PRINCETON UNIVERSITY

Demo Day
-
rust-histogram

Peter Fackeldey

10/28/24

About me

- I’m a postdoctoral research associate with Princeton University
- Working on Analysis Systems in IRIS-HEP
- Started ~2 months ago, based in Princeton
- Previously:
  - PhD at RWTH Aachen University
  - Scientific advisor for the ErUM-Data-Hub
  - HH→bbWW in CMS (di-leptonic channel & statistics contact)

rust-histogram

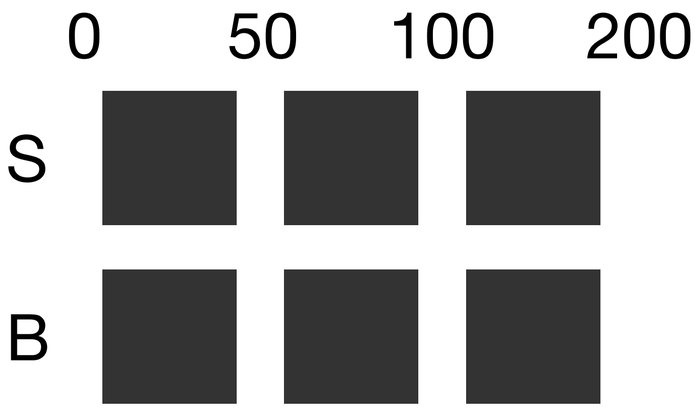
- Tried a Rust based implementation of boost-histogram
  - Invested 2 days (+ a Sunday evening)
  - Repo: https://github.com/pfackeldey/rust-histogram
  - Why?
    - Learning Rust
    - Learning boost-histogram (internals)
    - Possibility to add & test *sparsity*

Not to be used in production!!!

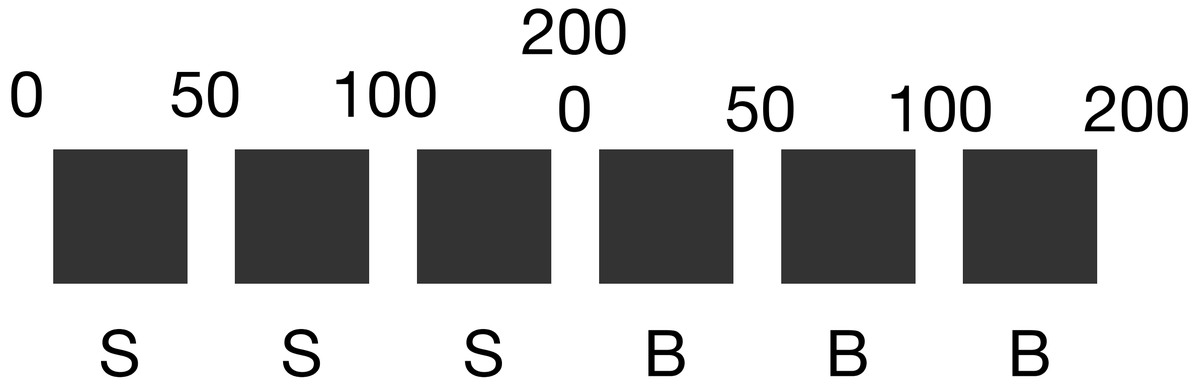
Histogram: internals

Hist with 2 axes: Categorical (“S”, “B”) and Variable

User Interface
(Python/NumPy-View)



Internal memory layout (C++/Rust)

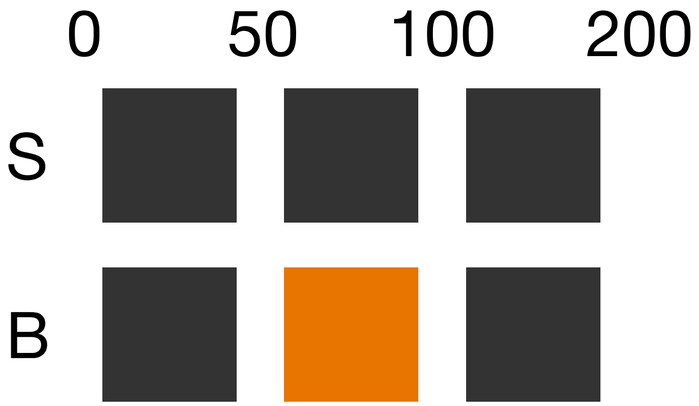


Histogram: memory layout & filling

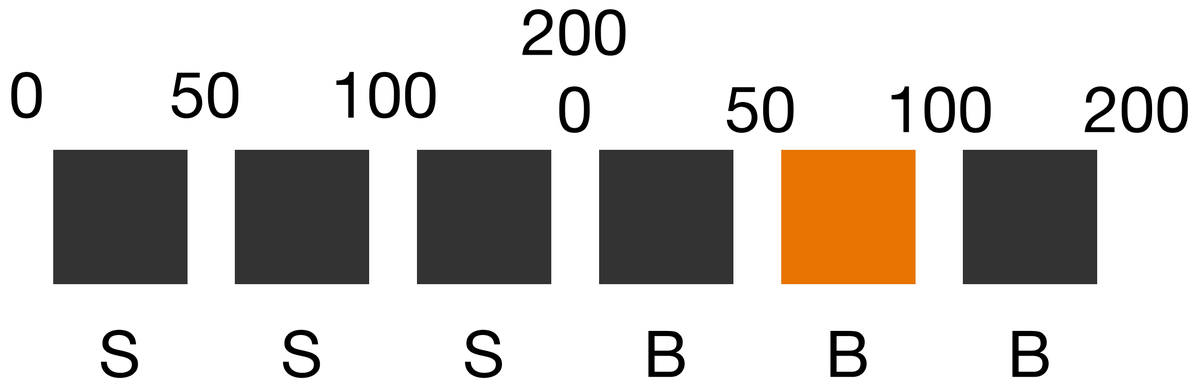
What happens when you call hist.fill(“B”, 60) ?

1. Find index of categorical axis: idx_cat = 1 (lookup: $\mathcal{O}(1)$)
2. Find index of variable axis: idx_var = 1 (binary search: $\mathcal{O}(\log n)$)
3. Calculate strided_index for internal memory layout

User Interface



Internal memory layout



Histogram: memory layout & filling

What happens when you call hist.fill(“B”, 60) ?

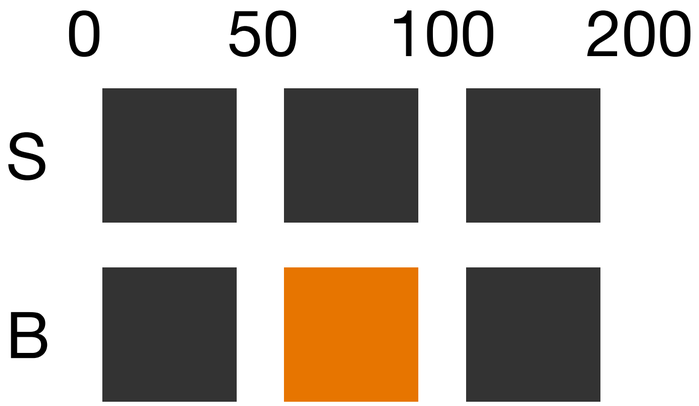
- axes: cat (# 2 bins), var (# 3 bins)
- indices: idx_cat = 1, idx_var = 1

$\rightarrow$ strided_index = $(0 \cdot 2 + 1) \cdot 3 + 1 = \mathbf{4}$

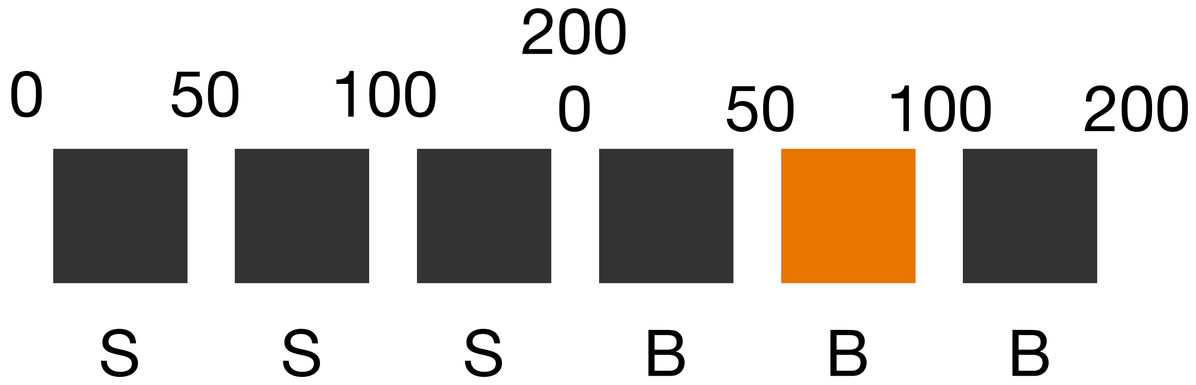
NumPy-style striding

```
let mut strided_index = 0;
for (axis, idx) in axes.iter().zip(indices.iter()) {
    let stride = axis.num_bins(true);
    strided_index = strided_index * stride + idx;
}
Ok(strided_index)
```

User Interface



Internal memory layout



rust-histogram: sparsity

- Often our histograms are quite sparse due to the high-dimensionality:
  We don't need to fill every systematic for every category, etc.
- We can introduce sparsity in 2 ways, instead of holding the dense ND array

  the idea is to only have *filled* bins in memory:

  1. Use 2 Vecs: one for bin content, one for it's index
  2. Use a HashMap: key corresponds to index, value to bin content
- (1) SparseHist
- (2) HashMapHist
- Side note: Rust's std lib HashMap is highly performant with SIMD instructions.
  It's a port of Google's "SwissTable".

HashMapHist: memory layout & filling

What happens when you call hist.fill(“B”, 60) ?

- Everything stays the same as before…

1. Find index of categorical axis: idx_cat = 1 (lookup: $\mathcal{O}(1)$)
2. Find index of variable axis: idx_var = 1 (binary search: $\mathcal{O}(\log n)$)
3. Calculate strided_index for internal memory layout

- …but now we store *only the filled* bins:

Internal memory layout HashMap:
strided_index → bin content

here:

4 → 

HashMapHist: memory layout & filling

What happens when you call hist.fill(“B”, 60) ?

- Everything stays the same as before…

1. Find index of categorical axis: idx_cat = 1 (lookup: $\mathcal{O}(1)$)
2. Find index of variable axis: idx_var = 1 (binary search: $\mathcal{O}(\log n)$)
3. Calculate strided_index for internal memory layout

- …but now we store *only the filled* bins:

Internal memory layout HashMap:
strided_index → bin content

here:

4 → 

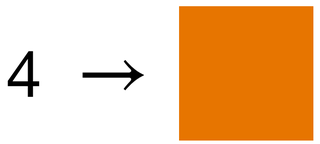
JavaScript arrays are just HashMaps…

```
❯ deno repl
Deno 2.0.3
exit using ctrl+d, ctrl+c, or close()
> const x = [1, 2, 3]
undefined
> x[100] = 4
4
> console.log(x)
[ 1, 2, 3, <97 empty items>, 4 ]
```

HashMapHist: memory consumption

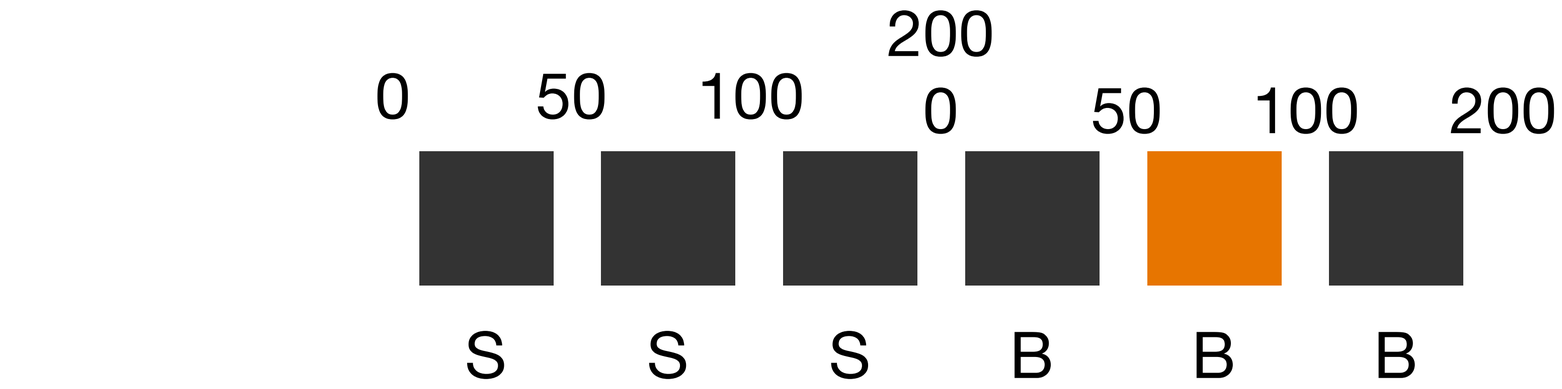
- How much is the memory consumption?

(HashMap) Sparse Hist



$2 \cdot 4$ bytes $= 8$ bytes

(Standard) Dense Hist



$6 \cdot 4$ bytes $= 24$ bytes

→ saved 66% of memory

- Drawback: densely filling a HashMapHist increases memory by 2x (we store bin contents *and* indices)

rust-histogram: features

- StorageTypes:
  - Double: stores *sumw* as f32
  - Int: stores *sumw* as i32
  - Weight: stores *sumw* and *sumw2* as (f32, f32)
- Axis:
  - Uniform: constructs a uniform axis with n bins between start and stop
  - Variable: constructs a variable axis with edges as bin edges
  - Category: constructs a categorical axis with String as bin labels
  - Integer: constructs a categorical axis with i32 as bin labels
- Hist:
  - VecHist: stores the histogram bins in a Vec<StorageType> (dense)
  - SparseHist: stores the *filled* histogram contents (StorageType) and indices (usize) in a Vec respectively
  - HashMapHist: stores the *filled* histogram indices and contents in a HashMap<usize, StorageType>

rust-histogram: features

- StorageTypes:
  - Double: stores *sumw* as f32
  - Int: stores *sumw* as i32
  - Weight: stores *sumw* and *sumw2* as (f32, f32)
- Axis:
  - Uniform: constructs a uniform axis with n bins between start and stop
  - Variable: constructs a variable axis with edges as bin edges
  - Category: constructs a categorical axis with String as bin labels
  - Integer: constructs a categorical axis with i32 as bin labels
- Hist:
  - VecHist: stores the histogram bins in a Vec<StorageType> (dense)
  - SparseHist: stores the *filled* histogram contents (StorageType) and indices (usize) in a Vec respectively
  - HashMapHist: stores the *filled* histogram indices and contents in a HashMap<usize, StorageType>

Missing bits

- Growable axes
- Multi-threading
- Axis transformations
- Circular axis

rust-histogram: example

```rust
use hist::hist::Histogram;
use hist_axes::axis::Axis;
use hist_axes::uniform::Uniform;
use hist_storages::{Storage, StorageType};

let axis1 = Uniform::new(10, 0.0, 10.0).unwrap();
let axis2 = Uniform::new(10, 0.0, 10.0).unwrap();

let axes = vec![
    Box::new(axis1.clone()) as Box<dyn Axis>,
    Box::new(axis2.clone()) as Box<dyn Axis>,
];
let mut hist = super::HashMapHist::new(axes, StorageType::Double);

let where2fill = vec![axis1.index(0.5), axis2.index(0.5)];
hist.fill(&where2fill, 1.0).unwrap();
```

Imports

Axes construction

Hist construction

.fill()

14 rust-histogram: performance

- Followed setup of: https://iscinumpy.gitlab.io/post/histogram-speeds-in-python/
- Fill 2D hist with 1M data points
- Results:
  - boost-histogram: 4.04ms
  - rust-histogram (VecHist): 22.60ms
  - rust-histogram (HashMapHist): 23.70ms
  - (rust-histogram (SparseHist): 1.29s)

Henry's benchmark
(different MBP…)

Example	KNL	MBP	X24
Physt	1.21 s	293 ms	246 ms
NumPy: histogram2d	456 ms	114 ms	88.3 ms
NumPy: add.at	247 ms	62.7 ms	49.7 ms
NumPy: bincount	81.7 ms	23.3 ms	20.3 ms
fast-histogram	53.7 ms	10.4 ms	7.31 ms
fast-hist threaded 0.5	(6) 62.5 ms	9.78 ms	(6) 15.4 ms
fast-hist threaded (m)	62.3 ms	4.89 ms	3.71 ms
Numba	41.8 ms	10.2 ms	9.73 ms
Numba threaded	(6) 49.2 ms	4.23 ms	(6) 4.12 ms
Cython	112 ms	12.2 ms	11.2 ms
Cython threaded	(6) 128 ms	5.68 ms	(8) 4.89 ms
pybind11 sequential	93.9 ms	9.20 ms	17.8 ms
pybind11 OpenMP atomic	4.06 ms	6.87 ms	1.91 ms
pybind11 C++11 atomic	(32) 10.7 ms	7.08 ms	(48) 2.65 ms
pybind11 C++11 merge	(32) 23.0 ms	6.03 ms	(48) 4.79 ms
pybind11 OpenMP merge	8.74 ms	5.04 ms	1.79 ms

$\mathcal{O}(1 - 100)$ ms

No optimizations attempted,
also I probably did some bad things

Rust experience

- Very different coding philosophy to what I'm used to
- Pros:
  - Memory safety (doesn't prevent us from bad performance or leaks though!)
  - Tooling (& especially compiler) is an amazing experience
    (I managed to write most of boost-histogram in just 2.5 days without prior knowledge in Rust!)
  - Functional programming
  - Traits (Composition instead of Inheritance), Tagged unions, Errors as values
  - Macros are surprisingly easy to use and very powerful
  - Not really have to deal with pointers
- Cons:
  - I spent 75% thinking about my types and traits, and 25% on the actual implementation (it's *hard* to think Rust as a Python programmer)
  - Axes growth: horrible in Rust!
    This basically mutates everything → very non-functional programming-like
  - No variadic args, no overloading, no OOP (inheritance etc), different philosophies (e.g. "rust libs never panic")

Sparsity in boost::histogram

- You can use std::map/std::unordered_map in C++ boost::histogram
  (I just learned about this on Wednesday…)
- There are no python bindings, because several things are unclear:
  - How to access the C++ buffers, e.g. keys/values as 1D NumPy arrays?
  - How to serialize?
  - How to slice?
- At some point we'd want to have a performant HashMap implementation
  (C++ std::map is slow)
- Starting to add HashMap storage with Henry soon…

Summary

- Sparse memory layout of histograms can help reduce memory consumption in Dask workers
- High dimensionality typically increases sparsity
- Performance of rust-histogram:
  - My implementation is worse than boost-histogram, but relatively close to NumPy
  - It's still fast: filling 10M events takes ~200ms (link)
    …but analyzing 10M events takes typically several minutes
- We need to think about memory of ND histograms
- Key takeaways: sparsity is possible in boost::histogram & Rust (tooling) ❤️