Plures

Release v0.2.0dev3

Dec 11, 2018

Contents

xnd is a package for general typed containers. xnd relies on the libndtypes library for typing and memory layout information.

CHAPTER 1

Libxnd

C library.

1.1 libxnd

libxnd implements support for typed memory blocks using the libndtypes type library.

Types include ndarrays, ragged arrays (compatible with the Arrow list type), optional data (bitmaps are compatible with Arrow), tuples, records (structs), strings, bytes and categorical values.

1.1.1 Data structures

libxnd is a lightweight container library that leaves most of the work to libndtypes. The underlying idea is to have a small struct that contains bitmaps, a linear index, a type and a data pointer.

The type contains all low level access information.

Since the struct is small, it can easily be passed around by value when it serves as a view on the data.

The owner of the data is a master buffer with some additional bookkeeping fields.

Bitmaps

```
typedef struct xnd_bitmap xnd_bitmap_t;

struct xnd_bitmap {
    uint8_t *data;      /* bitmap */
    int64_t size;       /* number of subtree bitmaps in the "next" array */
    xnd_bitmap_t *next; /* array of bitmaps for subtrees */
};
```

libxnd supports data with optional values. Any type can have a bitmap for its data. Bitmaps are addressed using the linear index in the `xnd_t` struct.

For a scalar, the bitmap contains just one addressable bit.

For fixed and variable arrays, the bitmap contains one bit for each item.

Container types like tuples and records have new bitmaps for each of their fields if any of the field subtrees contains optional data.

These field bitmaps are in the *next* array.

View

```
/* Typed memory block, usually a view. */
typedef struct xnd {
    xnd_bitmap_t bitmap; /* bitmap tree */
    int64_t index;       /* linear index for var dims */
    const ndt_t *type;   /* type of the data */
    char *ptr;           /* data */
} xnd_t;
```

This is the xnd view, with a typed data pointer, the current linear index and a bitmap.

When passing the view around, the linear index needs to be maintained because it is required to determine if a value is missing (NA).

The convention is to apply the linear index (adjust the *ptr* and set it to *0*) only when a non-optional element at *ndim == 0* is actually accessed.

This happens for example when getting the value of an element or when descending into a record.

Flags

```
#define XND_OWN_TYPE     0x00000001U /* type pointer */
#define XND_OWN_DATA     0x00000002U /* data pointer */
#define XND_OWN_STRINGS  0x00000004U /* embedded string pointers */
#define XND_OWN_BYTES    0x00000008U /* embedded bytes pointers */
#define XND_OWN_POINTERS 0x00000010U /* embedded pointers */

#define XND_OWN_ALL (XND_OWN_TYPE |     \
                     XND_OWN_DATA |     \
                     XND_OWN_STRINGS |  \
                     XND_OWN_BYTES |    \
                     XND_OWN_POINTERS)

#define XND_OWN_EMBEDDED (XND_OWN_DATA |     \
                          XND_OWN_STRINGS |  \
                          XND_OWN_BYTES |    \
                          XND_OWN_POINTERS)
```

The ownership flags for the xnd master buffer (see below). Like libndtypes, libxnd itself has no notion of how many exported views a master buffer has.

This is deliberately done in order to prevent two different memory management schemes from getting in each other's way.

However, for deallocating a master buffer the flags must be set correctly.

`XND_OWN_TYPE` is set if the master buffer owns the `ndt_t`.

`XND_OWN_DATA` is set if the master buffer owns the data pointer.

The *string*, *bytes* and *ref* types have pointers that are embedded in the data. Usually, these are owned and deallocated by libxnd.

For strings, the Python bindings use the convention that `NULL` strings are interpreted as the empty string. Once a string pointer is initialized it belongs to the master buffer.

Macros

```
/* Convenience macros to extract embedded values. */
#define XND_POINTER_DATA(ptr) (*((char **)ptr))
#define XND_BYTES_SIZE(ptr) (((ndt_bytes_t *)ptr)->size)
#define XND_BYTES_DATA(ptr) (((ndt_bytes_t *)ptr)->data)
```

These macros should be used to extract embedded *ref*, *string* and *bytes* data.

Master buffer

```
/* Master memory block. */
typedef struct xnd_master {
    uint32_t flags; /* ownership flags */
    xnd_t master;   /* typed memory */
} xnd_master_t;
```

This is the master buffer. *flags* are explained above, the *master* buffer should be considered constant.

For traversing memory, copy a new view buffer by value.

Slice and index keys

```
enum xnd_key { Index, FieldName, Slice };
typedef struct {
  enum xnd_key tag;
  union {
    int64_t Index;
    const char *FieldName;
    ndt_slice_t Slice;
  };
} xnd_index_t;
```

Slicing and indexing uses the same model as Python. Indices are usually integers, but record fields may also be indexed with field names.

ndt_slice_t has *start*, *stop*, *step* fields that must be filled in with normalized values following the same protocol as `PySlice_Unpack`.

1.1.2 Functions

Create typed memory blocks

The main use case for libxnd is to create and manage typed memory blocks. These blocks are fully initialized to *0*. References to additional memory blocks are allocated and initialized recursively.

bytes and *string* types are initialized to `NULL`, since their actual length is not known yet.

```
xnd_master_t *xnd_empty_from_string(const char *s, uint32_t flags, ndt_context_t␣
↪*ctx);
```

Return a new master buffer according to the type string in *s*. *flags* must include `XND_OWN_TYPE`.

```
xnd_master_t *xnd_empty_from_type(const ndt_t *t, uint32_t flags, ndt_context_t *ctx);
```

Return a new master buffer according to *type*. *flags* must not include `XND_OWN_TYPE`, i.e. the type is externally managed.

This is the case in the Python bindings, where the ndtypes module creates and manages types.

Delete typed memory blocks

```
void xnd_del(xnd_master_t *x);
```

Delete the master buffer according to its flags. *x* may be `NULL`. *x->master.ptr* and *x->master.type* may be `NULL`.

The latter situation should only arise when breaking up reference cycles. This is used in the Python module.

Bitmaps

```
xnd_bitmap_t xnd_bitmap_next(const xnd_t *x, int64_t i, ndt_context_t *ctx);
```

Get the next bitmap for the *Tuple*, *Record*, *Ref* and *Constr* types.

This is a convenience function that checks if the types have optional subtrees.

If yes, return the bitmap at index *i*. If not, it return an empty bitmap that must not be accessed.

```
void xnd_set_valid(xnd_t *x);
```

Set the validity bit at *x->index*. *x* must have an optional type.

```
void xnd_set_na(xnd_t *x);
```

Clear the validity bit at *x->index*. *x* must have an optional type.

```
int xnd_is_valid(const xnd_t *x);
```

Check if the element at *x->index* is valid. If *x* does not have an optional type, return *1*. Otherwise, return the validity bit (zero or nonzero).

```
int xnd_is_na(const xnd_t *x);
```

Check if the element at *x->index* is valid. If *x* does not have an optional type, return *0*. Otherwise, return the negation of the validity bit.

```
xnd_t xnd_subtree(const xnd_t *x, const xnd_index_t indices[], int len,
                  ndt_context_t *ctx);
```

Apply zero or more indices to the input *x* and return a typed view. Valid indices are integers or strings for record fields.

This function is more general than pure array indexing, hence the name. For example, it is possible to index into nested records that in turn contain arrays.

```
xnd_t xnd_multikey(const xnd_t *x, const xnd_index_t indices[], int len,
                   ndt_context_t *ctx);
```

Apply zero or more keys to the input *x* and return a typed view. Valid keys are integers or slices.

This function differs from `xnd_subtree` in that it allows mixed indexing and slicing for fixed dimensions. Records and tuples cannot be sliced.

Variable dimensions can be sliced, but do not support mixed indexing and slicing.

CHAPTER 2

Xnd

Python bindings for libxnd.

2.1 xnd

The xnd module implements a container type that maps most Python values relevant for scientific computing directly to typed memory.

Whenever possible, a single, pointer-free memory block is used.

xnd supports ragged arrays, categorical types, indexing, slicing, aligned memory blocks and type inference.

Operations like indexing and slicing return zero-copy typed views on the data.

Importing PEP-3118 buffers is supported.

2.1.1 Types

The xnd object is a container that maps a wide range of Python values directly to memory. xnd unpacks complex types of arbitrary nesting depth to a single memory block.

Pointers only occur in explicit pointer types like *Ref* (reference), *Bytes* and *String*, but not in the general case.

Type inference

If no explicit type is given, xnd supports type inference by assuming types for the most common Python values.

Fixed arrays

```
>>> from xnd import *
>>> x = xnd([[0, 1, 2], [3, 4, 5]])
>>> x
xnd([[0, 1, 2], [3, 4, 5]], type='2 * 3 * int64')
```

As expected, lists are mapped to ndarrays and integers to int64. Indexing and slicing works the usual way. For performance reasons these operations return zero-copy views whenever possible:

```
>>> x[0][1] # Indexing returns views, even for scalars.
xnd(1, type='int64')
>>>
>>> y = x[:, ::-1] # Containers are returned as views.
>>> y
xnd([[2, 1, 0], [5, 4, 3]], type='2 * 3 * int64')
```

Subarrays are views and properly typed:

```
>>> x[1]
xnd([3, 4, 5], type='3 * int64')
```

The representation of large values is abbreviated:

```
>>> x = xnd(10 * [200 * [1]])
>>> x
xnd([[1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     [1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
     ...],
    type='10 * 200 * int64')
```

Values can be accessed in full using the *value* property:

```
>>> x = xnd(11 * [1])
>>> x
xnd([1, 1, 1, 1, 1, 1, 1, 1, 1, ...], type='11 * int64')
>>> x.value
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Types can be accessed using the *type* property:

```
>>> x = xnd(11 * [1])
>>> x.type
ndt("11 * int64")
```

Ragged arrays

Ragged arrays are compatible with the Arrow list representation. The data is pointer-free, addressing the elements works by having one offset array per dimension.

```
>>> xnd([[0.1j], [3+2j, 4+5j, 10j]])
xnd([[0.1j], [(3+2j), (4+5j), 10j]], type='var * var * complex128')
```

Indexing and slicing works as usual, returning properly typed views or values in the case of scalars:

```
>>> x = xnd([[0.1j], [3+2j, 4+5j, 10j]])
>>> x[1, 2]
xnd(10j, type='complex128')

>>> x[1]
xnd([(3+2j), (4+5j), 10j], type='var * complex128')
```

Eliminating dimensions through mixed slicing and indexing is not supported because it would require copying and adjusting potentially huge offset arrays:

```
>>> y = x[:, 1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: mixed indexing and slicing is not supported for var dimensions
```

Records (structs)

From Python 3.6 on, dicts retain their order, so they can be used directly for initializing C structs.

```
>>> xnd({'a': 'foo', 'b': 10.2})
xnd({'a': 'foo', 'b': 10.2}, type='{a : string, b : float64}')
```

Tuples

Python tuples are directly translated to the libndtypes tuple type:

```
>>> xnd(('foo', b'bar', [None, 10.0, 20.0]))
xnd(('foo', b'bar', [None, 10.0, 20.0]), type='(string, bytes, 3 * ?float64)')
```

Nested arrays in structs

xnd seamlessly supports nested values of arbitrary depth:

```
>>> lst = [{'name': 'John', 'internet_points': [1, 2, 3]},
...        {'name': 'Jane', 'internet_points': [4, 5, 6]}]
>>> xnd(lst)
xnd([{'name': 'John', 'internet_points': [1, 2, 3]}, {'name': 'Jane', 'internet_points
↪': [4, 5, 6]}],
    type='2 * {name : string, internet_points : 3 * int64}')
```

Optional data (missing values)

Optional data is currently specified using *None*. It is under debate if a separate *NA* singleton object would be more suitable.

```
>>> lst = [0, 1, None, 2, 3, None, 5, 10]
>>> xnd(lst)
xnd([0, 1, None, 2, 3, None, 5, 10], type='8 * ?int64')
```

Categorical data

Type inference would be ambiguous, so it cannot work directly. xnd supports the *levels* argument that is internally translated to the type.

```
>>> levels = ['January', 'August', 'December', None]
>>> x = xnd(['January', 'January', None, 'December', 'August', 'December', 'December
↪'], levels=levels)
>>> x.value
['January', 'January', None, 'December', 'August', 'December', 'December']
>>> x.type
ndt("7 * categorical('January', 'August', 'December', NA)")
```

The above is equivalent to specifying the type directly:

```
>>> from ndtypes import *
>>> t = ndt("7 * categorical('January', 'August', 'December', NA)")
>>> x = xnd(['January', 'January', None, 'December', 'August', 'December', 'December
↪'], type=t)
>>> x.value
['January', 'January', None, 'December', 'August', 'December', 'December']
>>> x.type
ndt("7 * categorical('January', 'August', 'December', NA)")
```

Explicit types

While type inference is well-defined, it necessarily makes assumptions about the programmer's intent.

There are two cases where types should be given:

Different types are intended

```
>>> xnd([[0,1,2], [3,4,5]], type="2 * 3 * uint8")
xnd([[0, 1, 2], [3, 4, 5]], type='2 * 3 * uint8')
```

Here, type inference would deduce `int64`, so `uint8` needs to be passed explicitly.

All supported types

Fixed arrays

Fixed arrays are similar to NumPy's ndarray. One difference is that internally xnd uses steps instead of strides. One step is the amount of indices required to move the linear index from one dimension element to the next.

This facilitates optional data, whose bitmaps need to be addressed by the linear index. The equation *stride* = *step* * *itemsize* always holds.

```
>>> xnd([[[1,2], [None, 3]], [[4, None], [5, 6]]])
xnd([[[1, 2], [None, 3]], [[4, None], [5, 6]]], type='2 * 2 * 2 * ?int64')
```

This is a fixed array with optional data.

```
>>> xnd([(1,2.0,3j), (4,5.0,6j)])
xnd([(1, 2.0, 3j), (4, 5.0, 6j)], type='2 * (int64, float64, complex128)')
```

An array with tuple elements.

Fortran order

Fortran order is specified by prefixing the dimensions with an exclamation mark:

```
>>> lst = [[1, 2, 3], [4, 5, 6]]
>>> x = xnd(lst, type='!2 * 3 * uint16')
>>>
>>> x.type.shape
(2, 3)
>>> x.type.strides
(2, 4)
```

Alternatively, steps can be passed as arguments to the fixed dimension type:

```
>>> from ndtypes import *
>>> lst = [[1, 2, 3], [4, 5, 6]]
>>> t = ndt("fixed(shape=2, step=1) * fixed(shape=3, step=2) * uint16")
>>> x = xnd(lst, type=t)
>>> x.type.shape
(2, 3)
>>> x.type.strides
(2, 4)
```

Ragged arrays

Ragged arrays with explicit types are easiest to construct using the *dtype* argument to the xnd constructor.

```
>>> lst = [[0], [1, 2], [3, 4, 5]]
>>> xnd(lst, dtype="int32")
xnd([[0], [1, 2], [3, 4, 5]], type='var * var * int32')
```

Alternatively, offsets can be passed as arguments to the var dimension type:

```
>>> from ndtypes import ndt
>>> t = ndt("var(offsets=[0,3]) * var(offsets=[0,1,3,6]) * int32")
>>> xnd(lst, type=t)
xnd([[0], [1, 2], [3, 4, 5]], type='var * var * int32')
```

Tuples

In memory, tuples are the same as C structs.

```
>>> xnd(("foo", 1.0))
xnd(('foo', 1.0), type='(string, float64)')
```

Indexing works the same as for arrays:

```
>>> x = xnd(("foo", 1.0))
>>> x[0]
xnd('foo', type='string')
```

Nested tuples are more general than ragged arrays. They can a) hold different data types and b) the trees they represent may be unbalanced.

They do not allow slicing though and are probably less efficient.

This is an example of an unbalanced tree that cannot be represented as a ragged array:

```
>>> unbalanced_tree = (((1.0, 2.0), (3.0)), 4.0, ((5.0, 6.0, 7.0), ()))
>>> x = xnd(unbalanced_tree)
>>> x.value
(((1.0, 2.0), 3.0), 4.0, ((5.0, 6.0, 7.0), ()))
>>> x.type
ndt("(((float64, float64), float64), float64, ((float64, float64, float64), ()))")
>>>
>>> x[0]
xnd(((1.0, 2.0), 3.0), type='((float64, float64), float64)')
>>> x[0][0]
xnd((1.0, 2.0), type='(float64, float64)')
```

Note that the data in the above tree example is packed into a single contiguous memory block.

Records

In memory, records are C structs. The field names are only stored in the type.

The following examples use Python-3.6, which keeps the dict initialization order.

```
>>> x = xnd({'a': b'123', 'b': {'x': 1.2, 'y': 100+3j}})
>>> x.value
{'a': b'123', 'b': {'x': 1.2, 'y': (100+3j)}}
>>> x.type
ndt("{a : bytes, b : {x : float64, y : complex128}}")
```

Indexing works the same as for arrays. Additionally, fields can be indexed by name:

```
>>> x[0]
xnd(b'123', type='bytes')
>>> x['a']
xnd(b'123', type='bytes')
>>> x['b']
xnd({'x': 1.2, 'y': (100+3j)}, type='{x : float64, y : complex128}')
```

The nesting depth is arbitrary. In the following example, the data – except for strings, which are pointers – is packed into a single contiguous memory block:

```
>>> from pprint import pprint
>>> item = {
...   "id": 1001,
```

(continues on next page)

(continued from previous page)

```
...   "name": "cyclotron",
...   "price": 5998321.99,
...   "tags": ["connoisseur", "luxury"],
...   "stock": {
...     "warehouse": 722,
...     "retail": 20
...   }
... }
>>> x = xnd(item)
>>>
>>> pprint(x.value)
{'id': 1001,
 'name': 'cyclotron',
 'price': 5998321.99,
 'stock': {'retail': 20, 'warehouse': 722},
 'tags': ['connoisseur', 'luxury']}
>>>
>>> x.type.pprint()
{
  id : int64,
  name : string,
  price : float64,
  tags : 2 * string,
  stock : {
    warehouse : int64,
    retail : int64
  }
}
```

Strings can be embedded into the array by specifying the fixed string type. In this case, the memory block is pointer-free.

```
>>> from ndtypes import ndt
>>>
>>> t = """
...   { id : int64,
...     name : fixed_string(30),
...     price : float64,
...     tags : 2 * fixed_string(30),
...     stock : {warehouse : int64, retail : int64}
...   }
... """
>>>
>>> x = xnd(item, type=t)
>>> x.type.pprint()
{
  id : int64,
  name : fixed_string(30),
  price : float64,
  tags : 2 * fixed_string(30),
  stock : {
    warehouse : int64,
    retail : int64
  }
}
```

Record of arrays

Often it is more memory efficient to store an array of records as a record of arrays. This example with columnar data is from the Arrow homepage:

```
>>> data = {'session_id': [1331247700, 1331247702, 1331247709, 1331247799],
...         'timestamp': [1515529735.4895875, 1515529746.2128427, 1515529756.4485607,␣
↪1515529766.2181058],
...         'source_ip': ['8.8.8.100', '100.2.0.11', '99.101.22.222', '12.100.111.200
↪']}
>>> x = xnd(data)
>>> x.type
ndt("{session_id : 4 * int64, timestamp : 4 * float64, source_ip : 4 * string}")
```

References

References are transparent pointers to new memory blocks (meaning a new data pointer, not a whole new xnd buffer).

For example, this is an array of pointer to array:

```
>>> t = ndt("3 * ref(4 * uint64)")
>>> lst = [[0,1,2,3], [4,5,6,7], [8,9,10,11]]
>>> xnd(lst, type=t)
xnd([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]], type='3 * ref(4 * uint64)')
```

The user sees no difference to a regular 3 by 4 array, but internally the outer dimension consists of three pointers to the inner arrays.

For memory blocks generated by xnd itself the feature is not so useful – after all, it is usually better to have a single memory block than one with additional pointers.

However, suppose that in the above columnar data example another application represents the arrays inside the record with pointers. Using the *ref* type, data structures borrowed from such an application can be properly typed:

```
>>> t = ndt("{session_id : &4 * int64, timestamp : &4 * float64, source_ip : &4 *␣
↪string}")
>>> x = xnd(data, type=t)
>>> x.type
ndt("{session_id : ref(4 * int64), timestamp : ref(4 * float64), source_ip : ref(4 *␣
↪string)}")
```

The ampersand is the shorthand for “ref”.

Constructors

Constructors are xnd’s way of creating distinct named types. The constructor argument is a regular type.

Constructors open up a new dtype, so named arrays can be the dtype of other arrays. Type inference currently isn’t aware of constructors, so types have to be provided.

```
>>> t = ndt("3 * SomeMatrix(2 * 2 * float32)")
>>> lst = [[[1,2], [3,4]], [[5,6], [7,8]], [[9,10], [11,12]]]
>>> x = xnd(lst, type=t)
>>> x
xnd([[[1.0, 2.0], [3.0, 4.0]], [[5.0, 6.0], [7.0, 8.0]], [[9.0, 10.0], [11.0, 12.0]]],
```

(continues on next page)

(continued from previous page)

```
    type='3 * SomeMatrix(2 * 2 * float32)')
>>> x[0]
xnd([[1.0, 2.0], [3.0, 4.0]], type='SomeMatrix(2 * 2 * float32)')
```

Categorical

Categorical types contain values. The data stored in xnd buffers are indices (`int64`) into the type's categories.

```
>>> t = ndt("categorical('a', 'b', 'c', NA)")
>>> data = ['a', 'a', 'b', 'a', 'a', 'a', 'foo', 'c']
>>> x = xnd(data, dtype=t)
>>> x.value
['a', 'a', 'b', 'a', 'a', 'a', None, 'c']
```

Fixed String

Fixed strings are embedded into arrays. Supported encodings are 'ascii', 'utf8', 'utf16' and 'utf32'. The string size argument denotes the number of code points rather than bytes.

```
>>> t = ndt("10 * fixed_string(3, 'utf32')")
>>> x = xnd.empty(t)
>>> x.value
['', '', '', '', '', '', '', '', '', '']
>>> x[3] = "\U000003B1\U000003B2\U000003B3"
>>> x.value
['', '', '', 'αβγ', '', '', '', '', '', '']
```

Fixed Bytes

Fixed bytes are embedded into arrays.

```
>>> t = ndt("3 * fixed_bytes(size=3)")
>>> x = xnd.empty(t)
>>> x[2] = b'123'
>>> x.value
[b'\x00\x00\x00', b'\x00\x00\x00', b'123']
>>> x.align
1
```

Alignment can be requested with the requirement that size is a multiple of alignment:

```
>>> t = ndt("3 * fixed_bytes(size=32, align=16)")
>>> x = xnd.empty(t)
>>> x.align
16
```

String

Strings are pointers to `NUL`-terminated UTF-8 strings.

```
>>> x = xnd.empty("10 * string")
>>> x.value
['', '', '', '', '', '', '', '', '', '']
>>> x[0] = "abc"
>>> x.value
['abc', '', '', '', '', '', '', '', '', '']
```

Bytes

Internally, bytes are structs with a size field and a pointer to the data.

```
>>> xnd([b'123', b'45678'])
xnd([b'123', b'45678'], type='2 * bytes')
```

The bytes constructor takes an optional *align* argument that specifies the alignment of the allocated data:

```
>>> x = xnd([b'abc', b'123'], type="2 * bytes(align=64)")
>>> x.value
[b'abc', b'123']
>>> x.align
8
```

Note that *x.align* is the alignment of the array. The embedded pointers to the bytes data are aligned at *64*.

Primitive types

As a short example, here is a tuple that contains all primitive types:

```
>>> s = """
...    (bool,
...     int8, int16, int32, int64,
...     uint8, uint16, uint32, uint64,
...     bfloat16, float16, float32, float64,
...     bcomplex32, complex32, complex64, complex128)
... """
>>> x = xnd.empty(s)
>>> x.value
(False, 0, 0, 0, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0j, 0j, 0j, 0j)
```

2.1.2 Alignment and packing

The xnd memory allocators support explicit alignment. Alignment is specified in the types.

Tuples and records

Tuples and records have the *align* and *pack* keywords that have the same purpose as gcc's *aligned* and *packed* struct attributes.

Field alignment

The *align* keyword can be used to specify an alignment that is greater than the natural alignment of a field:

```
>>> from xnd import *
>>> s = "(uint8, uint64 |align=32|, uint64)"
>>> x = xnd.empty(s)
>>> x.align
32
>>> x.type.datasize
64
```

Field packing

The *pack* keyword can be used to specify an alignment that is smaller than the natural alignment of a field:

```
>>> s = "(uint8, uint64 |pack=2|, uint64)"
>>> x = xnd.empty(s)
>>> x.align
8
>>> x.type.datasize
24
```

Struct packing

The *pack* and *align* keywords can be applied to the entire struct:

```
>>> s = "(uint8, uint64, uint64, pack=1)"
>>> x = xnd.empty(s)
>>> x.align
1
>>> x.type.datasize
17
```

Individual field and struct directives are mutually exclusive:

```
>>> s = "2 * (uint8 |align=16|, uint64, pack=1)"
>>> x = xnd.empty(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot have 'pack' tuple attribute and field attributes
```

Array alignment

An array has the same alignment as its elements:

```
>>> s = "2 * (uint8, uint64, pack=1)"
>>> x = xnd.empty(s)
>>> x.align
1
>>> x.type.datasize
18
```

2.1.3 Buffer protocol

xnd supports importing PEP-3118 buffers.

From NumPy

Import a simple ndarray:

```
>>> import numpy as np
>>> from xnd import *
>>> x = np.array([[[0,1,2], [3,4,5]], [[6,7,8], [9,10,11]]])
>>> y = xnd.from_buffer(x)
>>> y.type
ndt("2 * 2 * 3 * int64")
>>> y.value
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
```

Import an ndarray with a struct dtype:

```
>>> x = np.array([(1000, 400.25, 'abc'), (-23, -1e10, 'cba')],
...              dtype=[('x', '<i4'), ('y', '>f4'), ('z', 'S3')])
>>> y = xnd.from_buffer(x)
>>> y.type
ndt("2 * {x : int32, y : >float32, z : fixed_bytes(size=3)}")
>>> y.value
[{'x': 1000, 'y': 400.25, 'z': b'abc'}, {'x': -23, 'y': -10000000000.0, 'z': b'cba'}]
```

2.1.4 Quick Start

Prerequisites

Python2 is not supported. If not already present, install the Python3 development packages:

```
# Debian, Ubuntu:
sudo apt-get install gcc make
sudo apt-get install python3-dev

# Fedora, RedHat:
sudo yum install gcc make
sudo yum install python3-devel

# openSUSE:
sudo zypper install gcc make
sudo zypper install python3-devel

# BSD:
# You know what to do.

# Mac OS X:
# Install Xcode and Python 3 headers.
```

Install

If pip is present on the system, installation should be as easy as:

```
pip install xnd
```

Otherwise:

```
tar xvzf xnd.2.0b1.tar.gz
cd xnd.2.0b1
python3 setup.py install
```

Windows

Refer to the instructions in the *vcbuild* directory in the source distribution.

CHAPTER 3

gumath

gumath is a package for extensible dispatch of computational kernels that target xnd containers. Kernels can be added at runtime, which allows the use of JIT compilers.

3.1 Libgumath

C library.

3.1.1 libgumath

libgumath is a library for dispatching computational kernels using ndtypes function signatures. Kernels are multimethods and can be JIT-generated and inserted in lookup tables at runtime.

Kernels target XND containers.

libgumath has a small number of generic math library kernels.

Data structures

libgumath is a lightweight library for managing and dispatching computational kernels that target XND containers.

Functions are multimethods in a lookup table. Typically, applications that use libgumath should create a new lookup table for each namespace. For example, Python modules generally should have a module-specific lookup table.

Kernel signatures

```
typedef int (* gm_xnd_kernel_t)(xnd_t stack[], ndt_context_t *ctx);
```

The signature of an *xnd* kernel. *stack* contains incoming and outgoing arguments. In case of an error, kernels are expected to set a context error message and return *-1*.

In case of success, the return value is *0*.

```
typedef int (* gm_strided_kernel_t)(char **args, intptr_t *dimensions, intptr_t␣
↪*steps, void *data);
```

The signature of a NumPy compatible kernel. These signatures are for applications that want to use existing NumPy compatible kernels on XND containers.

XND containers are automatically converted to a temporary ndarray before kernel application.

Kernel set

```
/* Collection of specialized kernels for a single function signature. */
typedef struct {
   ndt_t *sig;
   const ndt_constraint_t *constraint;

   /* Xnd signatures */
   gm_xnd_kernel_t C;       /* dispatch ensures c-contiguous */
   gm_xnd_kernel_t Fortran; /* dispatch ensures f-contiguous */
   gm_xnd_kernel_t Xnd;     /* selected if non-contiguous or both C and Fortran are␣
↪NULL */

   /* NumPy signature */
   gm_strided_kernel_t Strided;
} gm_kernel_set_t;
```

A kernel set contains the function signature, an optional constraint function, and up to four specialized kernels, each of which may be *NULL*.

The dispatch calls the kernels in the following order of preference:

If the inner dimensions of the incoming arguments are C-contiguous, the *C* kernel is called first. In case of *Fortran* inner dimensions, *Fortran* is called first.

If an *Xnd* kernel is present, it is called next, then the *Strided* kernel.

Kernel set initialization

```
typedef struct {
   const char *name;
   const char *sig;
   const ndt_constraint_t *constraint;

   gm_xnd_kernel_t C;
   gm_xnd_kernel_t Fortran;
   gm_xnd_kernel_t Xnd;
   gm_strided_kernel_t Strided;
} gm_kernel_init_t;

int gm_add_kernel(gm_tbl_t *tbl, const gm_kernel_init_t *kernel, ndt_context_t *ctx);
```

The *gm_kernel_init_t* is used for initializing a kernel set. Usually, a C translation unit contains an array of hundreds of *gm_kernel_init_t* structs together with a function that initializes a specific lookup table.

Multimethod struct

```
/* Multimethod with associated kernels */
typedef struct gm_func gm_func_t;
typedef const gm_kernel_set_t *(*gm_typecheck_t)(ndt_apply_spec_t *spec,
                  const gm_func_t *f, const ndt_t *in[], int nin, ndt_context_t *ctx);
struct gm_func {
   char *name;
   gm_typecheck_t typecheck; /* Experimental optimized type-checking, may be NULL. */
   int nkernels;
   gm_kernel_set_t kernels[GM_MAX_KERNELS];
};
```

This is the multimethod struct for a given function name. Each multimethod has a *nkernels* associated kernel sets with unique type signatures.

If *typecheck* is *NULL*, the generic libndtypes multimethod dispatch is used to locate the kernel. This is an O(N) operation, whose search time is negligible for large array operations.

The *typecheck* field can be set to an optimized lookup function that has internal knowledge of kernel set locations. The only restriction to the function is that it must behave exactly as the generic libndtypes typecheck.

Functions

Create a new multimethod

```
gm_func_t *gm_add_func(gm_tbl_t *tbl, const char *name, ndt_context_t *ctx);
```

Add a new multimethod with no associated kernels to a lookup table. If *name* is already present in *tbl* or if *name* contains invalid characters, return *NULL* and set an error.

On success, return the pointer to the new multimethod. The multimethod belongs to *tbl*.

Add a kernel to a multimethod

```
int gm_add_kernel(gm_tbl_t *tbl, const gm_kernel_init_t *kernel, ndt_context_t *ctx);
```

Add a kernel set to a multimethod. For convenience the multimethod is created and inserted into the table if not already present.

```
int gm_add_kernel_typecheck(gm_tbl_t *tbl, const gm_kernel_init_t *kernel, ndt_
↪context_t *ctx, gm_typecheck_t f);
```

Add a kernel set to a multimethod, using a custom typecheck function. For convenience, the multimethod is created and inserted into the table if not already present.

Select a kernel based on the input types

```
gm_kernel_t gm_select(ndt_apply_spec_t *spec, const gm_tbl_t *tbl, const char *name,
                      const ndt_t *in_types[], int nin, const xnd_t args[],
                      ndt_context_t *ctx);
```

The function looks up a multimethod by *name*, using table *tbl*. If the multimethod has an optimized custom typecheck function, it is called on the input types for kernel selection.

Otherwise, the generic *ndt_typecheck* is called on each kernel associated with the multimethod in order to find a match for the input arguments.

Apply a kernel to input

```
int gm_apply(const gm_kernel_t *kernel, xnd_t stack[], int outer_dims, ndt_context_t␣
↪*ctx);
```

Apply a kernel to input arguments. *stack* is expected to contain a list of input arguments followed by output arguments. *outer_dims* are the number of dimensions to traverse before applying the kernel to the inner dimensions.

Builtin kernels

libgumath has a number of builtin kernels that use optimized type checking and kernel lookup.

Unary kernels

```
int gm_init_unary_kernels(gm_tbl_t *tbl, ndt_context_t *ctx);
```

Add all builtin unary kernels to *tbl*. The kernels include *fabs*, *exp*, *exp2*, *expm1*, *log*, *log2*, *log10*, *log1p*, *logb*, *sqrt*, *cbrt*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *sinh*, *cosh*, *tanh*, *asinh*, *acosh*, *atanh*, *erf*, *erfc*, *lgamma*, *tgamma*, *ceil*, *floor*, *trunc*, *round*, *rearbyint*.

Binary kernels

```
int gm_init_binary_kernels(gm_tbl_t *tbl, ndt_context_t *ctx);
```

Add all binary kernels to *tbl*. The kernels currently only include *add*, *subtract*, *multiply*, *divide*.

3.2 Gumath

Python bindings for libgumath.

3.2.1 gumath

The gumath Python module provides the infrastructure for managing and dispatching libgumath kernels. Kernels target xnd containers from the xnd Python module.

gumath supports modular namespaces. Typically, a namespace is implemented as one Python module that uses gumath for calling kernels.

The xndtools project automates generating kernels and creating namespace modules.

Builtin functions

The gumath.functions module wraps the builtin libgumath kernels and serves as an example of a modular namespace.

All builtin functions

```
>>> from gumath import functions as fn
>>> dir(fn)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
↪'acosh', 'add', 'asin', 'asinh', 'atan', 'atanh', 'bitwise_and', 'bitwise_or',
↪'bitwise_xor', 'cbrt', 'ceil', 'copy', 'cos', 'cosh', 'divide', 'erf', 'erfc', 'exp
↪', 'exp2', 'expm1', 'fabs', 'floor', 'greater', 'greater_equal', 'invert', 'less',
↪'less_equal', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'logb', 'multiply',
↪'nearbyint', 'negative', 'round', 'sin', 'sinh', 'sqrt', 'subtract', 'tan', 'tanh',
↪'tgamma', 'trunc']
```

Unary functions

```
>>> from xnd import xnd
>>> x = xnd([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
>>> fn.log(x)
xnd([[0.0, 0.6931471805599453, 1.0986122886681098], [1.3862943611198906, 1.
↪6094379124341003, 1.791759469228055]],
    type='2 * 3 * float64')
```

On an array with a *float64* dtype, *log* works as expected.

```
>>> x = xnd([[1, 2, 3], [4, 5, 6]])
>>> x.type
ndt("2 * 3 * int64")
>>> fn.log(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid dtype
```

This function call would require an implicit inexact conversion from *int64* to *float64*. All builtin libgumath kernels only allow exact conversions, so the example fails.

```
>>> x = xnd([[1, 2, 3], [4, 5, 6]], dtype="int32")
>>> fn.log(x)
xnd([[0.0, 0.6931471805599453, 1.0986122886681098], [1.3862943611198906, 1.
↪6094379124341003, 1.791759469228055]],
    type='2 * 3 * float64')
```

int32 to *float64* conversions are exact, so the call succeeds.

3.3 XndTools

XndTools is a Python package containing development tools for the XND project.

3.3.1 xndtools

xndtools is a Python package containing development tools for the XND project.

kernel_generator

Python package for generating gumath kernels from C function prototypes.

Kernel generator

kernel_generator is a Python package for generating gumath kernels from C function prototypes.

Notes

Developer notes

Notes

Developer notes on xnd project.

Representation of XND objects

The XND project aims at providing a set of C libraries for storing data (libxnd), for describing the data (libndtypes), and for manipulating the data (libgumath). Each library is developed separately, however, libxnd requires libndtypes, and libgumath depends on both libxnd and libndtypes. In this note libxnd and libndtypes are discussed, in particular, how xnd objects are represented in memory and what are the relations between various members of data structures.

Embedding data in memory

In libxnd the data is stored using the following data strucure:

```
xnd_master_t
  flags : uint32_t
  master : xnd_t

xnd_t
  bitmap : xnd_bitmap_t
  index : int64_t
  type : ndt_t*
  ptr : char*

xnd_bitmap_t
  data : uint8_t*
```

(continues on next page)

(continued from previous page)

```
  size : int64_t
  next : xnd_bitmap_t*

ndt_bytes_t
  size : int64_t
  data : uint8_t*
```

where * represents pointer values of the corresponding data types. The corresponding C *typedef* definitions are in *libxnd/xnd.h*.

A short decription of data type members is given in the following table:

Member	Description
`xnd_master_t::flags`	Contains ownership information about the `data`, `type`, and `ptr` members. Needed for memory management.
`xnd_master_t::master`	An xnd view of data.
`xnd_t::bitmap`	Implements optional value support.
`xnd_t::index`	Linear index of data items. Used only when `type->tag` is `FixedDim|VarDim` (data is an array of items).
`xnd_t::type`	Points to a type value (`ndt_t` is provided by libndtypes, see below).
`xnd_t::ptr`	Points to a computer memory where data is embedded. Data can be stored as bytes (`ptr` points to `ndt_bytes_t` value) or referred by its pointer value (`ptr` points to data value pointer). In the case of arrays and `type->ndim==0`, `ptr` points to the data item given by `index`.
`xnd_bitmap_t::data`	Points to a bitmap data. Each data item (in `ptr`) has the corresponding bit value in `data`. Bit value `0` means that data item value has been provided.
`xnd_bitmap_t::size`	Number of subtree bitmaps. This is nonzero when `type->tag` is `Tuple|Record|Ref|Constr|Nominal`.
`xnd_bitmap_t::next`	Refers to bitmaps of subtrees. Used when data has tree-like structure.
`ndt_bytes_t::size`	Number of bytes needed to store data.
`ndt_bytes_t::data`	Points to computer memory where data is stored as bytes.

Note that the linear index is varied during iterations.

Typing the data

Typing the data means attaching a meaning to a junk of data stored in computer memory. The libndtypes implements data types using ndtypes type language (Blaze datashape) that combines the shape and element type information in one unit. Note that ndtypes is similar to the Python based implementation of Blaze datashape but there are several differences to make these distinct.

Ndtypes uses the following data structure:

```
ndt_t
  tag : enum ndt {Module, Function, AnyKind, ..., Typevar}
  access : enum ndt_access {Abstract, Concrete}
  flags : uint32_t
  ndim : int
  datasize : int64_t
  align : uint16_t

  // Abstract part
  union
    Module
```

(continues on next page)

(continued from previous page)

```
      name : char*
      type : ndt_t*
    Function
      nin : int64_t
      nout : int64_t
      nargs : int64_t
      types : ndt_t**
    FixedDim | SymbolicDim | EllipsisDim | Constr
      shape : int64_t
      type : ndt_t*
    VarDim | Ref
      type : ndt_t*
    Tuple
      flag : enum ndt_variadic {Nonvariadic, Variadic}
      shape : int64_t
      types : ndt_t**
    Record
      flag : enum ndt_variadic
      shape : int64_t
      names : char**
      types : ndt_t**

  Concrete
    union
      FixedDim
        itemsize : int64_t
        step : int64_t
      VarDim
        flag : enum ndt_offsets {InternalOffsets, ExternalOffsets}
        itemsize : int64_t
        noffsets : int32_t
        offsets : int32_t*
        nslices : int
        slices : ndt_slice_t*;
      Tuple | Record
        offset : int64_t*
        align : uint16_t*
        pad : uint16_t*
      Nominal
        name : char*
        type : ndt_t*
        meth : ndt_methods_t*
      Categorical
        ntypes : int64_t
        types : ndt_value_t*
      FixedString
        size : int64_t
        encoding : enum ndt_encoding {Ascii, Utf8, Utf16, Utf32, Ucs2}
      FixedBytes
        size : int64_t
        align : uint16_t
      Bytes
        align : uint16_t
      Char
        encoding : enum ndt_encoding
      Typevar
        name : char*
```

While the definition of `ndt_t` looks long, the union parts share the same memory and the interpretation of this depends on the ndtypes kind (specified by `ndt_t::tag` member).

Note that `ndt_t` holds both the shape and item type information of the xnd view object.

The ndtypes implementation `ndt_t` can be used in two modes, *abstract* or *concrete*, specified by `ndt_t::access` member. The ndtypes is in concrete mode when it contains enough information needed to compute what is the contiguous memory size (`datasize`) that fits the first and last item of the data. Otherwise the ndtypes is in abstract mode.

The abstract ndtypes can be used only as patterns. The concrete ndtypes can be used as patterns as well as for describing the structure of a data stored in a xnd view object (`xnd_t` instance).

Here follows a summary of data type members:

Member	Description
`xnd_t::tag`	Specifies ndtypes kind.
`xnd_t::access`	Specifies ndtypes mode, abstract or concrete.
`xnd_t::flags`	Contains various information about the data type: endianess, optional, subtree, ellipses.
`xnd_t::ndim`	Specifies dimension index. Ndtypes with `ndim==0` is interpreted as the ndtypes of a scalar value.
`xnd_t::datasize`	Size of data item in bytes [undefined in abstract mode]
`xnd_t::align`	Alignemnt of data in bytes [undefined in abstract mode]
`xnd_t::Module`	Abstract part of `Module` type kind.
`...`	...
`xnd_t::Concrete::FixedDim`	Concrete part of `FixedDim` type kind.
`...`	...

In the following each ndtypes kind is described in separate subsections.

Arrays

Array is a data structure that contains data items of the same data type. When data items use the same amount of memory, representation of an array is particularly simple: one only needs to know the location of the first data item in memory and the byte-size of data item type in order to have access to any data item in the array. However, there exists data types such as strings or ragged arrayswhere the data item byte-size depends on the data item content and a more general representation of array structure is needed.

In libndtypes several kinds of array representations are supported.

In abstract mode one can represent arrays of different item types (named type variable), of different dimensions (ellipses), and of different shapes (symbolic dimensions) using a single ndtypes instance. The purpose of such ndtypes instances is to define patterns of arrays that is used in libgumath. The libgumath library provides computational kernels that implement algorithms to manipulate data with specific structure. The kernels can be called only on data that ndtypes matches the signature of a particular kernel.

In concrete mode the main purpose of a ndtypes instance is to provide information how to access the data items in an array (using also the member `xnd_t::index`).

Abstract array ndtypes

Ndtypes is in abstract mode when `ndt_t::access==Abstract`.

Member	Description
`ndt_t::tag==FixedDim`	Ndtypes represents an array dimension with fixed shape value
`ndt_t::FixedDim::shape`	Specifies the shape value of the array dimension
`ndt_t::FixedDim::type`	Points to data item type specification.
`xnd_t::datasize`	undefined
`xnd_t::align`	undefined
`xnd_t::Concrete::...`	undefined

Undefined means that the value is set to `0` or `NULL`.

Note that `FixedDim` ndtypes is in abstract mode when `type` ndtypes is in abstract mode.

Member	Description
`ndt_t::tag==VarDim`	Ndtypes represents an array dimension that shape may vary
`ndt_t::VarDim::type`	Points to data item type specification.
`xnd_t::datasize`	undefined
`xnd_t::align`	undefined
`xnd_t::Concrete::...`	undefined

Member	Description
`ndt_t::tag==SymbolicDim`	Ndtypes represents an array dimension that shape is a symbolic.
`ndt_t::SymbolicDim::name`	Contains symbol name.
`ndt_t::SymbolicDim::type`	Points to data item type specification.
`xnd_t::datasize`	undefined
`xnd_t::align`	undefined
`xnd_t::Concrete::...`	undefined

The shape symbol must start with a capital letter.

Member	Description
`ndt_t::tag==EllipsisDim`	Ndtypes represents 0 or more dimensions.
`ndt_t::EllipsisDim::name`	Contains ellipsis name.
`ndt_t::EllipsisDim::type`	Points to data item type specification.
`xnd_t::datasize`	undefined
`xnd_t::align`	undefined
`xnd_t::Concrete::...`	undefined

The named ellipsis name must start with a capital letter or be equal to `'var'`. Ellipsis `var...` is special and it is used only for ragged arrays (when `ndt_t::tag==VarDim`).

Concrete array ndtypes

Ndtypes is in concrete mode when `ndt_t::access==Concrete`.

In the case of fixed shape arrays we have:

Member	Description
`ndt_t::tag==FixedDim`	Ndtypes represents an array dimension with fixed shape value
`ndt_t::FixedDim::shape`	Specifies the shape value of the array dimension
`ndt_t::FixedDim::type`	Points to data item type specification.
`xnd_t::datasize`	Specifies the byte-size of array data
`xnd_t::align`	Alignment of data item.
`xnd_t::Concrete::FixedDim::itemsize`	Specifies the byte-size of array item.
`xnd_t::Concrete::FixedDim::step`	Specifies the byte-step to the next item in array. E.g. in the case of slice view the step is a integer multiple of itemsize.

Note that the `datasize` is defined as the byte-size of memory that is occupied between the first and the last array item (including the items that are discarded due to slicing with `step!=1`). So, `datasize` does not correspond to the size of memory needed to hold the data defined by xnd view, it only defines the upper bound.

In the case of ragged arrays we have:

Member	Description
`ndt_t::tag==VarDim`	Ndtypes represents an array dimension that shape may vary
`ndt_t::FixedDim::type`	Points to data item type specification.
`xnd_t::datasize`	Specifies the byte-size of array data
`xnd_t::align`	Alignment of data item.
`xnd_t::Concrete::VarDim::flag`	Specifies the ownership of offsets data.
`xnd_t::Concrete::VarDim::itemsize`	Specifies the byte-size of array item.
`xnd_t::Concrete::VarDim::noffsets`	Specifies the byte-size of `offsets` member.
`xnd_t::Concrete::VarDim::offsets`	
`xnd_t::Concrete::VarDim::nslices`	Specifies the byte-size of `slices` member.
`xnd_t::Concrete::VarDim::slices`	

Tutorials

Tutorials on using XND tools.

Generating new kernels for gumath

Operations on XND containers live in the gumath library. It already provides binary operations like add, subtract, multiply and divide, and unary operations like abs, exponential, logarithm, power, trigonometric functions, etc. They can operate on various data types, e.g. `int32` or `float64`. But you might want to create your own kernel, either from a code that you wrote or from an existing library. This tutorial will show how to do that using the kernel generator from XND Tools.

Kernel generator

XND Tools provide development tools for XND. Among them, `xndtools.kernel_generator` facilitates the creation of new kernels by semi-automatically wrapping e.g. C code. Fortran will be supported in the future, as there are tons of high performance libraries out there.

Wrapping C code

Let's say we want to make a kernel out of the following C function, which just squares a number:

```
// file: square.c

#include "square.h"

double square(double a)
{
    return a * a;
}
```

This is the implementation of the function that we want to turn into a kernel, but the kernel generator is only concerned about its prototype, which looks like this:

```
// file: square.h

extern double square(double);
```

Note the `extern` keyword: because this header file will be used by the kernel generator to build the kernel, the `square` function will be assumed to be available somewhere else. We will then be able to compile the kernel and the `square.c` file independently, and link them later. This is not so important in our simple example, but it could be if we were wrapping an existing library like LAPACK, which has its own build process. The kernel generator doesn't need to know how to build the library, all it needs is a header file with the function prototypes.

Now run the following command:

```
$ xnd_tools config square.h
```

This creates an initial kernel configuration file `square-kernels.cfg`, which looks like this:

```
[MODULE square]
typemaps =
      double: float64
includes =
      square.h
include_dirs =

libraries =

library_dirs =

header_code =
kinds = Xnd
ellipses = ..., var...

[KERNEL square]
skip = # REMOVE THIS LINE WHEN READY
prototypes =
      double square(double   a);
description =
dimension =
input_arguments = a
inplace_arguments =
inout_arguments =
output_arguments =
hide_arguments =
```

For this simple kernel, we don't actually have to change anything, so we'll just remove the `skip` line as indicated, and save the file.

The next step is about creating the interface to `gumath`, and registering our `square` function as a kernel. The following command will create a `square-kernels.c` file:

```
$ xnd_tools kernel square-kernels.cfg
```

For now we are still in the C world, so we also need to expose our kernel to Python. This is done by creating an extension module. Fortunately, XND tools does that for us as well. The following command will create the `square-python.c` file. Note that it also creates the `square-kernels.c` file if it does not already exists, so the previous command is not necessary here.

```
$ xnd_tools module square-kernels.cfg
```

Assuming the variable `$SITE_PACKAGES` contains the path to your Python `site-packages` directory, where `xnd`, `ndtypes`, `gumath` and `xndtools` are installed (given by `python -c "from distutils.sysconfig import get_python_lib; print(get_python_lib())"`), you can compile the square function, its kernel, and create a static library with the following commands:

```
$ gcc -fPIC                                       \
  -c square.c                                     \
  -c square-kernels.c -fPIC                       \
  -I$SITE_PACKAGES/ndtypes                        \
  -I$SITE_PACKAGES/xnd                            \
  -I$SITE_PACKAGES/gumath                         \
  -I$SITE_PACKAGES/xndtools/kernel_generator
$ ar rcs libsquare-kernels.a square-kernels.o square.o
```

Then building a C extension for CPython can be done using `distutils`. It just needs a `setup.py` script, which for our simple case looks like this:

```
# file: setup.py

from distutils.core import setup, Extension
from distutils.sysconfig import get_python_lib

site_packages = get_python_lib()
libs = ['ndtypes','gumath', 'xnd']
lib_dirs = [f'{site_packages}/{lib}' for lib in libs]

module1 = Extension('square',
                    include_dirs = lib_dirs,
                    libraries = ['square-kernels'] + libs,
                    library_dirs = ['.'] + lib_dirs,
                    sources = ['square-python.c'])

setup (name = 'square',
       version = '1.0',
       description = 'This is a gumath kernel extension that squares an XND container
↪',
       ext_modules = [module1])
```

Finally, we can build and install our extension with the following command:

```
$ python setup.py install
```

If everything went fine, we can now test it in the Python console:

```
>>> from xnd import xnd
>>> from square import square
>>> a = xnd([1., 2., 3.])
>>> a
xnd([1.0, 2.0, 3.0], type='3 * float64')
>>> square(a)
xnd([1.0, 4.0, 9.0], type='3 * float64')
```

Running kernels on the GPU

We will see how we can run kernels on the GPU. The following is a typical CUDA code which adds the elements of two arrays of a given size:

```
// file: add_gpu.cu

#include "gpu.h"

__global__
void add(int n, float* x, float* y, float* r)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        r[i] = x[i] + y[i];
}

void add_gpu(int n, float* x, float* y, float* r)
{
    int blockSize = 256;
    int numBlocks = (n + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>(n, x, y, r);
    cudaDeviceSynchronize();
}
```

The `add` function is called a CUDA kernel (not to be confused with the `gumath` kernels!). This is what will actually run on the GPU. The reason why a GPU is faster than a CPU is because it can massively parallelize computations, and this is why we have these `index` and `stride` variables: the kernel will be applied on different parts of the data at the same time.

Our `gumath` kernel however will use the `add_gpu` function, which internally calls `add` with a special CUDA syntax and some extra-parameters (basically specifying how much it will be parallelized). The `cudaDeviceSynchronize()` function call blocks the CPU execution until all GPU computations are done.

The GPU has its own memory, which is different from the CPU memory. When we want to do a computation on the GPU, we first have to copy the data from the CPU side to the GPU side, and when we want to retrieve the results from the GPU, we have to copy its data back to the CPU. This can be taken care of by the so called "unified memory", which provides a single memory space accessible by the GPU and the CPU. The following file contains functions to allocate memory in the unified memory and to delete it, here again through special CUDA functions:

```
// file: gpu.cu

#include "gpu.h"

float* get_array(int n)
{
```

(continues on next page)

(continued from previous page)

```
    float* x;
    cudaMallocManaged(&x, n * sizeof(float));
    return x;
}

void del_array(float* x)
{
    cudaFree(x);
}
```

Now let's see how the `gpu.h` file looks like:

```
// file: gpu.h

extern "C" void add_gpu(int n, float* x, float* y, float *r);
extern "C" float* get_array(int n);
extern "C" void del_array(float* x);
```

It consists of the prototypes of the `add_gpu` function for which we want to make a kernel, and the `get_array` and `del_array` functions which we will use to manage the memory for our data. Note the `extern "C"` declaration: because `nvcc` (the CUDA compiler) is a C++ compiler, we need to expose them as C functions to Python.

Since `gpu.cu` only manages the GPU memory, and is independent of the `gumath` kernel generation, we will simply access its functions through a shared library. This is how we compile it:

```
$ nvcc -o libgpu.so --compiler-options "-fPIC" --shared gpu.cu
```

This gives us a `libgpu.so` library that we can interface with in Python using `ctypes`. The following code wraps the C functions to Python functions:

```
# file: gpu.py

import ctypes
import numpy as np
from xnd import xnd

gpu = ctypes.CDLL('./libgpu.so')
gpu.get_array.restype = ctypes.POINTER(ctypes.c_float)
gpu.del_array.argtypes = [ctypes.POINTER(ctypes.c_float), ]

def xnd_gpu(size):
    addr = gpu.get_array(size)
    a = np.ctypeslib.as_array(addr, shape=(size,))
    x = xnd.from_buffer(a)
    return x, addr

def del_gpu(addr):
    gpu.del_array(addr)
```

`xnd_gpu` returns an XND container (and its data pointer) whose data live in the unified memory, and `del_gpu` frees the memory referenced by a pointer.

Now we need to generate the `gumath` kernel for our `add_gpu` function. We save its prototype in the following file:

```
// file: add_gpu.h

extern void add_gpu(int n, float* x, float* y, float *r);
```

The corresponding configuration file looks like this:

```
# file: add_gpu-kernels.cfg

[MODULE add_gpu]
typemaps =
     float: float32
     int: int32
includes =
     add_gpu.h
include_dirs =
sources =
     add_gpu.c

libraries =

library_dirs =

header_code =
kinds = C
ellipses = none

[KERNEL add_gpu]
prototypes =
     void add_gpu(int   n, float *  x, float *  y, float *  r);
description =
dimension = x(n), y(n), r(n)
input_arguments = x, y
inplace_arguments = r
hide_arguments = n = len(x)
```

We can now generate the kernel:

```
$ xnd_tools kernel add_gpu-kernels.cfg
$ xnd_tools module add_gpu-kernels.cfg
```

And create a static library:

```
$ nvcc --compiler-options '-fPIC' -c add_gpu.cu
$ gcc -fPIC                                                \
  -c add_gpu-kernels.c                                     \
  -c $SITE_PACKAGES/xndtools/kernel_generator/xndtools.c  \
  -I$SITE_PACKAGES/xndtools/kernel_generator               \
  -I$SITE_PACKAGES/xnd                                     \
  -I$SITE_PACKAGES/ndtypes                                 \
  -I$SITE_PACKAGES/gumath
$ ar rcs libadd_gpu-kernels.a add_gpu.o add_gpu-kernels.o xndtools.o
```

Finally, launch `python setup.py install` with this `setup.py` file:

```
# file: setup.py

from distutils.core import setup, Extension
from distutils.sysconfig import get_python_lib

site_packages = get_python_lib()
lib_dirs = [f'{site_packages}/{i}' for i in ['ndtypes', 'gumath', 'xnd']]
```

(continues on next page)

(continued from previous page)

```
module1 = Extension('add_gpu',
                    include_dirs = lib_dirs,
                    libraries = ['add_gpu-kernels', 'ndtypes','gumath', 'xnd', 'cudart
↪', 'stdc++'],
                    library_dirs = ['.', '/usr/local/cuda-9.2/lib64'] + lib_dirs,
                    sources = ['add_gpu-python.c'])

setup (name = 'add_gpu',
       version = '1.0',
       description = 'This is a gumath kernel extension that adds two XND containers␣
↪on the GPU',
       ext_modules = [module1])
```

If everything went fine, you should be able to run the kernel on the GPU:

```
>>> from gpu import xnd_gpu, del_gpu
>>> from add_gpu import add_gpu
>>> size = 1 << 20
>>> x0, a0 = xnd_gpu(size)
>>> x1, a1 = xnd_gpu(size)
>>> x2, a2 = xnd_gpu(size)
>>> for i in range(size):
...     x0[i] = i
...     x1[i] = 1
>>> x0
xnd([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, ...], type='1048576 * float32')
>>> x1
xnd([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, ...], type='1048576 * float32')
>>> add_gpu(x0, x1, x2)
>>> x2
xnd([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, ...], type='1048576 * float32')
>>> del_gpu(a0)
>>> del_gpu(a1)
>>> del_gpu(a2)
```

3.4 Releases

3.4.1 Releases

v0.2.0b2 (February 5th 2018)

Second release (beta2). This release addresses several build and packaging issues:

- Avoid copying libraries into the Python package if system libraries are used.
- The build and install partially relied on the dev setup (ndtypes checked out in the xnd directory). This dependency has been removed.
- The conda build now supports separate library and Python module installs.
- Configure now has a **–without-docs** option for skipping the doc install.
- The generated parsers are now checked into the source tree to avoid bison/flex dependencies and unnecessary rebuilds after cloning.
- Non-API global symbols are hidden on Linux (as long as the compiler supports gcc pragmas).

- The conda build supports separate library and Python module installs.
- Configure now has a **–without-docs** option for skipping the doc install.

v0.2.0b1 (January 20th 2018)

First release (beta1).

Index