Extending Numba for HEP data types

Jim Pivarski

Princeton University – DIANA-HEP

April 23, 2018

Can we use Numba for LHC data?

Numba provides a smooth transition between high-level tinkering in Python and high-throughput processing, and we've seen a real-world demonstration in XENONnT.

Can we use it for the LHC?

- Must access ROOT data.
- We need arbitrary length lists of particles per event, not rectangular arrays.
- Converting all of our events to lists of namedtuples to feed to Numba would be a serious performance hit (memory and throughput).

Can we use Numba for LHC data?

Numba provides a smooth transition between high-level tinkering in Python and high-throughput processing, and we've seen a real-world demonstration in XENONnT.

Can we use it for the LHC?

- Must access ROOT data. Solution: uproot, as well as ROOT PR#943, PR#1872.
- We need arbitrary length lists of particles per event, not rectangular arrays.
- Converting all of our events to lists of namedtuples to feed to Numba would be a serious performance hit (memory and throughput).

Can we use Numba for LHC data?

Numba provides a smooth transition between high-level tinkering in Python and high-throughput processing, and we've seen a real-world demonstration in XENONnT.

Can we use it for the LHC?

- Must access ROOT data. Solution: uproot, as well as ROOT PR#943, PR#1872.
- We need arbitrary length lists of particles per event, not rectangular arrays.
- Converting all of our events to lists of namedtuples to feed to Numba would be a serious performance hit (memory and throughput).

  Solution: object-array mapping (OAMap).

Object-array mapping (OAMap)

By analogy with object-relational mapping (ORM), which translates between classes in object-oriented programming (OOP) and relational tables in databases.

OAMap translates between OOP objects and low-level, read-only arrays:

- → Physicists write arbitrary code, assuming OOP-style objects.
- → Actual data are stored in arrays (e.g. TBaskets), never deserialized into objects.
- → Physicist's function is translated into operations on arrays.

Object-array mapping (OAMap)

By analogy with object-relational mapping (ORM), which translates between classes in object-oriented programming (OOP) and relational tables in databases.

OAMap translates between OOP objects and low-level, read-only arrays:

- → Physicists write arbitrary code, assuming OOP-style objects.
- → Actual data are stored in arrays (e.g. TBaskets), never deserialized into objects.
- → Physicist's function is translated into operations on arrays.

This is a kind of compilation. Last year's talks about "Femtocode" are this idea in a functional language, but the data representation can be handled on its own.

Object-array mapping (OAMap)

By analogy with object-relational mapping (ORM), which translates between classes in object-oriented programming (OOP) and relational tables in databases.

OAMap translates between OOP objects and low-level, read-only arrays:

- → Physicists write arbitrary code, assuming OOP-style objects.
- → Actual data are stored in arrays (e.g. TBaskets), never deserialized into objects.
- → Physicist's function is translated into operations on arrays.

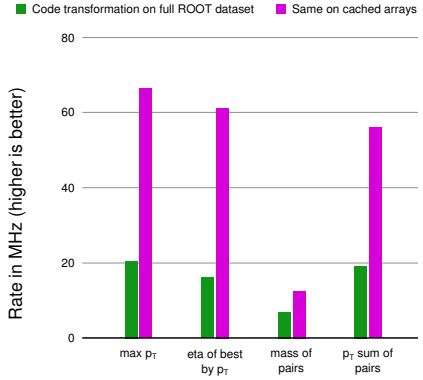
This is a kind of compilation. Last year's talks about "Femtocode" are this idea in a functional language, but the data representation can be handled on its own.

In particular, Python is a high-level language that physicists already use, and Numba provides optimized compilation through LLVM.

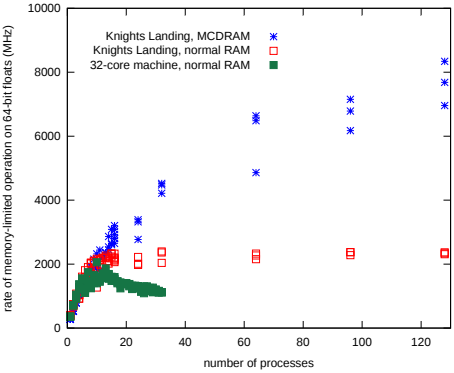
Performance tests *preceded* design

I wanted to make sure that committing to a library (Numba) did not give away performance from the start. I observed no loss in throughput compared to pure C.

Single-threaded rates of millions of events per second, limited only by trig functions!



Multi-threaded scaling limited only by physical memory bus bandwidth.



Moral

If we can deliver arrays to main memory and the CPU cache quickly enough,

. . . and physicists' analyses can be expressed as array operations on sequential data,

. . . then it can be compiled to run at this scale.

Moral

If we can deliver arrays to main memory and the CPU cache quickly enough,

. . . and physicists' analyses can be expressed as array operations on sequential data,

. . . then it can be compiled to run at this scale.

Instead of converting TBaskets into objects,
let's convert OOP code into array code!

“General enough” data types

1. Primitives:
2. Lists:
3. Unions:
4. Records:
5. Tuples:

6. Pointers:

The data types and their representations have compositional symmetry: any type can be plugged into any other type and the array representations follow suit.

“General enough” data types

1. Primitives: any fixed-width data, such as numbers.
2. Lists:
3. Unions:
4. Records:
5. Tuples:

6. Pointers:

The data types and their representations have compositional symmetry: any type can be plugged into any other type and the array representations follow suit.

“General enough” data types

1. Primitives: any fixed-width data, such as numbers.
2. Lists: arbitrary-length collections of data with a given type (homogeneous).
3. Unions:
4. Records:
5. Tuples:

6. Pointers:

The data types and their representations have compositional symmetry: any type can be plugged into any other type and the array representations follow suit.

“General enough” data types

1. Primitives: any fixed-width data, such as numbers.
2. Lists: arbitrary-length collections of data with a given type (homogeneous).
3. Unions: set of possible types for heterogeneity (e.g. list of “electrons *or* muons”).
4. Records:
5. Tuples:

6. Pointers:

The data types and their representations have compositional symmetry: any type can be plugged into any other type and the array representations follow suit.

"General enough" data types

1. Primitives: any fixed-width data, such as numbers.
2. Lists: arbitrary-length collections of data with a given type (homogeneous).
3. Unions: set of possible types for heterogeneity (e.g. list of "electrons *or* muons").
4. Records: objects containing a set of typed fields (a.k.a. classes or structs).
5. Tuples:

6. Pointers:

The data types and their representations have compositional symmetry: any type can be plugged into any other type and the array representations follow suit.

"General enough" data types

1. Primitives: any fixed-width data, such as numbers.
2. Lists: arbitrary-length collections of data with a given type (homogeneous).
3. Unions: set of possible types for heterogeneity (e.g. list of "electrons *or* muons").
4. Records: objects containing a set of typed fields (a.k.a. classes or structs).
5. Tuples: fixed-length collections of typed fields (like records with index positions instead of field names).
6. Pointers:

The data types and their representations have compositional symmetry: any type can be plugged into any other type and the array representations follow suit.

“General enough” data types

1. Primitives: any fixed-width data, such as numbers.
2. Lists: arbitrary-length collections of data with a given type (homogeneous).
3. Unions: set of possible types for heterogeneity (e.g. list of “electrons *or* muons”).
4. Records: objects containing a set of typed fields (a.k.a. classes or structs).
5. Tuples: fixed-length collections of typed fields (like records with index positions instead of field names).
6. Pointers: objects identified by position in another field. Intended for linking relationships among particles, but also useful for expressing skimmed data as event lists, making non-tree data structures, emulating Arrow/Parquet’s “dictionary encoding” of categorical strings...

The data types and their representations have compositional symmetry: any type can be plugged into any other type and the array representations follow suit.

Mapping between objects and arrays

Example: `List(List(Record({"x": "char", "y": "int"})))`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer list stops	`[ 3, 3, 4]`
inner list stops	`[ 4, 4, 6, 7 ]`
"x" attribute	`[ a, b, c, d, e, f, g ]`
"y" attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

Transformation rules:

- Primitive data at leaves of schema are stored contiguously, with no structure.
- List structure encoded in a separate "stops" array, computed from the cumulative number of entries at its level of depth, written at each closing bracket.
- Other schema types have similar rules.

Proxy objects only need to know where they are in the schema (compile-time) and where they are in their arrays (integer index at run-time).

Similarity to ROOT and Apache Arrow

OAMap transformation rules deliberately resemble those of ROOT and Arrow, so that both can be used as inputs to calculations. With minor modifications, both represent nested data as described on the previous page.

But unlike ROOT,

OAMap performs calculations on columns in memory. ROOT *stores* data in columns (TBranches of TBaskets).

But unlike Arrow,

OAMap performs arbitrary functions (even procedural code) on object-like data, rather than a data-frame model.

Unlike both, OAMap focuses only on *calculation.* Any software that provides arrays in the right format may be used as sources.

Example of ROOT data as an OAMap schema

```
>>> import uproot
>>> import oamap.source.root    # old ROOT interface

>>> url = "http://scikit-hep.org/uproot/examples/HZZ.root"
>>> events = uproot.open(url)["events"].oamap()
>>> events.schema.content["muons"].show()
List(
  starts = 'NMuon',     # schema maps object attributes to array names
  stops = 'NMuon',      # in this case, ROOT TBranch names
  content = Record(
    fields = {          # at all levels of nesting
      'px': Primitive(dtype('float32'), data='Muon_Px'),
      'py': Primitive(dtype('float32'), data='Muon_Py'),
      'pz': Primitive(dtype('float32'), data='Muon_Pz'),
      'energy': Primitive(dtype('float32'), data='Muon_E'),
      'charge': Primitive(dtype('int32'), data='Muon_Charge'),
      'isolation': Primitive(dtype('float32'), data='Muon_Iso')
    }))
```

Example use of OAMap proxies

The dataset "looks and feels" like a nested Python list.

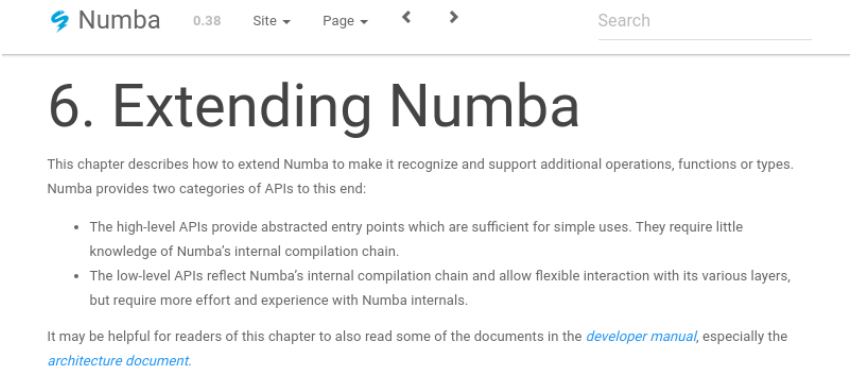
```
>>> events
[<Event at index 0>, <Event at index 1>, <Event at index 2>, ...,
 <Event at index 2418>, <Event at index 2419>, <Event at index 2420>]

>>> events[0].muons
[<Muon at index 0>, <Muon at index 1>]

>>> [x.px for x in events[0].muons]
[-52.899456, 37.73778]
```

But it is generated on demand from arrays. Here's what's loaded now:

```
"NMuon":   array([2, 1, 2, ..., 1, 1, 1], dtype=int32)
"Muon_Px": array([-52.899456,  37.73778,  -0.81645936, ...,
                  -29.756786, 1.1418698,  23.913206  ], dtype=float32)
```

Compilation with Numba

Originally, I was doing AST-to-AST transformations, replacing object references with array references, followed by calling Numba on the result. This required me to do my own type inference. Later, I embedded the objects-to-arrays transformation in the Numba compilation pass itself to gain integration with other Python types for free.


Numba 0.38 Site Page Search

6. Extending Numba

This chapter describes how to extend Numba to make it recognize and support additional operations, functions or types. Numba provides two categories of APIs to this end:

- The high-level APIs provide abstracted entry points which are sufficient for simple uses. They require little knowledge of Numba's internal compilation chain.
- The low-level APIs reflect Numba's internal compilation chain and allow flexible interaction with its various layers, but require more effort and experience with Numba internals.

It may be helpful for readers of this chapter to also read some of the documents in the *developer manual*, especially the *architecture document*.

Example of compiling a user function

```
>>> import numpy
>>> import numba
>>> import oamap.compiler

>>> @numba.njit                    # declares the following function to be compiled
... def compute(events, out):
...     i = 0
...     for event in events:       # "event" and "event.muons" are a compiler fiction
...         if len(event.muons) == 2:
...             mu1, mu2 = event.muons[0], event.muons[1]
...             px = mu1.px + mu2.px
...             py = mu1.py + mu2.py
...             pz = mu1.pz + mu2.pz
...             energy = mu1.energy + mu2.energy
...             out[i] = sqrt(energy**2 - px**2 - py**2 - pz**2)
...             i += 1

>>> out = numpy.empty(1371)
>>> compute(events, out)          # compilation and array-fetching happen on first call

>>> out
array([90.22780609, 74.74654388, 89.75765991, ..., 92.06494904,
       85.44384003, 75.96061707])
```

Inside the OAMap compiler: typing and implementing list brackets

```
@nb.typing.templates.infer
class ListProxyGetItem(nb.typing.templates.AbstractTemplate):
    key = "getitem"
    def generic(self, args, kwds):
       if len(args) == 2:
          tpe, idx = args
          if isinstance(tpe, ListProxyNumbaType) and isinstance(idx, nb.types.Integer):
             return typeof_generator(tpe.generator.content)(tpe, idx)

@nb.extending.lower_builtin("getitem", ListProxyNumbaType, nb.types.Integer)
def listproxy_getitem(context, builder, sig, args):
    listtpe, indextpe = sig.args; listval, indexval = args
    listproxy = nb.cgutils.create_struct_proxy(listtpe)(context, builder, listval)
    indexval = cast_int64(builder, indexval)

    normindex_ptr = nb.cgutils.alloca_once_value(builder, indexval)
    with builder.if_then(builder.icmp_signed("<", indexval, literal_int64(0))):
        builder.store(builder.add(indexval, listproxy.length), normindex_ptr)
    normindex = builder.load(normindex_ptr)

    at = builder.add(listproxy.whence, builder.mul(listproxy.stride, normindex))
    return generate(context, builder, listtpe.generator.content, listproxy.baggage,
                    listproxy.ptrs, listproxy.lens, at)
```

Conclusion: benefits of implementing OAMap in Numba

- Numba is a mature type-inferring Python compiler, feeding into LLVM.
- Numba's extension mechanism lets me write arbitrary logic to infer OAMap types and emit specialized LLVM bytecode.
- OAMap proxies can be passed into or out of any `@numba.jit`-compiled function, like any other Python data type.
- Compilation pass over all functions called by the entry function tells me exactly which arrays need to be loaded (selective reading from ROOT).
- Low-level access to fix performance bugs, should they arise.

Future of OAMap (why the ROOT interface doesn't work now)

1. OAMap objects are immutable, but users will want to edit them.

Added global operations:

```
new = define(old, "pz", lambda mu: mu.pt*sinh(mu.eta), at="muons")
```

to make a new version of the dataset in which every muon in every event now has a `pz`.

In `new`, all arrays associated with old data (e.g. `pt`, `eta`) come from the original ROOT files and `pz` comes from somewhere else (e.g. HDF5 files).

Future of OAMap (why the ROOT interface doesn't work now)

1. OAMap objects are immutable, but users will want to edit them.

Added global operations:

```
new = define(old, "pz", lambda mu: mu.pt*sinh(mu.eta), at="muons")
```

to make a new version of the dataset in which every muon in every event now has a `pz`.

In `new`, all arrays associated with old data (e.g. `pt`, `eta`) come from the original ROOT files and `pz` comes from somewhere else (e.g. HDF5 files).

2. We need to enable parallel and/or remote processing.

Added Spark-like handle to remote, distributed data:

```
db = DatabaseConnection("http://query-server.cern.ch")
db.data.new = db.data.old.filter(lambda event: len(event.muons) >= 2)
local_events = db.data.new[:10000]
```

The database maintains potentially overlapping schemas and assigning to `db.data` launches Dask tasks to distribute the work and save results in the database.