Columnar data processing for HEP analysis

Jim Pivarski[a] Jaydeep Nandi[b] David Lange[a] Peter Elmer[a]

[a]Princeton University

[b]National Institute of Technology, Silchar, India

July 10, 2018

Motivation

In the late stages of data analysis, only order-of-magnitude speedups translate into increased human productivity, and only if they're easy to set up.

Motivation

In the late stages of data analysis, only order-of-magnitude speedups translate into increased human productivity, and only if they're easy to set up.

Producing a plot in a second instead of an hour is life-changing, but not if it takes two hours to write the script.

Motivation

Most HEP analysis workflows don't optimize for the most critical performance bottleneck: data layout in memory.

Modern processors are much faster than memory, so arranging data for dense, sequential scanning is critical. (a.k.a. "struct of arrays")

It can be hard to set up and analyze data in this form, though.

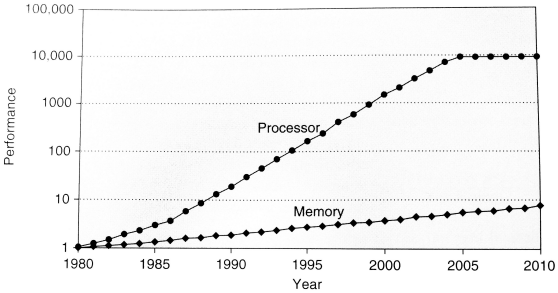
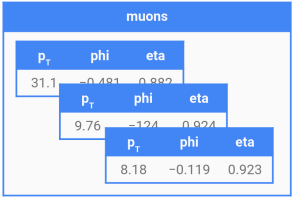


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time.

From Hennessy & Patterson, *Computer Architecture, A Quantitative Approach.*

Motivation

Columnar data representations are particularly complex for hierarchically nested data.

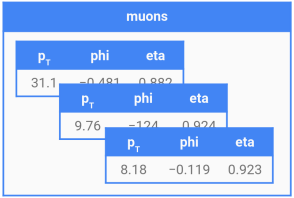


vs

mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

Motivation

Columnar data representations are particularly complex for hierarchically nested data.



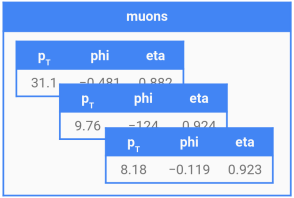
vs

mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)], ...
```

Motivation

Columnar data representations are particularly complex for hierarchically nested data.



vs

mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)], ...
```

becomes four contiguous arrays; need an array of offsets to express the "jagged" structure:

offsets	0,			3,	4,	5,	7
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	−0.481,	−0.123,	−0.119,	1.246,	−0.207,	−1.754,	0.185
eta	0.882,	0.924,	0.923,	−0.991,	0.953,	−0.264,	0.629

This talk will be about

Vertical performance from columnar data processing

and

Convenient syntax for analysis scripts (in Python)

Reading columnar data

The ROOT file format stores data in columns, but ROOT reads them back as C++ objects. If you want arrays for columnar data processing, you have to undo that step.

Reading columnar data

The ROOT file format stores data in columns, but ROOT reads them back as C++ objects. If you want arrays for columnar data processing, you have to undo that step.

uproot, an implementation of ROOT I/O in Python+Numpy, reads TTree branches directly into Numpy arrays.

Reading columnar data

The ROOT file format stores data in columns, but ROOT reads them back as C++ objects. If you want arrays for columnar data processing, you have to undo that step.

uproot, an implementation of ROOT I/O in Python+Numpy, reads TTree branches directly into Numpy arrays.

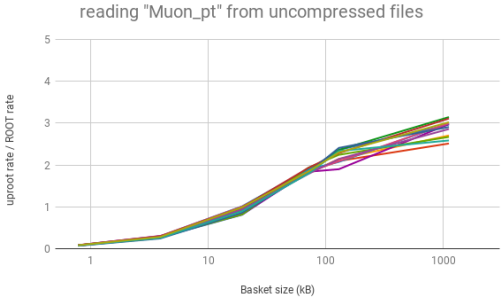
```
>>> import uproot
>>> tree = uproot.open("NanoAOD-DYJetsToLL.root")["Events"]
>>> tree.array("Jet_pt")
jaggedarray([[],
             [29.96875   21.3125    19.671875  17.046875  15.4453125],
             [41.46875  27.625    22.       18.734375],
             ...,
             [34.03125  18.828125 18.359375],
             [42.78125   18.640625  17.640625  16.734375  15.921875  15.7890625],
             [23.75     23.640625 18.96875  ... 16.78125 16.28125 16.25   ]])
```

Branches with nested structure, like `std::vector<float>`, are read as columnar JaggedArrays.

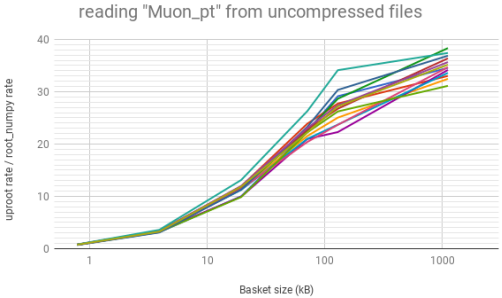
Reading columnar data

Despite the fact that uproot is pure Python, throughput can exceed ROOT (C++) and root_numpy (Cython) for baskets $\gtrsim$ 20 kB.

speedup vs. ROOT



speedup vs. root_numpy

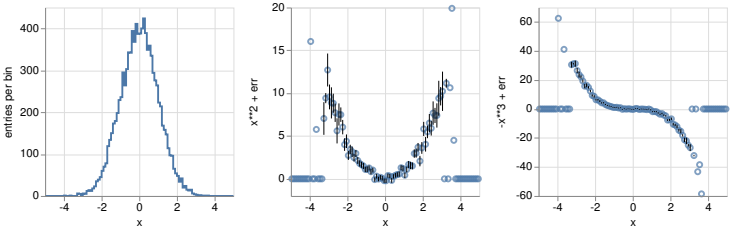


This is because uproot is *doing less work;* only casting basket bytes as arrays.

Plotting columnar data

The Numpy ecosystem lacks comprehensive, HEP-style histogramming and ROOT is designed for events. histbook provides array-at-a-time histogramming.

```
>>> from histbook import *
>>> h = Hist(bin("x", 100, -5, 5), profile("x**2 + err"), profile("-x**3 + err"))
>>> h.fill(x=numpy.random.normal(0, 1, 10000),
...        err=numpy.random.normal(0, 5, 10000))
>>> beside(h.step("x"), h.marker("x", "x**2 + err"), h.marker("x", "-x**3 + err"))
```




```
>>> h.project("x").root()
<ROOT.TH1D object at 0x62a1500>
```

Manipulating columnar data

The real problem, though, is *doing an analysis* on columnar data. JaggedArrays are awkward:

```
>>> for event in range(event_muon_pts):
...     for pt, eta in zip(muon_pts[event], muon_etas[event]):
...         pz = pt * math.sinh(eta)
```

especially if you want view all muon attributes as an object (as a row of a jagged table).

Manipulating columnar data

The real problem, though, is *doing an analysis* on columnar data. JaggedArrays are awkward:

```
>>> for event in range(event_muon_pts):
...     for pt, eta in zip(muon_pts[event], muon_etas[event]):
...         pz = pt * math.sinh(eta)
```

especially if you want view all muon attributes as an object (as a row of a jagged table).

awkward-array is a library (in development!) to manipulate such structures like Numpy.

```
>>> # do all events and particles in one call because they have the same structure:
>>> events["muons"]["pt"] * numpy.sinh(events["muons"]["eta"])
<JaggedArray [[31.128 10.358 8.669] [-6.120] [5.211] [-2.295 5.850]] at 0x7fd394033080>
```

Manipulating columnar data

The real problem, though, is *doing an analysis* on columnar data. JaggedArrays are awkward:

```
>>> for event in range(event_muon_pts):
...     for pt, eta in zip(muon_pts[event], muon_etas[event]):
...         pz = pt * math.sinh(eta)
```

especially if you want view all muon attributes as an object (as a row of a jagged table).

awkward-array is a library (in development!) to manipulate such structures like Numpy.

```
>>> # do all events and particles in one call because they have the same structure:
>>> events["muons"]["pt"] * numpy.sinh(events["muons"]["eta"])
<JaggedArray [[31.128 10.358 8.669] [-6.120] [5.211] [-2.295 5.850]] at 0x7fd394033080>
```

```
>>> # add new attributes to a jagged table:
>>> events["muons"]["pz"] = events["muons"]["pt"] * numpy.sinh(events["muons"]["eta"])
```

Manipulating columnar data

The real problem, though, is *doing an analysis* on columnar data. JaggedArrays are awkward:

```
>>> for event in range(event_muon_pts):
...     for pt, eta in zip(muon_pts[event], muon_etas[event]):
...         pz = pt * math.sinh(eta)
```

especially if you want view all muon attributes as an object (as a row of a jagged table).

awkward-array is a library (in development!) to manipulate such structures like Numpy.

```
>>> # do all events and particles in one call because they have the same structure:
>>> events["muons"]["pt"] * numpy.sinh(events["muons"]["eta"])
<JaggedArray [[31.128 10.358 8.669] [-6.120] [5.211] [-2.295 5.850]] at 0x7fd394033080>

>>> # add new attributes to a jagged table:
>>> events["muons"]["pz"] = events["muons"]["pt"] * numpy.sinh(events["muons"]["eta"])

>>> events["muons"][0].tolist()
[{"pt": 31.1, "phi": -0.481, "eta": 0.882, "pz": 31.128},
 {"pt": 9.76, "phi": -0.123, "eta": 0.924, "pz": 10.358},
 {"pt": 8.18, "phi": -0.119, "eta": 0.923, "pz": 8.669}]
```

Manipulating columnar data

awkward-array is a suite of **composable** high-level array types:

- jagged arrays: for lists of lists of lists of lists...
- tables: for sets of objects (may be jagged if composed with JaggedArray)
- chunked: for data that are not completely contiguous (i.e. ROOT baskets)
- indexed: for pointers, cross-references, dictionary-encoding, event lists, trees...
- masked: for missing data (especially Apache Arrow bitmasks)
- virtual: for read-on-demand, e.g. only a few branches of a ROOT file
- unions: for polymorphism

Manipulating columnar data

awkward-array is a suite of **composable** high-level array types:

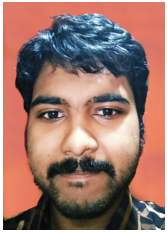
- jagged arrays: for lists of lists of lists of lists...
- tables: for sets of objects (may be jagged if composed with JaggedArray)
- chunked: for data that are not completely contiguous (i.e. ROOT baskets)
- indexed: for pointers, cross-references, dictionary-encoding, event lists, trees...
- masked: for missing data (especially Apache Arrow bitmasks)
- virtual: for read-on-demand, e.g. only a few branches of a ROOT file
- unions: for polymorphism

The data model is very flexible, but the data are accessed as columns:

```
>>> import awkward
>>> columnar_data = awkward.fromiter([1, 2, 3.3, None, [4, 5], {"six": 6}])
>>> columnar_data.tolist()
[1.0, 2.0, 3.3, None, [4, 5], {'six': 6}]
```

Manipulating columnar data

Jaydeep Nandi, our Google Summer of Code student, is investigating vectorized algorithms to replace for-loop manipulations.



```
>>> # broadcast per-event attributes to per-particle attributes:
>>> events["MET"]["phi"] - events["jets"]["phi"]

>>> # explode to event-wise pairs (using only SIMD operations):
>>> pairs = events.pairs()
>>> pairs
<JaggedArray [[<Pair 0> <Pair 1> <Pair 2>] [] [] [<Pair 3>]] at 0x7fd394033080>
>>> pairs[0][0].tolist()
{"_0": {"pt": 31.1, "phi": -0.481, "eta": 0.882, "pz": 31.128},
 "_1": {"pt": 9.76, "phi": -0.123, "eta": 0.924, "pz": 10.358}}

>>> # compute invariant mass without a variable-length loop; this is auto-vectorizable
>>> pt1, eta1, phi1 = pairs["_0"]["pt"], pairs["_0"]["eta"], pairs["_0"]["phi"]
>>> pt2, eta2, phi2 = pairs["_1"]["pt"], pairs["_1"]["eta"], pairs["_1"]["phi"]
>>> mass = numpy.sqrt(2*pt1*pt2*(numpy.cosh(eta1 - eta2) - numpy.cos(phi1 - phi2)))
```

Incidentally, anything that can be expressed this way is ripe for GPU vectorization.

Manipulating columnar data

Analysis functions that can't be expressed as explosions, masks, and reductions can at least be JIT-compiled. Numba (from Anaconda) is a JIT-compiler for Python code.

I implemented awkward-array's predecessor, OAMap, as a Numba extension to get ∼500× speedups. The same techniques will be applied to the new library (not yet).

Runs in 12.9 seconds

```
def run(pz, events):
  k = 0
  for event in events:
    for muon in event.muons:
      pz[k] = muon.pt * math.sinh(muon.eta)
      k += 1
```

Runs in 0.023 seconds

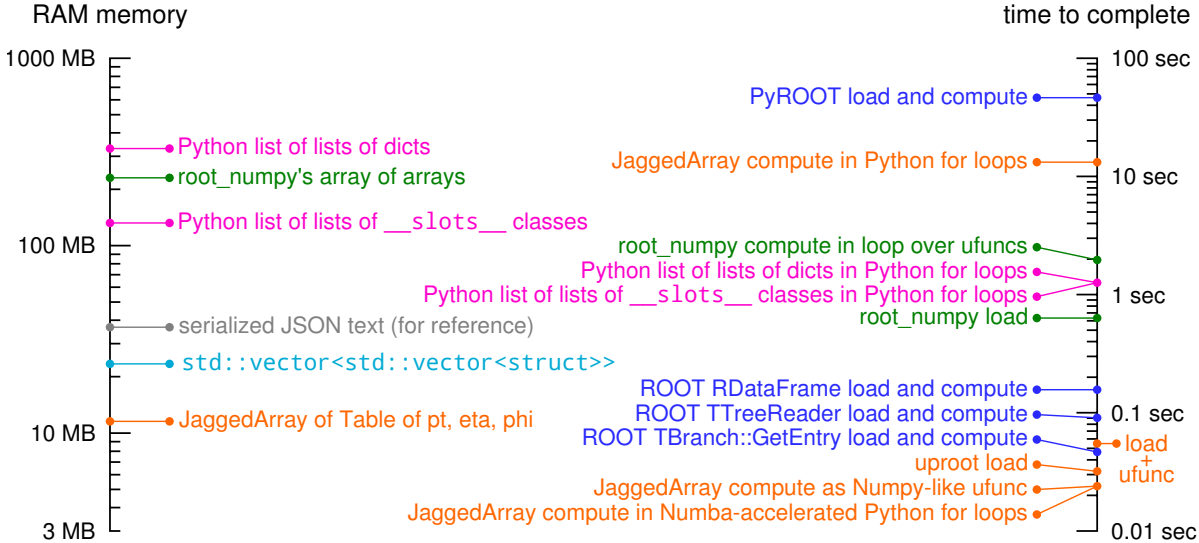
```
import numba
@numba.jit
def run(pz, events):
  k = 0
  for event in events:
    for muon in event.muons:
      pz[k] = muon.pt * math.sinh(muon.eta)
      k += 1
```

What does columnar data buy us?

One simple scenario, many frameworks:

1. Load events containing arbitrarily many muons with {pt, eta, phi} as float32.
2. Compute `pz = pt*sinh(eta)` for all muons in all events.
3. Plot on log scale.

What does columnar data buy us?



What does columnar data buy us?

RAM memory occupied by data (MB)	
311.95	Python list of lists of dicts
215.11	root_numpy's array of arrays
139.79	Python list of lists of `__slots__` classes
37.19	serialized JSON text (for reference)
22.38	`std::vector<std::vector<struct>>`
11.67	JaggedArray of Table of pt, eta, phi

1 MB = 1024^2 bytes
701,716 events containing 552,056 muons
storing pt, eta, phi as float32

time to complete load, compute, or both (sec)	
PyROOT load and compute	45.9
JaggedArray compute in Python for loops	13.4
root_numpy compute in loop over ufuncs	1.96
Python list of lists of dicts in Python for loops	1.24
Python list of lists of `__slots__` classes in Python for loops	1.23
root_numpy load	0.635
ROOT RDataFrame load and compute	0.163
ROOT TTreeReader load and compute	0.091
ROOT TBranch::GetEntry load and compute	0.046
uproot load	0.031
JaggedArray compute as Numpy-like ufunc	0.023
JaggedArray compute in Numba-accelerated Python for loops	0.023

all with warmed disk cache in the same environment

Conclusions

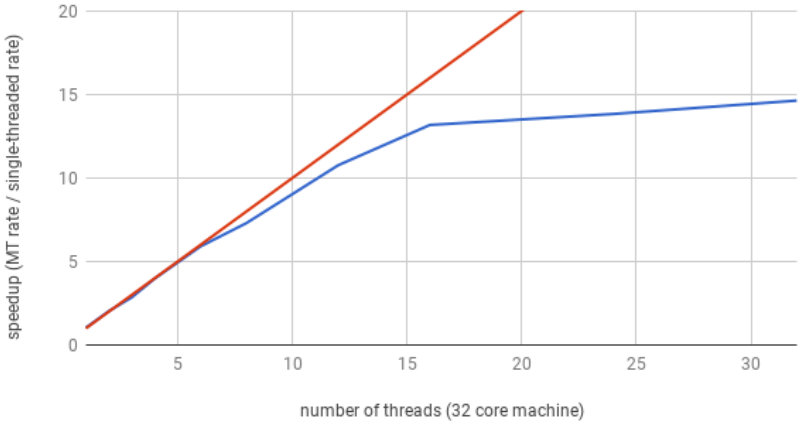
- Python has a model for expressing operations on columnar data: Numpy.
- That model has to be extended to handle the variable-length structures that are ubiquitous in HEP data.
- Opportunities for fundamental work: e.g. how do you do gen/reco jet matching using vectorized instructions? (Part of Jaydeep's project.)
- Can hold more data in memory at a time than non-columnar C++ structures and process it faster, all with Pythonic syntax.

https://github.com/scikit-hep/uproot
https://github.com/scikit-hep/histbook
https://github.com/scikit-hep/awkward-array

Backup

uproot scaling

Read and decompress LZMA with `executor=ThreadPoolExecutor(N)`



JaggedArray compute in Python for loops

```
%%timeit
k = 0
for event in events:
    for muon in event:
        pz[k] = muon.pt * math.sinh(muon.eta)
        k += 1
```

. . . with Numba acceleration:

```
import numba

@numba.jit
def callme(pz, events):
    k = 0
    for event in events:
        for muon in event:
            pz[k] = muon.pt * math.sinh(muon.eta)
            k += 1

%%timeit
callme(pz, events)
```

JaggedArray and root_numpy ufuncs

JaggedArray compute as Numpy-like ufunc:

```
import numpy

%%timeit
pz = events["pt"] * numpy.sinh(events["eta"])
```

root_numpy compute in loop over ufuncs:

```
%%timeit
k = 0
for event in events:
    pt = event["Muon_pt"]
    eta = event["Muon_eta"]
    pz[k : k + len(pt)] = pt * numpy.sinh(eta)
    k += len(pt)
```

Python list of lists of dicts/classes compute in Python for loops

```python
from math import sinh

events = [
    [],
    [{"pt": 129.8,
      "eta": -1.006,
      "phi": -0.581},
     {"pt": 73.08,
      "eta": -0.719,
      "phi": -1.51}],
    ...
    ]

%%timeit
k = 0
for event in events:
    for muon in event:
        pz[k] = (muon["pt"] *
                 sinh(muon["eta"]))
        k += 1
```

```python
class Muon(object):
    __slots__ = ["pt", "eta", "phi"]
    def __init__(self, pt, eta, phi):
        self.pt = pt
        self.eta = eta
        self.phi = phi

events = [
    [],
    [Muon(129.8, -1.006, -0.581),
     Muon(73.08, -0.719, -1.51)],
    ...
    ]

%%timeit
k = 0
for event in asobjs:
    for muon in event:
        pz[k] = (muon.pt *
                 sinh(muon.eta))
        k += 1
```

root_numpy load and uproot load

```
import ROOT
import root_numpy

file = ROOT.TFile("NanoAOD-DYJetsToLL.root")
tree = file.Get("tree")

%%timeit
root_numpy.tree2array(tree, ["Muon_pt", "Muon_eta", "Muon_phi"])
```

```
import uproot
tree = uproot.open("NanoAOD-DYJetsToLL.root")["tree"]

%%timeit
pt, eta, phi = tree.arrays(["Muon_pt", "Muon_eta", "Muon_phi"], outputtype=tuple)
```

PyROOT load and compute

```
import math
import numpy
import ROOT

file = ROOT.TFile("NanoAOD-DYJetsToLL.root")
tree = file.Get("tree")

tree.SetBranchStatus("*", 0)
tree.SetBranchStatus("nMuon", 1)
tree.SetBranchStatus("Muon_pt", 1)
tree.SetBranchStatus("Muon_eta", 1)

pz = numpy.empty(552056, dtype=numpy.float32)

%%timeit
k = 0
for event in tree:
    for pt, eta in zip(event.Muon_pt, event.Muon_eta):
        pz[k] = pt * math.sinh(eta)
        k += 1
```

ROOT RDataFrame load and compute

```
#include <ctime>
#include <sys/time.h>
struct timeval starttime, endtime;

auto file = TFile::Open("NanoAOD-DYJetsToLL.root")
ROOT::RDataFrame rdf("tree", file);
TTree* tree; file->GetObject("tree", tree);    // perhaps unnecessary, but just in case...
tree->SetBranchStatus("*", 0);
tree->SetBranchStatus("nMuon", 1);
tree->SetBranchStatus("Muon_pt", 1);
tree->SetBranchStatus("Muon_eta", 1);

float pz[552056];
gettimeofday(&starttime, 0);
int k = 0;
rdf.Foreach([&k](const ROOT::VecOps::RVec<float> &Muon_pt,
                 const ROOT::VecOps::RVec<float> &Muon_eta) {
    for (int i = 0;  i < Muon_pt.size();  i++) {
        pz[k]= Muon_pt[i] * sinh(Muon_eta[i]);
        k++;
    }
}, {"Muon_pt", "Muon_eta"});
gettimeofday(&endtime, 0);
```

ROOT TTreeReader load and compute

```
#include <ctime>
#include <sys/time.h>
struct timeval starttime, endtime;

auto file = TFile::Open("NanoAOD-DYJetsToLL.root")
TTree* tree; file->GetObject("tree", tree);    // perhaps unnecessary, but just in case...
tree->SetBranchStatus("*", 0);
tree->SetBranchStatus("nMuon", 1);
tree->SetBranchStatus("Muon_pt", 1);
tree->SetBranchStatus("Muon_eta", 1);

TTreeReader reader("tree", file);
TTreeReaderArray<float> pt(reader, "Muon_pt");
TTreeReaderArray<float> eta(reader, "Muon_eta");

gettimeofday(&starttime, 0);
int k = 0;
while (reader.Next()) {
    for (int i = 0;  i < pt.GetSize();  i++) {
        pz[k] = pt[i] * sinh(eta[i]);
        k++;
    }
}
gettimeofday(&endtime, 0);
```

ROOT TBranch::GetEntry load and compute

```
#include <ctime>
#include <sys/time.h>
struct timeval starttime, endtime;

auto file = TFile::Open("NanoAOD-DYJetsToLL.root")
TTree* tree; file->GetObject("tree", tree);

UInt_t nMuon; float pts[10]; float etas[10];
TBranch* nbranch = tree->GetBranch("nMuon");        tree->SetBranchAddress("nMuon", &nMuon);
TBranch* ptbranch = tree->GetBranch("Muon_pt");     tree->SetBranchAddress("Muon_pt", pts);
TBranch* etabranch = tree->GetBranch("Muon_eta");   tree->SetBranchAddress("Muon_eta", etas);

gettimeofday(&starttime, 0);
int k = 0;
for (int i = 0;  i < 701716;  i++) {
    // TBranch::GetEntry, rather than TTree::GetEntry, avoids a loop over branches
    nbranch->GetEntry(i);    ptbranch->GetEntry(i);    etabranch->GetEntry(i);
    for (int j = 0;  j < nMuon;  j++) {
        pz[k] = pts[j] * sinh(etas[j]);
        k++;
    }
}
gettimeofday(&endtime, 0);
```