```
In [8]: "%.2g" % np.prod(h.shape)
Out[8]: '5.5e+09'

In [9]: h.view(flow=True).nbytes / (1 << 30)
Out[9]: 82.93021202087402
```

```
In [6]: h.shape
Out[6]: (253, 360, 152, 400)

In [7]: h.axes.name
Out[7]: ('dataset', 'category', 'systematic', 'dnn_score_max')
```

~~offloading @ coffea~~

Huge Dense Histograms

Benjamin Fischer

PyHEP.dev
August 28th 2024

SPONSORED BY THE
Federal Ministry of Education and Research

IDIUM

RWTH AACHEN UNIVERSITY

What? Where? Why?

What: Huge Dense Histograms

- ~O(10^9) bins
- >50% filled
- >10GB in memory

```
In [8]: "%.2g" % np.prod(h.shape)
Out[8]: '5.5e+09'

In [9]: h.view(flow=True).nbytes / (1 << 30)
Out[9]: 82.93021202087402
```

Where: (some) HEP analyses - e.g. HHbb WW(DL)

- can easily have O(100)
  - datasets
  - categories
  - systematics
  - bins (for variable)
- can become even worse

```
In [6]: h.shape
Out[6]: (253, 360, 152, 400)

In [7]: h.axes.name
Out[7]: ('dataset', 'category', 'systematic', 'dnn_score_max')
```

Why: trimming excess is cumbersome

- strong driver
  - data-driven modeling methods: ABCD-methods ➜ easily 4x
- unneeded/neglectable stuff not always (exactly) known ahead of time

Solution? HDF5 (& co.)

histogram data generally view()-able as numpy arrays

➜ why not store this in HDF5 (via h5py) ?

- can grow underlying array cheaply
  - no need to copy everything to new allocation! - a major bottleneck
- can compress data transparently
  - very helpful for sparse/empty parts
- can be done in-memory
  - python’s io.BytesIO are valid “files” for h5py

Filling it

histogram filling

- may access any data of it
- but, often filled one *categorical* axes entry at time
  - i.e. only one dataset/systematic at a time
  - generally coincides with *growable* axes
- can "offload" into temporary, small, in-memory histogram

Step-by-step: filling events into a h5py backed histogram

1. create "empty" temporary histogram (i.e. from hist)
   - growables axes are trimmed i.e. "empty"
2. fill temporary using normal fill() call e.g. via boost-histogram
3. accumulate view() of temporary onto h5py backed array (i.e. Dataset)
   - needs some care with axes entry ordering etc. - not impossible
4. throw away temporary

Other Pros & Cons

Pros:

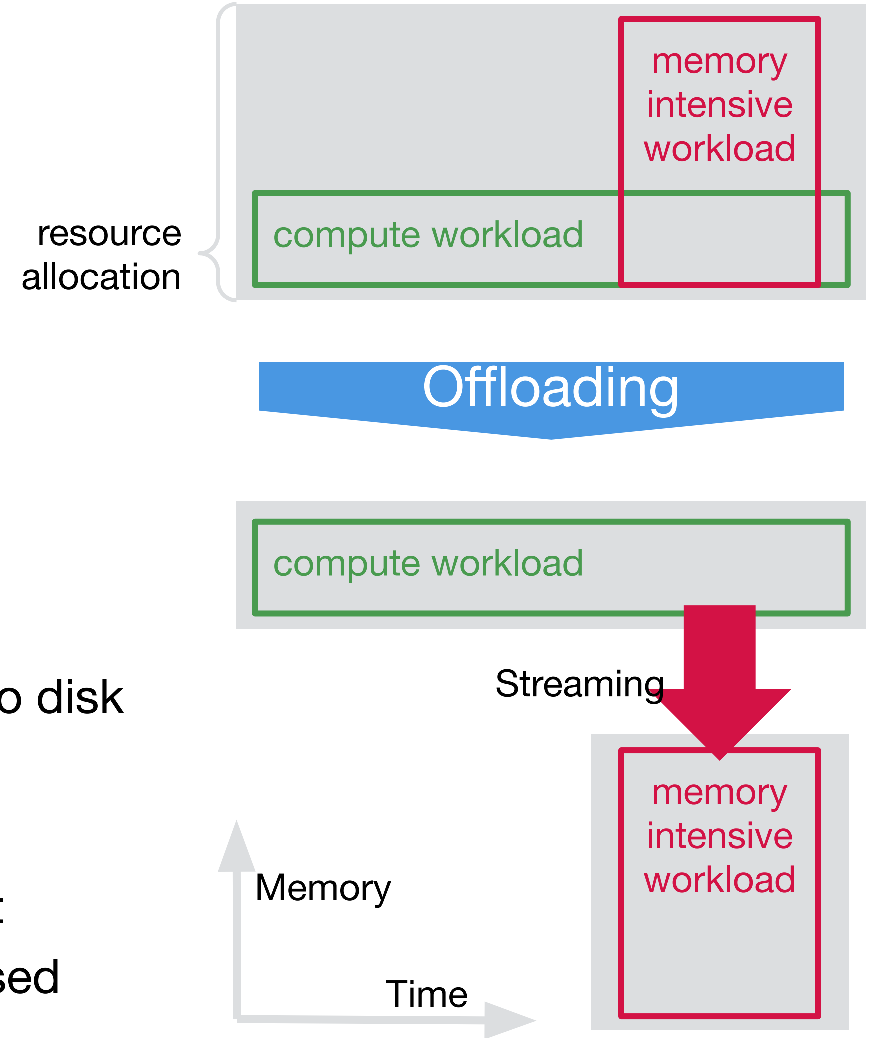
- can have h5py file actually on disk
  - no need to ever keep everything (even compressed) in memory
  - enables absurdly large histograms (>1TB on disk - compressed)
- principle should also work with h5py peers
  - zarr
  - blosc2

Cons:

- non-trivial implementation needed for
  - advanced UHI operations (e.g. ::sum)
  - other (partial) projections - esp. for analyses

HaaaS - Histogram accumulation as a Service

- using:
  - h5py/zarr backed histogram
  - dask distributed Actor
- coffea worker's fill a virtual histogram
  - results are buffered on worker
  - large chunks are sent to central accumulator "service"
- massively reduced memory footprint
  - worker offload large histograms
  - central "server" can directly write to disk
- coffea integration fairy easy
- already used similar implementation
  - each dask worker 3.5GB RAM, but
  - histogram 83/4.5 GB (un)compressed



Backup

Memory Offloading

Benjamin Fischer, August 28th 2024

Antipattern

excessive buffering of intermediate output
➜ footprint of workload varies a lot
➜ poor resource allocation use

Histograms: pathological affected

- esp. when multiple workloads produce same histograms to be summed later

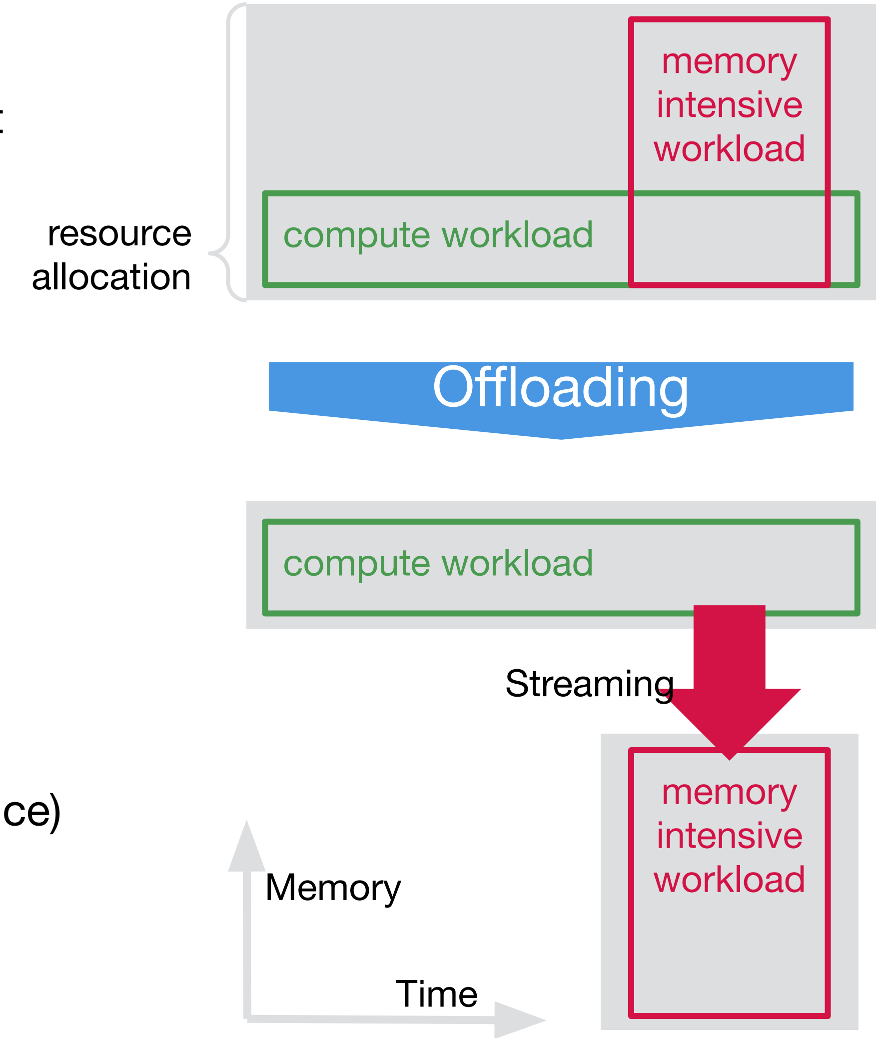
➜ unnecessary copies

Solution: eagerly aggregate outputs

- opportunity for centralization

Implementation

- data streamed (decoupled from MapReduce)
- transparent via custom coffea Aggregator
- communication via Dask (Actors)



Compute Offloading

RWTHAACHEN UNIVERSITY
Benjamin Fischer, August 28th 2024

Idea

- offload onto specialized hardware i.e. GPU
- works well with heavy workloads:
    - esp. DNN evaluation
    - possibly even fits/ME-calc.

Advantages

- batching possible/needed
    - favors columnar data processing
    - automatic batching possible
- enables parallel/async processing
- also has Memory Offloading benefits

Implementation

- using Tensorflow Modelserver
- single low-end GPU sufficient
    - 10^9 events/hour