The Case for Columnar Analysis (a two-part series)

Nick Smith, on behalf of the Coffea team
Lindsey Gray, Matteo Cremonisi, Bo Jayatilaka, Oliver Gutsche, Nick Smith, Allison Hall, Kevin Pedro (FNAL); Andrew Melo (Vanderbilt); and others

In collaboration with iris-hep members:
Jim Pivarski (Princeton); Ben Galewsky (NCSA); Mark Neubauer (UIUC)

HOW 2019
21 Mar. 2019
Prologue: terminology

• Event loop analysis:
  - Load relevant values for a specific event into local variables
  - Evaluate several expressions
  - Store derived values
  - Repeat (explicit outer loop)

• Columnar analysis:
  - Load relevant values for many events into contiguous arrays
    • Nested structure (array of arrays) → flat content + offsets
      - This is how TTree works!
  - Evaluate several **array programming** expressions
    • Implicit *inner* loops
  - Store derived values
Prologue: technology

• Array programming:
  - Simple, composable operations
  - Extensions to manipulate offsets
  - *Not declarative* but towards goal

• Awkward array programming:
  - Extension of numpy syntax
  - Variable-length dimensions: “jagged arrays”
  - View SoA as AoS, familiar object syntax, e.g. p4.pt()
  - References, masks, other useful extensions
  - See awkward, talk by J. Pivarski at ACAT2019

• Coffea framework:
  - Prototype analysis framework utilizing columnar approach
  - Provide lookup tools, histogramming, other ‘missing pieces’ usually
  - See fnal-column-analysis-tools
    • Functionality will be factorized as it matures
Part I: Analyzer Experience
User experience

• Unsurprisingly, #1 user priority
  - Any working analysis code can scale up…for now
  - c.f. usage of PyROOT event loops despite dismal performance
    • (this will never change)

• Fast learning curve for scientific python stack
  - Excellent ‘google-ability’
  - The quality and quantity of off-the-shelf components is impressive—many analysis tool implementations contain very little original code
  - Essentially all functions available in a vectorized form

• Challenge: re-frame problem in array programming primitives rather than imperative style (for+if)
  - User interviews conducted:
    • “its different, not necessarily harder”
    • “easier to read than write” ?!
Code samples I

• Idea of what $Z$ candidate selection can look like
• Python allows very flexible interface, under-the-hood data structure is columnar

```python
ele = electrons[(electrons.p4.pt > 20) &
    (np.abs(electrons.p4.eta) < 2.5) &
    (electrons.cutBased >= 4)]
mu = muons[(muons.p4.pt > 20) &
    (np.abs(muons.p4.eta) < 2.4) &
    (muons.tightId > 0)]
```

• Selects good candidates (per-entry selection)

```python
ee = ele.distincts()
mu = mu.distincts()
em = ele.cross(mu)
```

• Creates pair combinatorics (creates new pairs array, also jagged)

```python
channels['ee'] = good_trigger & (ee.counts == 1) & (mu.counts == 0)
channels['mm'] = good_trigger & (mm.counts == 1) & (ele.counts == 0)
channels['em'] = good_trigger & (em.counts == 1) & (ele.counts == 1) & (mu.counts == 1)
```

• Selects good events, partitioning by type (per-event selection)

```python
dileptons['ee'] = ee[(ee.i0.pdgId*ee.i1.pdgId == -11*11) & (ee.i0.p4.pt > 25)]
dileptons['mm'] = mm[(mm.i0.pdgId*mm.i1.pdgId == -13*13)]
dileptons['em'] = em[(em.i0.pdgId*em.i1.pdgId == -11*13)]
```

• Selects good pairs, partitioning by type (per-entry selection on pairs array)
Code samples II

- Enable expressive abstractions without python interpreter overhead
  - e.g. storing boolean event selections from systematic-shifted variables in named bitmasks: each add() line operates on O(100k) events

```python
shiftSystematics = ['JESUp', 'JESDown', 'JERUp', 'JERDown']
shiftedQuantities = {'AK8Puppijet0_pt', 'pfmet'}
shiftedSelections = {'jetKinematics', 'jetKinematicsMuonCR', 'pfmet'}
for syst in shiftSystematics:
    selection.add('jetKinematics'+syst, df['AK8Puppijet0_pt_'+syst] > 450)
    selection.add('jetKinematicsMuonCR'+syst, df['AK8Puppijet0_pt_'+syst] > 400.)
    selection.add('pfmet'+syst, df['pfmet_'+syst] < 140.)
```

- Columnar analysis is a lifestyle brand
  - Opens up scientific python ecosystem. e.g. interpolator from 2D ROOT histogram:

```python
def centers(edges):
    return (edges[:-1] + edges[1:])/2

h = uproot.open("histo.root")['a2dhisto']
xedges, yedges = h.edges
xcenters, ycenters = np.meshgrid(centers(xedges), centers(yedges))
points = np.hstack([xcenters.flatten(), ycenters.flatten()])
interp = scipy.interpolate.LinearNDInterpolator(points, h.values.flatten())
x, y = np.array([1, 2, 3]), np.array([3., 1., 15.])
interp(x, y)
```

- Don’t want linear interpolation? Try one of several other options
Domain of applicability

• Domain of applicability depends on:
  - Complexity of algorithms
  - Size of per-event input state

• Examples:
  - JEC (binned parametric function): use binary search, masked evaluation: **columnar ok**
  - Object gen-matching, cross-cleaning: min(metric(pairs of offsets)): **columnar ok**
  - Deterministic annealing PV reconstruction: large input state, iterative: **probably not**

• How far back can columnar go?
  - *Missing array programming primitives not a barrier, can always implement our own*

---

event loop

<table>
<thead>
<tr>
<th>Event Reconstruction</th>
<th>Analysis Objects</th>
<th>Filtering &amp; Projection</th>
<th>Empirical PDFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB/evt</td>
<td>40-400 kB/evt</td>
<td>1 kB/evt</td>
<td>(histograms)</td>
</tr>
<tr>
<td>Complex algorithms operating on large per-event input state</td>
<td>Fewer complex algorithms, smaller per-event input state</td>
<td>Few complex algorithms, O(1 column) input state</td>
<td>No event scaling</td>
</tr>
<tr>
<td>Inter-event SIMD</td>
<td></td>
<td></td>
<td>Trivial operations</td>
</tr>
</tbody>
</table>
Scalability

- Present a unified data structure to analysis function or class
  - Dataframe of awkward arrays
  - Decouple data delivery system from analysis system

- We can run real-world analyses at a range of scales
  - With home-grown and commercial scheduler software

- Lessons learned so far:
  - Fast time-to-backtrace as important as time-to-insight, keep in mind for analysis facilities!
  - Physics-driven bookkeeping (dataset names, cross sections, storage of derived data, etc.) is nontrivial in all cases, needs to be decoupled
  - Inherently higher memory footprint, solved by adjusting partitioning (chunking) scheme

- Tradeoff with data delivery overhead

<table>
<thead>
<tr>
<th>Data delivery system</th>
<th>Z peak wall-time throughput</th>
<th>Subjective ‘ease of use’</th>
</tr>
</thead>
<tbody>
<tr>
<td>uproot on laptop</td>
<td>~100 kHz</td>
<td>5/5</td>
</tr>
<tr>
<td>uproot + xrootd + multiprocessing</td>
<td>~250 kHz @ 10 cores *</td>
<td>5/5</td>
</tr>
<tr>
<td>uproot + condor jobs</td>
<td>Arbitrary</td>
<td>3/5</td>
</tr>
<tr>
<td>striped system</td>
<td>~10 MHz @ 100 cores</td>
<td>2/5</td>
</tr>
<tr>
<td>Apache spark</td>
<td>~1 MHz @ 100 cores **</td>
<td>4/5</td>
</tr>
</tbody>
</table>

* constrained by bandwidth
** pandas_udf issue
Part II: Technical Underpinnings
Theoretical Motivation

• Aligned with strengths of modern CPUs
  - Simple instruction kernels aid pipelining, branch prediction, and pre-fetching
  - Event loop = input data controlling instruction pointer = less likely to exploit all three!
    - *Unnecessary work is cheaper than unusable work*
• Inherently SIMD-friendly
  - Event loop cannot leverage SIMD unless inter-event data sufficiently large
• In-memory data structure *exactly* matches on-disk serialized format
  - Event loop must transform data structure - significant overhead
  - Memory consumption managed by chunking (event groups, or baskets)
• Array programming kernels form computation graph
  - Could allow query planning, automated caching, non-trivial parallelization schemes
The Coffea framework

- Column Object Framework For Effective Analysis:
  - Prototype analysis framework utilizing columnar approach
  - Provides object-class-style view of underlying arrays
  - Implements typical recipes needed to operate on NANOAOD-like nTuples
  - One monolith for now: `final-column-analysis-tools`
    - Functionality will be factorized into targeted packages as it matures

- Realized using scientific python ecosystem
  - numpy: general-purpose array manipulation library
  - numba: uses llvm to JIT-compile python code, understands numpy
    - Work ongoing to extend to awkward arrays as well
  - scipy: large library of specialized functions
  - cloudpickle: serialize arbitrary python objects, even function signatures
  - matplotlib: python visualization library
Factorized Data Delivery

• **Uproot**
  - Direct conversion from TTree to numpy arrays and/or awkward JaggedArrays

• **Striped**
  - NoSQL database delivers ‘stripes’: numpy arrays
    • Re-assemble awkward structure via object counts + content
  - memcached layer, python job scheduler, ~150 core cluster
  - Derived columns persistable

• **Spark**
  - Interface using vectorized UDF (user-defined function)
  - Currently restricted to intermediate pandas format (pyarrow UDF to be implemented)
  - Derived columns persistable
Prototype analyses are using the workflow in blue:
- `fcat = fnal-column-analysis-tools`
- Future pyHEP ecosystem analysis packages in grey.
Performance

• Z peak benchmark
  - Includes many typical corrections: lumimask, PU correction, ID scale factors, flavor-categorized
  - 350 lines jupyter notebook, 25 columns accessed
  - 6 µs/evt/thread (125 kHz) wall time
    • ROOT C++ TBranch::GetEntry(): ~1.5x faster

• Two prototype analyses
  - “end-to-end” = NanoAOD-like nTuple to templates
  - Varies from 30-150 µs/evt/thread
  - Already being used to steer analysis, present results in analysis group meetings

• Many inefficiencies known
  - Can be removed with further development in awkward and helper libraries
Future Directions

• As Coffea (& underlying libraries) matures, invite beta testers
  - I encourage everyone to try uproot+numpy now

• Target first release this summer
  - Two full analysis implemented
  - Data delivery mechanisms fully separated
  - User interface improvements and documentation

• Far future: analysis facility
  - This feeds towards the dream of a “short time-to-insight” “analysis as a service” facility
    • Tendering bids for additional buzzwords
  - Array programming allows easier construction of computation graphs
    • Query planning can detect common patterns and execute them once
    • By removing manual cache management, we can optimize throughput and storage

• First, lets see if we are happy and productive with the columnar approach
  - So far, the answer appears to be yes