2 aspects of future software with ATLAS

Introduction

For Run3 and beyond software in ATLAS is evolving significantly

Focusing on 2 selected topics related to reconstruction and analysis

1) Next configuration system for reconstruction (and analysis!)
2) Columnar based analysis

… trying to find a balance between overview and important details

New configuration for Athena

Configuration in Athena

- An Athena job is composed of several C++ components (~C++ classes)
  - Algorithm : executed once per event by Athena
    - ex: “build EMPFlow jets”
  - Tools : piece of code for a specific task. Shared amongst algorithms
    - ex: “calculate a jet width” ← used by each jet alg
- Components have properties which are configurable (==can be changed without recompiling)
  - properties are just members of the c++ class
  - can be simple types (int, float, bool, string, vector<int>,…)
  - … or pointers to Tool, vector<pointers to Tool>, …
- Configuring an Athena job == putting together all the components, with their properties, in the right order

Configuration in Athena

- Configuration is written in python
- Each c++ component has an equivalent python class
  - generated automatically after compilation
- write python scripts :
  - instantiate python Tools & Algs, then add to the global sequence
- Athena executes the scripts, then translate python instances to C++

```
from MyPackage.MyPackageConf import MyAlg, CalcToolA

athAlgSeq += MyAlg("AName", Prop1 = 3,
                    ToolProp = CalcToolA() )
```

Configuration in Run II

- A main script includes domain specific scripts (a.k.a "jobOptions") according to "configuration flags"
- domain scripts add their algs to the main sequence according to flags

```
if rec.doCalo:
    include("CaloRec_jobOptions.py")
if rec.doEGamma :
    include("EGamma_jobOptions.py")
```

- Tens of domains, hundreds of flags, thousands of algs & tools !
- Algs depends on each other
  - dependencies fulfilled thanks to flags and careful manual ordering
- Not very robust, hard to setup partial reconstruction

Configuration in Run III

New config system aiming at automatically solving dependencies at config level. Relying on

- Multi-Threading Scheduler (C++ side) : automatically orders & runs algorithm
  - makes use of component properties to understand dependencies
- A new ComponentAccumulator object (python side)
  - A container of algs which knows how to prevent duplication of algorithms

Components, dependencies & properties

- In order to organize parallel execution of algs, the **scheduler** must know what the algs
  - require as input
  - produce as output

ex : requires “ElectronContainer”
produces decoration “EMFrac” onto “AntiKT4EMPFlowJets”

- Each component declares this info through properties

```
class myAlg : public AthAlgorithm {
  …
SG::ReadHandleKey<xAOD::JetContainer> m_jetkey= {this, "JetContainer", "AntiKt4LCtopoJets", "doc"};

SG::WriteDecorHandleKey<xAOD::JetContainer> m_decorXYZ={this, "DecoXYZ", "AntiKt4LCtopoJets.DecoXYZ", 
""};
};
```

ComponentAccumulator

- Python config object to accumulate algs without duplication

```
                Example : configure an alg working with Electrons AND photons

from AthenaConfiguration.ComponentAccumulator import ComponentAccumulator
from ... import GammaCfg
from ... import ElectronCfg
from MyPackageCong import EGammaCombinerAlg

def EGammaCombinerCfg(flags):
    acc = ComponentAccumulator()

    # Require electrons :
    acc.merge( ElectronCfg(flags) )

    # Require photons :
    acc.merge( GammaCfg(flags) )

     # add our own alg :
    acc.addEventAlgo( EgammaCombinerAlg("egammacomb") )
    return acc
```

Internally invoke
`CaloClusterCfg`

ComponentAccumulator

- Python config object to accumulate algs without duplication

Example : configure an alg working with Electrons AND photons

```
from AthenaConfiguration.ComponentAccumulator import ComponentAccumulator
from ... import GammaCfg
from ... import ElectronCfg
from MyPackageCong import EGammaCombinerAlg

def EGammaCombinerCfg(flags):
    acc = ComponentAccumulator()

    # Require electrons :
    acc.merge( ElectronCfg(flags) )

    # Require photons :
    acc.merge( GammaCfg(flags) )

     # add our own alg :
    acc.addEventAlgo( EgammaCombinerAlg("egammacomb") )
    return acc
```

Internally invoke `CaloClusterCfg`

clusters are required

clusters are required again BUT merge() avoids duplication !

Clients need **only** to invoke :
`acc.merge( EGammaCombinerCfg(flags))`

RunIII configuration summary

- ComponentAccumulator mechanism allows to build a hierarchy of XYZCfg() function calls
  - effectively solving all dependencies
  - without duplication
- MT scheduler allows to run algs in parallel and correct order

Much simpler and robust configuration !

This has a price :

- Higher complexity for package developers
  - write thread safe algs & tools
  - deal with Read/WriteDecorHandle
- **Much** higher complexity for core developers !

Deployment status :

- many (all?) domains have CA-based config ready
- already possible to invoke RunIII style config from RunII jobOptions
- full switch still not there yet

Columnar based analysis

Analysis coding pattern in HEP

pick an event

- Read in information
  - ex: electrons 4-vector
- Reject event or..
- Calculate quantities and fill histograms

repeat with next event

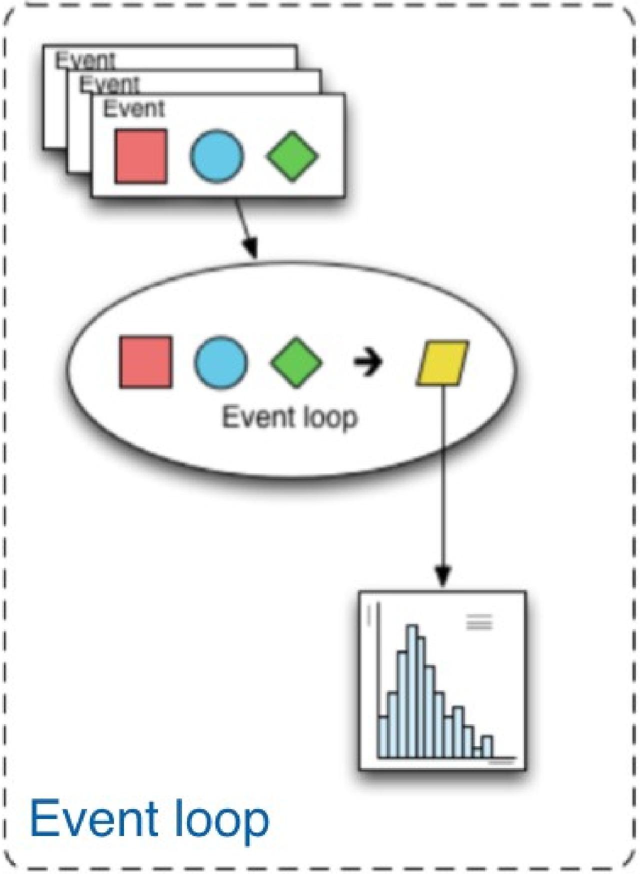
Event loop

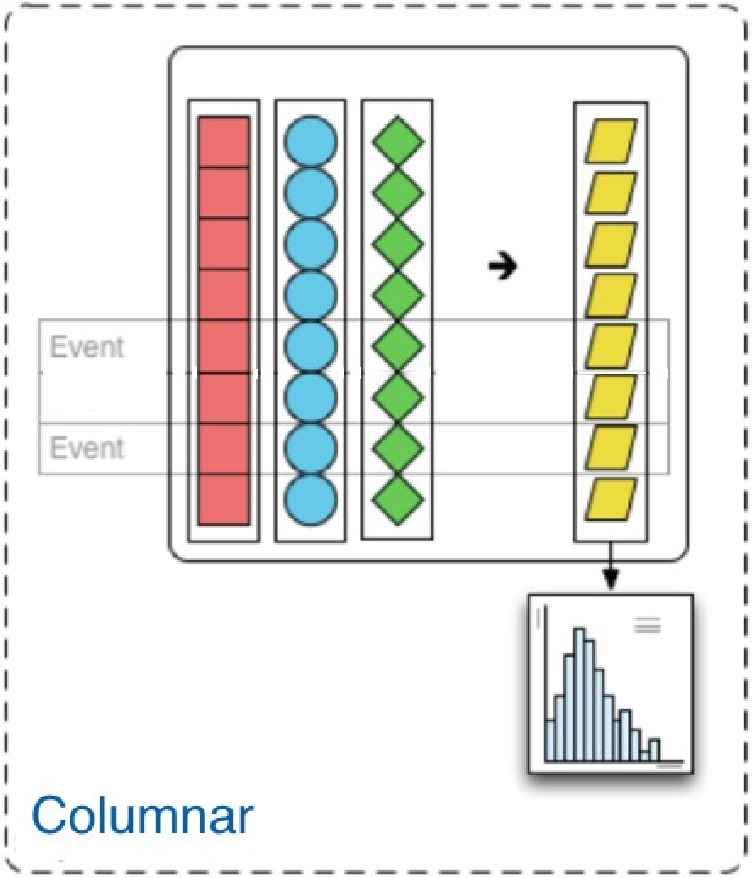
An alternative coding pattern

- Assume we can get analysis quantities in different big arrays (each 1 array entry per event)

```
import numpy as np

el1_pt = readFromFile("pt of leading el for each event")
# same for el1_E, el2_pt, el2_E etc…

# select events based on electron pt
validEvents = (el1_pt > 50) and (el2_pt > 30)

# calculate invariant mass of selected events
invM = np.sqrt( 2*el1_E[validEvents]*el2_E[validEvents]*(1-el_cos12[validEvents]) )

# create histogram of invariant mass
h = np.hist( invM, bins=200, range=(0,500) )
```

- Valid python/numpy code
- Similar to what is used in many other scientifics domain, including preprocessing ML data !


Event
Event
Event
Event loop
Event loop


Event
Event
Columnar

Why columnar analysis ?

- Do not write event loop, concentrate on physics code
- Write code in python
  - re-use vast ecosystem, used in ML domain
- Reading columnar data from file is efficient
- Array libraries already optimized
  - make use of contiguous memory location

Expect efficient code, easy to read and write

Usable with non simplistic analysis ?

- Yes ! with proper frameworks
  - awkward : like numpy but allowing multi-dim arrays with variable length dimensions
    - num of el, jets varies from event to event…
  - uproot : read/write ROOT file to/from awkward arrays
  - coffea : wrap awkward arrays into physics oriented python object
- Demo analysis have been performed in CMS and ATLAS

Example with coffea in ATLAS

meta-array representing all events

select events with >=1 elec

in all remaining event (':'), leading electrons ('0')

```
>>> import awkward as ak
>>> events[ak.num(events.Electrons) >= 1].Electrons.pt[:, 0]
<Array [7.36e+03, 8.84e+04, ... 3.27e+04] type='20194 * float32'>
```

```
>>> electrons = Events.electrons
>>> jets = Events.jets
>>> electrons.delta_r(electrons.nearest(jets)) < 0.2
<Array [[True], [], [], ... True], [], [True]] type='50000 * var * ?bool'>
```

```
>>> events.Electrons.trackParticles.z0
<Array [[[-47]], [], ... ] type='50000 * var * var * float32'>
```

Automatic cross-reference of array of tracks from links associated to electrons

Analysis Demo in ATLAS

- Reproduce a simple event selection analysis
  - including **overlap removal** logic
  - With SUSYTools
  - With coffea (and uproot+awkward)
- Verify object counts are identical (jets, muons, electrons)
- Compare perfomances

Measurement	Total time [s]	average no. events / s
Athena/SUSYTools	22	2300
Columnar	3.8	13000
Columnar (cached)	1.2	42000

Other case comparison : jet calibration

Fill 1620 histograms, each in its (E,η) bin, from 12M events

- From a RootDataFrame compiled C++ code
  - 22 sec
- uproot+numpy array operations
  - 7 sec

Similar conclusions on more complex, although not directly comparable analysis

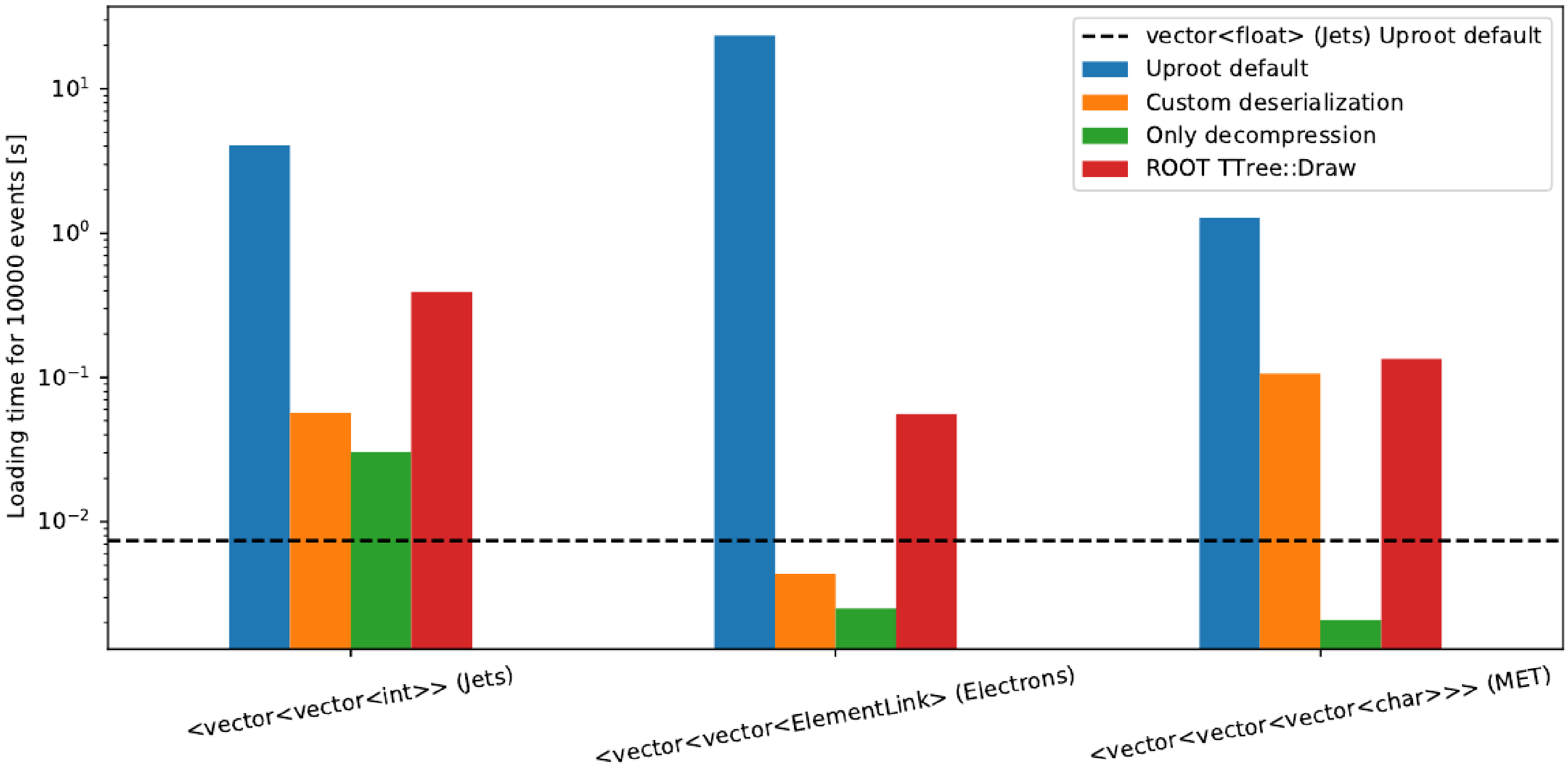
How far can we go with columnar analysis ?

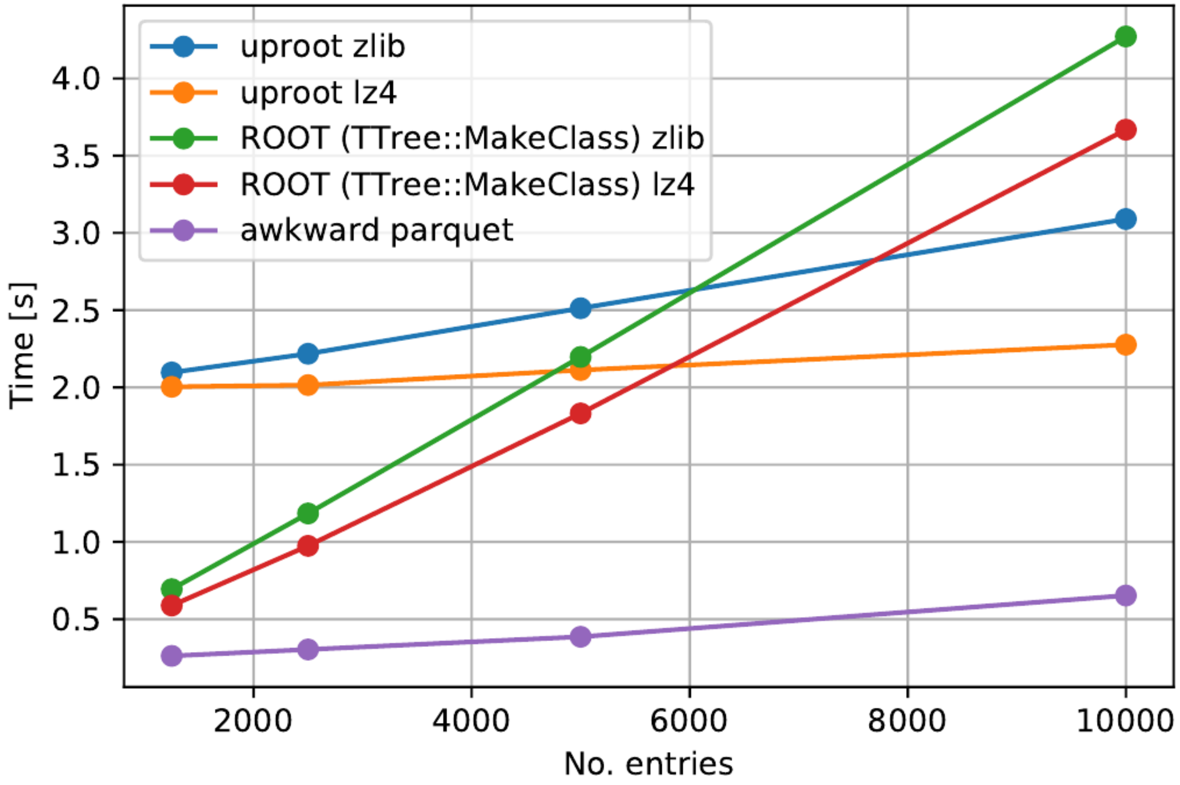
Not clear what the limits of this type of analysis are

- Event loop based analysis have no limitation on complexity of in-loop operations… Can columnar analysis deal with :
  - combinatorial calculations
  - MET re-evaluation
  - systematics (?)
- Many arrays x many events : does not fit in memory
  - analysis must be split in chunks
  - increase code complexity… can it be (partially) automated ?

Summary, discussion

- Columnar analysis is a very promising solution to implement simple analysis
  - python oriented, close to ML practice
  - simple yet very efficient code thanks to optimized libraries
- Frameworks under development
  - pure python : uproot, coffea
  - ROOT: development of future TTree : RNtuple
    - designed for columnar analysis, excellent performances
- Limitation of these types of analysis unclear
  - no support from ATLAS yet (a.f.a.i.k)
- How do people feel about this approach ?


vector<float> (Jets) Uproot default
Uproot default
Custom deserialization
Only decompression
ROOT TTree::Draw
Loading time for 10000 events [s]
10^{1}
10^{0}
10^{-1}
10^{-2}
<vector<vector<int>> (Jets)
<vector<vector<ElementLink> (Electrons)
<vector<vector<vector<char>>> (MET)



- "Flat" means branches of only fundamental types or arrays thereof (no `std::vector`)
- Comparison: EventLoop in ROOT vs. Columnar access in uproot/awkward.