Multi-processing for ND-LAr in **larnd-sim** and **ndlar_flow**

Matt Kramer, LBNL
(on behalf of P. Madigan, R. Soleti, and many others)

LArTPC multi-threading workshop, Mar 3, 2023

Outline

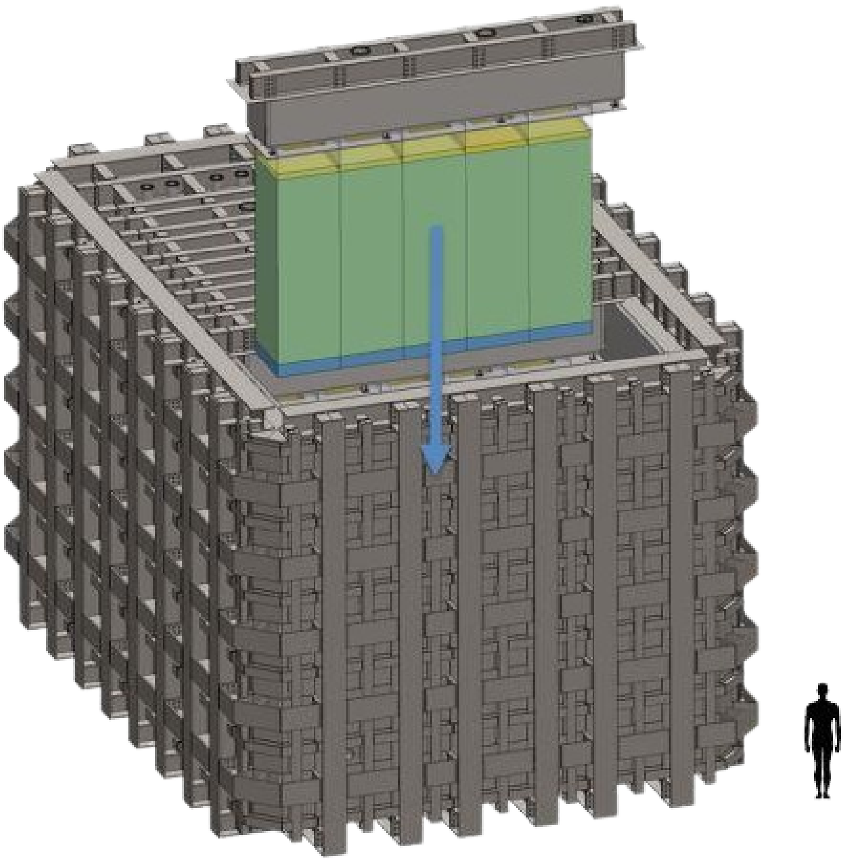
- DUNE ND-LAr, ArgonCube, LArPix, and the prototypes
- The **larnd-sim** simulation and its use of GPUs
- The **ndlar_flow** calib/reco and its use of MPI (via **h5flow**)
- Philosophical ramblings, mercifully brief

Some quick remarks

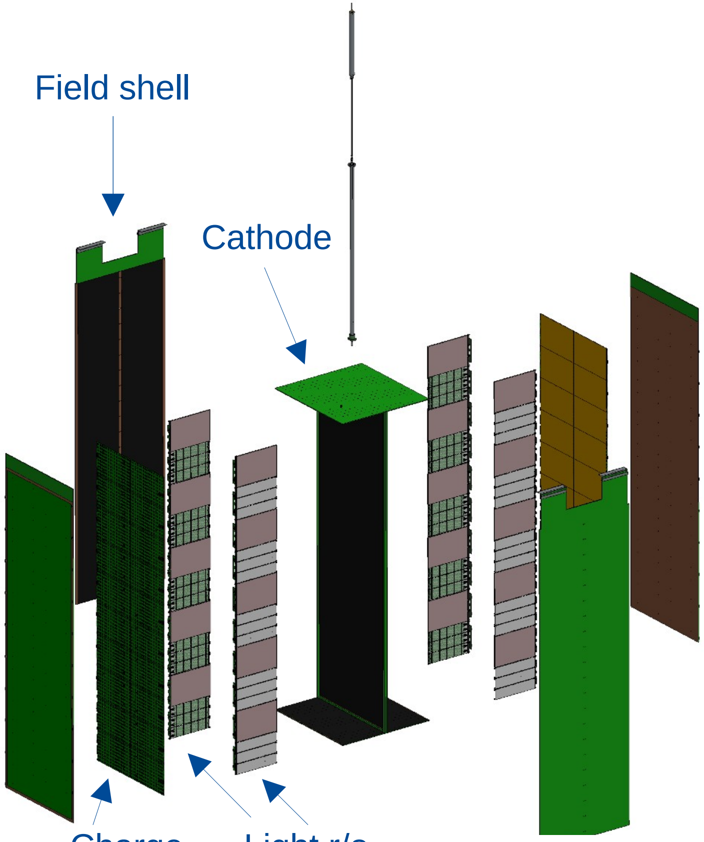
- This is a fluffy talk, full of stolen material; **I did not write larnd-sim or ndlar_flow,** nor have I had much past involvement in ND-LAr, ArgonCube, or LArPix
  - I also know very little about LArSoft (but do know Gaudi)
- But I work with those who *do* deserve the credit, and am broadly interested in the use of HPC within HEP
- I've aimed for this talk to serve a few purposes:
  - Share the multi-processing strategies used by larnd-sim and ndlar_flow
    - They differ both from each other and from others discussed in this workshop
  - Provide a general intro to the existing ND-LAr/2x2 software chain
  - Promote the idea of a diverse, interoperable ecosystem of software and data, so that creativity can flourish and we can learn from each other

DUNE ND-LAr

- Liquid argon near detector: Essential for the DUNE long-baseline analysis
  - Constrains flux, xsec, detector systematics
  - Same target as far detectors
- Pileup in high-rate environment
  - Traditional wire readout unfeasible
- ND-LAr: A 7x5 array of optically segmented *ArgonCube* modules with **pixel readout**

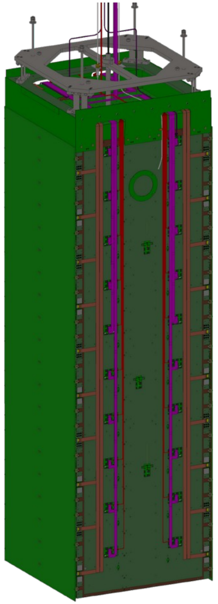


ArgonCube



- A **modular LArTPC** for combined deployment in arbitrarily large, segmented detectors
- Central cathode: 2 drift regions
- Pixelated charge readout using novel LArPix ASIC
- Flat-panel light detectors coupled to SiPMs; 2 alternating designs
  - ArcLight: Continuous dichroic film
  - LCM: Winding fibers

ProtoDUNE-ND and the 2x2

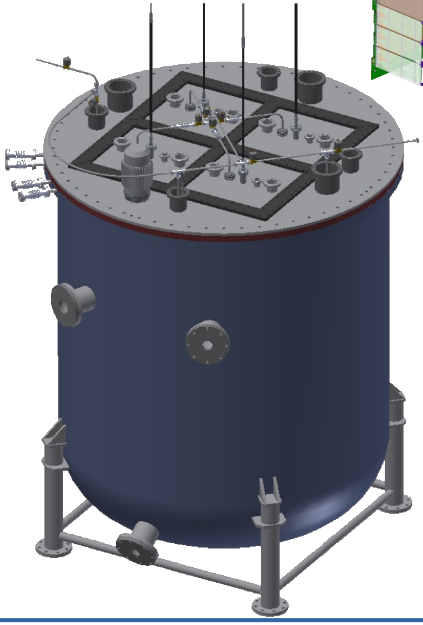


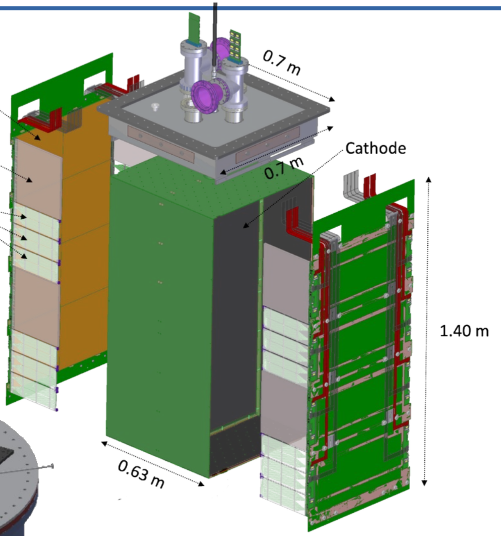
Full scale
ND-LAr module
$1 \times 1 \times 3$ m^3

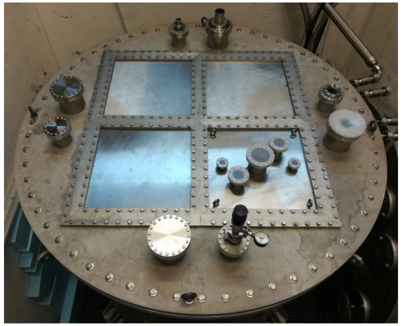
ProtoDUNE-ND
$0.7 \times 0.7 \times 1.4$ m^3

x4

2x2 demonstrator



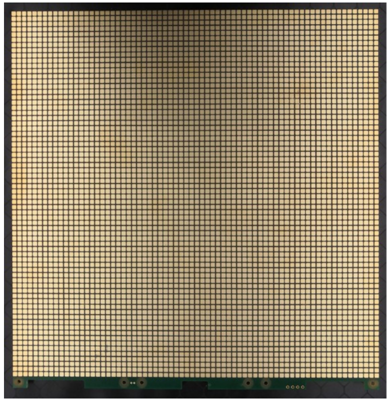




- **ProtoDUNE-ND:** Full-featured, scaled-down ArgonCube prototype
  - Successful demonstration of technology
- 4 modules to be deployed in **2x2 demonstrator** in NuMI beam @ FNAL
  - Tested @ LHEP in Bern
  - Commissioning @ FNAL; neutrino beam this year!

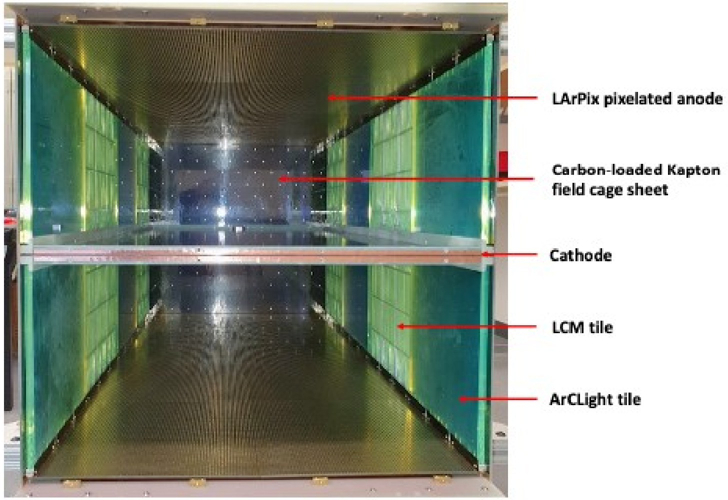
Charge readout: LArPix

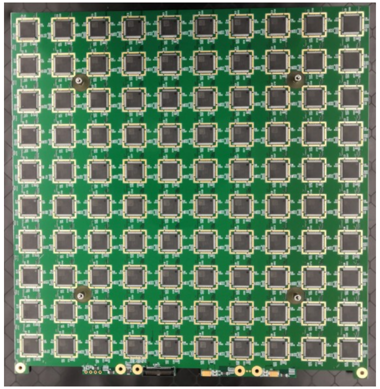
- Novel ASIC for pixelated charge readout
  - Cold amplifiers, ADCs, IO
- Self-triggering pixels; continous stream of hits
- Flexible “hydra” network for IO routing
- Driven and read out by “Pacman” board;
  - Pacman communicates to DAQ machine via ZeroMQ/ethernet
  - DAQ → “raw” HDF5 files
    - For further analysis, convert to “packet” HDF5 (same as produced by larnd-sim)



Tile front: pixel pads

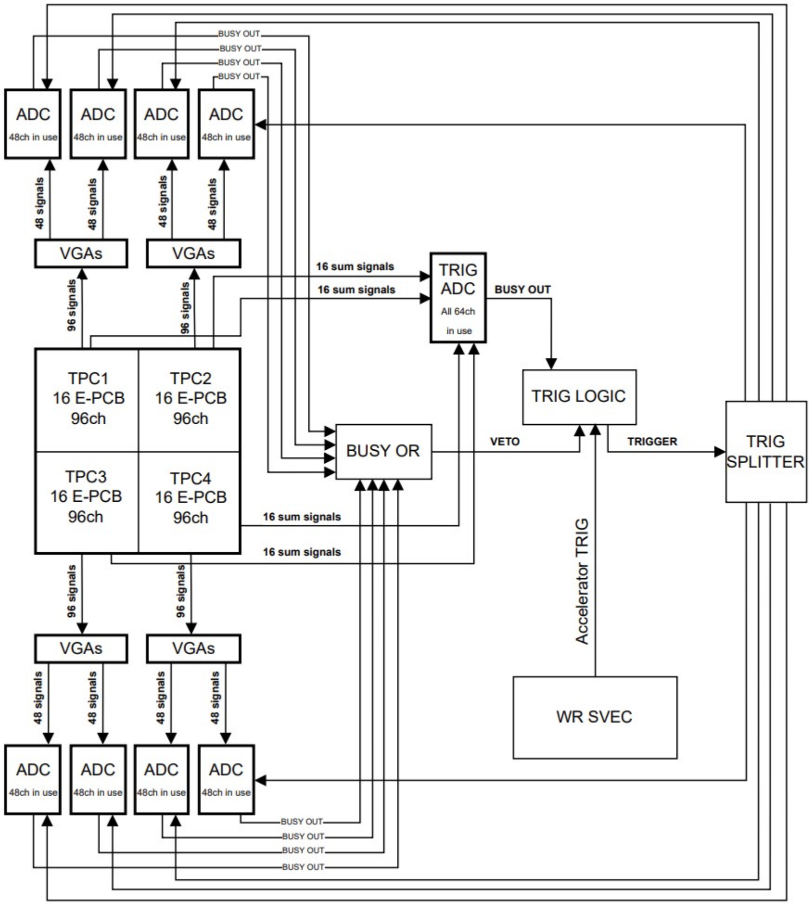
8 (16) tiles per ProtoDUNE-ND TPC (Module)





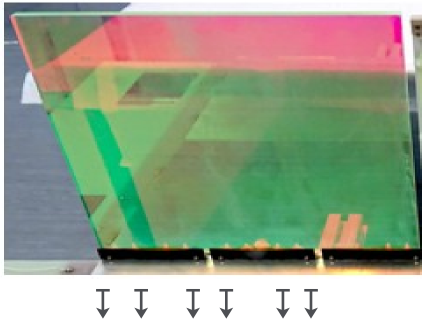
Tile back: 10x10 LArPix ASICs

Light readout: ADC64



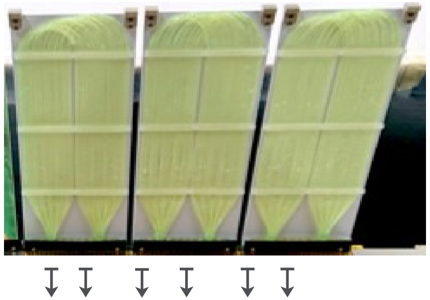
- Independent DAQ from LArPix
- 6 SiPMs per light tile
  - 8 (16) tiles per ProtoDUNE-ND TPC (module)
- 1 (1) ADC64 for all 8 ArcLight (LCM) tiles in a module
  - 8*6 = 48 active channels per ADC (out of 64)
  - Add'l ADC takes tile-sum signals for triggering (16 channels)
- DAQ → triggered waveforms in binary ADC64 format
  - Converted on-the-fly to HDF5 in ndlar_flow

ArCLight tile



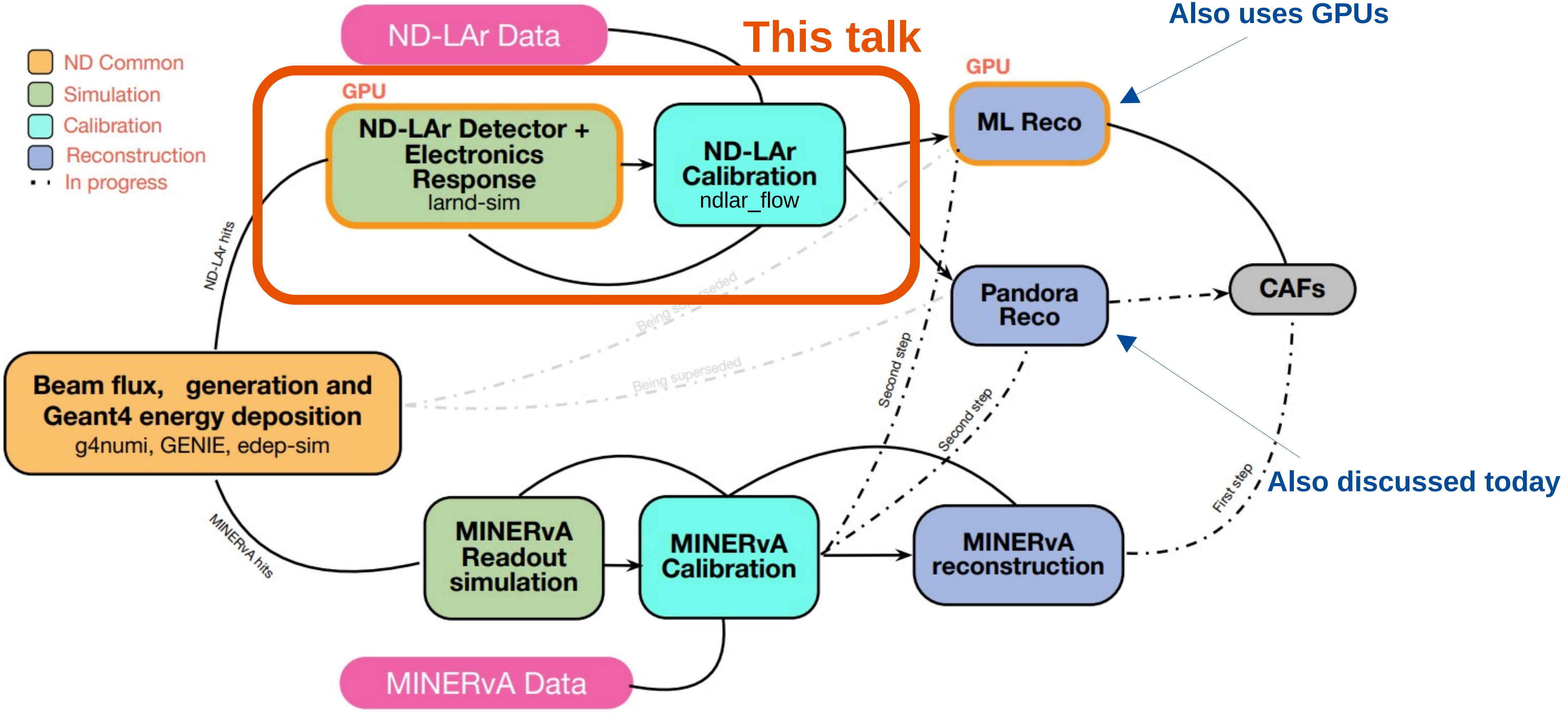
6 SiPMs

LCM tile



2 SiPMs 2 SiPMs 2 SiPMs

2x2 software chain



Array-oriented Python

- Both larnd-sim and ndlar_flow are written in **array-oriented Python**
  - Vector operations (numpy, …), not loops
    - Avoids the performance penalty of looping in Python
    - Makes automatic acceleration (numba, …) more likely to succeed (larnd_sim)
    - Makes MPI-ification easy: Just slice (ndlar_flow / h5flow)
  - Structs of arrays, not arrays of structs
    - OK, fine, sometimes arrays of *simple* structs
    - In any case, Plain Old Data -- no attached behavior (i.e. methods)
      - Can use and interpret without specialized libraries

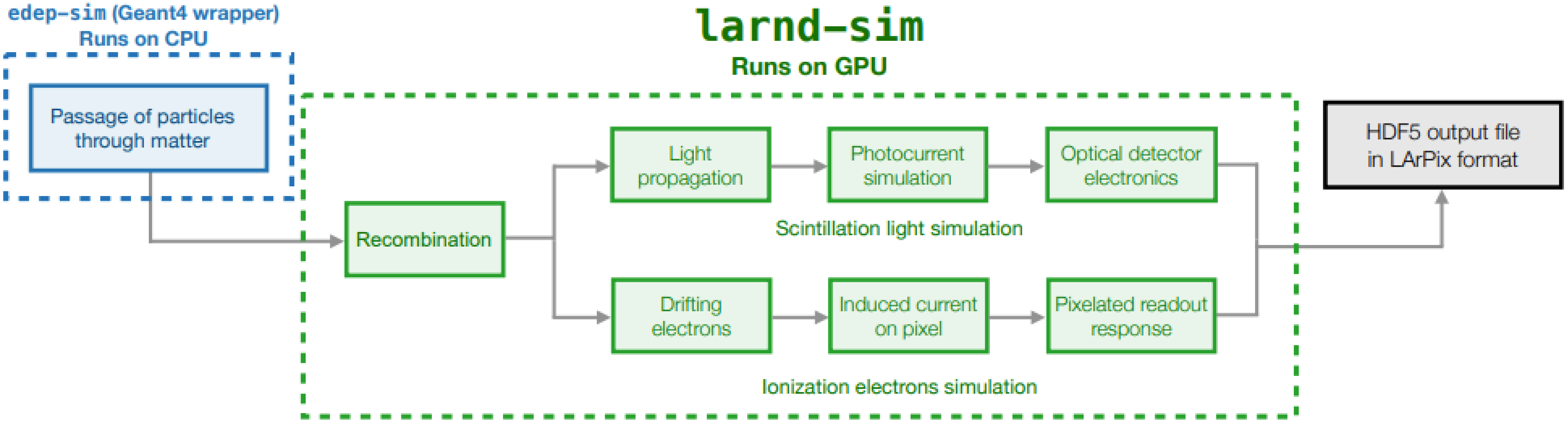
HDF5

- Both larnd-sim and ndlar_flow use HDF5 almost everywhere
  - Widely adopted, supported in many programming languages; small, specialized library
    - Compared to ROOT: Either install all of ROOT (great, but huge), or use something like uproot (great, but incomplete)
  - HDF5 datasets map well to Numpy arrays; good match for this programming style
  - Long-term data accessibility: Formally specified format, self-describing files, readable with nothing but a generic HDF5 library
  - To be fair, the official C++ API is painful, but there's e.g. HighFive
    - The de facto Python interface (h5py) is *nice*, though

larnd-sim design

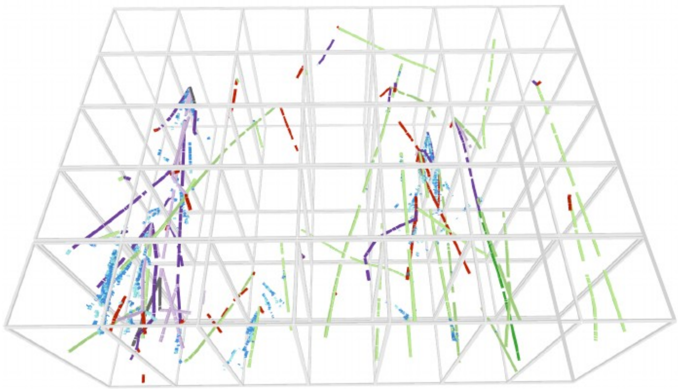
- Completely written in Python
- All heavy computations on GPU (~15 kernels)
- Largely developed by 2 people
  - With pieces from many others; low barrier to contributing

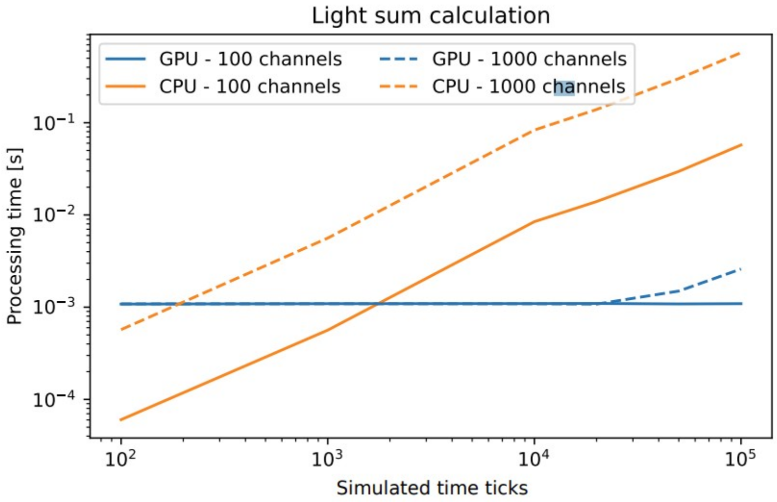
- Input: edep-sim energy deposits in HDF5
- Output: “Packet” data, as from DAQ; plus truth info
- Idiomatic Python, JIT-compiled to CUDA
  - Just apply @numba.cuda.jit decorator
  - cupy: numpy on the GPU



larnd-sim: Why GPUs?

- A massively parallelizable problem, over:
  - Energy depositions: ionization, recombination, diffusion, drifting, scintillation
  - Photons: propagation, detection
  - Pixels: induced current, electronics response, digitization
- *N* is high, elements are independent, and calculations can be expressed with “just math” and minimal branching
  - GPU’s bread and butter
- HPC facilities increasingly providing GPUs
  - Follow the FLOPS





larnd-sim: Some code

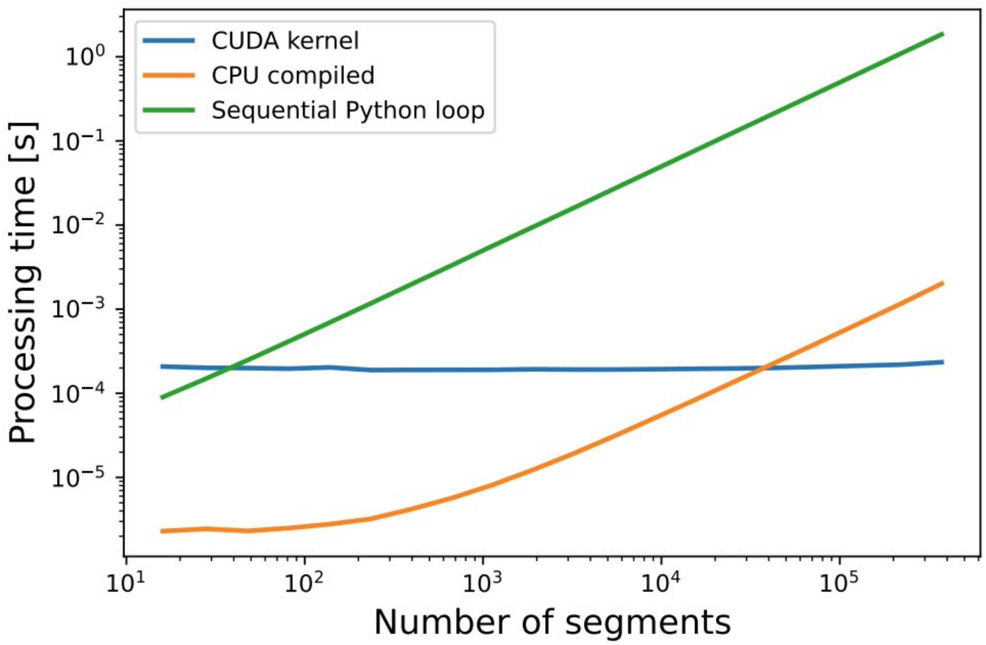
Calling a kernel

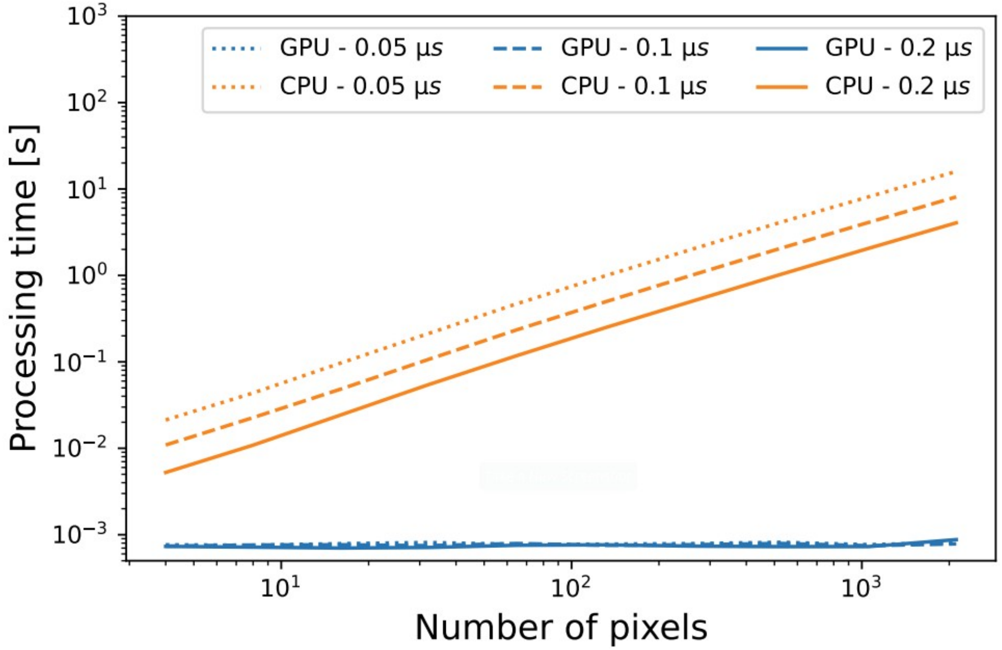
```
track_pixel_map = cupy.full((unique_pix.shape[0], detsim.MAX_TRACKS_PER_PIXEL), -1)
TPB = 32                                             # threads per block
BPG = max(ceil(unique_pix.shape[0] / TPB),1)   # blocks per grid
detsim.get_track_pixel_map[BPG, TPB](track_pixel_map, unique_pix, neighboring_pixels)
```

```
@cuda.jit
def get_track_pixel_map(track_pixel_map, unique_pix, pixels):
    index = cuda.grid(1)
    upix = unique_pix[index]
    for itrk in range(pixels.shape[0]):
        for ipix in range(pixels.shape[1]):
            pID = pixels[itrk][ipix]
            if upix == pID:
                imap = 0
                while (imap < track_pixel_map.shape[1] and track_pixel_map[index][imap] != -1
                       and track_pixel_map[index][imap] != itrk):
                    imap += 1
                if imap < track_pixel_map.shape[1]:
                    track_pixel_map[index][imap] = itrk
```

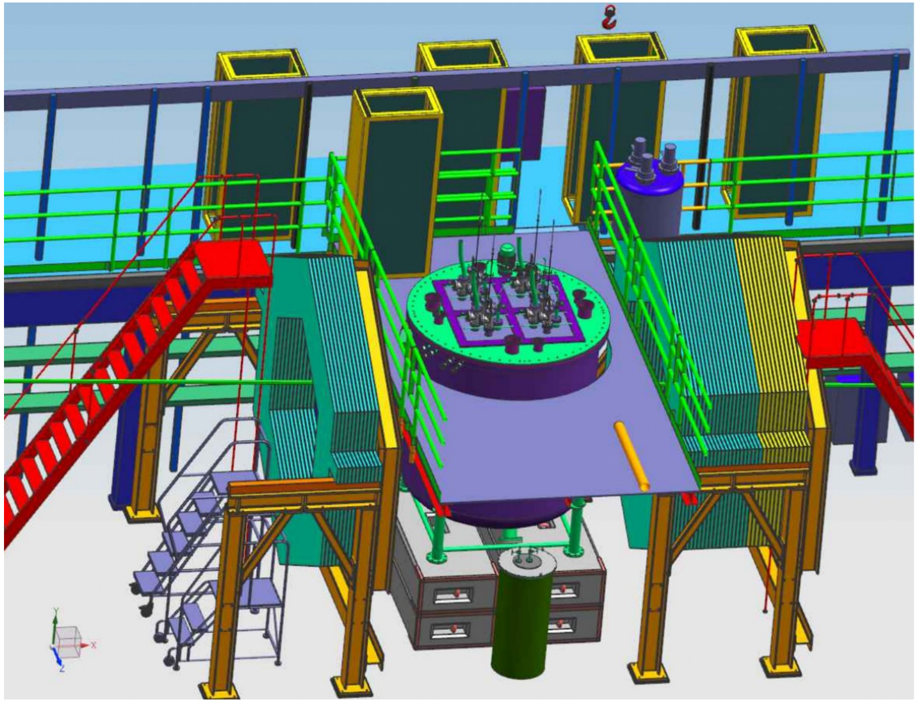
Defining a kernel

larnd-sim scaling





larnd-sim deployment

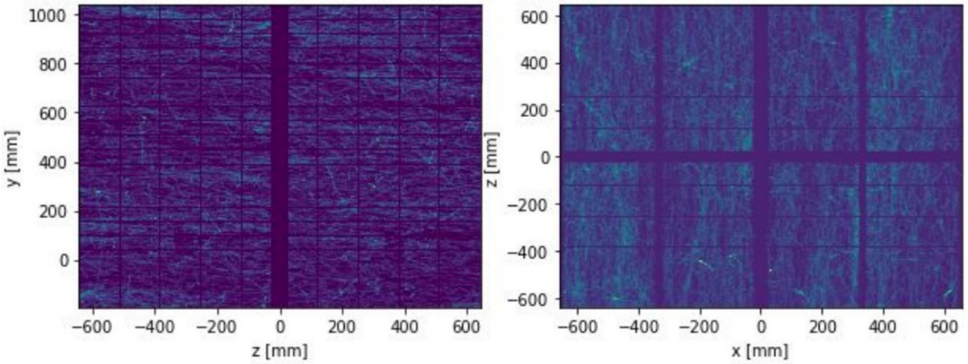


- For analysis of expected ~1yr 2x2-NuMI data, need ~10x sim statistics: O(10^{22} POT)
- Plan is to produce on NERSC Perlmutter system (A100 GPUs, 4x/node)
- O(100k) GPU-node-hours
- Compare to the cost of crunching all those numbers on CPUs!

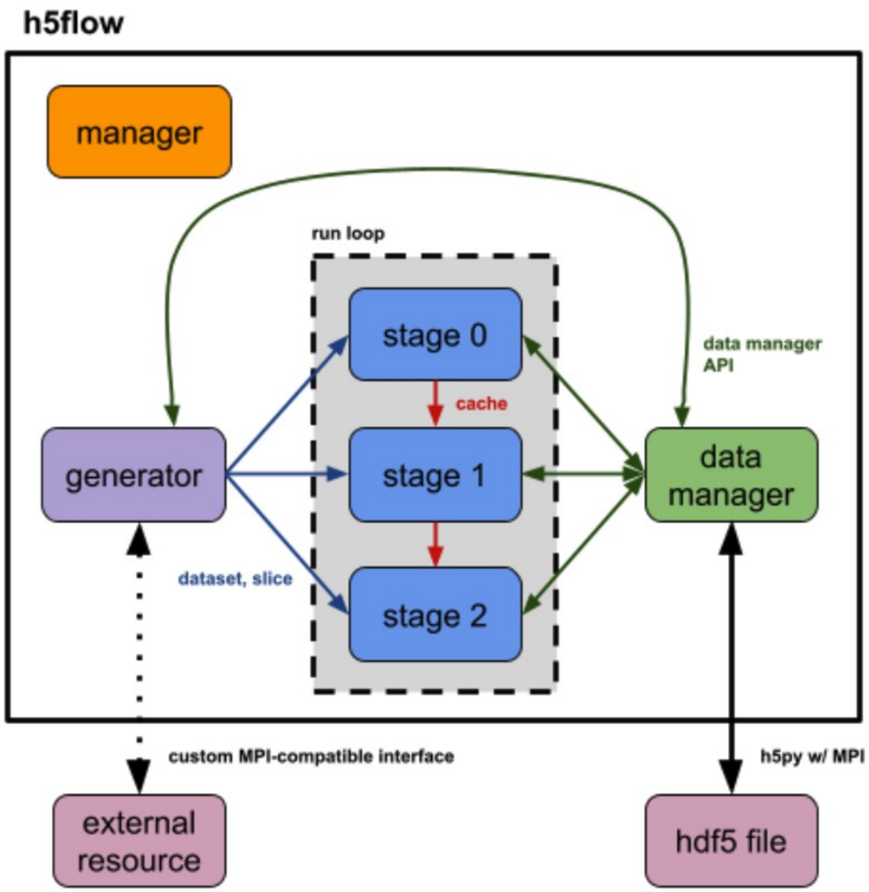
Flowing along: ndlar_flow

- **ndlar_flow**: Low-level calibration and basic reconstruction of charge+light data, real/simulated
  - Light waveforms: denoising, deconvolution, hit finding
  - Charge hits: Pedestal subtraction, ADC → charge → energy, “undrifting”
  - Event building, charge/light matching (t0)
  - Combined reco (tracklets)

1. LArPix packets from Pacman DAQ
   1.1. Map software channel to detector location
   1.2. Subtract predetermined pedestals
   1.3. ADC → ke$^-$ calibration assuming uniform gain,
   1.4. ke$^-$ → MeV calibration assuming fixed dx
   1.5. Reconstruct drift coordinate
   1.6. Correct ADC → ke$^-$ for gain variations
   1.7. ke$^-$ → MeV calibration (refined)
   1.8. Calibrate for detector distortion, electric field, etc.



ndlar_flow design



- Also pure Python, but runs on CPUs, not GPUs
- Performance from array ops, avoiding loops
- Built on **h5flow** framework:
  - Can be mentally mapped onto <insert_framework_here>, but much simpler
  - "Automatic" parallelism: Dataset slices distributed via MPI
  - Flexible configuration via YAML files
  - Provenance tracking: Reference links are stored between parent and child datasets
    - Dereferencing possible in both directions, across multiple links
  - Start with arrays of raw data; successively add arrays of higher-level quantities

Closing thoughts

- larnd_sim and ndlar_flow were largely developed by one grad student and one postdoc, yet:
  - They both contain an incredible amount of carefully validated physics
  - They are inherently parallel and ready to take advantage of next-gen GPU (larnd-sim) and CPU (ndlar_flow) facilities
- How much of this can be credited to the flexibility and productivity offered by Python, simple data formats, etc.?
- How do we weigh those advantages against the complementary advantages of a formalized C++ framework like LArSoft?
- How do we balance coherence and consistency against creativity, innovation, and readiness for new hardware architectures?
- How do we get the best of both worlds? And if there are many worlds, how do we ensure they can interoperate?

Further reading

- larnd-sim paper:
  Highly-parallelized simulation of a pixelated LArTPC on a GPU
- Githubs:
  - larnd-sim
  - h5flow
  - ndlar_flow
  - larpix-control
  - adc64format