coffea - Columnar Object Framework For Effective Analysis

Nick Smith, on behalf of the coffea team:

Lindsey Gray, Matteo Cremonesi, Bo Jayatilaka, Oliver Gutsche, Nick Smith, Allison Hall, Kevin Pedro, Maria Acosta (FNAL); Andrew Melo (Vanderbilt); Stefano Belforte (INFN); and others

In collaboration with iris-hep members:

Jim Pivarski (Princeton); Ben Galewsky (NCSA); Mark Neubauer (UIUC)

CHEP 2019

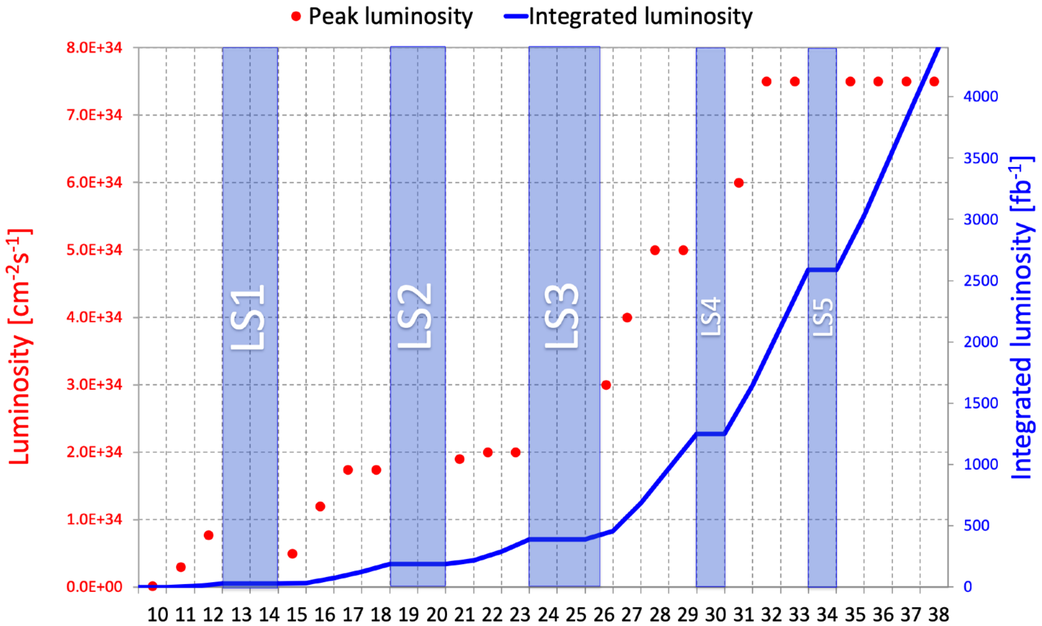
4 November 2019

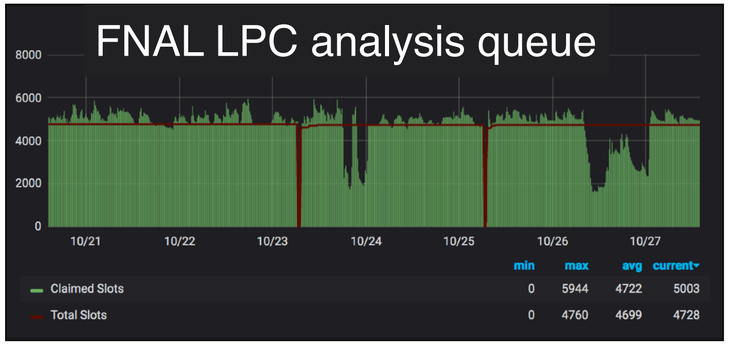
DOI 10.5281/zenodo.3345994

Physics - The motivation

- The present challenge:
  - Analyze all LHC Run 2 data: O(10 billion) events
  - Investigate data quality issues with fast time-to-insight
  - Optimize complex (e.g. deep learning) algorithms
- Multiply by O(1000) data analysts
- These challenges magnified 20x in HL-LHC

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





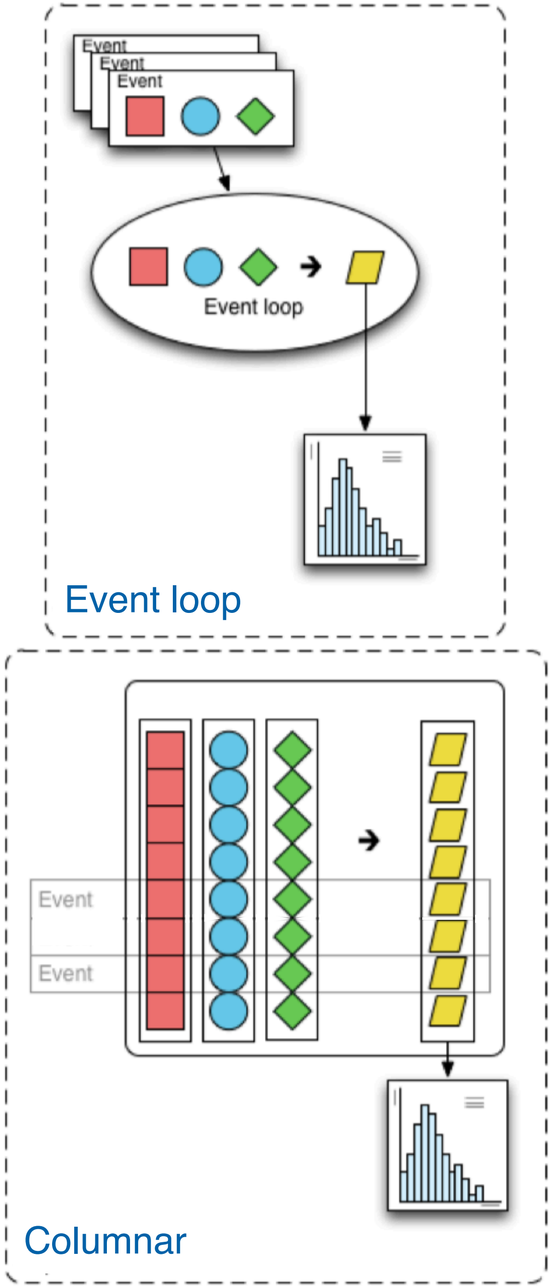
Fermilab

Columnar analysis - The paradigm

- 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
  - Evaluate several **array programming** expressions
    - Implicit *inner* loop**s**
  - Store derived values

- Coffea started 1 year ago with one goal: perform CMS analyses *at scale* using columnar analysis techniques



Coffea - The solution

Coffea is:

- A package in the scientific python ecosystem
  - `$ pip install coffea`
- A user interface for columnar analysis
  - With missing pieces of the stack filled in
- A minimum viable product
  - We are data analyzers too #dogfooding
- A really strong glue
  - To be glued in:

Boost Histogram

pyhf

fast-carpenter

Coffea

Fermilab

CMS

Coffea - The solution

= CHEP contribution (link)

Coffea is:

- A package in the scientific python ecosystem
  - `$ pip install coffea`
- A user interface for columnar analysis
  - With missing pieces in the stack filled in
- A minimum viable product
  - We are data analyzers too #dogfooding
- A really strong glue
  - To be glued in:

Boost Histogram

pyhf

fast-carpenter

Coffea

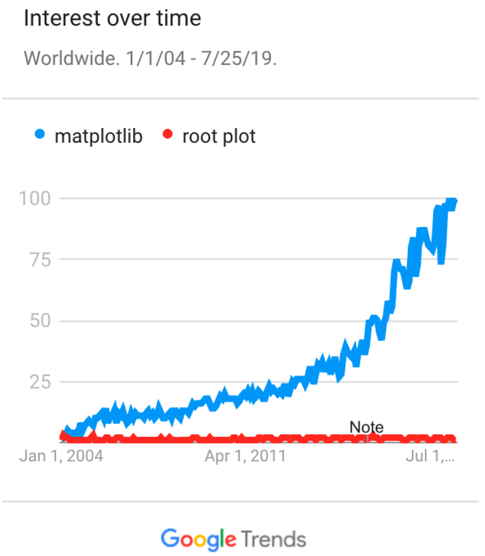
Visualization	Coffea, matplotlib, mplhep
Algorithms	SciPy, Numba, Coffea
Array API	Apache Arrow, NumPy, Awkward Array
Data ingestion	Laurelin, ServiceX, uproot
Task scheduler	Apache Spark, DASK, Striped, Parsl
Resource provisioning	kubernetes, HTCondor, slurm workload manager, etc.

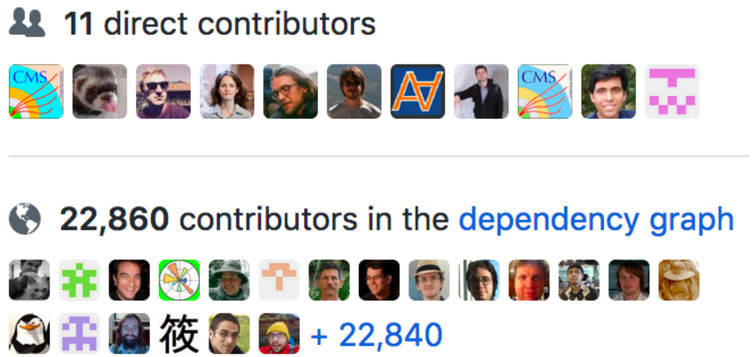
User experience

- 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

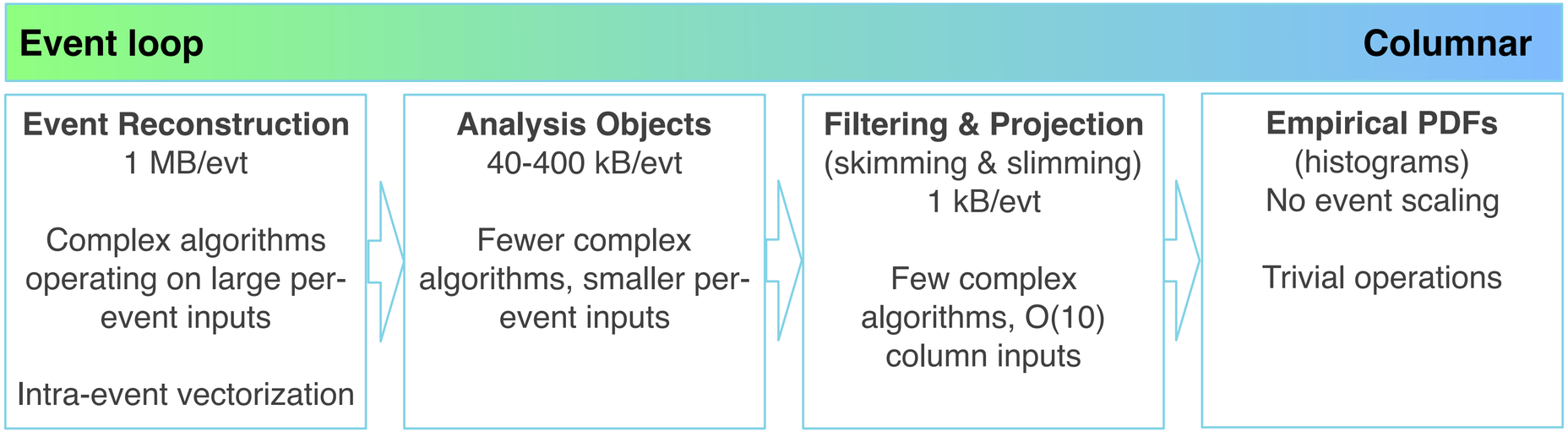




Fermilab

Domain of applicability

- Is columnar analysis always the best approach? No:
  - Too complex or too much intra-event data makes array programming a headache
  - When intra-event data is large, we can already use vectorized approaches in event loop
- What is the limit on complexity?
  - In principle, none—missing array programming primitives can easily be implemented
  - In practice, on IRIS-HEP analysis benchmark tasks, only one required a new primitive
  - https://nbviewer.jupyter.org/github/mat-adamec/coffea-benchmarks/tree/master/benchmarks/
  - CMS NanoAOD schema (*incl. cross-references*) is fully describable with awkward-array



Fermilab

Analyst interface

- Jupyter notebooks
  - Combine source code and results in one document
  - More effective for data exploration
- Traditional CLI, scripts
  - Best for established processing workflows
  - Can integrate user python libraries

jupyter muonspectrum_v3 Last Checkpoint: Last Wednesday at 7:43 PM (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

In [3]:

```
tstart = time.time()

fileset = {
    'DoubleMuon': [
        'data/Run2012B_DoubleMuParked.root',
        'data/Run2012C_DoubleMuParked.root',
    ]
}

output = processor.run_uproot_job(fileset,
                                  treename='Events',
                                  processor_instance=DimuonProcessor(),
                                  executor=processor.futures_executor,
                                  executor_args={'workers': 4},
                                  chunksize=500000,
                                 )

elapsed = time.time() - tstart
print(output)
```

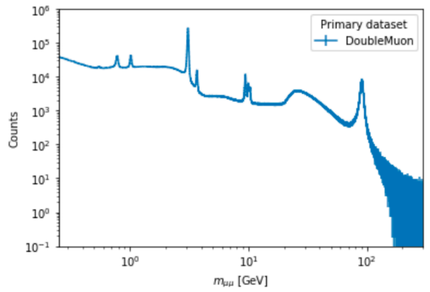
```
Preprocessing: 100%|██████████| 1/1 [00:00<00:00,  5.84it/s]
Processing: 100%|██████████| 133/133 [01:19<00:00,  1.77items/s]
```

```
{'mass': <Hist (dataset,mass) instance at 0x7fe9d028cb70>, 'cutflow': defaultdict(<class 'int'>, {'all events':
66128870, 'two muons': 33370298, 'opposite charge': 25794885}), '_bytesread': <coffea.processor.accumulator.accumulat
or object at 0x7fe9e122db38>}
```

In [4]:

```
fig, ax, _ = hist.plot1d(output['mass'], overlay='dataset')
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_ylim(0.1, 1e6)
```

Out[4]: (0.1, 1000000.0)



render nbviewer Open in SWAN (select 96Python3 stack)

Fermilab

Scalability - The coffea processor

- User is provided data frame of columns they wish to process
  - Lazily evaluated access in uproot-based executors
- User fills a defined set of accumulators
  - Histograms, dictionaries of counts, appendable arrays, …
- Coffea executor takes care of the rest
  - Row chunking (a.k.a. partitioning or job splitting)
  - Scale-out to many workers
  - Input caching (support varies by executor)
  - Tree reduction of accumulators
- Supported executors:
  - Local machine, dask cluster, spark cluster, parsl cluster (and condor)

```python
from coffea import hist, processor

class MyProcessor(processor.ProcessorABC):
    def __init__(self, flag=False):
        self._flag = flag
        self._accumulator = processor.dict_accumulator({
            "sumw": processor.defaultdict_accumulator(float),
        })

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

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

        # PHYSICS GOES HERE

        return output

    def postprocess(self, accumulator):
        return accumulator

p = MyProcessor()
```

Fermilab

Scaling out - lessons learned

- Every cluster is unique!
  - Executor + resource provisioning combinatorics leads to new issues at each site
  - Examples: network firewall between workers; filesystem parallel read limits; user code packaging.
  - **However, no issue required changes to user code**
- We run real-world analyses at a range of scales
  - 10 GB up to 10 TB demonstrated
- Factorizing the data delivery enables
  - Fast local prototyping
  - Seamless scale-out
- Intermediate scale resources are more important
  - An O(100) core-hour resource can execute a full CMS Run 2 columnar analysis
  - If executor is to provide <10 min time-to-insight, **resources must be dedicated or provisioned quickly**!

Looking forward

- As ecosystem matures, coffea will be factorized into separate packages
  - Subpackages that remain useful will stay
  - Some subpackages will be replaced by better alternatives
- The next frontier for resource and productivity gains is a multi-user analysis facility
  - Shared input cache at column granularity
  - Intermediate result cache
  - Unified metadata and schema database
  - Exportable derived columns
    - “slim and skim” with coffea, do the rest however you like
  - And more
- Coffea farms!
  - Plan to transition from current user-provisioned resources to a simple-to-deploy multi-user cluster
  - Primary technology focus has been Spark, but plans in place to test other schedulers

Conclusions

- Columnar analysis is effective for HEP analyst use cases
  - 10 CMS analysis groups have implemented or are implementing their analysis in coffea
    - Opportune moment: switch from private nTuples to NanoAOD
  - Prototype efforts in ATLAS, DUNE ongoing

- Columnar analysis enables performant code
  - Users write high-level operations, vectorized code stays in library

- Coffea simplifies interface to scale-out mechanisms

- Try it yourself: `pip install coffea`
  - Or poke around: launch binder

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

```
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)

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

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

```
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)

```
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

```
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:

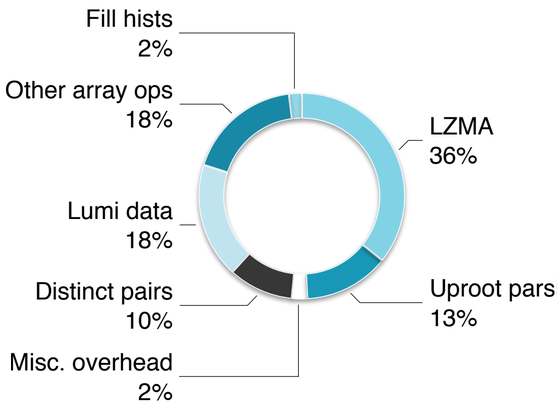
```
def centers(edges):
    return (edges[:-1] + edges[1:])/2

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

Per-thread performance

- 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 reading
    - Other executors may have different performance
  - Some awkward-array kernels are expensive
    - Work ongoing to port awkward-array to C++
      - https://github.com/scikit-hep/awkward-1.0




```
    def run(self, events, job):
        times = OrderedDict()
        times['start'] = time.time()

        if self.weights_eval is None:
            self.weights_eval = cloudpickle.loads(zlib.decompress(job["weights_eval"]))
        if self.lumimask is None:
            self.lumimask = cloudpickle.loads(zlib.decompress(job["lumimask"]))


        dataset = job['dataset']
        hists = job['hists']

        isRealData = False
        good_trigger = np.zeros(events.nevents, dtype=bool)
        if dataset in self.triggers:
            isRealData = True
            # Real data, prevent overlaps
            for trigger in self.triggers[dataset]:
                good_trigger |= getattr(events, trigger)
            # apply 'golden json' certified lumiblocks
            good_trigger &= lumimask(events.run, events.luminosityBlock)
            # record lumis processed
            #  in uproot jobs, full lumi completion guaranteed by full-file processing
            #  in striped jobs, partial lumi completion is not traceable at present
            if dataset == "DoubleEG":
                hists['lumis_ee'] += LumiList(events.run, events.luminosityBlock)
            elif dataset == "DoubleMuon":
                hists['lumis_mm'] += LumiList(events.run, events.luminosityBlock)
        else:
            # MC, OR all
            for tlist in self.triggers.values():
                for trigger in tlist:
                    good_trigger |= getattr(events, trigger)
            times['lumimask, trigger'] = time.time()
            genw = np.sign(events.LHEWeight_originalXWGTUP)


        electronCols = PhysicalColumnGroup(events, "Electron", **{k:k for k in self.electron_cols})
        electrons = jaggedFromColumnGroup(electronCols)
        electrons['weight'] = weights_eval["eleScaleFactor_TightId_POG"](electrons.p4.eta, electrons.p4.pt)

        muonCols = PhysicalColumnGroup(events, "Muon", **{k:k for k in self.muon_cols})
        muons = jaggedFromColumnGroup(muonCols)
        muons['weight'] = weights_eval["muScaleFactor_TightId_Iso"](np.abs(muons.p4.eta), muons.p4.pt)

        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)]

        times['good leptons'] = time.time()

        ee = ele.distincts()
        mm = mu.distincts()
        em = ele.cross(mu)

        dileptons = {}
        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)]

        times['good pairs'] = time.time()

        channels = {}
        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)

        times['channels'] = time.time()

        dupe = np.zeros(events.nevents, dtype=bool)
        tot = 0
        for channel, cut in channels.items():
            zcands = dileptons[channel][cut]
            dupe |= cut
            tot += cut.sum()
            weight = np.array(1.)
            if not isRealData:
                weight = zcands.i0['weight'] * zcands.i1['weight'] * genw[cut]
                hists['genw'].fill(dataset=dataset, genw=genw)
            hists['lepton_pt'].fill(dataset=dataset, channel=channel,
                                    lep0_pt=zcands.i0.pt.flatten(),
                                    lep1_pt=zcands.i0.pt.flatten(),
                                    weight=weight.flatten(),
                                    )
            hists['zMass'].fill(dataset=dataset, channel=channel,
                                mass=zcands.p4.mass.flatten(),
                                weight=weight.flatten(),
                                )

        if dupe.sum() != tot:
            raise Exception("Double-counting events!")

        times['plots'] = time.time()

        #profiling info
        t = list(times.values())
        for i, name in enumerate(times):
            if i==0: continue
            dt = t[i] - t[i-1]
            hists['profile'].fill(op=name, dt=1e6*(dt/len(events.Muon.count)))
        hists['profile'].fill(op="total", dt=1e6*(t[-1]-t[0])/events.nevents)

        job.send(hists=hists)
```

Scale-out performance

- Spark tests at FNAL show MHz-level performance with 500 cores for a full analysis code
  - Reading ~500 GB of ROOT TTree data over xrootd through laurelin

In [6]:
```
import time
from coffea.processor import run_spark_job
from coffea.processor.spark.spark_executor import spark_executor

tic = time.time()
final_accumulator = run_spark_job(datasets_spark, processor_instance, spark_executor,
                                  spark=spark, partitionsize=partitionsize, thread_workers=thread_workers,
                                  executor_args={'file_type': 'root', 'cache': True})
dt = time.time() - tic

```

```
loading:   0%|          | 0/44 [00:00<?, ?datasets/s]
pyspark version: 2.4.3
loading: 100%|██████████| 44/44 [00:51<00:00,  1.16s/datasets]
Processing: 100%|██████████| 44/44 [01:37<00:00,  2.21s/datasets]
```

1.29 MHz (no cache)

<- 3.15 MHz (from cache)

- Parsl tests at 500-core scale already show <10m turnaround, more work to be done here

In [ ]:
```
import time
from fnal_column_analysis_tools.processor import run_parsl_job
from fnal_column_analysis_tools.processor.parsl.parsl_executor import parsl_executor

tic = time.time()
treenames = ['otree', 'Events']  # deal with mixed skims and full derived trees
final_accumulator = run_parsl_job(datasets, treenames, processor_instance, parsl_executor,
                                  executor_args={'config':None}, data_flow=dfk, chunksize=chunksize)
dt = time.time() - tic
```

```
Processing: 100%|██████████| 1355/1355 [08:17<00:00,  1.04s/items]
```


Fermilab

CMS