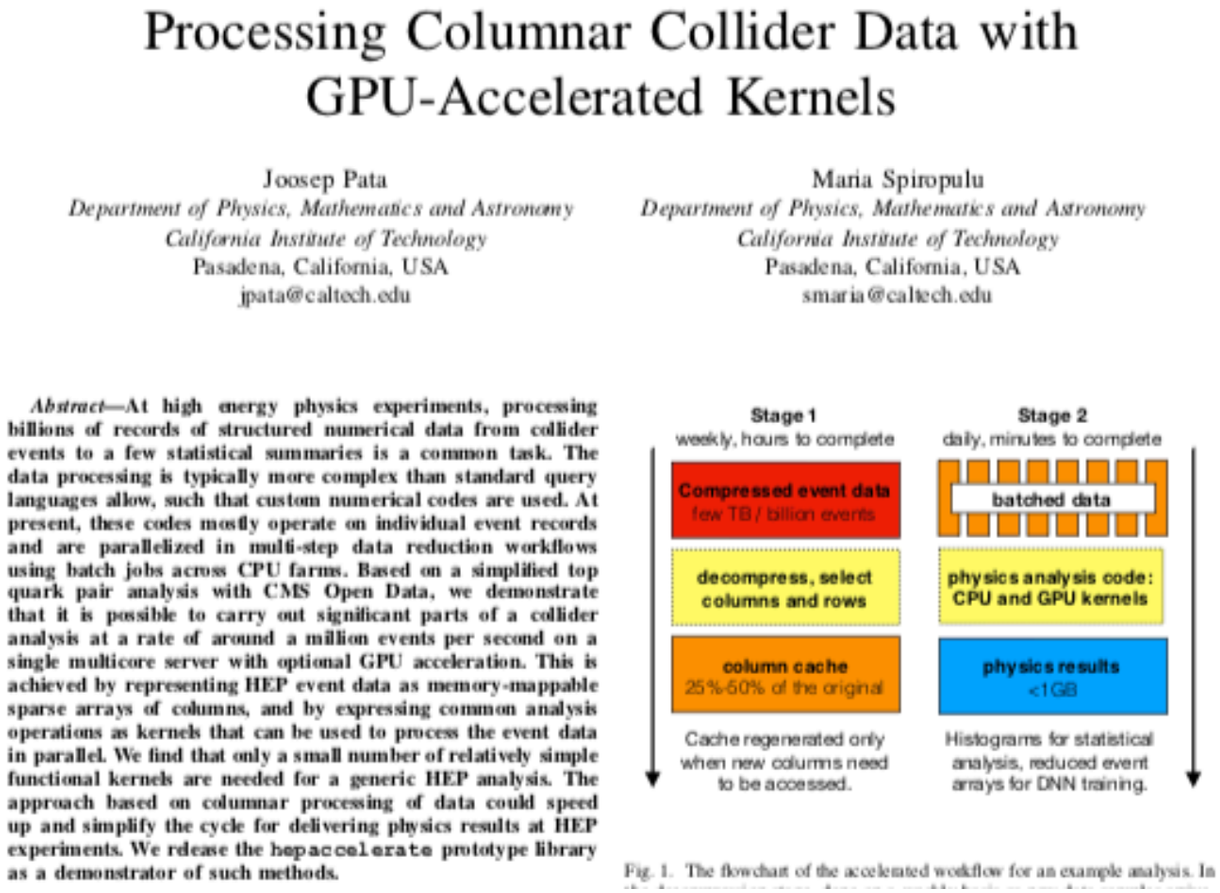
hepaccelerate: data analysis with jagged arrays on GPUs

Joosep Pata, Maria Spiropulu
California Institute of Technology

October 16, 2019
PyHEP 2019 @ Abingdon, UK

Overview

- Back-of-the-envelope feasibility study for MHz data analysis
- Numba as a simple way to write down and test array kernels
- CMS Open Data top quark pair proto-analysis: implementation and performance
- Discussion on applicability for analysis



Processing Columnar Collider Data with GPU-Accelerated Kernels

Joosep Pata
Department of Physics, Mathematics and Astronomy
California Institute of Technology
Pasadena, California, USA
jpata@caltech.edu

Maria Spiropulu
Department of Physics, Mathematics and Astronomy
California Institute of Technology
Pasadena, California, USA
smaria@caltech.edu

***Abstract*—At high energy physics experiments, processing billions of records of structured numerical data from collider events to a few statistical summaries is a common task. The data processing is typically more complex than standard query languages allow, such that custom numerical codes are used. At present, these codes mostly operate on individual event records and are parallelized in multi-step data reduction workflows using batch jobs across CPU farms. Based on a simplified top quark pair analysis with CMS Open Data, we demonstrate that it is possible to carry out significant parts of a collider analysis at a rate of around a million events per second on a single multicore server with optional GPU acceleration. This is achieved by representing HEP event data as memory-mappable sparse arrays of columns, and by expressing common analysis operations as kernels that can be used to process the event data in parallel. We find that only a small number of relatively simple functional kernels are needed for a generic HEP analysis. The approach based on columnar processing of data could speed up and simplify the cycle for delivering physics results at HEP experiments. We release the `hepaccelerate` prototype library as a demonstrator of such methods.**

Fig. 1. The flowchart of the accelerated workflow for an example analysis. In

https://arxiv.org/pdf/1906.06242v2.pdf
github.com/hepaccelerate/hepaccelerate

Problem setup

- Suppose you are working on an analysis of the full Run 2 dataset, roughly 1 billion skimmed MC + data events, preapproval imminent

- Suddenly, a new set of jet calibrations (or b-tag, lepton, etc) arrive from the physics object group!

- Need to re-ntuplize all the things on the grid, every plot and table will have to be remade! This could take weeks!

- Can we skip the ntuplization and **go directly to the physics studies**?

- HEP analysis on the fly using high-performance machines!

- **Goal: make it effortless to reproduce the physics results from a well-understood baseline dataset such as CMS NanoAOD**

Back-of-the-envelope

- **1B events ~ 2TB** for a typical NanoAOD-like analysis ntuple
- Suppose a desired turn-around time of **500s @ 100kHz / thread**
- Need just **20 threads, bandwidth of 4 GB/s** → feasible on a multicore server with fast local disk
- Set up a **realistic but open physics analysis** within these parameters
- Find the bottlenecks and possibilities for **accelerating analysis using modern hardware**

Analysis flow

Stage 1
weekly, hours to complete

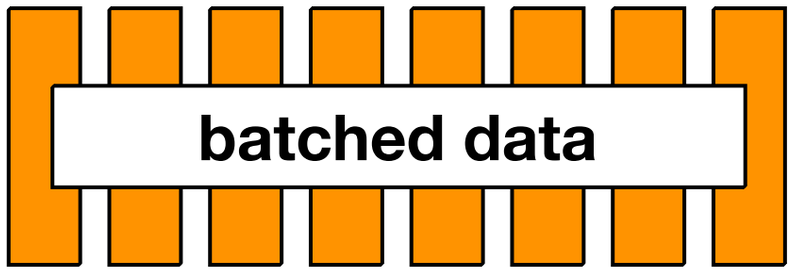
Compressed event data
few TB / billion events

decompress, select columns and rows

column cache
25%-50% of the original

Cache regenerated only when new columns need to be accessed.

Stage 2
daily, minutes to complete



physics analysis code: CPU and GPU kernels

physics results
<1GB

Histograms for statistical analysis, reduced event arrays for DNN training.

Jagged arrays

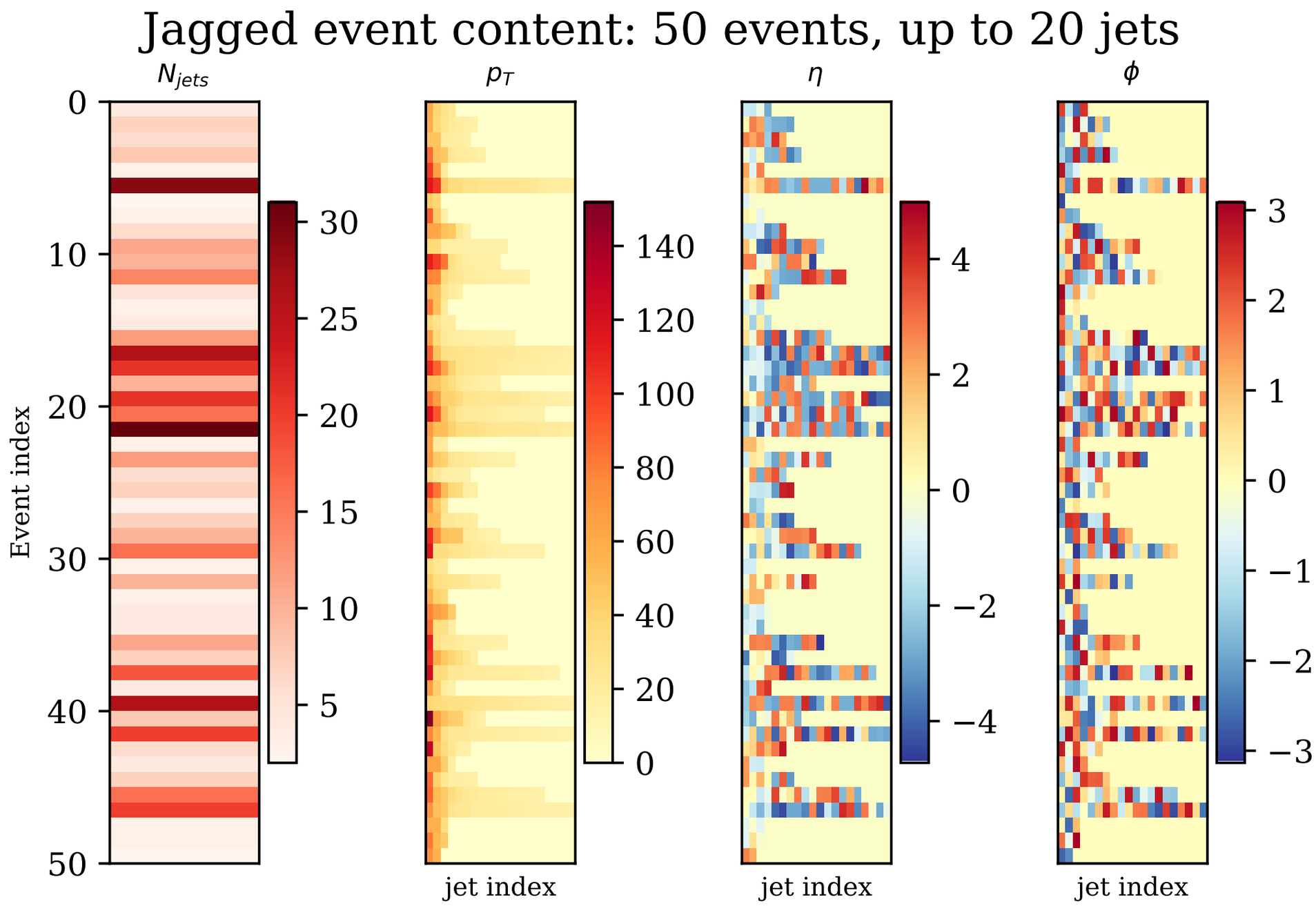
HEP analysis datasets can be loaded as numpy arrays with sparse structure using `uproot` & `awkwkard`!

We can use the `arr.content` and `arr.offsets` directly in numerical codes!

```
arr.content
memmap([61.29351 , 39.545116, 28.863787, ..., 86.62336 , 80.02655 ,
        60.16085 ], dtype=float32)

arr.offsets
memmap([       0,        4,       11, ..., 18039610, 18039617, 18039620],
       dtype=uint64)
```

Jagged arrays



Jagged arrays are essentially a specialized sparse matrix format!

Kernels

Suppose we want to compute the sum(pT) of all the jets in all the events.

In [11]:
```
def sumpt_event(data, offsets, ret):
    ret[:] = 0
    for iev in range(len(offsets)-1):
        i0 = offsets[iev]
        i1 = offsets[iev+1]
        for j in range(i0, i1):
            ret[iev] += data[j]
    return ret

sum1 = np.zeros(len(pt.offsets)-1, dtype=np.float64)
%timeit sumpt_event(pt.content, pt.offsets, sum1)
```

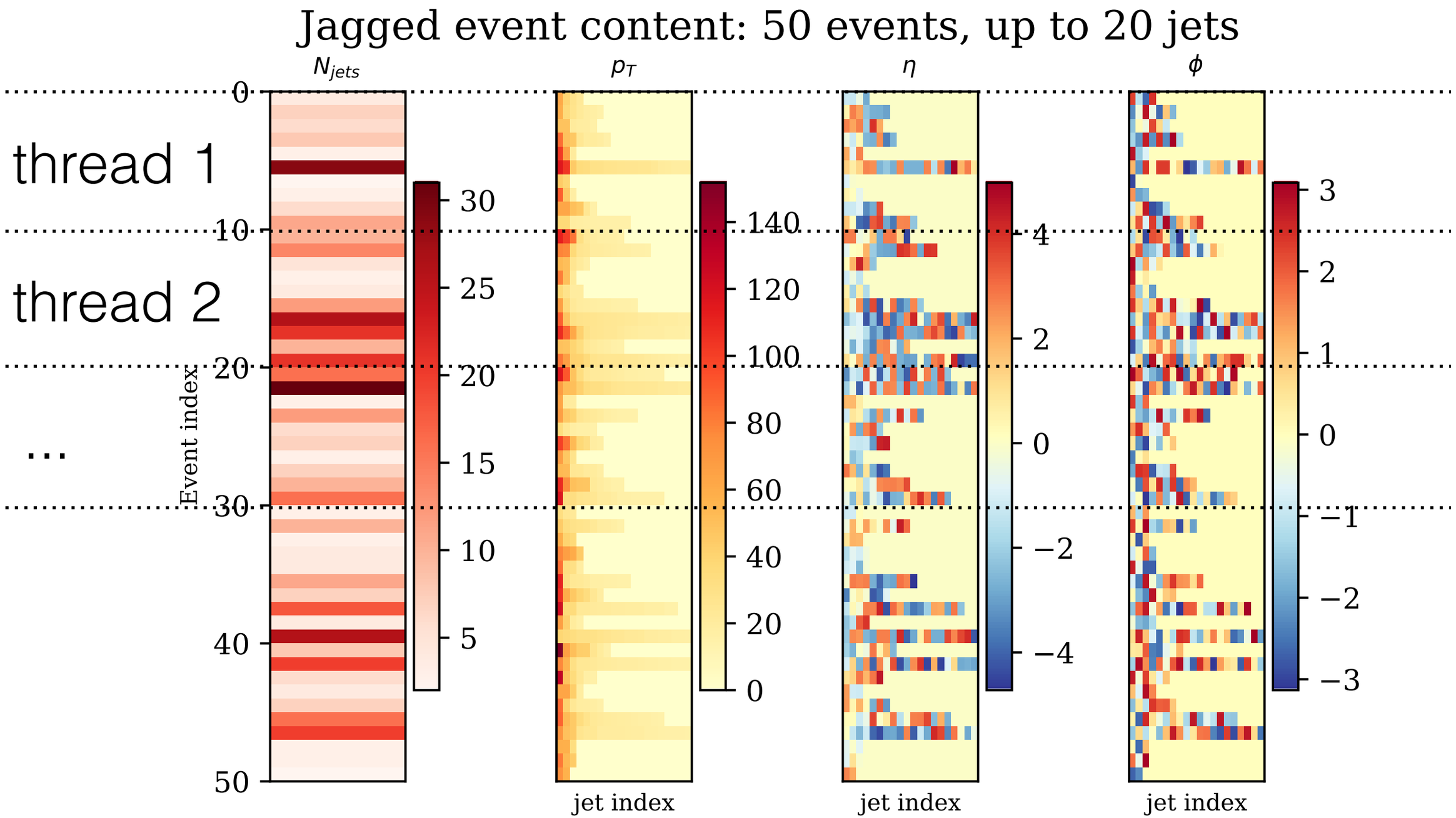
```
18.9 s ± 63.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [12]:
```
%timeit -n3 pt.sum()
```

```
136 ms ± 221 µs per loop (mean ± std. dev. of 7 runs, 3 loops each)
```

Of course, this is kind of sum is much more easily and fast computed with `awkward`!

Multithreading



Can use multithreading to process events in parallel!

Numba for kernels

Numba is a JIT compiler for numerical python: http://numba.pydata.org/

In [16]:
```
@numba.njit(parallel=True)
def sumpt_event(data, offsets, ret):
    ret[:] = 0
    for iev in numba.prange(len(offsets)-1):
        i0 = offsets[iev]
        i1 = offsets[iev+1]
        for j in range(i0, i1):
            ret[iev] += data[j]
    return ret
```

In [19]:
```
sum2 = np.zeros(len(pt.offsets)-1, dtype=np.float64)
%timeit -n3 sumpt_event(pt.content, pt.offsets, sum1)
```

```
8.16 ms ± 297 µs per loop (mean ± std. dev. of 7 runs, 3 loops each)
```

We can compile the same function using Numba, **targetting a multithreaded CPU**!

Numba on GPUs

In [57]:
```
@cuda.jit
def sumpt_event_cudakernel(data, offsets, ret):
    xi = cuda.grid(1)
    xstride = cuda.gridsize(1)

    for iev in range(xi, offsets.shape[0]-1, xstride):
        ret[iev] = 0.0
        start = offsets[iev]
        end = offsets[iev + 1]
        for ielem in range(start, end):
            ret[iev] += data[ielem]
```

In [58]:
```
sum3 = cupy.zeros(len(pt.offsets)-1, dtype=cupy.float32)
d1 = cupy.array(pt.content, dtype=cupy.float32)
d2 = cupy.array(pt.offsets, dtype=cupy.uint64)
```

In [60]:
```
%%timeit
sumpt_event_cudakernel[128,1024](d1, d2, sum3)
cuda.synchronize()
```

```
2.19 ms ± 2.69 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Minimal modifications to run on GPU!

Why Numba kernels?

- Originally, was planning to `s/numpy/cupy/g` in `awkward`

- However, for minimal dependencies and maximal generality, `awkward` relies on complex numpy functions (`reduce`, `reduceat`), some of these are not implemented in cupy

- Rather than commit to developing a very generic implementation upstream in cupy immediately, **identify strengths and weaknesses of GPUs for a complete prototype analysis**

- Hence, end up implementing necessary kernels directly by hand in Numba, not so many after all

- In addition, vivid memories of MATLAB constraints... freedom for straightforward data manipulation is a plus (see also: PyTorch vs Tensorflow)

Implicit vs explicit loops

- `awkward` and `coffea` offer tools to write all operations directly on arrays: `pair_idx = charges.argchoose(2)`

- In some cases, it may be more natural (this is subjective) and faster to express an operation as a loop over the array contents via numba: `os_mask = select_opposite_sign(mu_charges)`

- Such "custom kernels" can still be called on arrays in a functional style, **without reverting to a big unfactorizable event loop** (current typical analysis code)

- End goal is to arrive at a **simple, concise and fast primitives for most analysis ops**, but not restrict physicists' freedom: https://github.com/scikit-hep/awkward-array/issues/107

Necessary kernels

Beloved 1D lookups
- **`get_bin_contents`**
- **`searchsorted_kernel`**
- **`fill_histogram`**
- **`fill_histogram_several`**: batched data

Physics-specific kernels
- **`select_opposite_sign`**
- **`mask_deltar_first`**
- **`comb_3_invmass_closest`**
- **`max_val_comb`**
- **`...`**

Simple array reductions
- **`sum_in_offsets`**
- **`max_in_offsets`**
- **`min_in_offsets`**
- **`get_in_offsets`**
- **`set_in_offsets`**
- **`prod_in_offsets`**
- **`...`**

In total, a few hundred lines of stable code total for CPU and GPU backends.

Straightforward to unit-test and reuse the kernels.

hepaccelerate

- CPU and GPU kernels available as a python package - meant for communication and discussion
- Used as a common stable backend for a few ongoing analyses

README.md

build passing | pipeline passed | DOI 10.5281/zenodo.3245494

hepaccelerate

Accelerated array analysis on flat ROOT data. Process 1 billion events to histograms in minutes on a single workstation. Weighted histograms, jet-lepton deltaR matching and more! Works on both the CPU and GPU!

https://github.com/hepaccelerate/hepaccelerate

```
@cuda.jit
def select_opposite_sign_muons_cudakernel(
  muon_charges_content, muon_charges_offsets,
  content_mask_in, content_mask_out):

  xi = cuda.grid(1)
  xstride = cuda.gridsize(1)

  for iev in range(xi, muon_charges_offsets.shape[0]-1, xstride):
    start = np.uint64(muon_charges_offsets[iev])
    end = np.uint64(muon_charges_offsets[iev + 1])

    ch1 = np.int32(0)
    idx1 = np.uint64(0)
    ch2 = np.int32(0)
    idx2 = np.uint64(0)

    for imuon in range(start, end):
      if not content_mask_in[imuon]:
        continue

      if idx1 == 0 and idx2 == 0:
        ch1 = muon_charges_content[imuon]
        idx1 = imuon
        continue
      else:
        ch2 = muon_charges_content[imuon]
        if (ch2 != ch1):
          idx2 = imuon
          content_mask_out[idx1] = 1
          content_mask_out[idx2] = 1
          break
```

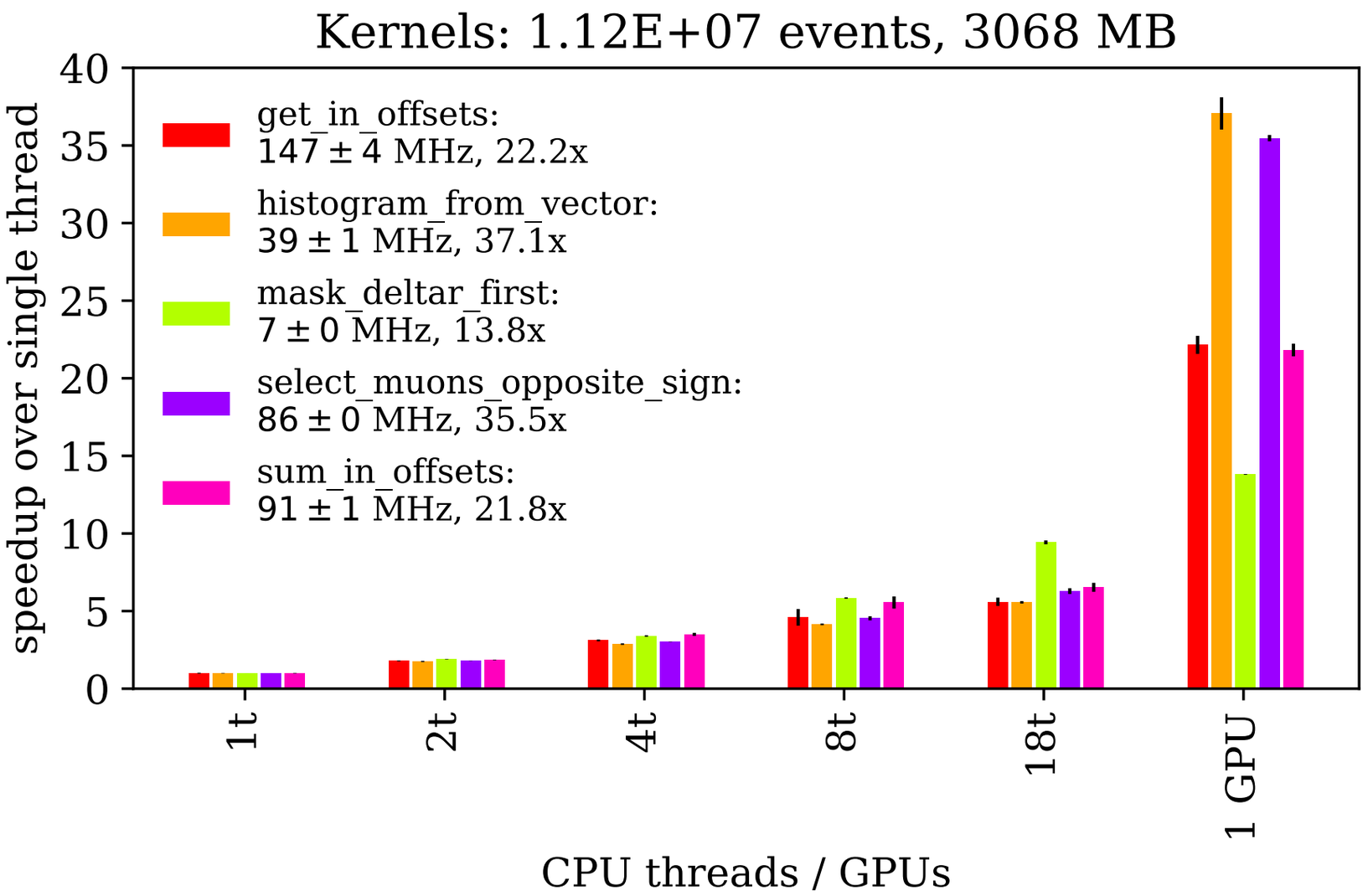
loop over events

loop over muons in event

mask muons of opposite charge

Not by nature GPU-friendly code, but it's a starting point!

Kernel benchmarks



- 11M preloaded events, scaling with respect to single-core baseline.
- Kernels run at **10-100 MHz / thread**, sublinear with multiple threads
- speedup **x10-40 on the GPU** wrt. a single thread

Multithreading helps, but there is still work to be done to take full advantage!

Open data example analysis

Goal: shareable analysis code & dataset for benchmarking, optimization studies
Work in progress on a baseline prototype **top quark pair analysis** based on CMS Open Data, inspired by ongoing Run 2 analyses...

- **lepton selection**: pT leading and subleading, eta, ID
- **jet selection**: pt, eta, ID & PU ID, remove jets dR-matched to leptons
- **event selection**: trigger, number of good jets & leptons
- **high-level variables**: trijet with invariant mass closest to top quark mass
- **event weights**: lepton corrections based on lookups
- **jet systematics**: on-the-fly jet pT corrections based on lookups
  - nJEC=20x2, results in ~x40 complexity increase in the analysis
- **DNN evaluation**: typical analyses need to rerun DNN on variated inputs derived from variated jets
- **many variated histograms**: all DNN inputs & outputs with all variations

Analysis performance

Benchmark analysis in single-threaded, multi-threaded and GPU-mode. We can always scale horizontally, here we investigate vertical scaling (within job)

job type	partial systematics	full systematics
processing speed (kHz)		
1 thread	50	1.4
4 threads	119	4.0
GPU	440	20
walltime to process a billion events (hours)		
1 thread	5.5	200
4 threads	2.3	70
GPU	0.6	13

What's best to use? Depends on the price and availability! GPU-capability can reduce the number of jobs needed by ~10x

Discussion

- Key to fast codes: do as little as possible. Data decompression is not necessarily needed to rederive physics results. Possible improvement - **decompress efficiently on GPU**!

- For a prototype, **simplicity > ultimate scalability**, horizontal scaling a separate topic in itself, addressed by e.g. the coffea project

- Main bottlenecks: kernel call overhead, **1d lookups based on searchsorted** (JEC variations, histograms)

- Need to **bring data as close as possible to the processor/GPU** to take full advantage of their speed: ideally local disk, local RAM, ideal for **analysis facilities**

- Effort needed to port complicated physics codes (e.g. likelihoods) to array-based approach!

Potential limitations

- How to **avoid duplicated kernel code** between backends? It seems like analysis logic dominates the development effort.

- Need to carefully propagate and keep track of masks, a **pythonic jagged dataframe abstraction** might be helpful

- For a real analysis with many variations, **allocation of temporary arrays during computation** can become an issue

- ML codes currently not very friendly about sharing the underlying compute resource

- **Avoid reimplementing the wheel**: find common path with e.g. coffea, RDataFrame, NAIL, allow direct use of existing codes and other familiar tools on a GPU backend

Reproducibility

- In practice, we can run a complete end-to-end analysis on a ~200 CPU threads in an hour using standard condor queues, reproduce multiple times per day

- Less reliance on external servers or complicated workflows: easier to reproduce complete analyses in a short time
  - Integration with continous delivery straightforward

- You should be able to try this on any CentOS7 machine, a singularity recipe with cupy support is provided on github

- Derived open data datasets for the benchmark analysis available: link

Generalizability

- most **SM analyses use the same objects and corrections**, a single coherent code should be able to serve many needs

- specialized jet reclustering or vertexing, not feasible from reduced event formats, likely need "Big Data" approaches

- External HEP-only thread-unsafe libraries not easily callable from the vectorized python ecosystem

- kinematic fits common in b-physics analyses

- TMVA BDT-s for syncing, but CMS GBRForest is multithreading-friendly

- MEM / madgraph: interesting work by V. Hirschi et al on exporting TF computational graphs, link

Future directions

- **Data formats:** fast decompression of ROOT data on GPUs
- **Hybrid usage**: use CPU and GPU simultaneously if available
- **Decouple description and implementation**: e.g. NAIL
- **Caching of intermediates**: avoid recomputing if DAG predicts output will not change
- **Kernel optimization:** threading optimization, kernel fusion, improvement of bottlenecks (e.g. searchsorted), export computational graph to e.g. tensorflow
- **Analysis facility**: integrate with e.g. coffea to easily spin up on various backends with accelerator awareness

Final words

- This was to solve a scoped problem for certain analyses, **but we should aim to develop common tools**

- Analysis codes have significant leeway for **testing new ideas and directly benefit physics results**

- Largely, I'm expanding on ideas put in place by others using **existing tools to solve a specific task** - do Higgs & top studies fast and painlessly 🙂

- Preliminary prototype library for kernels, but will be refactored as ideas and needs develop: https://github.com/hepaccelerate/hepaccelerate

- CMS-specific vectorized code, including Rochester corrections, lepton scale factors, multithreaded GBRForest + Hmumu analysis: https://github.com/hepaccelerate/hepaccelerate-cms (documented in a CMS internal note)

- **Aiming for joint effort on common tools!**

Thank you!

This conference travel was supported by IRIS-HEP and US-DOE!

Part of this work was conducted at iBanks, the AI GPU cluster at Caltech. We acknowledge NVIDIA, SuperMicro and the Kavli Foundation for their support of iBanks.

The open scientific software ecosystem was essential - thank you to the devs and maintainers of numpy, numba, cupy, uproot, awkward, coffea and other tools!

Backup

- https://indico.cern.ch/event/815710/contributions/3404720/attachments/1833376/3003050/arraycomputing.pdf

- https://indico.cern.ch/event/816211/contributions/3407177/attachments/1834757/3005590/26_04_2019_coffeameeting.pdf

- https://indico.cern.ch/event/808455/contributions/3366487/attachments/1821708/2979913/26_03_2019_gtc_summary.pdf

NVidia profiling

Disclaimer

- I'm not offering to solve CERN/HL-LHC computing problems in perpetuity - but rather showing a **practical method for numerical data analysis** that can take advantage of CPU and GPU architectures, directly from python

  - Perhaps free up shared clusters from low-efficiency analysis jobs?

- I'm **not in any way inventing anything new** - it's a question of using ideas that already exist in the world (including at CERN) and adapting them to our use case

- I'm not suggesting CERN to completely adopt a new paradigm or to throw out all Tier2 resources and replace them with something else: I'm suggesting methods how to do **analyses on local data, on local machines with a fast turnaround-time, with the resources that you have, without needing extensive scaleout infrastructure**

- **Does not rely on NanoAOD or other particular *AOD format** - but it can make use of it

- *How does this relate to the awkward-array project?* We use the jagged structure provided by the awkward arrays, but implement common HEP functions such as deltaR matching as loops or 'kernels' running directly over the array contents, taking into account the event structure. We make these loops fast with Numba, but allow you to debug them by going back to standard python when disabling the compilation.
- *Why don't you use the array operations (`JaggedArray.sum`, `argcross` etc) implemented in awkward-array?* They are great! However, in order to easily use the same code on either the CPU or GPU, we chose to implement the most common operations explicitly, rather than relying on numpy/cupy to do it internally. This also seems to be faster, at the moment.
- *What if I don't have access to a GPU?* You should still be able to see event processing speeds in the hundreds of kHz to a few MHz for common analysis tasks.
- *How do I plot my histograms that are saved in the JSON?* Load the JSON contents and use the `edges` (left bin edges, plus last rightmost edge), `contents` (weighted bin contents) and `contents_w2` (bin contents with squared weights, useful for error calculation) to access the data directly.
- *I'm a GPU programming expert, and I worry your CUDA kernels are not optimized. Can you comment?* Good question! At the moment, they are indeed not very optimized, as we do a lot of control flow (`if` statements) in them. However, the GPU analysis is still about 2x faster than a pure CPU analysis, as the CPU is more free to work on loading the data, and this gap is expected to increase as the analysis becomes more complicated (more systematics, more templates). At the moment, we see pure GPU processing speeds of about 8-10 MHz for in-memory data, and data loading from cache at about 4-6 MHz. Have a look at the nvidia profiler results nvprof1, nvprof2 to see what's going on under the hood. Please give us a hand to make it even better!
- *What about running this code on multiple machines?* You can do that, currently just using usual batch tools, but we are looking at other ways (dask, joblib, spark) to distribute the analysis across multiple machines.