The Coffea project

Nick Smith, on behalf of the Coffea team:

Lindsey Gray, Matteo Cremonesi, 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)

DPF 2019
30 July 2019

DOI 10.5281/zenodo.3345994
Physics - The primary motivation

• The challenges
  - Analyze all LHC Run 2 data: ~billion events
  - No longer needle-in-haystack searches
    • Separating hay from straw
  - Investigate issues with fast time-to-insight
  - Optimize complex (e.g. deep learning) algorithms
  - These challenges magnified 100x in HL-LHC

• Solutions must be:
  - Easy to use
  - Scalable
  - Fast

• My analysis: probing high-pT Higgs
  - Separate Higgs jets from QCD jets

- arXiv:1709.05543
The Coffea project

- Provides an easy to use columnar analysis environment
- Provides physicist-friendly interfaces to multiple scale-out mechanisms
- Provides performant code
- Provides an easy to deploy package: `pip install coffea`
  - See [coffea](https://github.com/coffea/groot) github repository for more details
What is columnar analysis?

• 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 the ROOT TTree internal data structure!
  - Evaluate several array programming expressions
    • Implicit inner loops
  - Store derived values
Columnar analysis ecosystem

- Array programming is a mainstay of data science industry:
  - **Data structure**: numpy, arrow
  - **Data exploration**: pandas, matplotlib, …
  - **Algorithms**: numpy, numba, …
  - **Machine Learning**: tensorflow, pytorch, …
  - **Big Data**: spark, dask, …

- Leverage industry tools, extend to fit our needs:
  - **Data structure**: awkward array
  - **Data exploration**: coffea histograms
  - **Algorithms**: coffea lookup tools
  - **Big Data**: coffea processor
Technology

• Awkward array:
  - Extension of numpy array programming syntax
  - Jagged arrays: variable-length dimensions
  - Table arrays: view SoA as AoS
    • Allows object method syntax, e.g. p4.pt()
  - Masked arrays, chunked arrays, lazy arrays, other extensions
  - See awkward, talk by J. Pivarski at ACAT2019

• Coffea framework:
  - Tools for common HEP analysis algorithms
  - Histograms and other accumulators
  - Processor framework
    • Write analysis code once, scale out in multiple ways
  - Any ‘missing pieces’ commonly used in HEP analysis
  - See coffea
Domain of applicability

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

• Examples:
  - JEC (binned parametric function): binary search, masked evaluation: columnar ok
  - Object gen-matching, cross-cleaning: \( \min(\text{metric}(\text{pairs of offsets})) \): 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</td>
<td>Fewer complex</td>
<td>Few complex</td>
<td>No event</td>
</tr>
<tr>
<td>operating on large</td>
<td>algorithms,</td>
<td>algorithms, O(1 column)</td>
<td>scaling</td>
</tr>
<tr>
<td>per-event input</td>
<td>smaller per-</td>
<td>input state</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>event input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inter-event SIMD</td>
<td>state</td>
<td></td>
<td>Trivial</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operations</td>
</tr>
</tbody>
</table>
User experience

• User time more expensive than CPU time
  - Any working analysis code can scale up on condor (for now)
  - c.f. usage of PyROOT event loops despite dismal performance

• Profit from wider user base
  - Excellent ‘google-ability’
  - More time exploring physics data
  - Less duplicated effort writing common algorithms
  - Skills transfer to data science industry

• Profit from wider developer base
  - Follow established conventions in the scientific python ecosystem
  - User interface more likely to be intuitive
Analyst interface

- Jupyter notebooks
  - Combine source code and results in one document
  - More effective for data exploration
- Traditional CLI
  - Available for established processing workflows
Package ecosystem

- Prototype analyses are using the workflow in blue
  - Future pyHEP ecosystem analysis packages in grey
Prototype analyses are using the workflow in blue
- Future pyHEP ecosystem analysis packages in grey
Scaling out

ROOT files
Parquet files
…

map

reduce

coffea.processor

Histories
Event lists
…

coffea executor
Scaling out - processor

• User code encapsulation:
  - Python class inheriting from ProcessorABC
  - Define accumulators
    • Histograms
    • Counters
    • (Small) output arrays
  - Process a common input data structure
    • Dataframe of awkward arrays
  - Return filled copy of accumulators
  - Optionally post-process the reduced accumulator output

• Once defined, processor can be passed to different executors with a single line change

```python
class DimuonProcessor(processor.ProcessorABC):
    def __init__(self):
        dataset_axis = hist.Cat("dataset", "Primary dataset")
        mass_axis = hist.Bin("mass", "\(m_{\mu\mu}\) [GeV]", 30000, 0.25, 300)

        self._accumulator = processor.dict_accumulator(
            'mass': hist.Hist('Counts', dataset_axis, mass_axis),
            'cutflow': processor.defaultdict_accumulator(int),
        )

    @property
def accumulator(self):
        return self._accumulator

    def process(self, sel, df):
        output = self.accumulator.identity()

        dataset = df['dataset']
        muons = JaggedCandidateArray.candidatesfromcounts(
            df['nMuon'],
            pt=df['Muon_pt'].content,
            eta=df['Muon_eta'].content,
            phi=df['Muon_phi'].content,
            mass=df['Muon_mass'].content,
            charge=df['Muon_charge'].content,
        )

        output['cutflow']['all events'] += muons.size
        twomuons = (muons.counts == 2)
        output['cutflow']['two muons'] += twomuons.sum()

        opposite_charge = twomuons & (muons['charge'].prod() == -1)
        output['cutflow']['opposite charge'] += opposite_charge.sum()

        dimuons = muons[opposite_charge].distincts()
        output['mass'].fill(dataset=dataset, mass=dimuons.mass.flatten())

        return output

def postprocess(self, accumulator):
    return accumulator
```

Launch in nbViewer
Scaling out - executors

- **Uproot**
  - Direct conversion from ROOT TTree to awkward arrays
  - Python multiprocessing enabled with one argument
  - Good solution to work with small (<100 GB) datasets on single node

- **Spark**
  - Conversion from TTree to Arrow-backed RDD via [laurelin](#)
  - Scalable to hundreds of cores, tree reduction of accumulators
  - Cluster deployment possible via kubernetes, industry-standard tech.

- **Parsl**
  - Parallelizes arbitrary python function execution on clusters
  - Support for condor, slurm, MPI, k8s, …
  - Utilizes uproot to read TTree data

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

- All executors accept ProcessorABC instances
Scaling out - lessons learned

• Every cluster is unique!
  - Scale-out + deployment mechanism combinatorics leads to new issues at each site
    • Successes with: condor@FNAL, parsl+condor@FNAL, spark@FNAL, parsl+condor@Wisconsin, parsl+slurm@Nebraska, spark@CERN, parsl+slurm@DESY, …

• We can run real-world analyses at a range of scales
  - 10 GB up to 10 TB demonstrated
  - With home-grown and commercial scheduler software

• Factorizing the data delivery enables
  - Fast local prototyping
  - Seamless scale-out (assuming the cluster deployment is successful)

• Intermediate scale resources are more important
  - A 100 core-hour resource can satisfy most CMS end-user analysis needs
  - Clusters must provide <10 min time-to-insight to be useful
Performance comparison

• Z peak benchmark compared to ROOT
  - 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
  - Half the time spent in the uproot executor
    • Other executors may have different performance
  - Some awkward-array kernels are expensive
    • Work ongoing to port more of awkward-array to C++
Path for future performance improvements

• 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
  - More portable to GPUs (almost just cuPy)
• 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
Conclusions

• Columnar analysis is effective for HEP use cases
  - Real-world CMS analyses implemented
  - Prototype analysis at ATLAS, DUNE, etc. ongoing

• Provides physicist-friendly interfaces to multiple scale-out mechanisms
  - Work ongoing for additional mechanisms

• Provides performant code
  - Users write high-level operations, vectorized code stays in library

• Try it yourself: pip install coffea
Backup
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
ele = ele.distincts()
mu = mu.distincts()
em = ele.cross(mu)
```

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

```python
channels['ee'] = good_trigger & (ele_counts == 1) & (mu_counts == 0)
channels['mm'] = good_trigger & (mu_counts == 1) & (ele_counts == 0)
channels['em'] = good_trigger & (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 == -1*1) & (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