Boosting RDataFrame performance with transparent bulk event processing

Enrico Guiraud, J. Blomer, P. Canal, A. Naumann
CHEP 2023, 8/5/2023


“bulk data processing in ROOT, high-energy physics, abstract art”

Bulk processing: what and why

A large R&D effort, for a large impact

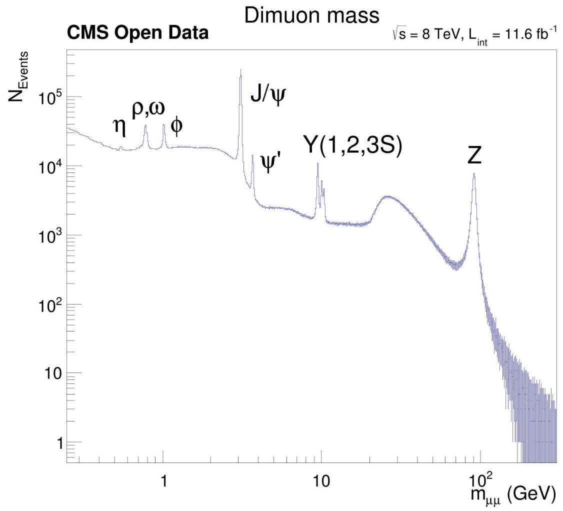
RDataFrame (RDF) is ROOT's modern analysis interface addressing most common use cases with **one high-level programming model** that performs well, scales well and enables **HEP-specific ergonomics**, in C++ and Python. See e.g. E. Guiraud, ICHEP 2022.

Given RDF's popularity, we decided to investigate the **potential performance benefits** of a **large refactoring** of its inner data processing loop.

What is presented here is **current R&D** that we plan to release as part of ROOT this year.

How things currently look

```
df.Filter("nMuon == 2 && charge[0]*charge[1] < 0")
  .Define("mass",
          InvariantMass<float>,
          {"pt", "eta", "phi", "mass"})
  .Histo1D("mass");
```



From the [Dimuon RDF tutorial](#).

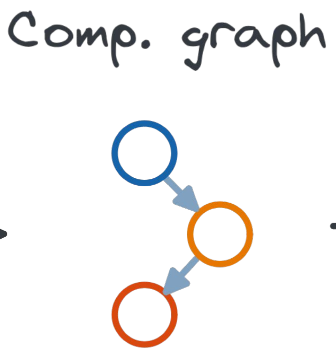
On my laptop, reading 61M events from warm cache: **2.5M events/s** or **101 MiB/s** (**ZSTD-compressed data** actually decompressed and processed, single-core).

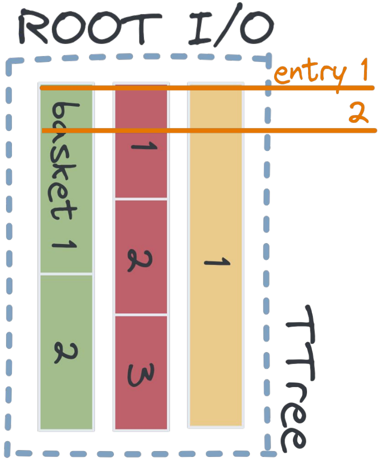
This R&D does not speed up raw I/O and decompression, but see [the RNTuple talk](#).

Previously: event by event processing

User code

```
df.Filter(...)
  .Define("mass", ...)
  .Histo1D("mass");
```





ROOT I/O loads data from storage in bulks, but RDF goes through it event-wise:

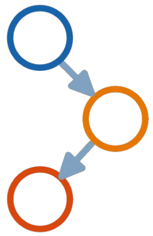
- **per-event overheads** related to preparing data for processing
- unfriendly to the CPU data/instruction caches
- prevents some optimization opportunities, e.g. **GPU offloading**

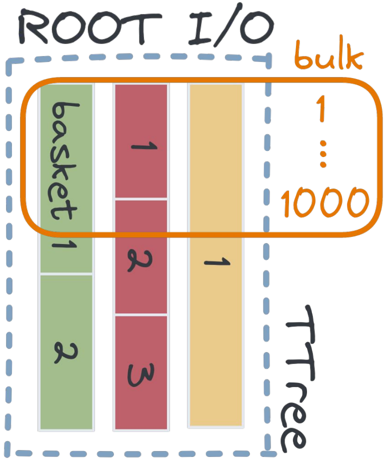
R&D: bulk data processing in RDF

User code

```
df.Filter(...)
  .Define("mass", ...)
  .Histo1D("mass");
```

Comp. graph

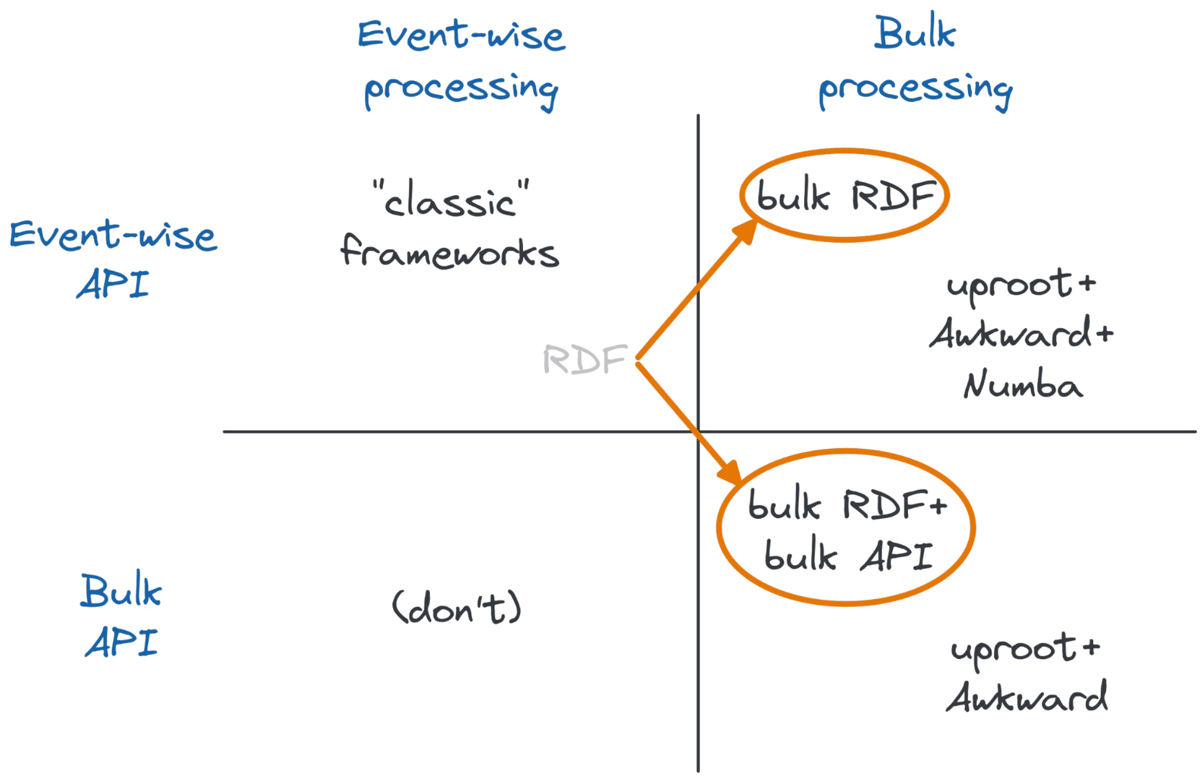




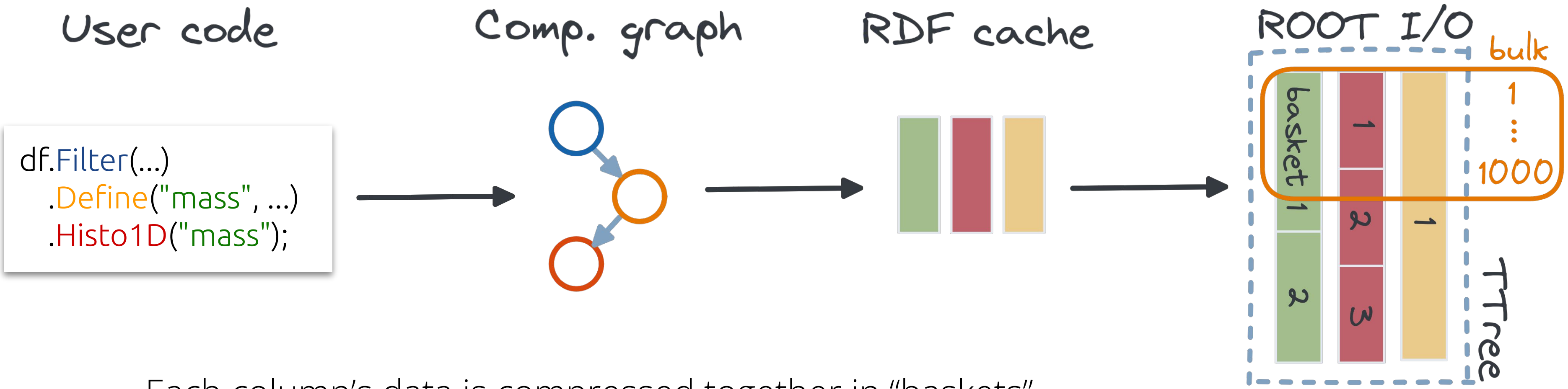
For each node of the comp. graph, RDF runs over a bulk of entries at a time:

- **overheads** related to data preparation are now **per-bulk**
- friendlier to CPU caches
- enables **GPU offloading** and specialized, vectorized operations

The big picture



Complication #1: unaligned baskets



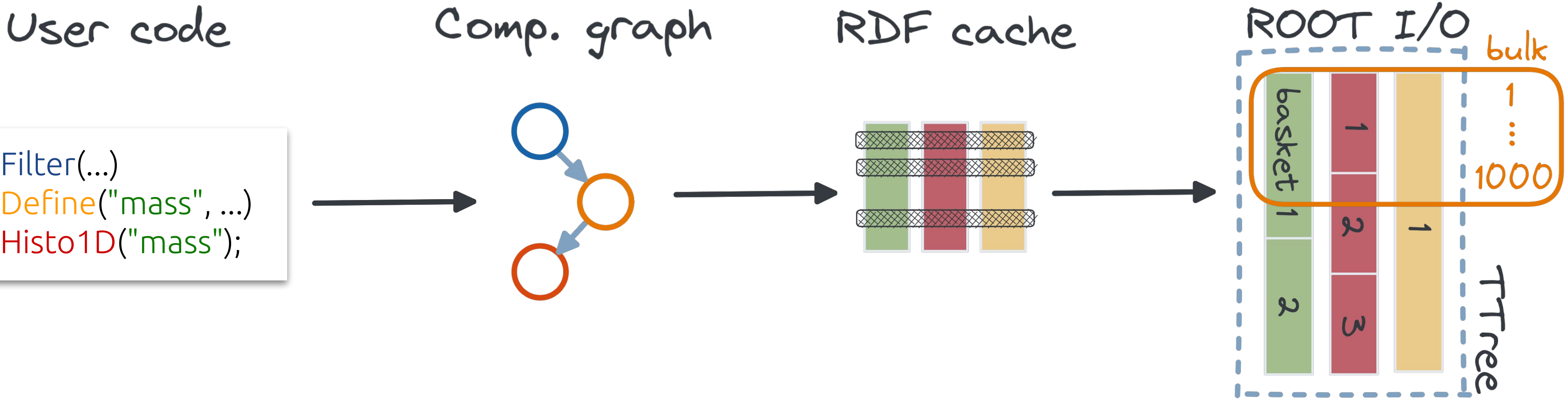
Each column's data is compressed together in "baskets".
Different columns have **different basket boundaries**.

➔ RDF bulks (transversal to columns) cannot respect basket boundaries

➔ **we decompress values into RDF's own storage** (may require +1 copy):

- guarantees all column values in a bulk are contiguous in memory
- avoids redundant decompressions due to basket hopping

Complication #2: event masks




```
df.Filter(...)
  .Define("mass", ...)
  .Histo1D("mass");
```

Most operations on the bulk (e.g. histogram fills) are **conditional on the event mask**

➔ very **hard for the compiler to auto-vectorize** operations

Different branches of the comp. graph require **same values with different masks**

➔ care required to coordinate loads/computations of values across the graph

Enabling vectorized operations

Event-wise

```
float square(float x);

df.Define("x2", square, {"x"});
```

Bulk

```
void bulkSquare(const REventMask &m,
                RVec<float> &results,
                const RVec<float> &xs);

df.Define("x2", bulkSquare, {"x"});
```

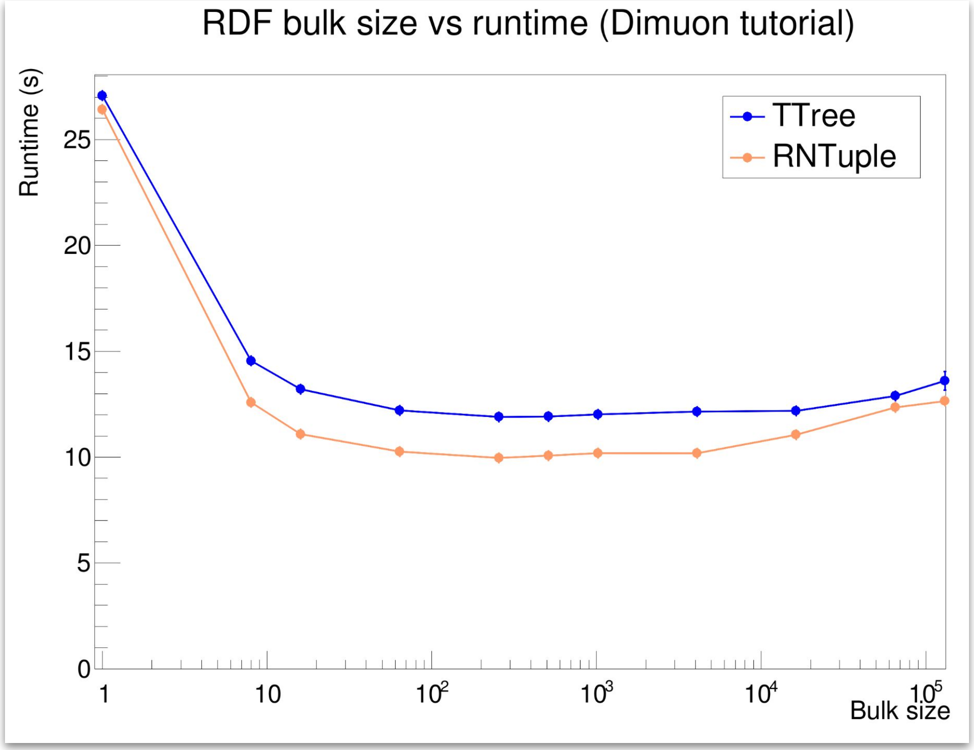
Event-wise operations remain the default,
we add the **option to implement bulk versions** of heavy computations.

bulkSquare operates on many contiguous values,
possibly **ignoring the event mask** in order to leverage **CPU vectorization**.
It could also dispatch the computation to a **GPU kernel** (e.g. for ML inference).

“me running performance benchmarks on my code as an Edward Hopper painting”

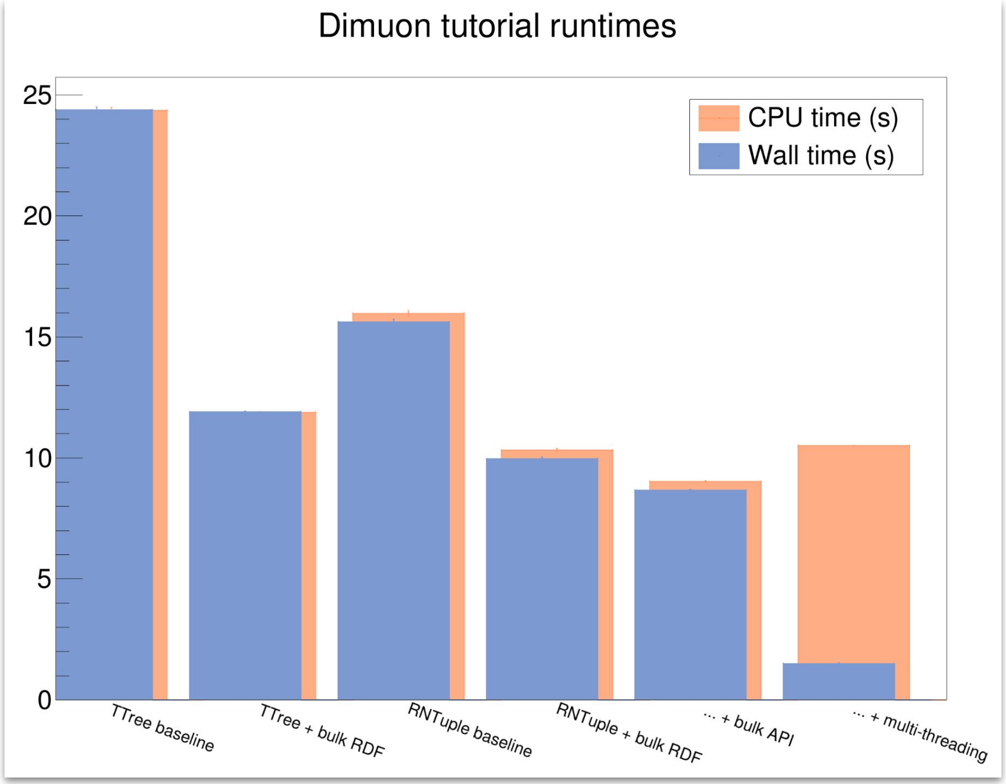
Performance benchmarks

A new free parameter: the bulk size



- **runtime plateaus in a sensible range** (256-4096, error bars are negligible)
- **no meaningful RAM usage increase** except for the largest bulk sizes
- plot shows trends for the dimuon analysis, but behavior has been consistent across different benchmarks, machines, TTree and RNTuple
- still, GPU kernels and specialized use cases will have different requirements
- can pick a **reasonable default** in the plateau range, **customizable at runtime**

Better runtimes on simple schemas

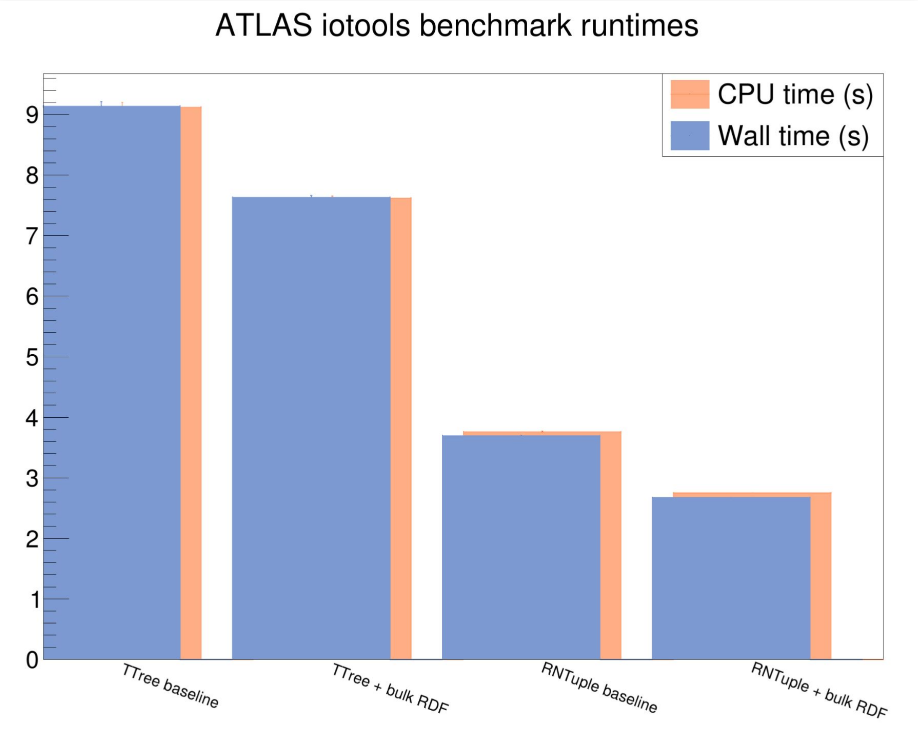


Speed-up on simple column types (floats and C-style arrays thereof)
2x for TTree
1.6x for RNTuple

- + bulk API: a more CPU-friendly version of the invariant mass calculation provides a further 15% speed-up (can likely be improved)
- speed-up compounds with multi-threading (8 threads here)
- error bars are negligible

Benchmark source code

Smaller gains on non-trivial types



Speed-up with non-trivial column types (std::vector<float>, std::vector<bool>)
1.2x for TTree
1.4x for RNTuple

- cannot leverage low-level TTree bulk I/O; situation is better with RNTuple
- harder to “bulkify” the value preparation logic for complex types, even if it is just STL collections
- std::vector<bool> introduces extra complications because of bit-packing

[Benchmark source code]

"an impressionist painting of happy CPU cores"

Conclusions

Moving from R&D to production

We showed:

- there is **potential for a much faster RDF** for common analysis use cases (for TTree and RNTuple, for each core of a multi-core run)
- ...especially for simple schemas ("flat ntuples")
- **bulk-wise computation kernels** can speed up expensive computations

Remaining challenges before prime time:

- graceful degradation in case of files with bad clustering/basket sizes
- some RDF features still unsupported, e.g. callbacks and DefinePerSample

Let me know if you are interested in becoming a beta tester!

“isometric diorama of a library”

Back-up

Code and benchmark setup

some core RDF features still unsupported

- development branch: [github.com/eguiraud/root/tree/df-bulk]
- benchmarks: [github.com/eguiraud/rdf-benchmarks]

- CPU: Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz
- Intel turbo boost, hyper-threading and speedstep turned off
- CPU frequency governor set to “performance”
- no JIT-ed code, all code built with -O3 optimization with gcc 12.2.1 (O2 vs O3 makes a huge difference for RNTuple!)
- all input files were ZSTD-compressed, read from warm filesystem cache

The algorithm in a nutshell

Current

```
for e in entries:
  if eval_filter(e):
    x = eval_define(e)
    fill_histo(x)
```

Bulk

```
max_size = get_max_bulk_size()
while not done:
  bulk = datasource.next_range(max_size)
  mask = event_mask(bulk)
  xs = eval_define(bulk, mask)
  fill_histo(bulk, mask, xs)
```

event_mask

```
vs = load_col_values(bulk)
for e in bulk:
  m[e] = eval_filter(vs[e])
return m
```

eval_define

```
vs = load_col_values(bulk)
for e in bulk:
  if (mask[e])
    xs[e] = eval_expr(vs[e])
return xs
```

fill_histo

```
for e in bulk:
  if (mask[e])
    histo.fill(xs[e])
```

- Filter/Define evaluation and histogram filling now in a hot loop
- RDF talks to the I/O layer once per bulk, not once per entry
- need to cache bulk.size() column values, filter/define results

What about memory usage?

In our tests, the increase in memory usage due to having to store bulkSize results and cache bulkSize column values was negligible with respect to the baseline memory usage of ROOT I/O, the interpreter, the histograms.

The latter factors contribute to O(100) MBs of allocations: that’s *a lot* of column values cached.

Code for runtime vs bulk size plot

In the speaker notes.

Code for dimuon bulk speed-up plot

In the speaker notes.

Code for ATLAS iotools bulk speed-up plot

See speaker notes