Nested data structures in array and SIMD frameworks

Jim Pivarski

Princeton University – DIANA-HEP, IRIS-HEP

March 14, 2019
Nested, variable-sized data structures are crucial in HEP. Analysis datasets are big lists of variable-length lists of structs/objects/records.

```
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),
...```

<table>
<thead>
<tr>
<th>muons</th>
<th>p_T</th>
<th>phi</th>
<th>eta</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_T</td>
<td>phi</td>
<td>eta</td>
<td></td>
</tr>
<tr>
<td>31.1</td>
<td>-0.481</td>
<td>0.882</td>
<td></td>
</tr>
<tr>
<td>9.76</td>
<td>-124</td>
<td>0.924</td>
<td></td>
</tr>
<tr>
<td>8.18</td>
<td>-0.119</td>
<td>0.923</td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>mu1</th>
<th>mu1</th>
<th>mu1</th>
<th>mu2</th>
<th>mu2</th>
<th>mu2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_T</td>
<td>phi</td>
<td>eta</td>
<td>p_T</td>
<td>phi</td>
<td>eta</td>
</tr>
<tr>
<td>31.1</td>
<td>-0.481</td>
<td>0.882</td>
<td>9.76</td>
<td>-0.124</td>
<td>0.924</td>
</tr>
<tr>
<td>5.27</td>
<td>1.246</td>
<td>-0.991</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>4.72</td>
<td>-0.207</td>
<td>0.953</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>8.59</td>
<td>-1.754</td>
<td>-0.264</td>
<td>8.714</td>
<td>0.185</td>
<td>0.629</td>
</tr>
</tbody>
</table>
```
Nested, variable-sized data structures are crucial in HEP Analysis datasets are big lists of variable-length lists of structs/objects/records.

\[
\text{muons}
\]

\[
\begin{array}{ccc}
 p_T & \phi & \eta \\
 31.1 & -0.481 & 0.882 \\
 9.76 & -1.244 & 0.924 \\
 8.18 & -0.119 & 0.923 \\
\end{array}
\]

\[
\begin{array}{ccc}
 p_T & \phi & \eta \\
 31.1 & -0.481 & 0.882 \\
 9.76 & -1.244 & 0.924 \\
 8.18 & -0.119 & 0.923 \\
\end{array}
\]

\[
\begin{array}{cccc}
 \text{mu1} & \text{mu1} & \text{mu1} & \text{mu2} \\
 p_T & \phi & \eta & p_T & \phi & \eta \\
 31.1 & -0.481 & 0.882 & 9.76 & -0.124 & 0.924 \\
 5.27 & 1.246 & -0.991 & n/a & n/a & n/a \\
 4.72 & -0.207 & 0.953 & n/a & n/a & n/a \\
 8.59 & -1.754 & -0.264 & 8.714 & 0.185 & 0.629 \\
\end{array}
\]

Analysis datasets are big lists of variable-length lists of structs/objects/records.

\[
[\text{Muon}(31.1, -0.481, 0.882), \text{Muon}(9.76, -0.124, 0.924), \text{Muon}(8.18, -0.119, 0.923)],
\]

\[
[\text{Muon}(5.27, 1.246, -0.991)],
\]

\[
[\text{Muon}(4.72, -0.207, 0.953)],
\]

\[
[\text{Muon}(8.59, -1.754, -0.264), \text{Muon}(8.714, 0.185, 0.629)],
\]

...
Columnar representation

But they don’t have to be “structs,” pointers to contiguous $p_T$, $\eta$, $\phi$ triples.

$\begin{array}{c}
\text{[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],} \\
\text{[Muon(5.27, 1.246, -0.991)],} \\
\text{[Muon(4.72, -0.207, 0.953)],} \\
\text{[Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)]}, \\
\ldots
\end{array}$

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

counts

<table>
<thead>
<tr>
<th>counts</th>
<th>3, 1, 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_T$</td>
<td>31.1, 9.76, 8.18, 5.27, 4.72, 8.59, 8.714</td>
</tr>
<tr>
<td>phi</td>
<td>-0.481, -0.123, -0.119, 1.246, -0.207, -1.754, 0.185</td>
</tr>
<tr>
<td>eta</td>
<td>0.882, 0.924, 0.923, -0.991, 0.953, -0.264, 0.629</td>
</tr>
</tbody>
</table>
But they don’t have to be “structs,” pointers to contiguous $p_T$, $\eta$, $\phi$ triples.

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

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

<table>
<thead>
<tr>
<th>offsets</th>
<th>0, 3, 4, 5, 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_T$</td>
<td>31.1, 9.76, 8.18, 5.27, 4.72, 8.59, 8.714</td>
</tr>
<tr>
<td>phi</td>
<td>-0.481, -0.123, -0.119, 1.246, -0.207, -1.754, 0.185</td>
</tr>
<tr>
<td>eta</td>
<td>0.882, 0.924, 0.923, -0.991, 0.953, -0.264, 0.629</td>
</tr>
</tbody>
</table>
But they don’t have to be “structs,” pointers to contiguous $p_T$, $\eta$, $\phi$ triples.

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

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

\begin{align*}
\text{starts} & : 0, 3, 4, 5 \\
\text{stops} & : 3, 4, 5, 7 \\
\rho_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
\end{align*}
But they don’t have to be “structs,” pointers to contiguous $p_T$, $\eta$, $\phi$ triples.

```plaintext
[[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)],
...
```

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

<table>
<thead>
<tr>
<th>parents</th>
<th>0, 0, 0, 1, 2, 3, 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_T$</td>
<td>31.1, 9.76, 8.18, 5.27, 4.72, 8.59, 8.714</td>
</tr>
<tr>
<td>phi</td>
<td>-0.481, -0.123, -0.119, 1.246, -0.207, -1.754, 0.185</td>
</tr>
<tr>
<td>eta</td>
<td>0.882, 0.924, 0.923, -0.991, 0.953, -0.264, 0.629</td>
</tr>
</tbody>
</table>
This allows for efficient ways of *manipulating* data

“Remove the first muon from each event.”

```
[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)],
...
```

“Remove the first muon from each event.”

| starts | 0, 3, 4, 5 |
| stops | 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 allows for efficient ways of *manipulating* data.

“Remove the first muon from each event.” \(\rightarrow\) rewrite all inner lists.

\[
[[],
\text{Muon}(9.76, -0.124, 0.924), \text{Muon}(8.18, -0.119, 0.923)],
[null],
[null],
\text{Muon}(8.714, 0.185, 0.629)],
\ldots
\]

“Remove the first muon from each event.” \(\rightarrow\) increase all starts by 1.

| starts | 1, 4, 5, 6 |
| stops  | 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 |
“Remove the first muon from each event.”  \[\rightarrow\] rewrite all inner lists.

\[
\begin{array}{c}
\text{Muon}(9.76, -0.124, 0.924), \text{Muon}(8.18, -0.119, 0.923), \\
\text{Muon}(8.714, 0.185, 0.629)
\end{array}
\]

“Remove the first muon from each event.”  \[\rightarrow\] increase all starts by 1.

| starts | 1, 4, 5, 6 |
| stops  | 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 |

We didn’t need to touch any contents (read them from disk, decompress them...).
Library for manipulating non-standard array types

- variable-length subarrays: “jagged arrays”
- struct-of-arrays viewed as array-of-structs
- nullable types
- heterogeneous types (tagged unions)
- cross-references or even cyclic references
- sparse, non-contiguous, lazy

https://github.com/scikit-hep/awkward-array
Library for manipulating non-standard array types

- variable-length subarrays: “jagged arrays”
- struct-of-arrays viewed as array-of- structs
- nullable types
- heterogeneous types (tagged unions)
- cross-references or even cyclic references
- sparse, non-contiguous, lazy

Fully composable: any awkward array can be placed within any other awkward array.
Not lacking for data types

Nullable, heterogeneous, multiple levels of depth, nested records...

```python
>>> 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}}])

>>> print(array)
# internally, these are all arrays
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [{Row 0} None {Row 1}]]

>>> print(array[:, -2:])
# all of outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [{Row 1}]]

>>> (array + 100).tolist()
# element-wise function applied to arrays
[[101.1, 102.2, None, 103.3, None],
 [104.4, [105.5]],
 [{"x": 106, "y": {"z": 107}}, None, {"x": 108, "y": {"z": 109}}]]
```
Not lacking for data types

Nullable, heterogeneous, multiple levels of depth, nested records...

```python
>>> 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}}]])

>>> print(array)  # internally, these are all arrays
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [{Row 0} None {Row 1}]]
```

```python
>>> print(array[:, -2:])  # all of outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [{Row 1}]]
```

```python
>>> (array + 100).tolist()  # element-wise function applied to arrays
[[101.1, 102.2, None, 103.3, None],
 [104.4, [105.5]],
 [{"x": 106, "y": {"z": 107}}, None, {"x": 108, "y": {"z": 109}}]]
```
Not lacking for data types

```
>>> 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}}]])

>>> print(array)  # internally, these are all arrays
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]

>>> print(array[:, -2:])  # all of outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [None <Row 1>]]
```
Not lacking for data types

Nullable, heterogeneous, multiple levels of depth, nested records...

```python
>>> 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}}]))

>>> print(array)  # internally, these are all arrays
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]

>>> print(array[:, -2:])  # all of outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [None <Row 1>]]

>>> (array + 100).tolist()  # element-wise function applied to arrays
[[101.1, 102.2, None, 103.3, None],
[104.4, [105.5]],
[{'x': 106, 'y': {'z': 107}}, None, {'x': 108, 'y': {'z': 109}}]]
```
Columnar data structures minimize memory use and time.

Example of one operation, deriving $p_z$ of a variable number of $p_T$ and $\eta$ per event, using awkward-array, ROOT, pure Python, and root_numpy.
a single operation $\neq$ a physics analysis
Beyond toy studies

Coffea

Columnar Object Framework For Efficient Analysis

Matteo Cremonesi, Lindsey Gray, Oliver Gutsche, Allison Hall, Bo Jayatilaka, Igor Mandrichenko, Kevin Pedro, Nick Smith [FNAL], and me [Princeton]  
https://github.com/CoffeaTeam

Performing two complete CMS analyses with columnar tools:

- Dark Higgs search
- Boosted SM $H \to b\bar{b}$

Also developing fnal-column-analysis-tools, a HEP layer on awkward-array, and a distributed query processing system with Ben Galewsky, Mark Neubauer [Illinois], and Andrew Melo [Vanderbilt].
Z peak is the “hello world” of analysis frameworks.

This implementation is realistic: run-lumi mask, pile-up correction, ID scale factors, and $ee/\mu\mu/e\mu$ channels. 350 lines in a Jupyter notebook, accessing 25 columns.
First finished analysis: Z peak

Z peak is the “hello world” of analysis frameworks.

This implementation is realistic: run-lumi mask, pile-up correction, ID scale factors, and $ee/\mu\mu/e\mu$ channels. 350 lines in a Jupyter notebook, accessing 25 columns.

columnar analysis: 
6 $\mu$s/event/thread (165 kHz)

ROOT C++:
4 $\mu$s/event/thread (250 kHz)

Columnar analysis is about 50% slower than its C++ equivalent.
Prototype of boosted $H \rightarrow b\bar{b}$ has

- recursive gen parent-finding
- gen-reco matching
- binned corrections
- parametric corrections
- systematics

70 $\mu$s/event/thread (14 kHz)

Uses about 100 columns.
Analysts’ “favorite stanzas” of the boosted $H \rightarrow b\bar{b}$ analysis

VBF signal region definition:

$$\text{AK4jet\_AK8jet\_matches} = \text{ak4\_goodjets}.\text{fastmatch}(\text{leadingak8jet})$$

$$\text{unmatched\_ak4} = \text{ak4\_goodjets}[\sim \text{AK4jet\_AK8jet\_matches}]$$

$$\text{vbf\_ak4\_pairs} = \text{unmatched\_ak4}.\text{p4}.\text{distincts}(\text{nested=True})$$

$$\text{vbf\_ak4\_detas} = \text{np.abs(}\text{vbf\_ak4\_pairs.i0.eta} - \text{vbf\_ak4\_pairs.i1.eta}).\text{flatten()}$$

$$\text{vbf\_ak4\_maxdetas} = \text{vbf\_ak4\_detas}.\text{argmax}()$$

$$\text{vbf\_ak4\_masses} = (\text{vbf\_ak4\_pairs.i0} + \text{vbf\_ak4\_pairs.i1}).\text{mass}.\text{flatten()}[\text{vbf\_ak4\_maxdetas}]$$

$$\text{vbf\_ak4\_pass} = (\text{vbf\_ak4\_detas}[\text{vbf\_ak4\_maxdetas}] > 3.25) \& (\text{vbf\_ak4\_masses} > 975.0)$$

Leading fat-jet selection with vetos:

$$\text{k8puppijet\_pt200} = \text{ak8puppijet}\[\text{passLooseJetSel(ak8puppijet)} \& (\text{ak8puppijet.pt} > 200) \& (\text{np.abs(ak8puppijet.eta)} < 2.5)]$$

$$\text{ak8veto} = \sim(\text{ak8puppijet\_pt200}.\text{fastmatch(vetoMuons, deltaRCut=0.4)} \mid \text{ak8puppijet\_pt200}.\text{fastmatch(vetoElectrons, deltaRCut=0.4)} \mid \text{ak8puppijet\_pt200}.\text{fastmatch(vetoPhotons, deltaRCut=0.4))}$$

$$\text{ak8jets\_veto} = \text{ak8puppijet\_pt200}[\text{ak8veto}]$$

$$\text{leadingak8jet} = \text{ak8jets\_veto}[\text{ak8jets\_veto.pt}.\text{argmax()}$$
Syntax is an extension of Numpy

Numpy arrays must all be rectangular: vectors, matrices, and tensors. Awkward arrays reproduce this behavior in rectangular cases and generalize in jagged cases.

- **Multidimensional slices:**
  
  \[ \text{rgb\_pixels}[0, 50:100, \ldots 3] \]

- **Elementwise operations:**
  
  \[ \text{all\_pz} = \text{all\_pt} \times \sinh(\text{all\_eta}) \]

- **Broadcasting:**
  
  \[ \text{all\_phi} - 2\pi \]

- **Masking (list compaction):**
  
  \[ \text{data}[\text{trigger} \& (\text{pt} > 40)] \]

- **Fancy indexing (gather/scatter):**
  
  \[ \text{all\_eta}[\text{argsort}(\text{all\_pt})] \]

- **Row/column commutativity (hides AoS ↔ SoA):**
  
  \[ \text{table}["column"][7] \text{ (row 7 of column array)} \]

  \[ \text{table}[7]["column"] \text{ (field of row tuple 7)} \]

- **Array reduction:**
  
  \[ \text{array}.\text{sum()} \rightarrow \text{scalar} \]
Syntax is an extension of Numpy

Numpy arrays must all be rectangular: vectors, matrices, and tensors. Awkward arrays reproduce this behavior in rectangular cases and generalize in jagged cases.

- Multidimensional slices:
  \[
  \text{events["jets"][\::, 0]} \rightarrow \text{first jet per event}
  \]

- Elementwise operations:
  \[
  \text{jetpt * sinh(jeteta)} \rightarrow \text{keep jagged structure}
  \]

- Broadcasting:
  \[
  \text{jetphi - metphi} \rightarrow \text{expand metphi from one-per-event to one-per-jet before operation}
  \]

- Masking (list compaction):
  \[
  \text{data[trigger]} \rightarrow \text{drop whole events}
  \]
  \[
  \text{data[jetpt > 40]} \rightarrow \text{drop jets from events}
  \]

- Fancy indexing (gather/scatter):
  \[
  a = \text{argmax(jetpt)} \rightarrow [[2], [], [1], [4]]
  \]
  \[
  \text{jeteta[a]} \rightarrow [[3.6], [], [-1.2], [0.4]]
  \]

- Row/column commutativity (hides AoS ↔ SoA):
  \[
  \text{events["jets"][:,:,7, 1],}
  \]
  \[
  \text{events["jets"][7]["pt"][1],}
  \]
  \[
  \text{events[7]["jets"]["pt"][1],}
  \]
  \[
  \ldots
  \]

- Jagged array reduction:
  \[
  \text{jetpt.max()} \rightarrow \text{array of max jet } p_T \text{ per event}
  \]
Structure for most physics data: multiple candidates per event

\[
\text{JaggedArray} \left( \text{ObjectArray} \left( \text{Table} \left( \text{px}, \text{py}, \text{pz}, \text{E} \right), \text{LorentzVector} \right) \right)
\]
Structure for most physics data: multiple candidates per event

\[
\text{JaggedArray}(\text{ObjectArray}(\text{Table}(\text{px}, \text{py}, \text{pz}, \text{E}), \text{LorentzVector}))
\]

- \(\text{px, py, pz, E}\) are flat Numpy arrays (all particles, all events).
- \text{Table} to present contiguous columns as an array of rows
- \text{ObjectArray} to interpret rows of the \text{Table} as \text{LorentzVector} objects
- \text{JaggedArray} because there’s a variable number of \text{LorentzVectors} per event
Structure for most physics data: multiple candidates per event

\[
\text{JaggedArray}(\text{ObjectArray}(\text{Table}(\text{px}, \text{py}, \text{pz}, \text{E}), \text{LorentzVector}))
\]

- \(\text{px}, \text{py}, \text{pz}, \text{E}\) are flat Numpy arrays (all particles, all events).
- \text{Table} to present contiguous columns as an array of rows
- \text{ObjectArray} to interpret rows of the \text{Table} as \text{LorentzVector} objects
- \text{JaggedArray} because there’s a variable number of \text{LorentzVectors} per event

Individual \text{LorentzVector} objects have kinematic methods—\(\text{pt}, \text{eta}, \text{mass},\) etc.—but so do the \text{ObjectArray} and \text{JaggedArray}. Whole-array methods are vectorized.

To compute the mass of all particles in all events are pack it into per-event sublists, you say

```python
>>> particles.mass
```

14 / 20
Putting it all together: a simple Z peak

```python
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")
```

```python
>>> array
# muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
TLorentzVector(37.738, 0.69347, -11.308, 39.402)]]>
```

```python
>>> array[0, 1]
# second muon in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)
```

```python
>>> hastwo = (array.counts >= 2)
# to select at least two muons
>>> leading = array[hastwo, 0]
# mask and select first
>>> subleading = array[hastwo, 1]
# mask and select second
>>> candidates = leading + subleading
# Lorentz vector sum across all
>>> candidates.mass
# compute mass for all
array([90.22779777, 74.74654928, ..., 85.44384208, 75.96066262])
```
Putting it all together: a simple Z peak

```python
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array  # muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
             TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>
```
Putting it all together: a simple Z peak

```python
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array
# muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
    TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>

>>> array[0, 1]
# second muon in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)
```
Putting it all together: a simple Z peak

```python
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array
# muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
             TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...

>>> array[0, 1]
# second muon in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)

>>> hastwo = (array.counts >= 2)
# to select at least two muons
>>> leading = array[hastwo, 0]
# mask and select first
>>> subleading = array[hastwo, 1]
# mask and select second

>>> candidates = leading + subleading
# Lorentz vector sum across all
>>> candidates.mass
# compute mass for all
array([90.22779777, 74.74654928, ..., 85.44384208, 75.96066262])
```
Putting it all together: a simple Z peak

```python
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array
# muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
            T LorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>

>>> array[0, 1]
# second muon in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)

>>> hastwo = (array.counts >= 2)
# to select at least two muons
>>> leading = array[hastwo, 0]
# mask and select first
>>> subleading = array[hastwo, 1]
# mask and select second

>>> candidates = leading + subleading
# Lorentz vector sum across all
>>> candidates.mass
# compute mass for all
array([90.22779777, 74.74654928, ..., 85.44384208, 75.96066262])
```
Feedback from physicists

- Many bug-fixes, of course.
Feedback from physicists

▶ Many bug-fixes, of course.

▶ `argmin()` was an analysis bottleneck. Reimplemented to be $125 \times$ faster.
Feedback from physicists

▶ Many bug-fixes, of course.

▶ `argmin()` was an analysis bottleneck. Reimplemented to be $125 \times$ faster.

▶ Validity checks (e.g. to ensure that all `starts/stops are within content`) are expensive. Need to avoid redundant checks without removing safety.
Feedback from physicists

- Many bug-fixes, of course.

- argmin() was an analysis bottleneck. Reimplemented to be $125 \times$ faster.

- Validity checks (e.g. to ensure that all starts/stops are within content) are expensive. Need to avoid redundant checks without removing safety.

- Discovering which operations are frequently used in analysis, which aren’t.
Feedback from physicists

Many bug-fixes, of course.

argmin() was an analysis bottleneck. Reimplemented to be $125 \times$ faster.

Validity checks (e.g. to ensure that all starts/stops are within content) are expensive. Need to avoid redundant checks without removing safety.

Discovering which operations are frequently used in analysis, which aren’t.

Interesting mistake: analysts must unlearn order-dependent coding habits!

(nMuons > 0) & (Muons_pt[:, 0] > 30) # intersection of masks

The latter might try to access the first of zero muons.
Feedback from physicists

- Many bug-fixes, of course.
- `argmin()` was an analysis bottleneck. Reimplemented to be $125 \times$ faster.
- Validity checks (e.g. to ensure that all starts/stops are within content) are expensive. Need to avoid redundant checks without removing safety.
- Discovering which operations are frequently used in analysis, which aren’t.
- Interesting mistake: analysts must unlearn order-dependent coding habits!

```
(nMuons > 0) & (Muons_pt[:, 0] > 30)  # intersection of masks
```

The latter might try to access the first of zero muons.

Instead,

```
Muons_pt[(nMuons > 0), 0] > 30  # mask first dim, pick 0
```
Is this making analysis easier?

If yes, great! Continue developing array operations, thinking about their “ergonomics,” and optimize their implementations.

If no, it’s still a useful abstraction layer, but we’ll need a more user-friendly interface on top of it, such as a functional or declarative language.
Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).
1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

Some motivated by execution speed, some by ease of use.

▶ “Thirty minutes is too long to wait for a plot.”
▶ “Will be run order-of-a-hundred times over the course of the year; this is a big investment.” but “For something that could be two times faster, I wouldn’t do these optimizations.”
▶ “Ease of use is paramount; I’ve always struggled with poorly written code.” and “Making it fast to run it again and again is going around ease of use.”
▶ “Ease of use is most important, even if execution speed decreases.”
Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

Some found it easier, some more difficult.

▶ “Way, way much easier than applying cuts with for loops.”
▶ “Surprised by how conceptually different you have to think about selections, combining objects.” but “Not good or bad, just surprising that it has a learning curve.”
▶ “Individual problems have been much more difficult than expected.” and “Translating ‘if’ statements is where I get hung up.” but “Not inherently harder; just harder now for those of us used to the ‘for’ loop version.”
Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

One point came up multiple times: easier to read than write.

▶ “The good thing is, once you figure it out, it’s clear why it works. It’s not magic, you just have to get the mapping right.”

▶ “If I ask a student to read my code, he’ll be able to read it. But five minutes later, he’ll try something similar and it won’t work.”
Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

One point came up multiple times: easier to read than write.

▶ “The good thing is, once you figure it out, it’s clear why it works. It’s not magic, you just have to get the mapping right.”

▶ “If I ask a student to read my code, he’ll be able to read it. But five minutes later, he’ll try something similar and it won’t work.”

Write-only Read-only code???
Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are “statically typed at runtime,” so I’m extending Numba to recognize and JIT-compile them.

  This will permit fast (compiled), **imperative** (for-loop style) calculations in Python.
Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are “statically typed at runtime,” so I’m extending Numba to recognize and JIT-compile them. This will permit fast (compiled), imperative (for-loop style) calculations in Python.

- Possible Google Summer of Code project to add precompiled and/or CUDA implementations, depending on the abilities and interests of the student.
Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are “statically typed at runtime,” so I’m extending Numba to recognize and JIT-compile them.

  This will permit fast (compiled), imperative (for-loop style) calculations in Python.

- Possible Google Summer of Code project to add precompiled and/or CUDA implementations, depending on the abilities and interests of the student.

- Michael Hedges (Purdue) is developing Pandas extensions, so that DataFrame columns can contain and operate on jagged data.
Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are “statically typed at runtime,” so I’m extending Numba to recognize and JIT-compile them.

  This will permit fast (compiled), imperative (for-loop style) calculations in Python.

- Possible Google Summer of Code project to add precompiled and/or CUDA implementations, depending on the abilities and interests of the student.

- Michael Hedges (Purdue) is developing Pandas extensions, so that DataFrame columns can contain and operate on jagged data.

- Giuseppe Cerati (Fermilab) is investigating the use of jagged arrays in C++, to write reconstruction algorithms that are equally efficient on CPUs and GPUs.

  Status: implemented track-propagation in Python and switched from CPU to GPU with from "import numpy as np" → "import cupy as np". Reimplemented in C++ using xtensor (Numpy clone for C++).
Conclusions

Awkward-array is a library for complex data, presented as arrays. Jagged arrays are the most important for HEP. (Perhaps the only type necessary for HEP?)

We’re beyond single-operation tests; we’re implementing complete analyses. Performance is within a factor of two of C++, and there’s low-hanging fruit for improvements.

Physicists find it hard to write, but easy to read.

This is an open area of development with many paths to follow!