Systematic optimization of the LHCB B2HHH analysis using LLAMA

Bernhard Manfred Gruber

Motivation

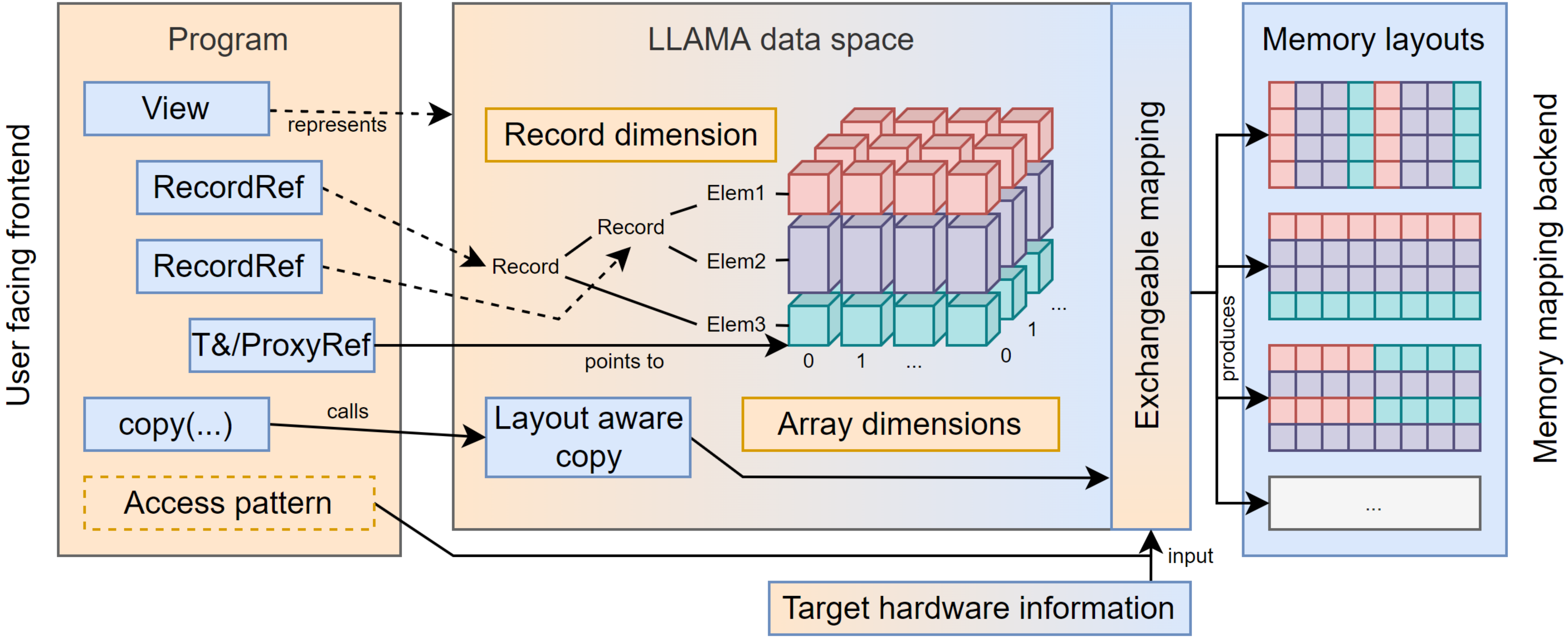
- TODO

- I need content for a new CS journal paper
- I would like to test and showcase a few new features in LLAMA, notably reduced precision mappings
- Jakob suggested to try the LHCB B2HHH analysis example

LLAMA

- Low-Level Abstraction of Memory Access
- Separates algorithmic view of data and mapping to memory
  - Different memory layouts may be chosen without touching the algorithm
- Header-only, portable, C++17/C++20 library, LGPL3+
- Designed to integrate with CUDA/HIP, SYCL, alpaka, …
  - … but orthogonal
- GitHub: https://github.com/alpaka-group/llama

Concept



User facing frontend

Program

View

represents

RecordRef

RecordRef

T&/ProxyRef

copy(...)

calls

Access pattern

LLAMA data space

Record dimension

Record

Record

Elem1

Elem2

Elem3

points to

0

1

...

0

1

...

Layout aware copy

Array dimensions

Exchangeable mapping

input

Target hardware information

produces

Memory layouts

...

Memory mapping backend

LHCB B2HHH analysis

- Is one of the test/benchmark examples for comparing RNTuple/TTree
  - See https://github.com/jblomer/iotools/blob/master/lhcb.cxx
- Characteristics
  - Small and simple analysis
  - No jagged arrays, which are not supported (yet) in LLAMA
  - Several filters leading to sparse reads of columns/events
  - Simple computation and one observable (histogram)
    - Thus, probably memory bound

Dataset and data types

```
root [0]  auto df = ROOT::RDF::Experimental::FromRNTuple("DecayTree", 
"../../iotools/B2HHH~none.ntuple");
Warning in <[ROOT.NTuple] Warning
/home/bgruber/dev/root/tree/ntuple/v7/src/RNTupleSerialize.cxx:1208 in static
ROOT::Experimental::RResult<void>
ROOT::Experimental::Internal::RNTupleSerializer::DeserializeHeaderV1(const
void*, uint32_t, ROOT::Experimental::RNTupleDescriptorBuilder&)>: Pre-release
format version: RC 1
root [2] *df.Count()
(unsigned long long) 8556118
root [e] df.Describe()
(ROOT::RDF::RDFDescription) Dataframe from datasource RNTupleDS

Property                Value
--------                -----
Columns in total           26
Columns from defines        0
Event loops run             1
Processing slots            1
```

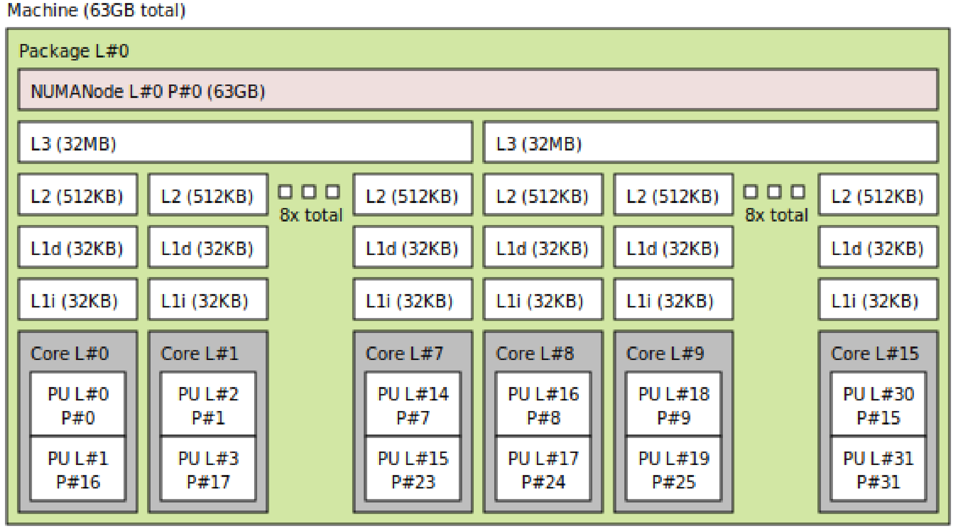
```
Column                  Type            Origin
------                  ----            ------
B_FlightDistance        double          Dataset
B_VertexChi2            double          Dataset
H1_Charge               std::int32_t    Dataset
H1_IpChi2               double          Dataset
H1_PX                   double          Dataset
H1_PY                   double          Dataset
H1_PZ                   double          Dataset
H1_ProbK                double          Dataset
H1_ProbPi               double          Dataset
H1_isMuon               std::int32_t    Dataset
H2_Charge               std::int32_t    Dataset
H2_IpChi2               double          Dataset
H2_PX                   double          Dataset
H2_PY                   double          Dataset
H2_PZ                   double          Dataset
H2_ProbK                double          Dataset
H2_ProbPi               double          Dataset
H2_isMuon               std::int32_t    Dataset
H3_Charge               std::int32_t    Dataset
H3_IpChi2               double          Dataset
H3_PX                   double          Dataset
H3_PY                   double          Dataset
H3_PZ                   double          Dataset
H3_ProbK                double          Dataset
H3_ProbPi               double          Dataset
H3_isMuon               std::int32_t    Dataset
```

Benchmark setup

- For this exploration we only look at in-memory data layouts
  - Typical analyses include reading data from disk
  - The RNTuple is loaded from disk and converted to a LLAMA view *before* the benchmark
- Analysis parallelized using OpenMP
- Reported times are average of 100 analysis runs
  - Just loading, filtering, computing one observable and histogram fill
  - Excluding histogram creation and reduction
- All code on GitHub: https://github.com/alpaka-group/llama/blob/develop/examples/root/lhcb_analysis/lhcb.cpp

Benchmark machine

- All benchmarks were run on my workstation
- AMD Ryzen 9 5950X 16 cores / 32 threads
- One thread per core
  - OMP_NUM_THREADS=16
  - OMP_PLACES=cores
  - OMP_PROC_BIND=true



- Fun side fact: with SMT disabled in BIOS, 8 threads was faster than 16 threads
  - I blamed it on the non-shared L3 caches …

LHCB B2HHH analysis RNTuple code (adapted)

```
constexpr double prob_k_cut = 0.5;
constexpr double prob_pi_cut = 0.5;

for (auto i : ntuple->GetEntryRange()) {
   if (viewH1IsMuon(i) || viewH2IsMuon(i) || viewH3IsMuon(i))
      continue;

   if (viewH1ProbK(i) < prob_k_cut) continue;
   if (viewH2ProbK(i) < prob_k_cut) continue;
   if (viewH3ProbK(i) < prob_k_cut) continue;

   if (viewH1ProbPi(i) > prob_pi_cut) continue;
   if (viewH2ProbPi(i) > prob_pi_cut) continue;
   if (viewH3ProbPi(i) > prob_pi_cut) continue;

   double b_px = viewH1PX(i) + viewH2PX(i) + viewH3PX(i);
   double b_py = viewH1PY(i) + viewH2PY(i) + viewH3PY(i);
   double b_pz = viewH1PZ(i) + viewH2PZ(i) + viewH3PZ(i);
   double b_p2 = GetP2(b_px, b_py, b_pz);
   double k1_E = GetKE(viewH1PX(i), viewH1PY(i), viewH1PZ(i));
   double k2_E = GetKE(viewH2PX(i), viewH2PY(i), viewH2PZ(i));
   double k3_E = GetKE(viewH3PX(i), viewH3PY(i), viewH3PZ(i));
   double b_E = k1_E + k2_E + k3_E;
   double b_mass = sqrt(b_E*b_E - b_p2);
   hMass->Fill(b_mass);
}
```

https://github.com/jblomer/iotools/blob/master/lhcb.cxx#L287-L348

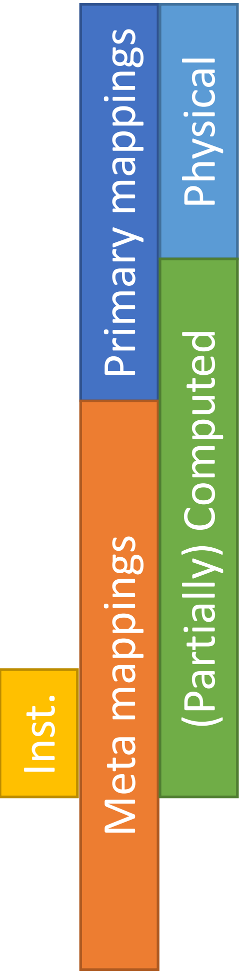
LLAMA version

```cpp
#pragma omp parallel for
for(RE::NTupleSize_t i = 0; i < n; i++) {
    auto&& event = view[i];

    if(event(H1isMuon{})) continue;
    if(event(H2isMuon{})) continue;
    if(event(H3isMuon{})) continue;

    if(event(H1ProbK{}) < probKCut) continue;
    if(event(H2ProbK{}) < probKCut) continue;
    if(event(H3ProbK{}) < probKCut) continue;

    if(event(H1ProbPi{}) > probPiCut) continue;
    if(event(H2ProbPi{}) > probPiCut) continue;
    if(event(H3ProbPi{}) > probPiCut) continue;

    const double h1px = event(H1PX{});
    const double h1py = event(H1PY{});
    const double h1pz = event(H1PZ{});
    const double h2px = event(H2PX{});
    const double h2py = event(H2PY{});
    const double h2pz = event(H2PZ{});
    const double h3px = event(H3PX{});
    const double h3py = event(H3PY{});
    const double h3pz = event(H3PZ{});

    const double bpx = h1px + h2px + h3px;
    const double bpy = h1py + h2py + h3py;
    const double bpz = h1pz + h2pz + h3pz;
    const double bp2 = getP2(bpx, bpy, bpz);
    const double k1e = getKE(h1px, h1py, h1pz);
    const double k2e = getKE(h2px, h2py, h2pz);
    const double k3e = getKE(h3px, h3py, h3pz);
    const double be = k1e + k2e + k3e;
    const double bmass = std::sqrt(be * be - bp2);

    hists[omp_get_thread_num()].Fill(bmass);
}
```

Avoid repeated access to the same data for more accurate instrumentation

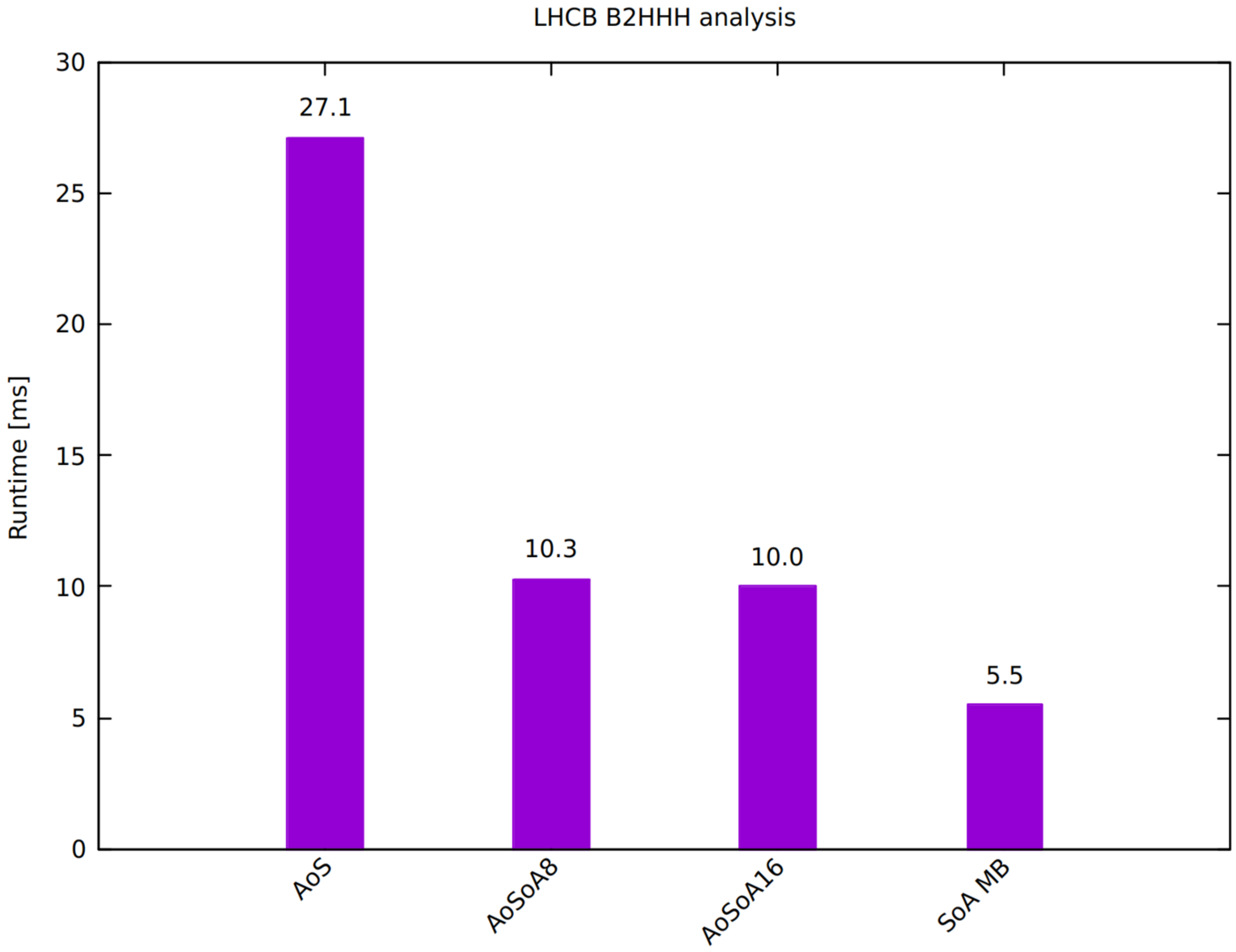
https://github.com/alpaka-group/llama/blob/develop/examples/root/lhcb_analysis/lhcb.cpp

Available mappings and their customization



- **AoS**: Aligned/Packed, ND-array linearizers, struct member reordering
- **SoA**: Single/Multi blob, Aligned/Packed sub arrays, ND-array linearizers, struct member reordering
- **AoSoA**: Inner array size, ND-array linearizers, struct member reordering
- **One**: Aligned/Packed, struct member reordering, Map all array indices to the same record instance
- **BitPackFloatSoA**, **BitPackIntSoA**: Bit count for value/mantissa/exponent, ND-array linearizers, storage type
- **Null**: Read returns default constructed value, writes are discarded
- **ChangeType**: Replace record dim types for storage, forward to inner mapping
- **Projection**: Run function on record dim types on load/store, forward to inner mapping
- **Bytesplit**: Split all types in static byte arrays, then forward to inner mapping
- **Byteswap**: Swap bytes of data types on load/store, forward to inner mapping
- **Trace**: Trace record dim access/read/write counts, then forward to inner mapping
- **Heatmap**: Count accesses per blob byte (or coarser), then forward to inner mapping
- **Split**: Split record dimension in two, forward each part to inner mappings, leave or merge blobs of inner mappings
- **PermuteArrayIndex**: Permutate array indices, forward to inner mapping

Benchmark: Trying a couple of layouts



Trying a couple of layouts

- Memory mappings can be switched without touching the algorithm
- LLAMA provides many ready-to-use memory mappings
- LLAMA is an ideal platform for rapid exploration by iterating through a list of common mappings

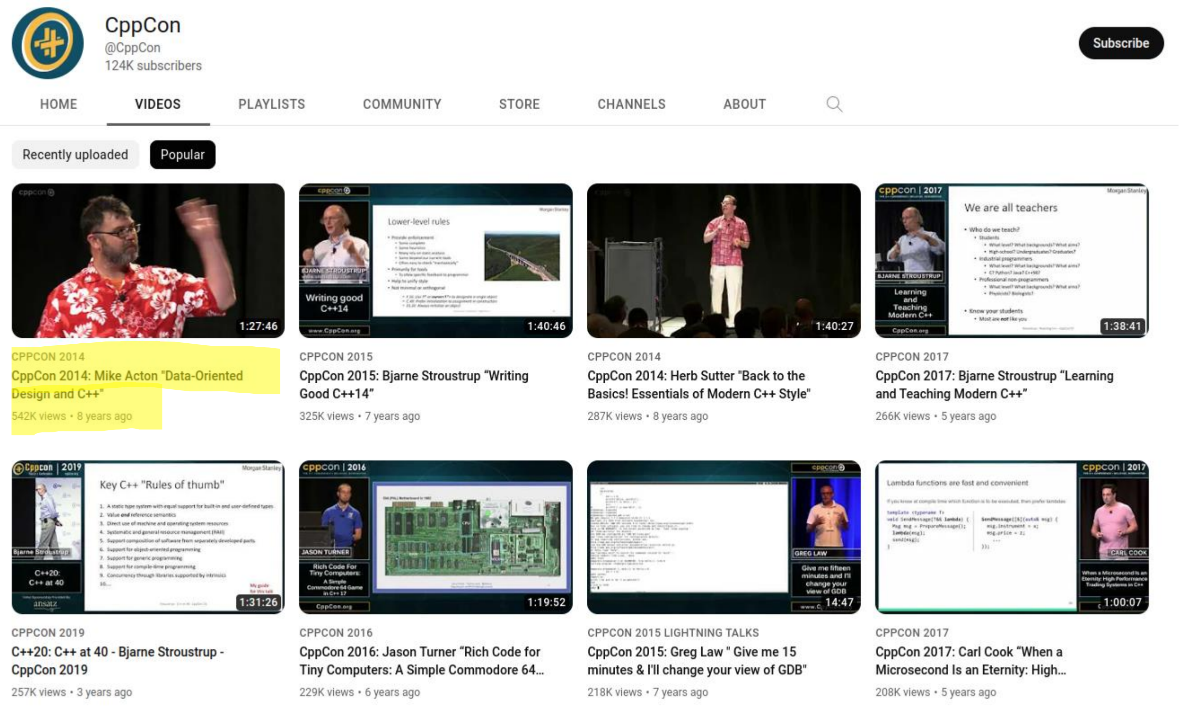
- However, the explorable space is huge:
  - LLAMA’s mappings have many tuning parameters
  - LLAMA mappings can be combined in many ways via meta mappings
  - You could add your own mappings – huge possibilities

- Finding the best layout requires insight, tools and some benchmarking. It’s unlikely, LLAMA already has it ☺

Data-oriented design

- … a program optimization approach motivated by efficient usage of the CPU cache, used in video game development. The approach is to focus on the **data layout**, **separating** and **sorting** fields according to when they are needed […]. Proponents include Mike Acton, Scott Meyers, and Jonathan Blow.
- … became especially popular in the mid to late 2000s
- From Wikipedia, emphasis mine

“Know your data”



CPPCON 2014
CppCon 2014: Mike Acton "Data-Oriented Design and C++"
542K views • 8 years ago

CPPCON 2015
CppCon 2015: Bjarne Stroustrup “Writing Good C++14”
325K views • 7 years ago

CPPCON 2014
CppCon 2014: Herb Sutter "Back to the Basics! Essentials of Modern C++ Style"
287K views • 8 years ago

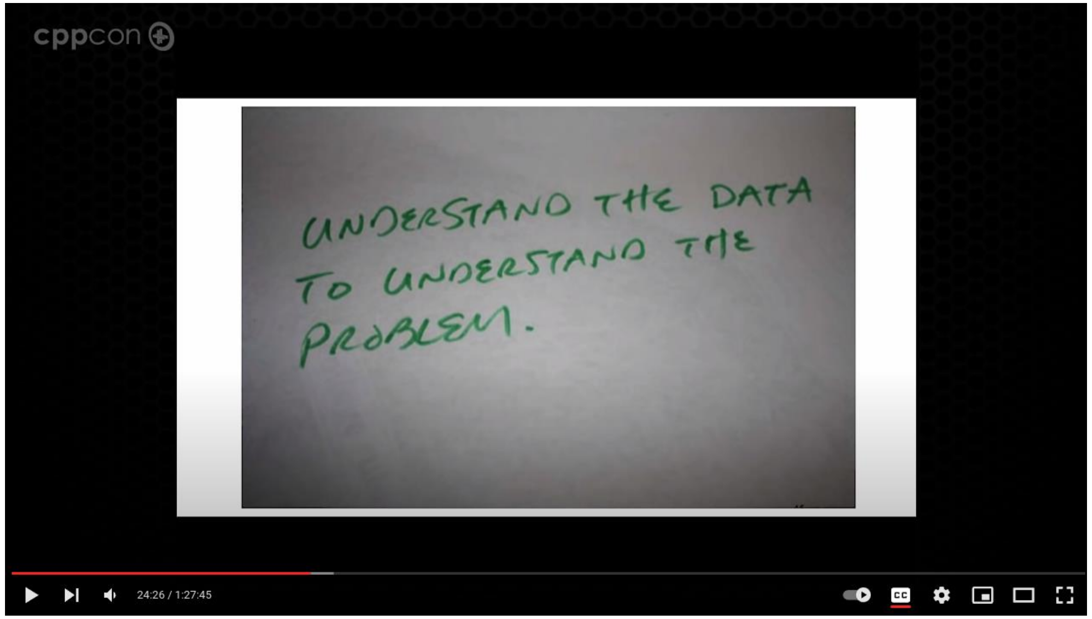
CPPCON 2017
CppCon 2017: Bjarne Stroustrup “Learning and Teaching Modern C++”
266K views • 5 years ago

CPPCON 2019
C++20: C++ at 40 - Bjarne Stroustrup - CppCon 2019
257K views • 3 years ago

CPPCON 2016
CppCon 2016: Jason Turner “Rich Code for Tiny Computers: A Simple Commodore 64…
229K views • 6 years ago

CPPCON 2015 LIGHTNING TALKS
CppCon 2015: Greg Law " Give me 15 minutes & I'll change your view of GDB"
218K views • 7 years ago

CPPCON 2017
CppCon 2017: Carl Cook “When a Microsecond Is an Eternity: High…
208K views • 5 years ago



UNDERSTAND THE DATA TO UNDERSTAND THE PROBLEM.

Quiz: how old is this quote?

“Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck [bus between CPU and memory]. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.”

Quiz: how old is this quote?

“Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck [bus between CPU and memory]. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.”

John Backus in his 1977 ACM Turing Award lecture

(replace “word” by “**cacheline**” and we are in 2023)

Gathering insight

- How often is which data touched?
- Depends a lot on the filters …

- We could insert an increment of an (atomic) counter after each filter to see how often the filter triggers
- We can use LLAMA’s software instrumentation to visualize the access pattern

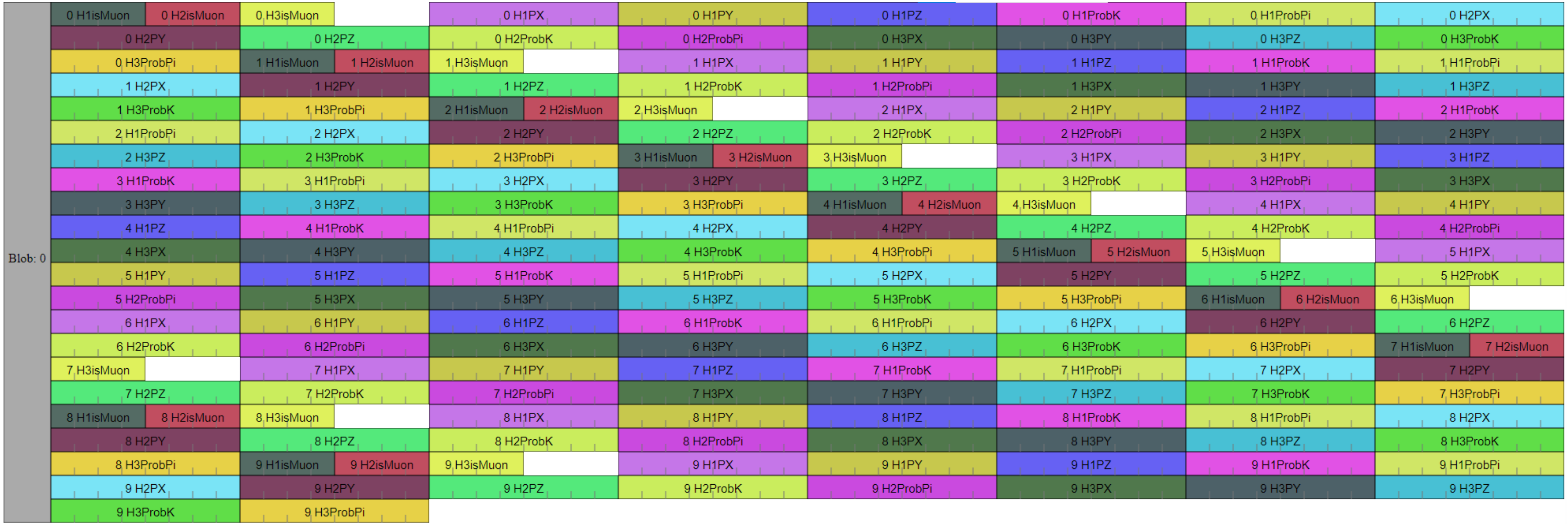
Gathering insight – filters

```
#pragma omp parallel for
for(RE::NTupleSize_t i = 0; i < n; i++) {
    auto&& event = view[i];

    if(event(H1isMuon{})) continue;
    if(event(H2isMuon{})) continue;
    if(event(H3isMuon{})) continue;

    if(event(H1ProbK{}) < probKCut) continue;
    if(event(H2ProbK{}) < probKCut) continue;
    if(event(H3ProbK{}) < probKCut) continue;

    if(event(H1ProbPi{}) > probPiCut) continue;
    if(event(H2ProbPi{}) > probPiCut) continue;
    if(event(H3ProbPi{}) > probPiCut) continue;

    // compute ...

    hists[omp_get_thread_num()].Fill(bmass);
}
```

Step	**Remaining events**	
before filtering	8556118	100.00%
H1isMuon filter	7368489	86.12%
H2isMuon filter	6951588	81.25%
H3isMuon filter	6311517	73.77%
H1PropK filter	623038	7.28%
H2PropK filter	95742	1.12%
H3PropK filter	26959	0.32%
J1ProbPi filter	26012	0.30%
J2ProbPi filter	25359	0.30%
J3ProbPi filter	23895	0.28%

Gathering insight – memory layout – AoS

wrap after 64 Bytes



Blob: 0

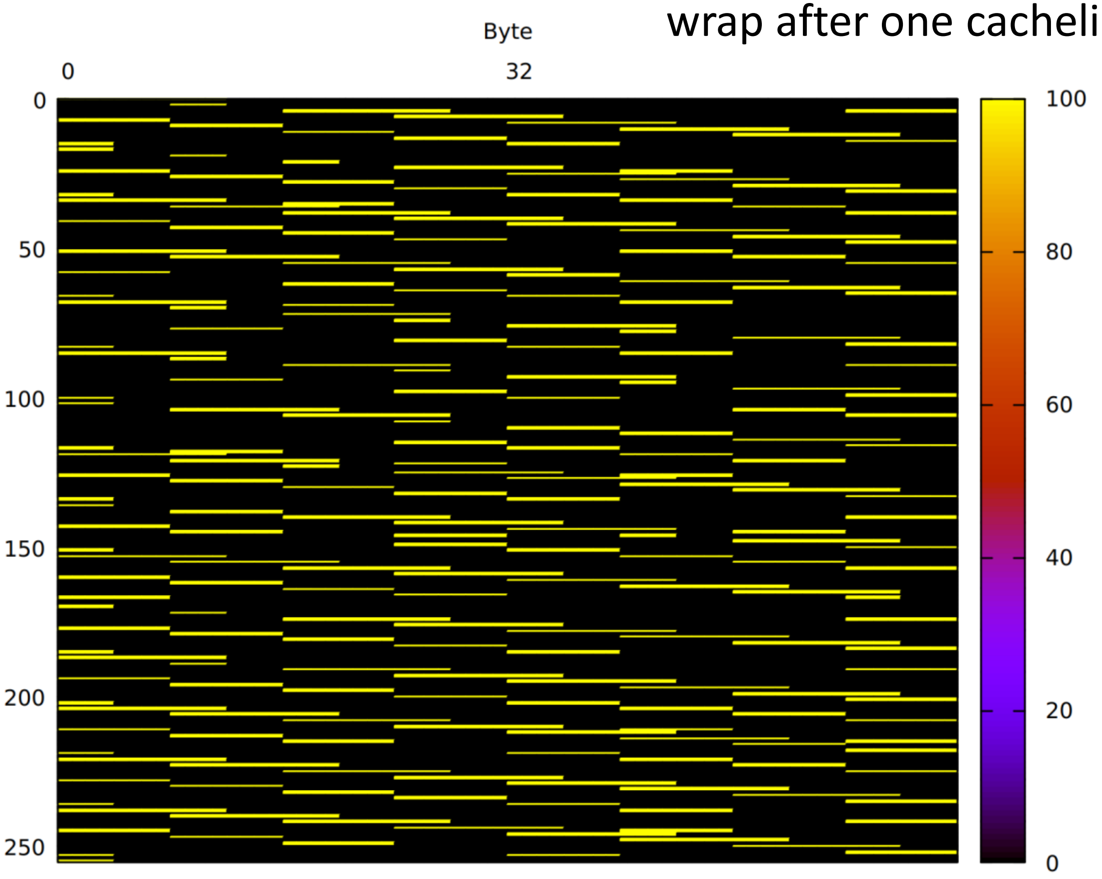
0 H1isMuon 0 H2isMuon	0 H3isMuon	0 H1PX	0 H1PY	0 H1PZ	0 H1ProbK	0 H1ProbPi	0 H2PX
0 H2PY	0 H2PZ	0 H2ProbK	0 H2ProbPi	0 H3PX	0 H3PY	0 H3PZ	0 H3ProbK
0 H3ProbPi	1 H1isMuon 1 H2isMuon	1 H3isMuon	1 H1PX	1 H1PY	1 H1PZ	1 H1ProbK	1 H1ProbPi
1 H2PX	1 H2PY	1 H2PZ	1 H2ProbK	1 H2ProbPi	1 H3PX	1 H3PY	1 H3PZ
1 H3ProbK	1 H3ProbPi	2 H1isMuon 2 H2isMuon	2 H3isMuon	2 H1PX	2 H1PY	2 H1PZ	2 H1ProbK
2 H1ProbPi	2 H2PX	2 H2PY	2 H2PZ	2 H2ProbK	2 H2ProbPi	2 H3PX	2 H3PY
2 H3PZ	2 H3ProbK	2 H3ProbPi	3 H1isMuon 3 H2isMuon	3 H3isMuon	3 H1PX	3 H1PY	3 H1PZ
3 H1ProbK	3 H1ProbPi	3 H2PX	3 H2PY	3 H2PZ	3 H2ProbK	3 H2ProbPi	3 H3PX
3 H3PY	3 H3PZ	3 H3ProbK	3 H3ProbPi	4 H1isMuon 4 H2isMuon	4 H3isMuon	4 H1PX	4 H1PY
4 H1PZ	4 H1ProbK	4 H1ProbPi	4 H2PX	4 H2PY	4 H2PZ	4 H2ProbK	4 H2ProbPi
4 H3PX	4 H3PY	4 H3PZ	4 H3ProbK	4 H3ProbPi	5 H1isMuon 5 H2isMuon	5 H3isMuon	5 H1PX
5 H1PY	5 H1PZ	5 H1ProbK	5 H1ProbPi	5 H2PX	5 H2PY	5 H2PZ	5 H2ProbK
5 H2ProbPi	5 H3PX	5 H3PY	5 H3PZ	5 H3ProbK	5 H3ProbPi	6 H1isMuon 6 H2isMuon	6 H3isMuon
6 H1PX	6 H1PY	6 H1PZ	6 H1ProbK	6 H1ProbPi	6 H2PX	6 H2PY	6 H2PZ
6 H2ProbK	6 H2ProbPi	6 H3PX	6 H3PY	6 H3PZ	6 H3ProbK	6 H3ProbPi	7 H1isMuon 7 H2isMuon
7 H3isMuon	7 H1PX	7 H1PY	7 H1PZ	7 H1ProbK	7 H1ProbPi	7 H2PX	7 H2PY
7 H2PZ	7 H2ProbK	7 H2ProbPi	7 H3PX	7 H3PY	7 H3PZ	7 H3ProbK	7 H3ProbPi
8 H1isMuon 8 H2isMuon	8 H3isMuon	8 H1PX	8 H1PY	8 H1PZ	8 H1ProbK	8 H1ProbPi	8 H2PX
8 H2PY	8 H2PZ	8 H2ProbK	8 H2ProbPi	8 H3PX	8 H3PY	8 H3PZ	8 H3ProbK
8 H3ProbPi	9 H1isMuon 9 H2isMuon	9 H3isMuon	9 H1PX	9 H1PY	9 H1PZ	9 H1ProbK	9 H1ProbPi
9 H2PX	9 H2PY	9 H2PZ	9 H2ProbK	9 H2ProbPi	9 H3PX	9 H3PY	9 H3PZ
9 H3ProbK	9 H3ProbPi						

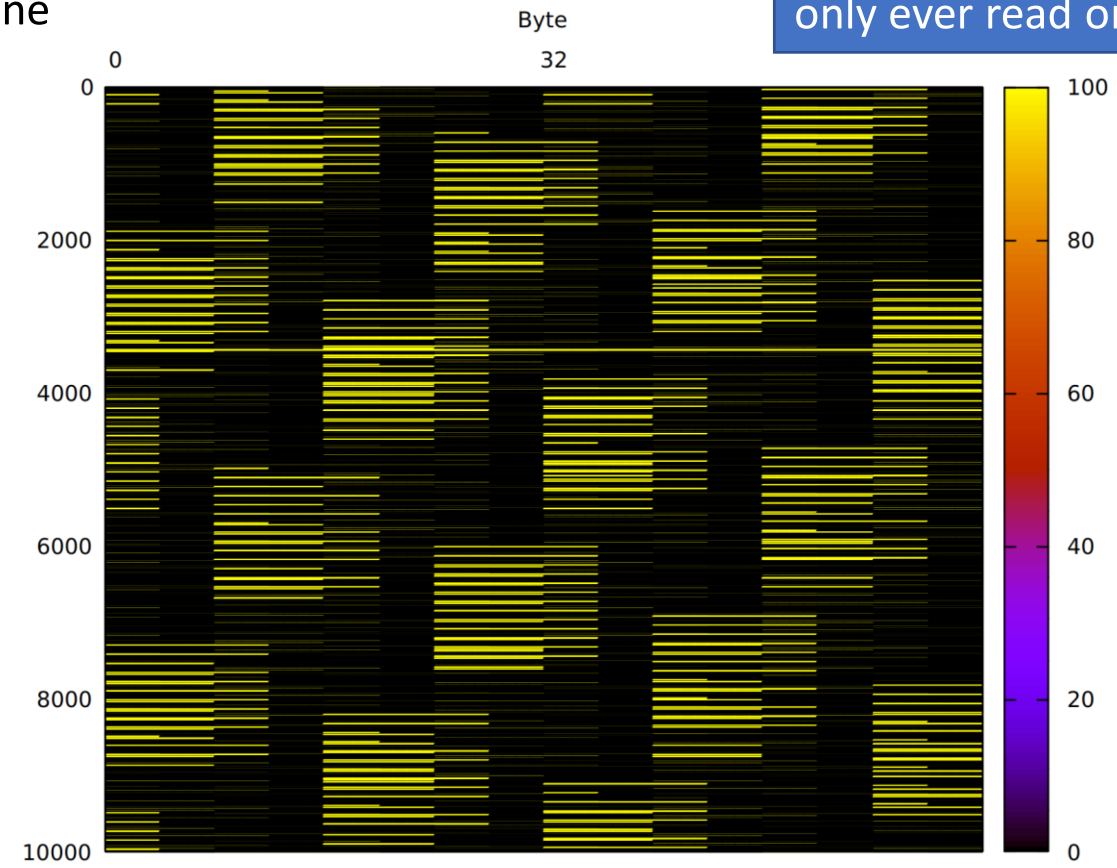
1 event = 136 byte

Gathering insight – Heatmaps – AoS

100 analyses, max 100 reads -> data is only ever read once!

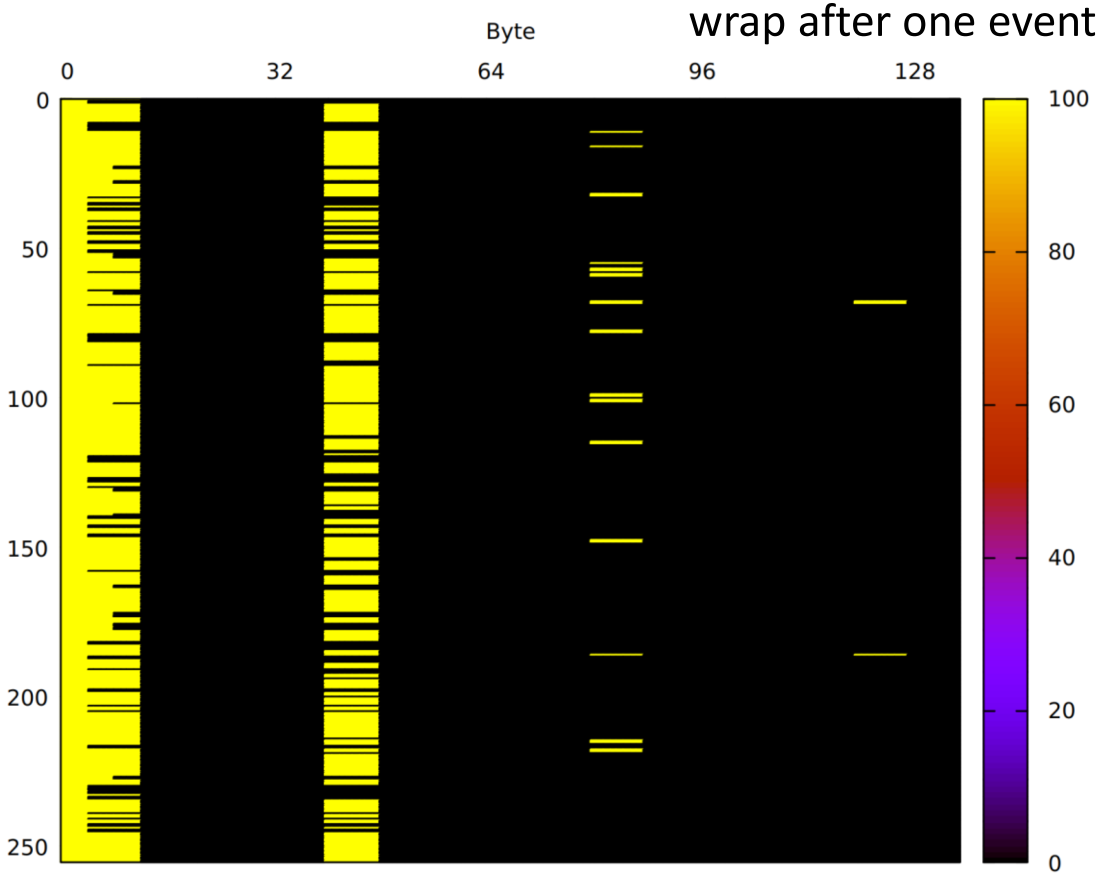
wrap after one cacheline

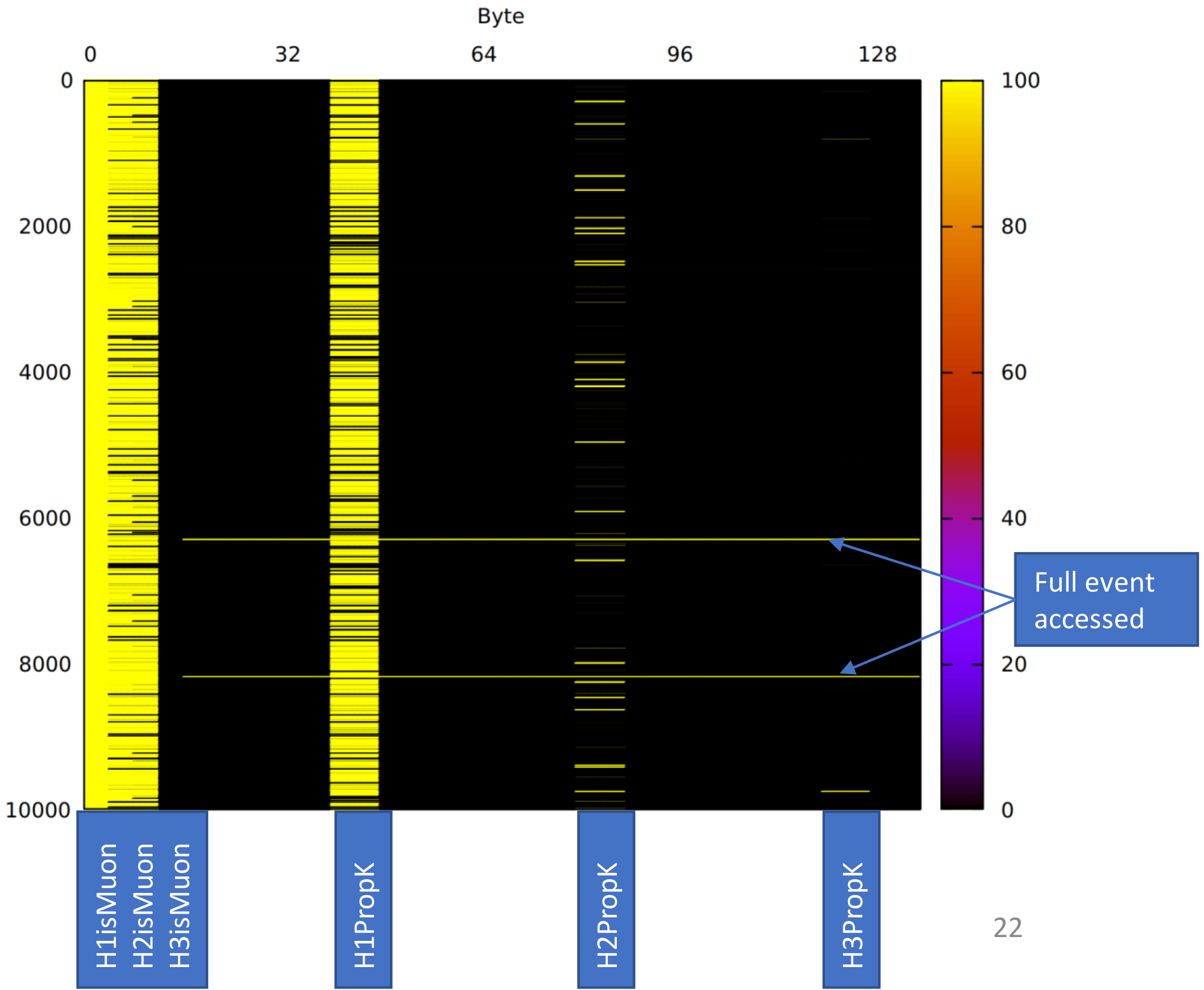




Gathering insight – Heatmaps – AoS

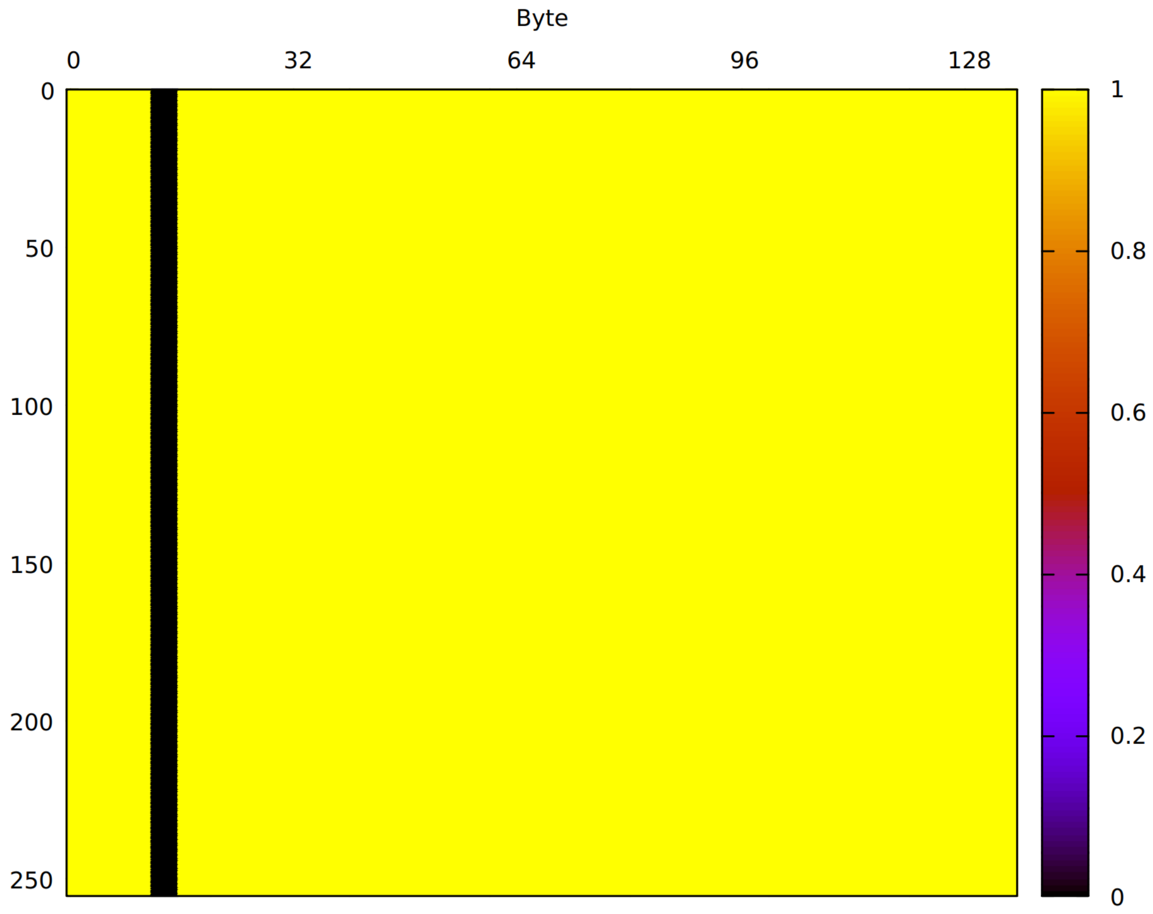
wrap after one event





Full event accessed

For completeness: conversion heatmap



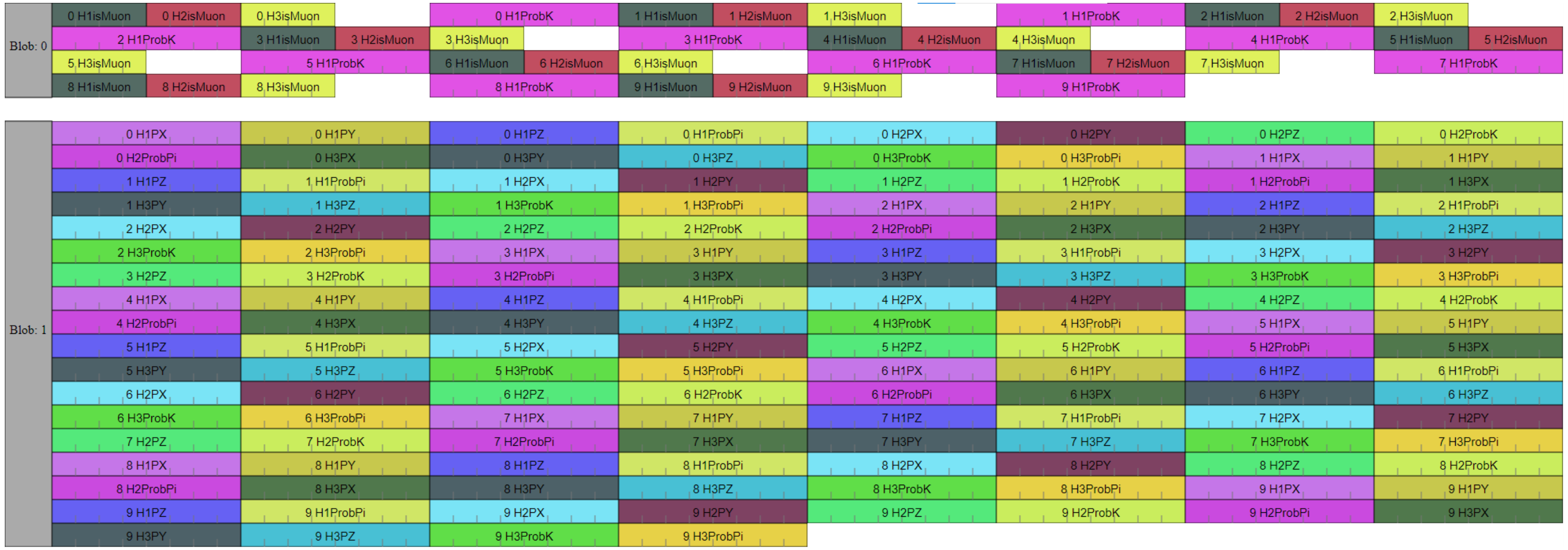
Insights so far

- Columns H1isMuon/H2isMuon/H3isMuon are hot and accessed densely (100% - 81.25%)
- H1PropK is also warm (73.77%), H2PropK still a bit (7.28%)
- H3PropK/J1PropPi/J2PropPi/J3PropPi are cold, similarly cold as the remaining event data used for computation (1.12% – 0.28%)

- Let’s design a custom memory layout fitting this access pattern!

Designing a memory layout 1/7

- Custom layout 1:
  - Separate out H1isMuon/H2isMuon/H3isMuon/H1PropK, but keep the 4 values close (AoS)
  - Keep the remaining values close (AoS)
  - But: Requires padding and wastes 1/6th of bandwidth on hot data

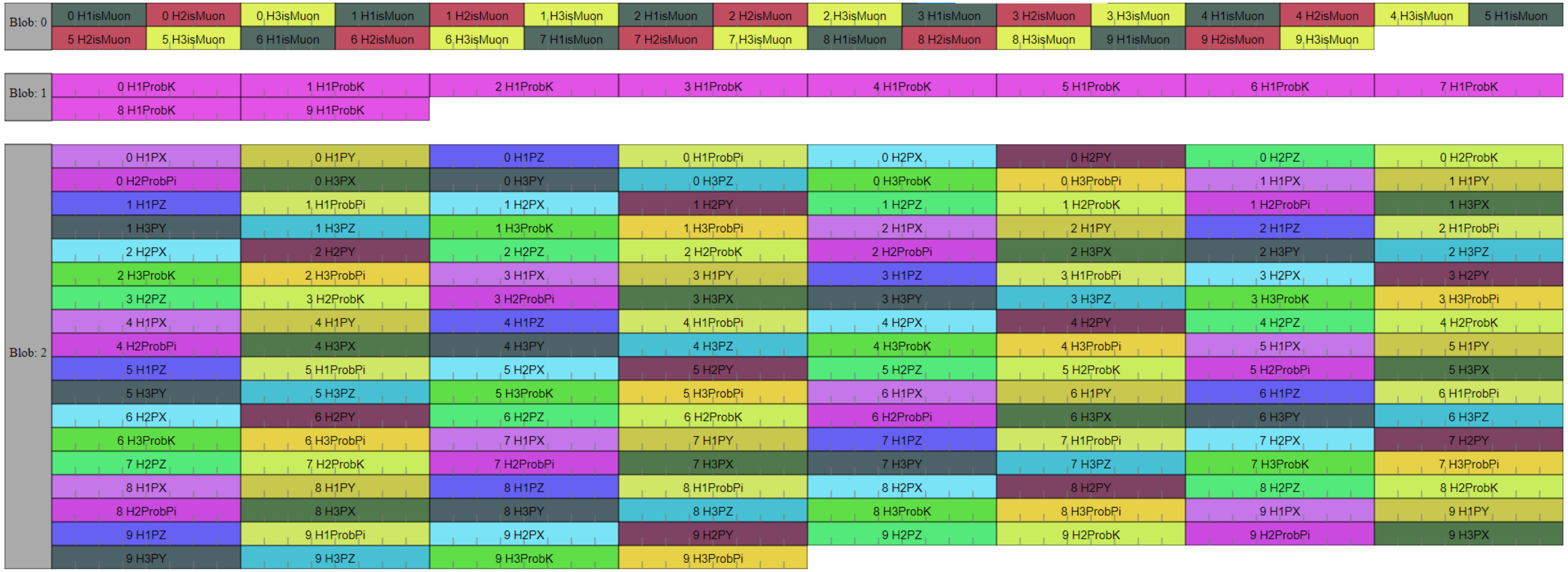


Blob: 0	0 H1isMuon 0 H2isMuon	0 H3isMuon	0 H1ProbK	1 H1isMuon 1 H2isMuon	1 H3isMuon	1 H1ProbK	2 H1isMuon 2 H2isMuon	2 H3isMuon
	2 H1ProbK	3 H1isMuon 3 H2isMuon	3 H3isMuon	3 H1ProbK	4 H1isMuon 4 H2isMuon	4 H3isMuon	4 H1ProbK	5 H1isMuon 5 H2isMuon
	5 H3isMuon	5 H1ProbK	6 H1isMuon 6 H2isMuon	6 H3isMuon	6 H1ProbK	7 H1isMuon 7 H2isMuon	7 H3isMuon	7 H1ProbK
	8 H1isMuon 8 H2isMuon	8 H3isMuon	8 H1ProbK	9 H1isMuon 9 H2isMuon	9 H3isMuon	9 H1ProbK		

Blob: 1	0 H1PX	0 H1PY	0 H1PZ	0 H1ProbPi	0 H2PX	0 H2PY	0 H2PZ	0 H2ProbK
	0 H2ProbPi	0 H3PX	0 H3PY	0 H3PZ	0 H3ProbK	0 H3ProbPi	1 H1PX	1 H1PY
	1 H1PZ	1 H1ProbPi	1 H2PX	1 H2PY	1 H2PZ	1 H2ProbK	1 H2ProbPi	1 H3PX
	1 H3PY	1 H3PZ	1 H3ProbK	1 H3ProbPi	2 H1PX	2 H1PY	2 H1PZ	2 H1ProbPi
	2 H2PX	2 H2PY	2 H2PZ	2 H2ProbK	2 H2ProbPi	2 H3PX	2 H3PY	2 H3PZ
	2 H3ProbK	2 H3ProbPi	3 H1PX	3 H1PY	3 H1PZ	3 H1ProbPi	3 H2PX	3 H2PY
	3 H2PZ	3 H2ProbK	3 H2ProbPi	3 H3PX	3 H3PY	3 H3PZ	3 H3ProbK	3 H3ProbPi
	4 H1PX	4 H1PY	4 H1PZ	4 H1ProbPi	4 H2PX	4 H2PY	4 H2PZ	4 H2ProbK
	4 H2ProbPi	4 H3PX	4 H3PY	4 H3PZ	4 H3ProbK	4 H3ProbPi	5 H1PX	5 H1PY
	5 H1PZ	5 H1ProbPi	5 H2PX	5 H2PY	5 H2PZ	5 H2ProbK	5 H2ProbPi	5 H3PX
	5 H3PY	5 H3PZ	5 H3ProbK	5 H3ProbPi	6 H1PX	6 H1PY	6 H1PZ	6 H1ProbPi
	6 H2PX	6 H2PY	6 H2PZ	6 H2ProbK	6 H2ProbPi	6 H3PX	6 H3PY	6 H3PZ
	6 H3ProbK	6 H3ProbPi	7 H1PX	7 H1PY	7 H1PZ	7 H1ProbPi	7 H2PX	7 H2PY
	7 H2PZ	7 H2ProbK	7 H2ProbPi	7 H3PX	7 H3PY	7 H3PZ	7 H3ProbK	7 H3ProbPi
	8 H1PX	8 H1PY	8 H1PZ	8 H1ProbPi	8 H2PX	8 H2PY	8 H2PZ	8 H2ProbK
	8 H2ProbPi	8 H3PX	8 H3PY	8 H3PZ	8 H3ProbK	8 H3ProbPi	9 H1PX	9 H1PY
	9 H1PZ	9 H1ProbPi	9 H2PX	9 H2PY	9 H2PZ	9 H2ProbK	9 H2ProbPi	9 H3PX
	9 H3PY	9 H3PZ	9 H3ProbK	9 H3ProbPi				

Designing a memory layout 2/7

- Custom layout 2:
  - Separate H1isMuon/H2isMuon/H3isMuon, keeping the 3 values close (AoS)
  - Separate H1PropK
  - Keep the remaining values close (AoS)



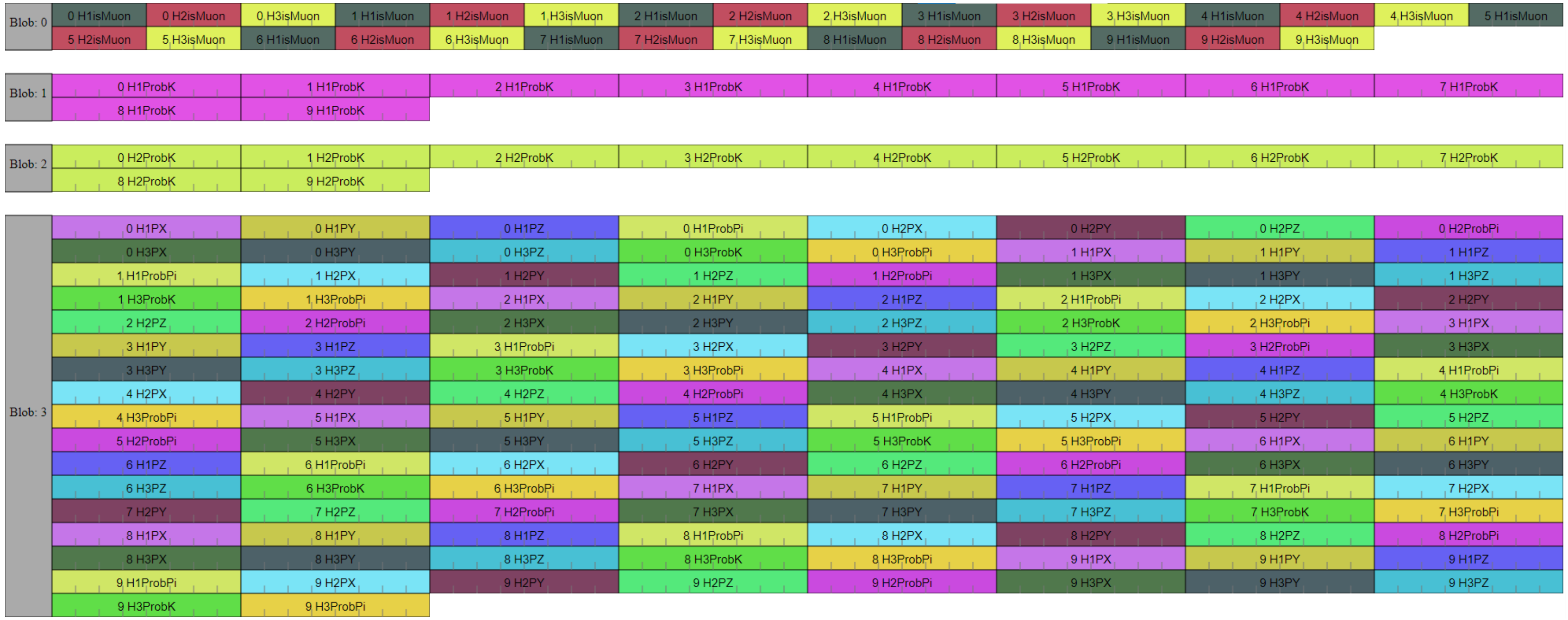
Blob: 0	0 H1isMuon	0 H2isMuon	0 H3isMuon	1 H1isMuon	1 H2isMuon	1 H3isMuon	2 H1isMuon	2 H2isMuon	2 H3isMuon	3 H1isMuon	3 H2isMuon	3 H3isMuon	4 H1isMuon	4 H2isMuon	4 H3isMuon	5 H1isMuon
	5 H2isMuon	5 H3isMuon	6 H1isMuon	6 H2isMuon	6 H3isMuon	7 H1isMuon	7 H2isMuon	7 H3isMuon	8 H1isMuon	8 H2isMuon	8 H3isMuon	9 H1isMuon	9 H2isMuon	9 H3isMuon		

Blob: 1	0 H1ProbK	1 H1ProbK	2 H1ProbK	3 H1ProbK	4 H1ProbK	5 H1ProbK	6 H1ProbK	7 H1ProbK
	8 H1ProbK	9 H1ProbK						

Blob: 2	0 H1PX	0 H1PY	0 H1PZ	0 H1ProbPi	0 H2PX	0 H2PY	0 H2PZ	0 H2ProbK
	0 H2ProbPi	0 H3PX	0 H3PY	0 H3PZ	0 H3ProbK	0 H3ProbPi	1 H1PX	1 H1PY
	1 H1PZ	1 H1ProbPi	1 H2PX	1 H2PY	1 H2PZ	1 H2ProbK	1 H2ProbPi	1 H3PX
	1 H3PY	1 H3PZ	1 H3ProbK	1 H3ProbPi	2 H1PX	2 H1PY	2 H1PZ	2 H1ProbPi
	2 H2PX	2 H2PY	2 H2PZ	2 H2ProbK	2 H2ProbPi	2 H3PX	2 H3PY	2 H3PZ
	2 H3ProbK	2 H3ProbPi	3 H1PX	3 H1PY	3 H1PZ	3 H1ProbPi	3 H2PX	3 H2PY
	3 H2PZ	3 H2ProbK	3 H2ProbPi	3 H3PX	3 H3PY	3 H3PZ	3 H3ProbK	3 H3ProbPi
	4 H1PX	4 H1PY	4 H1PZ	4 H1ProbPi	4 H2PX	4 H2PY	4 H2PZ	4 H2ProbK
	4 H2ProbPi	4 H3PX	4 H3PY	4 H3PZ	4 H3ProbK	4 H3ProbPi	5 H1PX	5 H1PY
	5 H1PZ	5 H1ProbPi	5 H2PX	5 H2PY	5 H2PZ	5 H2ProbK	5 H2ProbPi	5 H3PX
	5 H3PY	5 H3PZ	5 H3ProbK	5 H3ProbPi	6 H1PX	6 H1PY	6 H1PZ	6 H1ProbPi
	6 H2PX	6 H2PY	6 H2PZ	6 H2ProbK	6 H2ProbPi	6 H3PX	6 H3PY	6 H3PZ
	6 H3ProbK	6 H3ProbPi	7 H1PX	7 H1PY	7 H1PZ	7 H1ProbPi	7 H2PX	7 H2PY
	7 H2PZ	7 H2ProbK	7 H2ProbPi	7 H3PX	7 H3PY	7 H3PZ	7 H3ProbK	7 H3ProbPi
	8 H1PX	8 H1PY	8 H1PZ	8 H1ProbPi	8 H2PX	8 H2PY	8 H2PZ	8 H2ProbK
	8 H2ProbPi	8 H3PX	8 H3PY	8 H3PZ	8 H3ProbK	8 H3ProbPi	9 H1PX	9 H1PY
	9 H1PZ	9 H1ProbPi	9 H2PX	9 H2PY	9 H2PZ	9 H2ProbK	9 H2ProbPi	9 H3PX
	9 H3PY	9 H3PZ	9 H3ProbK	9 H3ProbPi				

Designing a memory layout 3/7

- Custom layout 3:
  - Separate H1isMuon/H2isMuon/H3isMuon, keeping the 3 values close (AoS)
  - Separate H1PropK and H2PropK into their own arrays
  - Keep the remaining values close (AoS)



Blob: 0	0 H1isMuon	0 H2isMuon	0 H3isMuon	1 H1isMuon	1 H2isMuon	1 H3isMuon	2 H1isMuon	2 H2isMuon	2 H3isMuon	3 H1isMuon	3 H2isMuon	3 H3isMuon	4 H1isMuon	4 H2isMuon	4 H3isMuon	5 H1isMuon
	5 H2isMuon	5 H3isMuon	6 H1isMuon	6 H2isMuon	6 H3isMuon	7 H1isMuon	7 H2isMuon	7 H3isMuon	8 H1isMuon	8 H2isMuon	8 H3isMuon	9 H1isMuon	9 H2isMuon	9 H3isMuon		

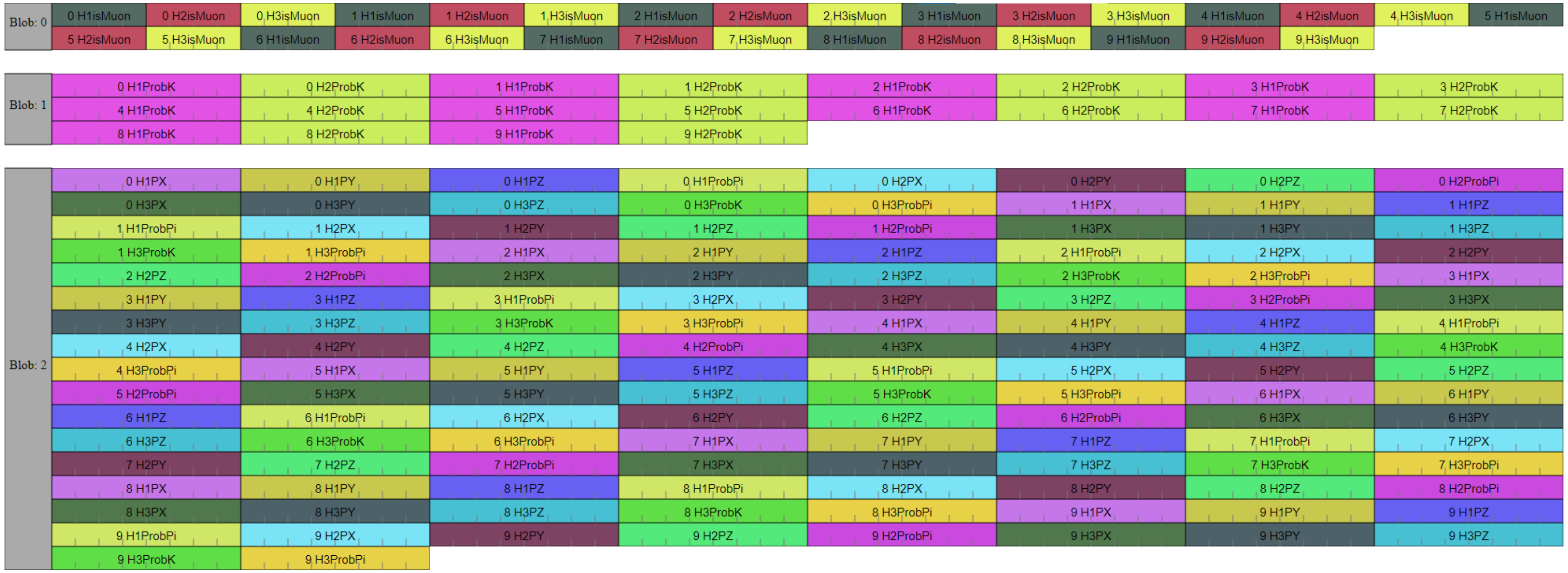
Blob: 1	0 H1ProbK	1 H1ProbK	2 H1ProbK	3 H1ProbK	4 H1ProbK	5 H1ProbK	6 H1ProbK	7 H1ProbK
	8 H1ProbK	9 H1ProbK						

Blob: 2	0 H2ProbK	1 H2ProbK	2 H2ProbK	3 H2ProbK	4 H2ProbK	5 H2ProbK	6 H2ProbK	7 H2ProbK
	8 H2ProbK	9 H2ProbK						

Blob: 3	0 H1PX	0 H1PY	0 H1PZ	0 H1ProbPi	0 H2PX	0 H2PY	0 H2PZ	0 H2ProbPi
	0 H3PX	0 H3PY	0 H3PZ	0 H3ProbK	0 H3ProbPi	1 H1PX	1 H1PY	1 H1PZ
	1 H1ProbPi	1 H2PX	1 H2PY	1 H2PZ	1 H2ProbPi	1 H3PX	1 H3PY	1 H3PZ
	1 H3ProbK	1 H3ProbPi	2 H1PX	2 H1PY	2 H1PZ	2 H1ProbPi	2 H2PX	2 H2PY
	2 H2PZ	2 H2ProbPi	2 H3PX	2 H3PY	2 H3PZ	2 H3ProbK	2 H3ProbPi	3 H1PX
	3 H1PY	3 H1PZ	3 H1ProbPi	3 H2PX	3 H2PY	3 H2PZ	3 H2ProbPi	3 H3PX
	3 H3PY	3 H3PZ	3 H3ProbK	3 H3ProbPi	4 H1PX	4 H1PY	4 H1PZ	4 H1ProbPi
	4 H2PX	4 H2PY	4 H2PZ	4 H2ProbPi	4 H3PX	4 H3PY	4 H3PZ	4 H3ProbK
	4 H3ProbPi	5 H1PX	5 H1PY	5 H1PZ	5 H1ProbPi	5 H2PX	5 H2PY	5 H2PZ
	5 H2ProbPi	5 H3PX	5 H3PY	5 H3PZ	5 H3ProbK	5 H3ProbPi	6 H1PX	6 H1PY
	6 H1PZ	6 H1ProbPi	6 H2PX	6 H2PY	6 H2PZ	6 H2ProbPi	6 H3PX	6 H3PY
	6 H3PZ	6 H3ProbK	6 H3ProbPi	7 H1PX	7 H1PY	7 H1PZ	7 H1ProbPi	7 H2PX
	7 H2PY	7 H2PZ	7 H2ProbPi	7 H3PX	7 H3PY	7 H3PZ	7 H3ProbK	7 H3ProbPi
	8 H1PX	8 H1PY	8 H1PZ	8 H1ProbPi	8 H2PX	8 H2PY	8 H2PZ	8 H2ProbPi
	8 H3PX	8 H3PY	8 H3PZ	8 H3ProbK	8 H3ProbPi	9 H1PX	9 H1PY	9 H1PZ
	9 H1ProbPi	9 H2PX	9 H2PY	9 H2PZ	9 H2ProbPi	9 H3PX	9 H3PY	9 H3PZ
	9 H3ProbK	9 H3ProbPi						

Designing a memory layout 4/7

- Custom layout 4:
  - Separate H1isMuon/H2isMuon/H3isMuon, keeping the 3 values close (AoS)
  - Separate H1PropK and H2PropK into a common array (AoS)
  - Keep the remaining values close (AoS)

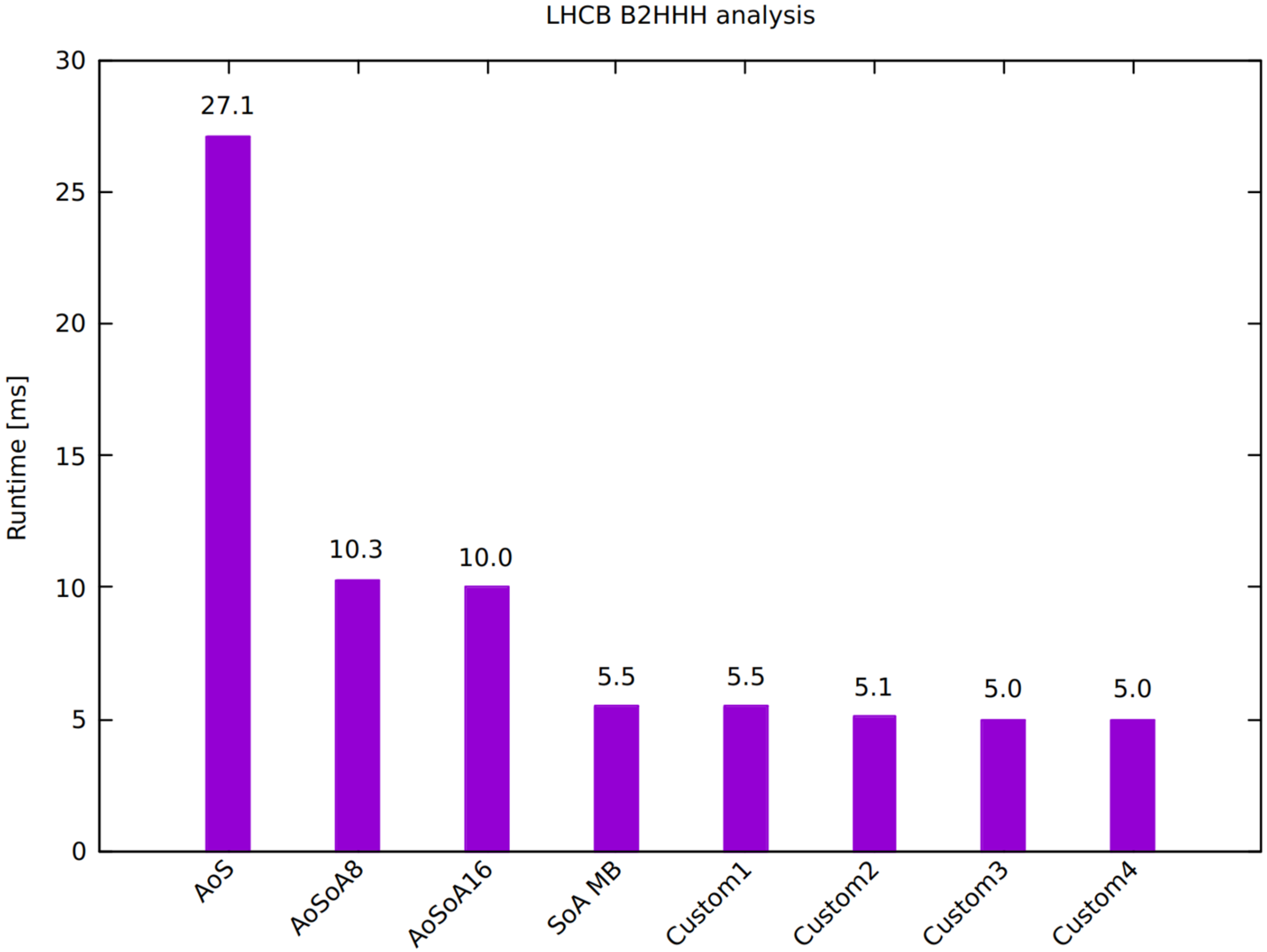


Blob: 0	0 H1isMuon	0 H2isMuon	0 H3isMuon	1 H1isMuon	1 H2isMuon	1 H3isMuon	2 H1isMuon	2 H2isMuon	2 H3isMuon	3 H1isMuon	3 H2isMuon	3 H3isMuon	4 H1isMuon	4 H2isMuon	4 H3isMuon	5 H1isMuon
	5 H2isMuon	5 H3isMuon	6 H1isMuon	6 H2isMuon	6 H3isMuon	7 H1isMuon	7 H2isMuon	7 H3isMuon	8 H1isMuon	8 H2isMuon	8 H3isMuon	9 H1isMuon	9 H2isMuon	9 H3isMuon		

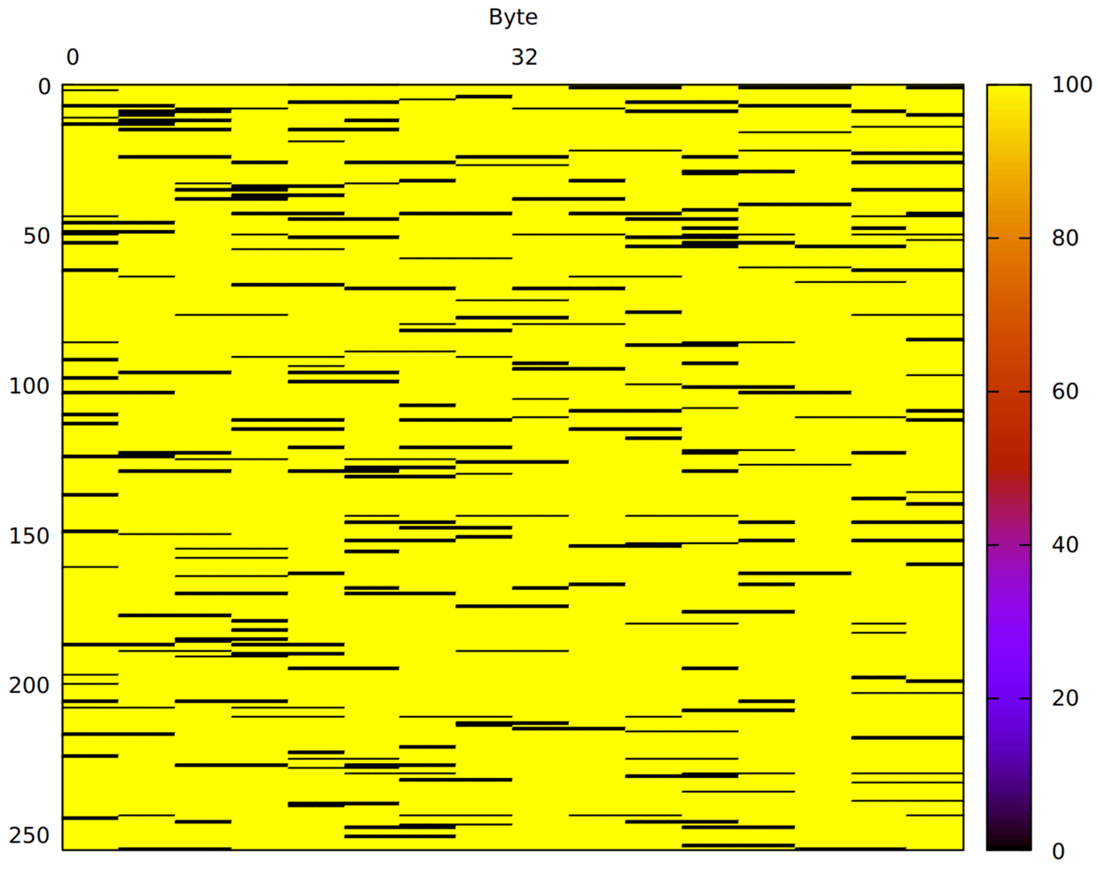
Blob: 1	0 H1ProbK	0 H2ProbK	1 H1ProbK	1 H2ProbK	2 H1ProbK	2 H2ProbK	3 H1ProbK	3 H2ProbK
	4 H1ProbK	4 H2ProbK	5 H1ProbK	5 H2ProbK	6 H1ProbK	6 H2ProbK	7 H1ProbK	7 H2ProbK
	8 H1ProbK	8 H2ProbK	9 H1ProbK	9 H2ProbK				

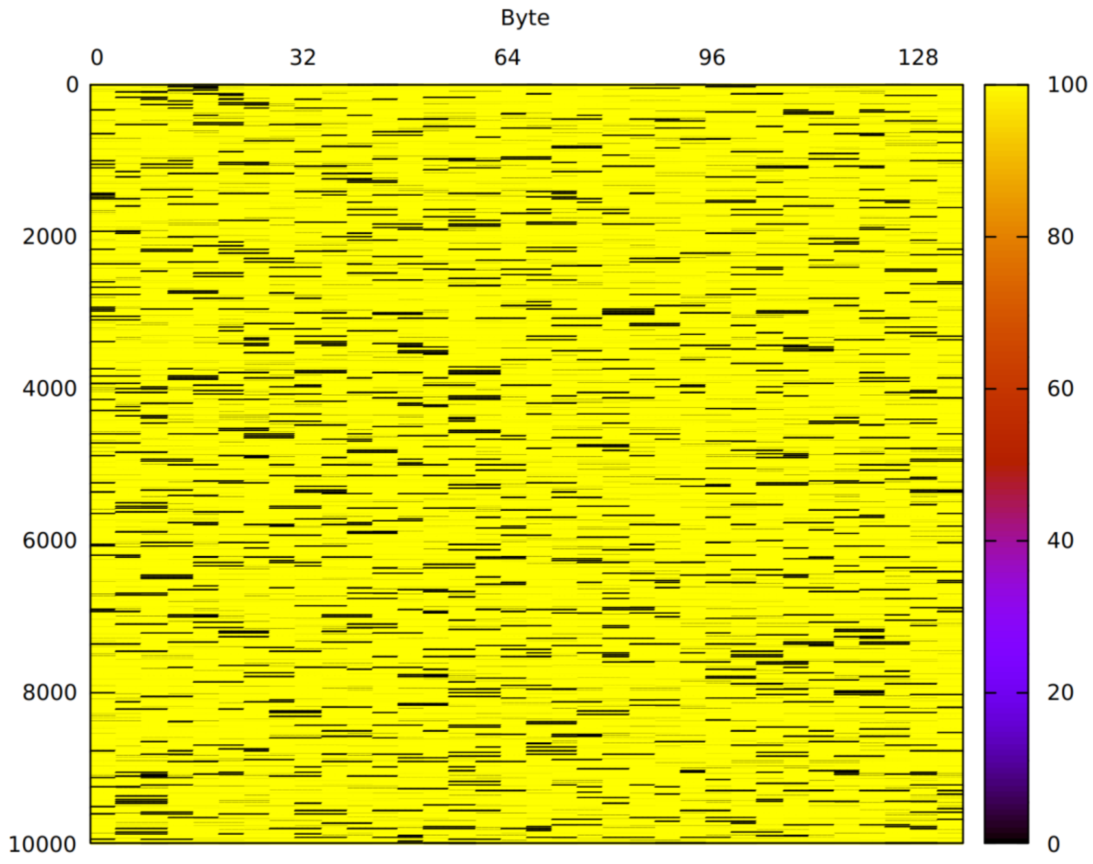
Blob: 2	0 H1PX	0 H1PY	0 H1PZ	0 H1ProbPi	0 H2PX	0 H2PY	0 H2PZ	0 H2ProbPi
	0 H3PX	0 H3PY	0 H3PZ	0 H3ProbK	0 H3ProbPi	1 H1PX	1 H1PY	1 H1PZ
	1 H1ProbPi	1 H2PX	1 H2PY	1 H2PZ	1 H2ProbPi	1 H3PX	1 H3PY	1 H3PZ
	1 H3ProbK	1 H3ProbPi	2 H1PX	2 H1PY	2 H1PZ	2 H1ProbPi	2 H2PX	2 H2PY
	2 H2PZ	2 H2ProbPi	2 H3PX	2 H3PY	2 H3PZ	2 H3ProbK	2 H3ProbPi	3 H1PX
	3 H1PY	3 H1PZ	3 H1ProbPi	3 H2PX	3 H2PY	3 H2PZ	3 H2ProbPi	3 H3PX
	3 H3PY	3 H3PZ	3 H3ProbK	3 H3ProbPi	4 H1PX	4 H1PY	4 H1PZ	4 H1ProbPi
	4 H2PX	4 H2PY	4 H2PZ	4 H2ProbPi	4 H3PX	4 H3PY	4 H3PZ	4 H3ProbK
	4 H3ProbPi	5 H1PX	5 H1PY	5 H1PZ	5 H1ProbPi	5 H2PX	5 H2PY	5 H2PZ
	5 H2ProbPi	5 H3PX	5 H3PY	5 H3PZ	5 H3ProbK	5 H3ProbPi	6 H1PX	6 H1PY
	6 H1PZ	6 H1ProbPi	6 H2PX	6 H2PY	6 H2PZ	6 H2ProbPi	6 H3PX	6 H3PY
	6 H3PZ	6 H3ProbK	6 H3ProbPi	7 H1PX	7 H1PY	7 H1PZ	7 H1ProbPi	7 H2PX
	7 H2PY	7 H2PZ	7 H2ProbPi	7 H3PX	7 H3PY	7 H3PZ	7 H3ProbK	7 H3ProbPi
	8 H1PX	8 H1PY	8 H1PZ	8 H1ProbPi	8 H2PX	8 H2PY	8 H2PZ	8 H2ProbPi
	8 H3PX	8 H3PY	8 H3PZ	8 H3ProbK	8 H3ProbPi	9 H1PX	9 H1PY	9 H1PZ
	9 H1ProbPi	9 H2PX	9 H2PY	9 H2PZ	9 H2ProbPi	9 H3PX	9 H3PY	9 H3PZ
	9 H3ProbK	9 H3ProbPi						

Benchmark


LHCB B2HHH analysis
Runtime [ms]
0
5
10
15
20
25
30
27.1
10.3
10.0
5.5
5.5
5.1
5.0
5.0
AoS
AoSoA8
AoSoA16
SoA MB
Custom1
Custom2
Custom3
Custom4

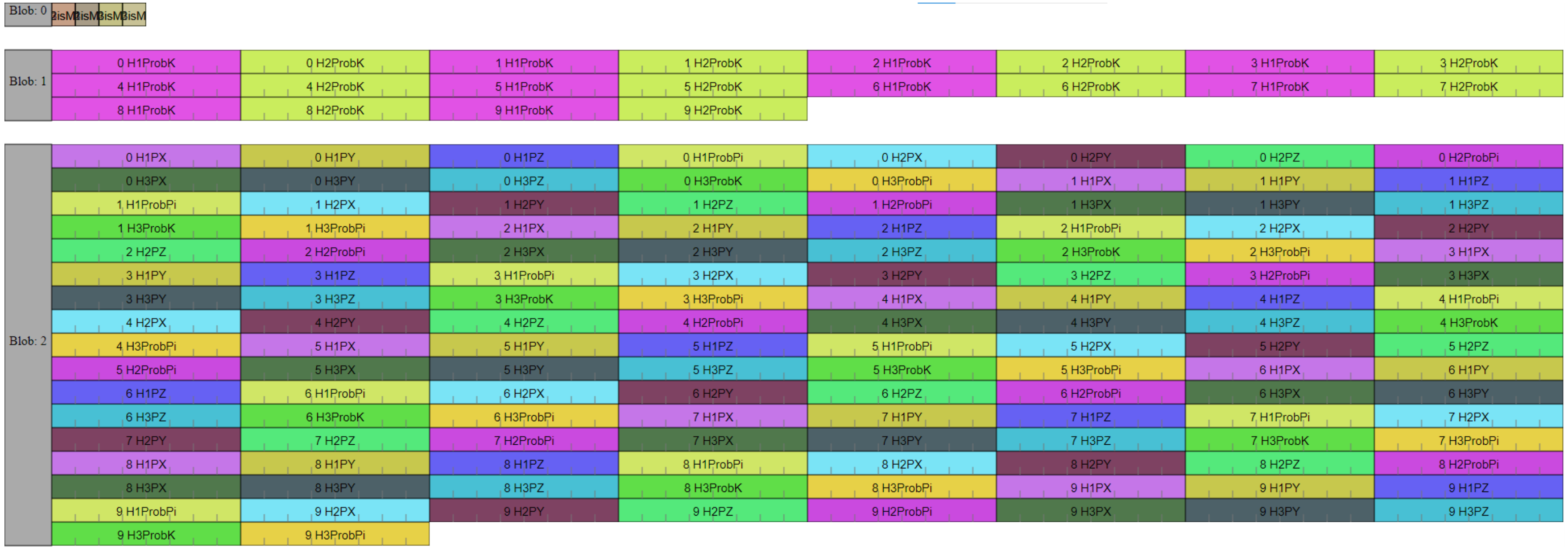
Btw: heatmaps for custom layout 4





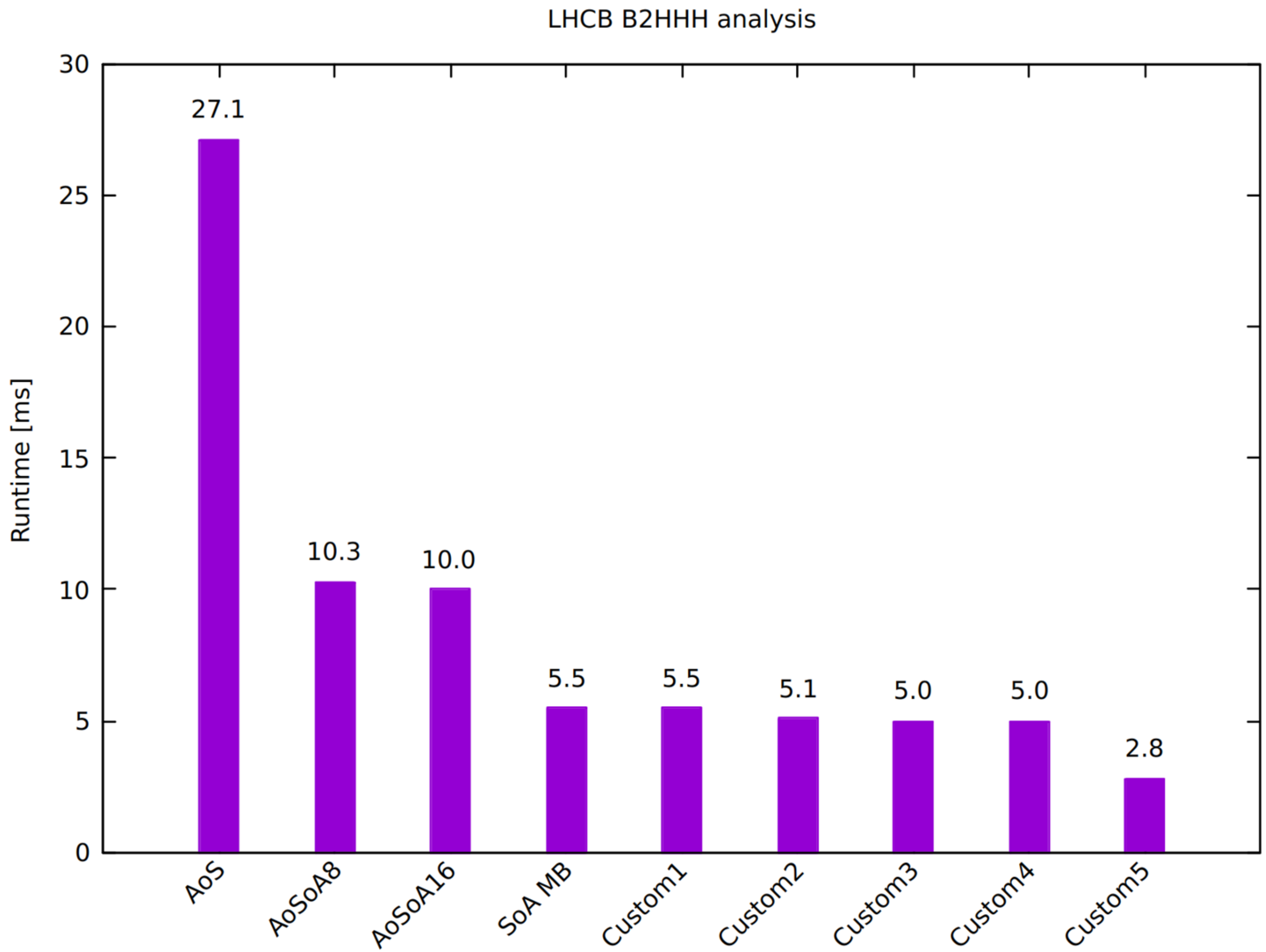
Designing a memory layout 5/7

- Know your data, again! H1isMuon/H2isMuon/H3isMuon are int32, but only store 0 or 1, we could pack those into bits!
- Custom layout 5:
  - Like custom layout 4, but pack the int32s to bits



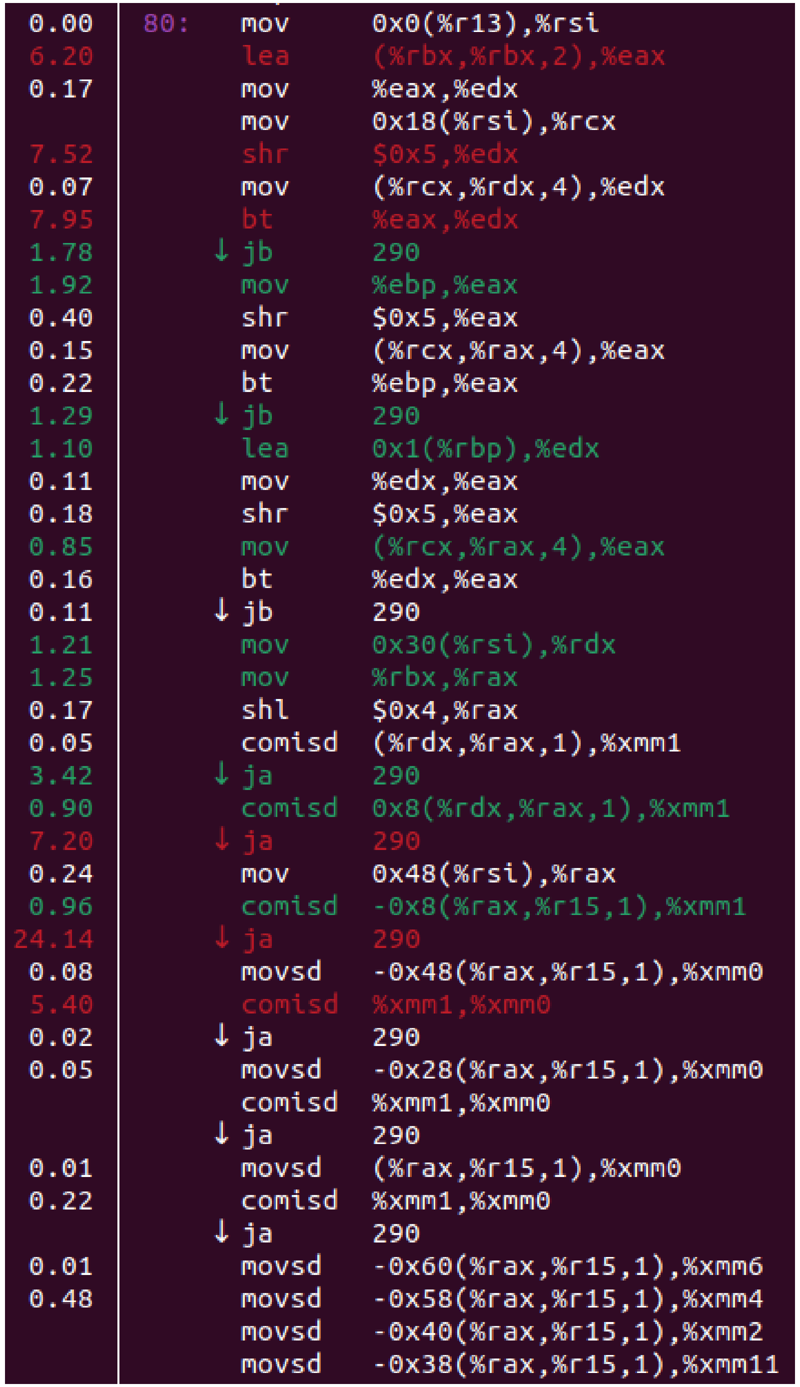
Blob: 0	2isM	2isM	3isM	3isM				
Blob: 1	0 H1ProbK	0 H2ProbK	1 H1ProbK	1 H2ProbK	2 H1ProbK	2 H2ProbK	3 H1ProbK	3 H2ProbK
	4 H1ProbK	4 H2ProbK	5 H1ProbK	5 H2ProbK	6 H1ProbK	6 H2ProbK	7 H1ProbK	7 H2ProbK
	8 H1ProbK	8 H2ProbK	9 H1ProbK	9 H2ProbK				
Blob: 2	0 H1PX	0 H1PY	0 H1PZ	0 H1ProbPi	0 H2PX	0 H2PY	0 H2PZ	0 H2ProbPi
	0 H3PX	0 H3PY	0 H3PZ	0 H3ProbK	0 H3ProbPi	1 H1PX	1 H1PY	1 H1PZ
	1 H1ProbPi	1 H2PX	1 H2PY	1 H2PZ	1 H2ProbPi	1 H3PX	1 H3PY	1 H3PZ
	1 H3ProbK	1 H3ProbPi	2 H1PX	2 H1PY	2 H1PZ	2 H1ProbPi	2 H2PX	2 H2PY
	2 H2PZ	2 H2ProbPi	2 H3PX	2 H3PY	2 H3PZ	2 H3ProbK	2 H3ProbPi	3 H1PX
	3 H1PY	3 H1PZ	3 H1ProbPi	3 H2PX	3 H2PY	3 H2PZ	3 H2ProbPi	3 H3PX
	3 H3PY	3 H3PZ	3 H3ProbK	3 H3ProbPi	4 H1PX	4 H1PY	4 H1PZ	4 H1ProbPi
	4 H2PX	4 H2PY	4 H2PZ	4 H2ProbPi	4 H3PX	4 H3PY	4 H3PZ	4 H3ProbK
	4 H3ProbPi	5 H1PX	5 H1PY	5 H1PZ	5 H1ProbPi	5 H2PX	5 H2PY	5 H2PZ
	5 H2ProbPi	5 H3PX	5 H3PY	5 H3PZ	5 H3ProbK	5 H3ProbPi	6 H1PX	6 H1PY
	6 H1PZ	6 H1ProbPi	6 H2PX	6 H2PY	6 H2PZ	6 H2ProbPi	6 H3PX	6 H3PY
	6 H3PZ	6 H3ProbK	6 H3ProbPi	7 H1PX	7 H1PY	7 H1PZ	7 H1ProbPi	7 H2PX
	7 H2PY	7 H2PZ	7 H2ProbPi	7 H3PX	7 H3PY	7 H3PZ	7 H3ProbK	7 H3ProbPi
	8 H1PX	8 H1PY	8 H1PZ	8 H1ProbPi	8 H2PX	8 H2PY	8 H2PZ	8 H2ProbPi
	8 H3PX	8 H3PY	8 H3PZ	8 H3ProbK	8 H3ProbPi	9 H1PX	9 H1PY	9 H1PZ
	9 H1ProbPi	9 H2PX	9 H2PY	9 H2PZ	9 H2ProbPi	9 H3PX	9 H3PY	9 H3PZ
	9 H3ProbK	9 H3ProbPi						

Benchmark


LHCB B2HHH analysis
Runtime [ms]
0
5
10
15
20
25
30
27.1
10.3
10.0
5.5
5.5
5.1
5.0
5.0
2.8
AoS
AoSoA8
AoSoA16
SoA MB
Custom1
Custom2
Custom3
Custom4
Custom5

What does perf say?

- Loads and bit-ops are hot (expected)
- There are 9 jumps
  - Just like our 9 filters
- Jump instructions are hot (what?)



```
  0.00 │ 80:   mov     0x0(%r13),%rsi
  6.20 │       lea     (%rbx,%rbx,2),%eax
  0.17 │       mov     %eax,%edx
       │       mov     0x18(%rsi),%rcx
  7.52 │       shr     $0x5,%edx
  0.07 │       mov     (%rcx,%rdx,4),%edx
  7.95 │       bt      %eax,%edx
  1.78 │     ↓ jb      290
  1.92 │       mov     %ebp,%eax
  0.40 │       shr     $0x5,%eax
  0.15 │       mov     (%rcx,%rax,4),%eax
  0.22 │       bt      %ebp,%eax
  1.29 │     ↓ jb      290
  1.10 │       lea     0x1(%rbp),%edx
  0.11 │       mov     %edx,%eax
  0.18 │       shr     $0x5,%eax
  0.85 │       mov     (%rcx,%rax,4),%eax
  0.16 │       bt      %edx,%eax
  0.11 │     ↓ jb      290
  1.21 │       mov     0x30(%rsi),%rdx
  1.25 │       mov     %rbx,%rax
  0.17 │       shl     $0x4,%rax
  0.05 │       comisd  (%rdx,%rax,1),%xmm1
  3.42 │     ↓ ja      290
  0.90 │       comisd  0x8(%rdx,%rax,1),%xmm1
  7.20 │     ↓ ja      290
  0.24 │       mov     0x48(%rsi),%rax
  0.96 │       comisd  -0x8(%rax,%r15,1),%xmm1
 24.14 │     ↓ ja      290
  0.08 │       movsd   -0x48(%rax,%r15,1),%xmm0
  5.40 │       comisd  %xmm1,%xmm0
  0.02 │     ↓ ja      290
  0.05 │       movsd   -0x28(%rax,%r15,1),%xmm0
       │       comisd  %xmm1,%xmm0
       │     ↓ ja      290
  0.01 │       movsd   (%rax,%r15,1),%xmm0
  0.22 │       comisd  %xmm1,%xmm0
       │     ↓ ja      290
  0.01 │       movsd   -0x60(%rax,%r15,1),%xmm6
  0.48 │       movsd   -0x58(%rax,%r15,1),%xmm4
       │       movsd   -0x40(%rax,%r15,1),%xmm2
       │       movsd   -0x38(%rax,%r15,1),%xmm11
```

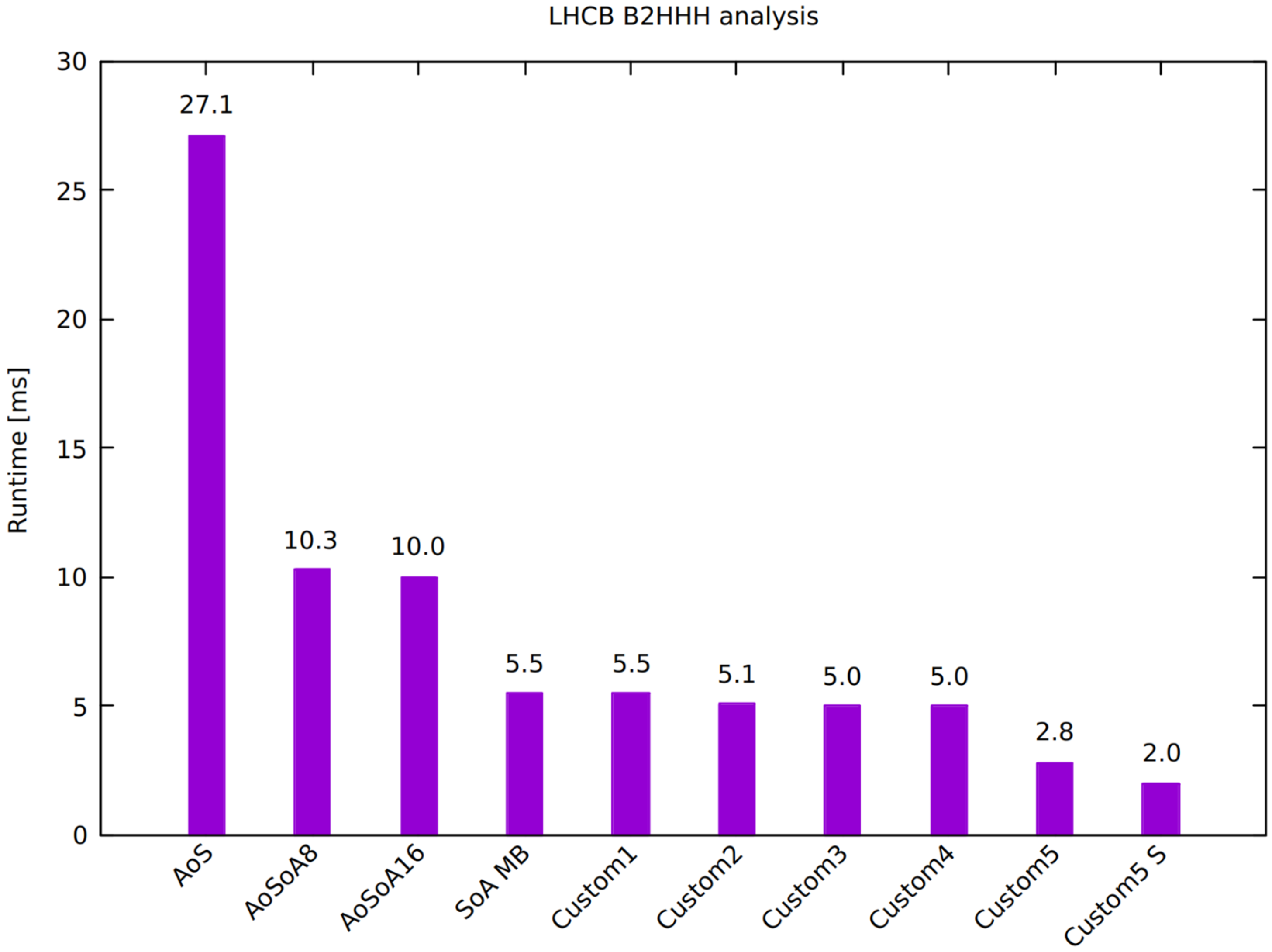
Branch prediction

- Most CPUs are pipelined
- CPU starts executing future instructions while current one is still in-flight
- When executing a branch, the CPU needs to guess where to continue (speculative execution)
- When the guess was wrong, CPU throws away a lot of work (pipeline flush)
    - Mispredicting a branch is expensive!
    - This is probably what we see here in our perf results
- Worst case: branch condition is random (like in our example)
- Best case: branch condition is always the same, or the same for a long time
- We can help the CPU by making branches more predictable
    - Easy solution: sort the data set based on branch conditions

Sorting dataset based on filter outcomes

```
template<typename View>
void sortView(View& v) {
  auto filterResults = [](const auto& e) {
    return std::tuple{
      e(H1isMuon{}), e(H2isMuon{}), e(H3isMuon{}),
      e(H1ProbK{}) < probKCut, ...,
      e(H1ProbPi{}) > probPiCut, ...};
  };
  std::sort(v.begin(), v.end(),
    [&](const auto& ea, const auto& eb) {
      return filterResults(ea) < filterResults(eb);
    });
}
```

Benchmark

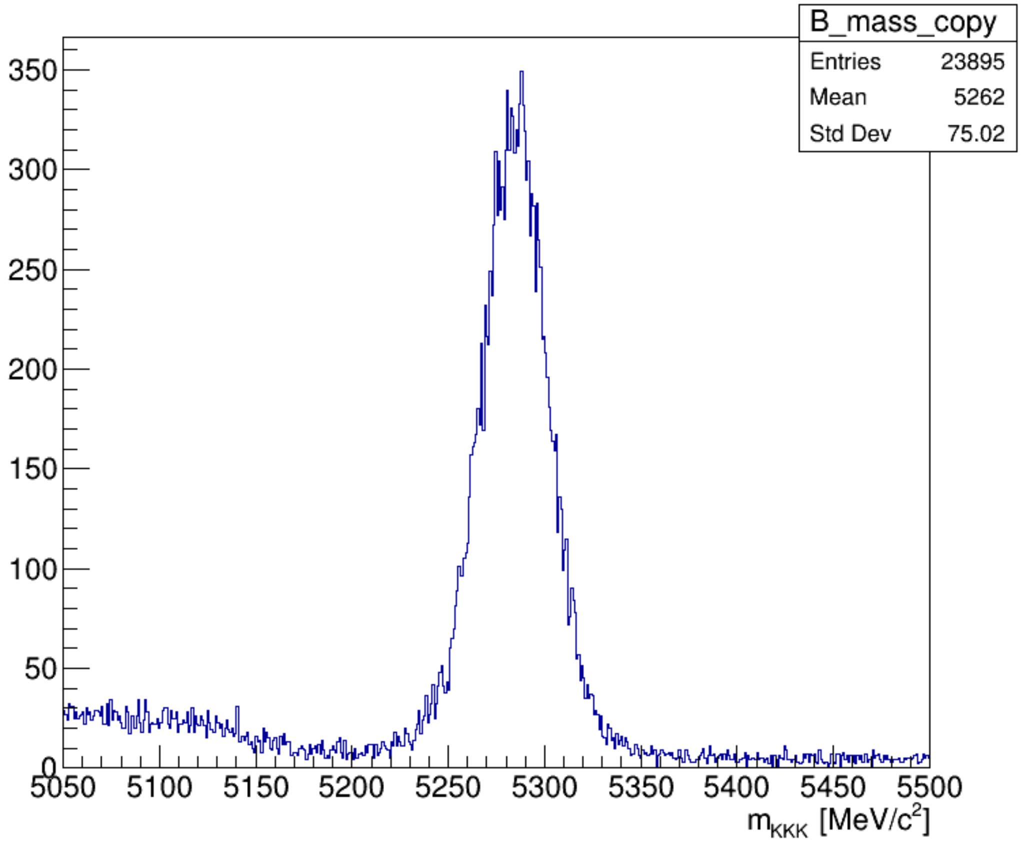


LHCB B2HHH analysis
Runtime [ms]
30
25
20
15
10
5
0
27.1
10.3
10.0
5.5
5.5
5.1
5.0
5.0
2.8
2.0
AoS
AoSoA8
AoSoA16
SoA MB
Custom1
Custom2
Custom3
Custom4
Custom5
Custom5 S

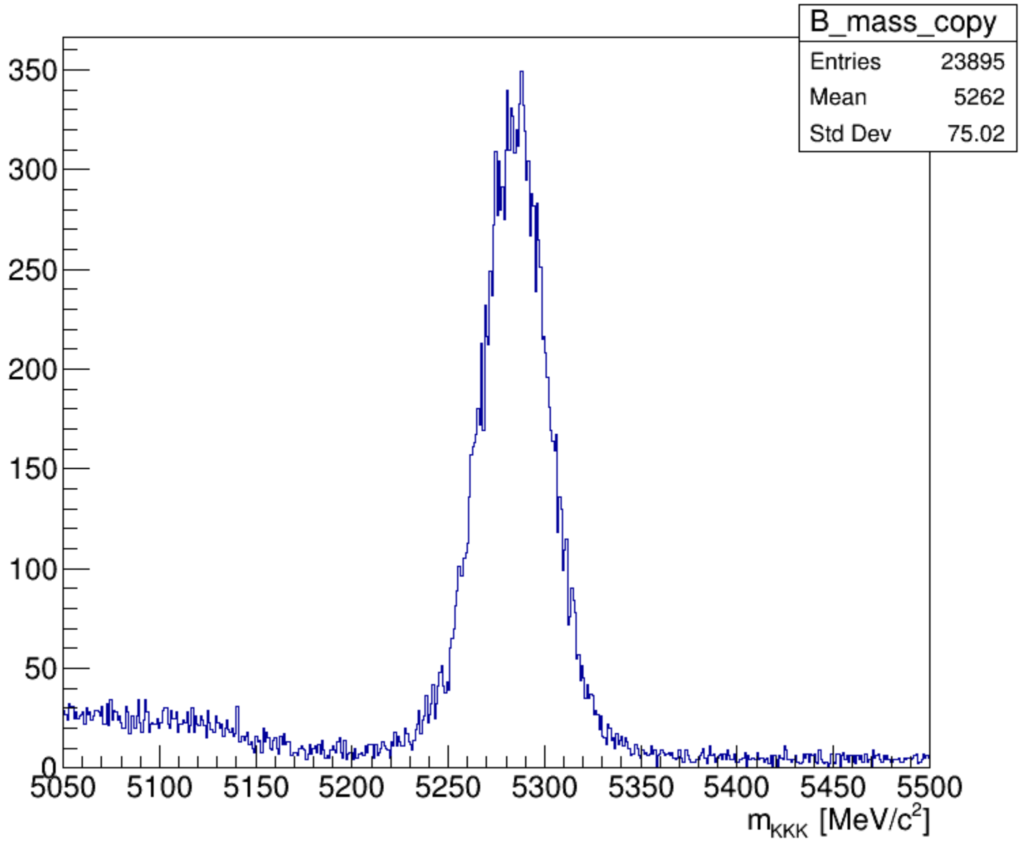
Reducing precision

- Execution speed of an analysis is not the only concern
- Vast amounts of experimental data is generated
- Data size matters as well, especially for storage
- How much can we reduce the value’s precision, before the result collapses?
- LLAMA can help here as well with corresponding mappings
    - Let’s run the analysis and plot some histograms

Reducing precision analysis results



BP SoA 52e11

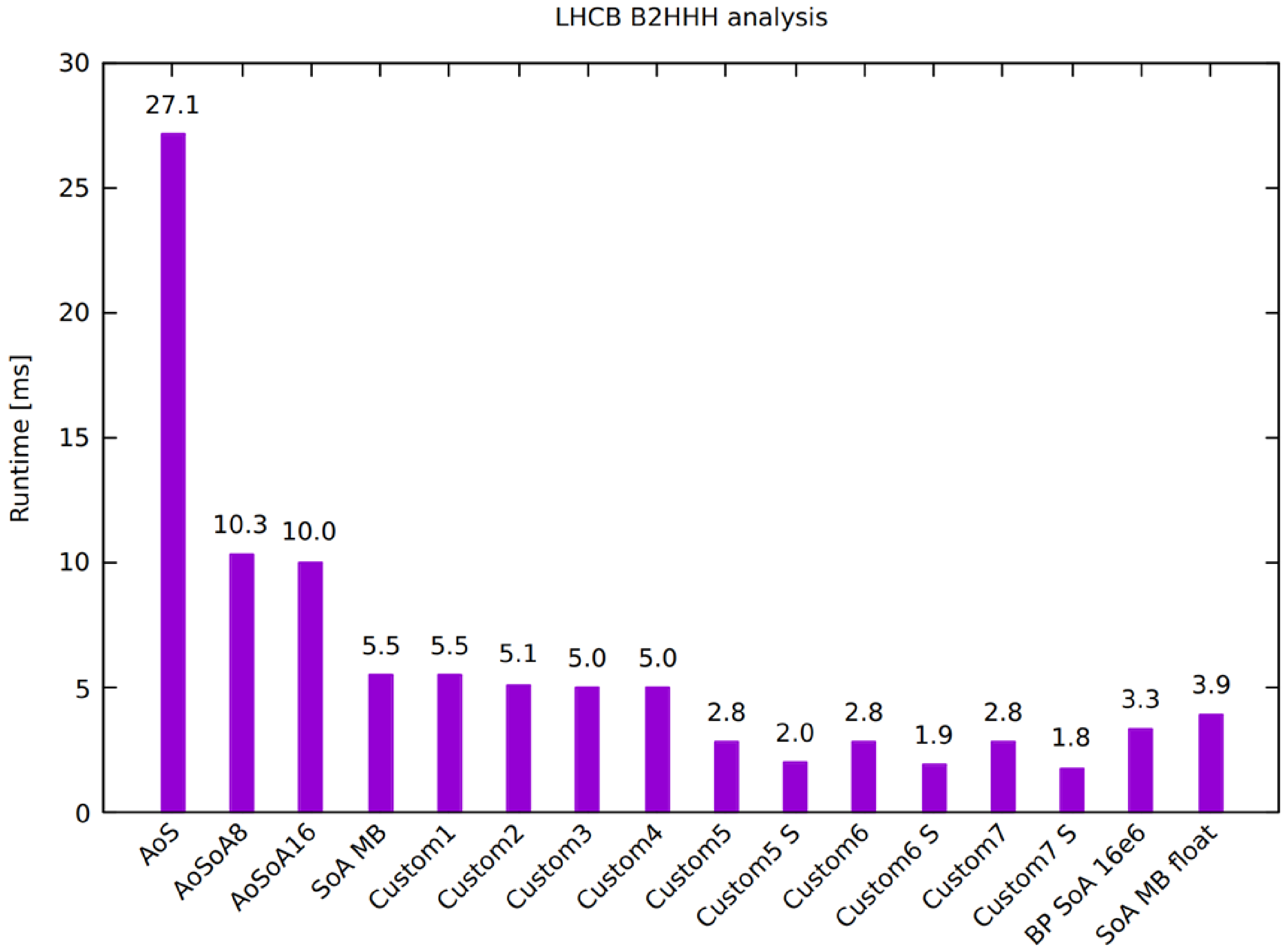


BP SoA 51e11

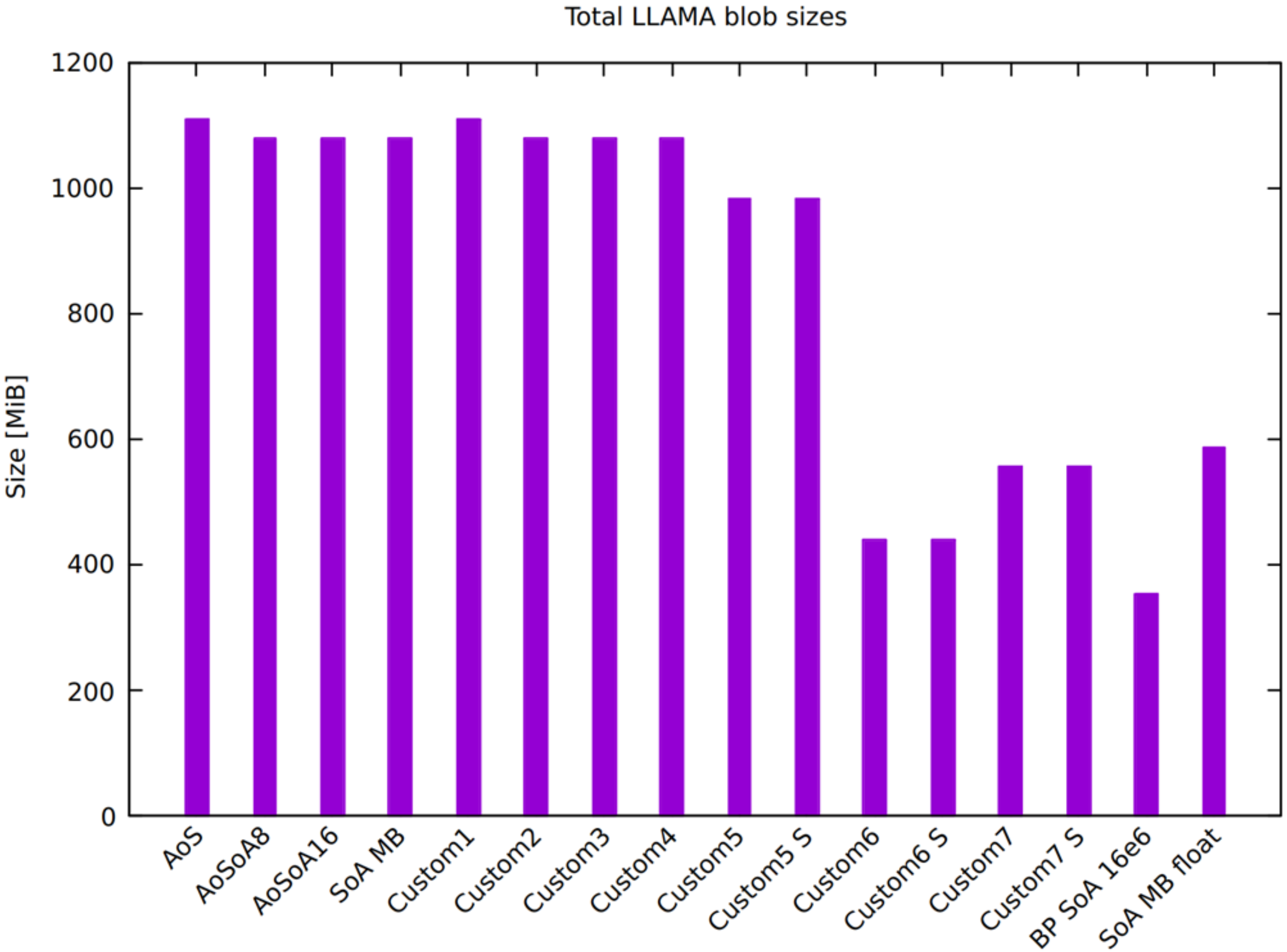
Designing a memory layout 6/6 and 7/6

- Know your data, again!
  - Floating point data may not need all bits for exponent and mantissa
  - Results start to get shaky at exponent < 6, and mantissa < 16
- Custom layout 6:
  - Separate H1isMuon/H2isMuon/H3isMuon, pack the int32s to bits, keeping the 3 values close (bitpacked AoS)
  - Separate H1PropK and H2PropK into a common arrays (AoS)
  - Keep the remaining values close per event, but reduce their precision
    - Either via bitpacking (bitpacked AoS)
    - Custom layout 7: Or by changing their data type (double -> float AoS)
- N.B.: bitpacking the H1PropK/H2PropK was too costly

Final benchmark



Memory consumption per layout



Summary and conclusions

- LLAMA allows for rapid layout exploration (after initial integration)
  - Common memory layouts readily available
- Know your data, and access pattern
  - LLAMA can help with layout visualization and access heatmaps
- Build custom memory layouts by combining existing ones
  - or even implement a memory mapping yourself
- Consider the range of values of your data to reduce precision/bits
- Changing data type is usually faster than arbitrary bit compression

Future work

- Data loaded for the analysis is only needed once. Caching is not needed. Can we benefit from non-temporal load instructions?
- The sorted data set puts all selected elements to the bottom of the view. The OpenMP static scheduling is probably not suited anymore.
- Calculate how much meaningful data must be loaded by the analysis
  - Based on filter branches and data element counts/sizes
- Measure how much data was actually pulled into CPU by the analysis
  - Might be an interesting metric in addition to runtime. Also independent of thread scheduling.

- Food for thought on RNTuple:
  - Is columnar (SoA) really the true layout for RNTuple? Data that is logically needed together should be kept together (e.g. a position's X, Y, Z)
  - Compressing Boolean values (also ints acting as such) to bits?
  - Arbitrary bit reductions for floating points?

Thank you!

Questions?