Recent developments in the Awkward Array world

Peter Fackeldey[1], Iason Krommydas[2], Ianna Osborne[1], Jim Pivarski[1], Andres Rios-Tascon[1]
Princeton University[1], Rice University[2]

🏷️ Named axis for Awkward Arrays

You can now add named axis (or algebraic shapes) to `ak.Array`, use them with awkward's operations, and leverage a new named axis based indexing syntax:

```python
import awkward as ak

array = ak.Array([[1, 2], [3], [], [4, 5, 6]], named_axis=("x", "y"))

ak.sum(array, axis="y")
# <Array [3, 3, 0, 15] x:0 type='4 * int64'>

array[{"y": np.s_[0:1]}]
# <Array [[1], [3], [], [4]] x:0,y:1 type='4 * var * int64'>
```

Named axis provide more safety and readability for array manipulations with Awkward Array. For more details, check out the named axis documentation.

✨ Virtual Arrays for Awkward Array

Awkward Array does support virtual arrays, i.e. representing not-yet-loaded arrays in memory. These arrays are loaded on-demand when Awkward Array needs them to perform an operation. An example of virtual arrays in the scope of the coffea 2025 project is shown in the following:

```python
import awkward as ak
from coffea.nanoevents import NanoEventsFactory

# construct a coffea.NanoEvents collection without reading!
events = NanoEventsFactory.from_root(
    {"nanoaod.root": "Events"},
    mode="virtual",
    access_log=(log := []),
).events()


print(events.Jet.pt)
# [??, ??, ??, ??, ..., ??, ??, ??, ??]
```

`??` indicates that the values are not-yet-loaded into memory. You can access the values of these virtual arrays in two ways:

1. Explicitly load them into memory using `ak.materialize()`:

```python
print(ak.materialize(events.Jet.pt))
# [[50.1, 47.3], ..., [62.4, ..., 16]]
```

2. Implicitly through any operation that needs the values:

```python
print(events.Electron.pt)
# [??, ??, ??, ??, ..., ??, ??, ??, ??]
print(events.Electron.pt > 40.)
# [[True, True], ..., [True, False]]
```

Finally, let's make sure we only loaded the columns we need from the file:

```python
print(log)
# ['nJet', 'Jet_pt', 'nElectron', 'Electron_pt']
```

The new virtual arrays in Awkward Array v2 have been carefully implemented at the lowest level enabling highly granular laziness. This integrates well with modern file formats like ROOT's RNTuple that allows reading data at the same level of granularity.

📄 RNTuple support in Uproot (reading and writing)

ROOT's RNTuple format is a modern file format for columnar data storage. Uproot supports reading (since v5.5.1) and writing (since v5.6.0) RNTuple v1.0.0.0 files.

An example of inspecting an RNTuple file with Uproot is shown in the following:

```python
import uproot

file = uproot.open("staff_rntuple_v1-0-0-0.root")
print(file.classnames())
# {'Staff;1': 'ROOT::RNTuple'}
```

Reading the RNTuple's contents is as easy as reading TTrees with Uproot:

```python
rntuple = file["Staff"]

print(rntuple["Age"].array())
# [58, 63, 56, 61, 52, 60, 53, ..., 38, 26, 51, 25, 35, 28, 43]
print(rntuple.arrays(["Age", "Cost", "Nation"]))
# [{Age: 58, Cost: 11975, ...}, ..., {Age: 43, Cost: 12716, ...}]
```

(full link to this rntuple file can be found here.)

🔥 Performance gains in Awkward Array and Vector

Several performance improvements have been made in Awkward Array and Vector. For example, the trijet mass reconstruction of the *Analysis Grand Challenge* (AGC) runs ~15-20% faster since Awkward Array v2.7.3. This is achieved by reducing the metadata overhead on the Python side of Awkward Array for any `ak.layout.RecordArray`, see the improvements for the trijet mass reconstruction with 100.000 events below:

	Relative improvement
CPU backend:	
• Runtime: 29.3 ms ± 434 µs → **25.1 ms ± 126 µs**	**14.3%**
• Allocations: 3691 → **2633**	**39.3%**
TypeTracer backend:	
• Runtime: 30.3 ms ± 145 µs → **24.4 ms ± 195 µs**	**19.5%**
• Allocations: 2882 → **1816**	**36.9%**

The Vector library has also seen performance improvements with v1.6.1 and newer. By bypassing metadata overhead and grouping operations, the performance of the *whole* Vector library is often improved by factors of 2 and more, see the p_T calculation of a four-vector below:

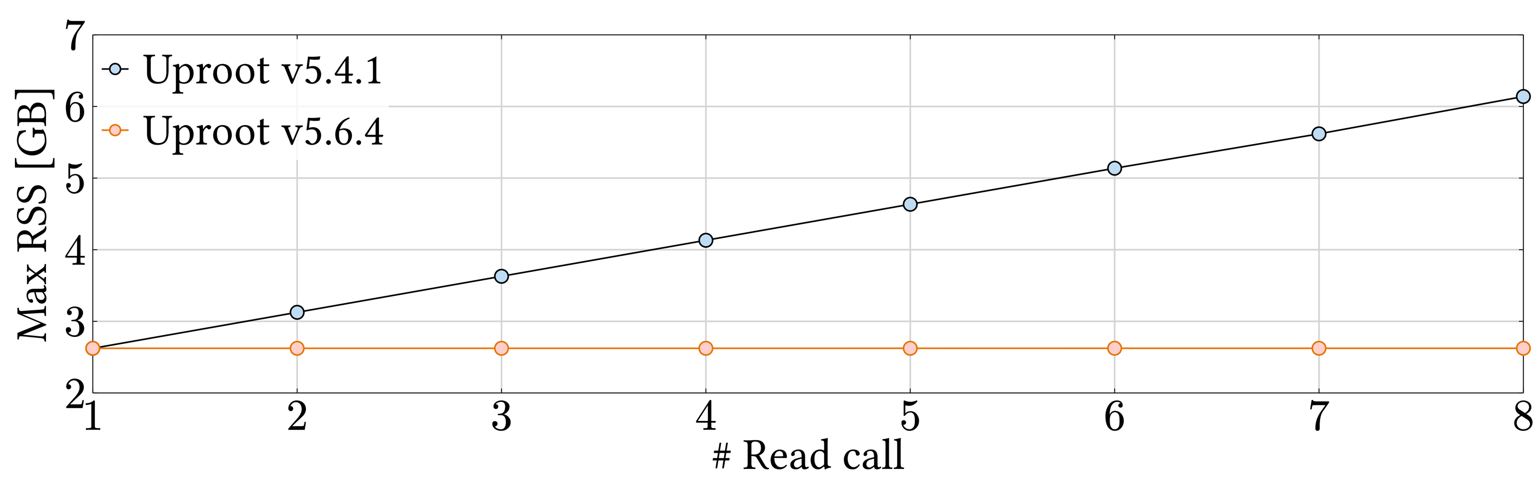
```python
import vector
vector.register_awkward()

vec = vector.awk([{"x": 1.0, "y": 2.0, "z": 3.0, "t": 4.0}])

# pip install --upgrade "vector>1.6.0"
%timeit vec.rho
# 305 µs ± 659 ns

# pip install --upgrade "vector<1.6.0"
%timeit vec.rho
# 731 µs ± 2.26 µs
```

These Vector improvements further bring the runtime of the trijet mass reconstruction of the AGC with 100.000 events down to **19.1 ms ± 203 µs** (**18.2 ms ± 49.6 µs**) with the CPU (TypeTracer) backend.

🧠 Memory improvements in Awkward Array and Uproot

Several memory improvements have been made in Awkward Array and Uproot resolving cyclic references on the Python side. This allows for more efficient memory management and thus usually reduces the memory footprint of physics analyses.

Especially, the memory footprint of performing multiple reads of the same file with Uproot has been improved significantly since v5.4.1, essentially eliminating growing memory usage, see the following figure:



📎 Additional Information and Tips

Checkout the new features and improvements in the Awkward Array world!

If you're already a coffea user: `coffea v2025.7` (July, CalVer) release (and newer) includes all of these improvements.

Stay up-to-date and follow our projects on GitHub at https://github.com/scikit-hep.

Acknowledgements: This work was supported by the National Science Foundation under Cooperative Agreement PHY-2323298 and grant OAC-2103945

Contact:
peter.fackeldey@cern.ch