reikna Documentation

Release 0.6.4

Bogdan Opanchuk

September 28, 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

This module contains information about the library version.

`reikna.version.`**`version`**
A tuple with version numbers, major components first.

`reikna.version.`**`full_version`**
A string fully identifying the current build.

`reikna.version.`**`git_revision`**
A string with Git SHA identifying the revision used to create this build.

`reikna.version.`**`release`**
A boolean variable, equals `True` if current version is a release version.

2.5.2 Helpers

This module contains various auxiliary functions which are used throughout the library.

`reikna.helpers.`**`bounding_power_of_2`**(*num*)
Returns the minimal number of the form `2**m` such that it is greater or equal to `n`.

`reikna.helpers.`**`factors`**(*num*, *limit=None*)
Returns the list of pairs `(factor, num/factor)` for all factors of `num` (including 1 and `num`), sorted by `factor`. If `limit` is set, only pairs with `factor <= limit` are returned.

class `reikna.helpers.`**`ignore_integer_overflow`**
Context manager for ignoring integer overflow in numpy operations on scalars (not ignored by default because of a bug in numpy).

`reikna.helpers.`**`log2`**(*num*)
Integer-valued logarigthm with base 2. If `n` is not a power of 2, the result is rounded to the smallest number.

`reikna.helpers.`**`min_blocks`**(*length*, *block*)
Returns minimum number of blocks with length `block` necessary to cover the array with length `length`.

reikna.helpers.**product**(*seq*)
Returns the product of elements in the iterable `seq`.

reikna.helpers.**template_def**(*signature*, *code*)
Returns a `Mako` template with the given `signature`.

Parameters **signature** – a list of postitional argument names, or a `Signature` object from `funcsigs` module.

Code a body of the template.

reikna.helpers.**template_for**(*filename*)
Returns the Mako template object created from the file which has the same name as `filename` and the extension `.mako`. Typically used in computation modules as `template_for(__filename__)`.

reikna.helpers.**template_from**(*template*)
Creates a Mako template object from a given string. If `template` already has `render()` method, does nothing.

reikna.helpers.**wrap_in_tuple**(*seq_or_elem*)
If `seq_or_elem` is a sequence, converts it to a `tuple`, otherwise returns a tuple with a single element `seq_or_elem`.

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.

class reikna.cluda.**Module**(*template_src*, *render_kwds=None*)
Contains a CLUDA module. See *Tutorial: modules and snippets* for details.

Parameters

- **template_src** (`str` or `Mako` template.) – a `Mako` template with the module code, or a string with the template source.
- **render_kwds** – a dictionary which will be used to render the template. Can contain other modules and snippets.

classmethod **create**(*func_or_str*, *render_kwds=None*)
Creates a module from the `Mako` def:

- if `func_or_str` is a function, then the def has the same signature as `func_or_str` (prefix will be passed as the first positional parameter), and the body equal to the string it returns;
- if `func_or_str` is a string, then the def has a single positional argument `prefix`. and the body `code`.

exception reikna.cluda.**OutOfResourcesError**
Thrown by `compile_static()` if the provided `local_size` is too big, or one cannot be found.

class reikna.cluda.**Snippet**(*template_src*, *render_kwds=None*)
Contains a CLUDA snippet. See *Tutorial: modules and snippets* for details.

Parameters

- **template_src** (`str` or `Mako` template.) – a `Mako` template with the module code, or a string with the template source.

- **render_kwds** – a dictionary which will be used to render the template. Can contain other modules and snippets.

classmethod `create`(*func_or_str*, *render_kwds=None*)
Creates a snippet from the `Mako` def:

- if `func_or_str` is a function, then the def has the same signature as `func_or_str`, and the body equal to the string it returns;
- if `func_or_str` is a string, then the def has empty signature.

`reikna.cluda.`**`any_api`**`()`
Returns one of the API modules supported by the system or raises an `Exception` if there are not any.

`reikna.cluda.`**`api_ids`**`()`
Returns a list of identifiers for all known (not necessarily available for the current system) APIs.

`reikna.cluda.`**`cuda_api`**`()`
Returns the `PyCUDA`-based API module.

`reikna.cluda.`**`cuda_id`**`()`
Returns the identifier of the `PyCUDA`-based API.

`reikna.cluda.`**`find_devices`**(*api*, *include_devices=None*, *exclude_devices=None*, *include_platforms=None*, *exclude_platforms=None*, *include_duplicate_devices=True*, *include_pure_only=False*)
Find platforms and devices meeting certain criteria.

Parameters

- **api** – a CLUDA API object.
- **include_devices** – a list of masks for a device name which will be used to pick devices to include in the result.
- **exclude_devices** – a list of masks for a device name which will be used to pick devices to exclude from the result.
- **include_platforms** – a list of masks for a platform name which will be used to pick platforms to include in the result.
- **exclude_platforms** – a list of masks for a platform name which will be used to pick platforms to exclude in the result.
- **include_duplicate_devices** – if `False`, will only include a single device from the several with the same name available on a platform.
- **include_pure_only** – if `True`, will include devices with maximum group size equal to 1.

Returns a dictionary with found platform numbers as keys, and lists of device numbers as values.

`reikna.cluda.`**`get_api`**(*api_id*)
Returns an API module with the generalized interface `reikna.cluda.api` for the given identifier.

`reikna.cluda.`**`ocl_api`**`()`
Returns the `PyOpenCL`-based API module.

`reikna.cluda.`**`ocl_id`**`()`
Returns the identifier of the `PyOpenCL`-based API.

`reikna.cluda.`**`supported_api_ids`**`()`
Returns a list of identifiers of supported APIs.

`reikna.cluda.`**`supports_api`**(*api_id*)
Returns `True` if given API is supported.

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`.

class `reikna.cluda.api.`**Buffer**
Low-level untyped memory allocation. Actual class depends on the API: `pycuda.driver.DeviceAllocation` for `CUDA` and `pyopencl.Buffer` for `OpenCL`.

size

class `reikna.cluda.api.`**Array**
A superclass of the corresponding API's native array (`pycuda.gpuarray.GPUArray` for `CUDA` and `pyopencl.array.Array` for `OpenCL`), with some additional functionality.

shape

dtype

get()
Returns `numpy.ndarray` with the contents of the array. Synchronizes the parent `Thread`.

thread
The `Thread` object for which the array was created.

class `reikna.cluda.api.`**DeviceParameters**(*device*)
An assembly of device parameters necessary for optimizations.

max_work_group_size
Maximum block size for kernels.

max_work_item_sizes
List with maximum local_size for each dimension.

max_num_groups
List with maximum number of workgroups for each dimension.

warp_size
Warp size (nVidia), or wavefront size (AMD), or SIMD width is supposed to be the number of threads that are executed simultaneously on the same computation unit (so you can assume that they are perfectly synchronized).

local_mem_banks
Number of local (shared in CUDA) memory banks is a number of successive 32-bit words you can access without getting bank conflicts.

local_mem_size
Size of the local (shared in CUDA) memory per workgroup, in bytes.

min_mem_coalesce_width
Dictionary `{word_size:elements}`, where `elements` is the number of elements with size `word_size` in global memory that allow coalesced access.

supports_dtype(*self*, *dtype*)
Checks if given `numpy` dtype can be used in kernels compiled using this thread.

class `reikna.cluda.api.`**Platform**
A vendor-specific implementation of the GPGPU API.

name
Platform name.

vendor
Vendor name.

version
Platform version.

get_devices()
Returns a list of device objects available in the platform.

class reikna.cluda.api.**Kernel**
An object containing GPU kernel.

max_work_group_size
Maximum size of the work group for the kernel.

__call__(**args*, ***kwds*)
A shortcut for successive call to `prepare()` and `prepared_call()`.

prepare(*global_size*, *local_size=None*, *local_mem=0*)
Prepare the kernel for execution with given parameters.

Parameters

- **global_size** – an integer or a tuple of integers, specifying total number of work items to run.
- **local_size** – an integer or a tuple of integers, specifying the size of a single work group. Should have the same number of dimensions as `global_size`. If `None` is passed, some `local_size` will be picked internally.
- **local_mem** – (**CUDA API only**) amount of dynamic local memory (in bytes)

prepared_call(**args*)
Execute the kernel. `Array` objects are allowed as arguments.

class reikna.cluda.api.**Program**
An object with compiled GPU code.

source
Contains module source code.

kernel_name
Contains `Kernel` object for the kernel `kernel_name`.

class reikna.cluda.api.**StaticKernel**
An object containing a GPU kernel with fixed call sizes.

source
Contains the source code of the program.

__call__(**args*)
Execute the kernel. `Array` objects are allowed as arguments.

class reikna.cluda.api.**Thread**(*cqd*, *async=True*, *temp_alloc=None*)
Wraps an existing context in the CLUDA thread object.

Parameters

- **cqd** – a `Context`, `Device` or `Stream`/`CommandQueue` object to base on. If a context is passed, a new stream/queue will be created internally.
- **async** – whether to execute all operations with this thread asynchronously (you would generally want to set it to `False` only for profiling purposes).

Note: If you are using CUDA API, you must keep in mind the stateful nature of CUDA calls. Briefly, this means that there is the context stack, and the current context on top of it. When the `create()` is called, the PyCUDA context gets pushed to the stack and made current. When the thread object goes out of scope (and the thread object owns it), the context is popped, and it is the user's responsibility to make sure the popped context is the correct one. In simple single-context programs this only means that one should avoid reference cycles involving the thread object.

Warning: Do not pass one `Stream`/`CommandQueue` object to several `Thread` objects.

api
Module object representing the CLUDA API corresponding to this `Thread`.

device_params
Instance of `DeviceParameters` class for this thread's device.

temp_alloc
Instance of `TemporaryManager` which handles allocations of temporary arrays (see `temp_array()`).

allocate(*size*)
Creates an untyped memory allocation object of type `Buffer` with size `size`.

array(*shape*, *dtype*, *strides=None*, *allocator=None*)
Creates an `Array` on GPU with given `shape`, `dtype` and `strides`. Optionally, an `allocator` is a callable returning any object castable to `int` representing the physical address on the device (for instance, `Buffer`).

compile(*template_src*, *render_args=None*, *render_kwds=None*, *fast_math=False*)
Creates a module object from the given template.

Parameters

- **template_src** – Mako template source to render
- **render_kwds** – an iterable with positional arguments to pass to the template.
- **render_kwds** – a dictionary with keyword parameters to pass to the template.
- **fast_math** – whether to enable fast mathematical operations during compilation.

Returns a `Program` object.

compile_static(*template_src*, *name*, *global_size*, *local_size=None*, *render_args=None*, *render_kwds=None*, *fast_math=False*)
Creates a kernel object with fixed call sizes, which allows to overcome some backend limitations. Global and local sizes can have any length, providing that `len(global_size) >= len(local_size)`, and the total number of work items and work groups is less than the corresponding total number available for the device. In order to get IDs and sizes in such kernels, virtual size functions have to be used (see `VIRTUAL_SKIP_THREADS` and others for details).

Parameters

- **template_src** – Mako template or a template source to render
- **name** – name of the kernel function
- **global_size** – global size to be used, in **row-major** order.
- **local_size** – local size to be used, in **row-major** order. If `None`, some suitable one will be picked.

- **local_mem** – (**CUDA API only**) amount of dynamically allocated local memory to be used (in bytes).
- **render_args** – a list of parameters to be passed as positional arguments to the template.
- **render_kwds** – a dictionary with additional parameters to be used while rendering the template.
- **fast_math** – whether to enable fast mathematical operations during compilation.

Returns a `StaticKernel` object.

copy_array(*arr*, *dest=None*, *src_offset=0*, *dest_offset=0*, *size=None*)
Copies array on device.

Parameters

- **dest** – the effect is the same as in `to_device()`.
- **src_offset** – offset (in items of `arr.dtype`) in the source array.
- **dest_offset** – offset (in items of `arr.dtype`) in the destination array.
- **size** – how many elements of `arr.dtype` to copy.

classmethod create(*interactive=False*, *device_filters=None*, ***thread_kwds*)
Creates a new `Thread` object with its own context and queue inside. Intended for cases when you want to base your whole program on CLUDA.

Parameters

- **interactive** – ask a user to choose a platform and a device from the ones found. If there is only one platform/device available, they will be chosen automatically.
- **device_filters** – keywords to filter devices (see the keywords for `find_devices()`).
- **thread_kwds** – keywords to pass to `Thread` constructor.
- **kwds** – same as in `Thread`.

empty_like(*arr*)
Allocates an array on GPU with the same attributes as `arr`.

from_device(*arr*, *dest=None*, *async=False*)
Transfers the contents of `arr` to a `numpy.ndarray` object. The effect of `dest` parameter is the same as in `to_device()`. If `async` is `True`, the transfer is asynchronous (the thread-wide asynchronisity setting does not apply here).

Alternatively, one can use `Array.get()`.

release()
Forcefully free critical resources (rendering the object unusable). In most cases you can rely on the garbage collector taking care of things. Calling this method explicitly may be necessary in case of CUDA API when you want to make sure the context got popped.

synchronize()
Forcefully synchronize this thread with the main program.

temp_array(*shape*, *dtype*, *strides=None*, *dependencies=None*)
Creates an `Array` on GPU with given `shape`, `dtype` and `strides`. In order to reduce the memory footprint of the program, the temporary array manager will allow these arrays to overlap. Two arrays will not overlap, if one of them was specified in `dependencies` for the other one. For a list of values `dependencies` takes, see the reference entry for `TemporaryManager`.

to_device(*arr*, *dest=None*)
Copies an array to the device memory. If `dest` is specified, it is used as the destination, and the method returns `None`. Otherwise the destination array is created internally and returned from the method.

`reikna.cluda.api.`**get_id**()
Returns the identifier of this API.

`reikna.cluda.api.`**get_platforms**()
Returns a list of available `Platform` objects. In case of `OpenCL` returned objects are actually instances of `pyopencl.Platform`.

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.

class `reikna.cluda.tempalloc.`**TemporaryManager**(*thr*, *pack_on_alloc=False*, *pack_on_free=False*)
Base class for a manager of temporary allocations.

Parameters

- **thr** – an instance of `Thread`.
- **pack_on_alloc** – whether to repack allocations when a new allocation is requested.
- **pack_on_free** – whether to repack allocations when an allocation is freed.

array(*shape*, *dtype*, *strides=None*, *dependencies=None*)
Returns a temporary array.

Parameters

- **shape** – shape of the array.
- **dtype** – data type of the array.
- **strides** – tuple of bytes to step in each dimension when traversing an array.
- **dependencies** – can be a `Array` instance (the ones containing persistent allocations will be ignored), an iterable with valid values, or an object with the attribute *__tempalloc__* which is a valid value (the last two will be processed recursively).

pack()
Packs the real allocations possibly reducing total memory usage. This process can be slow.

class `reikna.cluda.tempalloc.`**TrivialManager**(**args*, ***kwds*)
Trivial manager — allocates a separate buffer for each allocation request.

class `reikna.cluda.tempalloc.`**ZeroOffsetManager**(**args*, ***kwds*)
Tries to assign several allocation requests to a single real allocation, if dependencies allow that. All virtual allocations start from the beginning of real allocations.

Function modules

This module contains `Module` factories which are used to compensate for the lack of complex number operations in OpenCL, and the lack of C++ synthax which would allow one to write them.

reikna.cluda.functions.**add**(**in_dtypes*, *out_dtype=None*)
Returns a Module with a function of len(in_dtypes) arguments that adds values of types in_dtypes. If out_dtype is given, it will be set as a return type for this function.

This is necessary since on some platforms the + operator for a complex and a real number works in an unexpected way (returning (a.x + b, a.y + b) instead of (a.x + b, a.y)).

reikna.cluda.functions.**cast**(*out_dtype*, *in_dtype*)
Returns a Module with a function of one argument that casts values of in_dtype to out_dtype.

reikna.cluda.functions.**conj**(*dtype*)
Returns a Module with a function of one argument that conjugates the value of type dtype (must be a complex data type).

reikna.cluda.functions.**div**(*in_dtype1*, *in_dtype2*, *out_dtype=None*)
Returns a Module with a function of two arguments that divides values of in_dtype1 and in_dtype2. If out_dtype is given, it will be set as a return type for this function.

reikna.cluda.functions.**exp**(*dtype*)
Returns a Module with a function of one argument that exponentiates the value of type dtype (must be a real or complex data type).

reikna.cluda.functions.**mul**(**in_dtypes*, *out_dtype=None*)
Returns a Module with a function of len(in_dtypes) arguments that multiplies values of types in_dtypes. If out_dtype is given, it will be set as a return type for this function.

reikna.cluda.functions.**norm**(*dtype*)
Returns a Module with a function of one argument that returns the 2-norm of the value of type dtype (product by the complex conjugate if the value is complex, square otherwise).

reikna.cluda.functions.**polar**(*dtype*)
Returns a Module with a function of two arguments that returns the complex-valued rho * exp(i * theta) for values rho, theta of type dtype (must be a real data type).

reikna.cluda.functions.**polar_unit**(*dtype*)
Returns a Module with a function of one argument that returns a complex number (cos(theta), sin(theta)) for a value theta of type dtype (must be a real data type).

reikna.cluda.functions.**pow**(*dtype*, *exponent_dtype=None*, *output_dtype=None*)
Returns a Module with a function of two arguments that raises the first argument of type dtype to the power of the second argument of type exponent_dtype (an integer or real data type). If exponent_dtype or output_dtype are not given, they default to dtype. If dtype is not the same as output_dtype, the input is cast to output_dtype *before* exponentiation. If exponent_dtype is real, but both dtype and output_dtype are integer, a ValueError is raised.

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.

`reikna.cluda.dtypes.`**align**(*dtype*)
Returns a new struct dtype with the field offsets changed to the ones a compiler would use (without being given any explicit alignment qualifiers). Ignores all existing explicit itemsizes and offsets.

`reikna.cluda.dtypes.`**c_constant**(*val*, *dtype=None*)
Returns a C-style numerical constant. If `val` has a struct dtype, the generated constant will have the form `{ ... }` and can be used as an initializer for a variable.

`reikna.cluda.dtypes.`**c_path**(*path*)
Returns a string corresponding to the `path` to a struct element in C. The `path` is the sequence of field names/array indices returned from `flatten_dtype()`.

`reikna.cluda.dtypes.`**cast**(*dtype*)
Returns function that takes one argument and casts it to `dtype`.

`reikna.cluda.dtypes.`**complex_ctr**(*dtype*)
Returns name of the constructor for the given `dtype`.

`reikna.cluda.dtypes.`**complex_for**(*dtype*)
Returns complex dtype corresponding to given floating point `dtype`.

`reikna.cluda.dtypes.`**ctype**(*dtype*)
For a built-in C type, returns a string with the name of the type.

`reikna.cluda.dtypes.`**ctype_module**(*dtype*, *ignore_alignment=False*)
For a struct type, returns a `Module` object with the `typedef` of a struct corresponding to the given `dtype` (with its name set to the module prefix); falls back to `ctype()` otherwise.

The structure definition includes the alignment required to produce field offsets specified in `dtype`; therefore, `dtype` must be either a simple type, or have proper offsets and dtypes (the ones that can be reporoduced in C using explicit alignment attributes, but without additional padding) and the attribute `isalignedstruct == True`. An aligned dtype can be produced either by standard means (`aligned` flag in `numpy.dtype` constructor and explicit offsets and itemsizes), or created out of an arbitrary dtype with the help of `align()`.

If `ignore_alignment` is True, all of the above is ignored. The C structures produced will not have any explicit alignment modifiers. As a result, the the field offsets of `dtype` may differ from the ones chosen by the compiler.

Modules are cached and the function returns a single module instance for equal `dtype`'s. Therefore inside a kernel it will be rendered with the same prefix everywhere it is used. This results in a behavior characteristic for a structural type system, same as for the basic dtype-ctype conversion.

> **Warning:** As of `numpy` 1.8, the `isalignedstruct` attribute is not enough to ensure a mapping between a dtype and a C struct with only the fields that are present in the dtype. Therefore, `ctype_module` will make some additional checks and raise `ValueError` if it is not the case.

`reikna.cluda.dtypes.`**`detect_type`**(*val*)
Find out the data type of `val`.

`reikna.cluda.dtypes.`**`extract_field`**(*arr*, *path*)
Extracts an element from an array of struct dtype. The `path` is the sequence of field names/array indices returned from `flatten_dtype()`.

`reikna.cluda.dtypes.`**`flatten_dtype`**(*dtype*)
Returns a list of tuples `(path, dtype)` for each of the basic dtypes in a (possibly nested) `dtype`. `path` is a list of field names/array indices leading to the corresponding element.

`reikna.cluda.dtypes.`**`is_complex`**(*dtype*)
Returns `True` if `dtype` is complex.

`reikna.cluda.dtypes.`**`is_double`**(*dtype*)
Returns `True` if `dtype` is double precision floating point.

`reikna.cluda.dtypes.`**`is_integer`**(*dtype*)
Returns `True` if `dtype` is an integer.

`reikna.cluda.dtypes.`**`is_real`**(*dtype*)
Returns `True` if `dtype` is a real.

`reikna.cluda.dtypes.`**`min_scalar_type`**(*val*)
Wrapper for `numpy.min_scalar_dtype` which takes into account types supported by GPUs.

`reikna.cluda.dtypes.`**`normalize_type`**(*dtype*)
Function for wrapping all dtypes coming from the user. `numpy` uses two different classes to represent dtypes, and one of them does not have some important attributes.

`reikna.cluda.dtypes.`**`normalize_types`**(*dtypes*)
Same as `normalize_type()`, but operates on a list of dtypes.

`reikna.cluda.dtypes.`**`real_for`**(*dtype*)
Returns floating point dtype corresponding to given complex `dtype`.

`reikna.cluda.dtypes.`**`result_type`**(**dtypes*)
Wrapper for `numpy.result_type` which takes into account types supported by GPUs.

`reikna.cluda.dtypes.`**`zero_ctr`**(*dtype*)
Returns the string with constructed zero value for the given `dtype`.

2.5.4 Core functionality

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

Computation signatures

class `reikna.core.`**`Type`**(*dtype*, *shape=None*, *strides=None*)
Represents an array or, as a degenerate case, scalar type of a computation parameter.

`shape`
A tuple of integers. Scalars are represented by an empty tuple.

dtype
A `numpy.dtype` instance.

ctype
A string with the name of C type corresponding to `dtype`, or a module if it is a struct type.

strides
Tuple of bytes to step in each dimension when traversing an array.

__call__(*val*)
Casts the given value to this type.

classmethod from_value(*val*)
Creates a `Type` object corresponding to the given value.

class reikna.core.**Annotation**(*type_*, *role=None*)
Computation parameter annotation, in the same sense as it is used for functions in the standard library.

Parameters

- **type** – a `Type` object.
- **role** – any of `'i'` (input), `'o'` (output), `'io'` (input/output), `'s'` (scalar). Defaults to `'s'` for scalars and `'io'` for arrays.

class reikna.core.**Parameter**(*name*, *annotation*, *default=<class 'funcsigs._empty'>*)
Computation parameter, in the same sense as it is used for functions in the standard library. In its terms, all computation parameters have kind `POSITIONAL_OR_KEYWORD`.

Parameters

- **name** – parameter name.
- **annotation** – an `Annotation` object.
- **default** – default value for the parameter, can only be specified for scalars.

rename(*new_name*)
Creates a new `Parameter` object with the new name and the same annotation and default value.

class reikna.core.**Signature**(*parameters*)
Computation signature, in the same sense as it is used for functions in the standard library.

Parameters **parameters** – a list of `Parameter` objects.

parameters
An `OrderedDict` with `Parameter` objects indexed by their names.

bind_with_defaults(*args*, *kwds*, *cast=False*)
Binds passed positional and keyword arguments to parameters in the signature and returns the resulting `BoundArguments` object.

Core classes

class reikna.core.**Computation**(*root_parameters*)
A base class for computations, intended to be subclassed.

Parameters **root_parameters** – a list of `Parameter` objects.

signature
A `Signature` object representing current computation signature (taking into account connected transformations).

parameter
A named tuple of `ComputationParameter` objects corresponding to parameters from the current `signature`.

_build_plan(*plan_factory*, *device_params*, **args*)
Derived classes override this method. It is called by `compile()` and supposed to return a `ComputationPlan` object.

Parameters

- **plan_factory** – a callable returning a new `ComputationPlan` object.
- **device_params** – a `DeviceParameters` object corresponding to the thread the computation is being compiled for.
- **args** – `KernelArgument` objects, corresponding to `parameters` specified during the creation of this computation object.

_update_attributes()
Updates `signature` and `parameter` attributes. Called by the methods that change the signature.

compile(*thread*, *fast_math=False*)
Compiles the computation with the given `Thread` object and returns a `ComputationCallable` object. If `fast_math` is enabled, the compilation of all kernels is performed using the compiler options for fast and imprecise mathematical functions.

connect(*_comp_connector*, *_trf*, *_tr_connector*, ***tr_from_comp*)
Connect a transformation to the computation.

Parameters

- **_comp_connector** – connection target — a `ComputationParameter` object beloning to this computation object, or a string with its name.
- **_trf** – a `Transformation` object.
- **_tr_connector** – connector on the side of the transformation — a `TransformationParameter` object beloning to `tr`, or a string with its name.
- **tr_from_comp** – a dictionary with the names of new or old computation parameters as keys, and `TransformationParameter` objects (or their names) as values.

Returns this computation object (modified).

Note: The resulting parameter order is determined by traversing the graph of connections depth-first (starting from the initial computation parameters), with the additional condition: the nodes do not change their order in the same branching level (i.e. in the list of computation or transformation parameters, both of which are ordered).

For example, consider a computation with parameters `(a, b, c, d)`. If you connect a transformation `(a', c) -> a`, the resulting computation will have the signature `(a', b, c, d)` (as opposed to `(a', c, b, d)` it would have for the pure depth-first traversal).

---

class `reikna.core.`**Transformation**(*parameters*, *code*, *render_kwds=None*, *connectors=None*)
A class containing a pure parallel transformation of arrays. Some restrictions apply:

•it cannot use local memory;

•it cannot use global/local id getters (and depends only on externally passed indices);

•it cannot have `'io'` arguments;

•it has at least one argument that uses `load_same` or `store_same`, and does it only once.

Parameters

- **parameters** – a list of `Parameter` objects.
- **code** – a source template for the transformation. Will be wrapped in a template def with positional arguments with the names of objects in `parameters`.
- **render_kwds** – a dictionary with render keywords that will be passed to the snippet.
- **connectors** – a list of parameter names suitable for connection. Defaults to all non-scalar parameters.

Result and attribute classes

class `reikna.core.`**`Indices`**
Encapsulates the information about index variables available for the snippet.

`__getitem__`(*dim*)
Returns the name of the index varibale for the dimension `dim`.

`all`()
Returns the comma-separated list of all index variable names (useful for passing the guiding indices verbatim in a load or store call).

class `reikna.core.computation.`**`ComputationCallable`**(*parameters*, *kernel_calls*, *internal_args*, *temp_buffers*)
A result of calling `compile()` on a computation. Represents a callable opaque GPGPU computation.

`signature`
A `Signature` object.

`parameter`
A named tuple of `Type` objects corresponding to the callable's parameters.

`__call__`(**args*, ***kwds*)
Execute the computation.

class `reikna.core.computation.`**`ComputationParameter`**
Bases: `Type`

Represents a typed computation parameter. Can be used as a substitute of an array for functions which are only interested in array metadata.

`connect`(*_trf*, *_tr_connector*, ***tr_from_comp*)
Shortcut for `connect()` with this parameter as a first argument.

class `reikna.core.computation.`**`KernelArgument`**
Bases: `Type`

Represents an argument suitable to pass to planned kernel or computation call.

class `reikna.core.computation.`**`ComputationPlan`**
Computation plan recorder.

`computation_call`(*computation*, **args*, ***kwds*)
Adds a nested computation call. The `computation` value must be a `Computation` object. `args` and `kwds` are values to be passed to the computation.

`kernel_call`(*template_def*, *args*, *global_size*, *local_size=None*, *render_kwds=None*)
Adds a kernel call to the plan.

Parameters

- **template_def** – Mako template def for the kernel.
- **args** – a list consisting of `KernelArgument` objects, or scalar values wrapped in `numpy.ndarray`, that are going to be passed to the kernel during execution.
- **global_size** – global size to use for the call, in **row-major** order.
- **local_size** – local size to use for the call, in **row-major** order. If `None`, the local size will be picked automatically.
- **render_kwds** – dictionary with additional values used to render the template.

persistent_array(*arr*)
Adds a persistent GPU array to the plan, and returns the corresponding `KernelArgument`.

temp_array(*shape*, *dtype*, *strides=None*)
Adds a temporary GPU array to the plan, and returns the corresponding `KernelArgument`. Temporary arrays can share physical memory, but in such a way that their contents is guaranteed to persist between the first and the last use in a kernel during the execution of the plan.

temp_array_like(*arr*)
Same as `temp_array()`, taking the array properties from array or array-like object `arr`.

class `reikna.core.transformation.`**TransformationParameter**(*trf*, *name*, *type_*)
Bases: `Type`

Represents a typed transformation parameter. Can be used as a substitute of an array for functions which are only interested in array metadata.

class `reikna.core.transformation.`**KernelParameter**
Providing an interface for accessing kernel arguments in a template. Depending on the parameter type, and whether it is used inside a computation or a transformation template, can have different load/store attributes available.

name
Parameter name

shape

dtype

ctype

strides
Same as in `Type`.

__str__()
Returns the C kernel parameter name corresponding to this parameter. It is the only method available for scalar parameters.

load_idx
A module providing a macro with the signature `(idx0, idx1, ...)`, returning the corresponding element of the array.

store_idx
A module providing a macro with the signature `(idx0, idx1, ..., val)`, saving `val` into the specified position.

load_combined_idx(*slices*)
A module providing a macro with the signature `(cidx0, cidx1, ...)`, returning the element of the array corresponding to the new slicing of indices (e.g. an array with shape `(2, 3, 4, 5, 6)` sliced as `slices=(2, 2, 1)` is indexed as an array with shape `(6, 20, 6)`).

store_combined_idx(*slices*)
A module providing a macro with the signature `(cidx0, cidx1, ..., val)`, saving `val` into the specified position corresponding to the new slicing of indices.

load_same
A module providing a macro that returns the element of the array corresponding to the indices used by the caller of the transformation.

store_same
A module providing a macro with the signature `(val)` that stores `val` using the indices used by the caller of the transformation.

2.5.5 Computations

Algorithms

General purpose algorithms.

Pure parallel computations

class reikna.algorithms.**PureParallel**(*parameters*, *code*, *guiding_array=None*, *render_kwds=None*)
Bases: Computation

A general class for pure parallel computations (i.e. with no interaction between threads).

Parameters

- **parameters** – a list of `Parameter` objects.
- **code** – a source code for the computation. Can be a `Snippet` object which will be passed `Indices` object for the `guiding_array` as the first positional argument, and `KernelParameter` objects corresponding to `parameters` as the rest of positional arguments. If it is a string, such `Snippet` will be created out of it, with the parameter names `idxs` for the first one and the names of `parameters` for the remaining ones.
- **guiding_array** – an tuple with the array shape, or the name of one of `parameters`. By default, the first parameter is chosen.
- **render_kwds** – a dictionary with render keywords for the `code`.

compiled_signature(**args*)

Parameters **args** – corresponds to the given `parameters`.

classmethod from_trf(*trf*, *guiding_array=None*)
Creates a `PureParallel` instance from a `Transformation` object. `guiding_array` can be a string with a name of an array parameter from `trf`, or the corresponding `TransformationParameter` object.

Transposition (permutation)

class reikna.algorithms.**Transpose**(*arr_t*, *axes=None*, *block_width_override=None*)
Bases: Computation

Changes the order of axes in a multidimensional array. Works analogous to `numpy.transpose`.

Parameters

- **arr_t** – an array-like defining the initial array.
- **axes** – tuple with the new axes order. If `None`, then axes will be reversed.

compiled_signature(*output:o*, *input:i*)

Parameters

- **output** – an array with all the attributes of `arr_t`, with the shape permuted according to `axes`.
- **input** – an array with all the attributes of `arr_t`.

Reduction

class reikna.algorithms.**Predicate**(*operation*, *empty*)
A predicate used in `Reduce`.

Parameters

- **operation** – a `Snippet` object with two parameters which will take the names of two arguments to join.
- **empty** – a numpy scalar with the empty value of the argument (the one which, being joined by another argument, does not change it).

reikna.algorithms.**predicate_sum**(*dtype*)
Returns a `Predicate` object which sums its arguments.

class reikna.algorithms.**Reduce**(*arr_t*, *predicate*, *axes=None*)
Bases: `Computation`

Reduces the array over given axis using given binary operation.

Parameters

- **arr_t** – an array-like defining the initial array.
- **predicate** – a `Predicate` object.
- **axes** – a list of non-repeating axes to reduce over. If `None`, the whole array will be reduced (in which case the shape of the output array is `(1,)`).

compiled_signature(*output:o*, *input:i*)

Parameters

- **input** – an array with the attributes of `arr_t`.
- **output** – an array with the attributes of `arr_t`, with its shape missing axes from `axes`.

Linear algebra

Linear algebra algorithms.

Matrix multiplication (dot product)

class reikna.linalg.**MatrixMul**(*a_arr*, *b_arr*, *out_arr=None*, *block_width_override=None*, *transposed_a=False*, *transposed_b=False*)
Bases: `Computation`

Multiplies two matrices using last two dimensions and batching over remaining dimensions. For batching to work, the products of remaining dimensions should be equal (then the multiplication will be performed piecewise), or one of them should equal 1 (then the multiplication will be batched over the remaining dimensions of the other matrix).

Parameters

- **a_arr** – an array-like defining the first argument.
- **b_arr** – an array-like defining the second argument.
- **out_arr** – an array-like definign the output; if not given, both shape and dtype will be derived from `a_arr` and `b_arr`.
- **block_width_override** – if provided, it will used as a block size of the multiplication kernel.
- **transposed_a** – if `True`, the first matrix will be transposed before the multiplication.
- **transposed_b** – if `True`, the second matrix will be transposed before the multiplication.

compiled_signature(*output:o*, *matrix_a:i*, *matrix_b:i*)

Parameters

- **output** – the output of matrix multiplication.
- **matrix_a** – the first argument.
- **matrix_b** – the second argument.

Matrix norms

class reikna.linalg.**EntrywiseNorm**(*arr_t*, *order=2*, *axes=None*)
Bases: Computation

Calculates the entrywise matrix norm (same as `numpy.linalg.norm`) of an arbitrary order r:

$$||A||_r = \left(\sum_{i,j,\ldots} |A_{i,j,\ldots}|^r \right)^{1/r}$$

Parameters

- **arr_t** – an array-like defining the initial array.
- **order** – the order r (any real number).
- **axes** – a list of non-repeating axes to sum over. If `None`, the norm of the whole array will be calculated.

compiled_signature(*output:o*, *input:i*)

Parameters

- **input** – an array with the attributes of `arr_t`.
- **output** – an array with the attributes of `arr_t`, with its shape missing axes from `axes`.

Fast Fourier transform

class reikna.fft.**FFT**(*arr_t*, *axes=None*)
Bases: Computation

Performs the Fast Fourier Transform. The interface is similar to `numpy.fft.fftn`. The inverse transform is normalized so that `IFFT(FFT(X)) = X`.

Parameters

- **arr_t** – an array-like defining the problem array.
- **axes** – a tuple with axes over which to perform the transform. If not given, the transform is performed over all the axes.

Note: Current algorithm works most effectively with array dimensions being power of 2 This mostly applies to the axes over which the transform is performed, beacuse otherwise the computation falls back to the Bluestein's algorithm, which effectively halves the performance.

---

compiled_signature(*output:o*, *input:i*, *inverse:s*)

`output` and `input` may be the same array.

Parameters

- **output** – an array wit the attributes of `arr_t`.
- **input** – an array wit the attributes of `arr_t`.
- **inverse** – a scalar value castable to integer. If `1`, `output` contains the forward FFT of `input`, if `0` the inverse one.

Discrete harmonic transform

reikna.dht.**get_spatial_grid**(*modes*, *order*, *add_points=0*)

Returns the spatial grid required to calculate the `order` power of a function defined in the harmonic mode space of the size `modes`. If `add_points` is 0, the grid has the minimum size required for exact transformation back to the mode space.

reikna.dht.**harmonic**(*mode*)

Returns an eigenfunction of order $n = \text{mode}$ for the harmonic oscillator:

$$\phi_n = \frac{1}{\sqrt[4]{\pi}\sqrt{2^n n!}} H_n(x) \exp(-x^2/2),$$

where H_n is the n-th order "physicists"' Hermite polynomial. The normalization is chosen so that $\int \phi_n^2(x)dx = 1$.

class reikna.dht.**DHT**(*mode_arr*, *add_points=None*, *inverse=False*, *order=1*, *axes=None*)

Bases: `Computation`

Discrete transform to and from harmonic oscillator modes. With `inverse=True` transforms a function defined by its expansion C_m, $m = 0 \dots M-1$ in the mode space with mode functions from `harmonic()`, to the coordinate space ($F(x)$ on the grid x from `get_spatial_grid()`). With `inverse=False` guarantees to recover first M modes of $F^k(x)$, where k is the `order` parameter.

For multiple dimensions the operation is the same, and the mode functions are products of 1D mode functions, i.e. $\phi^{3D}_{l,m,n}(x,y,z) = \phi_l(x)\phi_m(y)\phi_n(z)$.

For the detailed description of the algorithm, see Dion & Cances, PRE 67(4) 046706 (2003)

Parameters

- **mode_arr** – an array-like object defining the shape of mode space. If `inverse=False`, its shape is used to define the mode space size.

- **inverse** – `False` for forward (coordinate space -> mode space) transform, `True` for inverse (mode space -> coordinate space) transform.
- **axes** – a tuple with axes over which to perform the transform. If not given, the transform is performed over all the axes.
- **order** – if `F` is a function in mode space, the number of spatial points is chosen so that the transformation `DHT[(DHT^{-1}[F])^order]` could be performed.
- **add_points** – a list of the same length as `mode_arr` shape, specifying the number of points in x-space to use in addition to minimally required (`0` by default).

compiled_signature_forward(*modes:o*, *coords:i*)

compiled_signature_inverse(*coords:o*, *modes:i*)

Depending on `inverse` value, either of these two will be created.

Parameters

- **modes** – an array with the attributes of `mode_arr`.
- **coords** – an array with the shape depending on `mode_arr`, `axes`, `order` and `add_points`, and the dtype of `mode_arr`.

Counter-based random number generators

This module is based on the paper by Salmon et al., P. Int. C. High. Perform. 16 (2011). and the source code of Random123 library.

A counter-based random-number generator (CBRNG) is a parametrized function $f_k(c)$, where k is the key, c is the counter, and the function f_k defines a bijection in the set of integer numbers. Being applied to successive counters, the function produces a sequence of pseudo-random numbers. The key is an analogue of the seed of stateful RNGs; if the CBRNG is used to generate random num bers in parallel threads, the key is a combination of a seed and a unique thread number.

There are two types of generators available, `threefry` (uses large number of simple functions), and `philox` (uses smaller number of more complicated functions). The latter one is generally faster on GPUs; see the paper above for detailed comparisons. These generators can be further specialized to use `words=2` or `words=4` `bitness=32`-bit or `bitness=64`-bit counters. Obviously, the period of the generator equals to the cardinality of the set of possible counters. For example, if the counter consits of 4 64-bit numbers, then the period of the generator is 2^{256}. As for the key size, in case of `threefry` the key has the same size as the counter, and for `philox` the key is half its size.

The `CBRNG` class sets one of the words of the key (except for `philox-2x64`, where 32 bit of the only word in the key are used), the rest are the same for all threads and are derived from the provided `seed`. This limits the maximum number of number-generating threads (`size`). `philox-2x32` has a 32-bit key and therefore cannot be used in `CBRNG` (although it can be used separately with the help of the kernel API).

The `CBRNG` class itself is stateless, same as other computations in Reikna, so you have to manage the generator state yourself. The state is created by the `create_counters()` method and contains a `size` counters. This state is then passed to, and updated by a `CBRNG` object.

class `reikna.cbrng.`**CBRNG**(*randoms_arr*, *generators_dim*, *sampler*, *seed=None*)

Bases: `Computation`

Counter-based pseudo-random number generator class.

Parameters

- **randoms_arr** – an array intended for storing generated random numbers.

- **generators_dim** – the number of dimensions (counting from the end) which will use independent generators. For example, if `randoms_arr` has the shape `(100, 200, 300)` and `generators_dim` is `2`, then in every sub-array `(j, :, :)`, `j = 0 .. 99`, every element will use an independent generator.
- **sampler** – a `Sampler` object.
- **seed** – `None` for random seed, or an integer.

classmethod **sampler_name**(*randoms_arr*, *generators_dim*, *sampler_kwds=None*, *seed=None*)
A convenience constructor for the sampler `sampler_name` from `samplers`. The contents of the dictionary `sampler_kwds` will be passed to the sampler constructor function (with `bijection` being created automatically, and `dtype` taken from `randoms_arr`).

compiled_signature(*counters:io*, *randoms:o*)

Parameters

- **counters** – the RNG "state". All attributes are equal to the ones of the result of `create_counters()`.
- **randoms** – generated random numbers. All attributes are equal to the ones of `randoms_arr` from the constructor.

create_counters()
Create a counter array for use in `CBRNG`.

Kernel API

class `reikna.cbrng.bijections.`**Bijection**
Contains a CBRNG bijection module and accompanying metadata. Supports `__process_modules__` protocol.

word_dtype
The data type of the integer word used by the generator.

key_words
The number of words used by the key.

counter_words
The number of words used by the counter.

key_dtype
The `numpy.dtype` object representing a bijection key. Contains a single array field `v` with `key_words` of `word_dtype` elements.

counter_dtype
The `numpy.dtype` object representing a bijection counter. Contains a single array field `v` with `key_words` of `word_dtype` elements.

raw_functions
A dictionary `dtype:function_name` of available functions `function_name` in `module` that produce a random full-range integer `dtype` from a `State`, advancing it. Available functions: `get_raw_uint32()`, `get_raw_uint64()`.

module
The module containing the CBRNG function. It provides the C functions below.

COUNTER_WORDS
Contains the value of `counter_words`.

KEY_WORDS
Contains the value of `key_words`.

Word
Contains the type corresponding to `word_dtype`.

Key
Describes the bijection key. Alias for the structure generated from `key_dtype`.

Word v[KEY_WORDS]

Counter
Describes the bijection counter, or its output. Alias for the structure generated from `counter_dtype`.

Word v[COUNTER_WORDS]

Counter **make_counter_from_int**(int *x*)
Creates a counter object from an integer.

Counter **bijection**(Key *key*, Counter *counter*)
The main bijection function.

State
A structure containing the CBRNG state which is used by `samplers`.

State **make_state**(Key *key*, Counter *counter*)
Creates a new state object.

Counter **get_next_unused_counter**(State *state*)
Extracts a counter which has not been used in random sampling.

uint32
A type of unsigned 32-bit word, corresponds to `numpy.uint32`.

uint64
A type of unsigned 64-bit word, corresponds to `numpy.uint64`.

uint32 **get_raw_uint32**(State **state*)
Returns uniformly distributed unsigned 32-bit word and updates the state.

uint64 **get_raw_uint64**(State **state*)
Returns uniformly distributed unsigned 64-bit word and updates the state.

`reikna.cbrng.bijections.`**philox**(*bitness*, *counter_words*, *rounds=10*)
A CBRNG based on a low number of slow rounds (multiplications).

Parameters

- **bitness** – `32` or `64`, corresponds to the size of generated random integers.
- **counter_words** – `2` or `4`, number of integers generated in one go.
- **rounds** – `1` to `12`, the more rounds, the better randomness is achieved. Default values are big enough to qualify as PRNG.

Returns a `Bijection` object.

`reikna.cbrng.bijections.`**threefry**(*bitness*, *counter_words*, *rounds=20*)
A CBRNG based on a big number of fast rounds (bit rotations).

Parameters

- **bitness** – `32` or `64`, corresponds to the size of generated random integers.
- **counter_words** – `2` or `4`, number of integers generated in one go.

- **rounds** – `1` to `72`, the more rounds, the better randomness is achieved. Default values are big enough to qualify as PRNG.

Returns a `Bijection` object.

class `reikna.cbrng.samplers.`**Sampler**

Contains a random distribution sampler module and accompanying metadata. Supports `__process_modules__` protocol.

deterministic

If `True`, every sampled random number consumes the same amount of counters.

randoms_per_call

How many random numbers one call to `sample` creates.

dtype

The data type of one random value produced by the sampler.

module

The module containing the distribution sampling function. It provides the C functions below.

RANDOMS_PER_CALL

Contains the value of `randoms_per_call`.

Value

Contains the type corresponding to `dtype`.

Result

Describes the sampling result.

value v[RANDOMS_PER_CALL]

Result **sample**(State **state*)

Performs the sampling, updating the state.

`reikna.cbrng.samplers.`**gamma**(*bijection*, *dtype*, *shape=1*, *scale=1*)

Generates random numbers from the gamma distribution

$$P(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is `shape`, and θ is `scale`. Supported dtypes: `float(32/64)`. Returns a `Sampler` object.

`reikna.cbrng.samplers.`**normal_bm**(*bijection*, *dtype*, *mean=0*, *std=1*)

Generates normally distributed random numbers with the mean `mean` and the standard deviation `std` using Box-Muller transform. Supported dtypes: `float(32/64)`, `complex(64/128)`. Produces two random numbers per call for real types and one number for complex types. Returns a `Sampler` object.

Note: In case of a complex `dtype`, `std` refers to the standard deviation of the complex numbers (same as `numpy.std()` returns), not real and imaginary components (which will be normally distributed with the standard deviation `std / sqrt(2)`). Consequently, while `mean` is of type `dtype`, `std` must be real.

`reikna.cbrng.samplers.`**uniform_float**(*bijection*, *dtype*, *low=0*, *high=1*)

Generates uniformly distributed floating-points numbers in the interval `[low, high)`. Supported dtypes: `float(32/64)`. A fixed number of counters is used in each thread. Returns a `Sampler` object.

`reikna.cbrng.samplers.`**uniform_integer**(*bijection*, *dtype*, *low*, *high=None*)

Generates uniformly distributed integer numbers in the interval `[low, high)`. If `high` is `None`, the interval is `[0, low)`. Supported dtypes: any numpy integers. If the size of the interval is a power of 2, a fixed number of counters is used in each thread. Returns a `Sampler` object.

class reikna.cbrng.tools.**KeyGenerator**
Contains a key generator module and accompanying metadata. Supports __process_modules__ protocol.

module
A module with the key generator function:

Key **key_from_int**(int *idx*)
Generates and returns a key, suitable for the bijection which was given to the constructor.

classmethod create(*bijection*, *seed=None*, *reserve_id_space=True*)
Creates a generator.

Parameters

- **bijection** – a Bijection object.
- **seed** – an integer, or numpy array of 32-bit unsigned integers.
- **reserve_id_space** – if True, the last 32 bit of the key will be reserved for the thread identifier. As a result, the total size of the key should be 64 bit or more. If False, the thread identifier will be just added to the key, which will still result in different keys for different threads, with the danger that different seeds produce the same sequences.

reference(*idx*)
Reference function that returns the key given the thread identifier. Uses the same algorithm as the module.

2.5.6 Transformations

This module contains a number of pre-created transformations.

reikna.transformations.**add_const**(*arr_t*, *param*)
Returns an addition transformation with a fixed parameter (1 output, 1 input): output = input + param.

reikna.transformations.**add_param**(*arr_t*, *param_dtype*)
Returns an addition transformation with a dynamic parameter (1 output, 1 input, 1 scalar): output = input + param.

reikna.transformations.**broadcast_const**(*arr_t*, *val*)
Returns a transformation that broadcasts the given constant to the array output (1 output): output = val.

reikna.transformations.**broadcast_param**(*arr_t*)
Returns a transformation that broadcasts the free parameter to the array output (1 output, 1 param): output = param.

reikna.transformations.**combine_complex**(*output_arr_t*)
Returns a transformation that joins two real inputs into complex output (1 output, 2 inputs): output = real + 1j * imag.

reikna.transformations.**copy**(*arr_t*, *out_arr_t=None*)
Returns an identity transformation (1 output, 1 input): output = input. Output array type out_arr_t may have different strides, but must have the same shape and data type.

reikna.transformations.**ignore**(*arr_t*)
Returns a transformation that ignores the output it is attached to.

reikna.transformations.**mul_const**(*arr_t*, *param*)
Returns a scaling transformation with a fixed parameter (1 output, 1 input): output = input * param.

reikna.transformations.**mul_param**(*arr_t*, *param_dtype*)
Returns a scaling transformation with a dynamic parameter (1 output, 1 input, 1 scalar): output = input * param.

reikna.transformations.**norm_const**(*arr_t*, *order*)
Returns a transformation that calculates the `order`-norm (1 output, 1 input): `output = abs(input) ** order`.

reikna.transformations.**norm_param**(*arr_t*)
Returns a transformation that calculates the `order`-norm (1 output, 1 input, 1 param): `output = abs(input) ** order`.

reikna.transformations.**split_complex**(*input_arr_t*)
Returns a transformation that splits complex input into two real outputs (2 outputs, 1 input): `real = Re(input), imag = Im(input)`.

2.6 Release history

2.6.1 0.6.4 (29 Sep 2014)

- CHANGED: renamed `power_dtype` parameter to `exponent_dtype` (a more correct term) in `pow()`.
- FIXED: (PR by @ringw) exception caused by printing CUDA program object.
- FIXED: `pow()` (0, 0) now returns 1 as it should.
- ADDED: an example of `FFT` with a custom transformation.
- ADDED: a type check in the `FFT` constructor.
- ADDED: an explicit `output_dtype` parameter for `pow()`.
- ADDED: `Array` objects for each backend expose the attribute `thread`.

2.6.2 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.

2.6.3 0.6.2 (20 Feb 2014)

- 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.

2.6.4 0.6.1 (4 Feb 2014)

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

2.6.5 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.

2.6.6 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`.

2.6.7 0.5.1 (30 Nov 2013)

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

2.6.8 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.

2.6.9 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.

2.6.10 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.

2.6.11 0.3.5 (6 Aug 2013)

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

2.6.12 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.

2.6.13 0.3.3 (29 Jul 2013)

- FIXED: package metadata.

2.6.14 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.

2.6.15 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.

2.6.16 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

2.6.17 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.

2.6.18 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`.

2.6.19 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()`.

2.6.20 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).

2.6.21 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.

2.6.22 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

2.6.23 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