Experience with vectorized Higgs analysis on GPUs

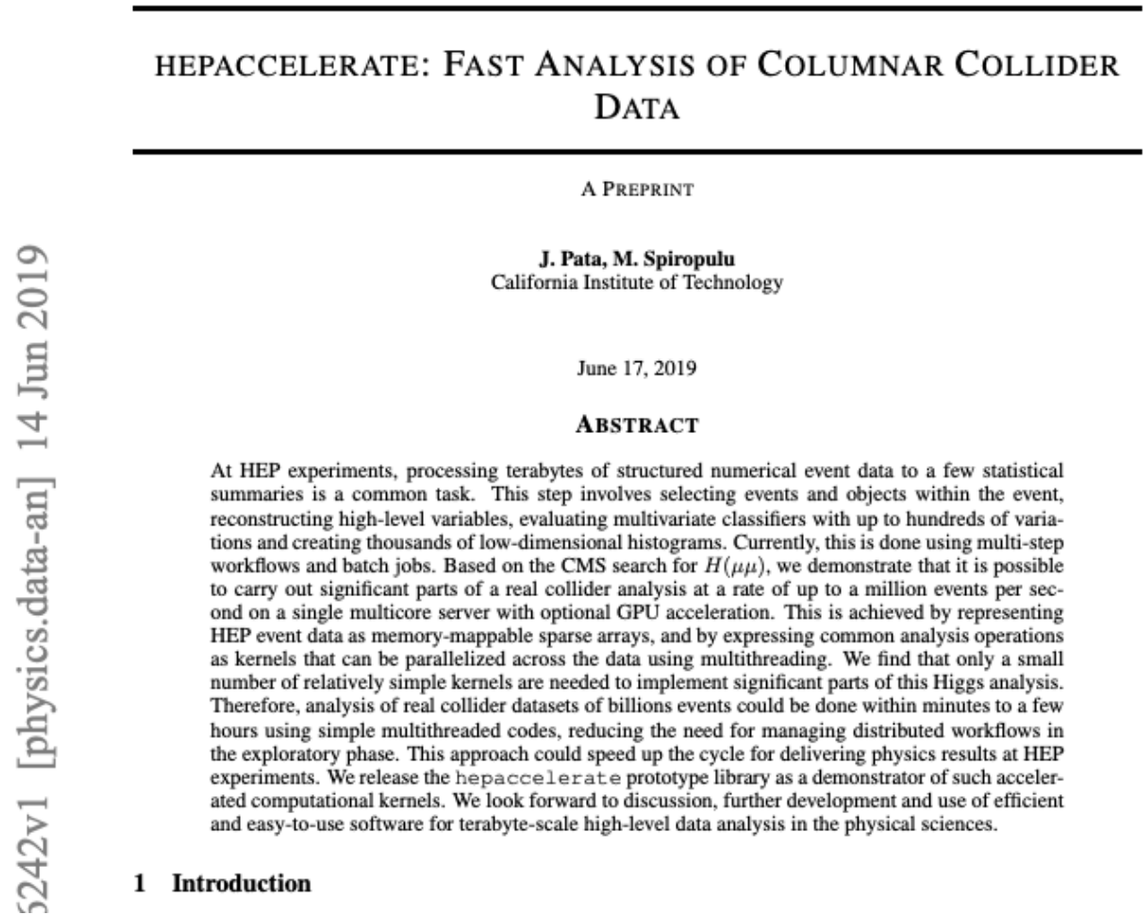
Joosep Pata, Maria Spiropulu

July 22, 2019

HEP SF WG

Overview

- Back-of-the-envelope feasibility study for MHz data analysis
- Numba as a way to write down and test array kernels fast
- HiggsMuMu-VBFchannel proto analysis: implementation and performance
- Good and bad points of GPUs for analysis



5242v1 [physics.data-an] 14 Jun 2019

HEPACCELERATE: FAST ANALYSIS OF COLUMNAR COLLIDER DATA

A PREPRINT

J. Pata, M. Spiropulu
California Institute of Technology

June 17, 2019

ABSTRACT

At HEP experiments, processing terabytes of structured numerical event data to a few statistical summaries is a common task. This step involves selecting events and objects within the event, reconstructing high-level variables, evaluating multivariate classifiers with up to hundreds of variations and creating thousands of low-dimensional histograms. Currently, this is done using multi-step workflows and batch jobs. Based on the CMS search for $H(\mu\mu)$, we demonstrate that it is possible to carry out significant parts of a real collider analysis at a rate of up to 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, and by expressing common analysis operations as kernels that can be parallelized across the data using multithreading. We find that only a small number of relatively simple kernels are needed to implement significant parts of this Higgs analysis. Therefore, analysis of real collider datasets of billions events could be done within minutes to a few hours using simple multithreaded codes, reducing the need for managing distributed workflows in the exploratory phase. This approach could speed up the cycle for delivering physics results at HEP experiments. We release the `hepaccelerate` prototype library as a demonstrator of such accelerated computational kernels. We look forward to discussion, further development and use of efficient and easy-to-use software for terabyte-scale high-level data analysis in the physical sciences.

1 Introduction

Preprint with technical details:
https://arxiv.org/abs/1906.06242

Data flow parameters

- For Hmm Run 2: 800M skimmed events, ~70 branches: ~640GB of uncompressed skimmed binary data (4.8B ev, 4.8TB NanoAOD)

- Supposing a modest SSD data read speed of 300MB/s, ~2000 seconds, ~400 kHz (new NVMEs up to 10x faster)

- Quick tests show ~100kHz ... 2 MHz of pure event processing rate easily achievable on data that is in memory on a multicore/GPU machine

- Both SSD read and data processing are of a similar order of magnitude and can be tightly coupled in a threaded loop

- **A single multicore workstation machine could process local data at a sustained rate of 500 kHz, which can be further accelerated with a GPU for computationally heavy tasks**

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 Higgs analysis

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

- In addition, still remember my days of being constrained by MATLAB and needing everything to be expressable in just a few array ops… want to have freedom for explicit data manipulation

Implicit vs explicit loops

- awkward-array 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

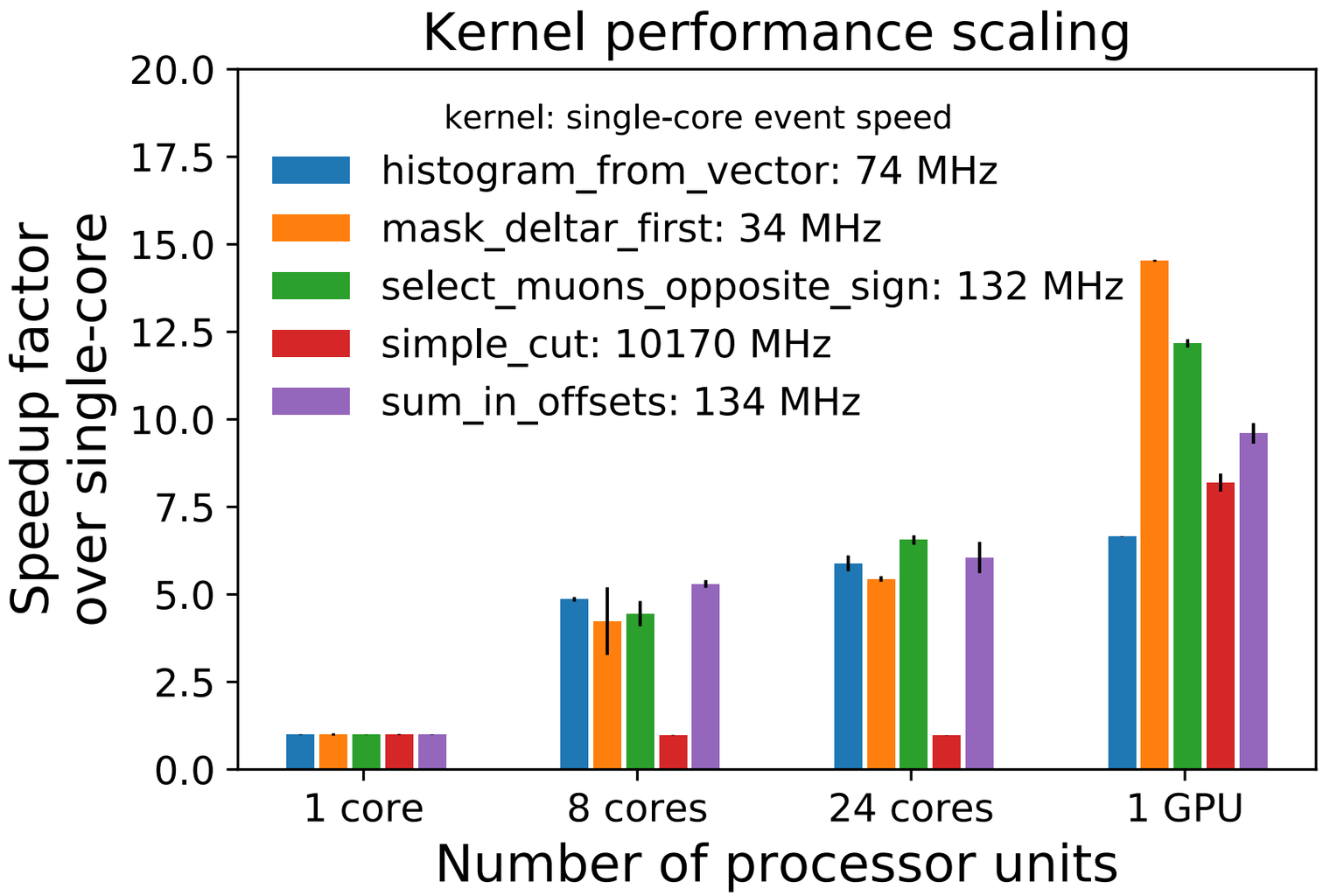
- **sum_in_offsets**(content, offsets, mask_rows, mask_content, out)
- **max_in_offsets**(content, offsets, mask_rows, mask_content, out)
- **min_in_offsets**(content, offsets, mask_rows, mask_content, out)
- **get_in_offsets**(content, offsets, indices, mask_rows, mask_content, out)
- **set_in_offsets**(content, offsets, indices, target, mask_rows, mask_content)

- **get_bin_contents**(values, edges, contents, out)
- **searchsorted_kernel**(vals, arr, inds_out)
- **fill_histogram**(data, weights, bins, out_w, out_w2)

- **select_opposite_sign**(charges_content, charges_offsets, content_mask_in, content_mask_out)
- **mask_deltar_first**(etas1, phis1, mask1, offsets1, etas2, phis2, mask2, offsets2, dr2, mask_out)

~ 250 lines of code total for GPU backend
~210 lines for multithreaded CPU backend

Kernel benchmarks



150k preloaded events, scaling with respect to the single-core baseline.

For example, the kernel for computing ΔR masking runs at a speed of 34 MHz on a single-core of the CPU and is sped up by about a factor x5 (x15) by multithreading using the CPU (GPU).

multithreaded CPU

```
@numba.njit(parallel=True, fastmath=True)
def sum_in_offsets_kernel(
  content, offsets,
  mask_rows, mask_content, out):

  for iev in numba.prange(offsets.shape[0]-1):
    if not mask_rows[iev]:
      continue
    start = offsets[iev]
    end = offsets[iev + 1]
    for ielem in range(start, end):
      if mask_content[ielem]:
        out[iev] += content[ielem]
```

masked loop over rows/events

masked loop over cols/objects

GPU/CUDA

```
@cuda.jit
def sum_in_offsets_cudakernel(
  content, offsets,
  mask_rows, mask_content, out):

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

  for iev in range(xi, offsets.shape[0]-1, xstride):
    if not mask_rows[iev]:
      continue

    start = offsets[iev]
    end = offsets[iev + 1]
    for ielem in range(start, end):
      if mask_content[ielem]:
        out[iev] += content[ielem]
```

```
@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 with threads

loop over muons in event

mask muons of opposite charge

```
def get_selected_muons(
    muons, trigobj, mask_events,
    mu_pt_cut_leading, mu_pt_cut_subleading,
    mu_aeta_cut, mu_iso_cut, muon_id_type,
    muon_trig_match_dr):

    muons_passing_os = ha.select_muons_opposite_sign(
        muons, muons_passing_id & passes_subleading_pt)

    events_passes_os = ha.sum_in_offsets(
        muons, muons_passing_os, mask_events,
        muons.masks["all"], NUMPY_LIB.int32) == 2
```

Create mask of chosen muons

```
....

ret_mu = get_selected_muons(
    muons, trigobj, mask_events,
    parameters["muon_pt_leading"],
    parameters["muon_pt"], parameters["muon_eta"],
    parameters["muon_iso"], parameters["muon_id"],
    parameters["muon_trigger_match_dr"]
    )
```

Analysis will be functions operating on arrays, producing the desired results: histograms, ntuples.

Hmm analysis

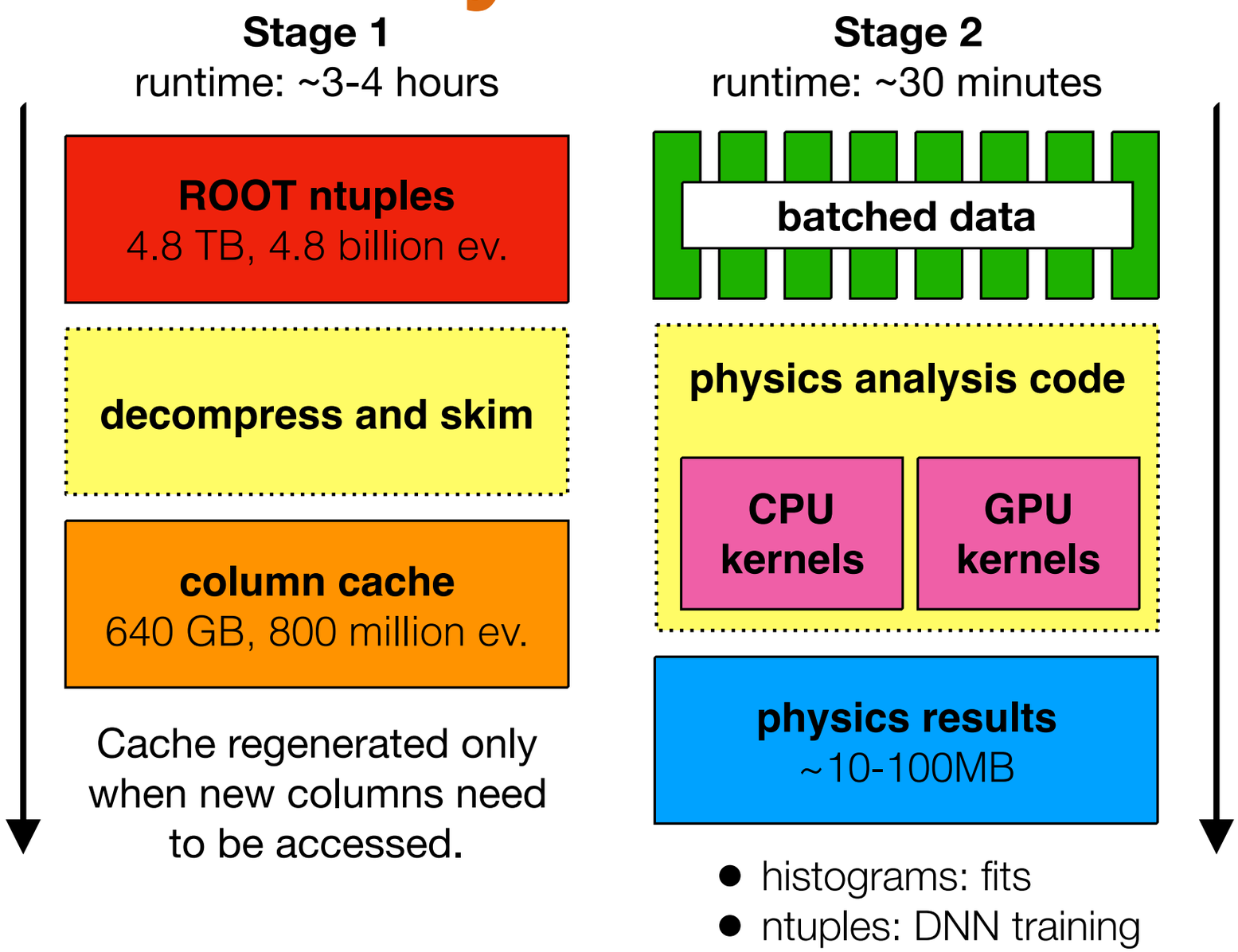
This proto-analysis implements the following:

- ☑ muon selection: pT leading and subleading, eta, ID, isolation, opposite charge, matching to trigger objects
- ☑ jet selection: pt, eta, ID & PU ID, remove jets dR-matched to leptons
- ☑ jet-lepton gen-level cleaning, gen-level
- ☑ event selection: MET filters, trigger, primary vertex quality cuts, two opposite sign muons
- ☑ high-level variables: dimuon invariant mass
- ☑ PU and gen weight computation
- ☑ on the fly luminosity calculation, golden JSON lumi filtering (via coffea/FNAL tools)
- ☑ weighted histograms of muon, jet and event variables
- ☑ muon momentum Rochester corrections: CMS code + OpenMP
- ☑ lepton scale factors: simple histogram lookup: CMS code + OpenMP
- ☑ JES/JER/MET correction reappliaction: via coffea + cupy
- ☑ evaluation of signal-to-bkg discriminators (tensorflow, GBRForest)

Not yet implemented:

- ☐ on-the-fly kinematic fits
- ☐ ME discriminator: don't see major issues with standalone madgraph amplitudes

Analysis flow



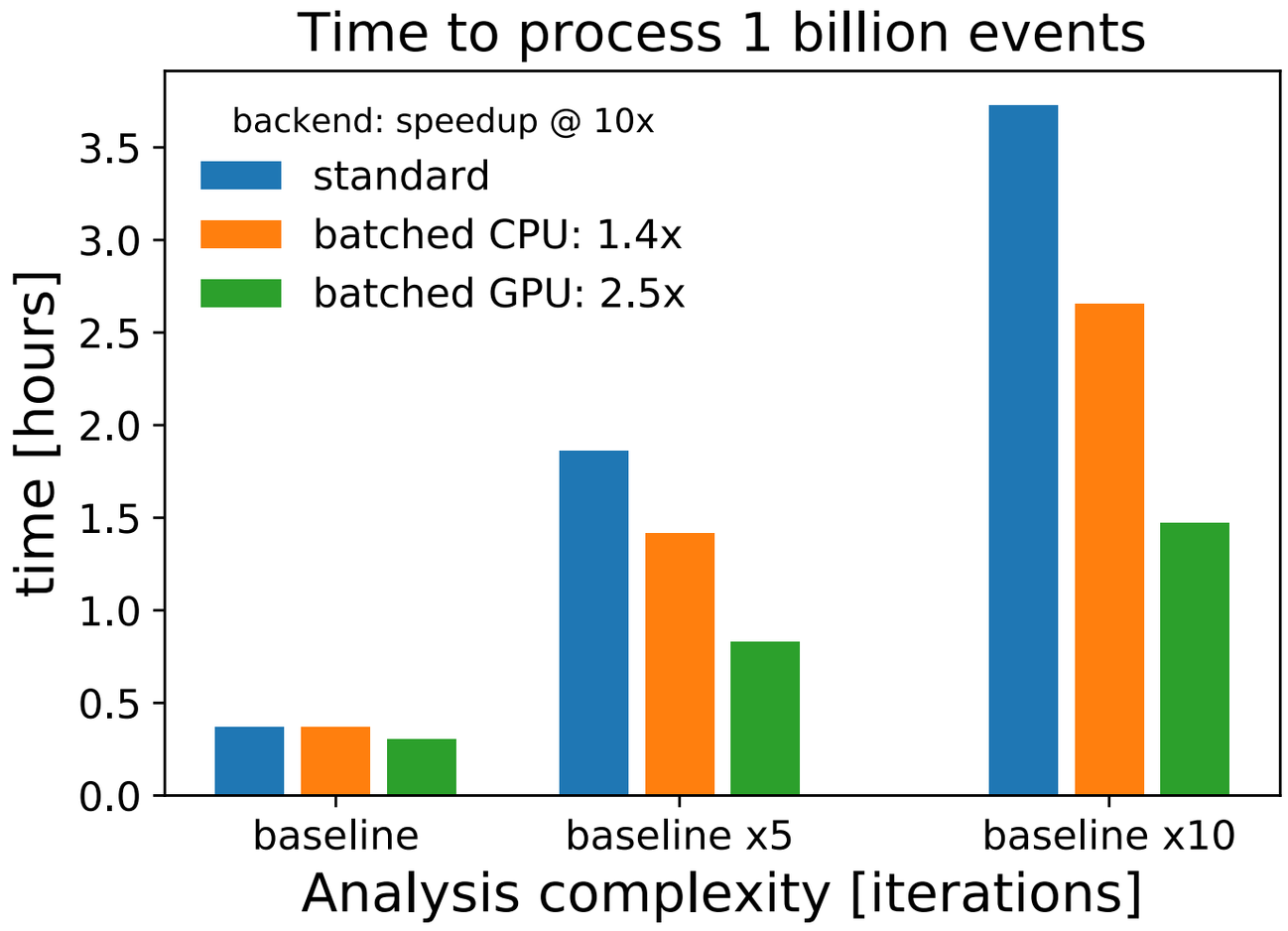
All code run on a single machine from a single multithreaded python job using ThreadPoolExecutor. This script can also be wrapped in a batch job and run either using one or multiple threads and optionally a GPU.

Performance

- Extrapolated performance based on processing 1B unskimmed events: ~0.5h on CPU or GPU

- This analysis is computationally simple: emulate complexity by rerunning the same analysis with variated parameters in memory
  - CPU time ~3.5h, GPU time ~1.2h

- Caveat: these benchmarks are already a few months old - analysis has matured, but conclusions stay roughly the same



Can iterate on complete analysis with systematics with **full Run2 dataset in a matter of a few hours on one server!**

Performance discussion

- To get MHz speeds, I skip the ROOT decompression as it is not strictly needed at every iteration

- My main goal is to implement a complete analysis and get feature-complete fast, rather than have a perfect framework

- Seek to find limitations of the GPU-analysis approach

- This said, in my experience, the slow part is often producing hundreds-thousands of variated histograms, and various IO-dominated ntuplization steps (YMMV)

When does it work?

- If we don't want to be constrained by decompression speed, **need to uncompress branches**: shouldn't use too many branches

- Need to **bring data as close as possible to the processor/GPU** to take full advantage of their speed: ideally local disk, local RAM, less ideally access T2 storage / EOS, xrootd probably does not make sense

- When data processing is relatively simple, no reliance on e.g. complicated RooStats/CMSSW modules

- When you don't need data from earlier stages that would need remote access

- Not everything has to be on the GPU! For example, LumiMask application, lepton SF, Rochester corrections are run on the CPU, transferring only the needed arrays

Potential limitations

- Several kernels are very similar, some duplicated code (but only a few hundred lines of kernel code in total right now - most is in defining the analysis flow)

- Need to carefully propagate and keep track of event/object masks, possibly a dataframe-type abstraction would be more user-friendly

- For a real analysis with many variations, allocation of temporary arrays during computation can become costly (e.g. 50 JEC variations)

- Memory management in python can get confusing, multi-hour jobs suffer currently suffer hard-to-trace leaks and excessive context switching

- Possibly RDataFrame will mature fast enough and allow direct use of ROOT fitting functions and other familiar tools on a GPU backend: I will be a happy user in this case

Reproducibility

- You should be able to try this on any CentOS7 machine and uproot-accessible to NanoAOD LFN-s, best if stored on local SSD

- ETH colleagues have already given a try and seems to be working for them for the ttH(bb) analysis

- **Can get started on CPU-only, GPU acceleration can be turned on as an option**

- **Main goal is to be able to run most of the analysis without relying on external servers**

- I also provide a singularity image, but lxplus does not support suid singularity, so have not gotten it to work on lxplus yet

Generalizability

- What I have experienced in the past: top mass, single-top, ttH, Hmumu, btag optimization and calibration
    - All these SM analyses use mostly the same objects (jets, leptons, MET), don't see any reason why cannot implement all
- More exotic things that probably don't make sense in this approach:
    - specialized jet reclustering or vertexing, not feasible from reduced event formats, likely need "Big Data" approaches
    - External HEP-only thread-unsafe libraries might be tricky to use: kinematic fits, TMVA BDT-s (actually it's fine using GBRForest), MEM/madgraph

What next?

- This is mostly an experiment to find the limitations: most expensive seem to be interpolations for JEC variations and histogram filling

- **Simplicity > performance**: we don't aim to max out the hardware at the cost of usability, this could be investigated separately

    - e.g. could further improve threading and data loading and streaming, GPU kernel efficiency

- **Decouple analysis description and execution**: compile for any backend, e.g. https://github.com/arizzi/nail

- Need common, portable, simple data structures for vectorized python codes: JaggedStruct, Lorentz stuff, Histograms

- Long term, aim to contribute accelerated kernels upstream to array analysis libraries (awkward, coffea, RDataFrame, cupy...)

- Ultra long term: jagged functionality in cuDF, be a code user rather than developer

Final words

- I'm solving a scoped problem I faced during my last years in HEP, for which no out of the box tool worked so far, **I'm hoping this will change in the future**

- However, you may have had a different experience and **you should use the tools that are suitable for you**

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

- **Not aiming to compete on framework development**, it's better to develop a small set of common tools

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

- CMS-specific vectorized code, including Rochester corrections, lepton SF, threaded GBRForest + Hmm analysis: https://github.com/jpata/hepaccelerate-cms

Bonus slides

Throughput

- Cache branches from ROOT files on fast local disk
  - Investigated blosc/lzma of arrays, but speed/size tradeoff was not favorable
  - Multi-user situation? Likely want to share uncompressed caches or possibly do ROOT decompression on GPU
- Besides that, main limitation seems to be SSD → memory data transfer
  - Various DMA approaches possible with engineering effort
- GPU processing bandwidth seems not to be a major issue

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.