Vectorized Processing of Nested Data

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

[a]Princeton University

[b]National Institute of Technology, Silchar, India

September 10, 2018

Intersection of three topics

Hardware vectorization

Single Instruction, Multiple Data (SIMD); primary mode of parallelization on GPUs, only way to fully exploit CPUs that have wide vector registers (e.g. AVX-512).

Intersection of three topics

Hardware vectorization

Single Instruction, Multiple Data (SIMD); primary mode of parallelization on GPUs, only way to fully exploit CPUs that have wide vector registers (e.g. AVX-512).

Array programming

High-level programming interface with the same structure as SIMD (Single *VM* Instruction, Multiple Data), which may or may not be vectorized in hardware.

Intersection of three topics

Hardware vectorization

Single Instruction, Multiple Data (SIMD); primary mode of parallelization on GPUs, only way to fully exploit CPUs that have wide vector registers (e.g. AVX-512).

Array programming

High-level programming interface with the same structure as SIMD (Single *VM* Instruction, Multiple Data), which may or may not be vectorized in hardware.

Operations on nested data structures

Often hard to vectorize or even express with array programming.

Examples of physics problems involving nested data operations

1. Compute the ϕ difference between each jet and its event's MET.

```
for event in dataset:
    for jet in event.jets:
        jet.phidiff = jet.phi - event.phi    # N(jets) != N(events)
```

Examples of physics problems involving nested data operations

1. Compute the ϕ difference between each jet and its event's MET.

```
for event in dataset:
    for jet in event.jets:
        jet.phidiff = jet.phi - event.phi    # N(jets) != N(events)
```

2. Compute mass of all particles from two collections, subjet to a cut.

```
for event in dataset:
    event.leptoquarks = []
    for jet in event.jets:
        for lepton in event.leptons:
            if cut(jet, lepton):
                event.pairs.append(mass(jet, lepton))
```

Examples of physics problems involving nested data operations

1. Compute the ϕ difference between each jet and its event's MET.

```
for event in dataset:
    for jet in event.jets:
        jet.phidiff = jet.phi - event.phi    # N(jets) != N(events)
```

2. Compute mass of all particles from two collections, subjet to a cut.

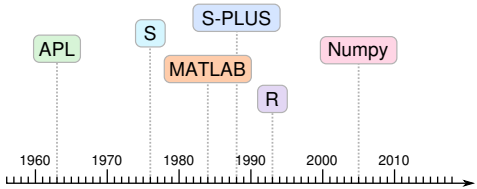
```
for event in dataset:
    event.leptoquarks = []
    for jet in event.jets:
        for lepton in event.leptons:
            if cut(jet, lepton):
                event.pairs.append(mass(jet, lepton))
```

3. Find the "best" candidate per event or per subcollection.

```
for event in dataset:
    event.best = None
    for leptoquark in leptoquarks:
        if event.best is None or \
            quality(leptoquark) > quality(event.best):
                event.best = leptoquark
```

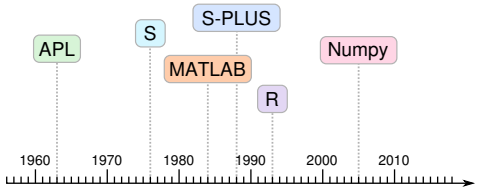
Primarily for data analysis languages

Express regular operations over rectangular data structures in shorthand.



Primarily for data analysis languages

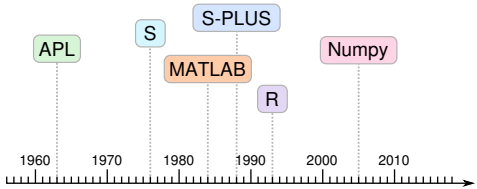
Express regular operations over rectangular data structures in shorthand.



- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`

Primarily for data analysis languages

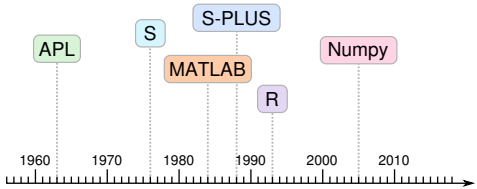
Express regular operations over rectangular data structures in shorthand.



- Multidimensional slices:
```
rgb_pixels[0, 50:100, ::3]
```
- Elementwise operations:
```
all_pz = all_pt * sinh(all_eta)
```

Primarily for data analysis languages

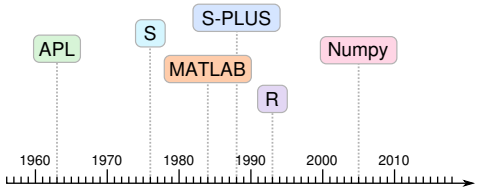
Express regular operations over rectangular data structures in shorthand.



- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- Broadcasting: `all_phi - 2*pi`

Primarily for data analysis languages

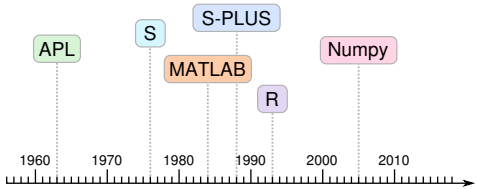
Express regular operations over rectangular data structures in shorthand.



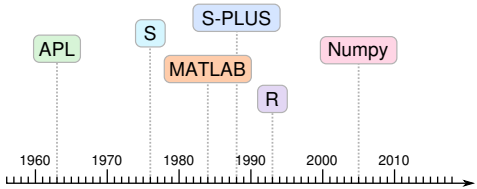
- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- Broadcasting: `all_phi - 2*pi`
- Masking (list compaction): `data[trigger & (pt > 40)]`

Primarily for data analysis languages

Express regular operations over rectangular data structures in shorthand.



- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- Broadcasting: `all_phi - 2*pi`
- Masking (list compaction): `data[trigger & (pt > 40)]`
- Fancy indexing (gather/scatter): `all_eta[argsort(all_pt)]`



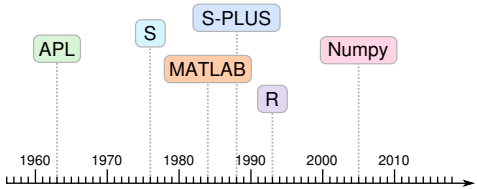
Primarily for data analysis languages

Express regular operations over rectangular data structures in shorthand.

- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- Broadcasting: `all_phi - 2*pi`
- Masking (list compaction): `data[trigger & (pt > 40)]`
- Fancy indexing (gather/scatter): `all_eta[argsort(all_pt)]`
- Row/column commutativity (hides AoS ↔ SoA): `table["column"][7]` (row 7 of column array)
  `table[7]["column"]` (field of row tuple 7)

Primarily for data analysis languages

Express regular operations over rectangular data structures in shorthand.



- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- Broadcasting: `all_phi - 2*pi`
- Masking (list compaction): `data[trigger & (pt > 40)]`
- Fancy indexing (gather/scatter): `all_eta[argsort(all_pt)]`
- Row/column commutativity (hides AoS ↔ SoA): `table["column"][7]` (row 7 of column array)
  `table[7]["column"]` (field of row tuple 7)
- Array reduction: `array.sum()` → scalar

Extension to variable-sized, nested structures

```
Logical structure: [[0, 1, 2], [], [3, 4], [5, 6, 7, 8, 9]]
Offsets:           [0,          3,  3,      5,              10]
Content:           [ 0, 1, 2,       3, 4,   5, 6, 7, 8, 9]
Parents:           [ 0, 0, 0        2, 2,   3, 3, 3, 3, 3]
```

Extension to variable-sized, nested structures

```
Logical structure: [[0, 1, 2], [], [3, 4], [5, 6, 7, 8, 9]]
Offsets:           [0,          3,  3,      5,             10]
Content:           [ 0, 1, 2,       3, 4,   5, 6, 7, 8, 9]
Parents:           [ 0, 0, 0        2, 2,   3, 3, 3, 3, 3]
```

A "jagged array" (content + offsets and/or content + parents) is a basic building block of variable-sized, nested structure.

Extension to variable-sized, nested structures


```
Logical structure: [[0, 1, 2], [], [3, 4], [5, 6, 7, 8, 9]]
Offsets:           [0,          3,  3,      5,             10]
Content:           [ 0, 1, 2,       3, 4,   5, 6, 7, 8, 9]
Parents:           [ 0, 0, 0        2, 2,   3, 3, 3, 3, 3]
```

A "jagged array" (content + offsets and/or content + parents) is a basic building block of variable-sized, nested structure.

- use a jagged array as the content of another jagged array to get `list<list<X>>`

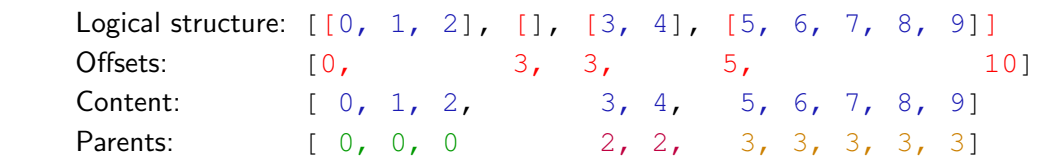
Extension to variable-sized, nested structures


```
Logical structure: [[0, 1, 2], [], [3, 4], [5, 6, 7, 8, 9]]
Offsets:           [0,          3,  3,      5,             10]
Content:           [ 0, 1, 2,       3, 4,   5, 6, 7, 8, 9]
Parents:           [ 0, 0, 0        2, 2,   3, 3, 3, 3, 3]
```

A "jagged array" (content + offsets and/or content + parents) is a basic building block of variable-sized, nested structure.

- use a jagged array as the content of another jagged array to get `list<list<X>>`
- use a fixed-size rectangular array of dimension `N` as content to get `list<X[N]>`

Extension to variable-sized, nested structures



A "jagged array" (content + offsets and/or content + parents) is a basic building block of variable-sized, nested structure.

- use a jagged array as the content of another jagged array to get `list<list<X>>`
- use a fixed-size rectangular array of dimension `N` as content to get `list<X[N]>`
- use a fixed-size rectangular array of dimension `M` as offsets to get `list<X>[M]`

Extension to variable-sized, nested structures


```
Logical structure: [[0, 1, 2], [], [3, 4], [5, 6, 7, 8, 9]]
Offsets:           [0,          3,  3,       5,             10]
Content:           [ 0, 1, 2,       3, 4,    5, 6, 7, 8, 9]
Parents:           [ 0, 0, 0        2, 2,    3, 3, 3, 3, 3]
```

A "jagged array" (content + offsets and/or content + parents) is a basic building block of variable-sized, nested structure.

- use a jagged array as the content of another jagged array to get `list<list<X>>`
- use a fixed-size rectangular array of dimension `N` as content to get `list<X[N]>`
- use a fixed-size rectangular array of dimension `M` as offsets to get `list<X>[M]`

When combined with a table type (column names → arrays), this is as expressive as any combination of `std::vector` and `struct` (i.e. as expressive as ProtoBuf).

Array programming can be extended to jagged arrays

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event

Array programming can be extended to jagged arrays

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep jagged structure

Array programming can be extended to jagged arrays

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep jagged structure
- Broadcasting: `jetphi - metphi` → expand `metphi` from one-per-event to one-per-jet before operation

Array programming can be extended to jagged arrays

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep jagged structure
- Broadcasting: `jetphi - metphi` → expand `metphi` from one-per-event to one-per-jet before operation
- Masking (list compaction): `data[trigger]` → drop whole events
  `data[jetpt > 40]` → drop jets from events

Array programming can be extended to jagged arrays

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep jagged structure
- Broadcasting: `jetphi - metphi` → expand `metphi` from one-per-event to one-per-jet before operation
- Masking (list compaction): `data[trigger]` → drop whole events
  `data[jetpt > 40]` → drop jets from events
- Fancy indexing (gather/scatter): `a = argmax(jetpt)` → `[[2], [], [1], [4]]`
  `jeteta[a]` → `[[3.6], [], [-1.2], [0.4]]`

Array programming can be extended to jagged arrays

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep jagged structure
- Broadcasting: `jetphi - metphi` → expand `metphi` from one-per-event to one-per-jet before operation
- Masking (list compaction): `data[trigger]` → drop whole events
  `data[jetpt > 40]` → drop jets from events
- Fancy indexing (gather/scatter): `a = argmax(jetpt)` → `[[2], [], [1], [4]]`
  `jeteta[a]` → `[[3.6], [], [-1.2], [0.4]]`
- Row/column commutativity (project jagged tables to jagged arrays before indexing):
  `events["jets"]["pt"][7, 1]` (all the same)
  `events["jets"][7]["pt"][1]`
  `events[7]["jets"]["pt"][1]`
  `events["jets"][7, 1]["pt"]`
  `events[7]["jets"][1]["pt"]`

Array programming can be extended to jagged arrays

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep jagged structure
- Broadcasting: `jetphi - metphi` → expand `metphi` from one-per-event to one-per-jet before operation
- Masking (list compaction): `data[trigger]` → drop whole events
  `data[jetpt > 40]` → drop jets from events
- Fancy indexing (gather/scatter): `a = argmax(jetpt)` → `[[2], [], [1], [4]]`
  `jeteta[a]` → `[[3.6], [], [-1.2], [0.4]]`
- Row/column commutativity (project jagged tables to jagged arrays before indexing):
  `events["jets"]["pt"][7, 1]` (all the same)
  `events["jets"][7]["pt"][1]`
  `events[7]["jets"]["pt"][1]`
  `events["jets"][7, 1]["pt"]`
  `events[7]["jets"][1]["pt"]`
- Jagged array reduction: `jetpt.max()` → array of max jet p_T per event

Problem 1: Compute the ϕ difference between each jet and its event's MET.

```
for event in dataset:
    for jet in event.jets:
        jet.phidiff = jet.phi - event.phi
```

Jagged array solution:

```
# because of extended broadcasting rules
events["jets"]["phidiff"] = (
                          events["jets"]["phi"] - events["MET"]["phi"])
```

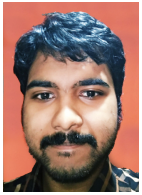
Problem 2: Compute mass of all particles from two collections, subjet to a cut.

```
for event in dataset:
    event.leptoquarks = []
    for jet in event.jets:
        for lepton in event.leptons:
            if cut(jet, lepton):
                event.pairs.append(mass(jet, lepton))
```

Jagged array solution:

```
# jagged cross-join makes (jet, lepton) pairs per event
pairs = events["jets"].cross(events["leptons"])

events["leptoquarks"] = mass(pairs[cut(pairs)])
```

Jagged cross-join has a fully vectorizable implementation (Jaydeep)



```
def cross(self, other):
    offsets = counts2offsets(self.counts * other.counts)
    parents = offsets2parents(offsets)
    indexes = numpy.arange(offsets[-1], dtype=int)

    # fancy indexing get -> SIMD gather
    ocp = other.counts[parents]
    iop = indexes - offsets[parents]
    iop_ocp = iop // ocp

    left = self.content[self.starts[parents] + iop_ocp]
    right = other.content[other.starts[parents] + iop - ocp * iop_ocp]

    return JaggedArray.fromoffsets(offsets, Table(left, right))
```

(There's also a solution for non-repeating pairs of a collection with itself.)

Problem 3: Find the "best" candidate per event or per subcollection.

```
for event in dataset:
    event.best = []
    for leptoquark in leptoquarks:
        if event.best == [] or \
            quality(leptoquark) > quality(event.best[0]):
                event.best = [leptoquark]
```

Jagged array solution:

```
# jagged argmax makes empty lists [] or singleton lists [N]
argbest = quality(events["leptoquarks"]).argmax()

# jagged fancy indexing transforms [] -> [] and [i] -> [sublist[i]]
events["best"] = events["leptoquarks"][argbest]
```

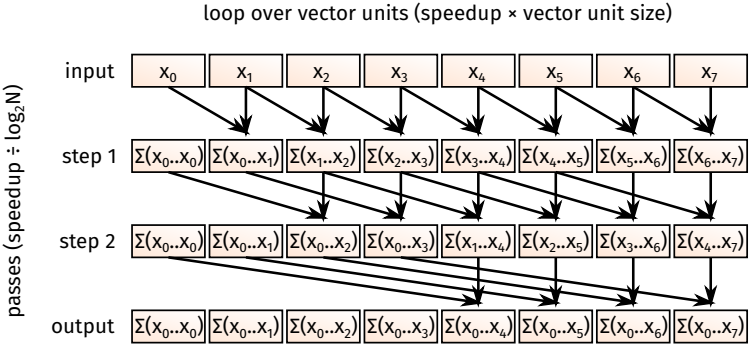
Problem 3: Find the "best" candidate per event or per subcollection.

```
for event in dataset:
    event.best = []
    for leptoquark in leptoquarks:
        if event.best == [] or \
            quality(leptoquark) > quality(event.best[0]):
                event.best = [leptoquark]
```

Jagged array solution:

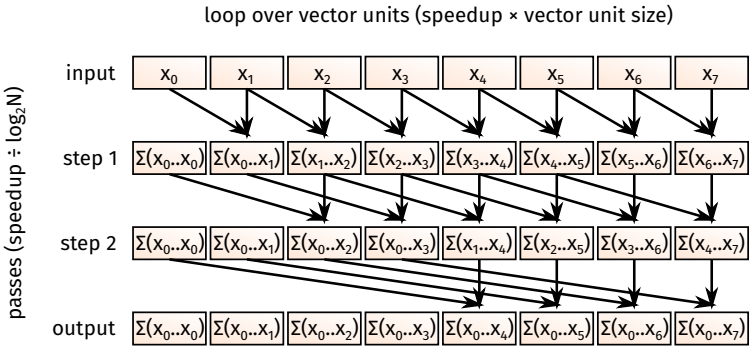
```
# jagged argmax makes empty lists [] or singleton lists [N]
argbest = quality(events["leptoquarks"]).argmax()

# jagged fancy indexing transforms [] -> [] and [i] -> [sublist[i]]
events["best"] = events["leptoquarks"][argbest]

# remove empty lists and concatenate singletons by dropping offsets
nonempty_best = events["best"].flatten()
```

Surprisingly, jagged reducers are fully vectorizable, too (Jaydeep)



This is known as the Hillis-Steele algorithm: creates a cumulative sum in parallel.

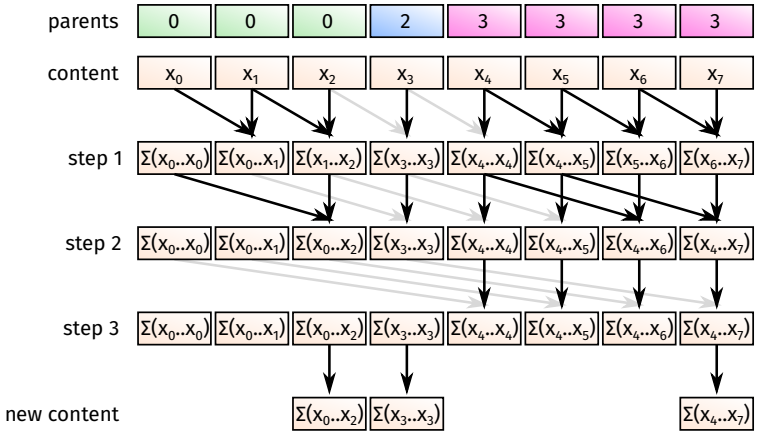
Surprisingly, jagged reducers are fully vectorizable, too (Jaydeep)



example

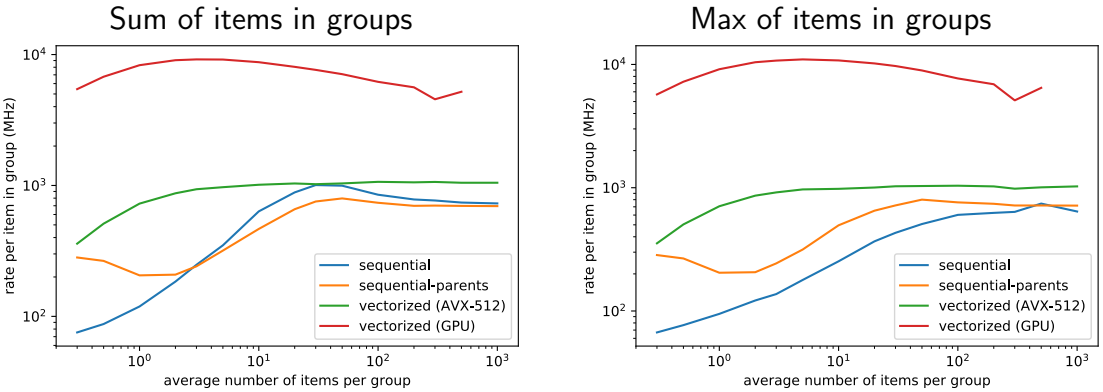
input	1.1	2.2	3.3	4.0	5.0	−10.0	0.0	5.0
step 1	1.1	3.3	5.5	7.3	9.0	−5.0	−10.0	5.0
step 2	1.1	3.3	6.6	10.6	14.5	2.3	−1.0	0.0
output	1.1	3.3	6.6	10.6	15.6	5.6	5.6	10.6

Surprisingly, jagged reducers are fully vectorizable, too (Jaydeep)



Modification of the Hillis-Steele algorithm: only combine pairs in the same event (by checking parents) and then take the last value in each event.

Performance of modified Hillis-Steele



cmslpcgpu3.fnal.gov (Xeon Silver 4110 CPU, Tesla P100 GPU)

https://github.com/jpivarski/jupyter-performance-studies/blob/master/2018-09-10-jagged-reduction.ipynb

Working examples in an interactive notebook: try it out!

https://mybinder.org/v2/gh/scikit-hep/uproot/master?filepath=binder%2Fversion-3-features.ipynb

In [66]:
```
# compute jets × muons, add the pairs to get leptoquark TLorentzVectors, then compute mass
jetp4.cross(muonp4).apply(lambda a, b: a + b).mass()
```

Out[66]:
```
<JaggedArray [[] [61.49117266] [] ... [99.96350949], [ 73.10963363 106.82646499], []] at 7f305cf8ada0>
```

In [67]:
```
# the same except compute mass in the apply— the apply can return anything
jetp4.cross(muonp4).apply(lambda a, b: (a + b).mass())
```

Out[67]:
```
<JaggedArray [[] [61.49117266] [] ... [99.96350949], [ 73.10963363 106.82646499], []] at 7f305cf9b208>
```

In [69]:
```
# compute muons × muons, make Z candidates, and only consider Z candidates with pT > 40
muonp4.pairs(same=False).apply(lambda a, b: a + b).filter(lambda z: z.pt() > 40)
```

Out[69]:
```
<JaggedArrayMethods [[] [] [TLorentzVector(49.815, 8.0774, 48.133, 102.23)] ... [], [], []] at 7f305cf9b3c8>
```

In [70]:
```
# now only consider events that have at least one Z candidate with pT > 40
muonp4.pairs(same=False).apply(lambda a, b: a + b).filter(lambda z: z.pt().max() > 40)
```

Out[70]:
```
<JaggedArrayMethods [[TLorentzVector(49.815, 8.0774, 48.133, 102.23)] [TLorentzVector(98.78, -99.792, 738.94, 757.
5)] [TLorentzVector(84.922, 92.652, -69.581, 172.15)] ... [TLorentzVector(59.597, 72.155, 35.339, 135.72)], [TLorent
zVector(52.18, 58.075, 53.209, 129.65)], [TLorentzVector(53.845, 8.6018, -227.56, 254.94)]] at 7f305cf9b668>
```

In [71]:
```
# get the mass of the Z candidate with the highest pT
muonp4.pairs(same=False).apply(lambda a, b: a + b).maxby(lambda z: z.pt()).mass()
```

Out[71]:
```
<JaggedArray [[90.22779777] [] [74.74654928] ... [], [], []] at 7f305d0011d0>
```

Relevance to ROOT

None of the preceeding needs to be in Python or Numpy.

None of the preceeding needs to be in Python or Numpy.

It was motivated to bypass Python's slow for loops, but it helps physicists express analysis code more succinctly (interface) and it sets up their analysis code for hardware parallelization (performance).

None of the preceeding needs to be in Python or Numpy.

It was motivated to bypass Python's slow for loops, but it helps physicists express analysis code more succinctly (interface) and it sets up their analysis code for hardware parallelization (performance).

- The jagged array structure aligns perfectly with the RForest concept.

None of the preceeding needs to be in Python or Numpy.

It was motivated to bypass Python's slow for loops, but it helps physicists express analysis code more succinctly (interface) and it sets up their analysis code for hardware parallelization (performance).

- The jagged array structure aligns perfectly with the RForest concept.
- An array programming model could be supported by RTensor.

None of the preceeding needs to be in Python or Numpy.

It was motivated to bypass Python's slow for loops, but it helps physicists express analysis code more succinctly (interface) and it sets up their analysis code for hardware parallelization (performance).

- The jagged array structure aligns perfectly with the RForest concept.
- An array programming model could be supported by RTensor.
- VecOps::RVec provides one-in-one-out vectorization; it could be extended to provide vectorized jagged operations as well.

None of the preceeding needs to be in Python or Numpy.

It was motivated to bypass Python's slow for loops, but it helps physicists express analysis code more succinctly (interface) and it sets up their analysis code for hardware parallelization (performance).

- The jagged array structure aligns perfectly with the RForest concept.
- An array programming model could be supported by RTensor.
- VecOps::RVec provides one-in-one-out vectorization; it could be extended to provide vectorized jagged operations as well.
- It could fit into RDataFrame if applied to clusters ("map partitions").

None of the preceeding needs to be in Python or Numpy.

It was motivated to bypass Python's slow for loops, but it helps physicists express analysis code more succinctly (interface) and it sets up their analysis code for hardware parallelization (performance).

- The jagged array structure aligns perfectly with the RForest concept.
- An array programming model could be supported by RTensor.
- VecOps::RVec provides one-in-one-out vectorization; it could be extended to provide vectorized jagged operations as well.
- It could fit into RDataFrame if applied to clusters ("map partitions").
- Doing so would provide a "C++ story" for how physicists can use BulkIO.

Where to find implementations and example code

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

  Implementations of jagged arrays and tables in pure Numpy. Vectorized accelerators will appear as `awkward-array-cpu` and `awkward-array-gpu`.

- Jaydeep's report on vectorized algorithms for cross ("combinations"), pairs, jagged reduction ("global argmin"):

  https://gitlab.com/Jayd_1234/GSoC_vectorized_proof_of_concepts

  Algorithms are demonstrated in Jupyter notebooks; final writeup is in Markdown.
  Earlier work:

  https://github.com/Jayd-1234/GSoC_vectorized_proof_of_concepts

- awkward-array is now used heavily by uproot and uproot-methods. Examples include array programming with TLorentzVectors and jagged arrays of TLorentzVectors.