Coffea Framework

Current Status, Recent Updates, and Community Impact

Iason Krommydas, Lindsey Gray, Peter Fackeldey, Nick Manganelli, Nick Smith // PyHEP.dev 2026 // September 7th 2026

What is Coffea?

What is Coffea?

- Coffea stands for Columnar Object Framework For Effective Analysis
- It is "Basic tools and wrappers for enabling not-too-alien syntax when running columnar Collider HEP analysis"
- It is a package that pulls together typical needs for HEP analyses
- It uses the scientific python ecosystem and builds on top of Uproot and Awkward Array to provide an array-based syntax for manipulating HEP event data in an efficient and numpythonic way
- It supplies submodules for specific needs
- It supplies tools for horizontal scaling
- It is fully up-to-date with the scientific python ecosystem when it comes to dependencies

Let's see it in action

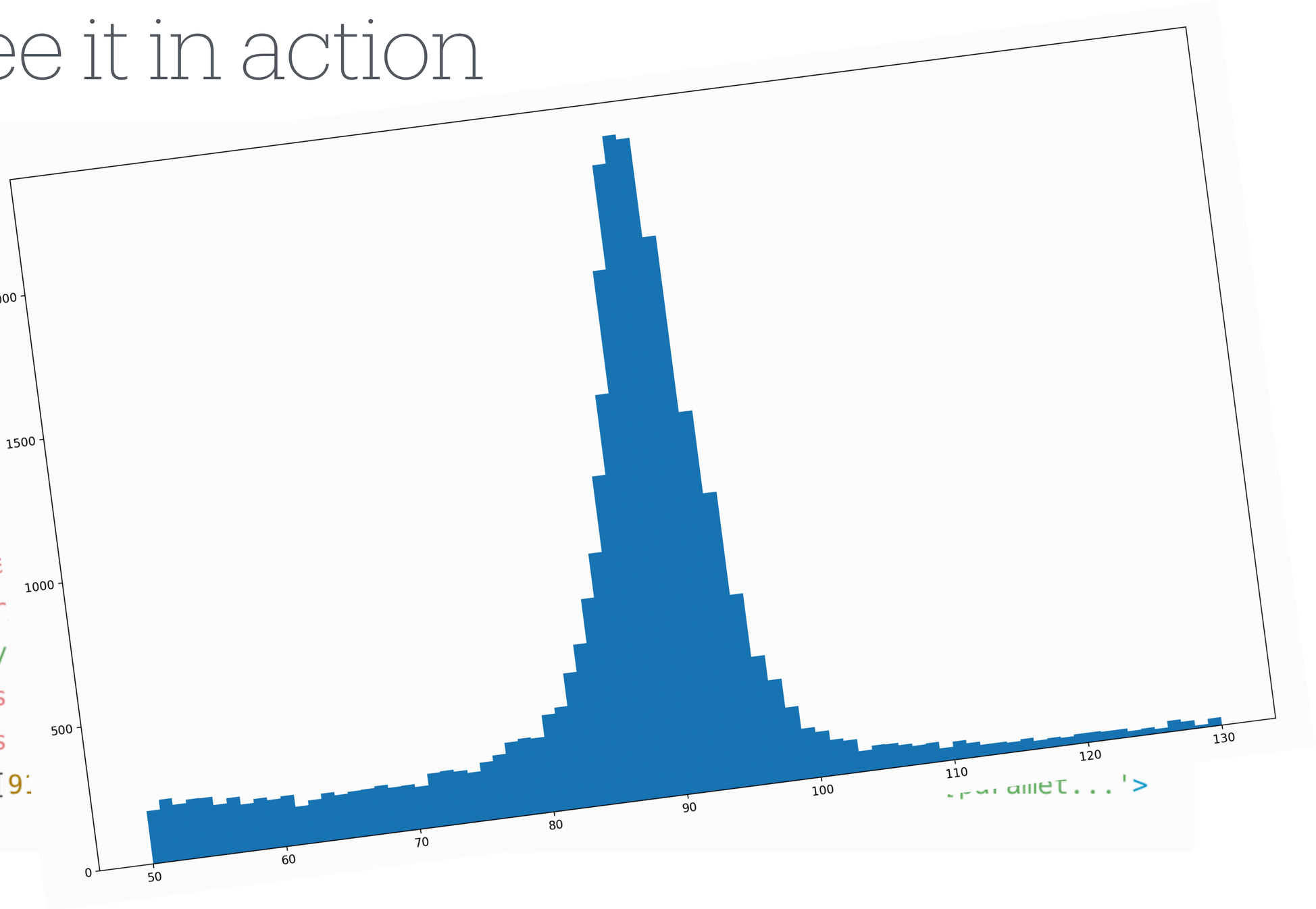
```
>>> from coffea.nanoevents import NanoEventsFactory, NanoAODSchema
>>> import awkward as ak
>>> events = NanoEventsFactory.from_root({"dy.root": "Events"}, schemaclass=NanoAODSchema).events()
>>> events
<NanoEventsArray [<NanoAOD event>, ...] type='143657 * event'>
>>> events.Electron
<ElectronArray [??, ??, ??, ??, ..., ??, ??, ??, ??] type='143657 * var * E...'>
>>> events.Electron.pt
<Array [??, ??, ??, ??, ..., ??, ??, ??, ??] type='143657 * var * float32[p...'>
>>> events.Electron.pt * 1
<Array [[], [], [], ..., [45, ...], [50, 41.3]] type='143657 * var * float3...'>
>>> events.Photon.pt
<Array [??, ??, ??, ??, ..., ??, ??, ??, ??] type='143657 * var * float32[p...'>
>>> ak.materialize(events.Photon.pt)
<Array [[], [], ..., [43.1, ...], [49.7, 41]] type='143657 * var * float32[...'>
```

Let's see it in action

```
>>> zcands = ak.combinations(events.Electron, 2, fields=["tag", "probe"])
>>> zcands
<Array [[], [], ..., [{tag: {...}, ...}]] type='143657 * var * {tag: Electr...'>
>>> zcands.fields
['tag', 'probe']
>>> pair_4mom = zcands.tag + zcands.probe
>>> pair_4mom
<CandidateArray [[], ..., [{x: -5.96, y: 7.07, ...}]] type='143657 * var * ...'>
>>> pair_4mom.fields
['x', 'y', 'z', 't', 'charge']
>>> mass = ak.flatten(pair_4mom.mass)
>>> mass
<Array [91, 130, 89.1, ..., 27.2, 90.8, 91.5] type='37909 * float32[paramet...'>
```

Let's see it in action

```
>>> z
>>> zc
<Array
>>> zc
['tag',
>>> pai
>>> pai
<Candida
>>> pair
['x', 'y
>>> mass
>>> mass
<Array [9
```



NanoEvents and Schemas

- All this "magic" is done by NanoEvents and its schemas
- The schemas tell NanoEvents how to structure the array
- The schemas are also responsible for attaching behaviors to the objects. For example we attach pt-eta-phi-mass 4-vector behaviors to particle collections that are defined by those variables
- Various schemas exist for various experiments and file formats
- Users can implement their own schemas

Laziness

- NanoEvents is lazy and utilizes Awkward Array's virtual arrays feature for this. See my presentation at ACAT 2025 about this feature

- We recommend all analyses to migrate to that while dask and dask-awkward are now optional dependencies and in maintenance mode

```
>>> events = NanoEventsFactory.from_root({"dy.root": "Events"}, schemaclass=NanoAODSchema, access_log=(access_log := [])).events()
>>> access_log
[]
>>> zcands = ak.combinations(events.Electron, 2, fields=["tag", "probe"])
>>> access_log
[Accessed(branch='nElectron', buffer_key='3a249176-c1dd-11ee-829f-ca090d0abeef/%2FEvents%3B1/0-143657/offsets/
nElectron%2C%21load%2C%21counts2offsets%2C%21skip%2C%21offsets')]
>>> mass = ak.flatten(zcands.tag + zcands.probe).mass
>>> access_log
[Accessed(branch='nElectron', buffer_key='3a249176-c1dd-11ee-829f-ca090d0abeef/%2FEvents%3B1/0-143657/offsets/
nElectron%2C%21load%2C%21counts2offsets%2C%21skip%2C%21offsets'), Accessed(branch='Electron_pt', buffer_key='3a249176-
c1dd-11ee-829f-ca090d0abeef/%2FEvents%3B1/0-143657/data/Electron_pt%2C%21load%2C%21content'), Accessed(branch='Electron_phi',
buffer_key='3a249176-c1dd-11ee-829f-ca090d0abeef/%2FEvents%3B1/0-143657/data/Electron_phi%2C%21load%2C%21content'),
Accessed(branch='Electron_eta', buffer_key='3a249176-c1dd-11ee-829f-ca090d0abeef/%2FEvents%3B1/0-143657/data/
Electron_eta%2C%21load%2C%21content'), Accessed(branch='Electron_mass', buffer_key='3a249176-c1dd-11ee-829f-ca090d0abeef/
%2FEvents%3B1/0-143657/data/Electron_mass%2C%21load%2C%21content'), Accessed(branch='Electron_charge', buffer_key='3a249176-
c1dd-11ee-829f-ca090d0abeef/%2FEvents%3B1/0-143657/data/Electron_charge%2C%21load%2C%21content')]
>>> mass
<Array [91, 130, 89.1, ..., 27.2, 90.8, 91.5] type='37909 * float32[paramet...'>
```

New NanoEvents features

Vector behavior validation

- The vector behaviors are inherited from scikit-hep/vector which is very strict regarding what variables can be present on a vector. For example, energy and mass conflict with each other. Vector silently chooses a basis and ignores the other

- We now raise explicit errors

```
>>> events = NanoEventsFactory.from_root({"dy.root": "Events"}, schemaclass=NanoAODSchema).events()
>>> events.Electron
<ElectronArray [??, ??, ??, ??, ..., ??, ??, ??, ??] type='143657 * var * E...'>
>>> events.Electron["energy"] = events.Electron.pt * np.cosh(events.Electron.eta)
File "/Users/iason/Dropbox/work/pyhep_dev/coffea/src/coffea/nanoevents/methods/vector.py", line 575, in __awkward_validation__
    raise ValueError(f"{type(self).__name__}: " + "; ".join(errors))
ValueError: ElectronArray: multiple temporal aliases present: ['energy', 'mass']

This error occurred while calling

    ak.Array.__setitem__(
        repr-raised-ValueError
        where = 'energy'
        what = <Array [[], [], ..., [57.1, 43.1]] type='143657 * var * floa...'>
    )
```

Tracing and Preloading

- Virtual arrays are loaded one by one. It's often more efficient to make a single network request for multiple branches over the network

- We can trace the necessary branches without needing any data and preload them with a single network request

```
from coffea.nanoevents.trace import trace

def analysis(events):
    # MY EXPENSIVE ANALYSIS CODE
    return things

events = NanoEventsFactory.from_root({"nanoaod.root": "Events"}).events()

# trace does not load any data
needed = trace(analysis, events)

events = NanoEventsFactory.from_root({"nanoaod.root": "Events"}, preload=needed).events()

# now actually run the analysis
analysis(events)
```

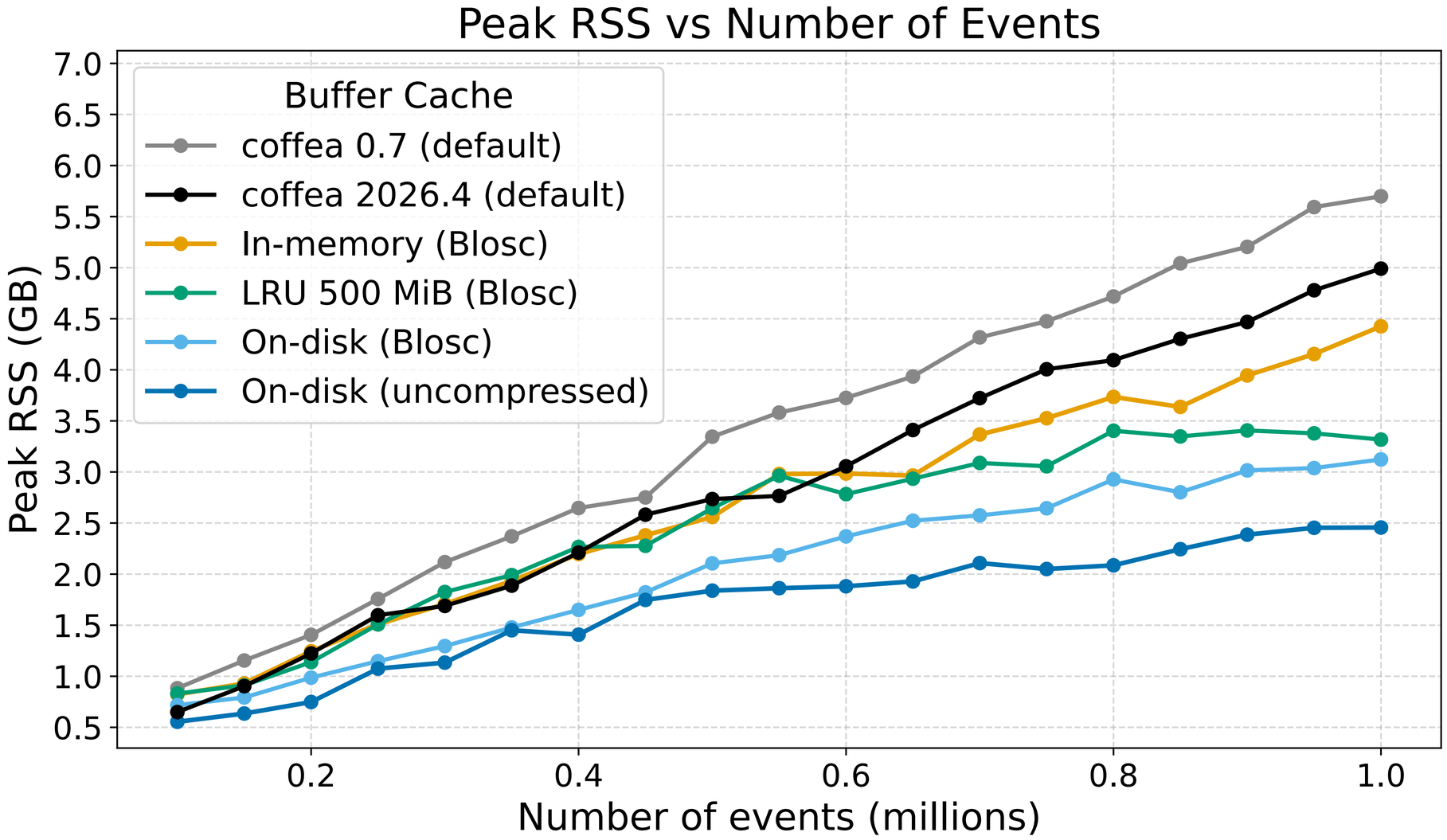
Buffer caches

- Cache and optionally compress buffers and evict them from the cache if needed

```python
from coffea.nanoevents import NanoEventsFactory, NanoAODSchema
from coffea.nanoevents.mapping import BufferCache
from numcodecs import Blosc
import zict

# --- choose a backend ---
lru      = zict.LRU(n=500 * 1024**2, d={}, weight=lambda k, v: len(v))          # 500 MiB in-memory LRU
ondisk   = zict.File("./cache_dir")                                             # on-disk
hier     = zict.Buffer(fast={}, slow=ondisk, n=100, weight=lambda k, v: len(v))  # hierarchical

# --- optionally compress cached buffers ---
codec = Blosc("zstd", clevel=1, shuffle=Blosc.BITSHUFFLE)

# --- wrap in BufferCache and attach to NanoEventsFactory ---
buffer_cache = BufferCache(cache=lru, codec=codec)

events = NanoEventsFactory.from_root({"dy.root": "Events"}, schemaclass=NanoAODSchema, buffer_cache=buffer_cache).events()

# first access reads from disk and populates the cache
ak.materialize(events.Jet.pt)

# subsequent accesses are served from cache
ak.materialize(events.Jet.pt)
```

Buffer caches

- We saw big improvements in memory usage



Parquet reading

- Parquet reading in NanoEvents has been revived from coffea 0.7

```
>>> from coffea.nanoevents import NanoEventsFactory, NanoAODSchema, BaseSchema
>>> events = NanoEventsFactory.from_parquet("tests/samples/nano_dy.parquet", schemaclass=BaseSchema).events()
>>> events.Electron_pt
<Array [??, ??, ??, ??, ..., ??, ??, ??, ??] type='40 * var * float32[param...'>
>>> events.Electron_eta
<Array [??, ??, ??, ??, ..., ??, ??, ??, ??] type='40 * var * float32[param...'>
>>> events = NanoEventsFactory.from_parquet("tests/samples/nano_dy.parquet", schemaclass=NanoAODSchema).events()
>>> events.Electron.pt
<Array [??, ??, ??, ??, ..., ??, ??, ??, ??] type='40 * var * float32[param...'>
>>> events.Electron.eta
<Array [??, ??, ??, ??, ..., ??, ??, ??, ??] type='40 * var * float32[param...'>
```

Other coffea features

Other coffea features

- Coffea has various modules and utilities that are not in the scope of this presentation
- Luminosity tools
- Boolean masking and weight handling tools including N-1 and cutflow selection tools
- Tools to apply corrections/systematics at the event- and object-level
- JME tools
- Dataset tools
- BTagging tools
- ML tools
- Most of those are CMS specific, possibility to expand to other experiments?

Structuring your analysis

The processor

```
from coffea import processor

# --- 1. The Processor: describe what to do with a chunk of events ---
class MyAnalysis(processor.ProcessorABC):
    def process(self, events):
        dataset = events.metadata["dataset"]
        # ... your analysis logic ...
        return {"nevents": len(events), "hist": h}

    def postprocess(self, accumulator):
        return accumulator

fileset = {
    "DoubleMuon": {"files": {"data.root": "Events"}},
    "ZZ to 4mu":  {"files": {"mc.root":   "Events"}},
}
```

The runner/executor

```
# --- 2. The Executor: apply the processor across all chunks ---

# laptop: single core
run = processor.Runner(executor=processor.IterativeExecutor())

# laptop: multiple cores — only the executor changes
run = processor.Runner(executor=processor.FuturesExecutor(workers=4))

# cluster: connect to a Dask cluster — only the executor changes
from dask.distributed import Client
client = Client("tls://localhost:8786")  # e.g. coffea-casa
run = processor.Runner(executor=processor.DaskExecutor(client=client))

out = run(fileset, processor_instance=MyAnalysis())
```

The processor, the executor, and the runner

- The runner applies the processor across all chunks of events from all input datasets and accumulates all the results using the executor

- Scaling from your laptop to a cluster requires very few code changes. You only need to change the execution part

- Having a copy of the histograms per worker and accumulating them is a problem. You may want to see Peter's talk about histogramming as a service

New Processor/Executor features

Tracing and Preloading with runners

```python
from coffea import processor
from coffea.nanoevents import NanoAODSchema
from coffea.nanoevents.trace import trace

fileset = {
    "ZJets": {"files": {"zjets.root": "Events"}},
    "Data":  {"files": {"data.root":  "Events"}},
}

# per-dataset preload specified directly in the fileset
fileset_with_preload = {
    "ZJets": {"preload": ["nJet", "Jet_pt"],   "files": {"zjets.root": "Events"}},
    "Data":  {"preload": ["nMuon", "Muon_pt"], "files": {"data.root":  "Events"}},
}

run = processor.Runner(
    executor=processor.IterativeExecutor(),
    schema=NanoAODSchema,
)

# option 1: preload from fileset metadata
out = run(fileset_with_preload, MyAnalysis())

# option 2: let coffea trace the processor and preload automatically
out = run(fileset, MyAnalysis(), trace=trace)
```

Buffer caches with runners

- In the future we can probably have caches shared between workers or even between analyzers that are perhaps in the same group and analyze the same data often

```
codec = Blosc("zstd", clevel=1, shuffle=Blosc.BITSHUFFLE)

# cachestrategy must be a callable that returns a MutableMapping
def make_cache():
    lru = zict.LRU(n=500 * 1024**2, d={}, weight=lambda k, v: len(v))
    return BufferCache(cache=lru, codec=codec)

run = processor.Runner(
    executor=processor.FuturesExecutor(workers=4),
    schema=NanoAODSchema,
    cachestrategy=make_cache,  # called once per chunk/worker
)

out = run(fileset, MyAnalysis())
```

Checkpointing

- Checkpointing functionality allows us to save the output of a processor so that when we re-process, we can ignore the already successful chunks and process/accumulate the rest

```
from coffea import processor
from coffea.nanoevents import NanoAODSchema

fileset = {
    "ZJets": {"treename": "Events", "files": ["zjets.root"]},
    "Data":  {"treename": "Events", "files": ["data.root"]},
}

checkpointer = processor.SimpleCheckpointer("./checkpoints")

run = processor.Runner(
    executor=processor.IterativeExecutor(),
    schema=NanoAODSchema,
    checkpointer=checkpointer,
)

# retry until all chunks succeed
while True:
    try:
        out = run(fileset, MyAnalysis())
        break
    except Exception:
        continue  # failed chunks are retried; completed ones load from checkpoint
```

Community Impact

Community Impact

- We see that coffea is reasonably successful
- We estimate a few hundred regular users
- They appear to like the interactive, pythonic, array-oriented experience
- However, many users are still using coffea 0.7 as opposed to CalVer coffea
- This is basically the reason virtual arrays were implemented

Community Impact

- I migrated HiggsDNA (CMS Higgs to diphoton framework) from coffea 0.7 to CalVer in 1 hour
- Feedback from other framework developers (PocketCoffea and Pepper and others) tell us that migrating from coffea 0.7 to CalVer is typically a matter of changing/fixing a few patterns only in the codebase
- CalVer dask-awkward to CalVer virtual arrays migration is not as trivial (as the compute model changes) but easily doable
- Users (including me) have noticed big memory usage improvements versus coffea 0.7
- We also notice that CalVer coffea is typically faster but that's because with CalVer coffea, you get a newer ecosystem in general (awkward 2, numpy 2, newest vector, etc)
- We very highly recommend that everyone upgrades to CalVer coffea with virtual arrays

Discussion

- What should coffea be and where should it head towards?
- What things from other experiments fit the scope of coffea?

Thank you!

Backup

Preloading

- Virtual arrays are loaded one by one. It's often more efficient to make a single network request for multiple branches over the network. We can explicitly preload certain branches

```
# explicitly preload "nJet" and "Jet_pt"
preload = {"nJet", "Jet_pt"}

events = NanoEventsFactory.from_root({"nanoaod.root": "Events"}, access_log=(access_log := []), preload=preload).events()

# this is expensive as it wasn't preloaded, needs to be loaded on the spot
%timeit -n 1 -r 1 ak.materialize(events.Jet["eta"])
# 468 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

# this is much faster because it's preloaded
%timeit -n 1 -r 1 ak.materialize(events.Jet["pt"])
# 2.19 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

# check what has been accessed
print(access_log)
# [Accessed(branch='nJet', buffer_key='d0c59d98-9b3e-11ee-baab-0103fea9beef/%2FEvents%3B1/0-918697/offsets/
nJet%2C%21load%2C%21counts2offsets%2C%21skip%2C%21offsets'), Accessed(branch='Jet_eta', buffer_key='d0c59d98-9b3e-11ee-
baab-0103fea9beef/%2FEvents%3B1/0-918697/data/Jet_eta%2C%21load%2C%21content'), Accessed(branch='Jet_pt',
buffer_key='d0c59d98-9b3e-11ee-baab-0103fea9beef/%2FEvents%3B1/0-918697/data/Jet_pt%2C%21load%2C%21content')]
```

Tracing

- Awkward Array has a feature where you can make a "tracer" array to trace the necessary branches without needing actual data. We can use it to find out the necessary branches

```
def analysis(events):
    # MY EXPENSIVE ANALYSIS CODE
    ...
    return things

from coffea.nanoevents import NanoEventsFactory
from coffea.nanoevents.trace import trace

events = NanoEventsFactory.from_root({"nanoaod.root": "Events"}).events()

necessary_columns = trace(analysis, events)  # makes a "tracer" out of events that holds no data and runs analysis

print(necessary_columns)
# frozenset({'Electron_pt', 'Jet_eta', 'Jet_btagDeepFlavB', 'nJet', 'Electron_eta', 'Electron_jetIdx', 'Muon_phi',
# 'Jet_pt', 'nElectron', 'nMuon', 'Muon_pt', 'Muon_eta', 'Jet_phi', 'Jet_mass'})
```

Buffer manager

- Rough ideas for a buffer manager. Needs internal discussion. No implementation yet

```
def process(self, events):
  manager = coffea.nanoevents.BufferManager(events)

  jets = events.Jet[events.Jet.pt > 50.0]

  # frees the underlying memory of the materialized buffers of `events.Jet.pt`
  manager.clear(events.Jet.pt)


def process(self, events):
  manager = coffea.nanoevents.BufferManager(events)

  # starts prefetching all `events.Jet` columns in background threads
  with manager.prefetch(events.Jet, nthreads=4):
    out = my_analysis(events)

  return out
```