# grunnur

***Release 0.3.1.dev0+g30173b4.d20230129***

**Bogdan Opanchuk**

**Jan 29, 2023**

# CONTENTS

Grunnur is a thin layer on top of PyCUDA and PyOpenCL that makes it easier to write platform-agnostic programs. It is a reworked `cluda` submodule of Reikna, extracted into a separate module.

CHAPTER

ONE

# MAIN FEATURES

- For the majority of cases, allows one to write platform-independent code.
- Simple usage of multiple GPUs (in particular, no need to worry about context switching for CUDA).
- A way to split kernel code into modules with dependencies between them (see *Module* and *Snippet*).
- Various mathematical functions (with complex numbers support) organized as modules.
- Static kernels, where you can use global/local shapes with any kinds of dimensions without worrying about assembling array indices from `blockIdx` and `gridIdx`.
- A temporary buffer manager that can pack several virtual buffers into the same physical one depending on the declared dependencies between them.

CHAPTER

TWO

# WHERE TO GET HELP

Please file issues in the the issue tracker.

Discussions and questions are handled by Github's discussion board.

CHAPTER

# THREE

# TABLE OF CONTENTS

## 3.1 Introduction

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(context.device)

program = Program(
    [context.device],
    """
    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(context.device, a.shape, a.dtype)

multiply_them(queue, [N], None, dest_dev, a_dev, b_dev)
print((dest_dev.get(queue) - 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(context.device)

N = 256
dtype = numpy.complex64

program = Program(
    [context.device],
    """
    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(context.device, a.shape, a.dtype)

multiply_them(queue, [N], None, dest_dev, a_dev, b_dev)
print(norm(dest_dev.get(queue) - 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*.

## 3.2 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.

### 3.2.1 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.

### 3.2.2 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.

### 3.2.3 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))
```

### 3.2.4 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))
```

### 3.2.5 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(...)
                                         }
                                     )
```

## 3.3 Public API

### 3.3.1 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**() → List[*API*]

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

**classmethod all_by_shortcut**(*shortcut: str | None = None*) → List[*API*]

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** – an API shortcut to match.

**classmethod from_api_id**(*api_id:* APIID) → *API*

Creates an *API* object out of an identifier.

**Parameters**
**api_id** – API identifier.

**id:** *APIID*

This API's ID.

**property platforms:** List[*Platform*]

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**() → *APIID*

Returns the identifier of CUDA API.

grunnur.**opencl_api_id**() → *APIID*

Returns the identifier of OpenCL API.

grunnur.**all_api_ids**() → List[*APIID*]

Returns a list of identifiers for all APIs available.

### 3.3.2 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**

A generalized GPGPU platform.

**classmethod all**(*api:* API) → List[*Platform*]

Returns a list of platforms available for the given API.

**Parameters**
**api** – the API to search in.

**classmethod all_filtered**(*api:* API, *filter:* PlatformFilter | *None* = *None*) → List[*Platform*]

Returns a list of all platforms satisfying the given criteria in the given API. If `filter` is not provided, returns all the platforms.

**classmethod from_backend_platform**(*obj: Any*) → *Platform*

Wraps a backend platform object into a Grunnur platform object.

**classmethod from_index**(*api:* API, *platform_idx: int*) → *Platform*

Creates a platform based on its index in the list returned by the API.

**Parameters**

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

**api:** *API*

The *API* object this platform belongs to.

**property devices:** List[*Device*]

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.

**class** grunnur.**PlatformFilter**

Bases: tuple

A set of filters for platform discovery.

Create new instance of PlatformFilter(include_masks, exclude_masks)

**exclude_masks:** List[str] | None

A list of strings (treated as regexes), neither of which must match the platform name.

**include_masks:** List[str] | None

A list of strings (treated as regexes), one of which must match the platform name.

### 3.3.3 Devices

**class** grunnur.**Device**

A generalized GPGPU device.

**classmethod all**(*platform:* Platform) → List[*Device*]

Returns a list of devices available for the given platform.

**Parameters**
**platform** – the platform to search in.

**classmethod all_filtered**(*platform:* Platform, *filter:* DeviceFilter | *None* = *None*) → List[*Device*]

Returns a list of all devices satisfying the given criteria in the given platform. If `filter` is not provided, returns all the devices.

**classmethod from_backend_device**(*obj: Any*) → *Device*

Wraps a backend device object into a Grunnur device object.

**classmethod from_index**(*platform:* Platform, *device_idx: int*) → *Device*

Creates a device based on its index in the list returned by the API.

**Parameters**

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

**name:** str

This device's name.

**property params:** *DeviceParameters*

Returns a *DeviceParameters* object associated with this device.

**platform:** *Platform*

The *Platform* object this device belongs to.

**class** grunnur.**DeviceFilter**

A set of filters for device discovery.

Create new instance of DeviceFilter(include_masks, exclude_masks, unique_only, exclude_pure_parallel)

**exclude_masks:** List[str] | None

A list of strings (treated as regexes), neither of which must match the device name.

**exclude_pure_parallel:** bool

If `True`, exclude devices with *params.max_total_local_size* equal to 1.

**include_masks:** List[str] | None

A list of strings (treated as regexes), one of which must match the device name.

**unique_only:** bool

If `True`, only return devices with unique names.

**class** grunnur.adapter_base.**DeviceParameters**

An object containing device's specifications.

**abstract property compute_units:** int

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

**abstract property local_mem_banks:** int

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

**abstract property local_mem_size:** int

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

**abstract property max_local_sizes:** Tuple[int, ...]

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

**abstract property max_num_groups:** Tuple[int, ...]

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

**abstract property max_total_local_size:** int

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

**abstract property type:** *DeviceType*

Device type.

**abstract property warp_size:** int

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

**class** grunnur.adapter_base.**DeviceType**

An enum representing a device's type.

**CPU = 1**

CPU type

**GPU = 2**

GPU type

### 3.3.4 Device discovery

grunnur.**platforms_and_devices_by_mask**(*api:* API, *quantity:* *int* | *None* = *1*, *device_filter:* DeviceFilter | *None* = *None*, *platform_filter:* PlatformFilter | *None* = *None*) → List[Tuple[*Platform*, List[*Device*]]]

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.

grunnur.**select_devices**(*api:* API, *interactive:* *bool* = *False*, *quantity:* *int* | *None* = *1*, *device_filter:* DeviceFilter | *None* = *None*, *platform_filter:* PlatformFilter | *None* = *None*) → List[*Device*]

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

### 3.3.5 Contexts

**class** grunnur.**Context**

GPGPU context.

**deactivate**() → None

For CUDA API: deactivates this context, popping all the CUDA context objects from the stack. Other APIs: no effect.

Only call it if you need to manage CUDA contexts manually, and created this object with *take_ownership = False*. If *take_ownership = True* contexts will be deactivated automatically in the destructor.

**classmethod from_backend_contexts**(*backend_contexts: Sequence[Any]*, *take_ownership: bool = False*) → *Context*

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 (only important for the CUDA backend).

**classmethod from_backend_devices**(*backend_devices: Sequence[Any]*) → *Context*

Creates a context from a single or several backend device objects.

**classmethod from_criteria**(*api:* API, *interactive: bool = False*, *devices_num: int | None = 1*, *device_filter:* DeviceFilter | *None = None*, *platform_filter:* PlatformFilter | *None = None*) → *Context*

Finds devices matching the given criteria and creates a *Context* object out of them.

**Parameters**

- **interactive** – passed to *select_devices()*.
- **devices_num** – passed to *select_devices()* as `quantity`.
- **device_filters** – passed to *select_devices()*.

**classmethod from_devices**(*devices: Sequence[*Device*]*) → *Context*

Creates a context from a device or an iterable of devices.

**Parameters**
**devices** – one or several devices to use.

**api:** *API*

The API this context is based on.

**property devices:** *BoundMultiDevice*

Returns the *BoundMultiDevice* encompassing all the devices in this context.

**platform:** *Platform*

The platform this context is based on.

**class** grunnur.context.**BoundDevice**

A *Device* object in a *Context*.

**context:** *Context*

The context this device belongs to.

**class** grunnur.context.**BoundMultiDevice**

Bases: `Sequence`[*BoundDevice*]

A sequence of bound devices belonging to the same context.

**__getitem__**(*idx: int*) → *BoundDevice*

**__getitem__**(*idx: slice | Iterable[int]*) → *BoundMultiDevice*

Given a single index, returns a single *BoundDevice*. Given a sequence of indices, returns a *BoundMultiDevice* object containing respective devices.

The indices correspond to the list of devices used to create this context.

**classmethod from_bound_devices**(*devices: Sequence[*BoundDevice*]*) → *BoundMultiDevice*

Creates this object from a sequence of bound devices (note that a `BoundMultiDevice` object itself can serve as such a sequence).

**context:** *Context*

The context these devices belong to.

### 3.3.6 Queues

**class** grunnur.**Queue**(*device:* BoundDevice)

A queue on a single device.

**Parameters**
**device** – a device on which to create a queue.

**synchronize**() → None

Blocks until sub-queues on all devices are empty.

**device:** *BoundDevice*

Device on which this queue operates.

**class** grunnur.**MultiQueue**(*queues: Sequence[*Queue*]*)

A queue on multiple devices.

**Parameters**
**queues** – single-device queues (must belong to distinct devices and the same context).

**classmethod on_devices**(*devices: Iterable[*BoundDevice*]*) → *MultiQueue*

Creates a queue from provided devices (belonging to the same context).

**synchronize**() → None

Blocks until queues on all devices are empty.

**devices:** *BoundMultiDevice*

Multi-device on which this queue operates.

**queues:** Dict[*BoundDevice*, *Queue*]

Single-device queues associated with device indices.

### 3.3.7 Buffers and arrays

**class** grunnur.**Buffer**

A memory buffer on device.

**classmethod allocate**(*device:* BoundDevice, *size: int*) → *Buffer*

Allocate a buffer of `size` bytes.

**Parameters**

- **device** – the device on which this buffer will be allocated.

- **size** – the buffer's size in bytes.

**get**(*queue:* Queue, *host_array:* *numpy.ndarray[Any, numpy.dtype[Any]]*, *async_:* *bool* = *False*) → None

Copy the contents of the buffer to the host array.

**Parameters**

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

**get_sub_region**(*origin:* *int*, *size:* *int*) → *Buffer*

Return a buffer object describing a subregion of this buffer.

**Parameters**

- **origin** – the offset of the subregion.
- **size** – the size of the subregion.

**set**(*queue:* Queue, *buf:* *numpy.ndarray[Any, numpy.dtype[Any]]* | Buffer, *no_async:* *bool* = *False*) → None

Copy the contents of the host array or another buffer to this buffer.

**Parameters**

- **queue** – the queue to use for the transfer.
- **buf** – the source - `numpy` array or a *`Buffer`* object.
- **no_async** – if *True*, the transfer blocks until completion.

**device:** *`BoundDevice`*

Device on which this buffer is allocated.

**property offset:** `int`

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

**property size:** `int`

This buffer's size (in bytes).

**class** grunnur.**ArrayMetadataLike**

Bases: `Protocol`

A protocol for an object providing array metadata. `numpy.ndarray` or *`Array`* follow this protocol.

**property dtype:** `numpy.dtype[Any]`

The type of an array element.

**property shape:** `Tuple[int, ...]`

Array shape.

**class** grunnur.**ArrayLike**

Bases: *`ArrayMetadataLike`*, `Protocol`

A protocol for an array-like object supporting views via `__getitem__()`. `numpy.ndarray` or *`Array`* follow this protocol.

**__getitem__**(*slices:* *slice* | *Tuple[slice, ...]*) → *_ArrayLike*

Returns a view of this array.

class grunnur.array._ArrayLike

Any type that follows the *ArrayLike* protocol.

alias of TypeVar('_ArrayLike', bound=*ArrayLike*)

class grunnur.Array

Array on a single device.

__getitem__(*slices: slice | Tuple[slice, ...]*) → *Array*

Returns a view of this array.

classmethod empty(*device:* BoundDevice, *shape: Sequence[int]*, *dtype:* DTypeLike, *strides: Sequence[int] | None = None*, *first_element_offset: int = 0*, *allocator: Callable[[*BoundDevice*, int], *Buffer*] | None = None*) → *Array*

Creates an empty array.

**Parameters**

- **device** – device on which this array will be allocated.
- **shape** – array shape.
- **dtype** – array data type.
- **allocator** – an optional callable taking two arguments (the bound device, and the buffer size in bytes) and returning a *Buffer* object. If None, will use *Buffer.allocate()*.

classmethod from_host(*queue_or_device:* Queue | BoundDevice, *host_arr: numpy.ndarray[Any, numpy.dtype[Any]]*) → *Array*

Creates an array object from a host array.

**Parameters**

- **queue** – the queue to use for the transfer.
- **host_arr** – the source array.

get(*queue:* Queue, *dest: numpy.ndarray[Any, numpy.dtype[Any]] | None = None*, *async_: bool = False*) → numpy.ndarray[Any, numpy.dtype[Any]]

Copies the contents of the array to the host array and returns it.

**Parameters**

- **queue** – the queue to use for the transfer.
- **dest** – the destination array. If None, the target array is created.
- **async** – if *True*, the transfer is performed asynchronously.

set(*queue:* Queue, *array: numpy.ndarray[Any, numpy.dtype[Any]] |* Array, *no_async: bool = False*) → None

Copies the contents of the host array to the array.

**Parameters**

- **queue** – the queue to use for the transfer.
- **array** – the source array.
- **no_async** – if *True*, the transfer blocks until completion.

device: *BoundDevice*

Device this array is allocated on.

**dtype:** *numpy.dtype[Any]*

Array item data type.

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

Array shape.

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

Array strides.

**class** grunnur.array.**BaseSplay**

Base class for splay strategies for *MultiArray*.

**abstract __call__**(*arr:* _ArrayLike, *devices:* *Sequence*[BoundDevice]) → Dict[*BoundDevice*, *_ArrayLike*]

Creates a dictionary of views of an array-like object for each of the given devices.

**Parameters**

- **arr** – an array-like object.
- **devices** – a multi-device object.

**class** grunnur.**MultiArray**

An array on multiple devices.

**class CloneSplay**

Copies the given array to each device.

**class EqualSplay**

Splays the given array equally between the devices using the outermost dimension. The outermost dimension should be larger or equal to the number of devices.

**classmethod empty**(*devices:* BoundMultiDevice, *shape:* *Sequence*[*int*], *dtype:* DTypeLike, *allocator:* *Callable*[[BoundDevice, *int*], Buffer] | *None* = *None*, *splay:* BaseSplay | *None* = *None*) → *MultiArray*

Creates an empty array.

**Parameters**

- **devices** – devices on which the sub-arrays will be allocated.
- **shape** – array shape.
- **dtype** – array data type.
- **allocator** – an optional callable taking two integer arguments (the device to allocate it on and the buffer size in bytes) and returning a *Buffer* object. If `None`, will use *Buffer.allocate()*.
- **splay** – the splay strategy (if `None`, an *EqualSplay* object is used).

**classmethod from_host**(*mqueue:* MultiQueue, *host_arr:* *numpy.ndarray*[Any, *numpy.dtype*[Any]], *splay:* BaseSplay | *None* = *None*) → *MultiArray*

Creates an array object from a host array.

**Parameters**

- **mqueue** – the queue to use for the transfer.
- **host_arr** – the source array.
- **splay** – the splay strategy (if `None`, an *EqualSplay* object is used).

**get**(*mqueue:* MultiQueue, *dest:* *numpy.ndarray[Any, numpy.dtype[Any]]* | *None* = *None*, *async_:* *bool* = *False*) → numpy.ndarray[Any, numpy.dtype[Any]]

Copies the contents of the array to the host array and returns it.

**Parameters**

- **mqueue** – the queue to use for the transfer.
- **dest** – the destination array. If `None`, the target array is created.
- **async** – if *True*, the transfer is performed asynchronously.

**set**(*mqueue:* MultiQueue, *array:* *numpy.ndarray[Any, numpy.dtype[Any]]* | MultiArray, *no_async:* *bool* = *False*) → None

Copies the contents of the host array to the array.

**Parameters**

- **mqueue** – the queue to use for the transfer.
- **array** – the source array.
- **no_async** – if *True*, the transfer blocks until completion.

**devices:** *BoundMultiDevice*

Devices on which the sub-arrays are allocated

**dtype:** `numpy.dtype[Any]`

Array item data type.

**shape:** `Tuple[int, ...]`

Array shape.

**shapes:** `Dict[BoundDevice, Tuple[int, ...]]`

Sub-array shapes matched to device indices.

### 3.3.8 Programs and kernels

**class** grunnur.**Program**(*devices:* *Sequence[*BoundDevice*]*, *template_src:* *str* | *Callable[[...], str]* | DefTemplate | Snippet, *no_prelude:* *bool* = *False*, *fast_math:* *bool* = *False*, *render_args:* *Sequence[Any]* = *()*, *render_globals:* *Mapping[str, Any]* = *{}*, *compiler_options:* *Sequence[str]* | *None* = *None*, *keep:* *bool* = *False*, *constant_arrays:* *Mapping[str,* ArrayMetadataLike*]* | *None* = *None*)

A compiled program on device(s).

**Parameters**

- **devices** – a single- or a multi-device object on which to compile this program.
- **template_src** – a string with the source code, or a Mako template source to render.
- **no_prelude** – do not add prelude to the rendered source.
- **fast_math** – compile using fast (but less accurate) math functions.
- **render_args** – a list of positional args to pass to the template.
- **render_globals** – a dictionary of globals to pass to the template.
- **compiler_options** – a list of options to pass to the backend compiler.
- **keep** – keep the intermediate files in a temporary directory.

- **constant_arrays** – (**CUDA only**) a dictionary `name: (size, dtype)` of global constant arrays to be declared in the program.

`set_constant_array`(*queue:* Queue, *name:* *str*, *arr:* Array | *numpy.ndarray[Any, numpy.dtype[Any]]*) → None

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

**Parameters**

- **queue** – the queue to use for the transfer.
- **name** – the name of the constant array symbol in the code.
- **arr** – either a device or a host array.

`devices: BoundMultiDevice`

The devices on which this program was compiled.

`kernel: KernelHub`

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

`sources: Dict[BoundDevice, str]`

Source files used for each device.

`class grunnur.program.KernelHub`

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

`__getattr__`(*kernel_name:* *str*) → *Kernel*

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

`class grunnur.program.Kernel`

A kernel compiled for multiple devices.

`__call__`(*queue:* Queue | MultiQueue, *global_size:* *Sequence[int]* | *Mapping[*BoundDevice, *Sequence[int]]*, *local_size:* *Sequence[int]* | *None* | *Mapping[*BoundDevice, *Sequence[int]* | *None]* = *None*, **args:* MultiArray | Array | Buffer | *generic*, *local_mem:* *int* = *0*) → Any

A shortcut for `Kernel.prepare()` and subsequent `PreparedKernel.__call__()`. See their doc entries for details.

`prepare`(*global_size:* *Sequence[int]* | *Mapping[*BoundDevice, *Sequence[int]]*, *local_size:* *Sequence[int]* | *None* | *Mapping[*BoundDevice, *Sequence[int]* | *None]* = *None*) → *PreparedKernel*

Prepares the kernel for execution.

If `local_size` or `global_size` are integer, they will be treated as 1-tuples.

One can pass specific global and local sizes for each device using dictionaries keyed with device indices. This achieves another purpose: the kernel will only be prepared for those devices, and not for all devices available in the context.

**Parameters**

- **global_size** – 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** – 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.

`property max_total_local_sizes: Dict[BoundDevice, int]`

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

class grunnur.program.**PreparedKernel**

A kernel specialized for execution on a set of devices with all possible preparations and checks performed.

__call__(*queue:* Queue | MultiQueue, **args:* MultiArray | Array | Buffer | *generic*, *local_mem:* *int* = *0*) → Any

Enqueues the kernel on the devices in the given queue. The kernel must have been prepared for all of these devices.

If an argument is a *Array* or *Buffer* object, it must belong to the device on which the kernel is being executed (so `queue` must only have one device).

If an argument is a *MultiArray*, it should have subarrays on all the devices from the given `queue`.

If an argument is a `numpy` scalar, it will be passed to the kernel directly.

If an argument is a integer-keyed `dict`, its values corresponding to the device indices the kernel is executed on will be passed as kernel arguments.

**Parameters**

- **args** – kernel arguments.
- **kwds** – backend-specific keyword parameters.

**Returns**

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

### 3.3.9 Static kernels

class grunnur.**StaticKernel**(*devices:* *Sequence[*BoundDevice*]*, *template_src:* *str* | *Callable[[...], str]* | DefTemplate | Snippet, *name:* *str*, *global_size:* *Sequence[int]* | *Mapping[*BoundDevice*,* *Sequence[int]]*, *local_size:* *Sequence[int]* | *None* | *Mapping[*BoundDevice*,* *Sequence[int]* | *None]* = *None*, *render_args:* *Sequence[Any]* = *()*, *render_globals:* *Mapping[str, Any]* = *{}*, *constant_arrays:* *Mapping[str,* ArrayMetadataLike*]* | *None* = *None*, *keep:* *bool* = *False*, *fast_math:* *bool* = *False*, *compiler_options:* *Sequence[str]* | *None* = *None*)

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**

- **devices** – a single- or a multi-device object on which to compile this program.
- **template_src** – a string with the source code, or a Mako template source to render.
- **name** – the kernel's name.
- **global_size** – see *prepare()*.
- **local_size** – see *prepare()*.
- **render_globals** – a dictionary of globals to pass to the template.
- **constant_arrays** – (**CUDA only**) a dictionary `name: (size, dtype)` of global constant arrays to be declared in the program.

__call__(*queue:* Queue, **args:* Array | *generic*) → Any

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. See *grunnur.program.PreparedKernel.__call__()*.

**set_constant_array**(*queue:* Queue, *name:* *str*, *arr:* Array | *numpy.ndarray[Any, numpy.dtype[Any]]*) → None

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

**Parameters**

- **queue** – the queue to use for the transfer.
- **name** – the name of the constant array symbol in the code.
- **arr** – either a device or a host array.

**devices:** *BoundMultiDevice*

Devices on which this kernel was compiled.

**queue:** *Queue*

The queue this static kernel was compiled and prepared for.

**sources:** Dict[*BoundDevice*, str]

Source files used for each device.

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

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

Create new instance of VsizeModules(local_id, local_size, group_id, num_groups, global_id, global_size, global_flat_id, global_flat_size, skip)

**global_flat_id:** *Module*

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

**global_flat_size:** *Module*

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

**global_id:** *Module*

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

**global_size:** *Module*

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

**group_id:** *Module*

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

**local_id:** *Module*

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

**local_size:** *Module*

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

**num_groups:** *Module*

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

**skip:** *Module*

Provides the function `bool ${skip}()` that should be used at the start of a static kernel function to see if the current thread/work item is inside the padding area and needs to be skipped. Usually one would write `if (${skip}()) return;`.

### 3.3.10 Utilities

**class** grunnur.**Template**(*mako_template: mako.template.Template*)

A wrapper for mako `Template` objects.

**classmethod from_associated_file**(*filename: str*) → *Template*

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__)`.

**classmethod from_string**(*template_source: str*) → *Template*

Returns a *Template* object created from source.

**get_def**(*name: str*) → *DefTemplate*

Returns the template def with the name `name`.

**class** grunnur.**DefTemplate**(*name: str*, *mako_def_template: mako.template.DefTemplate*, *source: str*)

A wrapper for Mako `DefTemplate` objects.

**classmethod from_callable**(*name: str*, *callable_obj: Callable[[...], str]*) → *DefTemplate*

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.

**classmethod from_string**(*name: str*, *argnames: Iterable[str]*, *source: str*) → *DefTemplate*

Creates a template def from a string with its body and a list of argument names.

**render**(**args: Any*, ***globals_: Any*) → str

Renders the template def with given arguments and globals.

**class** grunnur.**RenderError**(*exception: Exception*, *args: Sequence[Any]*, *globals_: Mapping[str, Any]*, *source: str*)

A custom wrapper for Mako template render errors, to facilitate debugging.

**exception:** `Exception`

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

**globals:** `Dict[str, Any]`

The globals used to render the template.

**source:** `str`

The source of the template.

**class** grunnur.**Snippet**(*template:* DefTemplate, *render_globals: Mapping[str, Any] = {}*)

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.

**classmethod from_callable**(*callable_obj: Callable[[...], str]*, *name: str = '_snippet'*, *render_globals: Mapping[str, Any] = {}*) → *Snippet*

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** – a callable returning the template source.
- **name** – the snippet's name (will simplify debugging)
- **render_globals** – a dictionary of "globals" to be used when rendering the template.

**classmethod from_string**(*source: str*, *name: str = '_snippet'*, *render_globals: Mapping[str, Any] = {}*) → *Snippet*

Creates a snippet from a template source, treated as a body of a template def with no arguments.

**Parameters**

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

**class** grunnur.**Module**(*template:* DefTemplate, *render_globals: Mapping[str, Any] = {}*)

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** –
- **render_globals** –

**classmethod from_callable**(*callable_obj: Callable[[...], str]*, *name: str = '_module'*, *render_globals: Mapping[str, Any] = {}*) → *Module*

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

**classmethod from_string**(*source: str*, *name: str = '_module'*, *render_globals: Mapping[str, Any] = {}*) → *Module*

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

**Parameters**

- **source** – a string with the template source.
- **name** – the module's name (will simplify debugging)

- **render_globals** – a dictionary of "globals" to be used when rendering the template.

### 3.3.11 Data type utilities

**class** numpy.typing.**DTypeLike**

intersphinx fails to pick this up. See numpy.typing.DTypeLike for the actual documentation.

#### C interop

grunnur.dtypes.**ctype**(*dtype:* DTypeLike) → str | *Module*

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()*).

grunnur.dtypes.**ctype_struct**(*dtype:* DTypeLike, *ignore_alignment:* *bool* = *False*) → *Module*

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.

**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.

grunnur.dtypes.**complex_ctr**(*dtype:* DTypeLike) → str

Returns name of the constructor for the given dtype.

grunnur.dtypes.**c_constant**(*val:* *int* | *float* | *complex* | *generic* | *numpy.ndarray[Any, numpy.dtype[Any]]*, *dtype:* DTypeLike | *None* = *None*) → str

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.

grunnur.dtypes.**align**(*dtype:* DTypeLike) → numpy.dtype[Any]

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.

#### Struct helpers

grunnur.dtypes.**c_path**(*path: List[str | int]*) → str

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()*.

grunnur.dtypes.**flatten_dtype**(*dtype:* DTypeLike) → List[Tuple[List[str | int], numpy.dtype[Any]]]

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.

grunnur.dtypes.**extract_field**(*arr: numpy.ndarray[Any, numpy.dtype[Any]]*, *path: List[str | int]*) → generic | numpy.ndarray[Any, numpy.dtype[Any]]

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

#### Data type checks and conversions

grunnur.dtypes.**is_complex**(*dtype:* DTypeLike) → bool

Returns `True` if `dtype` is complex.

grunnur.dtypes.**is_double**(*dtype:* DTypeLike) → bool

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

grunnur.dtypes.**is_integer**(*dtype:* DTypeLike) → bool

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

grunnur.dtypes.**is_real**(*dtype:* DTypeLike) → bool

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

grunnur.dtypes.**result_type**(**dtypes:* DTypeLike) → numpy.dtype[Any]

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

grunnur.dtypes.**min_scalar_type**(*val: int | float | complex | numpy.number[Any]*, *force_signed: bool = False*) → numpy.dtype[Any]

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

grunnur.dtypes.**complex_for**(*dtype:* DTypeLike) → numpy.dtype[Any]

Returns complex dtype corresponding to given floating point `dtype`.

grunnur.dtypes.**real_for**(*dtype:* DTypeLike) → numpy.dtype[Any]

Returns floating point dtype corresponding to given complex `dtype`.

### 3.3.12 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: numpy.dtype[Any]*, *out_dtype: numpy.dtype[Any] | None = None*) → *Module*

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)`).

grunnur.functions.**cast**(*in_dtype: numpy.dtype[Any]*, *out_dtype: numpy.dtype[Any]*) → *Module*

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

grunnur.functions.**conj**(*dtype: numpy.dtype[Any]*) → *Module*

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).

grunnur.functions.**div**(*dividend_dtype: numpy.dtype[Any]*, *divisor_dtype: numpy.dtype[Any]*, *out_dtype: numpy.dtype[Any] | None = None*) → *Module*

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.

grunnur.functions.**exp**(*dtype: numpy.dtype[Any]*) → *Module*

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).

grunnur.functions.**mul**(**in_dtypes: numpy.dtype[Any]*, *out_dtype: numpy.dtype[Any] | None = None*) → *Module*

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.

grunnur.functions.**norm**(*dtype: numpy.dtype[Any]*) → *Module*

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).

grunnur.functions.**polar**(*dtype: numpy.dtype[Any]*) → *Module*

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).

grunnur.functions.**polar_unit**(*dtype: numpy.dtype[Any]*) → *Module*

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).

grunnur.functions.**pow**(*base_dtype: numpy.dtype[Any]*, *exponent_dtype: numpy.dtype[Any] | None = None*, *out_dtype: numpy.dtype[Any] | None = None*) → *Module*

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.

### 3.3.13 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**(*device:* BoundDevice)

Base class for a manager of virtual allocations.

**Parameters**

**context** – an instance of *Context*.

**allocator**(*dependencies: Any | None = None*) → *VirtualAllocator*

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).

**pack**(*queue:* Queue) → None

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

**statistics**() → *VirtualAllocationStatistics*

Returns allocation statistics.

**class** grunnur.virtual_alloc.**TrivialManager**(*device:* BoundDevice)

Trivial manager — allocates a separate buffer for each allocation request.

**class** grunnur.virtual_alloc.**ZeroOffsetManager**(*device:* BoundDevice)

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:* VirtualManager, *dependencies: Set[int]*)

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**

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.

### 3.3.14 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;
```

## 3.4 Version history

### 3.4.1 Current development version

- (CHANGED) `device_idx` parameters are gone; now high level functions take `BoundDevice` or `BoundMultiDevice` arguments to indicate which devices to use; these objects include the corresponding contexts as well, so they don't have to be passed separately.
- Now API adapters only use device indices in a sense of "device index in the platform"; context adapters keep internal objects in dictionaries indexed by these indices, instead of in lists.

### 3.4.2 0.2.0 (10 Mar 2021)

- (CHANGED) Arrays don't hold queues any more; they are passed explicitly to `get()` or `set()`.
- (CHANGED) Prepared kernels don't hold queues any more; they are passed on call.
- (CHANGED) `Queue` now stands for a single-device queue only; multi-device queues are extracted into `MultiQueue`.
- (ADDED) `MultiArray` to simplify simultaneous kernel execution on multiple devices.

### 3.4.3 0.1.1 (9 Oct 2020)

Package build fixed.

#### 3.4.4 0.1.0 (9 Oct 2020)

Initial version

CHAPTER

FOUR

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## g



## n

# INDEX

## Symbols



## A



## B



## C



## D

## E



## F



## G



## I



## K



## L

## M



## N



## O



## P



## Q



## R



## S



## T

## U



## V



## W



## Z