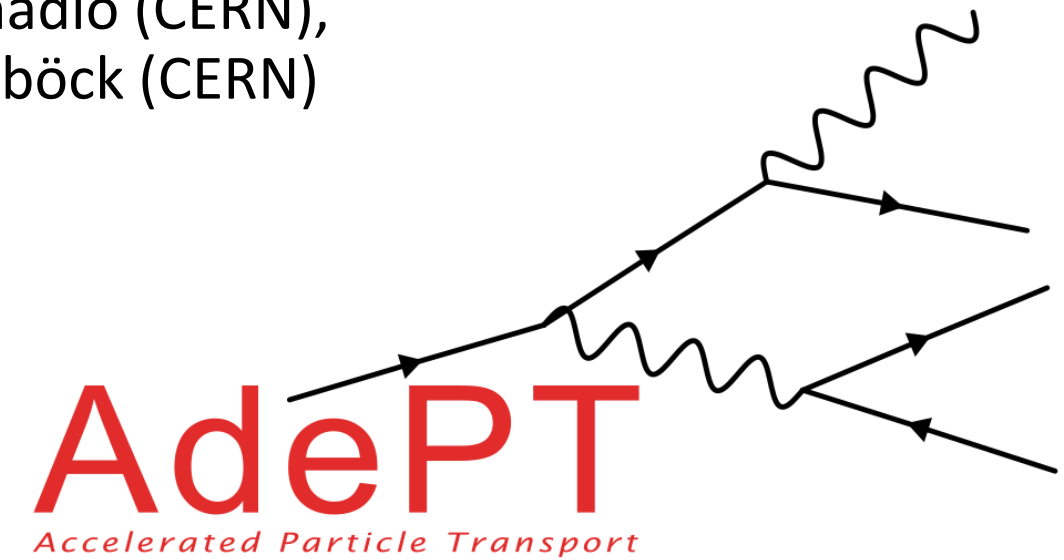
Challenges and opportunities integrating LLAMA into AdePT

Bernhard Manfred Gruber (CERN, CASUS, HZDR, TU Dresden),
Guilherme Amadio (CERN),
Stephan Hageböck (CERN)

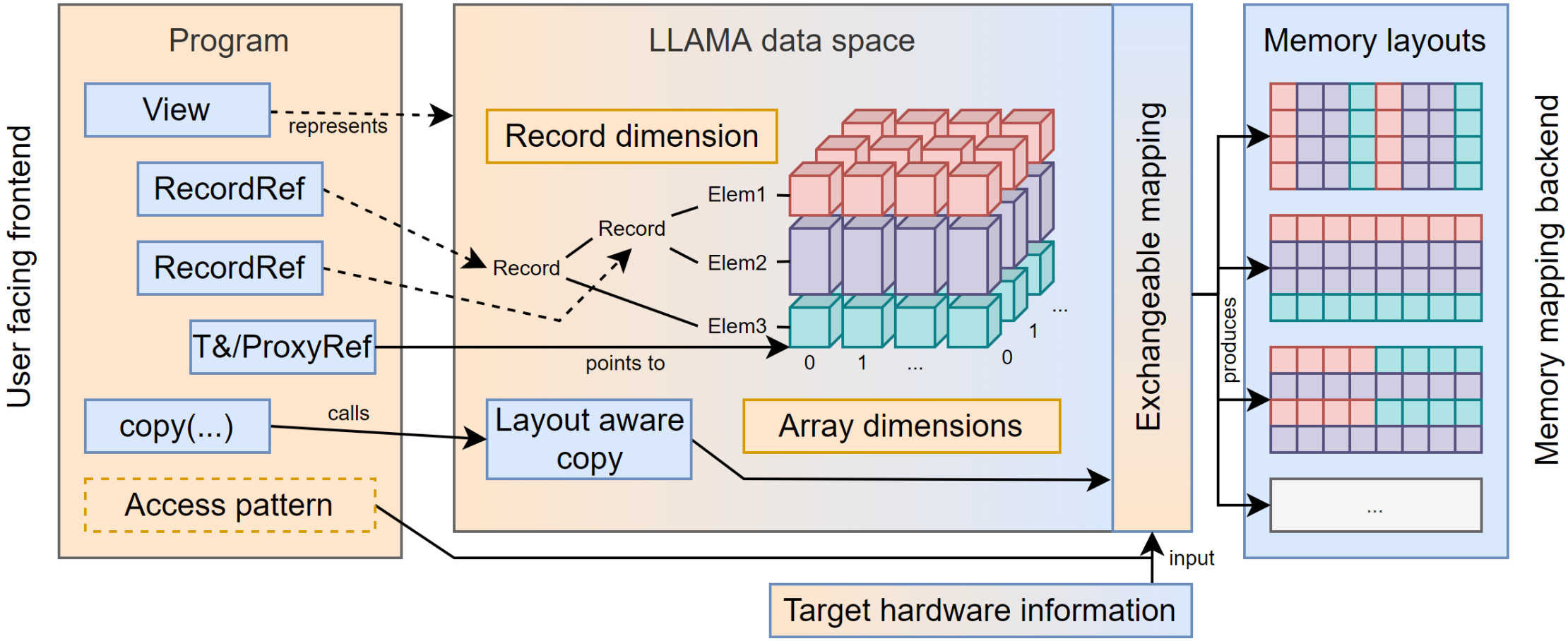


Low-Level Abstraction of Memory Access

- Motivation: Programs are increasingly memory-bound. Performance comes from full customization of data layout for each target architecture.
- Splits algorithmic view of data and mapping of the data 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 alpaka, CUDA/HIP, SYCL, …, but orthogonal
- GitHub: https://github.com/alpaka-group/llama

- Checkout our posters on alpaka and LLAMA at the poster session!

LLAMA concept



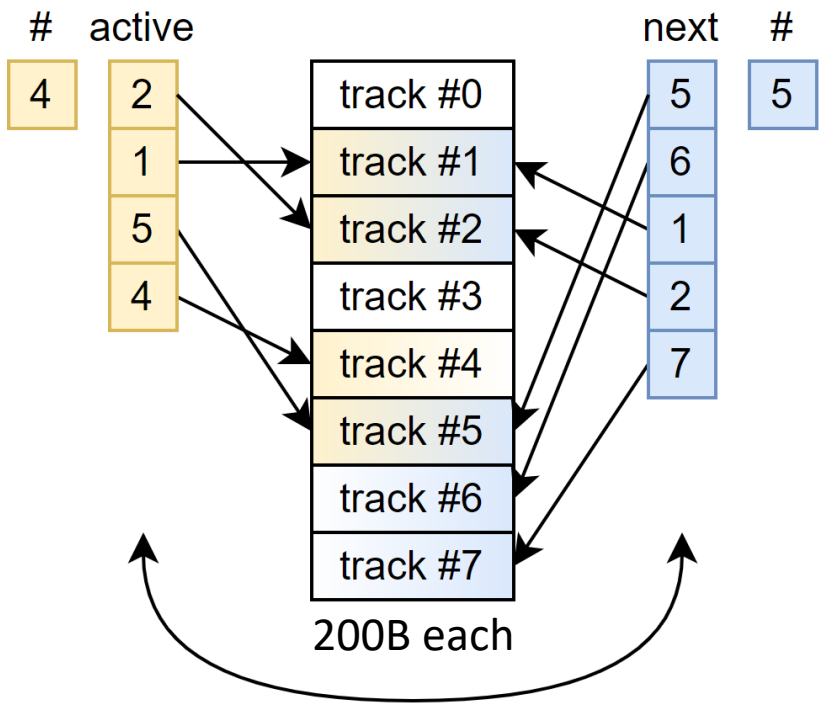
AdePT

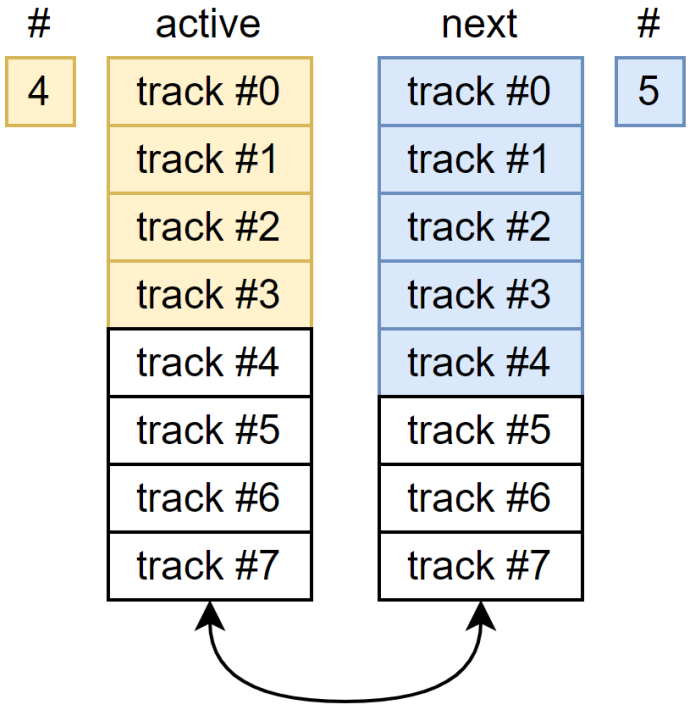
- Almost half of the compute workload in HEP is particle transport simulation
- AdePT is a C++/CUDA prototype for offloading EM transport simulations to GPUs
  - Can run standalone, or as module of Geant4 (fast simulation hook)
- Uses VecGeom for geometry (GDML loading, volumes, acceleration structure)
- Uses G4HepEm, a compact EM physics implementation
- AdePT is bound by memory access (Nsight Compute); an ideal testbed for LLAMA!

- Checkout our talk at 27th Geant4 Collaboration meeting: “AdePT status report and discussion”, and our talk and proceedings at ACAT21: “Offloading electromagnetic shower transport to GPUs: the AdePT project”
- GitHub: https://github.com/apt-sim/AdePT

AdePT track data structure

- Default: A sparse array of track structures (once per $e^-/e^+/\gamma$)
  - List of active slots and list of survivors/new particles for next iteration
- Experimental: Two dense arrays of track structures without slot arrays



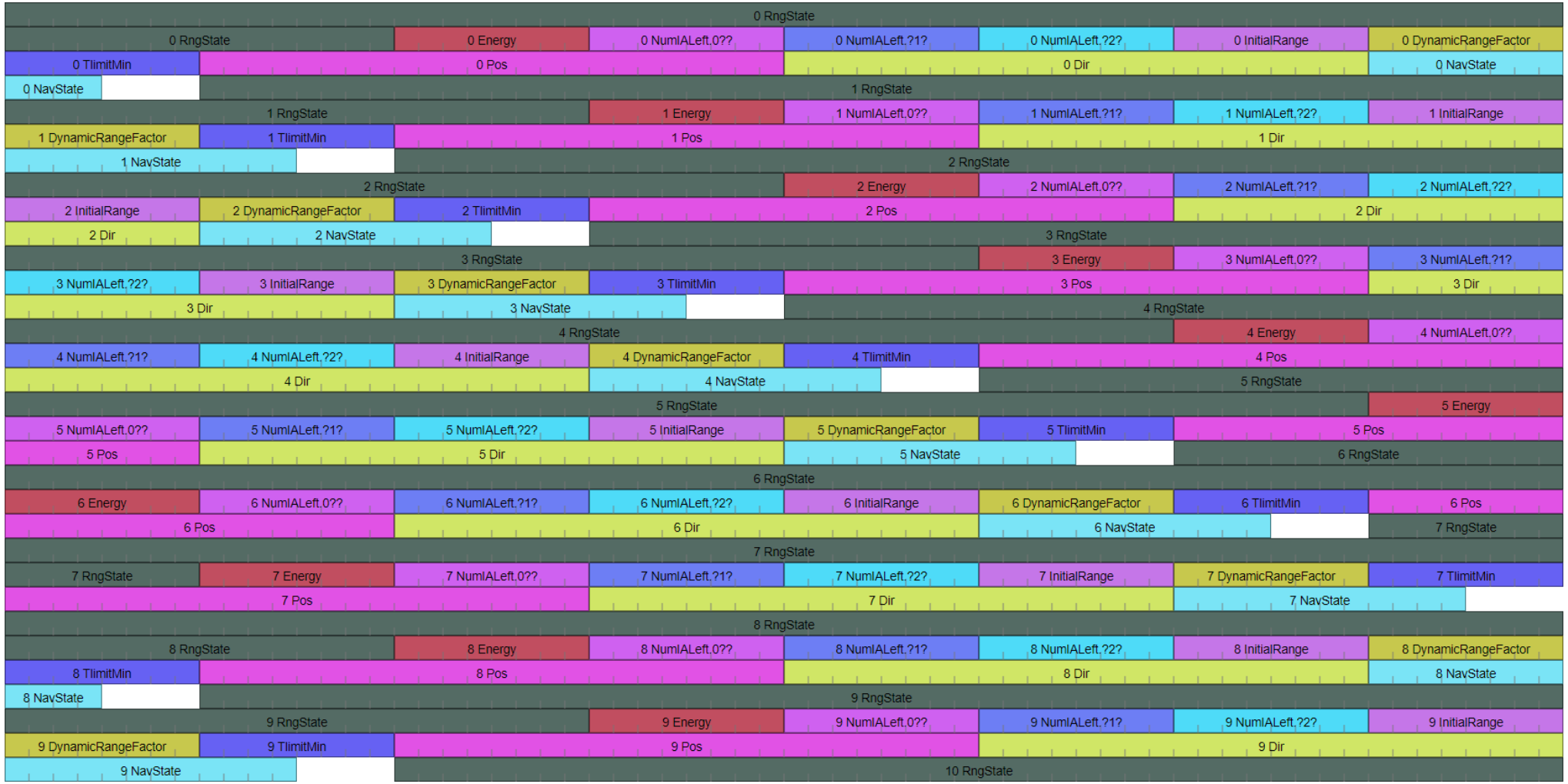


Track before and after LLAMA integration

```
struct Track {
  RanluxppDouble rngState;
  double energy;
  double numIALeft[3];
  double initialRange;
  double dynamicRangeFactor;
  double tlimitMin;
  vecgeom::Vector3D<Precision> pos;
  vecgeom::Vector3D<Precision> dir;
  vecgeom::NavStateIndex navState;


  __device__ void InitAsSecondary(
      const Track &parent) {

    // ...
    this->pos      = parent.pos;
    this->navState = parent.navState;
  }
};
```

```
struct RngState {}; struct Energy {}; // ...
using Track = llama::Record<
  llama::Field<RngState, RanluxppDouble>,
  llama::Field<Energy, double>,
  llama::Field<NumIALeft, double[3]>,
  llama::Field<InitialRange, double>,
  llama::Field<DynamicRangeFactor, double>,
  llama::Field<TlimitMin, double>,
  llama::Field<Pos, vecgeom::Vector3D<vecgeom::Precision>>,
  llama::Field<Dir, vecgeom::Vector3D<vecgeom::Precision>>,
  llama::Field<NavState, vecgeom::NavStateIndex>>;

template <typename SecondaryTrack>
__device__ void InitAsSecondary(SecondaryTrack &&track,
    const vecgeom::Vector3D<Precision> &parentPos,
    const vecgeom::NavStateIndex &parentNavState) {
  // ...
  track(Pos{})      = parentPos;
  track(NavState{}) = parentNavState;
}
```

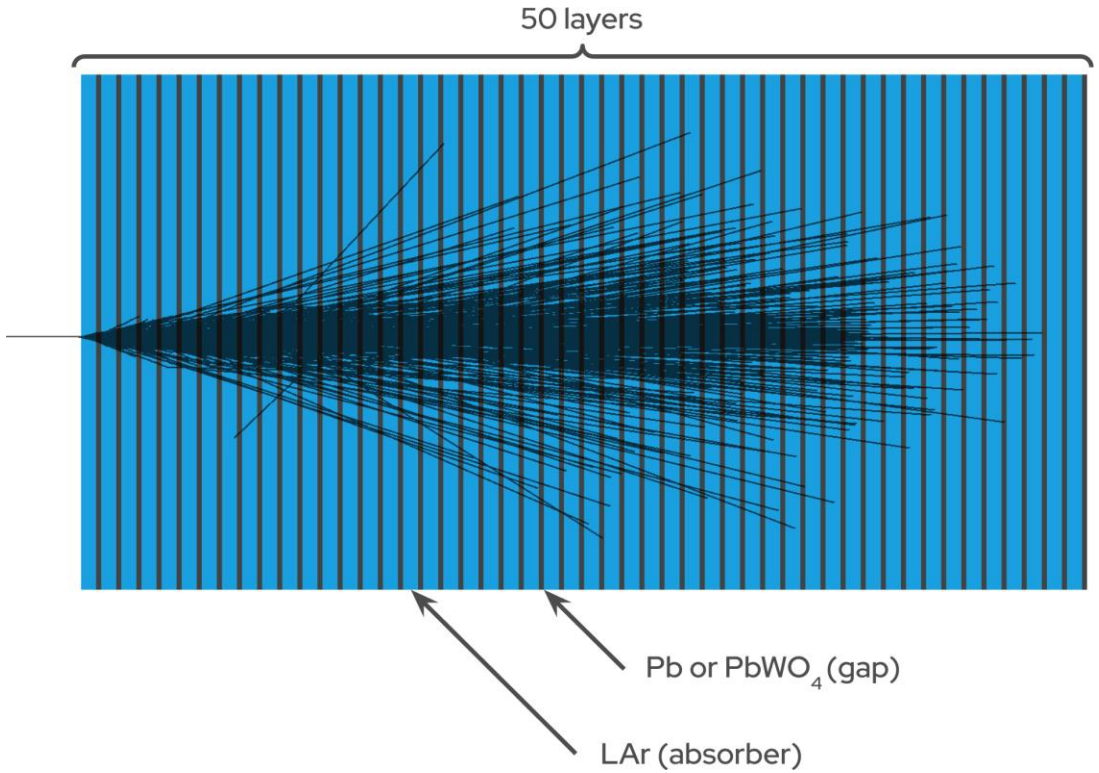
LLAMA's layout visualization (track array AoS)



Wrapped after 64B

Benchmark scenario: TestEm3

- Realistic enough test scenario
  - Physics implementation is complete → realistic compute workload
  - Track management offers interesting memory-layout optimization problem
- Simple geometry
  - Geometry code is not GPU-friendly yet
  - Optimization in VecGeom is pending (R&D on different surface models)
  - Avoid noise in the profiler



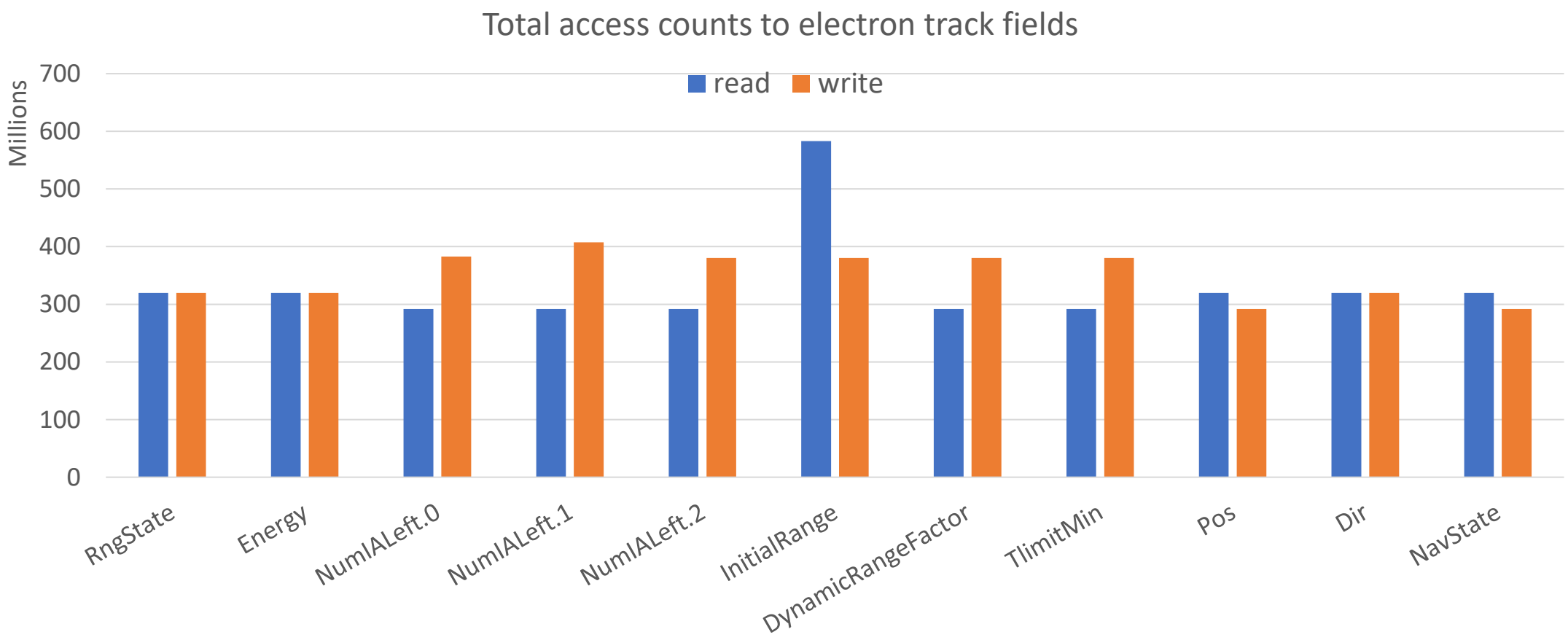
Benchmark settings

- All benchmarks are on an Nvidia V100S
  - 5120 CUDA cores (80 SMs), 1597MHz, 8.2 TFLOPS (DP), 32 GB HBM2/ECC, 1124 GB/S memory bandwidth, 250 W TPD
  - CentOS Stream 8, GCC 11, CUDA 11.7
  - VecCore 0.8.0, VecGeom 1.1.20, AdePT (git 449222d + branches)
- Reported numbers are average of 5 runs
  - $ example19 \
    –particles 10000 -batch 5000 \
    -gdml_file testEm3.gdml \
    -gunpos -220,0,0 -gundir 1,0,0

Memory access instrumentation with LLAMA

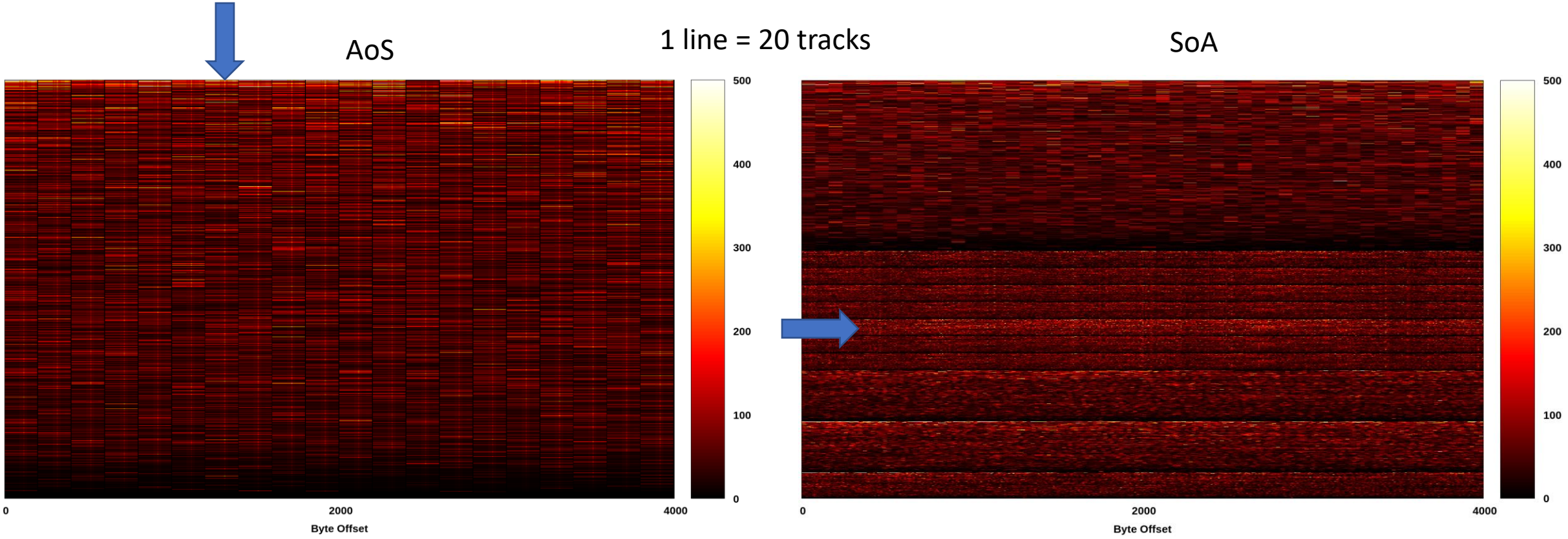
- LLAMA's memory mappings can be instrumented
  - Either count total read/writes per field (light), or per byte of memory (heavy)
  - Integrates effortlessly with user-defined memory mappings
- Counting is performed as side effect of data structure access
  - Cost: 1 atomic increment per access (AoS vs. traced AoS: 3.1x slowdown)
- Limitations of software instrumentation
  - We cannot observe what the hardware does
    - E.g., whether a memory read is served from VRAM or cache
  - We cannot observe what the compiler/optimizer does
    - E.g., whether a second memory read to the same memory location is optimized away
- Preliminary refactoring of your code can improve accuracy
  - E.g., replace repeated access to memory by a local variable

Lightweight access count tracing with LLAMA



Negligible memory overhead: 1 counter per field

Heatmap – sparse buffer – electrons



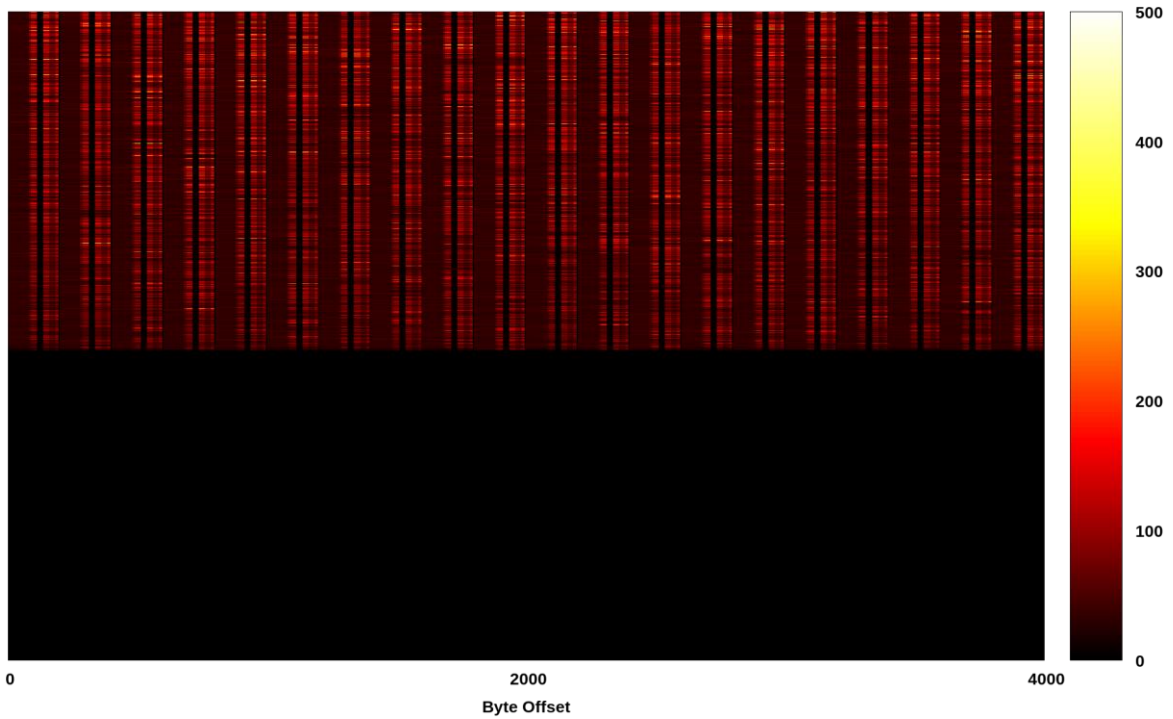
High memory overhead, 1 counter per byte. All heatmap runs with:
-particles 25 -batch 5 … (Hardcoded Capacity 50k)

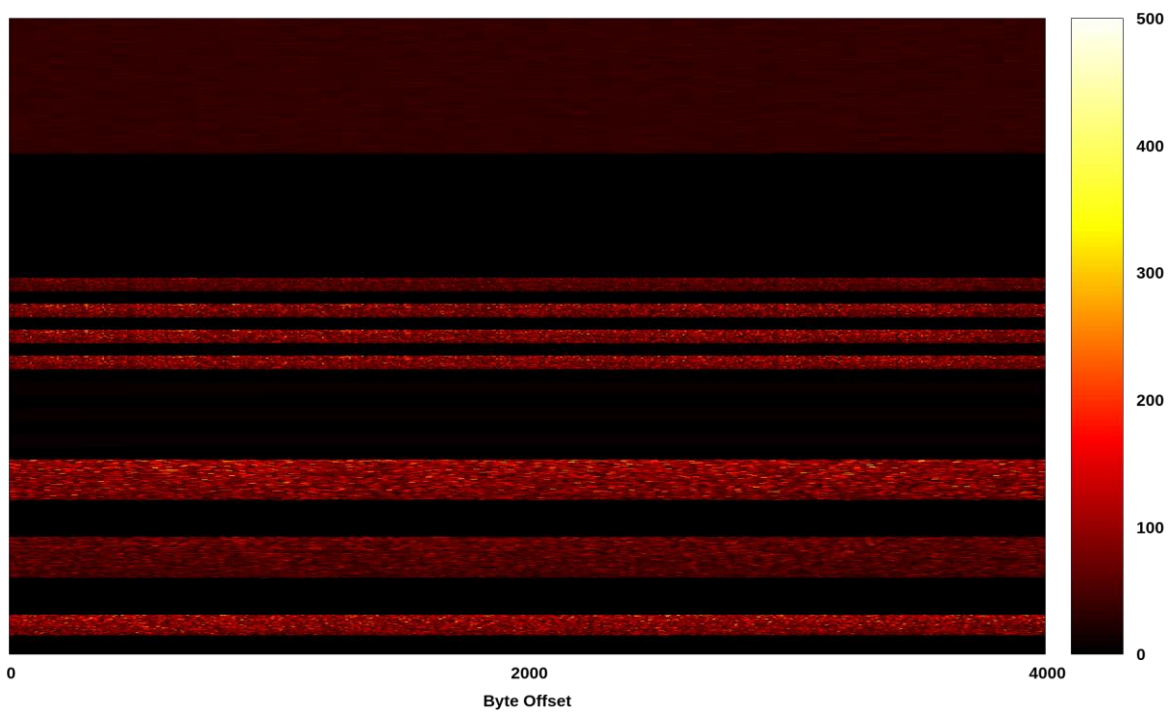
Heatmap – sparse buffer – photons

AoS

1 line = 20 tracks

SoA



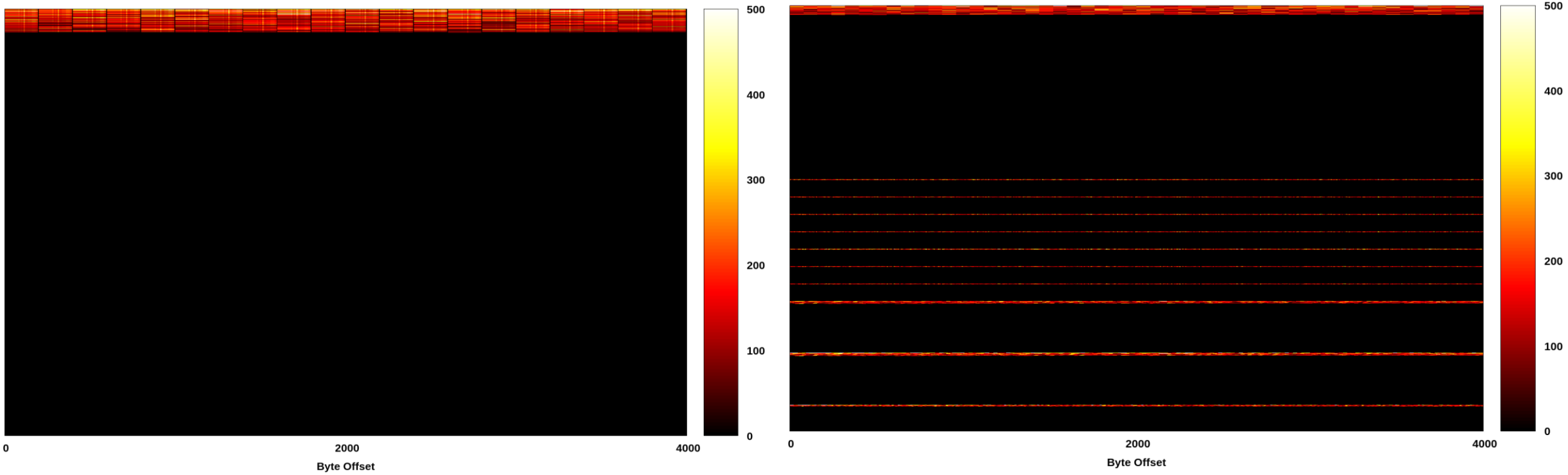


Heatmap – sparse buffer – positrons

AoS

1 line = 20 tracks

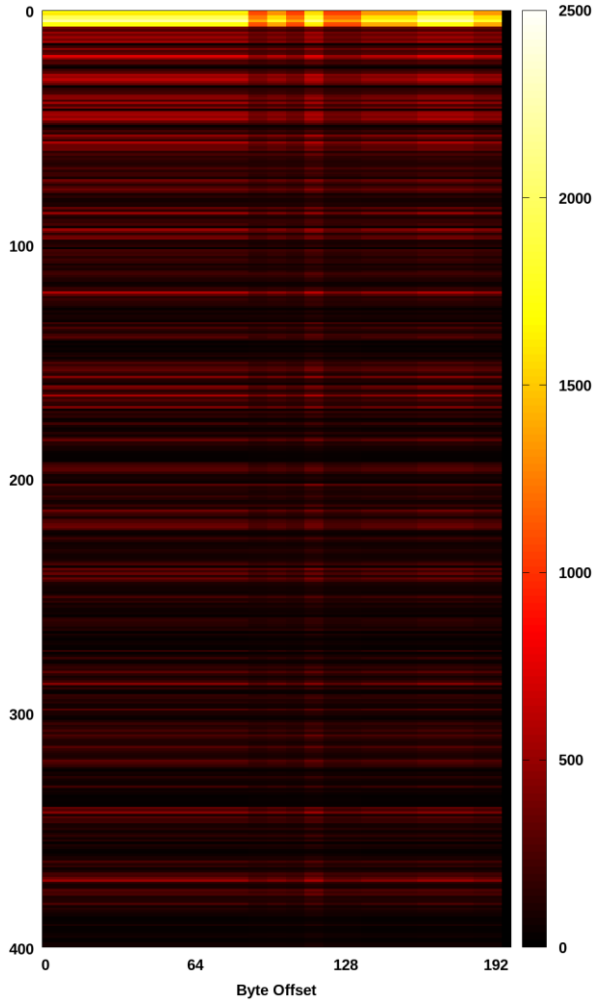
SoA



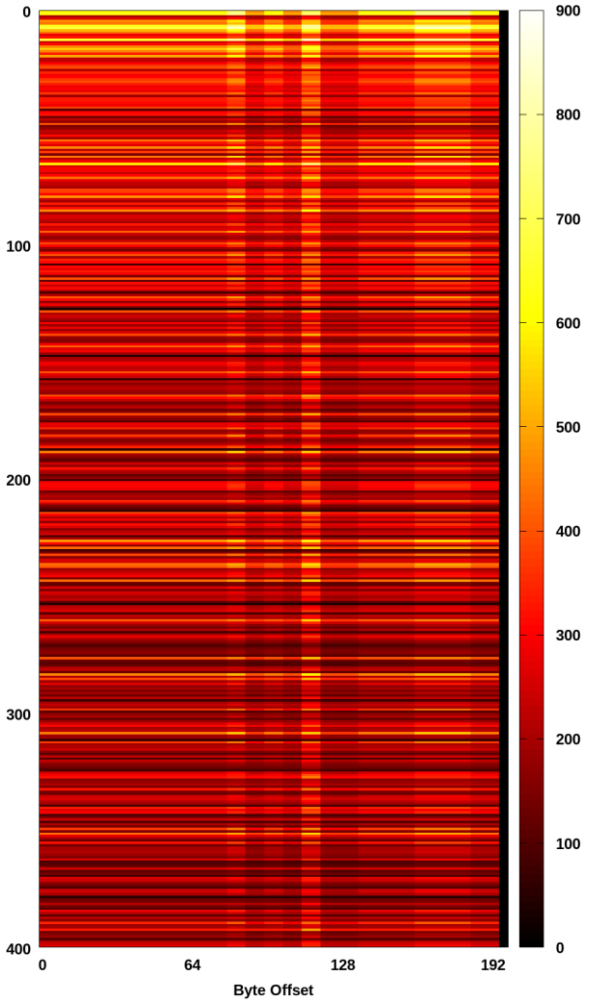
Sparse buffer – AoS

1 line = 1 track

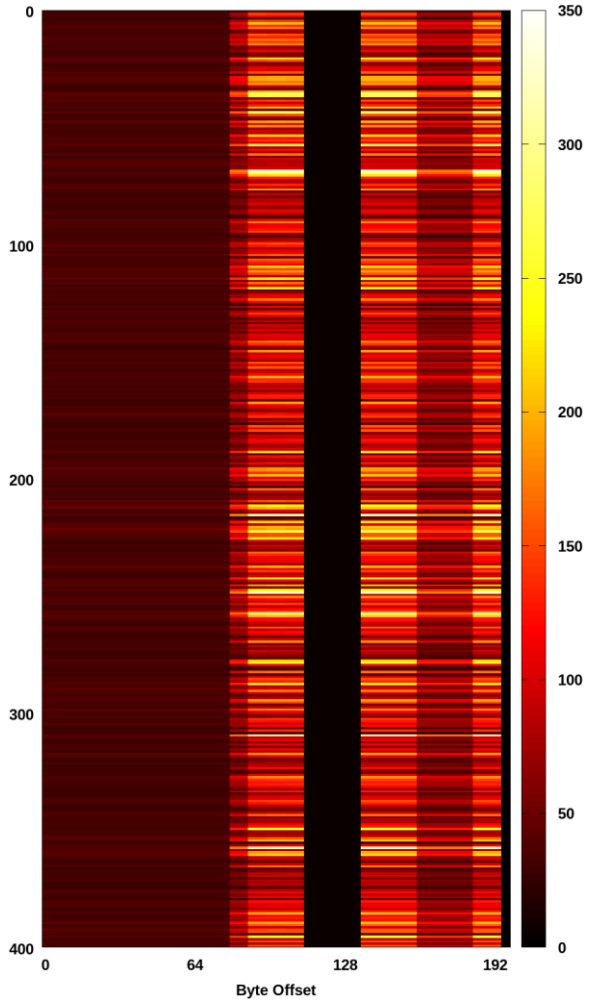
Electrons



Positrons



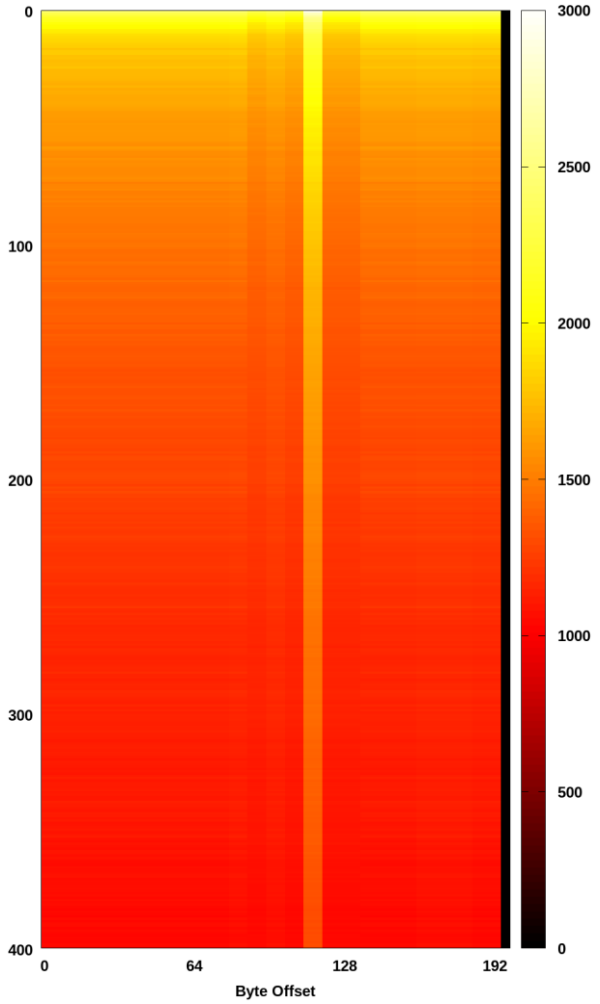
Photons



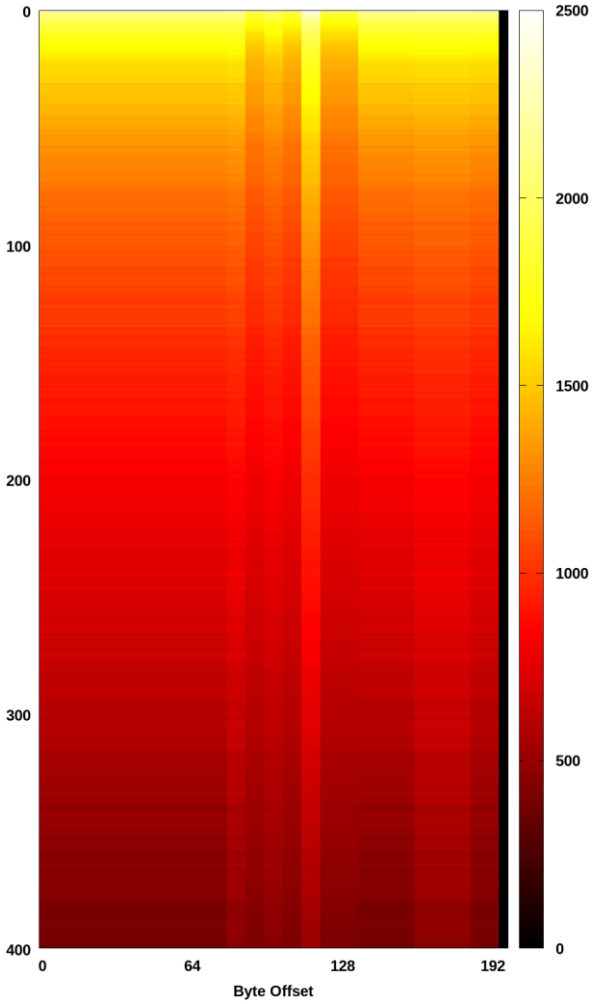
Dense buffers – AoS

1 line = 1 track

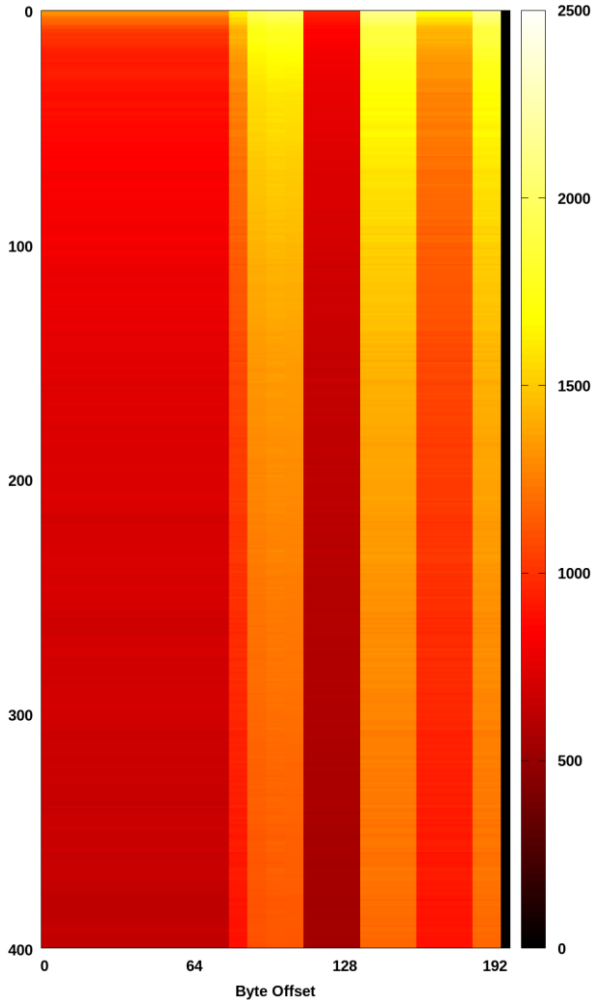
Electrons



Positrons

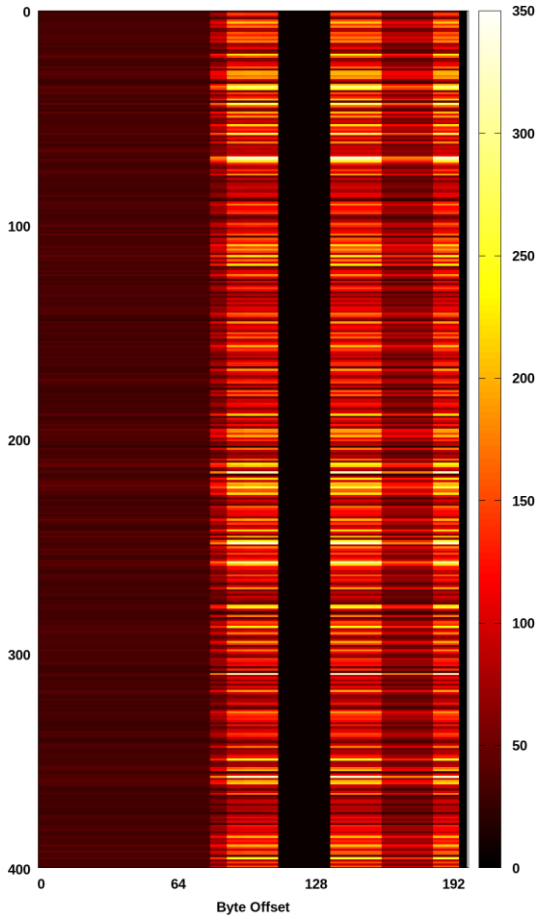


Photons

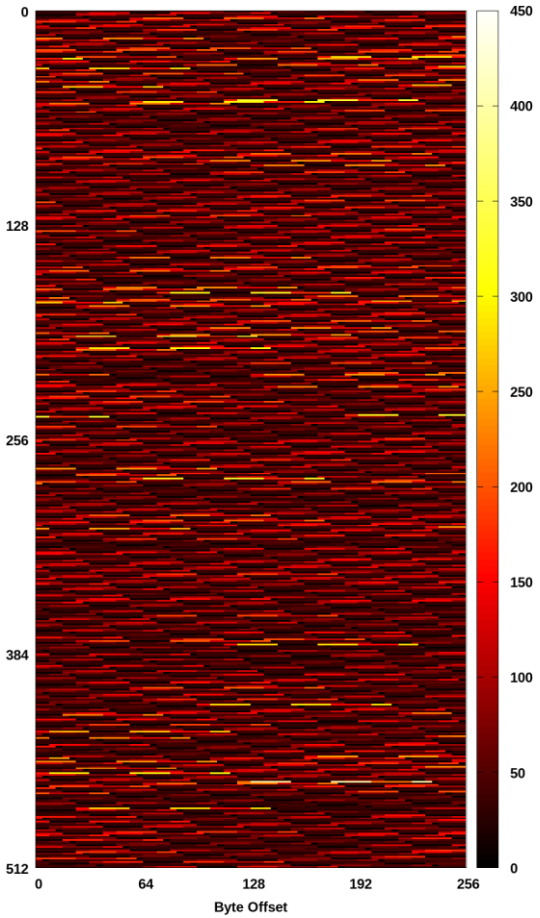


Perspective matters

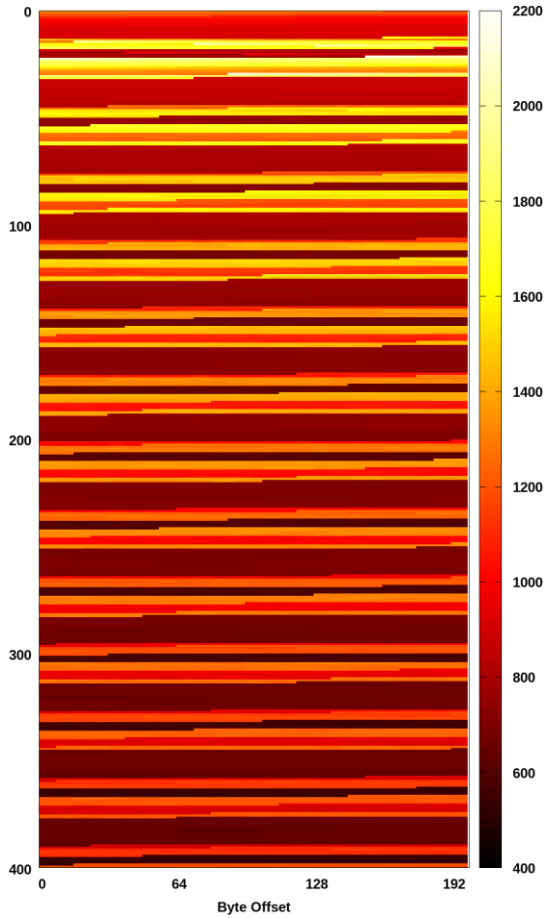
AoS, wrap 200



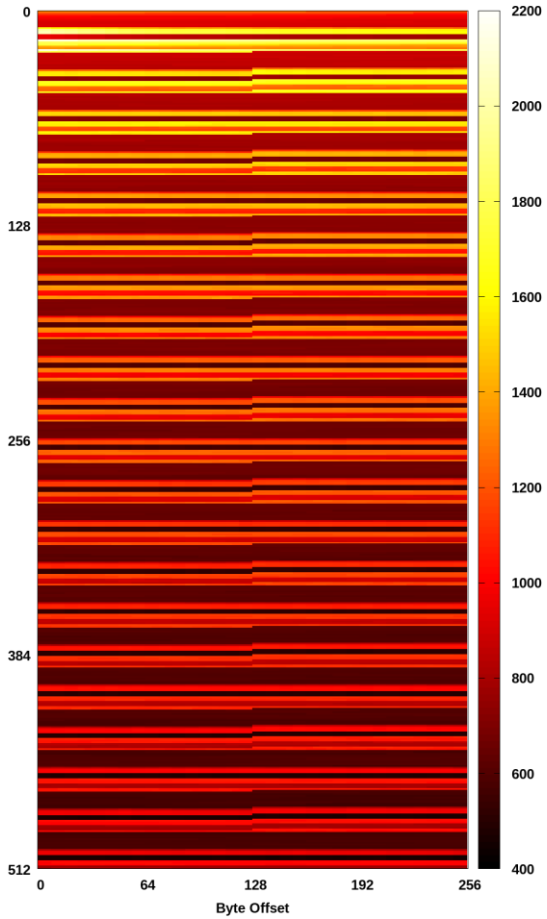
AoS, wrap 256



AoSoA32, wrap 200

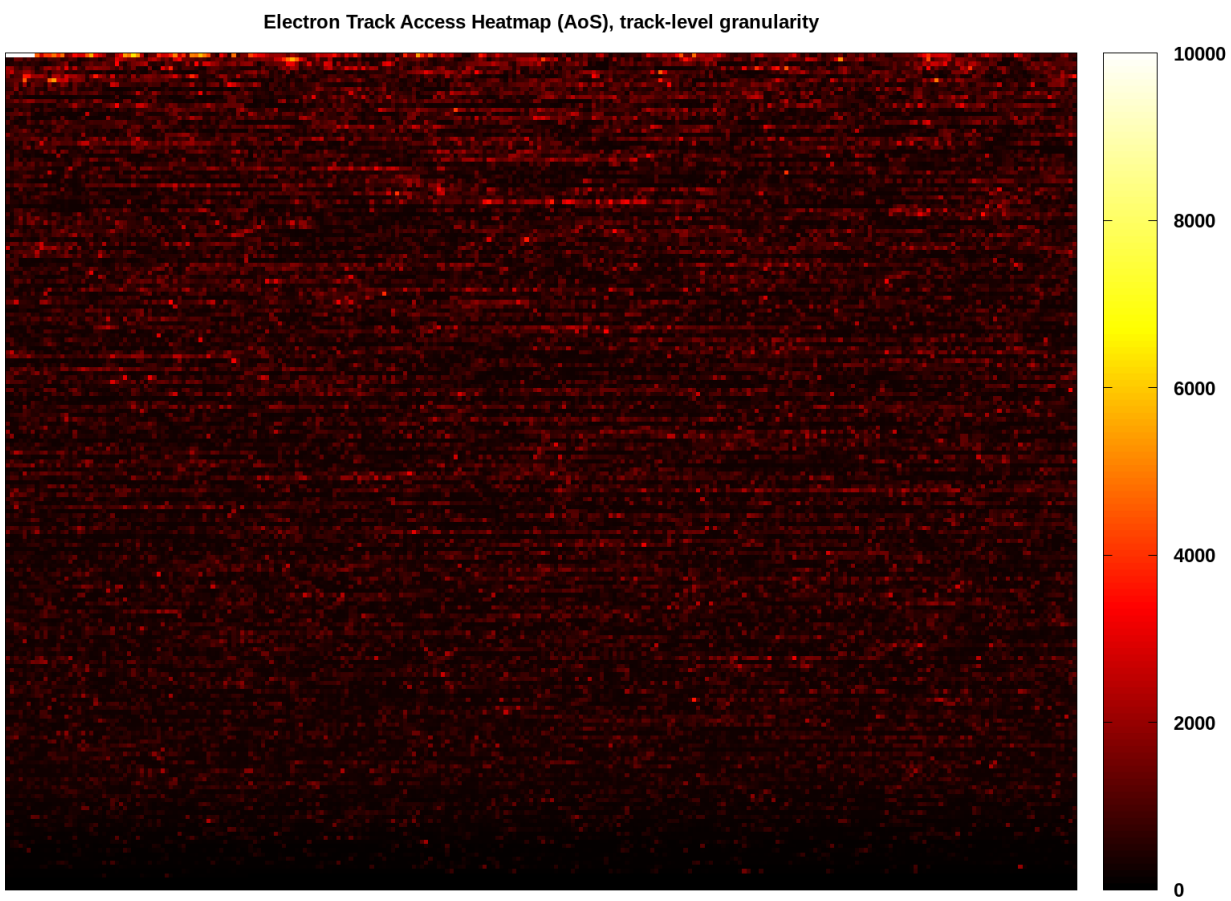


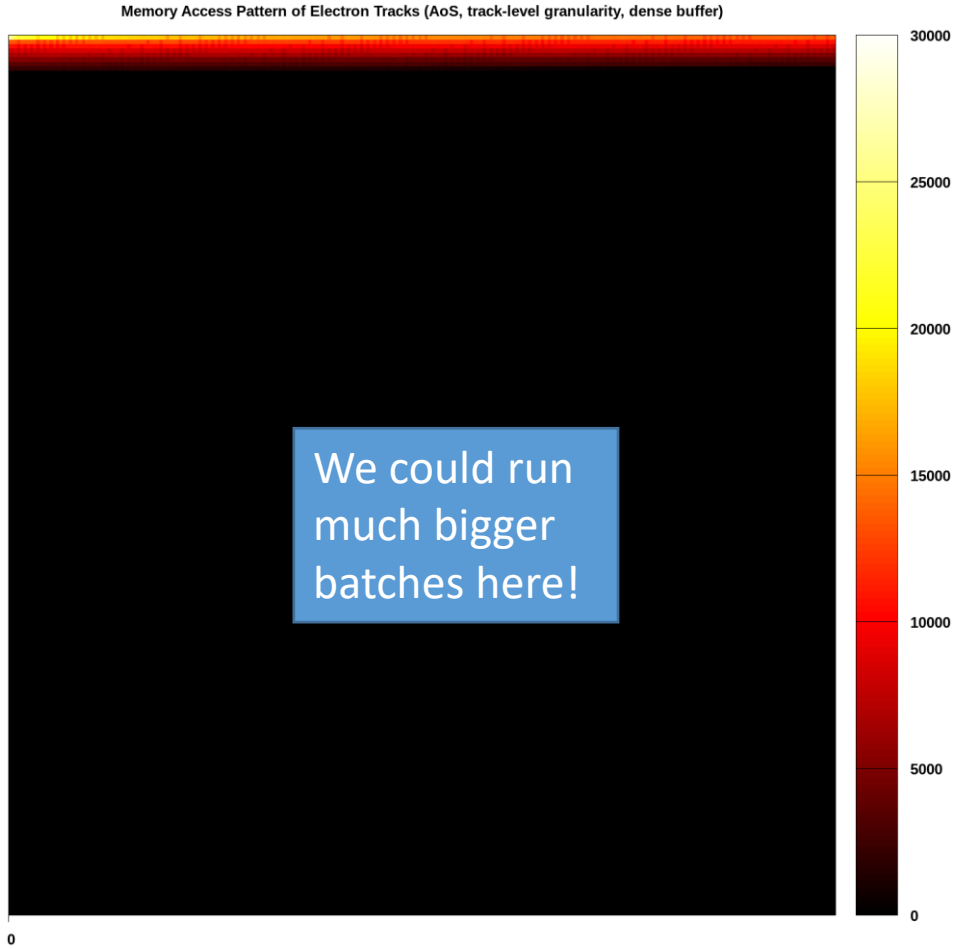
AoSoA32, wrap 256



Perfectly coalescing

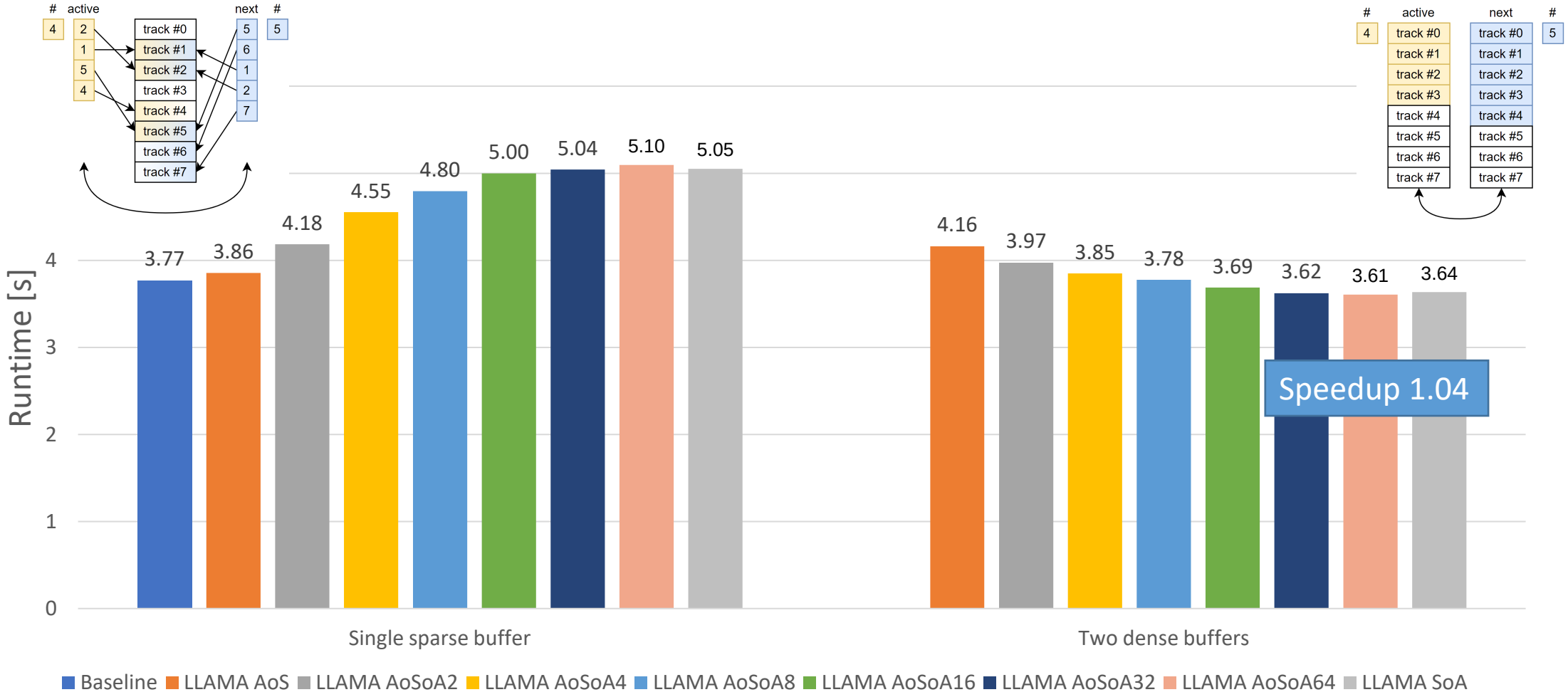
e^- track slot usage – single vs. double buffer





Both heatmaps use granularity sizeof(Track) instead of 1

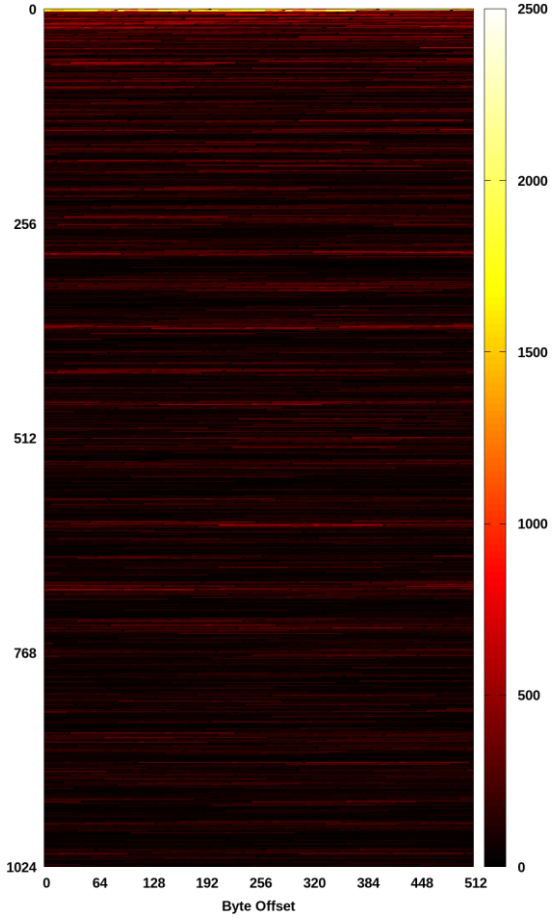
Benchmark



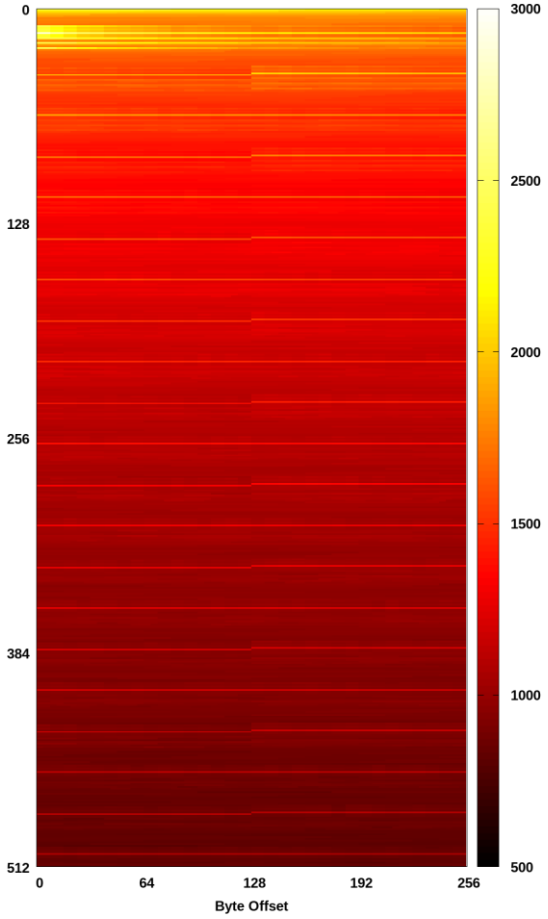
With LLAMA, it took me only 30min to test all the memory layouts!

Baseline vs. AoSoA32/SoA (better)

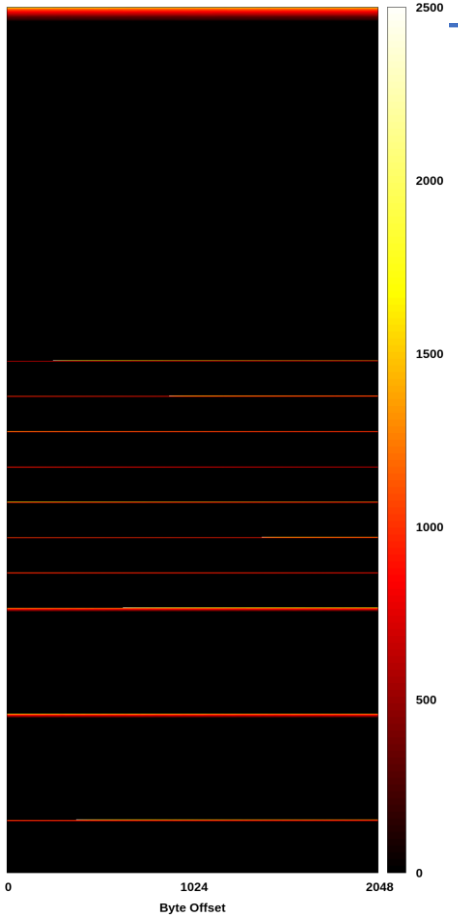
Baseline (AoS)



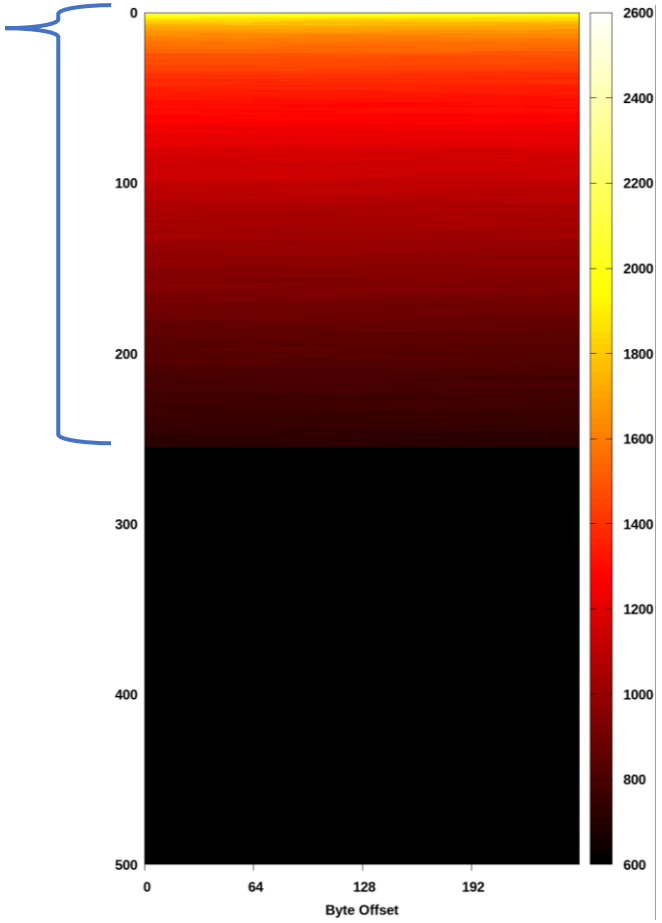
AoSoA32 dense



SoA dense



SoA dense (first subarray)



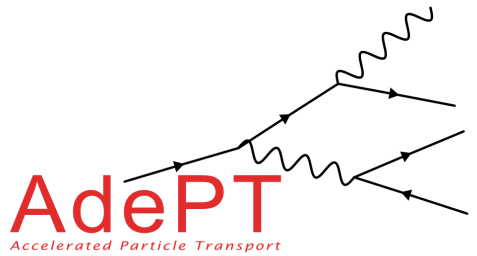
Summary and conclusions 1/2

- We integrated LLAMA into AdePT
  - We could experiment with different memory layouts easily and fast
- Memory layout and access pattern must fit together
  - SoA is not a silver bullet, requires dense access pattern
  - AoS work substantially better with sparse and random access
  - AoSoA with various blocking factors balances between them
- Memory access visualization can give incredible insights
  - … and comes almost for free with LLAMA!
  - Same data structure != same access pattern
  - Split and regroup data structs (hot/cold separation) based on access pattern
  - Cross-check on padding, coalescing, cache lines, …

Summary and conclusions 2/2

- LLAMA comes with some abstraction overhead
  - E.g., a RecordRef in LLAMA is more than just a T*, requiring extra registers
  - Compiler sometimes fails to optimize it away
- Template metaprogramming stresses compiler additionally
  - But we gained a lot of flexibility!
  - Compile time increased for incremental build by 27% (1 .cpp, 3 .cu files)
- Invasive code changes necessary around your data structure
  - example19 has 1336 LoCs (cloc), LLAMA integration: 178 ins. 226 del. (git)
- AdePT is still bound by memory access
  - Latency, access pattern, register spilling
  - But not limited by bandwidth/throughput
    - With LLAMA we could find a layout giving a small edge!

Future work



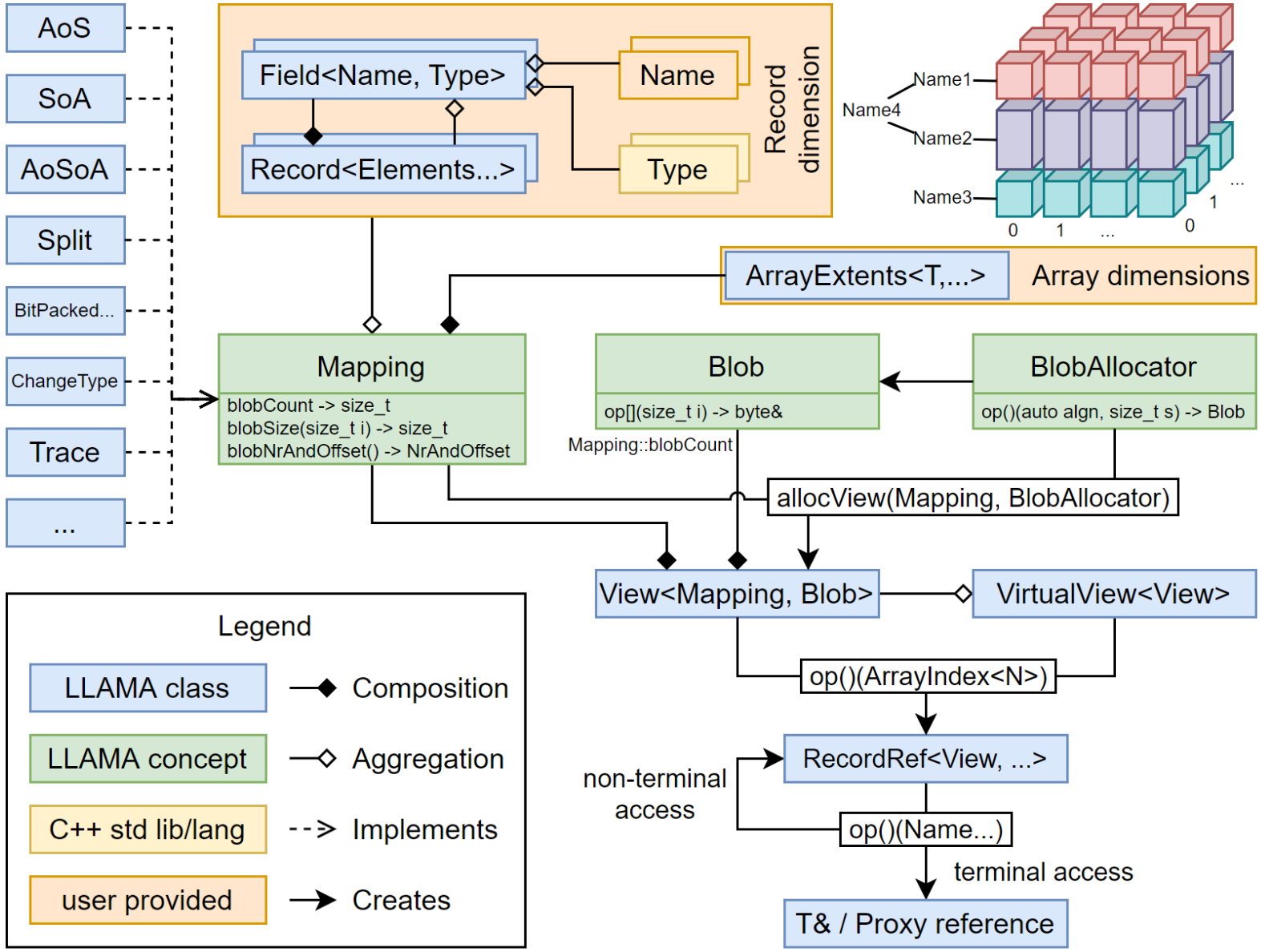
- Develop a dense, single-buffer track storage
  - We have prototypes with various compaction approaches
- Use different structures for $e^-/e^+/\gamma$ to account for access pattern
- Performance model correlating access density and memory layout
- Explore mixing global/shared memory behind a single LLAMA view
- More elaborate tracing of memory access pattern
  - E.g.: Which part of the data structure is hot at which time/stage of the kernel
- Better visualization of large memory traces
  - How to show a byte-wise trace on a 10GiB buffer on 1 screen?

Thank you, questions?

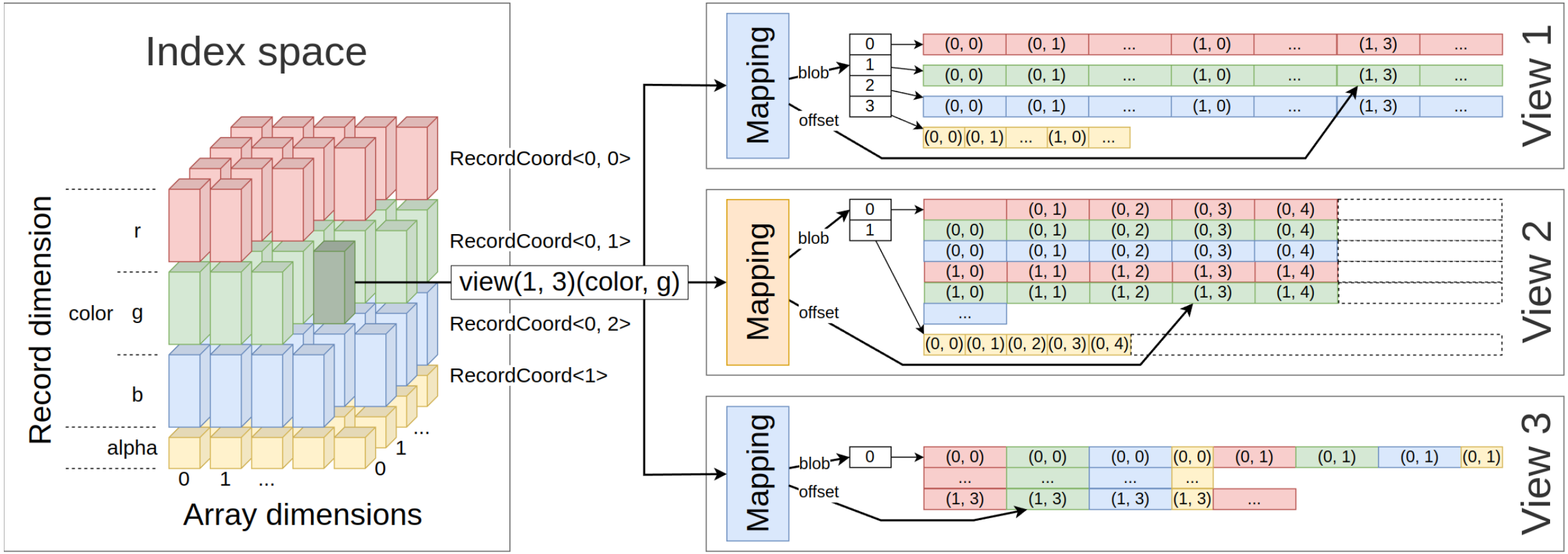
This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA)

Backup slides

LLAMA API Overview



LLAMA Mapping



LLAMA available mappings

- **AoS**: Aligned, Packed, ND-array linearizers, struct member reordering, …
- **SoA**: Single/Multi blob, ND-array linearizers, struct member reordering, …
- **AoSoA**: Inner array size configurable, ND-array linearizers, struct member reordering
- **BitPackFloatSoA**, **BitPackIntSoA**: Reduce value/mantissa/exponent bits
- **ChangeType**: Use different type for storage, then map again
- **Bytesplit**: Split all types in byte arrays, then map again
- **Trace**: Trace access/read/write counts, then map again
- **Heatmap**: Trace byte wise access counts, then map again
- **One**: Map all array indices to the same record instance
- **Null**: Read returns default constructed value, writes discarded
- **Split**: Split record in two, map each part again

Steps to migrate the code to LLAMA

- Structs need to be formulated via type lists (LLAMA record)
- Member functions become free functions
- Functions with LLAMA arguments/return values become templates
- Struct instances/references become LLAMA constructs or deduced types (auto everywhere)
- Buffers (pointers to CUDA memory) become LLAMA views
- For some mappings: code needs to work with proxy references

Track allocation – before/after

```
Track *tracks;
```

```
COPCORE_CUDA_CHECK(cudaMalloc(&tracks, TracksSize));
```

```
using Mapping = llama::mapping::AoS<
    llama::ArrayExtentsDynamic<std::size_t, 1>, Track>;
// using Mapping  = llama::mapping::PackedSingleBlobSoA<
//     llama::ArrayExtentsDynamic<std::size_t, 1>, Track>;
// ...
using BlobType = std::byte *;
using View     = llama::View<Mapping, BlobType>;
```

```
View tracks;
```

```
Mapping mapping(llama::ArrayExtentsDynamic<std::size_t,
1>{Capacity});

tracks = llama::allocViewUninitialized(mapping,
    [](auto alignment, auto size) {
  std::byte *p = nullptr;
  COPCORE_CUDA_CHECK(cudaMalloc(&p, size));
  return p;
});
```

Implications of proxy references

Proxy reference

```
RanluxppDouble state =
    currentTrack(RngState{});
double v = state.Rndm();
currentTrack(RngState{}) = state;

// decltype(currentTrack(RngState{})) is
// ProxyReference<RanluxppDouble>
```

l-value reference

```
double v = currentTrack(RngState{}).Rndm();

// decltype(currentTrack(RngState{})) is
// RanluxppDouble&
```

AdePT simulation iteration - kernels

- TransportElectrons<bool IsElectron>
  - 1. Obtain safety unless the track is currently on a boundary.
  - 2. Determine physics step limit, including conversion to geometric step length according to MSC.
  - 3. Query geometry (or optionally magnetic field) to get geometry step length.
  - 4. Convert geometry to true step length according to MSC, apply net direction change and displacement.
  - 5. Apply continuous effects; kill track if stopped.
  - 6. If the particle reaches a boundary, perform relocation.
  - 7. If not, and if there is a discrete process, hand over to interaction kernel.
- TransportGammas
  - 1. Determine the physics step limit.
  - 2. Query VecGeom to get geometry step length (no magnetic field for neutral particles!).
  - 3. If the particle reaches a boundary, perform relocation.
  - 4. If not, and if there is a discrete process, hand over to interaction kernel.
- Interaction kernels
  - 1. Find which particles will undergo the interaction that the respective kernel will take care of.
  - 2. Sample the final state.
  - 3. Update the primary and produce secondaries.
- FinishIteration
  - Clear the queues and return the tracks in flight.
  - This kernel runs after all secondary particles were produced.

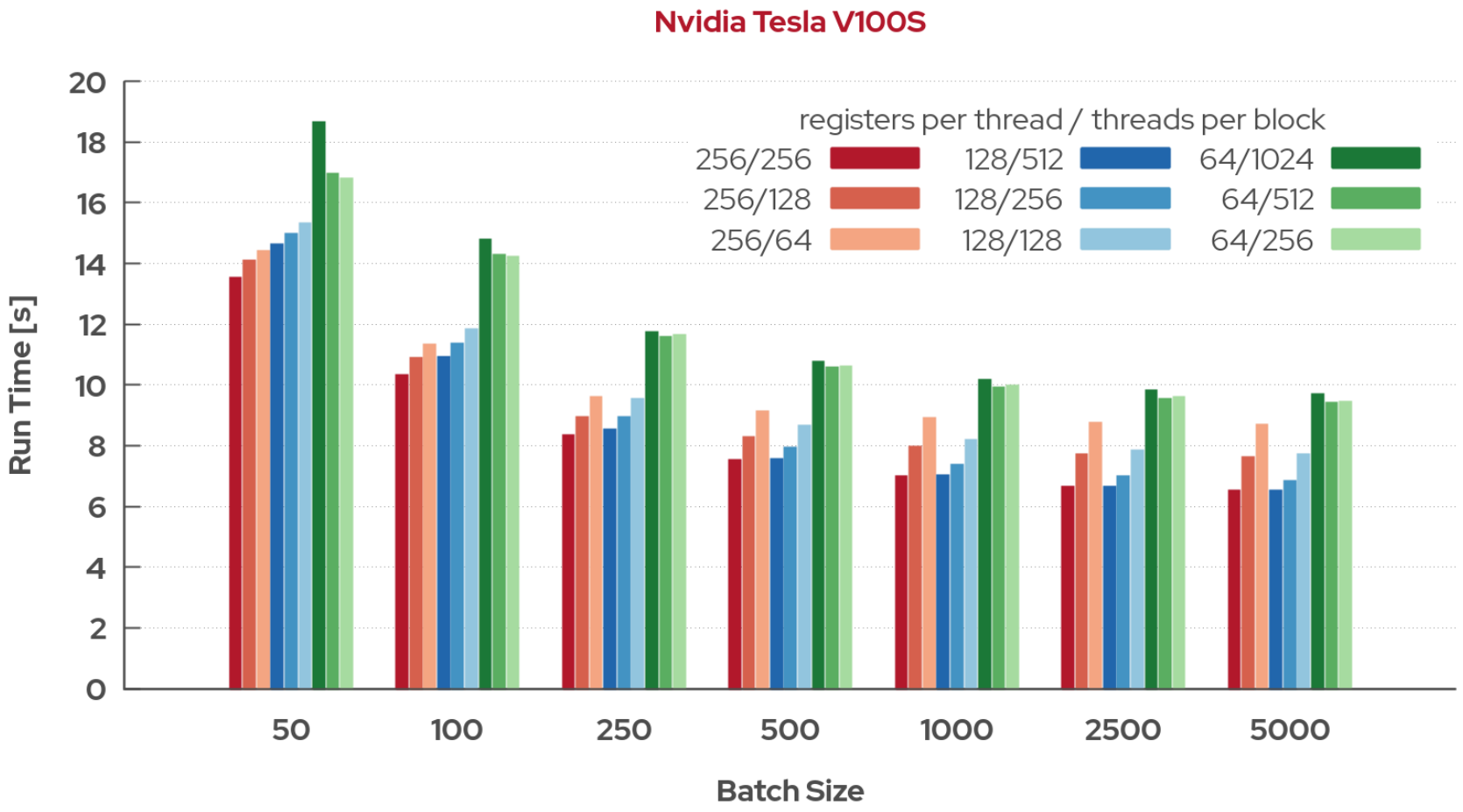
Optimization attempts that did not work

- SoA layout for Ranlux++ RNG state
  - (ncu shows big stalls when accessing the RNG state)
  - Because threads consume different quantities of RNs -> divergence
  - We tried 3 different versions with various slowdowns
- Different log implementations
- Using single precision CUDA intrinsics for some log/sqrt calls
  - We observed different outcomes but where not sure if those are still valid
- Replacing per-thread binary search in FindLowerBinIndex by strided linear search
- Put the G4HepEmTrack in shared memory (because it is too big and needs a reduction in block size)
- Different stack handling in BVH::LevelLocate
- (Use curand instead of Ranlux++, 26% gain, but bad statistics)

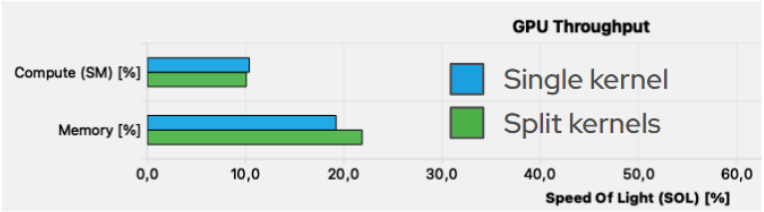
Optimizations that worked

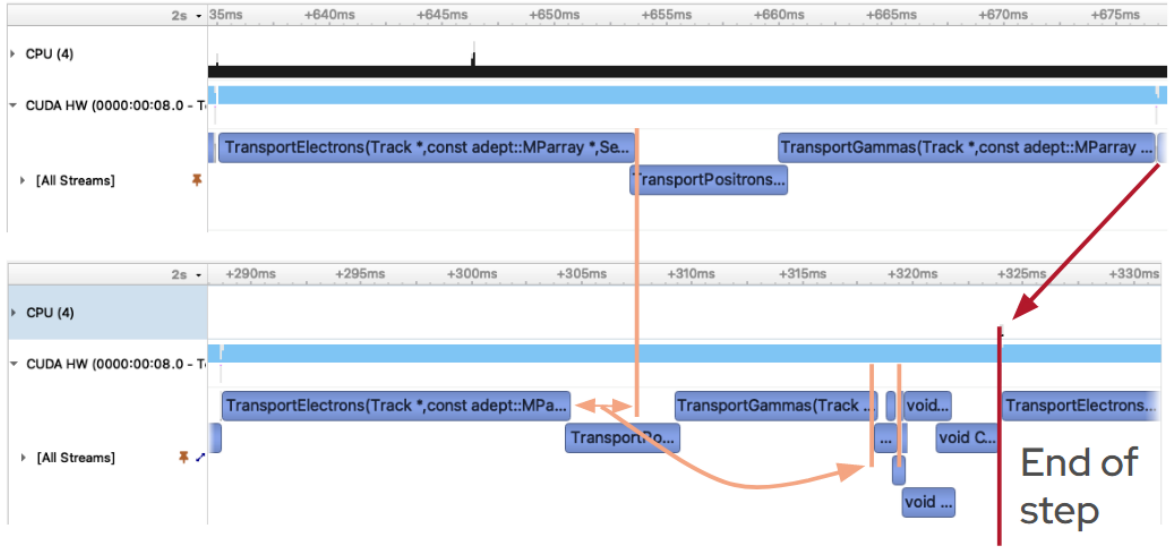
- TopMatrixImpl recursive -> iterative
- Various hints on inlining and not inlining
  - E.g. noinline on RNG Advance()
- Put the RNG into shared memory, even if it isn’t shared
  - SM hardware handles the irregular access much better than global memory (where RNG was spilled into from registers)
- Launch bounds (influences register allocation)
- Launch parameters (block/grid sizes)
- Use a larger initial capacity
- Avoid repeated access to global memory and keep more data in registers
- Aligning transformation data in BVH to have better load instructions
- Remove the type erased wrapper G4HepEmRandomEngine
- Less upfront initialization of stack structure in BVH::LevelLocate
- Kernel fission of interactions to reduce thread divergence

Varying launch bounds and batch size



Separate interaction kernels



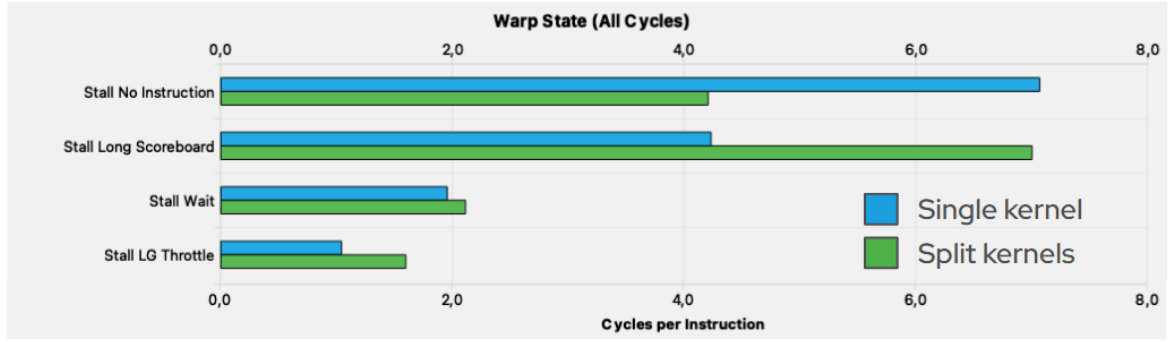


Problem: Threads in transport kernels diverge because of diverging interactions
→ **13 / 32** threads active on average

Here: Split off interaction computations from cross-section and geometry kernels (one kernel for pair creation, one for ionisation, …)

Result: **17 / 32** threads active for physics + geo
29 / 32 threads active for Bremsstr.
Run time: 6.4 s → 5.5 s

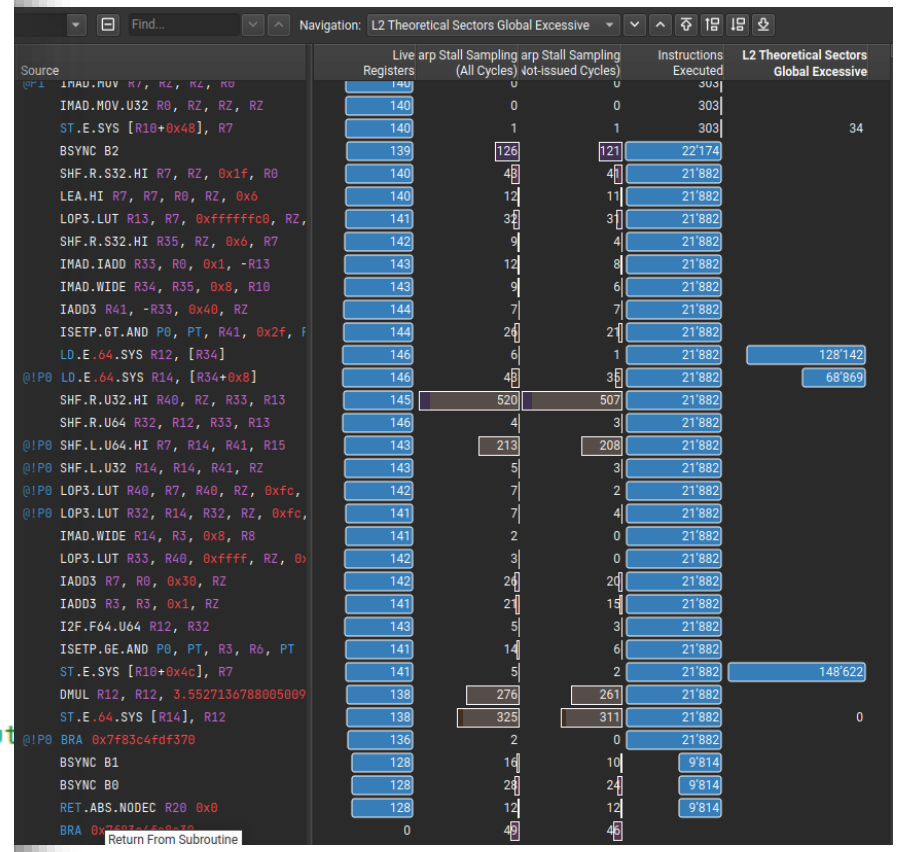
Conclusion: Keeping threads coherent is key for detector simulation
Generally difficult; stochastic processes



Bottleneck: Ranlux state memory layout

- State is stored as AoS (uint64_t[9])
- Mostly 1 or 2 elements needed
  - Advance(), touching all elements, called rarely

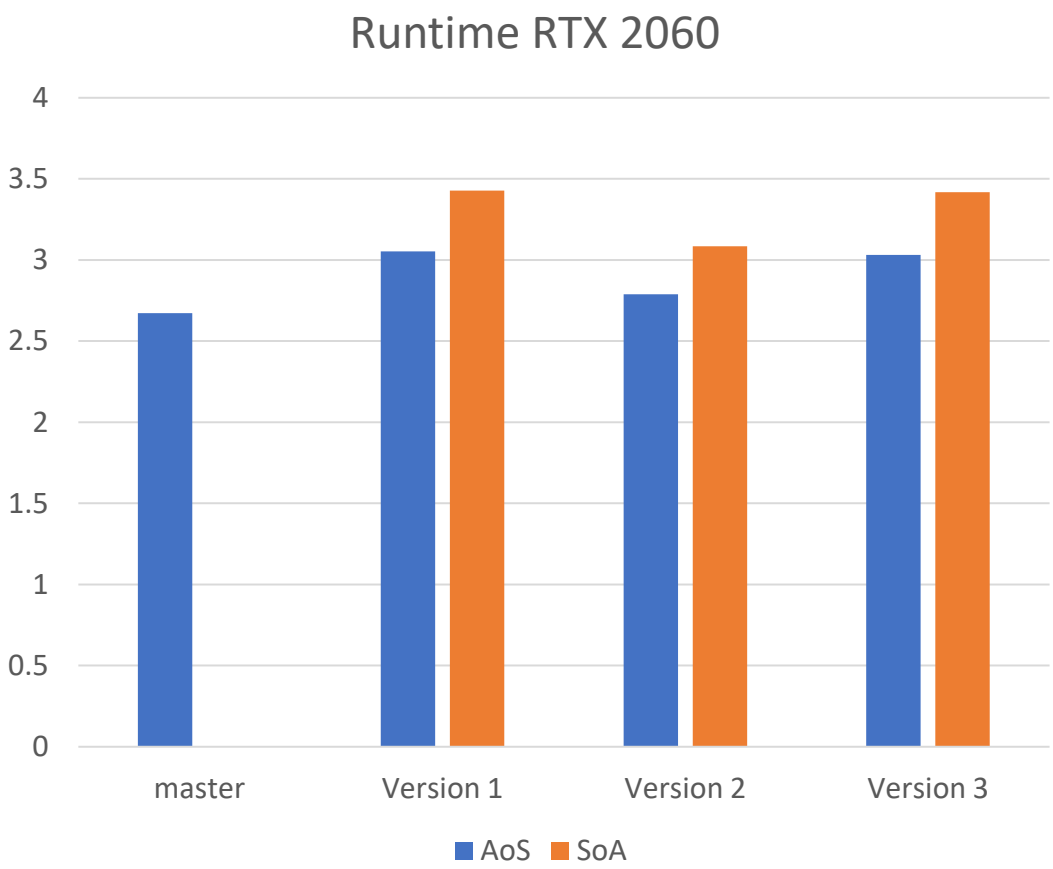
```
__host__ __device__ uint64_t NextRandomBits()
{
  if (fPosition + w > kMaxPos) {
    Advance();
  }

  int idx    = fPosition / 64;
  int offset = fPosition % 64;
  int numBits = 64 - offset;

  uint64_t bits = fState[idx] >> offset;
  if (numBits < w) {
    bits |= fState[idx + 1] << numBits;
  }
  bits &= ((uint64_t(1) << w) - 1);

  fPosition += w;
  assert(fPosition <= kMaxPos && "position out

  return bits;
}
```



LLAMA SoA for RNG state

1. Store a LLAMA RecordRef (=handle) to the RNG state in RanluxppEngineImpl
2. Resolve all SoA addresses via LLAMA and change the state in RanluxppEngineImpl from a uint64_t[9] to a uint64_t*[9]
3. Replace all classes and member functions by free functions and pass LLAMA RecordRef to RNG state around

- None of these worked because each thread needs a different count of random numbers (=indexing divergence)



Runtime characteristics