Awkward Array: Numba

Jim Pivarski

Princeton University – IRIS-HEP

April 17, 2019

Knowing your audience

I presented an "Accelerating Python" tutorial to non-particle physics scientists:

- 8 Computer Science/Software Engineering/Electrical Engineering
- 7 Physics/Astronomy/Energy Science/Atmospheric & Ocean Science
- 5 Finance/Business/Political Science
- 2 Neuroscience
- 2 Civil Engineering

Knowing your audience

I presented an "Accelerating Python" tutorial to non-particle physics scientists:

- 8 Computer Science/Software Engineering/Electrical Engineering
- 7 Physics/Astronomy/Energy Science/Atmospheric & Ocean Science
- 5 Finance/Business/Political Science
- 2 Neuroscience
- 2 Civil Engineering

I started by showing how for-loopy code must be fundamentally rewritten to take advantage of Numpy and why it might be worth the effort.

Knowing your audience

I presented an "Accelerating Python" tutorial to non-particle physics scientists:

- 8 Computer Science/Software Engineering/Electrical Engineering
- 7 Physics/Astronomy/Energy Science/Atmospheric & Ocean Science
- 5 Finance/Business/Political Science
- 2 Neuroscience
- 2 Civil Engineering

I started by showing how for-loopy code must be fundamentally rewritten to take advantage of Numpy and why it might be worth the effort.

Surprise! They were *more comfortable* with the vectorized form (Numpy/Pandas). Going the other way—from Numpy to for loops—was the novelty for them.

Knowing your audience

Regardless of which side of the divide you start from, event-at-a-time and operation-at-a-time approaches are rather different and have different advantages.

event-at-a-time

```
for event in everything:
    a = step1(event)
    b = step2(a)
    write_one(b)
```

operation-at-a-time

```
a = step1(everything)
b = step2(a)
write_all(b)
```

Knowing your audience

Regardless of which side of the divide you start from, event-at-a-time and operation-at-a-time approaches are rather different and have different advantages.

event-at-a-time

```
for event in everything:
    a = step1(event)
    b = step2(a)
    write_one(b)
```

operation-at-a-time

```
a = step1(everything)
b = step2(a)
write_all(b)
```

- Good for debugging: insert breakpoints, watch variables to understand a single event.

Knowing your audience

Regardless of which side of the divide you start from, event-at-a-time and operation-at-a-time approaches are rather different and have different advantages.

event-at-a-time

```
for event in everything:
    a = step1(event)
    b = step2(a)
    write_one(b)
```

operation-at-a-time

```
a = step1(everything)
b = step2(a)
write_all(b)
```

- Good for debugging: insert breakpoints, watch variables to understand a single event.
- Detail can obscure big picture.

Knowing your audience

Regardless of which side of the divide you start from, event-at-a-time and operation-at-a-time approaches are rather different and have different advantages.

event-at-a-time

```
for event in everything:
    a = step1(event)
    b = step2(a)
    write_one(b)
```

- Good for debugging: insert breakpoints, watch variables to understand a single event.
- Detail can obscure big picture.

operation-at-a-time

```
a = step1(everything)
b = step2(a)
write_all(b)
```

- Composition of functions can read like natural language.

Knowing your audience

Regardless of which side of the divide you start from, event-at-a-time and operation-at-a-time approaches are rather different and have different advantages.

event-at-a-time

```
for event in everything:
    a = step1(event)
    b = step2(a)
    write_one(b)
```

- Good for debugging: insert breakpoints, watch variables to understand a single event.
- Detail can obscure big picture.

operation-at-a-time

```
a = step1(everything)
b = step2(a)
write_all(b)
```

- Composition of functions can read like natural language.
- Indexes can be hard to align: "error driven development!"

Most talks on awkward-array (including this meeting) are about the value of introducing operation-at-a-time into particle physics.

This talk will be about getting event-at-a-time in Python without a speed penalty.

Most talks on awkward-array (including this meeting) are about the value of introducing operation-at-a-time into particle physics.

This talk will be about getting event-at-a-time in Python without a speed penalty.

Programming strategy should be a *separate question* from performance.

Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

Learn More | Try Numba »

Accelerate Python Functions

Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

You don't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. Just apply one of the Numba decorators to your Python function, and Numba does the rest.

Learn More » | Try Now »

```
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

I've been using Numba for more than 2 years...

...and it always wins in my ease-of-use judgements and performance tests.

Method	Configuration	Speedup	Cores
Plain Python	for-loopy	1×	1
Numba	for-loopy	50×	1
Numba-parallel	for-loopy	165×	all (12)
Numpy	columnar	15×	1
CuPy	columnar	77×	GPU
Dask	columnar	26×	all (12)
Numba-CUDA	CUDA details	800×	GPU
pybind11 `-O3`	for-loopy C++	34×	1
pybind11 `-ffast-math`	for-loopy C++	90×	1
Cython	dual language	3.7×	1

(Sorted by my ease-of-use judgement.)

For-loopy plain Python code

```
import numpy


def run_plain(height, width, maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.full(c.shape, maxiterations, dtype=numpy.int32)
    for h in range(height):
        for w in range(width):                  # for each pixel (h, w)...
            z = c[h, w]
            for i in range(maxiterations):      # iterate at most 20 times
                z = z**2 + c[h, w]              # applying z → z^2 + c
                if abs(z) > 2:                  # if it diverges (|z| > 2)
                    fractal[h, w] = i           # color with iteration number
                    break                       # we're done; go away
    return fractal

fractal = run_plain(6400, 9600)
```

For-loopy Numba-accelerated code

```
import numpy, numba

@numba.jit
def run_numba(height, width, maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.full(c.shape, maxiterations, dtype=numpy.int32)
    for h in range(height):
        for w in range(width):                  # for each pixel (h, w)...
            z = c[h, w]
            for i in range(maxiterations):      # iterate at most 20 times
                z = z**2 + c[h, w]              # applying z → z² + c
                if abs(z) > 2:                  # if it diverges (|z| > 2)
                    fractal[h, w] = i           # color with iteration number
                    break                       # we're done; go away
    return fractal

fractal = run_numba(6400, 9600)                 # runs 50× faster than plain
```

Columnar Numpy code

```
import numpy


def run_numpy(height, width, maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.full(c.shape, maxiterations, dtype=numpy.int32)
    z = c
    for i in range(maxiterations):          # can't break early
        z = z**2 + c                        # applying z → z² + c
        diverged = numpy.absolute(z) > 2    # |z| > 2 is "divergence"
        diverging_now = diverged & (fractal == maxiterations)
        fractal[diverging_now] = i          # only set the new ones
        z[diverged] = 2                     # clamp diverged at 2
    return fractal


fractal = run_numpy(6400, 9600)             # runs 15× faster than plain
```

Here's the catch:

Numba can only accelerate functions and data structures that it recognizes (mostly numbers and arrays).

Here's the catch:

Numba can only accelerate functions and data structures that it recognizes (mostly numbers and arrays).

They must be statically typed (all types known before execution).

Here's the catch:

Numba can only accelerate functions and data structures that it recognizes (mostly numbers and arrays).

They must be statically typed (all types known before execution).

`@numba.jit(nopython=True)` only allows accelerated code;
`@numba.jit()` only accelerates what it can.

We can teach Numba to recognize more functions and types

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*.

Awkward arrays are statically typed

Arbitrarily complex data:

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]
...     ])
```

Has a data type known before execution, which is to say, before (JIT) compilation.

```
>>> print(array.type)
```

```
[0, 3) -> [0, inf) -> ?((float64                |
                         [0, inf) -> float64 |
                         'x' -> int64
                         'y' -> 'z' -> int64 ))
```

Goal: unbox all array types and lower all functions and methods, so that they can be used in Numba functions written by users.

Goal: unbox all array types and lower all functions and methods, so that they can be used in Numba functions written by users.

Status: done with `JaggedArray`, next is `Table`.

Goal: unbox all array types and lower all functions and methods, so that they can be used in Numba functions written by users.

Status: done with `JaggedArray`, next is `Table`.

To use:

```
pip install awkward-numba or
conda install -c conda-forge awkward-numba
```

and then **`import awkward.numba`** in Python.

Physics-motivated example: pairs of muons and jets

```
import numpy, numba, awkward, awkward.numba

def random_particles(num_per_event, num_events):
    num = numpy.random.poisson(num_per_event, num_events)
    pt  = numpy.random.exponential(10, num.sum())
    eta = numpy.random.normal(0, 1, num.sum())
    phi = numpy.random.uniform(-numpy.pi, numpy.pi, num.sum())
    return (num, awkward.JaggedArray.fromcounts(num, pt),
                 awkward.JaggedArray.fromcounts(num, eta),
                     awkward.JaggedArray.fromcounts(num, phi))

num_muons, pt_muons, eta_muons, phi_muons = random_particles(1.5, 1000000)
num_jets,  pt_jets,  eta_jets,  phi_jets  = random_particles(3.5, 1000000)
```

Physics-motivated example: pairs of muons and jets

```
import numpy, numba, awkward, awkward.numba

def random_particles(num_per_event, num_events):
    num = numpy.random.poisson(num_per_event, num_events)
    pt  = numpy.random.exponential(10, num.sum())
    eta = numpy.random.normal(0, 1, num.sum())
    phi = numpy.random.uniform(-numpy.pi, numpy.pi, num.sum())
    return (num, awkward.JaggedArray.fromcounts(num, pt),
                 awkward.JaggedArray.fromcounts(num, eta),
                     awkward.JaggedArray.fromcounts(num, phi))

num_muons, pt_muons, eta_muons, phi_muons = random_particles(1.5, 1000000)
num_jets,  pt_jets,  eta_jets,  phi_jets  = random_particles(3.5, 1000000)
```

Each of these is a jagged array of particle attributes. Mass of all muon-jet pairs is

```
def unzip(pairs): return pairs.i0, pairs.i1
pt1, pt2   = unzip(pt_muons.cross(pt_jets))       # make a big array of all pairs
eta1, eta2 = unzip(eta_muons.cross(eta_jets))     # separately for each attribute
phi1, phi2 = unzip(phi_muons.cross(phi_jets))     # because we don't have Tables yet

# compute mass for all muon-jet pairs in all events in one line
mass = numpy.sqrt(2*pt1*pt2*(numpy.cosh(eta1 - eta2) - numpy.cos(phi1 - phi2)))
```

Physics-motivated example: pairs of muons and jets

For-loopy code to do the same thing (i.e. a conventional analysis):

```
def run_plain(num_muons, pt_muons, eta_muons, phi_muons,
              num_jets,  pt_jets,  eta_jets,  phi_jets):
    offsets = numpy.empty(len(num_muons) + 1, numpy.int64)
    content = numpy.empty((num_muons * num_jets).sum())
    offsets[0] = 0
    for i in range(len(num_muons)):
        offsets[i + 1] = offsets[i]
        for muoni in range(num_muons[i]):
            pt1  = pt_muons[i][muoni]                          # more verbose than it
            eta1 = eta_muons[i][muoni]                         # would be with Table
            phi1 = phi_muons[i][muoni]
            for jeti in range(num_jets[i]):
                pt2  = pt_jets[i][jeti]
                eta2 = eta_jets[i][jeti]
                phi2 = phi_jets[i][jeti]
                content[offsets[i + 1]] = numpy.sqrt(
                    2*pt1*pt2*(numpy.cosh(eta1 - eta2) - numpy.cos(phi1 - phi2)))
                offsets[i + 1] += 1
    return awkward.JaggedArray(offsets[:-1], offsets[1:], content)
```

Physics-motivated example: pairs of muons and jets

For-loopy code to do the same thing (i.e. a conventional analysis):

```
@numba.jit(nopython=True)
def run_numba(num_muons, pt_muons, eta_muons, phi_muons,      # can pass JaggedArrays
              num_jets,  pt_jets,  eta_jets,  phi_jets):      # into Numba-JIT function
    offsets = numpy.empty(len(num_muons) + 1, numpy.int64)
    content = numpy.empty((num_muons * num_jets).sum())
    offsets[0] = 0
    for i in range(len(num_muons)):
        offsets[i + 1] = offsets[i]
        for muoni in range(num_muons[i]):
            pt1  = pt_muons[i][muoni]                         # more verbose than it
            eta1 = eta_muons[i][muoni]                        # would be with Table
            phi1 = phi_muons[i][muoni]
            for jeti in range(num_jets[i]):
                pt2  = pt_jets[i][jeti]
                eta2 = eta_jets[i][jeti]
                phi2 = phi_jets[i][jeti]
                content[offsets[i + 1]] = numpy.sqrt(
                    2*pt1*pt2*(numpy.cosh(eta1 - eta2) - numpy.cos(phi1 - phi2)))
                offsets[i + 1] += 1
    return awkward.JaggedArray(offsets[:-1], offsets[1:], content)          # and out!
```

What this buys us

Method	**Pro**	**Con**	**Runtime**
`JaggedArray.cross`	concise	inflexible	1.1 seconds
Plain Python loop	explicit	verbose	120 seconds
Numba-compiled	explicit	verbose	0.62 seconds (to compile)
			0.22 seconds (to run)

Apart from a factor of 5 between `JaggedArray.cross` and Numba-compiled (which may leap-frog as implementations improve), we can now write arbitrary for-loop algorithms on `JaggedArray` without an enormous cost.

What this buys us

Method	**Pro**	**Con**	**Runtime**
`JaggedArray.cross`	concise	inflexible	1.1 seconds
Plain Python loop	explicit	verbose	120 seconds
Numba-compiled	explicit	verbose	0.62 seconds (to compile)
			0.22 seconds (to run)

Apart from a factor of 5 between `JaggedArray.cross` and Numba-compiled (which may leap-frog as implementations improve), we can now write arbitrary for-loop algorithms on `JaggedArray` without an enormous cost.

The choice can be made based on the type of problem, not performance.

Summary

- I highly recommend Numba for physics analysis.
- Code blocks are only accelerated by Numba if they consist of recognized functions and data structures, and if all types can be statically known.
- Awkward array types are statically known; I can extend Numba to recognize them and their operations.
- Done with `JaggedArrays` (the hardest); usable in a limited way.
- When `Tables`, `ObjectArrays`, and maybe `MaskedArrays` are done, most physics code will work.
- There are 9 other awkward array types...