy$.

cscalar_add_prodc_op(const cscalar& r, const cscalar& x, const cscalar& y)

Set $r \leftarrow r + x\overline{y}$.

```
cscalar_add_prodcc_op(const cscalar& r, const cscalar& x, const cscalar& y)
```
Set $r \leftarrow r + \overline{xy}$.

```
cscalar_add_div_op(const cscalar& r, const cscalar& x, const cscalar& y)
```
Set $r \leftarrow r + x/y$.

```
cscalar_add_to_real_op(const cscalar& r, const rscalar& x)
```
Set $r \leftarrow r + (x, 0)$.

```
cscalar_add_to_imag_op(const cscalar& r, const rscalar& x)
```
Set $r \leftarrow r + (0, x)$.

```
cscalar_add_abs_op(const cscalar& r, const cscalar& z)
```
Set $r \leftarrow r + |\, z\,|$.

```
cscalar_add_exp_op(const cscalar& r, const cscalar& z)
```
Set $r \leftarrow r + e^z$.

```
cscalar_add_pow_op(const cscalar& r, const cscalar& z, const float p, const float c)
```
Set $r \leftarrow r + c\, z^p$.

```
rscalar_add_ReLU_op(const rscalar& r, const rscalar& z, const float c)
```
Apply the soft-ReLU operator to the real and imaginary parts of z separately and add the result to r.

```
rscalar_add_sigmoid_op(const rscalar& r, const rscalar& z)
```
Apply the sigmoid operator to the real and imaginary parts of z separately and add the result to r.

BACKWARD OPERATORS

The "backward" operators are for use in automatic differentiation.

```
cscalar_add_div_back0_op(const rscalar& r, const rscalar& g, const rscalar& x,
                         const rscalar& y)
```
Set $r \leftarrow r + g/\overline{y}$.

```
cscalar_add_div_back1_op(const rscalar& r, const rscalar& g, const rscalar& x,
                         const rscalar& y)
```
Set $r \leftarrow r - g\, \overline{x}/\overline{y}^2$.

```
cscalar_add_pow_back_op(const rscalar& r, const rscalar& x, const rscalar& g, const float p,
                        const float c)
```
Set $r \leftarrow r + c\, g\, \overline{x^p}$.

```
rscalar_add_abs_back_op(const rscalar& r, const rscalar& g, const rscalar& x)
```
Set $r \leftarrow r + g$ if $x \geq 0$ and $r \leftarrow r - g$ otherwise.

BLOCKING OPERATIONS

The following functions are called directly (as opposed to being pushed to the engine as an operator).

```
vector< complex<float> > cscalar_get(const cscalar& z)
```
Flush `z` and return its value(s) in a `std::vector`.

ctensor

The `ctensor` virtual type represents complex valued matrices and tensors in single precision arithmetic. A `ctensor` object may have a bundle dimension n_{bu}. User level code can access `ctensor` objects using the following operators.

CONSTRUCTORS

`new_ctensor_op(const Gdims& dims, const int nbu=-1, const int dev =0)`
`new_ctensor_zero_op(const Gdims& dims, const int nbu=-1, const int dev=0)`
`new_ctensor_ones_op(const Gdims& dims, const int nbu=-1, const int dev=0)`
`new_ctensor_identity_op(const Gdims& dims, const int nbu=-1, const int dev=0)`
`new_ctensor_sequential_op(const Gdims& dims, const int nbu=-1, const int dev=0)`
`new_ctensor_gaussian_op(const Gdims& dims, const int nbu=-1, const int dev=0)`

Construct a new `ctensor` object of size `dims` with `nbu` bundles on `dev`. The six cases correspond to the object being (a) uninitialized, (b) initialized to zero, (c) the ones tensor, (d) the identity matrix, (e) initilized with entries $1, 2, \ldots$ in sequence, (g) initialized with random standard normal entries. `nbu=-1` signifies that the object is not bundled and `device=0` is the host.

`new_ctensor_from_gtensor_op(const Gtensor& T, const int nbu=-1, const int dev=0)`

Create a new `ctensor` from the `Gtensor T`.

`ctensor_copy_op(const ctensor& X)`

Create a new `ctensor` by copying `X`.

IN-PLACE OPERATORS

`ctensor_set_zero_op(const ctensor& x)`

Set `x` to zero.

NOT IN-PLACE OPERATORS

`ctensor_conj_op(const ctensor& X)`

Return the conjugate tensor $\overline{X}$.

`ctensor_transp_op(const ctensor& X)`

Return $X^\top$, the transpose of X.

`ctensor_herm_op(const ctensor& X)`

Return $X^\dagger$, the Hermitian conjugate of X.

CUMULATIVE OPERATORS

`ctensor_add_op(const ctensor& R, const ctensor& X)`

Set $r \leftarrow R + X$.

`ctensor_add_conj_op(const ctensor& R, const ctensor& X)`

Set $r \leftarrow R + \overline{X}$.

`ctensor_add_transp_op(const ctensor& R, const ctensor& X)`

Set $r \leftarrow R + X^\top$.

```
ctensor_add_herm_op(const ctensor& R, const ctensor& X)
```
Set $r \leftarrow R + X^\dagger$.

```
ctensor_add_times_real_op(const ctensor& R, const ctensor& X, const float c)
```
Set $R \leftarrow R + cX$.

```
ctensor_add_times_complex_op(const ctensor& R, const ctensor& X, const complex<float> c)
```
Set $R \leftarrow R + cX$.

```
ctensor_add_prod_rA_op(const ctensor& R, const rscalar& c, const ctensor& X)
```
Set $R \leftarrow R + cX$.

```
ctensor_add_prod_cA_op(const ctensor& R, const cscalar& c, const ctensor& X)
```
Set $R \leftarrow R + cX$.

```
ctensor_add_prod_cc_A_op(const ctensor& R, const cscalar& c, const ctensor& X)
```
Set $R \leftarrow R + \bar{c}X$.

```
ctensor_add_prod_c_Ac_op(const ctensor& R, const cscalar& c, const ctensor& X)
```
Set $R \leftarrow R + c\overline{X}$.

```
ctensor_add_Mprod_op(const ctensor& R, const ctensor& A, const ctensor& B)
ctensor_add_Mprod_AT_op(const ctensor& R, const ctensor& A, const ctensor& B)
ctensor_add_Mprod_TA_op(const ctensor& R, const ctensor& A, const ctensor& B)
ctensor_add_Mprod_AC_op(const ctensor& R, const ctensor& A, const ctensor& B)
ctensor_add_Mprod_TC_op(const ctensor& R, const ctensor& A, const ctensor& B)
ctensor_add_Mprod_AH_op(const ctensor& R, const ctensor& A, const ctensor& B)
ctensor_add_Mprod_HA_op(const ctensor& R, const ctensor& A, const ctensor& B)
```
Set $R \leftarrow R{+}AB,\ R \leftarrow R{+}AB^\top,\ R \leftarrow R{+}A,\ R \leftarrow R{+}A\overline{B},\ R \leftarrow R{+}A^\top\overline{B},\ R \leftarrow R{+}AB^\dagger,\ R \leftarrow R{+}A.$

```
ctensor_add_column_norms_op(const ctensor& R, const ctensor& X)
```
Increment $R(i_1, \ldots, i_{k-1})$ with the ℓ_2 norm of the column $X(i_1, \ldots, i_{k-1}, \cdot)$.

```
ctensor_divide_cols_op(const ctensor& R, const ctensor& N)
```
Divide each element of the column $R(i_1, \ldots, i_{k-1}, \cdot)$ by $N(i_1, \ldots, i_{k-1})$.

INTO OPERATORS

```
ctensor_add_inp_op(const cscalar& r, const ctensor& A, const ctensor& B)
```
Set $r \leftarrow r + \langle A, B\rangle$.

BACKWARD OPERATORS

The following "backward" operators are for use in automatic differentiation.

```
ctensor_add_col_norms_back_op(const ctensor& R, const ctensor& G, const ctensor& X,
                                                                   const ctensor& N)
ctensor_add_divide_cols_back0_op(const ctensor& R, const ctensor& G, const ctensor& X,
                                                                   const ctensor& N)
ctensor_add_divide_cols_back1_op(const ctensor& R, const ctensor& G, const ctensor& X,
                                                                   const ctensor& N)
```

BLOCKING OPERATIONS

The following functions are called directly as opposed to being pushed to the engine as an operator.

`Gtensor<complex<float> > ctensor_get(const ctensor& X)`

Flush `X` and return its value as a `Gtensor`.