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 that dynamically parallelizes numerical computations in a combination of two ways:

- Automatically distributing operations across multiple CPU threads,
- Batching together operations of the same kind for parallel execution on the GPU.

Cengine employs the delayed execution model of computation. Internally, Cengine maintains a DAG to ensure that dependency relationships are not violated.

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

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

telling it to add tensors `a` and `b` and store the result in `c`. The instructions are not executed immediately. Instead, Cengine just adds the `ctensor_add_op` operator to its internal queue, to be executed when one of the CPU threads becomes available or a sufficient number of operations of the same type have accumulated to be executed in parallel on the GPU.

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 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 then returns a pointer to 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`.

Asynchronous execution

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 is generally not the same as the order that they were issued to the engine. Therefore, the engine needs to keep track of dependencies between operations to ensure that the final result is always correct.

The delayed execution model 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 that includes a member variable `hdl` to store the handle returned by engine. To implement in-place matrix addition, the user may then add the 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 the computation has to be extracted from the `Cengine`. 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 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 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`, the object `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 implemented as operators (in the sense of each command having a corresponding class) and the template argument of `Cengine::push` command is the name of the corresponding 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 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 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` classes, if extended by user defined operators, are already sufficient for many purposes. However, there is nothing stopping the user from adding new backend classes, simply by subclassing the `Cengine::Cobject` abstract class. Cengine can manage any type of user defined backend object, as long as its class 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 the process of executing multiple instances of the same operation together, in parallel. Batching is particularly important for GPU computations, since GPU threads in general are not independent: on NVIDIA architectures, for example, all threads running on the same multiprocessor must essentially be executing the same set of instructions at any given time. Some types of computations, such as solving a systems of partial differential equations on a regular grid are relatively well suited to this paradigm, since each thread can take care of a single grid point.

Many other types of computations however are much less structured. In a graph neural network, for example, the actual 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 very 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 in batch on the GPU, when a certain number of the same type have accumulated. Taking dynamic batching too far can lead to situations where a large number of batched operations are waiting on each other and none of the batches is getting executed. Therefore, in general, it is best to only batch a relatively small set of frequent operations that are expensive enough to be performance critical, but not so expensive that individually (without batching) a single instance of the operation could itself saturate most of the GPU's processing power. As an exmple, 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, say `UserOp1`, the engine must internally create a separate object of class `BatcherA<UserOp1>` to manage the batching process. For the engine to be able to keep track of the correspondence between operators and corresponding batchers, each batched operator class must have a static integer `batcher_id` variable.

Meta-batchers

Many batched operators require separate batchers for each setting 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`, and it just 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 then have a `signature()` method that returns the signature object corresponding to the given pair of matrices to be multiplied. The engine will 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 is to define the signature class and 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) is

`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 has a a small collection of simple built-in types corresponding to real and complex scalars/tensors to seed this process:

```
rscalar
cscalar
ctensor .
```

To match the GPU, 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 and GPU functionality

Similarly to popular libraries 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)` commend, 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, each arithmetic operation is implemented twice: once for the CPU in straight-line C++ and once for the GPU. Whenever possible that latter is done with CUBLAS calls, but in certain cases it requires custom CUDA kernels. Whether a given operation is executed on the CPU or GPU depends on where its arguments are: 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 storage format.

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