reikna Documentation

Release 0.6.3

Bogdan Opanchuk

June 17, 2014

Contents

`Reikna` is a library containing various GPU algorithms built on top of PyCUDA and PyOpenCL. The main design goals are:

- separation of computation cores (matrix multiplication, random numbers generation etc) from simple transformations on their input and output values (scaling, typecast etc);
- separation of the preparation and execution stage, maximizing the performance of the execution stage at the expense of the preparation stage (in other words, aiming at large simulations)
- partial abstraction from CUDA/OpenCL

The installation is as simple as

```
$ pip install reikna
```

CHAPTER 1

Community resources

- Source repository on GitHub;
- Issue tracker, *ibid.*;
- Discussion forum on Google Groups.

CHAPTER 2

Contents

2.1 Introduction

This section contains a brief illustration of what `reikna` does. For more details see *basic* and *advanced* tutorials.

2.1.1 CLUDA

CLUDA is an abstraction layer on top of PyCUDA/PyOpenCL. Its main purpose is to separate the rest of `reikna` from the difference in their APIs, but it can be used by itself too for some simple tasks.

Consider the following example, which is very similar to the one from the index page on PyCUDA documentation:

```
import numpy
import reikna.cluda as cluda

N = 256

api = cluda.ocl_api()
thr = api.Thread.create()

program = thr.compile("""
KERNEL void multiply_them(
    GLOBAL_MEM float *dest,
    GLOBAL_MEM float *a,
    GLOBAL_MEM float *b)
{
  const SIZE_T i = get_local_id(0);
  dest[i] = a[i] * b[i];
}
""")

multiply_them = program.multiply_them

a = numpy.random.randn(N).astype(numpy.float32)
b = numpy.random.randn(N).astype(numpy.float32)
a_dev = thr.to_device(a)
b_dev = thr.to_device(b)
dest_dev = thr.empty_like(a_dev)

multiply_them(dest_dev, a_dev, b_dev, local_size=N, global_size=N)
print((dest_dev.get() - a * b == 0).all())
```

If you are familiar with `PyCUDA` or `PyOpenCL`, you will easily understand all the steps we have made here. The `cluda.ocl_api()` call is the only place where OpenCL is mentioned, and if you replace it with `cluda.cuda_api()` it will be enough to make the code use CUDA. The abstraction is achieved by using generic API module on the Python side, and special macros (`KERNEL`, `GLOBAL_MEM`, and others) on the kernel side.

The argument of `compile()` method can also be a template, which is quite useful for metaprogramming, and also used to compensate for the lack of complex number operations in CUDA and OpenCL. Let us illustrate both scenarios by making the initial example multiply complex arrays. The template engine of choice in `reikna` is Mako, and you are encouraged to read about it as it is quite useful. For the purpose of this example all we need to know is that `${python_expression()}` is a synthax construction which renders the expression result.

```
import numpy
from numpy.linalg import norm

from reikna import cluda
from reikna.cluda import functions, dtypes

N = 256
dtype = numpy.complex64

api = cluda.ocl_api()
thr = api.Thread.create()

program = thr.compile("""
KERNEL void multiply_them(
    GLOBAL_MEM ${ctype} *dest,
    GLOBAL_MEM ${ctype} *a,
    GLOBAL_MEM ${ctype} *b)
{
  const SIZE_T i = get_local_id(0);
  dest[i] = ${mul}(a[i], b[i]);
}
""", render_kwds=dict(
    ctype=dtypes.ctype(dtype),
    mul=functions.mul(dtype, dtype)))

multiply_them = program.multiply_them

r1 = numpy.random.randn(N).astype(numpy.float32)
r2 = numpy.random.randn(N).astype(numpy.float32)
a = r1 + 1j * r2
b = r1 - 1j * r2
a_dev = thr.to_device(a)
b_dev = thr.to_device(b)
dest_dev = thr.empty_like(a_dev)

multiply_them(dest_dev, a_dev, b_dev, local_size=N, global_size=N)
print(norm(dest_dev.get() - a * b) / norm(a * b) <= 1e-6)
```

Note that CLUDA `Thread` is created by means of a static method and not using the constructor. The constructor is reserved for more probable scenario, where we want to include some `reikna` functionality in a larger program, and we want it to use the existing context and stream/queue (see the `Thread` constructor). In this case all further operations with the thread will be performed using the objects provided.

Here we have passed two values to the template: `ctype` (a string with C type name), and `mul` which is a `Module` object containing a single multiplication function. The object is created by a function `mul()` which takes data types being multiplied and returns a module that was parametrized accordingly. Inside the template the variable `mul` is essentially the prefix for all the global C objects (functions, structures, macros etc) from the module. If there is only one public object in the module (which is recommended), it is a common practice to give it the name consisting just

of the prefix, so that it could be called easily from the parent code.

For more information on modules, see *Tutorial: modules and snippets*; the complete list of things available in CLUDA can be found in *CLUDA reference*.

2.1.2 Computations

Now it's time for the main part of the functionality. `reikna` provides GPGPU algorithms in the form of `Computation`-based cores and `Transformation`-based plug-ins. Computations contain the algorithm itself; examples are matrix multiplication, reduction, sorting and so on. Transformations are parallel operations on inputs or outputs of computations, used for scaling, typecast and other auxiliary purposes. Transformations are compiled into the main computation kernel and are therefore quite cheap in terms of performance.

As an example, we will consider the matrix multiplication.

```
import numpy
from numpy.linalg import norm
import reikna.cluda as cluda
from reikna.linalg import MatrixMul

api = cluda.ocl_api()
thr = api.Thread.create()

shape1 = (100, 200)
shape2 = (200, 100)

a = numpy.random.randn(*shape1).astype(numpy.float32)
b = numpy.random.randn(*shape2).astype(numpy.float32)
a_dev = thr.to_device(a)
b_dev = thr.to_device(b)
res_dev = thr.array((shape1[0], shape2[1]), dtype=numpy.float32)

dot = MatrixMul(a_dev, b_dev, out_arr=res_dev)
dotc = dot.compile(thr)
dotc(res_dev, a_dev, b_dev)

res_reference = numpy.dot(a, b)

print(norm(res_dev.get() - res_reference) / norm(res_reference) < 1e-6)
```

Most of the code above should be already familiar, with the exception of the creation of `MatrixMul` object. The computation constructor takes two array-like objects, representing arrays that will participate in the computation. After that the computation object has to be compiled. The `compile()` method requires a `Thread` object, which serves as a source of data about the target API and device, and provides an execution queue.

2.1.3 Transformations

Now imagine that you want to multiply complex matrices, but real and imaginary parts of your data are kept in separate arrays. You could create additional kernels that would join your data into arrays of complex values, but this would require additional storage and additional calls to GPU. Transformation API allows you to connect these transformations to the core computation — matrix multiplication — effectively adding the code into the main computation kernel and changing its signature.

Let us change the previous example and connect transformations to it.

```
import numpy
from numpy.linalg import norm
import reikna.cluda as cluda
from reikna.core import Type
from reikna.linalg import MatrixMul
from reikna.transformations import combine_complex

api = cluda.ocl_api()
thr = api.Thread.create()

shape1 = (100, 200)
shape2 = (200, 100)

a_re = numpy.random.randn(*shape1).astype(numpy.float32)
a_im = numpy.random.randn(*shape1).astype(numpy.float32)
b_re = numpy.random.randn(*shape2).astype(numpy.float32)
b_im = numpy.random.randn(*shape2).astype(numpy.float32)

arrays = [thr.to_device(x) for x in [a_re, a_im, b_re, b_im]]
a_re_dev, a_im_dev, b_re_dev, b_im_dev = arrays

a_type = Type(numpy.complex64, shape=shape1)
b_type = Type(numpy.complex64, shape=shape2)
res_dev = thr.array((shape1[0], shape2[1]), dtype=numpy.complex64)

dot = MatrixMul(a_type, b_type, out_arr=res_dev)
combine_a = combine_complex(a_type)
combine_b = combine_complex(b_type)

dot.parameter.matrix_a.connect(
    combine_a, combine_a.output, a_re=combine_a.real, a_im=combine_a.imag)
dot.parameter.matrix_b.connect(
    combine_b, combine_b.output, b_re=combine_b.real, b_im=combine_b.imag)

dotc = dot.compile(thr)

dotc(res_dev, a_re_dev, a_im_dev, b_re_dev, b_im_dev)

res_reference = numpy.dot(a_re + 1j * a_im, b_re + 1j * b_im)

print(norm(res_dev.get() - res_reference) / norm(res_reference) < 1e-6)
```

We have used a pre-created transformation `combine_complex()` from `reikna.transformations` for simplicity; developing a custom transformation is also possible and described in *Writing a transformation*. From the documentation we know that it transforms two inputs into one output; therefore we need to attach it to one of the inputs of `dot` (identified by its name), and provide names for two new inputs.

Names to attach to are obtained from the documentation for the particular computation; for `MatrixMul` these are `out`, `a` and `b`.

In the current example we have attached the transformations to both inputs. Note that the computation has a new signature now, and the compiled `dot` object now works with split complex numbers.

2.2 Tutorial: modules and snippets

Modules and snippets are important primitives in CLUDA which are used in the rest of `reikna`, although mostly internally. Even if you do not write modules yourself, you will most likely use operations from the `functions` module, or common transformations from the `transformations` module, which are essentially snippet and module factories (callables returning `Snippet` and `Module` objects). Therefore it helps if you know how they work under the hood.

2.2.1 Snippets

Snippets are `Mako` template defs (essentially functions returning rendered text) with the associated dictionary of render keywords. Some computations which are parametrized by custom code (for example, `PureParallel`) require this code to be provided in form of a snippet with a certain call signature. When a snippet is used in a template, the result is quite straightworward: its template function is called, rendering and returning its contents, just as a normal `Mako` def.

Let us demonstrate it with a simple example. Consider the following snippet:

```
add = Snippet("""
<%def name="add(varname)">
${varname} + ${num}
</%def>
""",
render_kwds=dict(num=1))
```

Now we can compile a template which uses this snippet:

```
program = thr.compile("""
KERNEL void test(int *arr)
{
    const SIZE_T idx = get_global_id(0);
    int a = arr[idx];
    arr[idx] = ${add('x')};
}
""",
render_kwds=dict(add=add))
```

As a result, the code that gets compiled is

```
KERNEL void test(int *arr)
{
    const SIZE_T idx = get_global_id(0);
    int a = arr[idx];
    arr[idx] = x + 1;
}
```

If the snippet is used without parentheses (e.g. `${add}`), it is equivalent to calling it without arguments (`${add()}`).

The root code that gets passed to `compile()` can be viewed as a snippet with an empty signature.

2.2.2 Modules

Modules are quite similar to snippets in a sense that they are also `Mako` defs with an associated dictionary of render keywords. The difference lies in the way they are processed. Consider a module containing a single function:

```
add = Module("""
<%def name="add(prefix, arg)">
WITHIN_KERNEL int ${prefix}(int x)
{
    return x + ${num} + ${arg};
}
</%def>
""",
render_kwds=dict(num=1))
```

Modules contain complete C entities (function, macros, structures) and get rendered in the root level of the source file. In order to avoid name clashes, their def gets a string as a first argument, which it has to use to prefix these entities' names. If the module contains only one entity that is supposed to be used by the parent code, it is a good idea to set its name to `prefix` only, to simplify its usage.

Let us now create a kernel that uses this module:

```
program = thr.compile("""
KERNEL void test(int *arr)
{
    const SIZE_T idx = get_global_id(0);
    int a = arr[idx];
    arr[idx] = ${add(2)}(x);
}
""",
render_kwds=dict(add=add))
```

Before the compilation render keywords are inspected, and if a module object is encountered, the following things happen:

1. This object's `render_kwds` are inspected recursively and any modules there are rendered in the same way as described here, producing a source file.
2. The module itself gets assigned a new prefix and its template function is rendered with this prefix as the first argument, with the positional arguments given following it. The result is attached to the source file.
3. The corresponding value in the current `render_kwds` is replaced by the newly assigned prefix.

With the code above, the rendered module will produce the code

```
WITHIN_KERNEL int _module0_(int x)
{
    return x + 1 + 2;
}
```

and the `add` keyword in the `render_kwds` gets its value changed to `_module0_`. Then the main code is rendered and appended to the previously renderd parts, giving

```
WITHIN_KERNEL int _module0_(int x)
{
    return x + 1;
}

KERNEL void test(int *arr)
{
    const SIZE_T idx = get_global_id(0);
    int a = arr[idx];
    arr[idx] = _module0_(x);
}
```

which is then passed to the compiler. If your module's template def does not take any arguments except for `prefix`, you can call it in the parent template just as `${add}` (without empty parentheses).

Warning: Note that `add` in this case is not a string, it is an object that has `__str__()` defined. If you want to concatenate a module prefix with some other string, you have to either call `str()` explicitly (`str(add) + "abc"`), or concatenate it inside a template (`${add} abc`).

Modules can reference snippets in their `render_kwds`, which, in turn, can reference other modules. This produces a tree-like structure with the snippet made from the code passed by user at the root. When it is rendered, it is traversed depth-first, modules are extracted from it and arranged in a flat list in the order of appearance. Their positions in `render_kwds` are replaced by assigned prefixes. This flat list is then rendered, producing a single source file being fed to the compiler.

Note that if the same module object was used without arguments in several other modules or in the kernel itself, it will only be rendered once. Therefore one can create a "root" module with the data structure declaration and then use that structure in other modules without producing type errors on compilation.

2.2.3 Shortcuts

The amount of boilerplate code can be somewhat reduced by using `Snippet.create` and `Module.create` constructors. For the snippet above it would look like:

```
add = Snippet.create(
    lambda varname: "${varname} + ${num}",
    render_kwds=dict(num=1))
```

Note that the lambda here serves only to provide the information about the `Mako` def's signature. Therefore it should return the template code regardless of the actual arguments passed.

If the argument list is created dynamically, you can use `template_def()` with a normal constructor:

```
argnames = ['varname']
add = Snippet(
    template_def(argnames, "${varname} + ${num}"),
    render_kwds=dict(num=1))
```

Modules have a similar shortcut constructor. The only difference is that by default the resulting template def has one positional argument called `prefix`. If you provide your own signature, its first positional argument will receive the prefix value.

```
add = Module.create("""
WITHIN_KERNEL int ${prefix}(int x)
{
    return x + ${num};
}
""",
render_kwds=dict(num=1))
```

Of course, both `Snippet` and `Module` constructors can take already created `Mako` defs, which is convenient if you keep templates in a separate file.

2.2.4 Module and snippet discovery

Sometimes you may want to pass a module or a snippet inside a template as an attribute of a custom object. In order for CLUDA to be able to discover and process it without modifying your original object, you need to make your object comply to a discovery protocol. The protocol method takes a processing function and is expected to return a

new object of the same class with the processing function applied to all the attributes that may contain a module or a snippet. By default, objects of type `tuple`, `list`, and `dict` are discoverable.

For example:

```
class MyClass:

    def __init__(self, coeff, mul_module, div_module):
        self.coeff = coeff
        self.mul = mul_module
        self.div = div_module

    def __process_modules__(self, process):
        return MyClass(self.coeff, process(self.mul), process(self.div))
```

2.2.5 Nontrivial example

Modules were introduced to help split big kernels into small reusable pieces which in `CUDA` or `OpenCL` program would be put into different source or header files. For example, a random number generator may be assembled from a function generating random integers, a function transforming these integers into random numbers with a certain distribution, and a `PureParallel` computation calling these functions and saving results to global memory. These two functions can be extracted into separate modules, so that a user could call them from some custom kernel if he does not need to store the intermediate results.

Going further with this example, one notices that functions that produce randoms with sophisticated distributions are often based on simpler distributions. For instance, the commonly used Marsaglia algorithm for generating Gamma-distributed random numbers requires several uniformly and normally distributed randoms. Normally distributed randoms, in turn, require several uniformly distributed randoms — with the range which differs from the one for uniformly distributed randoms used by the initial Gamma distribution. Instead of copy-pasting the function or setting its parameters dynamically (which in more complicated cases may affect the performance), one just specifies the dependencies between modules and lets the underlying system handle things.

The final render tree may look like:

```
Snippet(
    PureParallel,
    render_kwds = {
      base_rng -> Snippet(...)
      gamma -> Snippet(
    }             Gamma,
                  render_kwds = {
                    uniform -> Snippet(...)
                    normal -> Snippet(
                  }             Normal,
                )               render_kwds = {
                                  uniform -> Snippet(...)
                                  }
                              )
```

2.3 Tutorial: basics

2.3.1 Usage of computations

All `reikna` computation classes are derived from the `Computation` class and therefore share the same API and behavior. A computation object is an opaque typed function-like object containing all the information necessary to

generate GPU kernels that implement some algorithm, along with necessary internal temporary and persistent memory buffers. Before use it needs to be compiled by calling `compile()` for a given `Thread` (thus using its associated device and queue). This method returns a `ComputationCallable` object which takes GPU arrays and scalar parameters and calls its internal kernels.

2.3.2 Computations and transformations

One often needs to perform some simple processing of the input or output values of a computation. This can be scaling, splitting complex values into components, padding, and so on. Some of these operations require additional memory to store intermediate results, and all of them involve additional overhead of calling the kernel, and passing values to and from the device memory. `Reikna` porvides an API to define such transformations and attach them to "core" computations, effectively compiling the transformation code into the main kernel(s), thus avoiding all these drawbacks.

2.3.3 Transformation tree

Before talking about transformations themselves, we need to take a closer look at the computation signatures. Every `Computation` object has a `signature` attribute containing `funcsigs.Signature` object. It is the same signature object as can be exctracted from any Python function using `funcsigs.signature` function (or `inspect.signature` from the standard library for Python >= 3.3). When the computation object is compiled, the resulting callable will have this exact signature.

The base signature for any computation can be found in its documentation (and, sometimes, can depend on the arguments passed to its constructor — see, for example, `PureParallel`). The signature can change if a user connects transformations to some parameter via `connect()`; in this case the `signature` attribute will change accordingly.

All attached transformations form a tree with roots being the base parameters computation has right after creation, and leaves forming the user-visible signature, which the compiled `ComputationCallable` will have.

As an example, let us consider a pure parallel computation object with one output, two inputs and a scalar parameter, which performs the calculation `out = in1 + in2 + param`:

```
from __future__ import print_function
import numpy

from reikna import cluda
from reikna.cluda import Snippet
from reikna.core import Transformation, Type, Annotation, Parameter
from reikna.algorithms import PureParallel
import reikna.transformations as transformations

arr_t = Type(numpy.float32, shape=128)
carr_t = Type(numpy.complex64, shape=128)

comp = PureParallel(
    [Parameter('out', Annotation(carr_t, 'o')),
    Parameter('in1', Annotation(carr_t, 'i')),
    Parameter('in2', Annotation(carr_t, 'i')),
    Parameter('param', Annotation(numpy.float32))],
    """
    VSIZE_T idx = ${idxs[0]};
    ${out.store_idx}(
        idx, ${in1.load_idx}(idx) + ${in2.load_idx}(idx) + ${param});
    """)
```

The details of creating the computation itself are not important for this example; they are provided here just for the sake of completeness. The initial transformation tree of `comp` object looks like:

```
   | out   | >>
>> | in1   |
>> | in2   |
>> | param |
```

Here the insides of `||` are the base computation (the one defined by the developer), and `>>` denote inputs and outputs provided by the user. The computation signature is:

```
>>> for param in comp.signature.parameters.values():
...     print(param.name + ":" + repr(param.annotation))
out:Annotation(Type(complex64, shape=(128,), strides=(8,)), role='o')
in1:Annotation(Type(complex64, shape=(128,), strides=(8,)), role='i')
in2:Annotation(Type(complex64, shape=(128,), strides=(8,)), role='i')
param:Annotation(float32)
```

Now let us attach the transformation to the output which will split it into two halves: `out1 = out / 2`, `out2 = out / 2`:

```
tr = transformations.split_complex(comp.parameter.out)
comp.parameter.out.connect(tr, tr.input, out1=tr.real, out2=tr.imag)
```

We have used the pre-created transformation here for simplicity; writing custom transformations is described in *Writing a transformation*.

In addition, we want `in2` to be scaled before being passed to the main computation. To achieve this, we connect the scaling transformation to it:

```
tr = transformations.mul_param(comp.parameter.in2, numpy.float32)
comp.parameter.in2.connect(tr, tr.output, in2_prime=tr.input, param2=tr.param)
```

The transformation tree now looks like:

```
                     | out   | ----> out1 >>
                     |       |   \-> out2 >>
                  >> | in1   |
>> in2_prime ------> | in2   |
>> param2 ----/      |       |
                     | param |
```

As can be seen, nothing has changed from the base computation's point of view: it still gets the same inputs and outputs to the same array. But user-supplied parameters (`>>`) have changed, which can be also seen in the value of the `signature`:

```
>>> for param in comp.signature.parameters.values():
...     print(param.name + ":" + repr(param.annotation))
out1:Annotation(Type(float32, shape=(128,), strides=(4,)), role='o')
out2:Annotation(Type(float32, shape=(128,), strides=(4,)), role='o')
in1:Annotation(Type(complex64, shape=(128,), strides=(8,)), role='i')
in2_prime:Annotation(Type(complex64, shape=(128,), strides=(8,)), role='i')
param2:Annotation(float32)
param:Annotation(float32)
```

Notice that the order of the final signature is obtained by traversing the transformation tree depth-first, starting from the base parameters. For more details see the note in the documentation for `connect()`.

The resulting computation returns the value `in1 + (in2_prime * param2) + param` split in half. In order to run it, we have to compile it first. When `prepare_for` is called, the data types and shapes of the given arguments

will be propagated to the roots and used to prepare the original computation.

```
api = cluda.ocl_api()
thr = api.Thread.create()

in1_t = comp.parameter.in1
in2p_t = comp.parameter.in2_prime

out1 = thr.empty_like(comp.parameter.out1)
out2 = thr.empty_like(comp.parameter.out2)
in1 = thr.to_device(numpy.ones(in1_t.shape, in1_t.dtype))
in2_prime = thr.to_device(numpy.ones(in2p_t.shape, in2p_t.dtype))

c_comp = comp.compile(thr)
c_comp(out1, out2, in1, in2_prime, 4, 3)
```

2.3.4 Transformation restrictions

There are some limitations of the transformation mechanics:

1. Transformations are purely parallel, that is they cannot use local memory. In fact, they are very much like `PureParallel` computations, except that the indices they use are defined by the main computation, and not set by the GPU driver.
2. External endpoints of the output transformations cannot point to existing nodes in the transformation tree. This is the direct consequence of the first limitation — it would unavoidably create races between memory writes from different branches. On the other hand, input transformations can be safely connected to existing nodes, including base nodes (although note that inputs are not cached; so even if you load twice from the same index of the same input node, the global memory will be queried twice).

2.4 Tutorial: advanced topics

This tutorial goes into more detail about the internals of computations and transformations, describing how to write them.

2.4.1 Mako basics

`Reikna` uses Mako extensively as a templating engine for transformations and computations. For the purpose of this tutorial you only need to know several things about the synthax:

- Most of Mako synthax is plain Python, with the set of global variables specified externally by the code doing the template rendering
- `${expr}` evaluates Python expression `expr`, calls `str()` on the result and puts it into the text
- a pair of `<%` and `%>` executes Python code inside, which may introduce some local variables
- a pair of `<%def name="func(a, b)">` and `</%def>` defines a template function, which actually becomes a Python function which can be called as `func(a, b)` from the other part of the template and returns a rendered string

2.4.2 Writing a transformation

Some common transformations are already available from `transformations` module. But you can create a custom one if you need to. Transformations are based on the class `Transformation`, and are very similar to `PureParallel` instances, with some additional limitations.

Let us consider a (not very useful, but quite involved) example:

```
tr = Transformation(
    [
        Parameter('out1', Annotation(Type(numpy.float32, shape=100), 'o')),
        Parameter('out2', Annotation(Type(numpy.float32, shape=80), 'o')),
        Parameter('in1', Annotation(Type(numpy.float32, shape=100), 'i')),
        Parameter('in2', Annotation(Type(numpy.float32, shape=100), 'i')),
        Parameter('param', Annotation(Type(numpy.float32))),
    ],
    """
    VSIZE_T idx = ${idxs[0]};
    float i1 = ${in1.load_same};
    float i2 = ${in2.load_idx}(100 - idx) * ${param};
    ${out1.store_same}(i1);
    if (idx < 80)
        ${out2.store_same}(i2);
    """,
    connectors=['in1', 'out1'])
```

Connectors. A transformation gets activated when the main computation attempts to load some value from some index in global memory, or store one to some index. This index is passed to the transformation attached to the corresponding parameter, and used to invoke loads/stores either without changes (to perform strictly elementwise operations), or, possibly, with some changes (as the example illustrates).

If some parameter is only queried once, and only using `load_same` or `store_same`, it is called a *connector*, which means that it can be used to attach the transformation to a computation. Currently connectors cannot be detected automatically, so it is the responsibility of the user to provide a list of them to the constructor. By default all parameters are considered to be connectors.

Shape changing. Parameters in transformations are typed, and it is possible to change data type or shape of a parameter the transformation is attached to. In our example `out2` has length 80, so the current index is checked before the output to make sure there is no out of bounds access.

Parameter objects. The transformation example above has some hardcoded stuff, for example the type of parameters (`float`), or their shapes (`100` and `80`). These can be accessed from argument objects `out1`, `in1` etc; they all have the type `KernelParameter`. In addition, the transformation code gets an `Indices` object with the name `idxs`, which allows one to manipulate index names directly.

2.4.3 Writing a computation

A computation must derive `Computation`. As an example, let us create a computation which calculates `output = input1 + input2 * param`.

Defining a class:

```
import numpy

from reikna.helpers import *
from reikna.core import *

class TestComputation(Computation):
```

Each computation class has to define the constructor, and the plan building callback.

Constructor. `Computation` constructor takes a list of computation parameters, which the deriving class constructor has to create according to arguments passed to it. You will often need `Type` objects, which can be extracted from arrays, scalars or other `Type` objects with the help of `from_value()` (or they can be passed straight to `Annotation`) which does the same thing.

```
def __init__(self, arr, coeff):
    assert len(arr.shape) == 1
    Computation.__init__(self, [
        Parameter('output', Annotation(arr, 'o')),
        Parameter('input1', Annotation(arr, 'i')),
        Parameter('input2', Annotation(arr, 'i')),
        Parameter('param', Annotation(coeff))])
```

In addition to that, the constructor can create some internal state which will be used by the plan builder.

Plan builder. The second method is called when the computation is being compiled, and has to fill and return the computation plan — a sequence of kernel calls, plus maybe some temporary or persistent internal allocations its kernels use. In addition, the plan can include calls to nested computations.

The method takes two predefined positional parameters, plus `KernelArgument` objects corresponding to computation parameters. The `plan_factory` is a callable that creates a new `ComputationPlan` object (in some cases you may want to recreate the plan, for example, if the workgroup size you were using turned out to be too big), and `device_params` is a `DeviceParameters` object, which is used to optimize the computation for the specific device. The method must return a filled `ComputationPlan` object.

For our example we only need one action, which is the execution of an elementwise kernel:

```
def _build_plan(self, plan_factory, device_params, output, input1, input2, param):
    plan = plan_factory()

    template = template_from(
        """
        <%def name='testcomp(kernel_declaration, k_output, k_input1, k_input2, k_param)'>
        ${kernel_declaration}
        {
            VIRTUAL_SKIP_THREADS;
            const VSIZE_T idx = virtual_global_id(0);
            ${k_output.ctype} result =
                ${k_input1.load_idx}(idx) +
                ${mul}(${k_input2.load_idx}(idx), ${k_param});
            ${k_output.store_idx}(idx, result);
        }
        </%def>
        """)

    plan.kernel_call(
        template.get_def('testcomp'),
        [output, input1, input2, param],
        global_size=output.shape,
        render_kwds=dict(mul=functions.mul(input2.dtype, param.dtype)))

    return plan
```

Every kernel call is based on the separate `Mako` template def. The template can be specified as a string using `template_def()`, or loaded as a separate file. Usual pattern in this case is to call the template file same as the file where the computation class is defined (for example, `testcomp.mako` for `testcomp.py`), and store it in some variable on module load using `template_for()` as `TEMPLATE = template_for(__file__)`.

The template function should take the same number of positional arguments as the kernel plus one; you can view `<%def ... >` part as an actual kernel definition, but with the arguments being `KernelParameter` objects containing parameter metadata. The first argument will contain the string with the kernel declaration.

Also, depending on whether the corresponding argument is an output array, an input array or a scalar parameter, the object can be used as `${obj.store_idx}(index, val)`, `${obj.load_idx}(index)` or `${obj}`. This will produce the corresponding request to the global memory or kernel arguments.

If you need additional device functions, they have to be specified between `<%def ... >` and `${kernel_declaration}`. Obviously, these functions can still use `dtype` and `ctype` object properties, although `store_idx` and `load_idx` will most likely result in compilation error (since they are rendered as macros using main kernel arguments).

Since kernel call parameters (`global_size` and `local_size`) are specified on creation, all kernel calls are rendered as CLUDA static kernels (see `compile_static()`) and therefore can use all the corresponding macros and functions (like `virtual_global_flat_id()` in our kernel). Also, they must have `VIRTUAL_SKIP_THREADS` at the beginning of the kernel which remainder threads (which can be present, for example, if the workgroup size is not a multiple of the global size).

2.5 API reference

2.5.1 Version queries

2.5.2 Helpers

2.5.3 CLUDA layer

CLUDA is the foundation of `reikna`. It provides the unified access to basic features of `CUDA` and `OpenCL`, such as memory operations, compilation and so on. It can also be used by itself, if you want to write GPU API-independent programs and happen to only need a small subset of GPU API. The terminology is borrowed from `OpenCL`, since it is a more general API.

API module

Modules for all APIs have the same generalized interface. It is referred here (and references from other parts of this documentation) as `reikna.cluda.api`.

Temporary Arrays

Each `Thread` contains a special allocator for arrays with data that does not have to be persistent all the time. In many cases you only want some array to keep its contents between several kernel calls. This can be achieved by manually allocating and deallocating such arrays every time, but it slows the program down, and you have to synchronize the queue because allocation commands are not serialized. Therefore it is advantageous to use `temp_array()` method to get such arrays. It takes a list of dependencies as an optional parameter which gives the allocator a hint about which arrays should not use the same physical allocation.

Function modules

Kernel toolbox

The stuff available for the kernel passed for compilation consists of two parts.

First, there are several objects available at the template rendering stage, namely `numpy`, `reikna.cluda.dtypes` (as `dtypes`), and `reikna.helpers` (as `helpers`).

Second, there is a set of macros attached to any kernel depending on the API it is being compiled for:

CUDA
If defined, specifies that the kernel is being compiled for CUDA API.

COMPILE_FAST_MATH
If defined, specifies that the compilation for this kernel was requested with `fast_math == True`.

LOCAL_BARRIER
Synchronizes threads inside a block.

WITHIN_KERNEL
Modifier for a device-only function declaration.

KERNEL
Modifier for the kernel function declaration.

GLOBAL_MEM
Modifier for the global memory pointer argument.

LOCAL_MEM
Modifier for the statically allocated local memory variable.

LOCAL_MEM_DYNAMIC
Modifier for the dynamically allocated local memory variable.

LOCAL_MEM_ARG
Modifier for the local memory argument in the device-only functions.

INLINE
Modifier for inline functions.

SIZE_T
The type of local/global IDs and sizes. Equal to `unsigned int` for CUDA, and `size_t` for OpenCL (which can be 32- or 64-bit unsigned integer, depending on the device).

SIZE_T **get_local_id**(int *dim*)

SIZE_T **get_group_id**(int *dim*)

SIZE_T **get_global_id**(int *dim*)

SIZE_T **get_local_size**(int *dim*)

SIZE_T **get_num_groups**(int *dim*)

SIZE_T **get_global_size**(int *dim*)
Local, group and global identifiers and sizes. In case of CUDA mimic the behavior of corresponding OpenCL functions.

VSIZE_T
The type of local/global IDs in the virtual grid. It is separate from `SIZE_T` because the former is intended to be equivalent to what the backend is using, while `VSIZE_T` is a separate type and can be made larger than `SIZE_T` in the future if necessary.

ALIGN(int)
Used to specify an explicit alignment (in bytes) for fields in structures, as

```
typedef struct {
    char ALIGN(4) a;
```

```
    int b;
} MY_STRUCT;
```

VIRTUAL_SKIP_THREADS
: This macro should start any kernel compiled with `compile_static()`. It skips all the empty threads resulting from fitting call parameters into backend limitations.

VSIZE_T **virtual_local_id**(int *dim*)

VSIZE_T **virtual_group_id**(int *dim*)

VSIZE_T **virtual_global_id**(int *dim*)

VSIZE_T **virtual_local_size**(int *dim*)

VSIZE_T **virtual_num_groups**(int *dim*)

VSIZE_T **virtual_global_size**(int *dim*)

VSIZE_T **virtual_global_flat_id**()

VSIZE_T **virtual_global_flat_size**()
: Only available in `StaticKernel` objects obtained from `compile_static()`. Since its dimensions can differ from actual call dimensions, these functions have to be used.

Datatype tools

This module contains various convenience functions which operate with `numpy.dtype` objects.

2.5.4 Core functionality

Classes necessary to create computations and transformations are exposed from the `core` module.

Computation signatures

Core classes

Result and attribute classes

2.5.5 Computations

Algorithms

Linear algebra

Fast Fourier transform

Discrete harmonic transform

Counter-based random number generators

2.5.6 Transformations

2.6 Release history

2.6.1 0.6.3 (18 Jun 2014)

- FIXED: (@schreon) a bug preventing the usage of `EntrywiseNorm` with custom `axes`.
- FIXED: (PR by @SyamGadde) removed syntax constructions incompatible with Python 2.6.
- FIXED: added Python 3.4 to the list of classifiers.

2.6.2 0.6.2 (20 Feb 2013)

- ADDED: `pow()` function module in CLUDA.
- ADDED: a function `any_api()` that returns some supported GPGPU API module.
- ADDED: an example of `Reduce` with a custom data type.
- FIXED: a Py3 compatibility issue in `Reduce` introduced in `0.6.1`.
- FIXED: a bug due to the interaction between the implementation of `from_trf()` and the logic of processing nested computations.
- FIXED: a bug in `FFT` leading to undefined behavior on some OpenCL platforms.

2.6.3 0.6.1 (4 Feb 2013)

- FIXED: `Reduce` can now pick a decreased work group size if the attached transformations are too demanding.

2.6.4 0.6.0 (27 Dec 2013)

- CHANGED: some computations were moved to sub-packages: `PureParallel`, `Transpose` and `Reduce` to `reikna.algorithms`, `MatrixMul` and `EntrywiseNorm` to `reikna.linalg`.
- CHANGED: `scale_const` and `scale_param` were renamed to `mul_const()` and `mul_param()`, and the scalar parameter name of the latter was renamed from `coeff` to `param`.
- ADDED: two transformations for norm of an arbitrary order: `norm_const()` and `norm_param()`.
- ADDED: stub transformation `ignore()`.
- ADDED: broadcasting transformations `broadcast_const()` and `broadcast_param()`.
- ADDED: addition transformations `add_const()` and `add_param()`.
- ADDED: `EntrywiseNorm` computation.
- ADDED: support for multi-dimensional sub-arrays in `c_constant()` and `flatten_dtype()`.
- ADDED: helper functions `extract_field()` and `c_path()` to work in conjunction with `flatten_dtype()`.
- ADDED: a function module `add()`.
- FIXED: casting a coefficient in the `normal_bm()` template to a correct dtype.
- FIXED: `cast()` avoids casting if the value already has the target dtype (since `numpy.cast` does not work with struct dtypes, see issue #4148).
- FIXED: a error in transformation module rendering for scalar parameters with struct dtypes.
- FIXED: normalizing dtypes in several functions from `dtypes` to avoid errors with `numpy` dtype shortcuts.

2.6.5 0.5.2 (17 Dec 2013)

- ADDED: `normal_bm()` now supports complex dtypes.
- FIXED: a nested `PureParallel` can now take several identical argument objects as arguments.
- FIXED: a nested computation can now take a single input/output argument (e.g. a temporary array) as separate input and output arguments.
- FIXED: a critical bug in `CBRNG` that could lead to the counter array not being updated.
- FIXED: convenience constructors of `CBRNG` can now properly handle `None` as `samplers_kwds`.

2.6.6 0.5.1 (30 Nov 2013)

- FIXED: a possible infinite loop in `compile_static()` local size finding algorithm.

2.6.7 0.5.0 (25 Nov 2013)

- CHANGED: `KernelParameter` is not derived from `Type` anymore (although it still retains the corresponding attributes).
- CHANGED: `Predicate` now takes a dtype'd value as `empty`, not a string.

- CHANGED: The logic of processing struct dtypes was reworked, and `adjust_alignment` was removed. Instead, one should use `align()` (which does not take a `Thread` parameter) to get a dtype with the offsets and itemsize equal to those a compiler would set. On the other hand, `ctype_module()` attempts to set the alignments such that the field offsets are the same as in the given numpy dtype (unless `ignore_alignments` flag is set).
- ADDED: struct dtypes support in `c_constant()`.
- ADDED: `flatten_dtype()` helper function.
- ADDED: added `transposed_a` and `transposed_b` keyword parameters to `MatrixMul`.
- ADDED: algorithm cascading to `Reduce`, leading to 3-4 times increase in performance.
- ADDED: `polar_unit()` function module in CLUDA.
- ADDED: support for arrays with 0-dimensional shape as computation and transformation arguments.
- FIXED: a bug in `Reduce`, which lead to incorrect results in cases when the reduction power is exactly equal to the maximum one.
- FIXED: `Transpose` now works correctly for struct dtypes.
- FIXED: `bounding_power_of_2` now correctly returns `1` instead of `2` being given `1` as an argument.
- FIXED: `compile_static()` local size finding algorithm is much less prone to failure now.

2.6.8 0.4.0 (10 Nov 2013)

- CHANGED: `supports_dtype()` method moved from `Thread` to `DeviceParameters`.
- CHANGED: `fast_math` keyword parameter moved from `Thread` constructor to `compile()` and `compile_static()`. It is also `False` by default, instead of `True`. Correspondingly, `THREAD_FAST_MATH` macro was renamed to `COMPILE_FAST_MATH`.
- CHANGED: CBRNG modules are using the dtype-to-ctype support. Correspondingly, the C types for keys and counters can be obtained by calling `ctype_module()` on `key_dtype` and `counter_dtype` attributes. The module wrappers still define their types, but their names are using a different naming convention now.
- ADDED: module generator for nested dtypes (`ctype_module()`) and a function to get natural field offsets for a given API/device (`adjust_alignment`).
- ADDED: `fast_math` keyword parameter in `compile()`. In other words, now `fast_math` can be set per computation.
- ADDED: `ALIGN` macro is available in CLUDA kernels.
- ADDED: support for struct types as `Computation` arguments (for them, the `ctypes` attributes contain the corresponding module obtained with `ctype_module()`).
- ADDED: support for non-sequential axes in `Reduce`.
- FIXED: bug in the interactive `Thread` creation (reported by James Bergstra).
- FIXED: Py3-incompatibility in the interactive `Thread` creation.
- FIXED: some code paths in virtual size finding algorithm could result in a type error.
- FIXED: improved the speed of test collection by reusing `Thread` objects.

2.6.9 0.3.6 (9 Aug 2013)

- ADDED: the first argument to the `Transformation` or `PureParallel` snippet is now a `reikna.core.Indices` object instead of a list.
- ADDED: classmethod `PureParallel.from_trf()`, which allows one to create a pure parallel computation out of a transformation.
- FIXED: improved `Computation.compile()` performance for complicated computations by precreating transformation templates.

2.6.10 0.3.5 (6 Aug 2013)

- FIXED: bug with virtual size algorithms returning floating point global and local sizes in Py2.

2.6.11 0.3.4 (3 Aug 2013)

- CHANGED: virtual sizes algorithms were rewritten and are now more maintainable. In addition, virtual sizes can now handle any number of dimensions of local and global size, providing the device can support the corresponding total number of work items and groups.
- CHANGED: id- and size- getting kernel functions now have return types corresponding to their equivalents. Virtual size functions have their own independent return type.
- CHANGED: `Thread.compile_static()` and `ComputationPlan.kernel_call()` take global and local sizes in the row-major order, to correspond to the matrix indexing in load/store macros.
- FIXED: requirements for PyCUDA extras (a currently non-existent version was specified).
- FIXED: an error in gamma distribution sampler, which lead to slightly wrong shape of the resulting distribution.

2.6.12 0.3.3 (29 Jul 2013)

- FIXED: package metadata.

2.6.13 0.3.2 (29 Jul 2013)

- ADDED: same module object, when being called without arguments from other modules/snippets, is rendered only once and returns the same prefix each time. This allows one to create structure declarations that can be used by functions in several modules.
- ADDED: reworked `cbrng` module and exposed kernel interface of bijections and samplers.
- CHANGED: slightly changed the algorithm that determines the order of computation parameters after a transformation is connected to it. Now the ordering inside a list of initial computation parameters or a list of a single transformation parameters is preserved.
- CHANGED: kernel declaration string is now passed explicitly to a kernel template as the first parameter.
- FIXED: typo in FFT performance test.
- FIXED: bug in FFT that could result in changing the contents of the input array to one of the intermediate results.
- FIXED: missing data type normalization in `c_constant()`.
- FIXED: Py3 incompatibility in `cluda.cuda`.

- FIXED: updated some obsolete computation docstrings.

2.6.14 0.3.1 (25 Jul 2013)

- FIXED: too strict array type check for nested computations that caused some tests to fail.
- FIXED: default values of scalar parameters are now processed correctly.
- FIXED: Mako threw name-not-found exceptions on some list comprehensions in FFT template.
- FIXED: some earlier-introduced errors in tests.
- INTERNAL: `pylint` was ran and many stylistic errors fixed.

2.6.15 0.3.0 (23 Jul 2013)

Major core API change:

- Computations have function-like signatures with the standard `Signature` interface; no more separation of inputs/outputs/scalars.
- Generic transformations were ditched; all the transformations have static types now.
- Transformations can now change array shapes, and load/store from/to external arrays in output/input transformations.
- No flat array access in kernels; all access goes through indices. This opens the road for correct and automatic stride support (not fully implemented yet).
- Computations and accompanying classes are stateless, and their creation is more straightforward.

Other stuff:

- Bumped Python requirements to >=2.6 or >=3.2, and added a dependency on `funcsig`.
- ADDED: more tests for cluda.functions.
- ADDED: module/snippet attributes discovery protocol for custom objects.
- ADDED: strides support to array allocation functions in CLUDA.
- ADDED: modules can now take positional arguments on instantiation, same as snippets.
- CHANGED: `Elementwise` becomes `PureParallel` (as it is not always elementwise).
- FIXED: incorrect behavior of functions.norm() for non-complex arguments.
- FIXED: undefined variable in functions.exp() template (reported by Thibault North).
- FIXED: inconsistent block/grid shapes in static kernels

2.6.16 0.2.4 (11 May 2013)

- ADDED: ability to introduce new scalar arguments for nested computations (the API is quite ugly at the moment).
- FIXED: handling prefixes properly when connecting transformations to nested computations.
- FIXED: bug in dependency inference algorithm which caused it to ignore allocations in nested computations.

2.6.17 0.2.3 (25 Apr 2013)

- ADDED: explicit `release()` (primarily for certain rare CUDA use cases).
- CHANGED: CLUDA API discovery interface (see the documentation).
- CHANGED: The part of CLUDA API that is supposed to be used by other layers was moved to the `__init__.py`.
- CHANGED: CLUDA `Context` was renamed to `Thread`, to avoid confusion with `PyCUDA`/`PyOpenCL` contexts.
- CHANGED: signature of `create()`; it can filter devices now, and supports interactive mode.
- CHANGED: `Module` with `snippet=True` is now `Snippet`
- FIXED: added `transformation.mako` and `cbrng_ref.py` to the distribution package.
- FIXED: incorrect parameter generation in `test/cluda/cluda_vsizes/ids`.
- FIXED: skipping testcases with incompatible parameters in `test/cluda/cluda_vsizes/ids` and `sizes`.
- FIXED: setting the correct length of `max_num_groups` in case of CUDA and a device with CC < 2.
- FIXED: typo in `cluda.api_discovery`.

2.6.18 0.2.2 (20 Apr 2013)

- ADDED: ability to use custom argument names in transformations.
- ADDED: multi-argument `mul()`.
- ADDED: counter-based random number generator `CBRNG`.
- ADDED: `reikna.elementwise.Elementwise` now supports argument dependencies.
- ADDED: Module support in CLUDA; see *Tutorial: modules and snippets* for details.
- ADDED: `template_def()`.
- CHANGED: `reikna.cluda.kernel.render_template_source` is the main renderer now.
- CHANGED: `FuncCollector` class was removed; functions are now used as common modules.
- CHANGED: all templates created with `template_for()` are now rendered with `from __future__ import division`.
- CHANGED: signature of `OperationRecorder.add_kernel` takes a renderable instead of a full template.
- CHANGED: `compile_static()` now takes a template instead of a source.
- CHANGED: `reikna.elementwise.Elementwise` now uses modules.
- FIXED: potential problem with local size finidng in static kernels (first approximation for the maximum workgroup size was not that good)
- FIXED: some OpenCL compilation warnings caused by an incorrect version querying macro.
- FIXED: bug with incorrect processing of scalar global size in static kernels.
- FIXED: bug in variance estimates in CBRNG tests.
- FIXED: error in the temporary varaiable type in `reikna.cluda.functions.polar()` and `reikna.cluda.functions.exp()`.

2.6.19 0.2.1 (8 Mar 2013)

- FIXED: function names for kernel `polar()`, `exp()` and `conj()`.
- FIXED: added forgotten kernel `norm()` handler.
- FIXED: bug in `Py.Test` testcase execution hook which caused every test to run twice.
- FIXED: bug in nested computation processing for computation with more than one kernel.
- FIXED: added dependencies between `MatrixMul` kernel arguments.
- FIXED: taking into account dependencies between input and output arrays as well as the ones between internal allocations — necessary for nested computations.
- ADDED: discrete harmonic transform `DHT` (calculated using Gauss-Hermite quadrature).

2.6.20 0.2.0 (3 Mar 2013)

- Added FFT computation (slightly optimized PyFFT version + Bluestein's algorithm for non-power-of-2 FFT sizes)
- Added Python 3 compatibility
- Added Thread-global automatic memory packing
- Added polar(), conj() and exp() functions to kernel toolbox
- Changed name because of the clash with another Tigger.

2.6.21 0.1.0 (12 Sep 2012)

- Lots of changes in the API
- Added elementwise, reduction and transposition computations
- Extended API reference and added topical guides

2.6.22 0.0.1 (22 Jul 2012)

- Created basic core for computations and transformations
- Added matrix multiplication computation
- Created basic documentation

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

r