s

Cengine

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

Risi Kondor
Flatiron Institute NYC, and
The University of Chicago

Contents

Overview

Cengine is a lightweight C++ compute engine designed to dynamically parallelize unstructured numerical computations in a combination of two ways:

- Distributing operations across multiple CPU threads, and/or
- Batching together large number of operations of the same kind for parallel execution on the GPU.

Cengine employs the delayed execution model of computation and internally uses a DAG-based scheduler to ensure that dependency relationships are not violated.

From the user side the engine just expects a sequence of simple instructions such as

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

instructing it to add tensors `a` and `b` and store the result in `c`. However, the instructions are not executed immediately, but rather buffered until one of the internal CPU threads becomes available and/or a sufficient number of operations have accumulated to be efficiently executed in parallel on the GPU. The result of a sequence of operations is returned to the user once it is ready.

Cengine offers a small number of built-in data types such as `rscalar, cscalar` and `ctensor` for real/complex valued scalar and tensor objects, together with a complement of basic linear algebra operators acting on them. However, is designed to function equaly well with user-defined data types and custom operators, and in fact we expect that in most use cases the engine will be extended in this way.

Cengine is written in standard C++11 and has no other dependencies besides CUDA/CUBLAS for GPU functionality.

Usage

The most important class in Cengine is that of the compute engine itself, `Cengine::Cengine`. Normally a single instance of this class is initialized at startup and keeps running until the user's 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 can only issues commands to the engine via its `push` method. The `push` method then returns a handle, which can subsequently be used to reference the object stored internally. 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`.

Cengine follows the asynchronous, delayed execution model of computation. This means that most commands are not executed immediately, but at some later moment 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 might also differ from the order that they were issued to the engine. However, the engine keeps track of dependencies between operations to ensure that the final result of any computation is correct.

In contrast, a small set of commands are *blocking*, which means that 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 only returns after 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 also automatically takes care of memory management. In particular, for any given backend object `T`, as soon as there are no operations pending that take `T` as an argument and there are no user side handles pointing to `T` either, `T` is scheduled for deletion. When the `Cengine::engine` object is shut down, all pending operations are flushed and all backend objects are destroyed.

Operators

Commands in Cengine are actually operators and the template argument of `Cengine::push` command is the name of the corresponding class. The abstract base class of all operator classes is `Cengine::Coperator`. For example, the internal definition of the `ctensor_add_op` operator in 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();
   }

};
```

In each operator class, the `exec` method is responsible for carrying out the actual operation on the operator's arguments. In the case of `ctensor_add_op` this is just involves calling the backend object's method for tensor summation. However, in user-defined operators, the `exec` method is often 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 of type `Censing::CtensorB`, which is the backend container for `ctensor` objects. The built-in `rscalar, scalar` and `ctensor`, extended by user defined operators, are sufficient for many purposes. However, new backend object classes are also easy to add, by subclassing the `Cengine::Cobject` abstract class.

Reference

Built-in types and operators

Rscalar

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_gaussian_op(const int nbu=-1, const int dev=0)`
: Construct a new `rscalar` object with `nbu` bundles on device `dev`. The three cases correspond to the object being uninitialized, initialized to zero, 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 $r \leftarrow r + c\,x^p$.

BLOCKING OPERATORS

Cscalar

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_gaussian_op(const int nbu=-1, const int dev=0)`
: Construct a new `cscalar` object with `nbu` bundles on `dev`. The three cases correspond to the object being uninitialized, initialized to zero, 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& x)`
: Set `x` to zero.

NOT IN-PLACE OPERATORS

`cscalar_conj_op(const cscalar& x)`
: Return $\overline{x}$.

`cscalar_get_real_op(const cscalar& x)`
: Return the real part of x.

`cscalar_get_imag_op(const cscalar& x)`
: Return the imaginary part of x.

CUMULATIVE OPERATORS

`cscalar_add_op(const cscalar& r, const cscalar& x)`
: Set $r \leftarrow r + x$.

`cscalar_subtract_op(const cscalar& r, const cscalar& x)`
: Set $r \leftarrow r - x$.

`cscalar_add_prod_op(const cscalar& r, const cscalar& x, const cscalar& y)`
: Set $r \leftarrow r + xy$.

`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& x)`
: Set $r \leftarrow r + |\, x \,|$.

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

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

Ctensor

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_identity_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 four cases correspond to the object being uninitialized, initialized to zero, the identity matrix, or 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 $\overline{x}$.

`ctensor_get_real_op(const ctensor& x)`
: Return the real part of x.

`ctensor_get_imag_op(const ctensor& x)`
: Return the imaginary part 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& A, const float c)`
: Set $R \leftarrow R + cA$.

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

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

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

`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 $R \leftarrow R+AB,\ R \leftarrow R+AB^\top,\ R \leftarrow R+A,\ R \leftarrow R+A\overline{B},\ R \leftarrow R+A^\top\overline{B},\ R \leftarrow R+AB^\dagger,\ R \leftarrow R+A$.

INTO OPERATORS

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

Wrapper classes