$\in$

numba in XENON

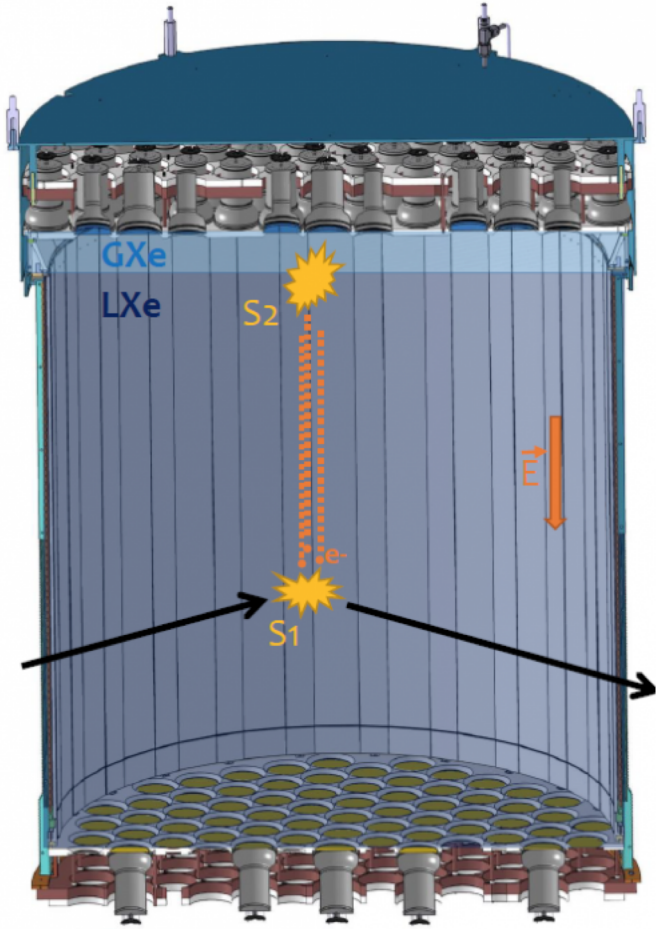
Christopher Tunnell (1) working with Jelle Aalbers (2)

1. Kavli Institute for Cosmological Physics, U. Chicago
2. Nikhef

DIANA-HEP, April 2018

What is XENON?

Liquid XENON dark matter detector instrumented with 248 photomultipliers and 10-ns flash ADCs. We make a world-leading new experiment every few years.



Scientific software design challenges

- Three key challenges, all with same solution

1. Changing requirements:
   - Experimental phases demand changing software:
     - detector understanding improves
     - analysis priorities change
   - Requires constant refactoring to fight ‘cancer growth’
2. Small core teams
   - *Every* experiment of *every* size complains about lack of skilled people
   - Requires refactoring effort to keep code base coherent and manageable with small team
3. Constant battle against ‘legacy ware’
   - ‘legacy’ code defined as code that cannot be touched without knowing if it breaks
   - Physics-ware suffers this due to publication pushes and high personnel turnover
   - Requires testing to allow sensible refactoring

- Solution: refactoring, which requires simplicity

Design decisions in XENON1T: Python

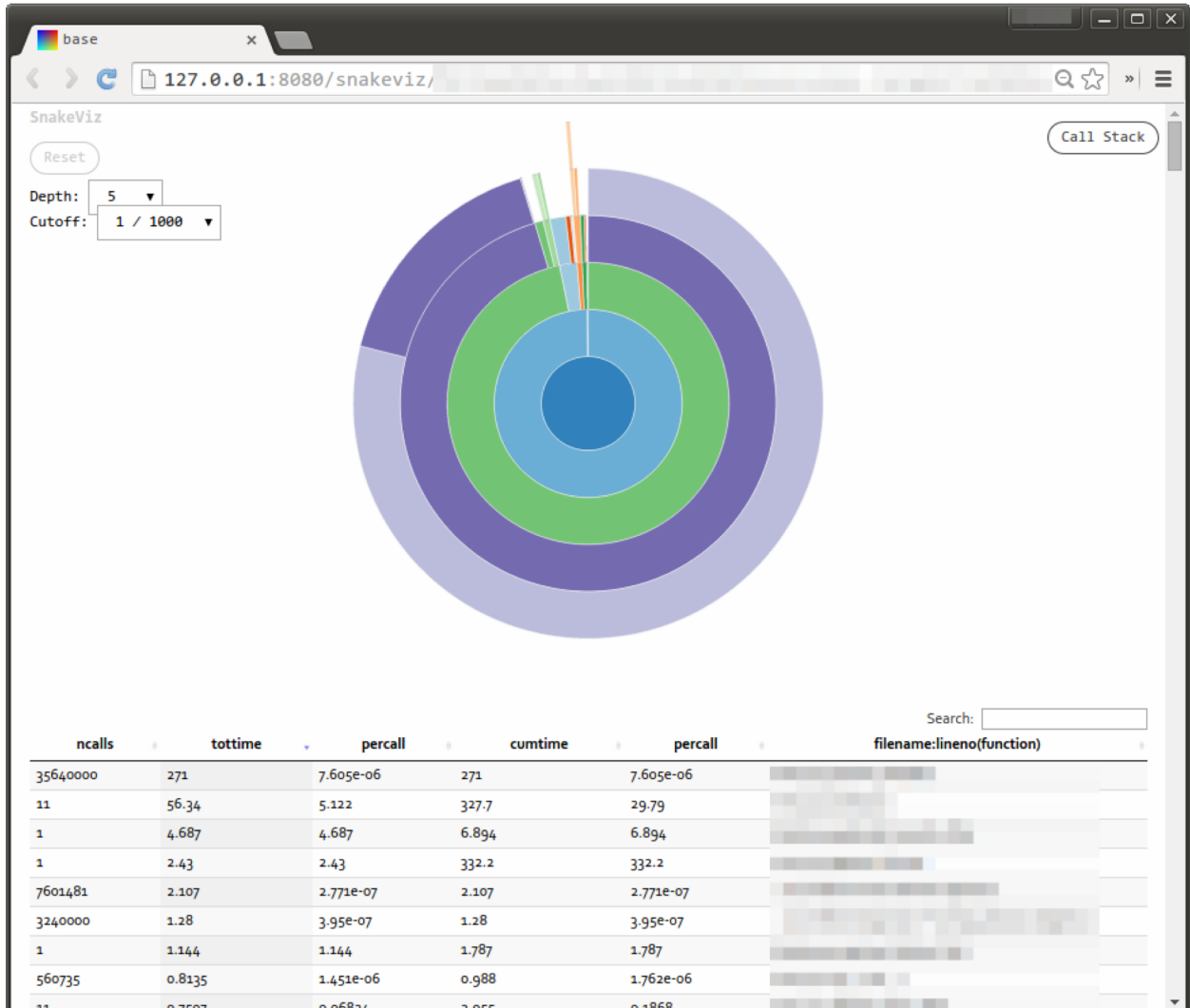
- Scientific python stack:
    - Mature
    - Large user base
    - Actively developed
    - Transferable skills
        - (Most of the software team thought they'd be in a cushy data science job by now)
    - More resistant to technology lock-in
- Pipeline programming based on around event class
    - Extending philosophy of SNOMAN and Stan's RAT (various neutrino and DM experiments)
- *Testing easier and sophisticated*
    - Helps fight legacy-ware and enables refactoring
- Speed
    - "Python is slow", right? Not with numba
    - Must keep throughput of 500 MB/s in DAQ and 1 PB/yr in offline processing
    - Physics software in C++ (almost always) slower* than Python
        - Profiling and refactoring more involved in C++
        - * caveat 1: if you can just-in-time compile
        - * caveat 2: 'in practice' due to the overhead of writing optimized C++ code by physicists

Why not just have C++ routines with glue Python?

- SWIG/boost/etc
  - Made various experimental frameworks this way but…
  - Multi-language packages are *nightmares* to:
    - Maintain: multiple style guides and doubles dependencies complexity
    - Test: two testing frameworks needed, C++ testing frameworks more involved and (in my opinion) not as sophisticated
    - Readability: hard to read and/or understand
    - Debug: exception handling through two languages tricky and using two debuggers is also not straightforward (‘why does my Python code segmentation fault?!’)
- Even depending on ROOT is complicated large dependency
  - attacks others, e.g. changing ‘import’ behavior
  - locks Python version, e.g. we are stuck at 3.4 until we find a new Anaconda-package maintainer
  - Development out of sync with other software ecosystems
- Why not just write Python code that turns itself into C code when slow?
  - i.e. *numba*

Three examples of our ‘method’ in practice

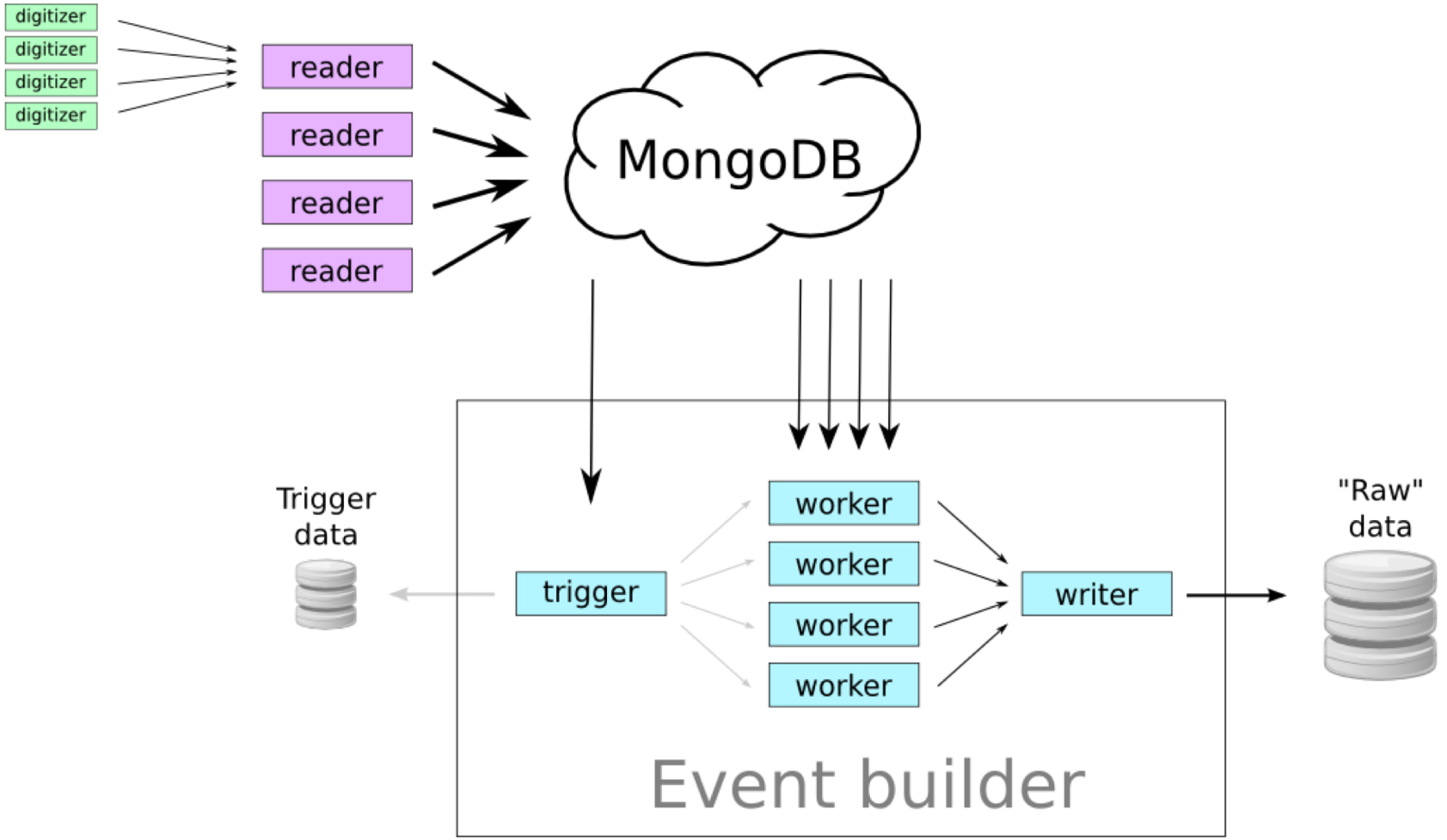
- Method
  - Quick prototype in Python
  - Profile with e.g. snakeviz
  - numba-ify and/or refactor
  - Rinse and repeat
- Examples of success
  1. XENON1T DAQ
  2. XENON1T event reconstruction
  3. XENONnT developments



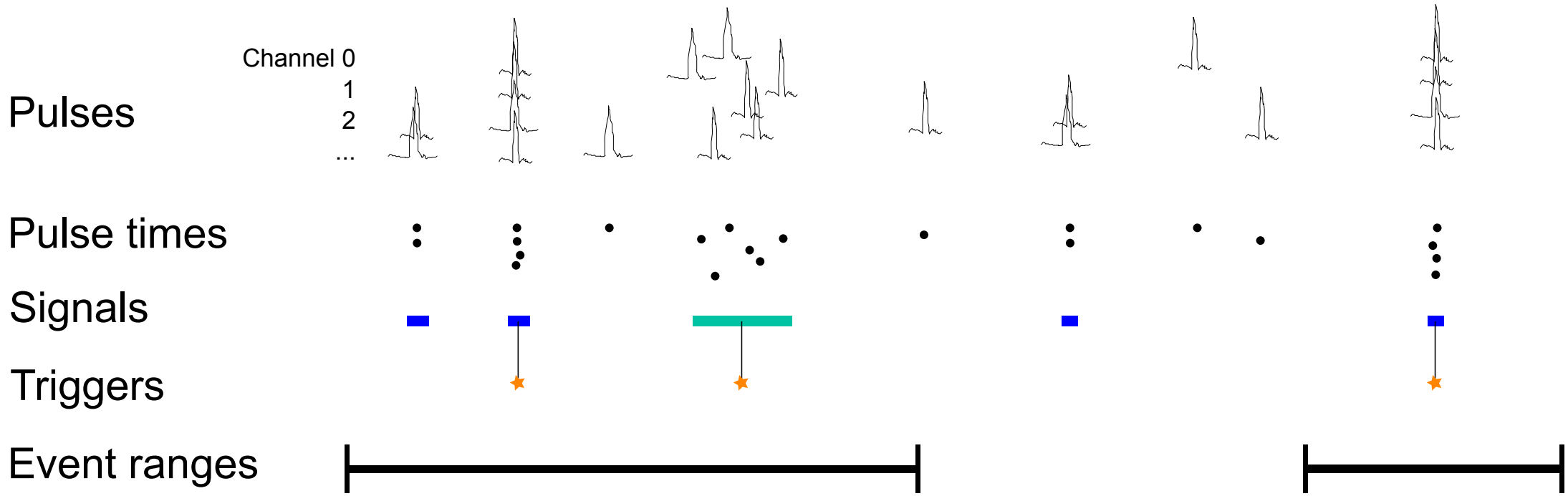
Use in XENON1T #1: DAQ

- Setup:
  - Asynchronous digitizer readout of payload “time, channel, data”
  - Data backend is MongoDB
  - 500 MB/s and 1 million inserts per second is ‘design’
  - Software trigger
- “DAQ in Python” “You mad?!”
- Software trigger and event builder
  - Cluster
  - Split
  - Sorting PMT signals
  - Decide if trigger

Use in XENON1T #1: DAQ: diagram



Use in XENON1T #1: DAQ: array operations



Use in XENON1T #1: DAQ, Example 1

```
@numba.jit(nopython=True)
def get_pmt_numbers(channels, modules, pmts_buffer, pmt_lookup):
    """Fills pmts_buffer with pmt numbers corresponding to channels, modules according to pmt_lookup matrix:
     - pmt_lookup: lookup matrix for pmt numbers. First index is digitizer module, second is digitizer channel.
    Modifies pmts_buffer in-place.
    """
    for i in range(len(channels)):
        pmts_buffer[i] = pmt_lookup[modules[i], channels[i]]
```

https://github.com/XENON1T/pax/blob/master/pax/trigger_plugins/SortData.py#L57

Use in XENON1T #1: DAQ, Example 2

```
@numba.jit(nopython=True)
def classify_signals(signals, s1_max_rms, s2_min_pulses):
    """Set the type field of signals to 0 (unknown), 1 (s1) or 2 (s2). Modifies signals in-place.
    """
    for signal_i, s in enumerate(signals):
        sigtype = 0
        if s.time_rms > s1_max_rms:
            if s.n_pulses >= s2_min_pulses:
                sigtype = 2
        else:
            sigtype = 1
        signals[signal_i].type = sigtype
```

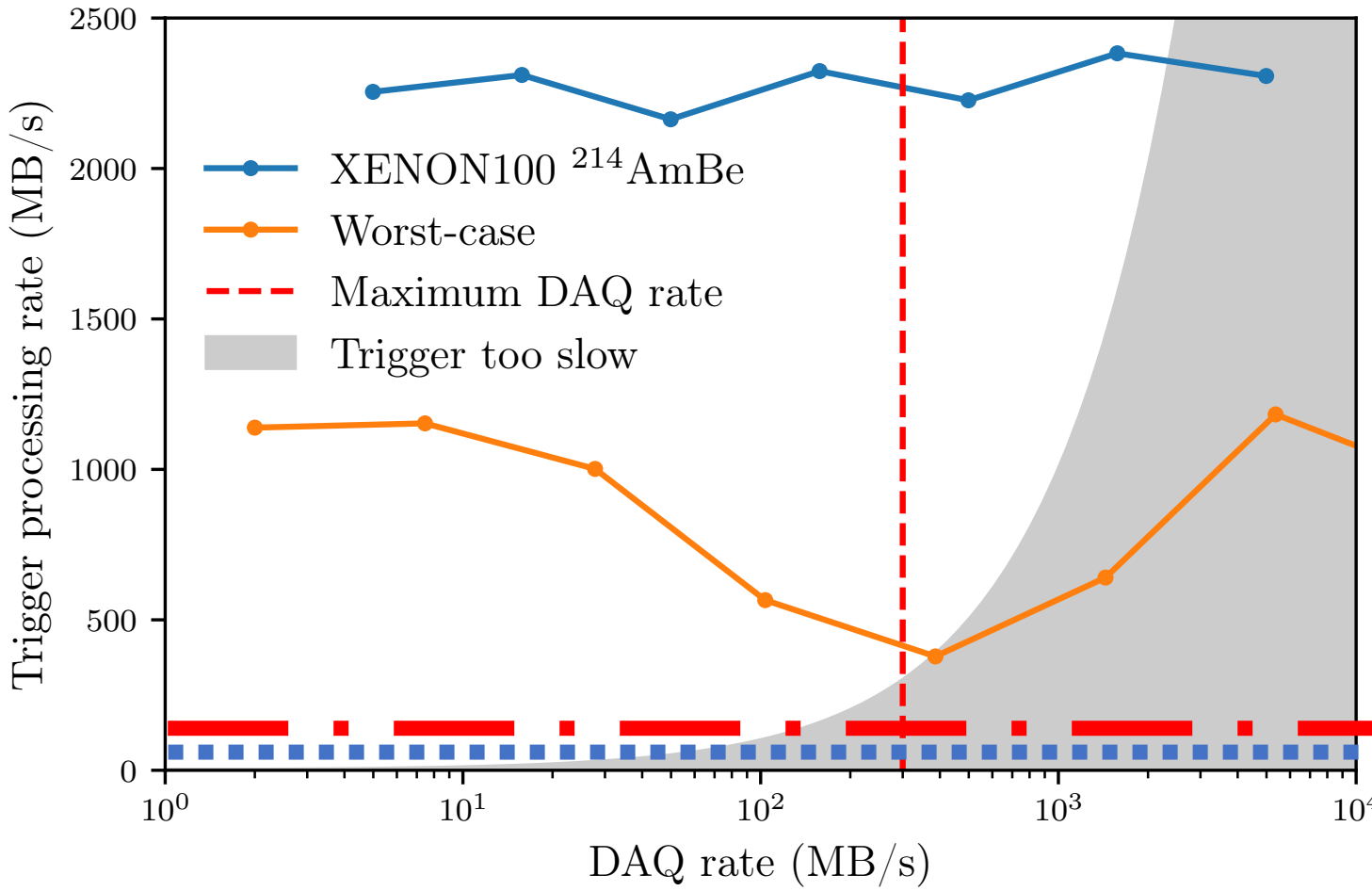
https://github.com/XENON1T/pax/blob/master/pax/trigger_plugins/ClassifySignals.py#L14

Use in XENON1T #1: DAQ, Example 3

```
@numba.jit(nopython=True)
def find_last_break(times, last_time, break_time):
    """Return the last index in times after which there is a gap >= break_time.
    If the last entry in times is further than signal_separation from last_time,
    that last index in times is returned.
    Returns -1 if no break exists anywhere in times.
    """
    imax = len(times) - 1
    # Start from the end of the list, iterate backwards
    for _i in range(len(times)):
        i = imax - _i
        t = times[i]
        if t < last_time - break_time:
            return i
        else:
            last_time = t
    return -1
```

https://github.com/XENON1T/pax/blob/master/pax/trigger_plugins/HandleEdgeThings.py#L73

Use in XENON1T #1: DAQ: speed

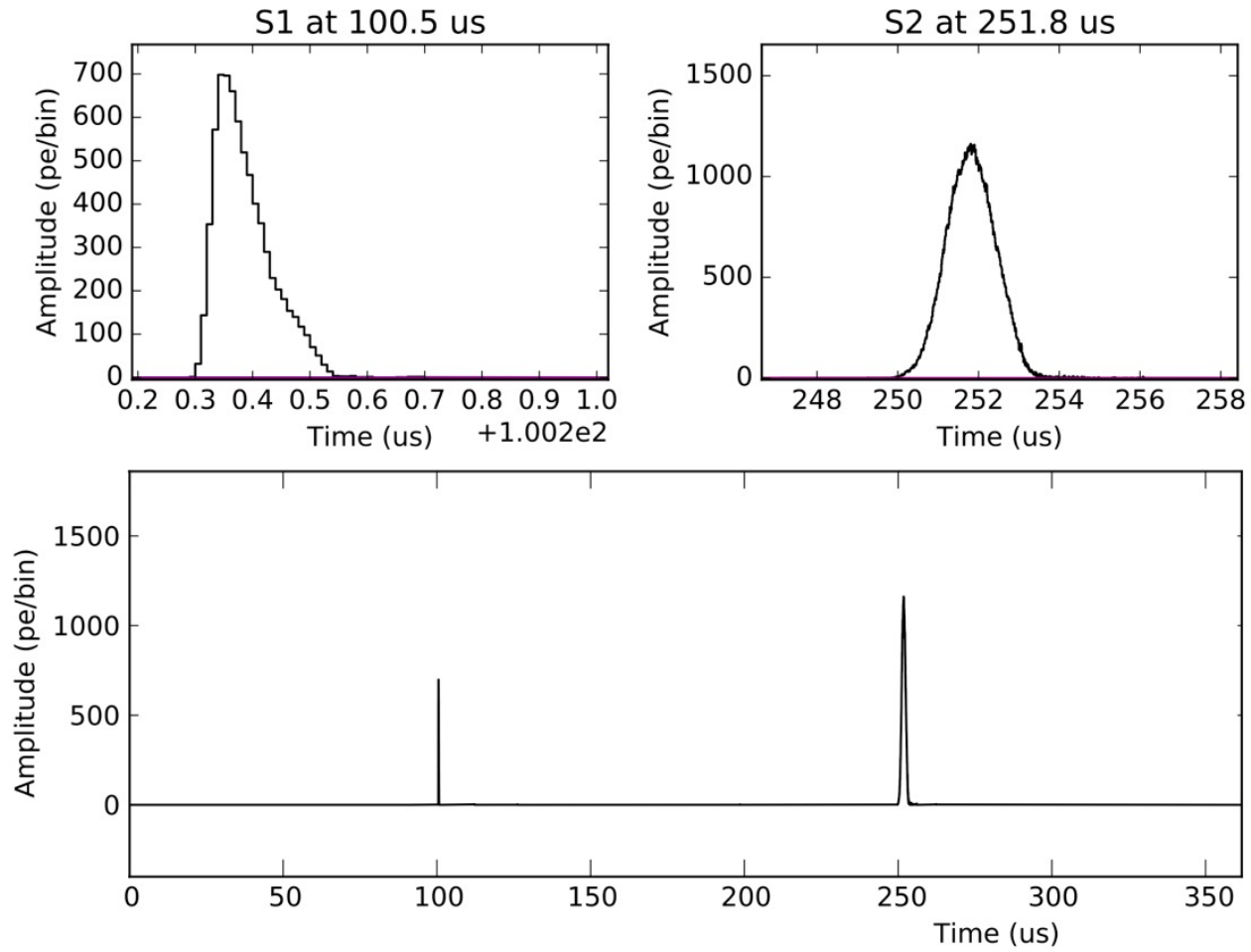


Previous iteration using C++ wrapping was 10-20 MB/s

Python-only was 0.1-1 MB/s

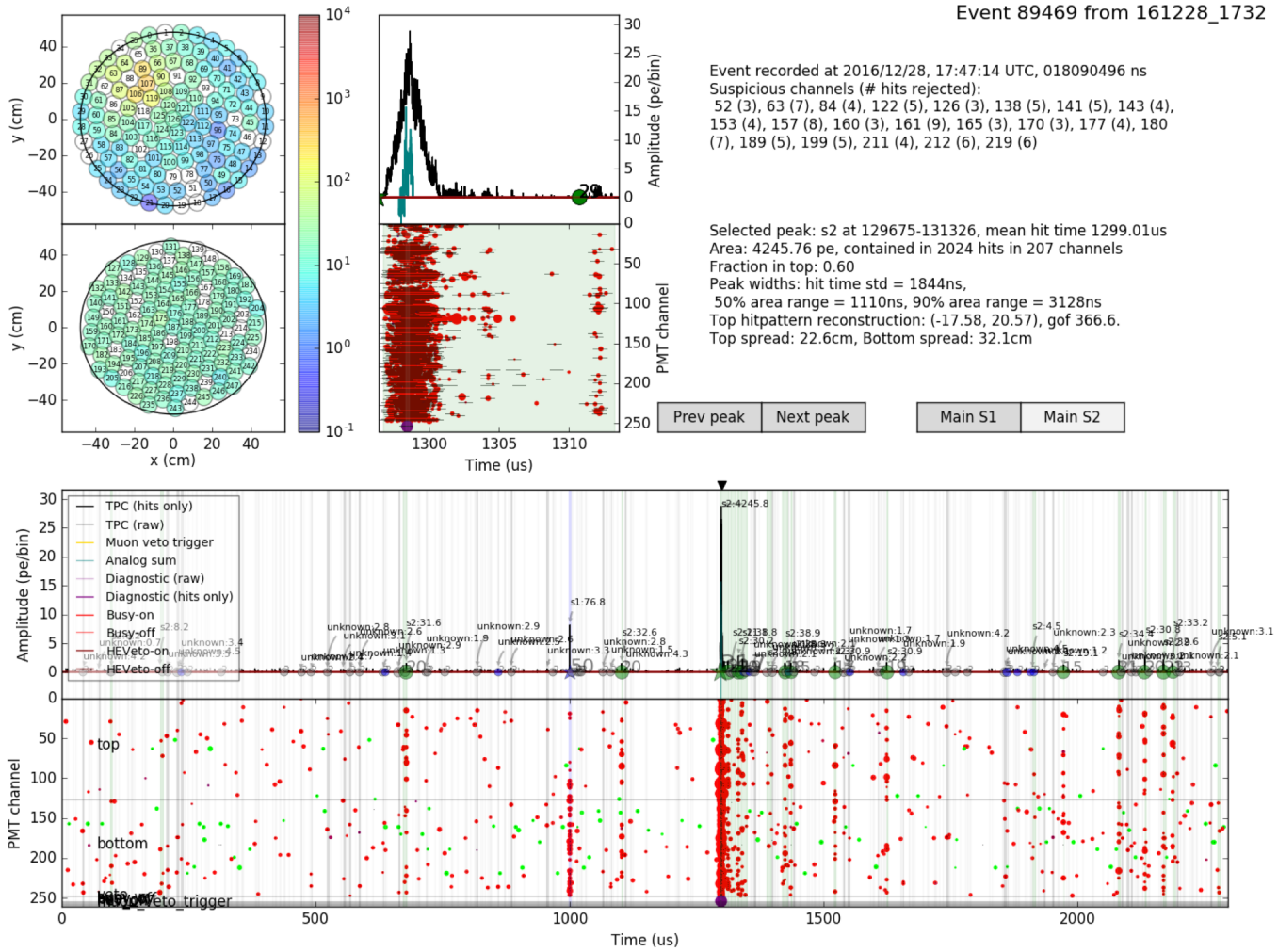
Use in XENON1T #2: event reconstruction

High-energy clean



Use in XENON1T #2: event reconstruction

Low energy, red dots PMT pulses



Pax's performance breakdown

Plugin	%	/event (ms)	#/s	Total (s)
ReadZipped	0.1	6.1		0.6
DecodeZPickle	1.0	53.5	18.7	5.3
SortPulses	0.2	8.2		0.8
PulseProperties	1.4	73.6	13.6	7.4
CheckBoundsAndCount	0.8	44.8	22.3	4.5
DesaturatePulses	2.9	156.0	6.4	15.6
FindHits	4.4	237.9	4.2	23.8
HitfinderDiagnosticPlots	0	0	n/a	0.0
SumWaveform	7.7	415.5	2.4	41.5
GapSizeClustering	26.3	1415.0	0.7	141.5
RejectNoiseHits	7.7	412.0	2.4	41.2
LocalMinimumClustering	0.5	29.3	34.1	2.9
NaturalBreaksClustering	42.0	2260.8	0.4	226.1
BasicProperties	0.3	16.6	60.2	1.7
SumWaveformProperties	1.9	103.7	9.6	10.4
CountCoincidentNoisePulses	0.1	3.3		0.3
PosRecWeightedSum	0.0	0.6		0.1
PosRecMaxPMT	0.0	0.4		0.0
PosRecRobustWeightedMean	0.1	2.7		0.3
PosRecNeuralNet	0.0	1.5		0.1
PosRecTopPatternFit	0.4	23.7	42.3	2.4
PosRecTopPatternFunctionFit	0.4	23.4	42.7	2.3
HitpatternSpread	0.2	12.1	83.0	1.2
S2SpatialCorrection	0.1	7.8		0.8
S2SaturationCorrection	0.0	1.1		0.1
AdHocClassification1T	0.0	0.3		0.0
BuildInteractions	0.0	0.1		0.0
BasicInteractionProperties	0.0	1.2		0.1
S1AreaFractionTopProbability	0.0	1.0		0.1
DeleteLowLevelInfo	1.3	67.5	14.8	6.8
DummyOutput	0.0	1.4		0.1
TOTAL	100.0	5380.7	0.2	538.1

Use in XENON1T #2: event reconstruction

- Setup:
  - Process 1 hour of data by reconstructing events
    - Finding hits in PMT pulses
    - Clustering hits into peaks
    - Computing peak properties including area, classification and position
- “signal processing in Python” “You mad?!”
- Challenges
  - In pipeline, between 30-50 steps that have to be optimized independently after profiling
    - Early part of chain more ‘array’ so numba-able, later depend on numpy/scipy/sklearn internal optimization
  - Needs to be understandable to analysts
  - Lot of PMT waveforms to integrate
- Hit finding and signal finding
  - Building PMT hits
  - Integrate waveforms
  - Clustering signals
  - Compute properties of signals (baselines)
  - Grouping signals

https://github.com/XENON1T/pax/blob/master/pax/config/XENON1T.ini#L5

Use in XENON1T #2: event reconstruction

```
@numba.jit(nopython=True)
def classify_signals(signals, s1_max_rms, s2_min_pulses):
    """Set the type field of signals to 0 (unknown), 1 (s1) or 2 (s2). Modifies signals in-place.
    """
    for signal_i, s in enumerate(signals):
        sigtype = 0
        if s.time_rms > s1_max_rms:
            if s.n_pulses >= s2_min_pulses:
                sigtype = 2
        else:
            sigtype = 1
        signals[signal_i].type = sigtype
```

Use in XENONnT: give it a rethink

- Redo software stack with array focus
  - Use XENON1T knowledge to rethink earlier design decisions
  - Abandon ‘event class’ internally and move toward arrays (or try OAmap)
    - numba-able, serialization easier, no object creation cost
    - This is artifact of ROOT C++ frameworks, where serializing event classes to ROOT files from Python was a constant sustainability challenge
      - Convert to ROOT at end from HDF5 if needed
    - Want to transparently switch between DB and files
- Initial tests promising, can’t show so much since in flux
  - 100 MB/s/core achievable in Python framework

Use in XENONnT: give it a rethink

Simple arrays >> lists of custom objects

Object creation has a penalty even in C

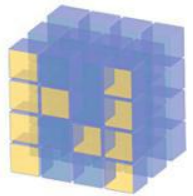
Example: make a histogram of tau-tau jets in CMS

0.018 MHz	full framework (CMSSW, single-threaded C++)
0.029 MHz	load all 95 jet branches in ROOT
2.8 MHz	load jet p_T branch (and no others) in ROOT
12 MHz	allocate C++ objects on heap, fill, delete
31 MHz	allocate C++ objects on stack, fill histogram
250 MHz	minimal "for" loop in memory (single-threaded C)

Pivarski, J. et. al. "Toward real-time data query systems in HEP" ACAT 2017 proceedings, arXiv:1711.01229

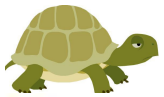
In python, object-level code is particularly slow

... but numpy and numba allow array ops at native performance



Use in XENONnT: give it a rethink

Throughput in uncompressed raw data / core



Pax: 0.3 MB

XENON1T eventbuilder: 3 MB

Strax: 100 MB

Conclusions

- numba worked for us at least
- Questions and discussion?