CPU trigger primitive creation

Philip Rodrigues

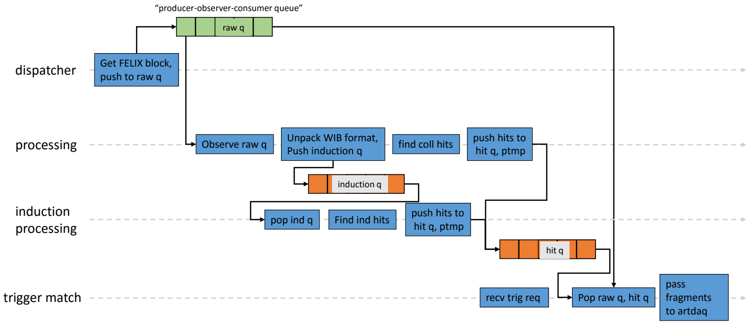
University of Oxford

July 29, 2020

Executive summary

- CPU-based hit finding is a relatively straightforward and flexible approach
- Hit finding builds on in-host buffering from FELIX, triggering from `artdaq`, and data selection with `ptmp`. The ProtoDUNE demonstrator is fully integrated with the rest of the DAQ
- The CPU hit-finding has been successfully used for the self-trigger demonstration, for purity monitoring, and for rapid neutron-source analysis
- Resource usage: 1 APA/server comfortably demonstrated (30 cores). Progress towards 2 APAs/server (with 2019 components)

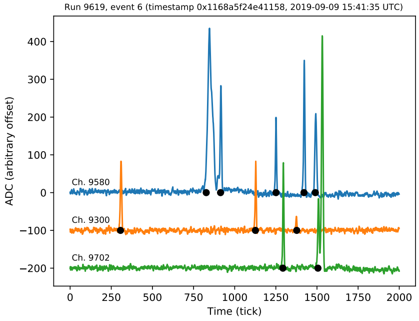
ProtoDUNE demonstrator structure

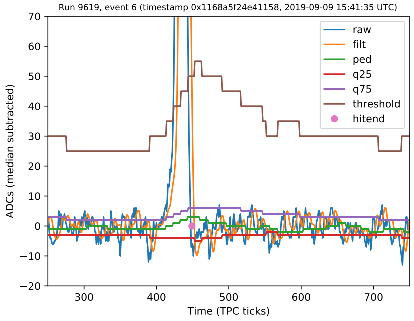


One of these per link. 10 links/APA

- Default FELIX buffering/triggering (see Adam's talk):
  1. Full data stream into host memory via PCIe
  2. Parse FELIX blocks (∼packets), separate links
  3. Copy data from each link to its own buffer
- Additional steps for hit-finding:
  1. Decode WIB format and expand 12-bit ADCs to 16-bit
  2. Push expanded collection/induction data onto separate queues
  3. Processing threads pop expanded data and find hits
  4. Hits buffered for readout

TP algorithm



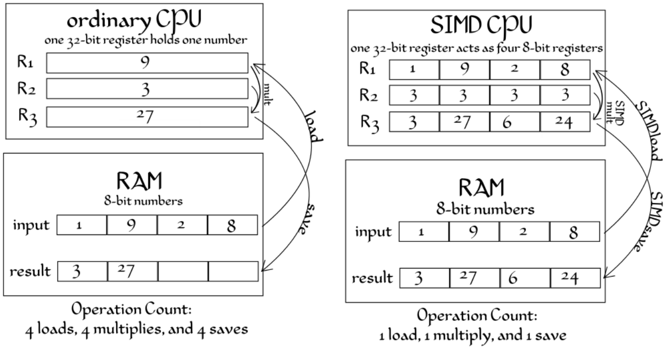


1. Dynamic per-channel pedestal finding and subtraction
2. Dynamic per-channel noise estimation
3. Finite impulse response filter (7 taps)
4. Sum charge above threshold (in number of σ)
5. Hits from noisy channels (according to offline list) filtered out

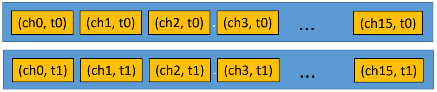
Techniques needed for performance

- Use AVX2 (SIMD) to act on 16 16-bit values with each instruction
- Reduce memory bandwidth by minimizing data copies
- CPU pinning and NUMA-awareness:
  - Fix CPU cores on which processing runs (reduce context switches ⇒ latency; increase throughput; could go further with `isolcpus`/`cgroups`)
  - Keep data and processing on same CPU socket (maximize memory bandwidth)

Single-instruction-multiple-data (SIMD)

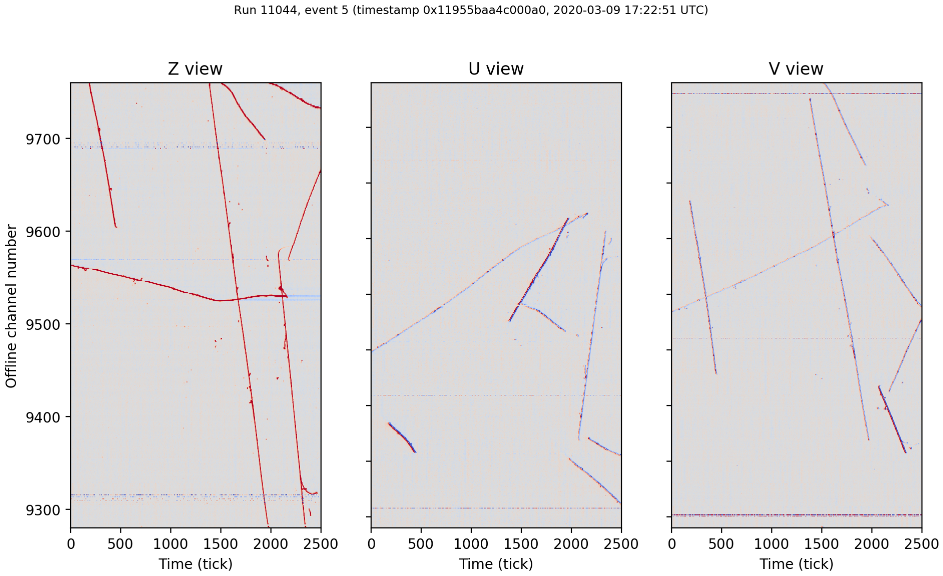


Credit: Decora at English Wikipedia. CC Attribution-Share Alike 3.0



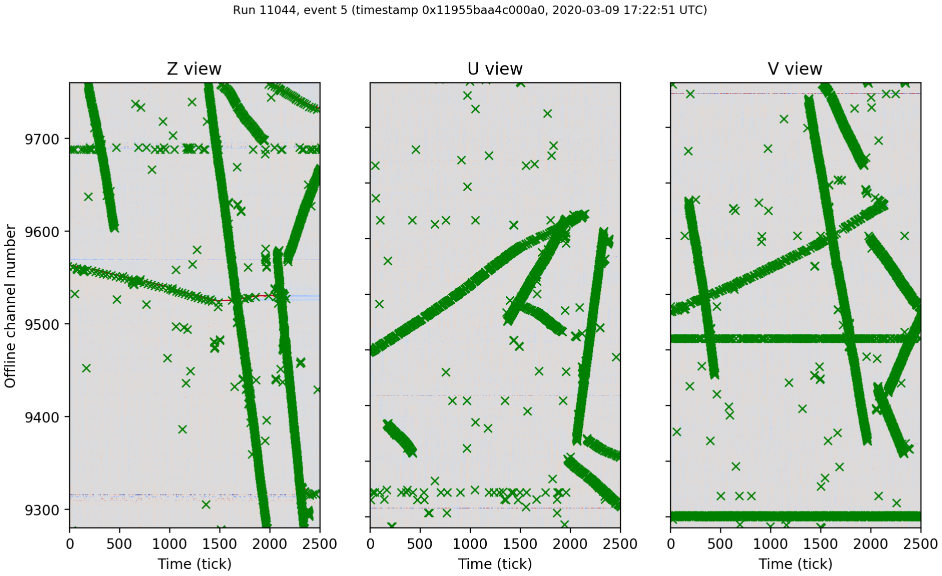
- Elementwise arithmetic, element rearrangement (“shuffles”), select based on mask (“blend”)
- Several generations of Intel SIMD instructions exist. I use AVX2, with 256-bit registers (introduced 2013)
- AVX-512 exists, but I don’t expect much improvement because clock rate reduces with AVX-512
  https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/
  https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/

Induction hit-finding example



- Full APA in one server. Hits found continuously, buffered and read out in response to trigger

Induction hit-finding example



- Full APA in one server. Hits found continuously, buffered and read out in response to trigger

Resource usage

```
  PID    VIRT     RES  %CPU   P   COMMAND
218656  29.9g   15.1g  59.2  17   processing
218697  29.9g   15.1g  58.3  22   processing-ind
218652  29.9g   15.1g  58.3  13   processing
218654  29.9g   15.1g  58.3  15   processing
218658  29.9g   15.1g  58.3  19   processing
218629  29.9g   15.1g  57.3  12   processing
218698  29.9g   15.1g  57.3  24   processing-ind
218699  29.9g   15.1g  57.3  28   processing-ind
218700  29.9g   15.1g  57.3  26   processing-ind
218701  29.9g   15.1g  57.3  30   processing-ind
218660  29.9g   15.1g  57.3  21   processing
218704  29.9g   15.1g  57.3  27   processing-ind
218705  29.9g   15.1g  57.3  29   processing-ind
218633  29.9g   15.1g  56.3  16   processing
218635  29.9g   15.1g  56.3  18   processing
218702  29.9g   15.1g  56.3  23   processing-ind
218703  29.9g   15.1g  56.3  25   processing-ind
218706  29.9g   15.1g  56.3  31   processing-ind
218637  29.9g   15.1g  55.3  20   processing
218631  29.9g   15.1g  54.4  14   processing
218690  29.9g   15.1g  42.7  32   boardreader
218685  29.9g   15.1g  40.8  35   flx-disp-0
218686  29.9g   15.1g  40.8  37   flx-disp-1
218687  29.9g   15.1g  40.8  39   flx-disp-2
218689  29.9g   15.1g  40.8  43   flx-disp-4
218677  29.9g   15.1g  39.8  42   flx-disp-4
218688  29.9g   15.1g  39.8  41   flx-disp-3
218673  29.9g   15.1g  38.8  34   flx-disp-0
218674  29.9g   15.1g  38.8  36   flx-disp-1
218675  29.9g   15.1g  38.8  38   flx-disp-2
218676  29.9g   15.1g  38.8  40   flx-disp-3
218678  29.9g   15.1g  30.1   0   boardreader
```

CPU threads according to `top`, run 10884, `np04-srv-029`, Intel Xeon Gold 6242 CPU @ 2.80GHz.
`processing`: frame decoding and collection hit finding
`processing-ind`: induction hit finding
`flx-disp-*`: FELIX packet decoding

- FELIX block parsing, decoding WIB format, hit finding on 1 APA of collection and induction channels on 32 CPU cores
- No cores running flat out

Towards 2 APAs per server

```
   PID   VIRT      RES   %CPU    P  COMMAND
259901  40.0g    27.7g   92.1    4  flx-disp-1
259903  40.0g    27.7g   92.1    8  flx-disp-3
259904  40.0g    27.7g   92.1   10  flx-disp-4
259889  40.0g    27.7g   92.1   36  flx-disp-1
259890  40.0g    27.7g   92.1   38  flx-disp-2
259900  40.0g    27.7g   91.6    2  flx-disp-0
259902  40.0g    27.7g   91.6    6  flx-disp-2
259888  40.0g    27.7g   91.6   34  flx-disp-0
259891  40.0g    27.7g   91.6   40  flx-disp-3
259892  40.0g    27.7g   91.6   42  flx-disp-4
259913  40.0g    27.7g   82.8   54  processing-ind
259912  40.0g    27.7g   80.8   44  processing-ind
259918  40.0g    27.7g   80.8   50  processing-ind
259917  40.0g    27.7g   79.8   48  processing-ind
259921  40.0g    27.7g   79.8   52  processing-ind
259915  40.0g    27.7g   79.3   46  processing-ind
259919  40.0g    27.7g   77.3   60  processing-ind
259916  40.0g    27.7g   76.8   58  processing-ind
259920  40.0g    27.7g   76.8   62  processing-ind
259914  40.0g    27.7g   75.9   56  processing-ind
259846  40.0g    27.7g   61.1   22  processing
259850  40.0g    27.7g   59.1   12  processing
259860  40.0g    27.7g   58.6   18  processing
259875  40.0g    27.7g   58.1   20  processing
259856  40.0g    27.7g   57.6   16  processing
259854  40.0g    27.7g   57.1   14  processing
259857  40.0g    27.7g   56.2   28  processing
259848  40.0g    27.7g   55.7   24  processing
259852  40.0g    27.7g   55.7   26  processing
259861  40.0g    27.7g   54.7   30  processing
259905  40.0g    27.7g   32.0   32  flx-reader-0
259893  40.0g    27.7g   31.5    0  flx-reader-0
```

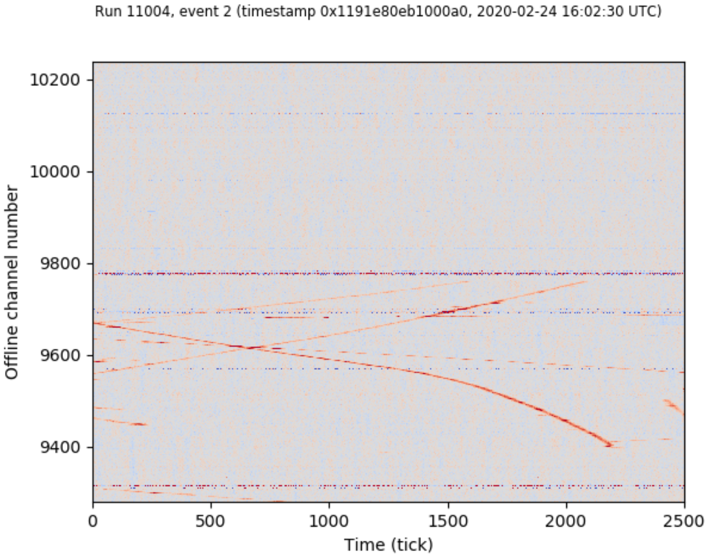
Run 10964, APA 6 only, `P` is core # of thread. All even, ie CPU 0

- In this test run, all parsing/decoding/hit finding is running on one of the two CPU sockets in the machine
  - (CPU % numbers change wrt previous slide because now both sibling HyperThread cores are being used)
- Further steps towards 2 APA/server demo:
  - Show that 2× more memory bandwidth available
  - Move `artdaq` threads onto same socket
  - Keep up when triggers are enabled

Improvements needed for final system

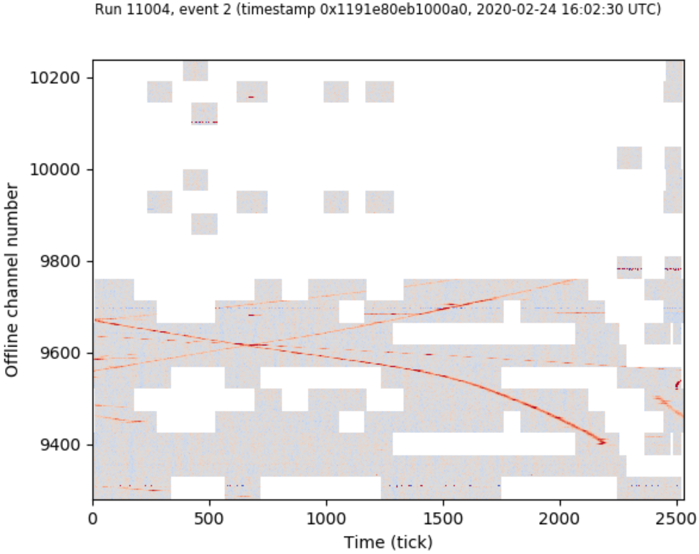
- Two APAs in one host
- Need to be able to take raw data triggers that aren't time-ordered (can already do this for hits)
- Error handling needs to be added:
  - Deal with errors in data
  - Deal with problems in hit finding (eg not keeping up, buffers fill)

Adaptability: proof-of-concept ROI readout



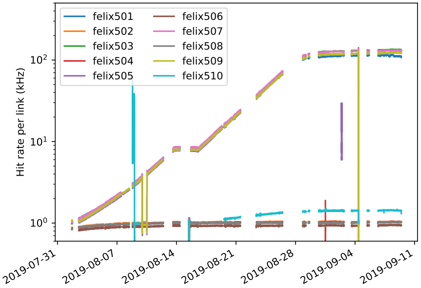
- ROIs for collection channels only
- Rate is low because detector running below nominal HV. Top half of channels are wall-facing

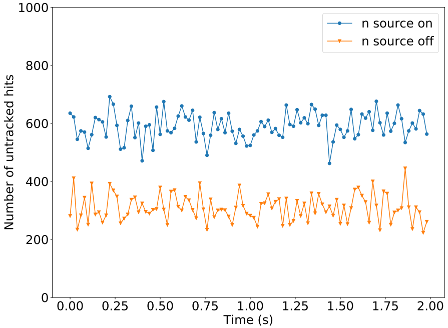
Adaptability: proof-of-concept ROI readout



- Each link holds two sets of contiguous channels, so 10 links ⇒ 20 rows in the plot

Hit finding successes





- Self-trigger
- Realtime purity monitoring
- Pulsed neutron source rapid analysis

Benefits and risks

- Benefits:
  - Development/testing/deployment relatively straightforward (1 FTE·yr for initial hit-finding PDSP demonstration, 1 more for refinements. $\mathcal{O}(1500)$ LOC for hit finding)
  - Highly flexible: easy to change algorithm
  - Gains from improvements in COTS components "for free"
- Risks:
  - Memory bandwidth for two APAs/server not yet demonstrated
  - Could imagine required algorithm changes that would be difficult (non-channel-parallel tasks like coherent noise removal; larger required FIR filter)
  - Might not meet electrical power consumption requirements

Marginal costs *vs* other options

- Suppose we picked another option as *baseline*, but wanted to keep development of CPU hit-finding in parallel, eg as potential risk mitigation. What would then be the marginal cost of retaining CPU hit finding capability?

Option	Hitfinding	10s buffer	SNB store
A	Host	Host	Host
B	FPGA	Host	Host
C	FPGA	FPGA	FPGA

- To add CPU-based hit finding to option B: Cost is more powerful CPUs and development time
- To add CPU-based hit finding to option C: Cost is more powerful CPUs, development time, and marginal cost of servers with sufficient memory bandwidth. FPGA also needs to support full PCIe data transfer

Criteria

Features

Which features have existing implementations and/or demonstrators?

- All demonstrated at ProtoDUNE-SP in FELIX board reader:
  - Receive data from WIBs
  - Noise filtering (extremely basic)
  - Hit finding: collection and induction
  - Forming trigger primitives and sending to data selection
  - Receive data request
  - Extract data from buffers
  - Send extracted data: raw data and hits
  - Decode extracted data offline

What features are missing and how much further development is required?

- No error handling
- Code could generally be cleaner

How does the solution interface to other components of the Upstream DAQ, and wider DAQ?

- In PDSP demonstrator, interfaced with FELIX, `ptmp` data selection, `artdaq` metrics

Adaptability

Were any components that rely on this technology integrated in existing DAQ systems? (e.g.: within ProtoDUNE-SP DAQ)

- All of it

How would the solution adapt to potential new requirements of the DUNE DAQ? (for example, different TP algorithms, RoI-based readout)

- Very adaptable: just write the algorithm in C++, recompile and go. Eg, I made a very basic ROI-based readout proof-of-concept (on top of the existing hit finding) in a few weeks.

Is it possible to tune and align the solution to support new ideas (e.g.: additional interfaces) or are there intrinsic structures that constrain potential extensions/modifications?

- Not really sure how to answer this. It's C++ code, so you can do anything code can do. It's already interfaced with artdaq, ptmp, the artdaq metrics system, etc. Was able to move it from the FELIX BR "publisher" config to the "on-host" config fairly straightforwardly. The SIMD approach means any algorithm that isn't embarrassingly parallel between channels would be more difficult

Reliability

Can you ensure data integrity using this technology?

- Yes. We're reading all the ADC data already, and reading timestamp and link info from the header data. Reading more header data will have some cost, but not a huge amount.

How do you handle and mitigate the known failure scenarios (above) and other errors in order to avoid their propagation to other components?

- FE failure of link. Mustn't affect other links.
  - Links are independent threads, so this is fine. Had this happen plenty of times on ProtoDUNE with no ill effects
- Link alignment (non-consecutive timestamps)
  - Didn't do this check, but could have (I read the timestamps of every superchunk for latency-measurement purposes)
- Handle uncompressed data
  - Ran everything uncompressed with on-host config, because quickassist zip wasn't working there.

Long term support and maintenance

What is the balance between in-house development and COTS components?

- Hardware all FELIX or COTS servers. Software all in-house

Does the solution require specific hardware products? (E.g.: only works with a specific SSD variant, or with any kind? Requires specific FPGA/CPU models or features? Are there implications for spares?)

- As written, uses AVX2, available on all recent Intel/AMD processors. Should rewrite using a library for easier adaptation to new SIMD instruction sets.

Can this solution profit from research and development outside DUNE (eg. by manufacturers, other experiments, etc.)?

- Maybe? Could have more specialized parsing of the DMA "packets" from FELIX. Other experiments' experiences with SIMD/profiling etc, eg
  `https://indico.ph.ed.ac.uk/event/66/contributions/824/attachments/682/830/EdinburghOptimisation17Feb2020.pdf`

How strong and connected is the community of users and partners around the technology?

- Hit finding builds on FELIX technology, so links to FELIX user base. Takes advantage of in-host data buffering work from CERN collaborators, artdaq (FNAL) for triggering. Hit-finding itself was mostly just me, but straightforward to expand community

Do we have sufficient engineers and developers with the necessary expertise to support this solution?

- Requirements are pretty low. I developed hit-finding code mostly on my own, having no real prior experience in this sort of low-level optimized code writing. Got a lot of support from Roland for system-level tuning.

Resource requirements

What are the resources (FPGA, cpu-cycles, memory) required by the solution as it stands now, per APA?

- One Intel Xeon Gold 6242 CPU (32 HT cores, 16 physical cores) can handle a full APA (just): decoding frames, hit finding on collection+induction channels

What are the future prospects for reducing resource use?

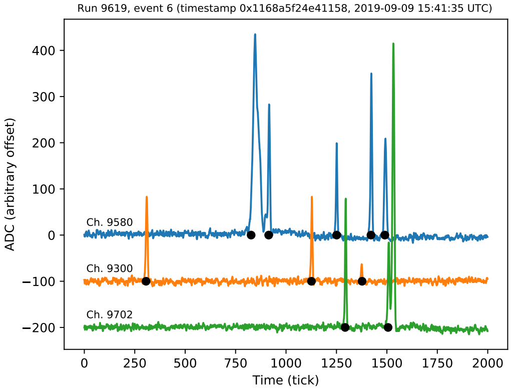
- Algorithm alterations: sum 2 or 4 ticks before processing; estimate pedestals less frequently; no filtering?
- Format alterations: More convenient format for WIB frames to reduce the processing needed to decode (already planned: docdb 14947). Make WIB frame format fit better with fixed-size FELIX blocks?
- Wait for technology improvements (2024 CPUs vs 2019 CPUs)

Conclusions

- CPU-based hit finding is a relatively straightforward and flexible approach
- Hit finding builds on in-host buffering from FELIX, triggering from `artdaq`, and data selection with `ptmp`. The ProtoDUNE demonstrator is fully integrated with the rest of the DAQ
- The CPU hit-finding has been successfully used for the self-trigger demonstration, for purity monitoring, and for rapid neutron-source analysis
- Resource usage: 1 APA/server comfortably demonstrated (30 cores). Progress towards 2 APAs/server (with 2019 components)

Backup slides

Trigger primitive (hit) finding

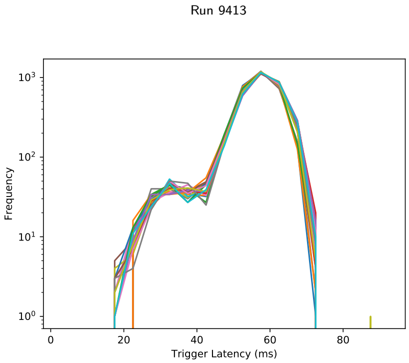


- Simple hit finding running in CPUs on FELIX BR hosts

1. Decode WIB format, select collection channels
2. Find pedestal and pedestal variance
3. Apply finite impulse response noise filter
4. Sum charge above threshold

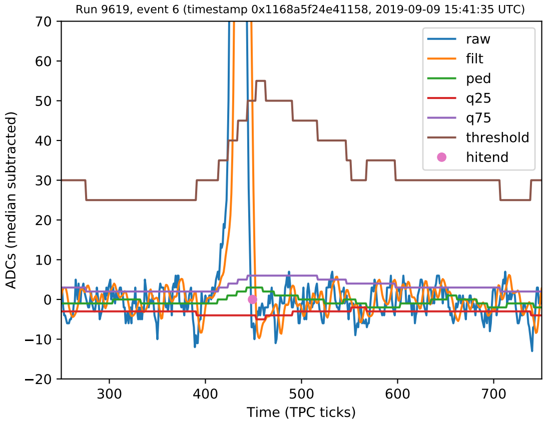
- Uses about 60% of a CPU core per link (10/APA)

Characterization: latency and CPU usage



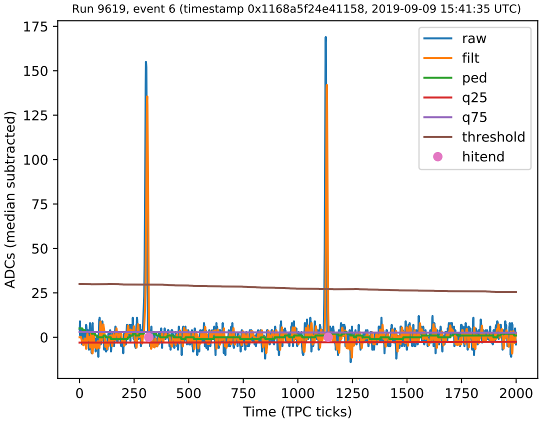
- Trigger latency, defined as time between data with timestamp T arriving in FELIX BR, and trigger request for data with timestamp T arriving in FELIX BR. One histogram per link
- $\sim$ 3000 triggers in this run, no latencies close to buffer depth ($\sim$ 1s)
- CPU usage: TODO, but approx 2–3 cores per APA for work downstream of hit-finding
-

Hit-finding details



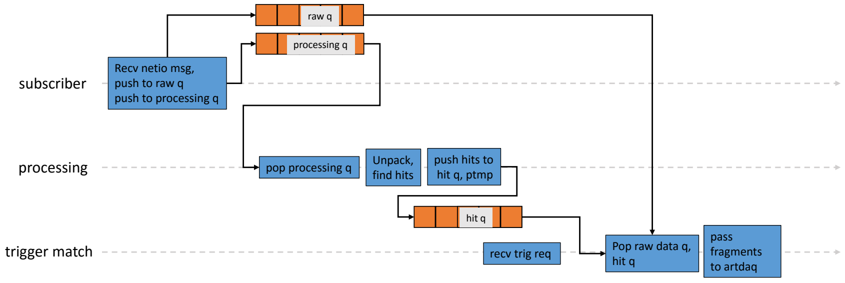
- Pedestal estimated using modified version of “frugal streaming” algorithm (arXiv:1407.1121). Estimate 25th and 75th %ile similarly to get interquartile range (IQR): threshold is 5 times IQR
- Noise filtering is via a 7-tap finite impulse response filter. It doesn’t really do a lot here. Might be filter coeff choice/integer approx, but probably just too few taps

Hit-finding thoughts/possible improvements



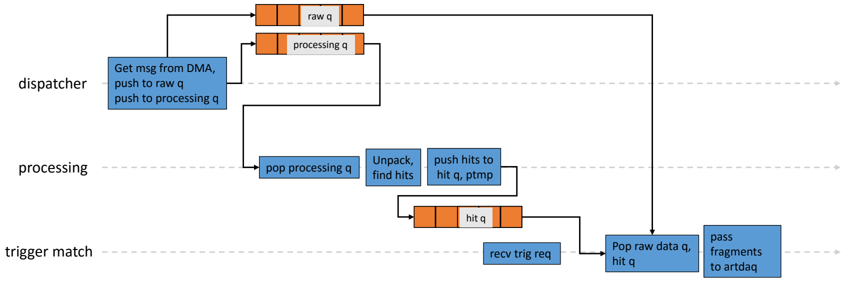
- I implemented the current pedestal RMS scheme to show it can be done efficiently, and because it's cool. Is it necessary?
    - Setting hit threshold as "$N\sigma$" is ~fixed-rate, but setting hit threshold as N ADC is ~fixed-efficiency. Not so obvious to me which is best
    - Pedestal and RMS don't really change on the single-tick timescale, so maybe this scheme isn't worth the processing
- 5 "sigma" threshold is way too high for serious physics
- IQR calculated in raw ADC but applied in filtered ADC. Should be consistent

Hit-finding organization, old setup



- One copy of this in each BR
- Trigger matcher and hit finder each get their own copy of the raw data
- Avoids sharing-related bugs, but requires data copy (extra 10 GB/s memory throughput, CPU cycles)

Hit-finding organization: on-host config



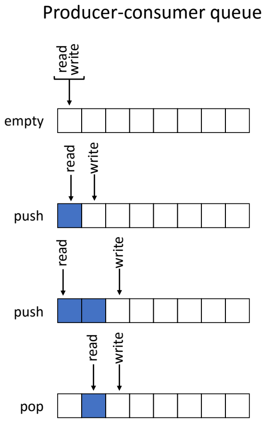
- The logic is not much different, but code got rearranged a bit
- One copy of this diagram per link (5 links/BR, 2 BRs/APA)

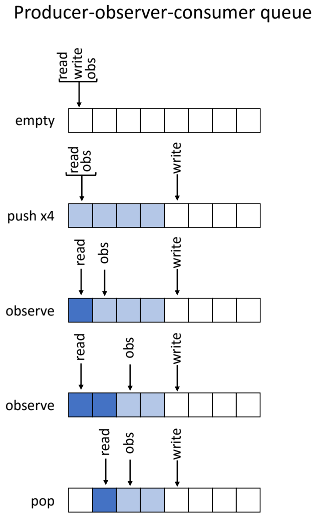
Digression: lockless single-producer single-consumer queues

- Single-producer, single-consumer queues allow exactly one *writer* thread to push items to the queue, and exactly one *reader* thread to pop items off the queue
- The FELIX BR uses `folly::ProducerConsumerQueue`. Interface is: `bool read(T&)`, `bool write(T)`, `T* frontPtr()`, `void popFront()`
- Optimization: reduce copies by adding `T* nextWritePtr()`, `void advanceWritePtr()` interface: can write FELIX data directly into queue

  This is just an accident of the way FELIX works, I think: the data payload we're interested in may be split across multiple PCIe "blocks", so the dispatcher thread has to parse the block info to find the data payload. Then there's a `netio::message::serialize_to_usr_buffer(char* buf)` function that puts the payload into `buf`. With `nextWritePtr()`, we can serialize the message straight into the SPSC queue without having to first serialize it to a temporary buffer, then copy it to the queue.
- Still need two queues (=two copies of data) for trigger matcher and hit finder. Would like to reduce this.

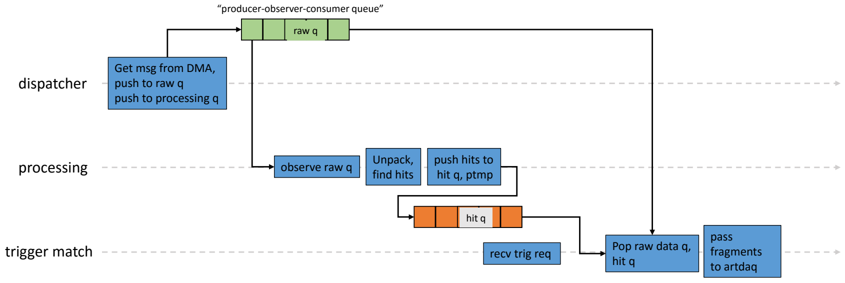
“Producer-Observer-Consumer” queue





Adds an “observer” thread (in this case, the hit finder). Read pointer can’t pass observer pointer, just like read can’t pass write in the SPSC queue. So every item gets pushed, then observed, then read, all without copies. There’s probably an alternative way of doing this with two queues and pointers, but this is the way I did it.

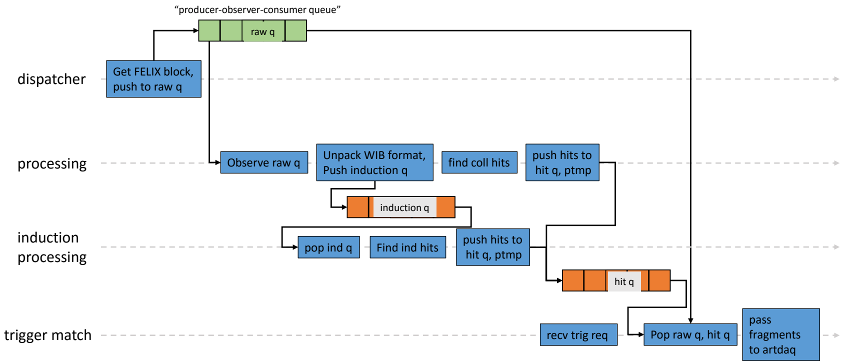
Using P-O-C queue



- One less copy; logic still substantially the same
- This was a noticeable improvement in performance (ie, lower CPU%[1]), but I can't find the number in my notes

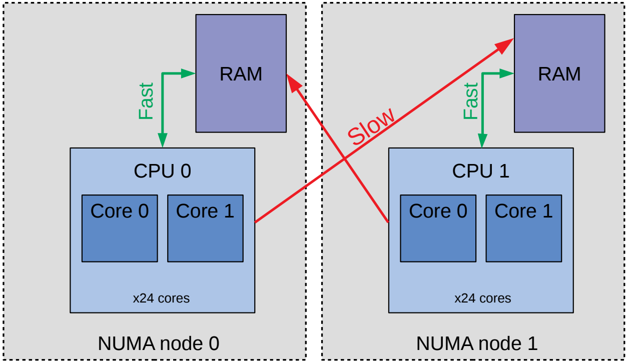
[1] Is CPU% really the most useful metric? Things we care about are the binary keeps up/doesn't keep up, and electrical power usage. I don't know the relationship between CPU% and power usage, though I expect it's complicated. Lower CPU% at least means that we *could* do more work and still keep up. See http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html for another critique of CPU% as a metric

Adding induction hit-finding



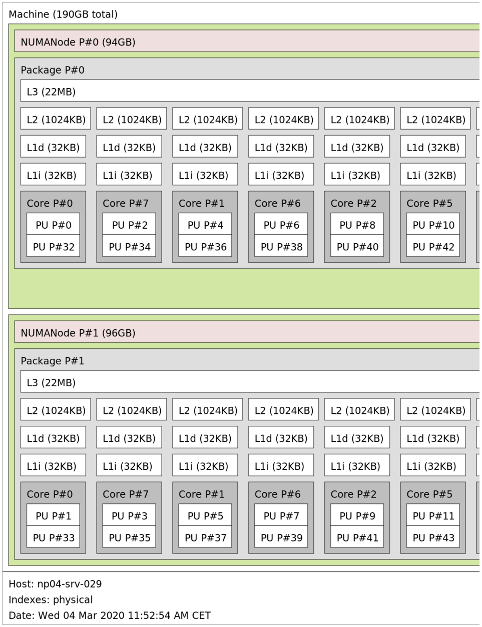
- Just run the collection-wire algorithm on the induction wires. (See http://indico.fnal.gov/event/22542/contribution/0/material/slides/0.pdf for the limitations of this approach)
- Note: WIB expansion and collection hit-finding on same thread. Induction wire processing gets its own thread. Ratio of induction:collection wires is 5:3
- How can we do better?

Digression: Non-uniform memory architecture (NUMA)



- RAM is "closer" to one CPU than the other. CPUs have faster access to the "closer" RAM, slower access to the "further" RAM
- Items within the same NUMA node are closest
- Upshot: arrangement of data in memory and cores used for processing is important

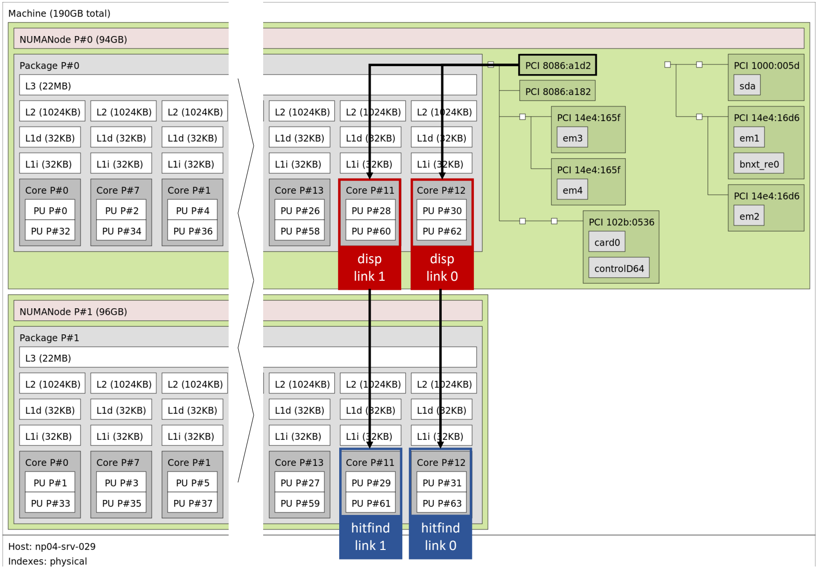
Digression: NUMA and CPU pinning



Host: np04-srv-029
Indexes: physical
Date: Wed 04 Mar 2020 11:52:54 AM CET

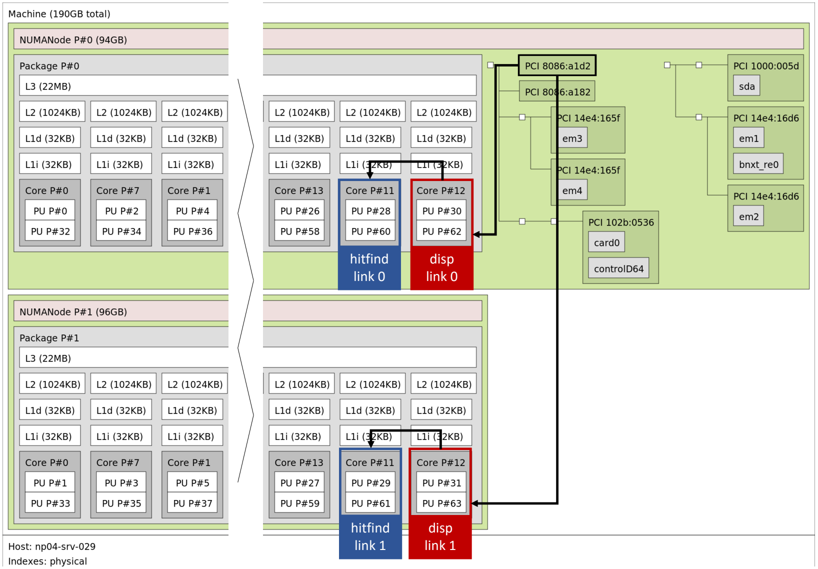
- Pin threads to a given core and ensure that memory is allocated on the same NUMA node. Maximizes throughput
- FELIX card is connected to NUMA 0

CPU pinning, old method



- All data crosses NUMA interconnect

CPU pinning, new method



- Only half of data crosses NUMA interconnect

Old vs new

Run 10876, old config

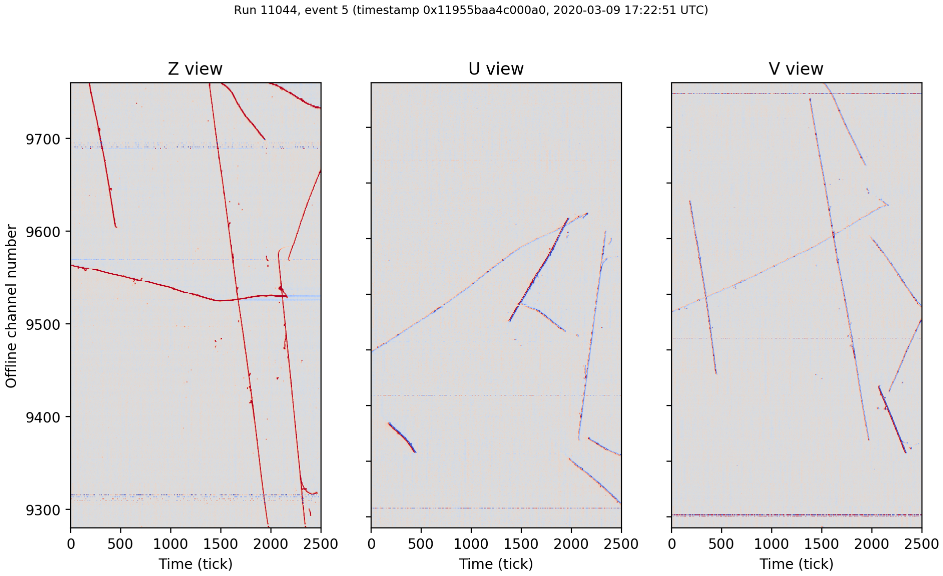
```
   PID    VIRT     RES  %CPU   P   COMMAND
104396   29.9g   15.1g  82.5  30   processing-ind
104397   29.9g   15.1g  81.6  22   processing-ind
104398   29.9g   15.1g  81.6  24   processing-ind
104399   29.9g   15.1g  81.6  26   processing-ind
104400   29.9g   15.1g  80.6  28   processing-ind
104327   29.9g   15.1g  73.8  13   processing
104329   29.9g   15.1g  73.8  15   processing
104333   29.9g   15.1g  72.8  19   processing
104337   29.9g   15.1g  72.8  21   processing
104331   29.9g   15.1g  71.8  17   processing
104342   29.9g   15.1g  60.2   1   processing
104352   29.9g   15.1g  60.2   3   processing
104354   29.9g   15.1g  60.2   5   processing
104356   29.9g   15.1g  60.2   7   processing
104358   29.9g   15.1g  60.2   9   processing
104405   29.9g   15.1g  58.3  23   processing-ind
104385   29.9g   15.1g  57.3  36   flx-disp-1
104387   29.9g   15.1g  57.3  40   flx-disp-3
104402   29.9g   15.1g  57.3  25   processing-ind
104372   29.9g   15.1g  57.3   4   flx-disp-1
104373   29.9g   15.1g  57.3   6   flx-disp-2
104384   29.9g   15.1g  56.3  34   flx-disp-0
104386   29.9g   15.1g  56.3  38   flx-disp-2
104388   29.9g   15.1g  56.3  42   flx-disp-4
104401   29.9g   15.1g  56.3  31   processing-ind
104403   29.9g   15.1g  56.3  27   processing-ind
104371   29.9g   15.1g  56.3   2   flx-disp-0
104404   29.9g   15.1g  55.3  29   processing-ind
104374   29.9g   15.1g  55.3   8   flx-disp-3
104375   29.9g   15.1g  55.3  10   flx-disp-4
104389   29.9g   15.1g  35.0  32   boardreader
104376   29.9g   15.1g  33.0   0   boardreader
```

Run 10884, new config

```
   PID    VIRT     RES  %CPU   P   COMMAND
218656   29.9g   15.1g  59.2  17   processing
218697   29.9g   15.1g  58.3  22   processing-ind
218652   29.9g   15.1g  58.3  13   processing
218654   29.9g   15.1g  58.3  15   processing
218658   29.9g   15.1g  58.3  19   processing
218629   29.9g   15.1g  57.3  12   processing
218698   29.9g   15.1g  57.3  24   processing-ind
218699   29.9g   15.1g  57.3  28   processing-ind
218700   29.9g   15.1g  57.3  26   processing-ind
218701   29.9g   15.1g  57.3  30   processing-ind
218660   29.9g   15.1g  57.3  21   processing
218704   29.9g   15.1g  57.3  27   processing-ind
218705   29.9g   15.1g  57.3  29   processing-ind
218633   29.9g   15.1g  56.3  16   processing
218635   29.9g   15.1g  56.3  18   processing
218702   29.9g   15.1g  56.3  23   processing-ind
218703   29.9g   15.1g  56.3  25   processing-ind
218706   29.9g   15.1g  56.3  31   processing-ind
218637   29.9g   15.1g  55.3  20   processing
218631   29.9g   15.1g  54.4  14   processing
218690   29.9g   15.1g  42.7  32   boardreader
218685   29.9g   15.1g  40.8  35   flx-disp-0
218686   29.9g   15.1g  40.8  37   flx-disp-1
218687   29.9g   15.1g  40.8  39   flx-disp-2
218689   29.9g   15.1g  40.8  43   flx-disp-4
218677   29.9g   15.1g  39.8  42   flx-disp-4
218688   29.9g   15.1g  39.8  41   flx-disp-3
218673   29.9g   15.1g  38.8  34   flx-disp-0
218674   29.9g   15.1g  38.8  36   flx-disp-1
218675   29.9g   15.1g  38.8  38   flx-disp-2
218676   29.9g   15.1g  38.8  40   flx-disp-3
218678   29.9g   15.1g  30.1   0   boardreader
```

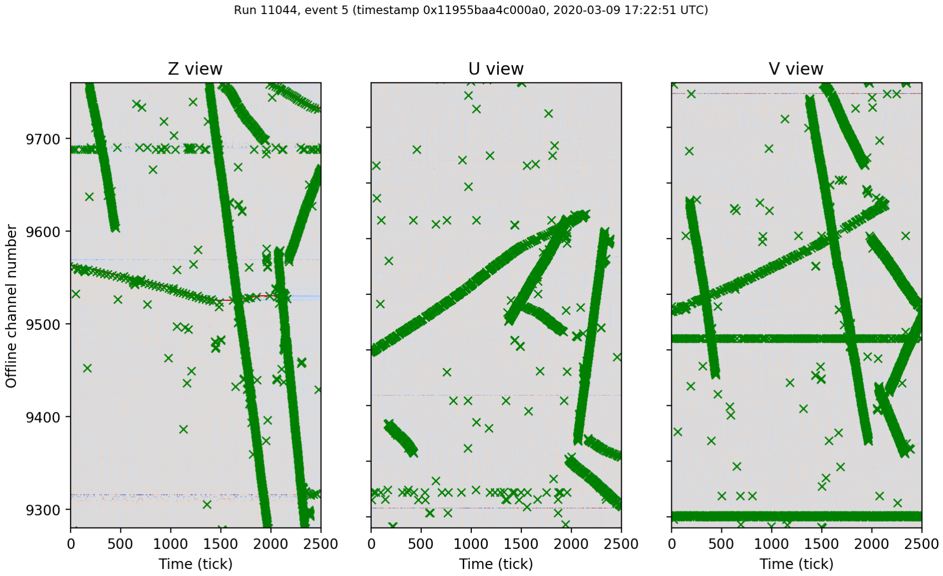
- Significant reduction in CPU% by rearranging thread pinning in a NUMA-aware way

Induction hit-finding example



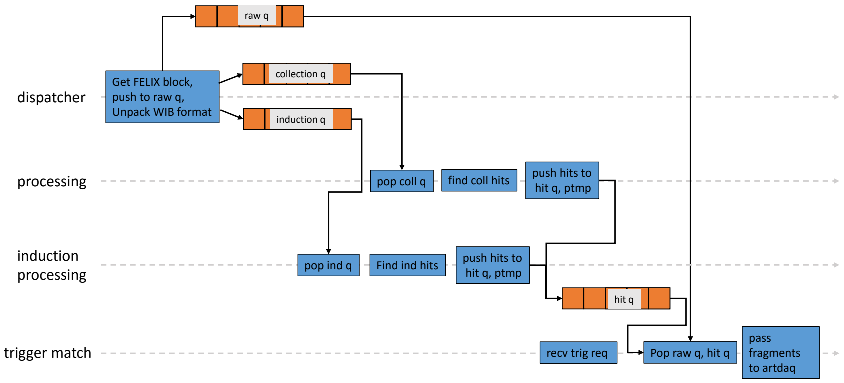
- Full APA in one server. Hits found continuously, buffered and read out in response to trigger

Induction hit-finding example



- Full APA in one server. Hits found continuously, buffered and read out in response to trigger

Improving induction hit-finding



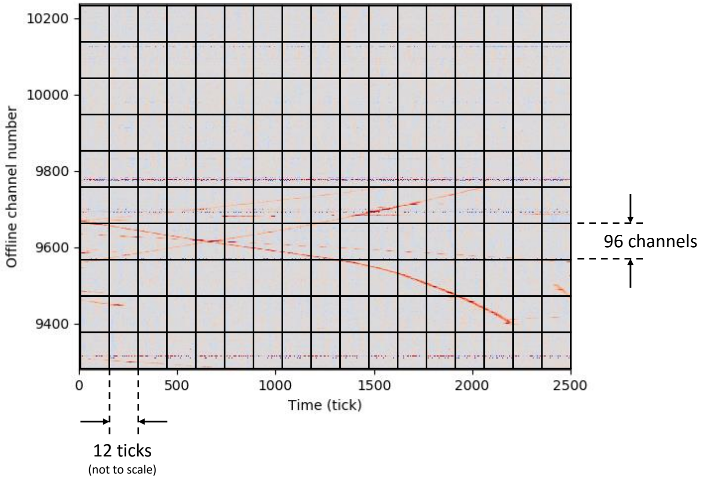
- Use some spare cycles on the dispatcher threads to free up cycles on the collection-wire processing thread
- Aim: reduce CPU% enough that we can use every HT core, process full APA on one *socket*

Full APA processing on half a server! (nearly)

```
   PID  VIRT      RES   %CPU   P  COMMAND
259901  40.0g    27.7g  92.1   4  flx-disp-1
259903  40.0g    27.7g  92.1   8  flx-disp-3
259904  40.0g    27.7g  92.1  10  flx-disp-4
259889  40.0g    27.7g  92.1  36  flx-disp-1
259890  40.0g    27.7g  92.1  38  flx-disp-2
259900  40.0g    27.7g  91.6   2  flx-disp-0
259902  40.0g    27.7g  91.6   6  flx-disp-2
259888  40.0g    27.7g  91.6  34  flx-disp-0
259891  40.0g    27.7g  91.6  40  flx-disp-3
259892  40.0g    27.7g  91.6  42  flx-disp-4
259913  40.0g    27.7g  82.8  54  processing-ind
259912  40.0g    27.7g  80.8  44  processing-ind
259918  40.0g    27.7g  80.8  50  processing-ind
259917  40.0g    27.7g  79.8  48  processing-ind
259921  40.0g    27.7g  79.8  52  processing-ind
259915  40.0g    27.7g  79.3  46  processing-ind
259919  40.0g    27.7g  77.3  60  processing-ind
259916  40.0g    27.7g  76.8  58  processing-ind
259920  40.0g    27.7g  76.8  62  processing-ind
259914  40.0g    27.7g  75.9  56  processing-ind
259846  40.0g    27.7g  61.1  22  processing
259850  40.0g    27.7g  59.1  12  processing
259860  40.0g    27.7g  58.6  18  processing
259875  40.0g    27.7g  58.1  20  processing
259856  40.0g    27.7g  57.6  16  processing
259854  40.0g    27.7g  57.1  14  processing
259857  40.0g    27.7g  56.2  28  processing
259848  40.0g    27.7g  55.7  24  processing
259852  40.0g    27.7g  55.7  26  processing
259861  40.0g    27.7g  54.7  30  processing
259905  40.0g    27.7g  32.0  32  flx-reader-0
259893  40.0g    27.7g  31.5   0  flx-reader-0
260117  40.0g    27.7g   2.5  37  boardreader
260118  40.0g    27.7g   2.0  21  boardreader
260129  40.0g    27.7g   1.0  25  boardreader
260122  40.0g    27.7g   0.5   0  ZMQbg/3
260128  40.0g    27.7g   0.5  11  boardreader
260131  40.0g    27.7g   0.5  13  boardreader
260130  40.0g    27.7g   0.5  19  boardreader
260132  40.0g    27.7g   0.5  29  boardreader
```

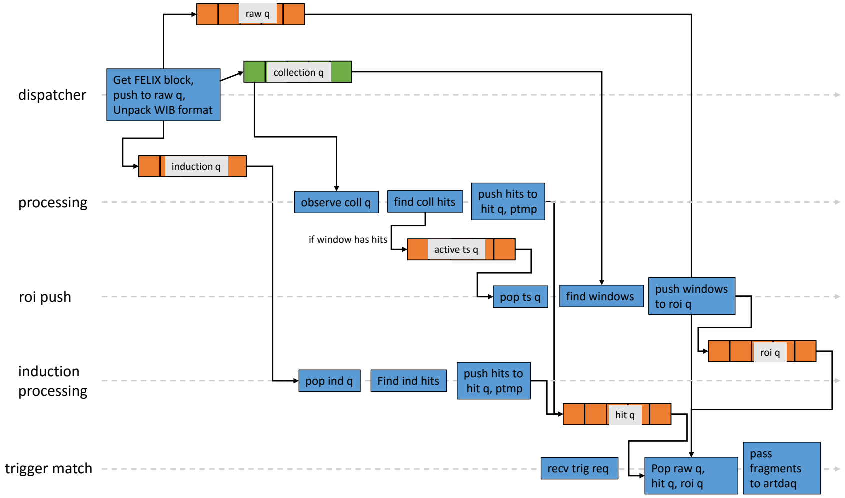
- It nearly worked! 10 dispatcher threads, 10 collection-wire processing threads, 10 induction-wire processing threads, 2 card-reader threads, fit on one CPU socket (Intel Xeon Gold 6242 CPU @ 2.80GHz, 32 (HT) cores)
- Have to use both HyperThread cores in each physical core
- Why you might care: suggests we can run 2 APAs per server, with 2019 technology
- (Not quite: haven't shown that there's a factor 2 more memory bandwidth available, and some `artdaq` threads are running on NUMA 1)
- I tried again yesterday: CPU usage is higher, and can't keep up when triggers start. But *this* close!
- Run 10964, APA 6 only, `P` is core # of thread. All even, ie CPU 0

Poor man's ROI readout



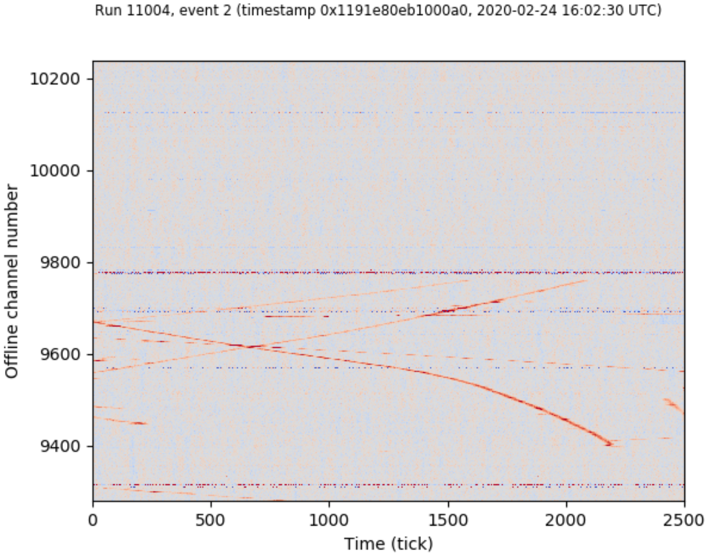
- Consider each 12-tick$\times$96-collection-channel block as a quantum for reading out
- Just collection channels for now
- A block is "active" if, in any of its (non-noisy) channels, either: a hit was ongoing at the end of the 12 ticks, or a hit ended during the 12 ticks. Read out active blocks and blocks $\pm$50 ticks

Poor man's ROI readout: implementation



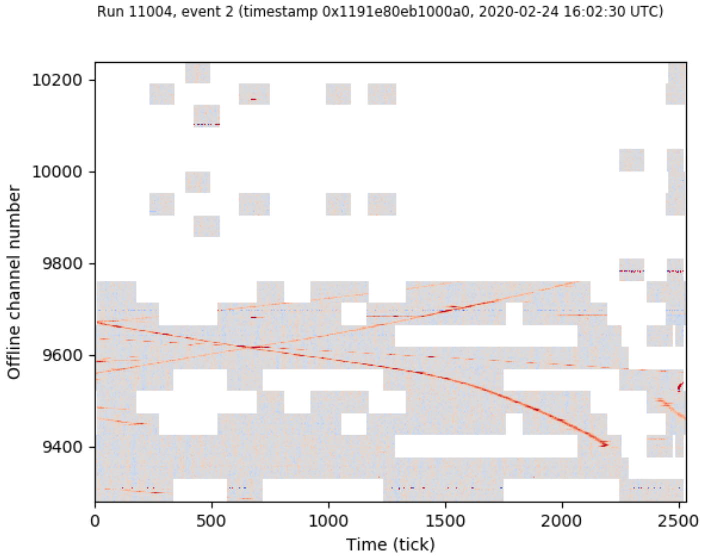
- Need complicated implementation to get the 50-tick-before condition (otherwise we could just directly push a block when it's active). Suppress known noisy channels
- ROIs, like hits, are found continuously and buffered, waiting for trigger request. Each ROI carries a timestamp

Poor man's ROI readout: example, all waveforms



- Rate is low because detector running below nominal HV. Top half of channels are wall-facing

Poor man's ROI readout: example, ROIs

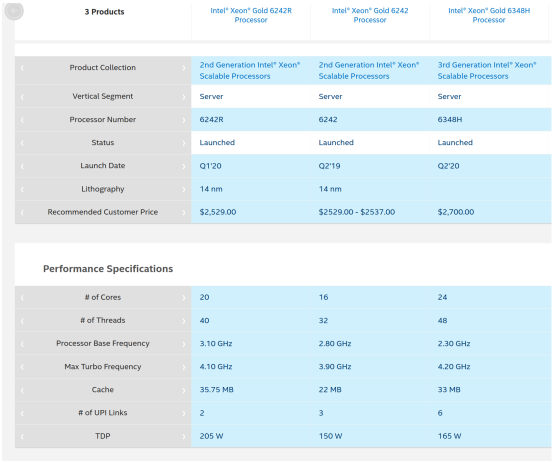


- Each link holds two contiguous sets of channels, so it looks like 20 rows in the plot

But what about latency?

- Linux is not a realtime system: no guarantees that your code is scheduled to run within a fixed time of some event occurring. The kernel scheduler could decide not to run your process for some arbitrary amount of time
- I don't think this is a problem in our case:
  - Buffer depth is several seconds, not microseconds
  - With CPU pinning, scheduler is smart and doesn't schedule other tasks on pinned cores
  - We could go further and *force* the kernel not to schedule on the cores we're using, with `isolcpus`

Processor comparison



3 Products	Intel® Xeon® Gold 6242R Processor	Intel® Xeon® Gold 6242 Processor	Intel® Xeon® Gold 6348H Processor
Product Collection	2nd Generation Intel® Xeon® Scalable Processors	2nd Generation Intel® Xeon® Scalable Processors	3rd Generation Intel® Xeon® Scalable Processors
Vertical Segment	Server	Server	Server
Processor Number	6242R	6242	6348H
Status	Launched	Launched	Launched
Launch Date	Q1'20	Q2'19	Q2'20
Lithography	14 nm	14 nm	
Recommended Customer Price	$2,529.00	$2529.00 - $2537.00	$2,700.00

Performance Specifications

# of Cores	20	16	24
# of Threads	40	32	48
Processor Base Frequency	3.10 GHz	2.80 GHz	2.30 GHz
Max Turbo Frequency	4.10 GHz	3.90 GHz	4.20 GHz
Cache	35.75 MB	22 MB	33 MB
# of UPI Links	2	3	6
TDP	205 W	150 W	165 W

- From ark.intel.com, retrieved 2020-07-29
- Middle column is the processor I've reported results from