grunnur

Release 0.1.0

Bogdan Opanchuk

Mar 10, 2021

CONTENTS:

CHAPTER

ONE

MANUAL

Grunnur is an abstraction layer on top of PyCUDA/PyOpenCL. Its main purpose is to provide a uniform API for high-level GPGPU algorithms automating some common tasks.

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

```
import numpy
from grunnur import any_api, Context, Queue, Program, Array

N = 256

context = Context.from_devices([any_api.platforms[0].devices[0]])
queue = Queue.on_all_devices(context)

program = Program(
    context,
    """
    KERNEL void multiply_them(
        GLOBAL_MEM float *dest,
        GLOBAL_MEM float *a,
        GLOBAL_MEM float *b)
    {
        const SIZE_T i = get_global_id(0);
        dest[i] = a[i] * b[i];
    }
    """)

multiply_them = program.kernel.multiply_them

a = numpy.random.randn(N).astype(numpy.float32)
b = numpy.random.randn(N).astype(numpy.float32)
a_dev = Array.from_host(queue, a)
b_dev = Array.from_host(queue, b)
dest_dev = Array.empty(queue, a.shape, a.dtype)

multiply_them(queue, N, None, dest_dev, a_dev, b_dev)
print((dest_dev.get() - a * b == 0).all())
```

If you are familiar with `PyCUDA` or `PyOpenCL`, you will easily understand most of the the steps we have made here. The `any_api` object returns some API of the ones available (so, depending of whether `PyOpenCL` or `PyCUDA` are installed). More precise control over API is available via *API discovery functions*.

The abstraction from specific C interface of OpenCL or CUDA 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 *`Program`* constructor 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 `grunnur` 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
import grunnur.dtypes as dtypes
import grunnur.functions as functions
from grunnur import any_api, Context, Queue, Program, Array

context = Context.from_devices([any_api.platforms[0].devices[0]])
queue = Queue.on_all_devices(context)

N = 256
dtype = numpy.complex64

program = Program(
    context, """
    KERNEL void multiply_them(
        GLOBAL_MEM ${ctype} *dest,
        GLOBAL_MEM ${ctype} *a,
        GLOBAL_MEM ${ctype} *b)
    {
      const SIZE_T i = get_global_id(0);
      dest[i] = ${mul}(a[i], b[i]);
    }
    """,
    render_globals=dict(
        ctype=dtypes.ctype(dtype),
        mul=functions.mul(dtype, dtype)))

multiply_them = program.kernel.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 = Array.from_host(queue, a)
b_dev = Array.from_host(queue, b)
dest_dev = Array.empty(queue, a.shape, a.dtype)

multiply_them(queue, N, None, dest_dev, a_dev, b_dev)
print(norm(dest_dev.get() - a * b) / norm(a * b) <= 1e-6)
```

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 Grunnur can be found in *API reference*.

CHAPTER

TWO

TUTORIAL: MODULES AND SNIPPETS

Modules and snippets are important primitives in Grunnur. Even if you do not write modules yourself, you will most likely use operations from the `functions` module, which are essentially module factories (callables returning `Module` objects). Therefore it helps if you know how they work under the hood.

2.1 Snippets

Snippets are `Mako` template defs (essentially functions returning rendered text) with the associated dictionary of render globals. 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.from_callable(
    lambda varname: """
    ${varname} + ${num}
    """,
    render_globals=dict(num=1))
```

Now we can compile a template which uses this snippet:

```
program = Program(
    context,
    """
    KERNEL void test(GLOBAL_MEM int *arr)
    {
        const SIZE_T idx = get_global_id(0);
        int x = arr[idx];
        arr[idx] = ${add('x')};
    }
    """,
    render_globals=dict(add=add))
```

As a result, the code that gets compiled is

```
KERNEL void test(GLOBAL_MEM int *arr)
{
    const SIZE_T idx = get_global_id(0);
    int x = 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 `Program()` can be viewed as a snippet with an empty signature.

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.from_callable(
    lambda prefix, arg: """
    FUNCTION int ${prefix}(int x)
    {
        return x + ${num} + ${arg};
    }
    """,
    name="foobar",
    render_globals=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 = Program(
    context,
    """
    KERNEL void test(GLOBAL_MEM int *arr)
    {
        const SIZE_T idx = get_global_id(0);
        int x = arr[idx];
        arr[idx] = ${add(2)}(x);
    }
    """,
    render_globals=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_globals` 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_globals` is replaced by the newly assigned prefix.

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

```
FUNCTION int _mod_foobar_0_(int x)
{
    return x + 1 + 2;
}
```

and the `add` keyword in the `render_globals` gets its value changed to `_mod_foobar_0_`. Then the main code is rendered and appended to the previously renderd parts, giving

```
FUNCTION int _mod_foobar_0_(int x)
{
    return x + 1 + 2;
}

KERNEL void test(GLOBAL_MEM int *arr)
{
    const SIZE_T idx = get_global_id(0);
    int x = arr[idx];
    arr[idx] = _mod_foobar_0_(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_globals`, 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_globals` 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.

2.3 Other constructors

If the arguments are not know at compile time, you can use *`DefTemplate.from_string`* with a regular constructor:

```
argnames = ['varname']
add = Snippet(
    DefTemplate.from_string("_func", argnames, "${varname} + ${num}"),
    render_globals=dict(num=1))
```

Modules can be constructed in a similar way. The only difference is that the template must have at least one positional parameter which will receive the prefix value.

Alternatively, one can create a snippet with no parameters or a module with a single prefix parameter with a `from_string()` constructor:

```
add = Module.from_string("""
    FUNCTION int ${prefix}(int x)
    {
        return x + ${num};
    }
    """,
    render_globals=dict(num=1))
```

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))
```

2.5 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 parallel computation calling these functions and saving results to global memory. These 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_globals={
        base_rng -> Snippet(...)
        gamma -> Snippet(
    }                   Gamma,
)                       render_globals = {
                            uniform -> Snippet(...)
                            normal -> Snippet(
                        }                   Normal,
                    )                       render_globals = {
                                                uniform -> Snippet(...)
                                            }
                                        )
```

CHAPTER

THREE

PUBLIC API

3.1 API discovery

In many applications it would be enough to use dynamic module attributes to get an *API* object:

```
from grunnur import cuda_api
from grunnur import opencl_api
from grunnur import any_api
```

For a finer programmatic control one can use the methods of the *API* class:

class grunnur.**API**

A generalized GPGPU API.

classmethod all_available()

Returns a list of *API* objects for which backends are available.

Return type List[API]

classmethod all_by_shortcut(*shortcut=None*)

If shortcut is a string, returns a list of one *API* object whose *id* attribute has its *shortcut* attribute equal to it (or raises an error if it was not found, or its backend is not available).

If shortcut is None, returns a list of all available *API* objects.

Parameters **shortcut** (Optional[str]) – an API shortcut to match.

Return type List[API]

classmethod from_api_id(*api_id*)

Creates an *API* object out of an identifier.

Parameters **api_id** (*APIID*) – API identifier.

Return type API

id: ***grunnur.adapter_base.APIID***

This API's ID.

property platforms

A list of this API's *Platform* objects.

shortcut: **str**

A shortcut for this API (to use in *all_by_shortcut()*, usually coming from some kind of a CLI). Equal to id.shortcut.

class grunnur.adapter_base.**APIID**

An ID of an *API* object.

shortcut: str
This API's shortcut.

`grunnur.`**`cuda_api_id`**`()`
Returns the identifier of CUDA API.

Return type *APIID*

`grunnur.`**`opencl_api_id`**`()`
Returns the identifier of OpenCL API.

Return type *APIID*

`grunnur.`**`all_api_ids`**`()`
Returns a list of identifiers for all APIs available.

Return type `List[`*APIID*`]`

3.2 Platforms

A platform is an OpenCL term, but we use it for CUDA API as well for the sake of uniformity. Naturally, there will always be a single (dummy) platform in CUDA.

class `grunnur.`**`Platform`**`(`*platform_adapter*`)`
A generalized GPGPU platform.

classmethod all(*api*)
Returns a list of platforms available for the given API.

Parameters **api** (`API`) – the API to search in.

Return type `List[Platform]`

classmethod all_by_masks(*api*, *include_masks=None*, *exclude_masks=None*)
Returns a list of all platforms with names satisfying the given criteria.

Parameters

- **api** (`API`) – the API to search in.
- **include_masks** (`Optional[Sequence[str]]`) – a list of strings (treated as regexes), one of which must match with the platform name.
- **exclude_masks** (`Optional[Sequence[str]]`) – a list of strings (treated as regexes), neither of which must match with the platform name.

Return type `List[Platform]`

classmethod from_backend_platform(*obj*)
Wraps a backend platform object into a Grunnur platform object.

Return type `Platform`

classmethod from_index(*api*, *platform_idx*)
Creates a platform based on its index in the list returned by the API.

Parameters

- **api** (`API`) – the API to search in.
- **platform_idx** (`int`) – the target platform's index.

Return type `Platform`

api: grunnur.api.API
The *API* object this platform belongs to.

property devices
A list of this device's *Device* objects.

name: str
The platform's name.

vendor: str
The platform's vendor.

version: str
The platform's version.

3.3 Devices

class grunnur.**Device**
A generalized GPGPU device.

classmethod all(*platform*)
Returns a list of devices available for the given platform.

Parameters **platform** (Platform) – the platform to search in.

Return type List[Device]

classmethod all_by_masks(*platform*, *include_masks=None*, *exclude_masks=None*, *unique_only=False*, *include_pure_parallel_devices=False*)
Returns a list of all devices satisfying the given criteria.

Parameters

- **platform** (Platform) – the platform to search in.
- **include_masks** (Optional[Sequence[str]]) – a list of strings (treated as regexes), one of which must match with the device name.
- **exclude_masks** (Optional[Sequence[str]]) – a list of strings (treated as regexes), neither of which must match with the device name.
- **unique_only** (bool) – if True, only return devices with unique names.
- **include_pure_parallel_devices** (bool) – if True, include devices with *params.max_total_local_size* equal to 1.

Return type List[Device]

classmethod from_backend_device(*obj*)
Wraps a backend device object into a Grunnur device object.

Return type Device

classmethod from_index(*platform*, *device_idx*)
Creates a device based on its index in the list returned by the API.

Parameters

- **platform** (Platform) – the API to search in.
- **device_idx** (int) – the target device's index.

Return type Device

name: str
This device's name.

property params
Returns a *DeviceParameters* object associated with this device.

Return type *DeviceParameters*

platform: grunnur.platform.Platform
The *Platform* object this device belongs to.

class grunnur.adapter_base.**DeviceParameters**
An object containing device's specifications.

abstract property compute_units
The number of multiprocessors (CUDA)/compute units (OpenCL) for the device.

Return type int

abstract property local_mem_banks
The number of independent channels for shared (CUDA)/local (OpenCL) memory, which can be used from one warp without request serialization.

Return type int

abstract property local_mem_size
The size of shared (CUDA)/local (OpenCL) memory (in bytes).

Return type int

abstract property max_local_sizes
The maximum number of threads in one block (CUDA), or work items in one work group (OpenCL) for each of the available dimensions.

Return type Tuple[int, ...]

abstract property max_num_groups
The maximum number of blocks (CUDA)/work groups (OpenCL) for each of the available dimensions.

Return type Tuple[int, ...]

abstract property max_total_local_size
The maximum total number of threads in one block (CUDA), or work items in one work group (OpenCL).

Return type int

abstract property type
Device type.

Return type *DeviceType*

abstract property warp_size
The number of threads (CUDA)/work items (OpenCL) that are executed synchronously (within one multiprocessor/compute unit).

Return type int

class grunnur.adapter_base.**DeviceType**
An enum representing a device's type.

CPU = 1
CPU type

GPU = 2
GPU type

3.4 Device discovery

`grunnur.platforms_and_devices_by_mask(api, quantity=1, platform_include_masks=None, platform_exclude_masks=None, device_include_masks=None, device_exclude_masks=None, unique_devices_only=False, include_pure_parallel_devices=False)`

Returns all tuples (platform, list of devices) where the platform name and device names satisfy the given criteria, and there are at least `quantity` devices in the list.

Parameters

- **quantity** (`Optional[int]`) – the number of devices to find. If `None`, find all matching devices belonging to a single platform.
- **platform_include_masks** (`Optional[Sequence[str]]`) – passed to `Platform.all_by_masks()`.
- **platform_exclude_masks** (`Optional[Sequence[str]]`) – passed to `Platform.all_by_masks()`.
- **device_include_masks** (`Optional[Sequence[str]]`) – passed to `Device.all_by_masks()`.
- **device_exclude_masks** (`Optional[Sequence[str]]`) – passed to `Device.all_by_masks()`.
- **unique_devices_only** (`bool`) – passed to `Device.all_by_masks()`.
- **include_pure_parallel_devices** (`bool`) – passed to `Device.all_by_masks()`.

Return type `List[Tuple[Platform, List[Device]]]`

`grunnur.select_devices(api, interactive=False, quantity=1, **device_filters)`

Using the results from `platforms_and_devices_by_mask()`, either lets the user select the devices (from the ones matching the criteria) interactively, or takes the first matching list of `quantity` devices.

Parameters

- **interactive** (`bool`) – if `True`, shows a dialog to select the devices. If `False`, selects the first matching ones.
- **quantity** (`Optional[int]`) – passed to `platforms_and_devices_by_mask()`.
- **device_filters** – passed to `platforms_and_devices_by_mask()`.

Return type `List[Device]`

3.5 Contexts

class `grunnur.Context`

GPGPU context.

deactivate()

CUDA API only: deactivates this context, popping all the CUDA context objects from the stack.

classmethod from_backend_contexts(*backend_contexts*, *take_ownership=False*)
Creates a context from a single or several backend device contexts. If `take_ownership` is `True`, this object will be responsible for the lifetime of backend context objects (important for CUDA backend).

Return type `Context`

classmethod from_backend_devices(*backend_devices*)
Creates a context from a single or several backend device objects.

Return type `Context`

classmethod from_criteria(*api*, *interactive=False*, *devices_num=1*, ***device_filters*)
Finds devices matching the given criteria and creates a `Context` object out of them.

Parameters

- **interactive** (`bool`) – passed to `select_devices()`.
- **devices_num** (`Optional[int]`) – passed to `select_devices()` as `quantity`.
- **device_filters** – passed to `select_devices()`.

Return type `Context`

classmethod from_devices(*devices*)
Creates a context from a device or an iterable of devices.

Parameters **devices** (`Union[Device, Iterable[Device]]`) – one or several devices to use.

Return type `Context`

api: grunnur.api.API
The API this context is based on.

devices: Tuple[grunnur.device.Device]
Devices in this context.

platform: grunnur.platform.Platform
The platform this context is based on.

3.6 Queues

class grunnur.**Queue**(*context*, *queue_adapter*, *device_idxs*)
A queue on multiple devices.

classmethod on_device_idxs(*context*, *device_idxs*)
Creates a queue from provided device indexes (in the context).

Parameters

- **context** (`Context`) – the context to create a queue in.
- **device_idxs** (`Iterable[int]`) – the indices of devices (in the context) to use.

Return type `Queue`

synchronize()
Blocks until sub-queues on all devices are empty.

context: grunnur.context.Context
This queue's context.

device_idxs: Tuple[int, ...]
Device indices (in the context) this queue operates on.

3.7 Buffers and arrays

class grunnur.**Buffer**(*context*, *buffer_adapter*)
A memory buffer on device.

classmethod allocate(*context*, *size*)
Allocate a buffer of `size` bytes.

Parameters

- **context** (`Context`) – the context to use.
- **size** (`int`) – the buffer's size in bytes.

Return type `Buffer`

get(*queue*, *host_array*, *async_=False*)
Copy the contents of the buffer to the host array.

Parameters

- **queue** (`Queue`) – the queue to use for the transfer.
- **host_array** (`ndarray`) – the destination array.
- **async** – if *True*, the transfer is performed asynchronously.

get_sub_region(*origin*, *size*)
Return a buffer object describing a subregion of this buffer.

Parameters

- **origin** (`int`) – the offset of the subregion.
- **size** (`int`) – the size of the subregion.

Return type `Buffer`

set(*queue*, *buf*, *no_async=False*)
Copy the contents of the host array or another buffer to this buffer.

Parameters

- **queue** (`Queue`) – the queue to use for the transfer.
- **buf** (`Union[ndarray, Buffer]`) – the source - `numpy` array or a `Buffer` object.
- **no_async** (`bool`) – if *True*, the transfer blocks until completion.

property offset
Offset of this buffer (in bytes) from the beginning of the physical allocation it resides in.

Return type `int`

property size
This buffer's size (in bytes).

Return type `int`

class grunnur.**Array**(*queue*, *array_metadata*, *data=None*, *allocator=None*)
Array on the device.

classmethod empty(*queue*, *shape*, *dtype*, *allocator=None*)
Creates an empty array.

Parameters

- **queue** (`Queue`) – the queue to use for the transfer.
- **shape** (`Sequence[int]`) – array shape.
- **dtype** (`dtype`) – array data type.
- **allocator** (`Optional[Callable[[int], Buffer]]`) – an optional callable returning a *Buffer* object.

Return type `Array`

classmethod from_host(*queue*, *host_arr*)
Creates an array object from a host array.

Parameters

- **queue** (`Queue`) – the queue to use for the transfer.
- **host_arr** (`ndarray`) – the source array.

Return type `Array`

get(*dest=None*, *async_=False*)
Copy the contents of the array to the host array and return it.

Parameters

- **dest** (`Optional[ndarray]`) – the destination array. If `None`, the target array is created.
- **async** – if *True*, the transfer is performed asynchronously.

Return type `ndarray`

set(*array*, *no_async=False*)
Copy the contents of the host array to the array.

Parameters

- **array** (`Union[ndarray, Array]`) – the source array.
- **no_async** (`bool`) – if *True*, the transfer blocks until completion.

single_device_view(*device_idx*)
Returns a subscriptable object that produces sub-arrays based on the device `device_idx`.

Return type *SingleDeviceFactory*

dtype: numpy.dtype
Array item data type.

shape: Tuple[int, ...]
Array shape.

strides: Tuple[int, ...]
Array strides.

class grunnur.array.**SingleDeviceFactory**
A subscriptable object that produces sub-arrays based on a single device.

__getitem__(*slices*)

Return a view of the parent array bound to the device this factory was created for (see `single_device_view()`).

Return type `Array`

3.8 Programs and kernels

class grunnur.**Program**(*context*, *template_src*, *device_idxs=None*, *no_prelude=False*, *fast_math=False*, *render_args=[]*, *render_globals={}*, *compiler_options=[]*, *keep=False*, *constant_arrays={}*)

A compiled program on device(s).

Parameters

- **context** (`Context`) – context to compile the program on.
- **template_src** (`Union[str, Callable[..., str], DefTemplate, Snippet]`) – a string with the source code, or a Mako template source to render.
- **device_idxs** (`Optional[Sequence[int]]`) – a list of device numbers to compile on. If `None`, compile on all context's devices.
- **no_prelude** (`bool`) – do not add prelude to the rendered source.
- **fast_math** (`bool`) – compile using fast (but less accurate) math functions.
- **render_args** (`Union[List, Tuple]`) – a list of positional args to pass to the template.
- **render_globals** (`Dict`) – a dictionary of globals to pass to the template.
- **compiler_options** (`Iterable[str]`) – a list of options to pass to the backend compiler.
- **keep** (`bool`) – keep the intermediate files in a temporary directory.
- **constant_arrays** (`Mapping[str, Tuple[int, dtype]]`) – (**CUDA only**) a dictionary `name: (size, dtype)` of global constant arrays to be declared in the program.

set_constant_array(*queue*, *name*, *arr*)

Uploads a constant array to the context's devices (**CUDA only**).

Parameters

- **queue** (`Queue`) – the queue to use for the transfer.
- **name** (`str`) – the name of the constant array symbol in the code.
- **arr** (`Union[Array, ndarray]`) – either a device or a host array.

context: grunnur.context.Context

The context this program was compiled for.

kernel: *grunnur.program.KernelHub*

An object whose attributes are `Kernel` objects with the corresponding names.

sources: Dict[int, str]

Source files used for each device.

class grunnur.program.**KernelHub**(*program*)

An object providing access to the host program's kernels.

__getattr__(*kernel_name*)
Returns a `Kernel` object for a function (CUDA)/kernel (OpenCL) with the name `kernel_name`.

Return type `Kernel`

class grunnur.program.**Kernel**(*program*, *sd_kernel_adapters*)
A kernel compiled for multiple devices.

__call__(*queue*, *global_size*, *local_size*, **args*, ***kwds*)
A shortcut for `Kernel.prepare()` and subsequent `PreparedKernel.__call__()`. See their doc entries for details.

prepare(*queue*, *global_size*, *local_size*)
Prepares the kernel for execution.

Parameters

- **queue** (`Queue`) – the multi-device queue to use.
- **global_size** (`Union[int, Sequence[int], MultiDevice[Union[int, Sequence[int]]]]`) – the total number of threads (CUDA)/work items (OpenCL) in each dimension (column-major). Note that there may be a maximum size in each dimension as well as the maximum number of dimensions. See `DeviceParameters` for details.
- **local_size** (`Union[int, Sequence[int], MultiDevice[Union[int, Sequence[int]]], None]`) – the number of threads in a block (CUDA)/work items in a work group (OpenCL) in each dimension (column-major). If `None`, it will be chosen automatically.

Return type `PreparedKernel`

property max_total_local_sizes
The maximum possible number of threads in a block (CUDA)/work items in a work group (OpenCL) for this kernel.

Return type `Dict[int, int]`

class grunnur.program.**PreparedKernel**(*sd_kernel_adapters*, *queue*, *device_idxs*, *global_sizes*, *local_sizes*, *hold_reference=None*)
A kernel specialized for execution on a given queue with all possible preparations and checks performed.

__call__(**args*, *device_idxs=None*, ***kwds*)
Enqueues the kernel on the chosen devices.

Parameters

- **args** – kernel arguments. Can be: `Array` objects, `Buffer` objects, `numpy` scalars.
- **device_idxs** (`Optional[Iterable[int]]`) – the devices to enqueue the kernel on (*in the context, not in the queue*). Must be a subset of the devices of the `queue` used for preparation. If `None`, all the `queue`'s devices are used. Note that the used devices must be among the ones the parent `Program` was compiled for.
- **kwds** – backend-specific keyword parameters.

Returns a list of `Event` objects for enqueued kernels in case of PyOpenCL.

class grunnur.**MultiDevice**(**args*)
A wrapper for a sequence of arguments where each should be passed to a separate device.

3.9 Static kernels

class grunnur.**StaticKernel**(*queue*, *template_src*, *name*, *global_size*, *local_size=None*, *device_idxs=None*, *render_globals={}*, *constant_arrays={}*, ***kwds*)

An object containing a GPU kernel with fixed call sizes.

The globals for the source template will contain an object with the name `static` of the type `VsizeModules` containing the id/size functions to be used instead of regular ones.

Parameters

- **context** – context to compile the kernel on.
- **template_src** (`Union[str, Callable[..., str], DefTemplate, Snippet]`) – a string with the source code, or a Mako template source to render.
- **name** (`str`) – the kernel's name.
- **global_size** (`Union[int, Sequence[int]]`) – the total number of threads (CUDA)/work items (OpenCL) in each dimension (column-major). Note that there may be a maximum size in each dimension as well as the maximum number of dimensions. See `DeviceParameters` for details.
- **local_size** (`Union[int, Sequence[int], None]`) – the number of threads in a block (CUDA)/work items in a work group (OpenCL) in each dimension (column-major). If `None`, it will be chosen automatically.
- **device_idxs** (`Optional[Sequence[int]]`) – a list of device numbers to compile on. If `None`, compile on all context's devices.
- **render_globals** (`Dict`) – a dictionary of globals to pass to the template.
- **constant_arrays** (`Mapping[str, Tuple[int, dtype]]`) – (**CUDA only**) a dictionary `name: (size, dtype)` of global constant arrays to be declared in the program.

__call__(**args*)

Execute the kernel. In case of the OpenCL backend, returns a `pyopencl.Event` object.

Parameters

- **queue** – the multi-device queue to use.
- **args** – kernel arguments. Can be: `Array` objects, `Buffer` objects, `numpy` scalars.

set_constant_array(*queue*, *name*, *arr*)

Uploads a constant array to the context's devices (**CUDA only**).

Parameters

- **queue** (`Queue`) – the queue to use for the transfer.
- **name** (`str`) – the name of the constant array symbol in the code.
- **arr** (`Union[Array, ndarray]`) – either a device or a host array.

queue: grunnur.queue.Queue

The queue this static kernel was compiled and prepared for.

sources: Dict[int, str]

Source files used for each device.

class grunnur.vsize.**VsizeModules**(*local_id*, *local_size*, *group_id*, *num_groups*, *global_id*, *global_size*, *global_flat_id*, *global_flat_size*, *begin*)

A collection of modules passed to `grunnur.StaticKernel`. Should be used instead of regular group/thread id functions.

begin: grunnur.modules.Module
Provides the statement `${begin}` that should be used at the start of a static kernel function.

global_flat_id: grunnur.modules.Module
Provides the function `VSIZE_T ${global_flat_id}()` returning the global id of the current thread with all dimensions flattened.

global_flat_size: grunnur.modules.Module
Provides the function `VSIZE_T ${global_flat_size}()`. returning the global size of with all dimensions flattened.

global_id: grunnur.modules.Module
Provides the function `VSIZE_T ${global_id}(int dim)` returning the global id of the current thread.

global_size: grunnur.modules.Module
Provides the function `VSIZE_T ${global_size}(int dim)` returning the global size along dimension `dim`.

group_id: grunnur.modules.Module
Provides the function `VSIZE_T ${group_id}(int dim)` returning the group id of the current thread.

local_id: grunnur.modules.Module
Provides the function `VSIZE_T ${local_id}(int dim)` returning the local id of the current thread.

local_size: grunnur.modules.Module
Provides the function `VSIZE_T ${local_size}(int dim)` returning the size of the current group.

num_groups: grunnur.modules.Module
Provides the function `VSIZE_T ${num_groups}(int dim)` returning the number of groups in dimension `dim`.

3.10 Utilities

class grunnur.**Template**(*mako_template*)
A wrapper for mako `Template` objects.

classmethod from_associated_file(*filename*)
Returns a *Template* object created from the file which has the same name as `filename` and the extension `.mako`. Typically used in computation modules as `Template.from_associated_file(__file__)`.

Return type `Template`

classmethod from_string(*template_source*)
Returns a *Template* object created from source.

get_def(*name*)
Returns the template def with the name `name`.

Return type `DefTemplate`

class grunnur.**DefTemplate**(*name*, *mako_def_template*, *source*)
A wrapper for Mako `DefTemplate` objects.

classmethod from_callable(*name*, *callable_obj*)
Creates a template def from a callable returning a string. The parameter list of the callable is used to create the pararameter list of the resulting template def; the callable should return the body of a Mako template def regardless of the arguments it receives.

Return type `DefTemplate`

classmethod from_string(*name*, *argnames*, *source*)
Creates a template def from a string with its body and a list of argument names.

Return type `DefTemplate`

render(**args*, ***globals_*)
Renders the template def with given arguments and globals.

Return type `str`

class grunnur.**RenderError**(*exception*, *args*, *globals_*, *source*)
A custom wrapper for Mako template render errors, to facilitate debugging.

exception: Exception
The original exception thrown by Mako's *render()*.

globals: dict
The globals used to render the template.

source: str
The source of the template.

class grunnur.**Snippet**(*template*, *render_globals={}*)
Contains a source snippet - a template function that will be rendered in place, with possible context that can include other `Snippet` or `Module` objects.

Creates a snippet out of a prepared template.

Parameters

- **template** (`DefTemplate`) –
- **render_globals** (`Mapping`) –

classmethod from_callable(*callable_obj*, *name='_snippet'*, *render_globals={}*)
Creates a snippet from a callable returning a string. The parameter list of the callable is used to create the pararameter list of the resulting template def; the callable should return the body of a Mako template def regardless of the arguments it receives.

Parameters

- **callable_obj** (`Callable[..., str]`) – a callable returning the template source.
- **name** (`str`) – the snippet's name (will simplify debugging)
- **render_globals** (`Mapping`) – a dictionary of "globals" to be used when rendering the template.

Return type `Snippet`

classmethod from_string(*source*, *name='_snippet'*, *render_globals={}*)
Creates a snippet from a template source, treated as a body of a template def with no arguments.

Parameters

- **source** (`str`) – a string with the template source.
- **name** (`str`) – the snippet's name (will simplify debugging)
- **render_globals** (`Mapping`) – a dictionary of "globals" to be used when rendering the template.

Return type `Snippet`

class grunnur.**Module**(*template*, *render_globals={}*)

Contains a source module - a template function that will be rendered at root level, and the place where it was called will receive its unique identifier (prefix), which is used to prefix all module's functions, types and macros in the global namespace.

Creates a module out of a prepared template.

Parameters

- **template** (DefTemplate) –
- **render_globals** (Mapping) –

classmethod from_callable(*callable_obj*, *name='_module'*, *render_globals={}*)

Creates a module from a callable returning a string. The parameter list of the callable is used to create the pararameter list of the resulting template def; the callable should return the body of a Mako template def regardless of the arguments it receives.

The prefix will be passed as the first argument to the template def on render.

Parameters

- **callable_obj** (Callable[..., str]) – a callable returning the template source.
- **name** (str) – the module's name (will simplify debugging)
- **render_globals** (Mapping) – a dictionary of "globals" to be used when rendering the template.

Return type Module

classmethod from_string(*source*, *name='_module'*, *render_globals={}*)

Creates a module from a template source, treated as a body of a template def with a single argument (prefix).

Parameters

- **source** (str) – a string with the template source.
- **name** (str) – the module's name (will simplify debugging)
- **render_globals** (Mapping) – a dictionary of "globals" to be used when rendering the template.

Return type Module

3.11 Data type utilities

3.11.1 C interop

grunnur.dtypes.**ctype**(*dtype*)

Returns an object that can be passed as a global to *Program()* and used to render a C equivalent of the given numpy dtype. If there is a built-in C equivalent, the object is just a string with the type name; otherwise it is a *Module* object containing the corresponding struct declaration.

Note: If dtype is a struct type, it needs to be aligned (see *ctype_struct()* and *align()*).

Parameters **dtype** (dtype) –

Return type `Union[str, Module]`

`grunnur.dtypes.`**`ctype_struct`**(*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).

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.

Parameters

- **`dtype`** (`Union[Type, dtype]`) –
- **`ignore_alignment`** (`bool`) –

> **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_struct` will make some additional checks and raise `ValueError` if it is not the case.

Return type `Module`

`grunnur.dtypes.`**`complex_ctr`**(*dtype*)

Returns name of the constructor for the given `dtype`.

Parameters **`dtype`** (`dtype`) –

Return type `str`

`grunnur.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.

Parameters

- **`val`** –
- **`dtype`** (`Optional[dtype]`) –

Return type `str`

`grunnur.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.

Parameters **`dtype`** (`dtype`) –

Return type `dtype`

3.11.2 Struct helpers

`grunnur.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()`*.

Parameters **path** (`List[Union[str, int]]`) –

Return type `str`

`grunnur.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.

Parameters **dtype** (`dtype`) –

Return type `List[Tuple[List[Union[str, int]], dtype]]`

`grunnur.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()`*.

Parameters

- **arr** (`ndarray`) –
- **path** (`List[Union[str, int]]`) –

Return type `ndarray`

3.11.3 Data type checks and conversions

`grunnur.dtypes.normalize_type(dtype)`

Numpy's dtype shortcuts (e.g. `numpy.int32`) are `type` objects and have slightly different properties from actual `numpy.dtype` objects. This function converts the former to `numpy.dtype` and keeps the latter unchanged.

Parameters **dtype** (`Union[Type, dtype]`) –

Return type `dtype`

`grunnur.dtypes.is_complex(dtype)`

Returns `True` if `dtype` is complex.

Parameters **dtype** (`dtype`) –

Return type `bool`

`grunnur.dtypes.is_double(dtype)`

Returns `True` if `dtype` is double precision floating point.

Parameters **dtype** (`dtype`) –

Return type `bool`

`grunnur.dtypes.is_integer(dtype)`

Returns `True` if `dtype` is an integer.

Parameters **dtype** (`dtype`) –

Return type `bool`

`grunnur.dtypes.is_real(dtype)`

Returns `True` if `dtype` is a real number (but not complex).

Parameters **dtype** (dtype) –

Return type bool

grunnur.dtypes.**result_type**(**dtypes*)

Wrapper for numpy.result_type() which takes into account types supported by GPUs.

Parameters **dtypes** (dtype) –

Return type dtype

grunnur.dtypes.**min_scalar_type**(*val*, *force_signed=False*)

Wrapper for numpy.min_scalar_dtype() which takes into account types supported by GPUs.

If force_signed is True, a signed type will be returned even if val is positive.

Return type dtype

grunnur.dtypes.**detect_type**(*val*)

Returns the data type of val.

Return type dtype

grunnur.dtypes.**complex_for**(*dtype*)

Returns complex dtype corresponding to given floating point dtype.

Parameters **dtype** (dtype) –

Return type dtype

grunnur.dtypes.**real_for**(*dtype*)

Returns floating point dtype corresponding to given complex dtype.

Parameters **dtype** (dtype) –

Return type dtype

grunnur.dtypes.**cast**(*dtype*)

Returns function that takes one argument and casts it to dtype.

Parameters **dtype** (dtype) –

Return type Callable[[Any], Any]

3.12 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.

grunnur.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 complex numbers are based on 2-vectors, and therefore 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)).

Parameters

- **in_dtypes** (dtype) –
- **out_dtype** (Optional[dtype]) –

Return type Module

`grunnur.functions.cast(in_dtype, out_dtype)`

Returns a `Module` with a function of one argument that casts values of `in_dtype` to `out_dtype`.

Parameters

- **in_dtype** (dtype) –
- **out_dtype** (dtype) –

Return type `Module`

`grunnur.functions.conj(dtype)`

Returns a `Module` with a function of one argument that conjugates the value of type `dtype` (if it is not a complex data type, the value will not be modified).

Parameters **dtype** (dtype) –

Return type `Module`

`grunnur.functions.div(dividend_dtype, divisor_dtype, out_dtype=None)`

Returns a `Module` with a function of two arguments that divides a value of type `dividend_dtype` by a value of type `divisor_dtype`. If `out_dtype` is given, it will be set as a return type for this function.

Parameters

- **dividend_dtype** (dtype) –
- **divisor_dtype** (dtype) –
- **out_dtype** (Optional[dtype]) –

Return type `Module`

`grunnur.functions.exp(dtype)`

Returns a `Module` with a function of one argument that exponentiates the value of type `dtype` (must be a real or a complex data type).

Parameters **dtype** (dtype) –

Return type `Module`

`grunnur.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.

Parameters

- **in_dtypes** (dtype) –
- **out_dtype** (Optional[dtype]) –

Return type `Module`

`grunnur.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).

Parameters **dtype** (dtype) –

Return type `Module`

`grunnur.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).

Parameters **dtype** (dtype) –

Return type `Module`

`grunnur.functions.`**`polar_unit`**(*dtype*)

Returns a `Module` with a function of one argument that returns a complex number `exp(i * theta) == (cos(theta), sin(theta))` for a value `theta` of type `dtype` (must be a real data type).

Parameters **`dtype`** (`dtype`) –

Return type `Module`

`grunnur.functions.`**`pow`**(*base_dtype*, *exponent_dtype=None*, *out_dtype=None*)

Returns a `Module` with a function of two arguments that raises the first argument of type `base_dtype` to the power of the second argument of type `exponent_dtype` (an integer or real data type).

If `exponent_dtype` or `out_dtype` are not given, they default to `base_dtype`. If `base_dtype` is not the same as `out_dtype`, the input is cast to `out_dtype` *before* exponentiation. If `exponent_dtype` is real, but both `base_dtype` and `out_dtype` are integer, a `ValueError` is raised.

Parameters

- **`base_dtype`** (`dtype`) –
- **`exponent_dtype`** (`Optional[dtype]`) –
- **`out_dtype`** (`Optional[dtype]`) –

Return type `Module`

3.13 Virtual buffers

Often one needs temporary buffers that are only used in one place in the code, but used many times. Allocating them each time they are used may involve too much overhead; allocating real buffers and storing them increases the program's memory requirements. A possible middle ground is using virtual allocations, where several of them can use the samy physical allocation. The virtual allocation manager will make sure that two virtual buffers that are used simultaneously (as declared by the user) will not share the same physical space.

class `grunnur.virtual_alloc.`**`VirtualManager`**(*queue*, *pack_on_alloc=False*, *pack_on_free=False*)

Base class for a manager of virtual allocations.

Parameters

- **`queue`** (`Queue`) – an instance of `Queue`.
- **`pack_on_alloc`** (`bool`) – whether to repack allocations when a new allocation is requested.
- **`pack_on_free`** (`bool`) – whether to repack allocations when an allocation is freed.

Note: The allocators returned by this object must be used for arrays attached to the same queue.

`allocator`(*dependencies=None*)

Create a callable to use for `Array` creation.

Parameters **`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 `__virtual_allocations__` which is a valid value (the last two will be processed recursively).

Return type `VirtualAllocator`

pack()
Packs the real allocations possibly reducing total memory usage. This process can be slow and may synchronize the base queue.

statistics()
Returns allocation statistics.

Return type `VirtualAllocationStatistics`

class grunnur.virtual_alloc.**TrivialManager**(**args*, ***kwds*)
Trivial manager — allocates a separate buffer for each allocation request.

class grunnur.virtual_alloc.**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.

class grunnur.virtual_alloc.**VirtualAllocator**(*manager*, *dependencies*)
A helper callable object to use as an allocator for `Array` creation. Encapsulates the dependencies (as identifiers, doesn't hold references for actual objects).

class grunnur.virtual_alloc.**VirtualAllocationStatistics**(*real_buffers*, *virtual_buffers*)
Virtual allocation details.

real_num: int
The number of physical allocations.

real_size_total: int
The total size of physical allocations (in bytes).

real_sizes: Dict[int, int]
A dictionary `size: count` with the counts for physical allocations of each size.

virtual_num: int
The number of virtual allocations.

virtual_size_total: int
The total size of virtual allocations (in bytes).

virtual_sizes: Dict[int, int]
A dictionary `size: count` with the counts for virtual allocations of each size.

3.14 Kernel toolbox

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

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

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

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

LOCAL_BARRIER
Synchronizes threads inside a block.

FUNCTION
Modifier for a device-only function declaration.

KERNEL
Modifier for a kernel function declaration.

GLOBAL_MEM
Modifier for a global memory pointer argument.

LOCAL_MEM_DECL
Modifier for a statically allocated local memory variable.

LOCAL_MEM_DYNAMIC
Modifier for a dynamically allocated local memory variable (CUDA only).

LOCAL_MEM
Modifier for a local memory argument in device-only functions.

CONSTANT_MEM_DECL
Modifier for a statically allocated constant memory variable.

CONSTANT_MEM
Modifier for a constant memory argument in 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**(unsigned int *dim*)

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

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

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

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

SIZE_T **get_global_size**(unsigned 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;
```

CHAPTER

FOUR

VERSION HISTORY

Some stuff.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

V

W

Z