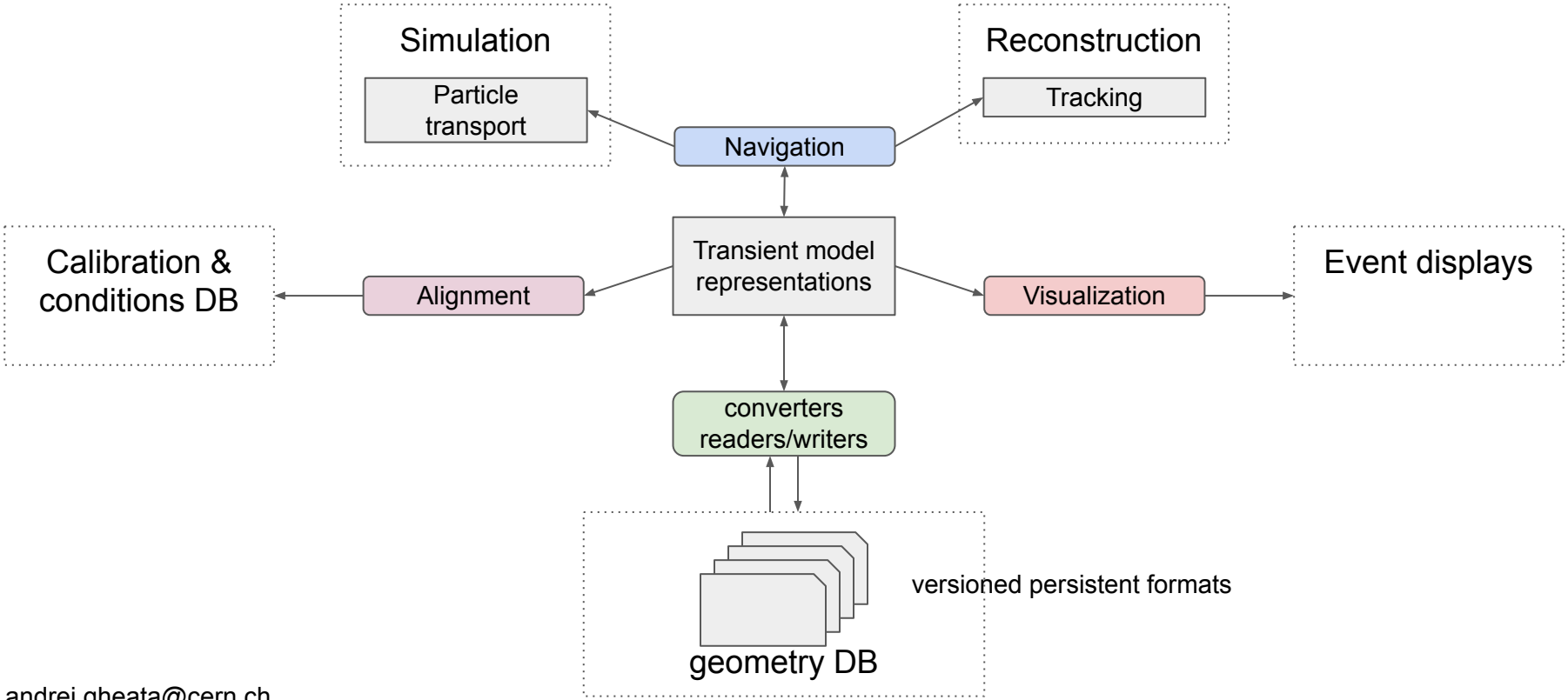
VecGeom Detector Geometry on GPU

andrei.gheata@cern.ch

Content

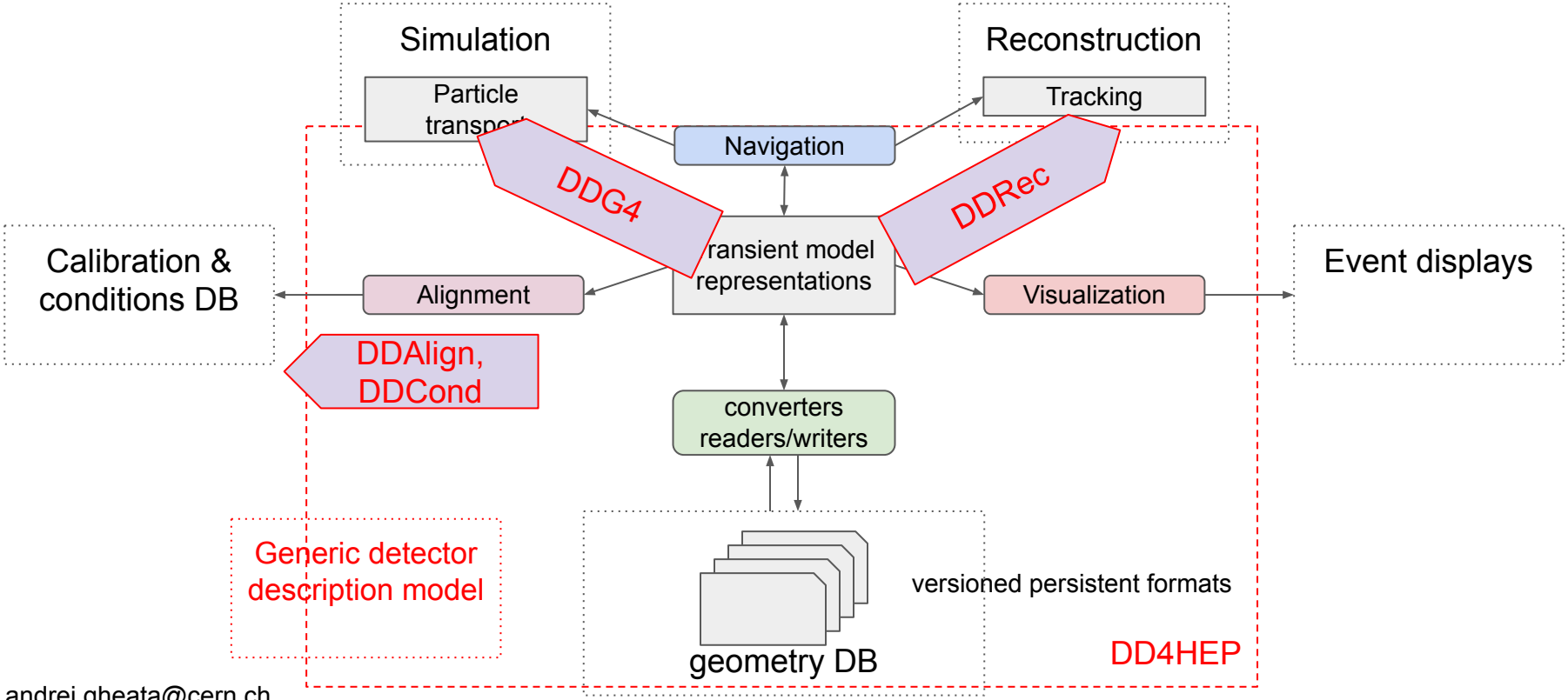
- Geometry: HEP context
- VecGeom in the picture, main features
- GPU awareness
- Preliminary performance
- GPU features roadmap

Geometry in HEP

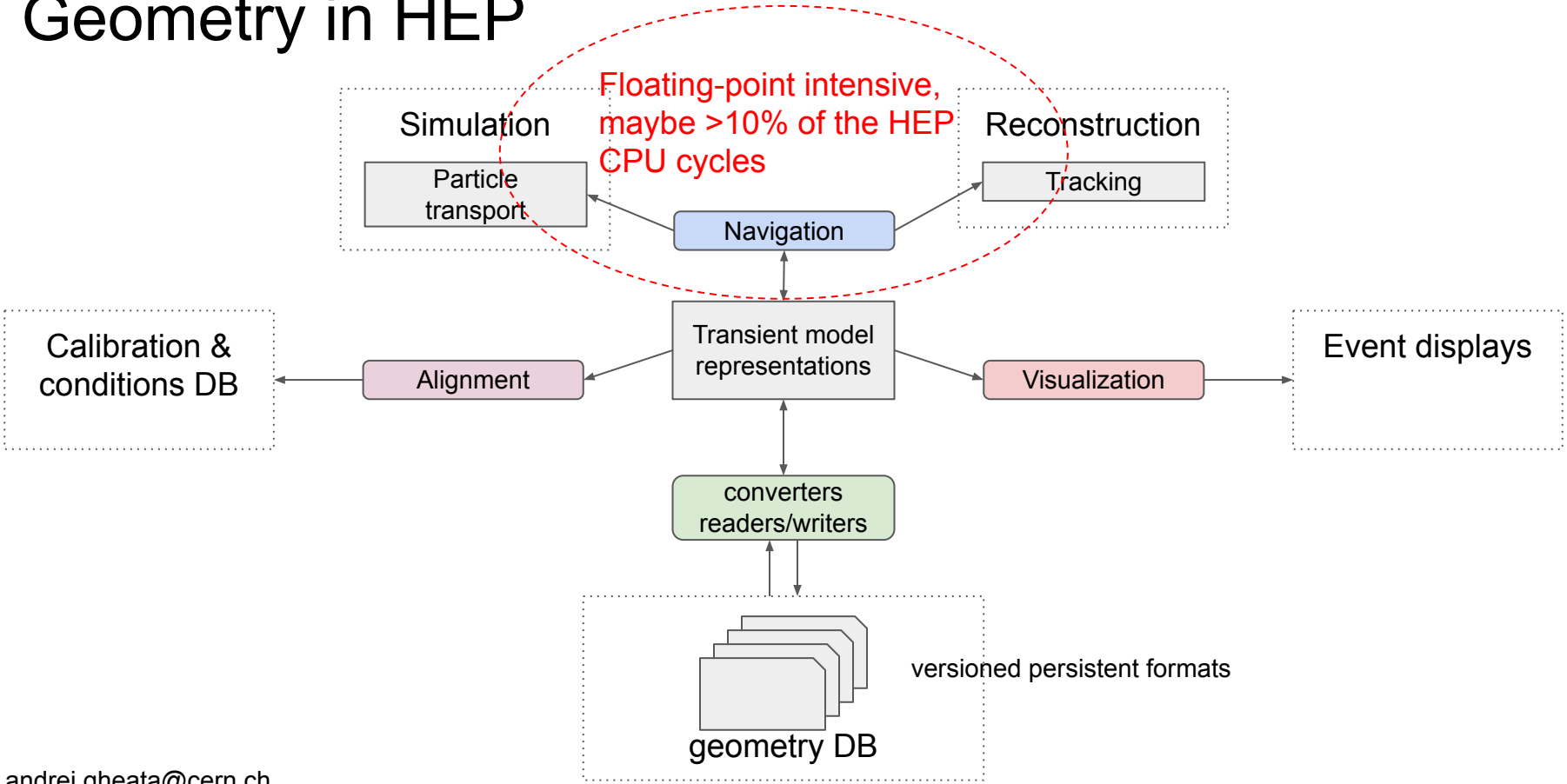


andrei.gheata@cern.ch

Geometry in HEP



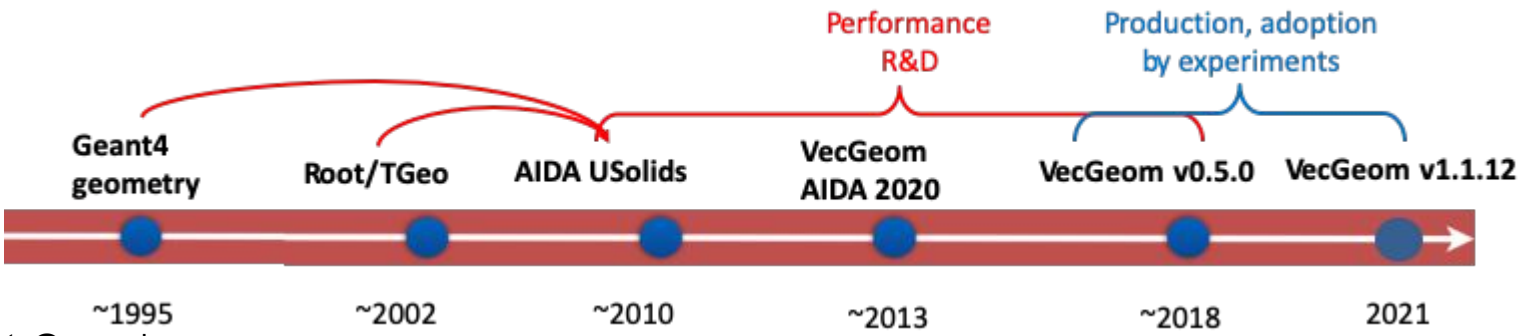
Geometry in HEP



andrei.gheata@cern.ch

Why VecGeom?

- Performance extension of a R&D effort for unifying geometry algorithms
  - Pick best, increase code quality and performance, maintain long term
- Provide support for multi-track parallelism
  - Encompass parallelism/vectorization
  - Advanced stateless navigation
  - Multi-architecture support and type abstraction for the navigation layer
- Using by default VecGeom solids planned for 10.8 release of Geant4

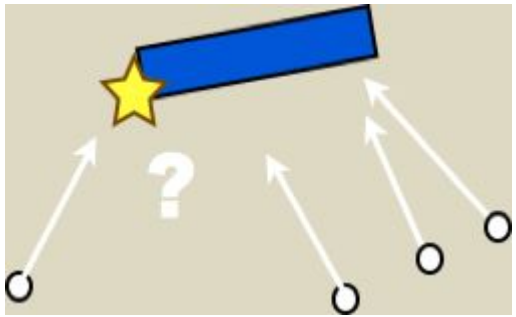


andrei.gheata@cern.ch

Performance: vectorization

Vector signatures

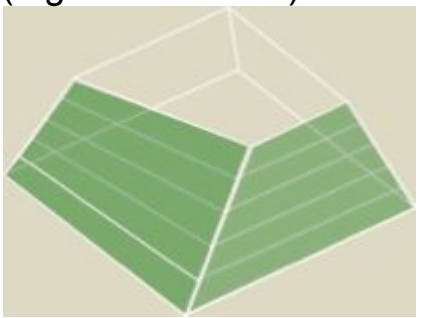
parallel collision detection



Multi-particle queries

Internal vectorization

internal loops on features (e.g. lateral faces)

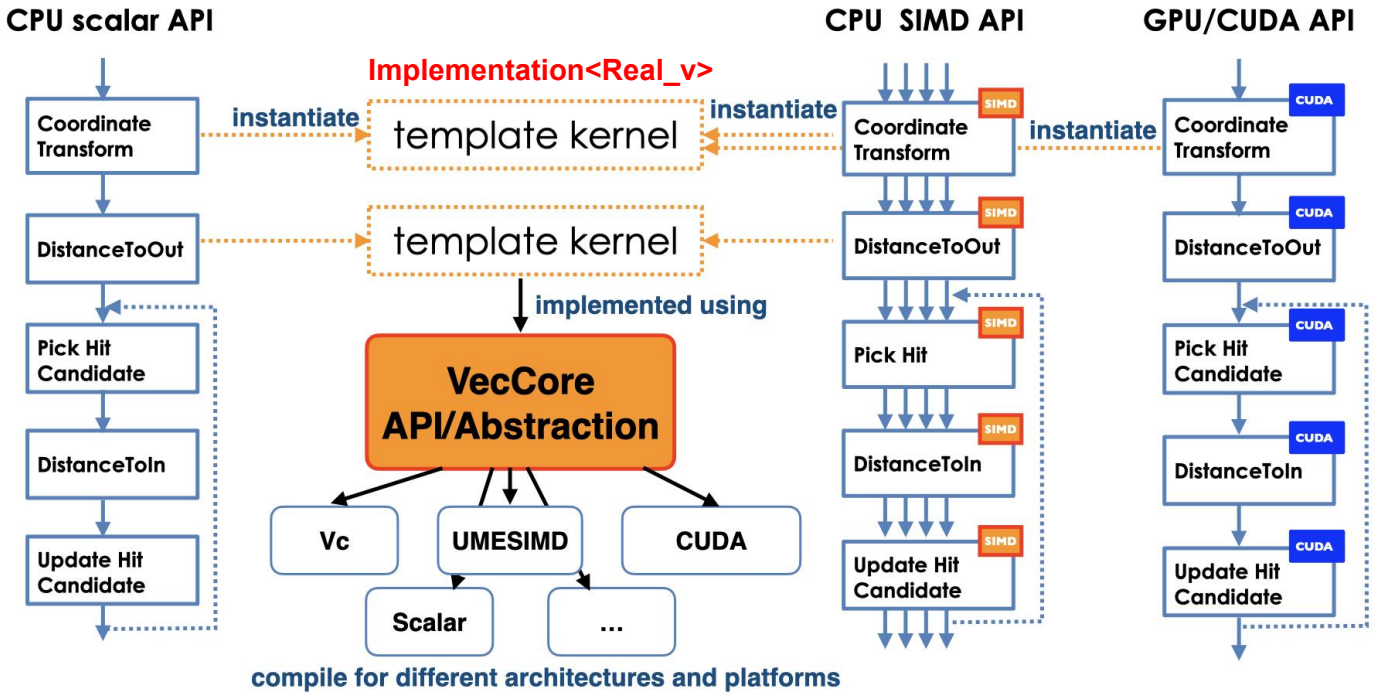


Single-particle queries

andrei.gheata@cern.ch

VecGeom portability

```
using VectorBackend = vecCore::backend::VcVectorT<Precision>;
using Real_v = vecgeom::VectorBackend::Real_v;
```

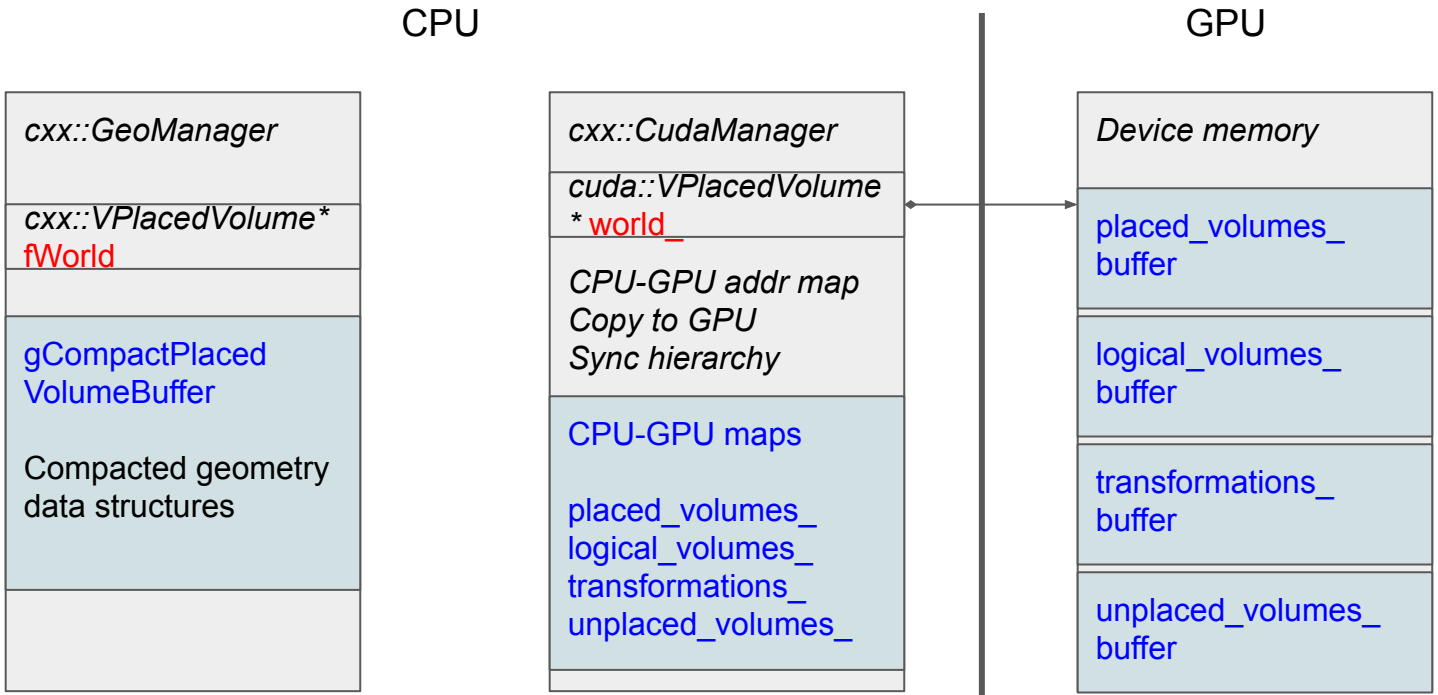


VecGeom & CUDA

- The choice: have a valid C++ object on device mapping the host one
  - Compile two layouts for geometry classes in different inline namespaces (cxx/cuda)
    - Separate nvcc compilation step producing an additional static library
  - Allowing the class layout to be different on host/device
    - E.g. some data members not visible in the *::cuda* namespace
  - Instrumenting VecGeom classes with methods allowing to copy CPU instances to GPU
    - Serialization in device memory buffers
- Copying the geometry logical tree to the device buffer
  - Re-creating the same hierarchical data structure as on host: solids, transformations, logical and placed volumes
- Enabling the solids navigation interfaces and their calees for __device__
  - Reproducing the same calling sequence as on __host__
  - Issues porting the global navigation layer (see later)

andrei.gheata@cern.ch

CudaManager



CudaManager::Instance::LoadGeometry(volume); // can copy a sub-tree
CudaManager::Instance::Synchronize(); // heavy lifting: allocation + copy

What is not ported on GPU

- Solids using SIMD data members
  - Tessellated, extruded, multi-union solids
- Level locators connected to each logical volume. Optimizing the search of children volume candidates for navigation
  - E.g voxelized or hierarchical bounding boxes, using SIMD helpers
  - Not portable “as is” to CUDA. Currently using an un-optimised level looper.
  - GPU-friendly level locator implementation planned for this year

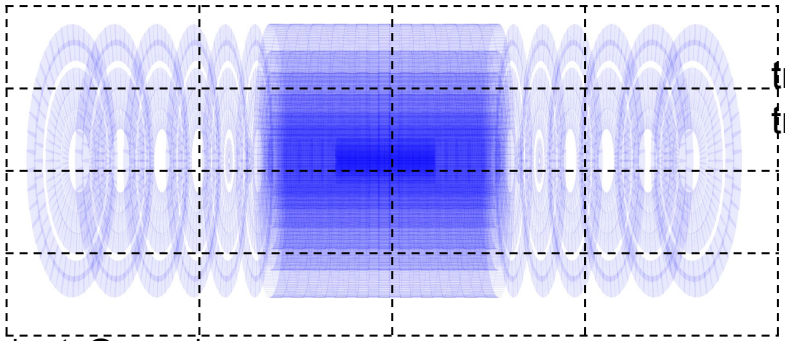
Roadmap for optimizing VecGeom for GPU

- Functionality and correctness demonstrator - done
  - Validity of the GPU data structures & comparison with CPU results
  - Ray-tracing CPU-GPU benchmark allowing to load arbitrary geometry on device
- Initial optimizations and performance evaluation - first half 2021
  - Optimizing the navigation layer and memory usage
  - Evaluation in the context of GPU simulation demonstrators (AdePT, Celeritas,)
  - Performance metrics: GPU occupancy, thread divergence in geometry kernels, register pressure, stack usage, memory footprint
- Single-precision usage in geometry - last half 2021
  - Double precision throughput limited compared to single precision, specially on consumer cards
- R&D for GPU-friendly geometry transformations & third-party solutions

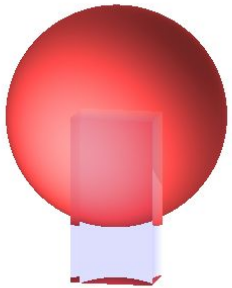
andrei.gheata@cern.ch

Raytracer prototype: initial validation

- Exercise a small part of the stepping workflow on GPU
  - Create/upload realistic geometry, perform geometry traversal with straight rays, invoking simple light models to generate an image
- Goals
  - Implement code so it runs on both CPU and GPU
  - Test efficiency of different work scheduling strategies for fixed and dynamic problem size
  - Profile the GPU application and try to optimize



trackML
transparent

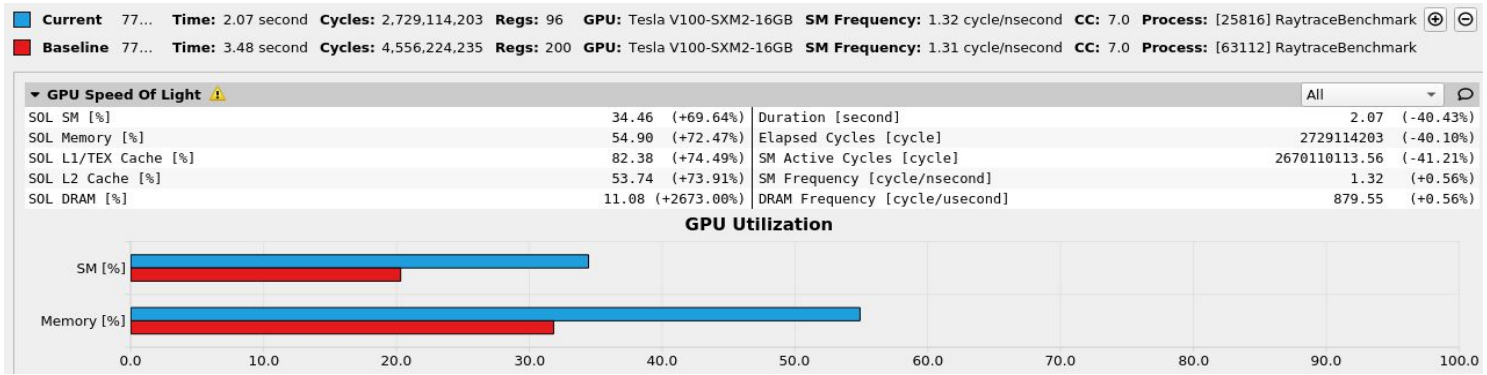


reflection +
refraction +
specular models

andrei.gheata@cern.ch

Raytracer prototype: initial performance

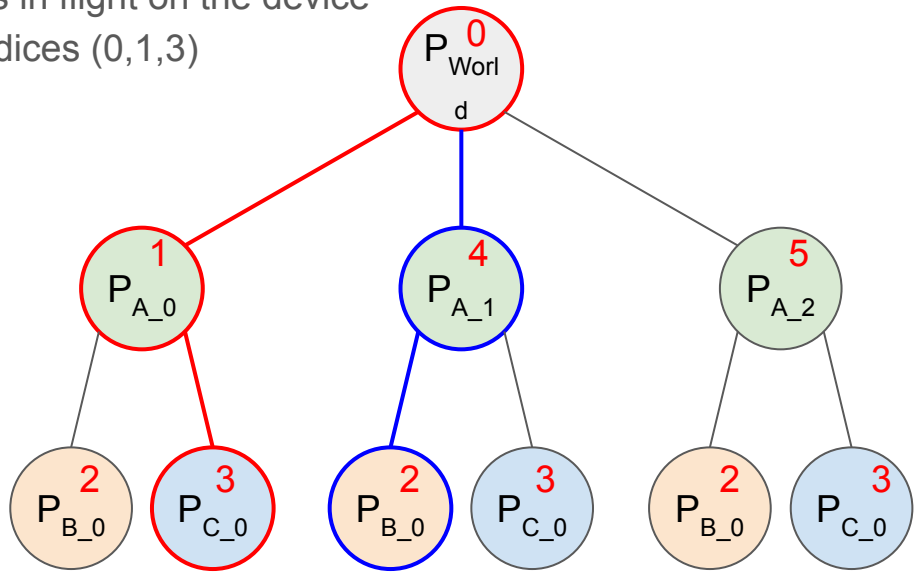
- NVidia hackathon@Sheffield last summer
  - Testing few scenarios: single image vs multiple streams, single big kernel vs. repeated small kernel, device link time optimizations
  - Low GPU utilization (expected)
  - Tuning launch configuration and register usage yielded 40% better performance
  - No major benefit observed for LTO for separate compilation mode on RTX2080



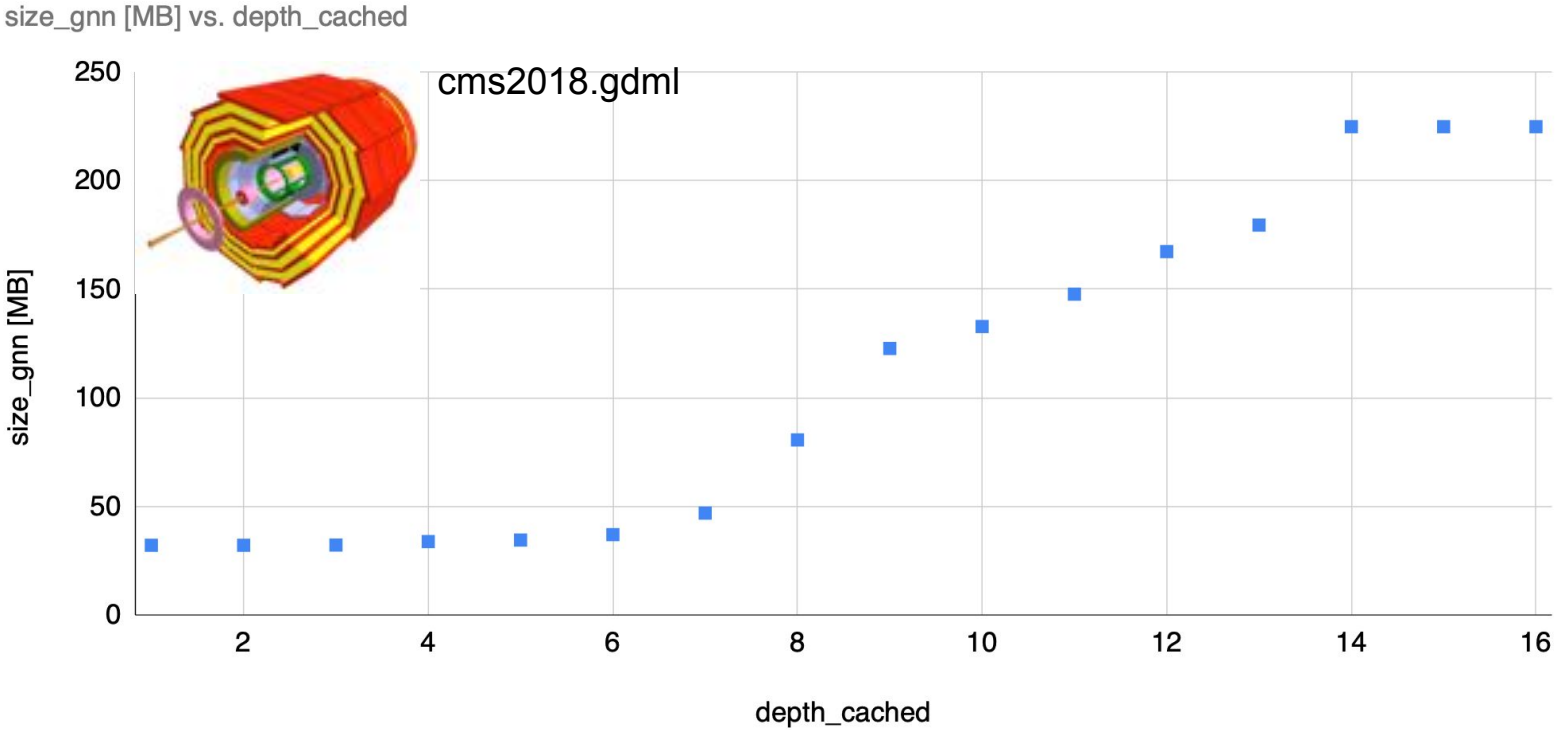
Memory optimization: using global navigation indexing

- Large memory footprint for storing the geometry state per ray/track
  - Potentially we want to handle O(10^6) tracks in flight on the device
  - Need to store full path of placed volume indices (0,1,3)
  - Large overhead for complex (deep) setups

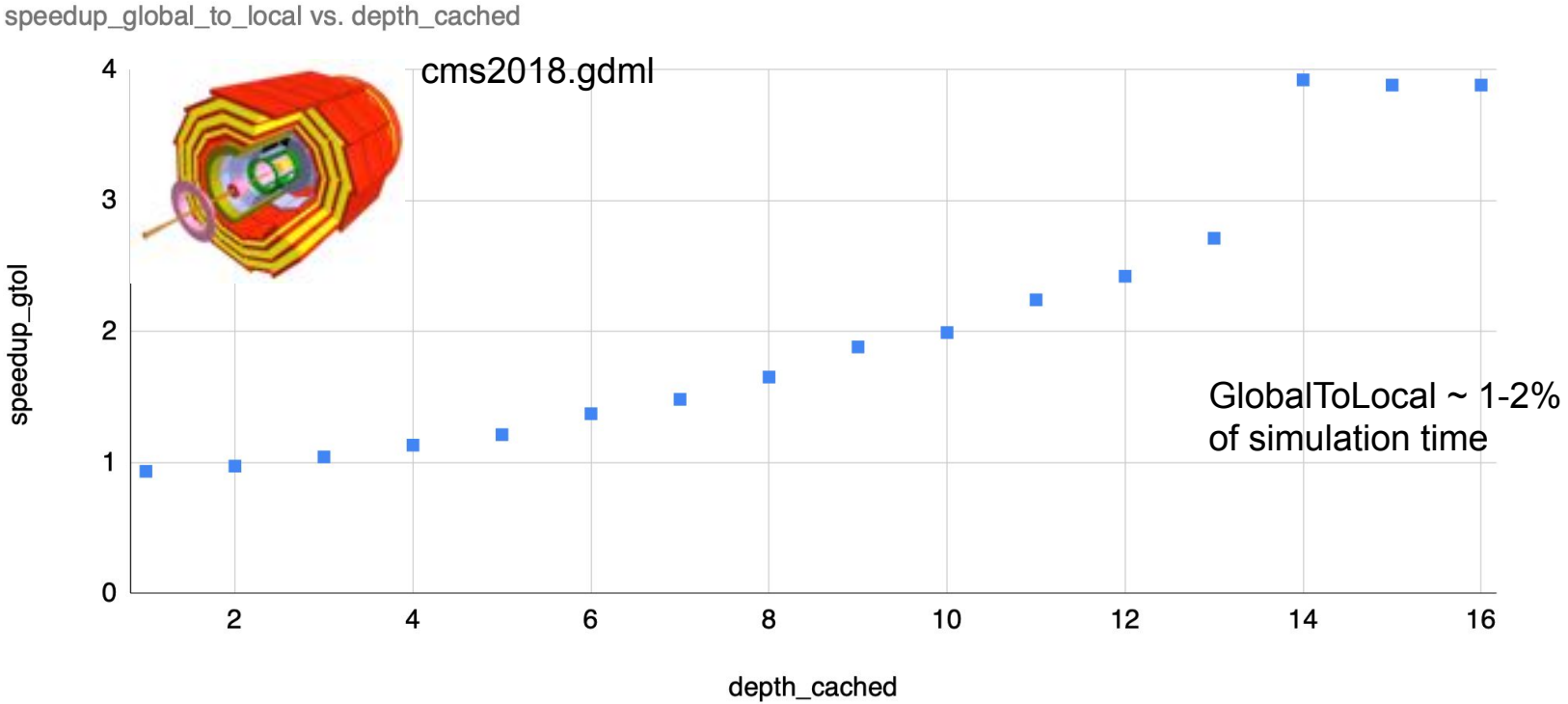
- In most cases, large memory benefit obtained if storing all touchable info in a table and identifying them using a 32-bit integer
  - Bonus: optional caching possible for global transformations



Memory size depending of matrix cache depth



Speedup for GlobalToLocal computation vs. geometry depth for caching global transformations



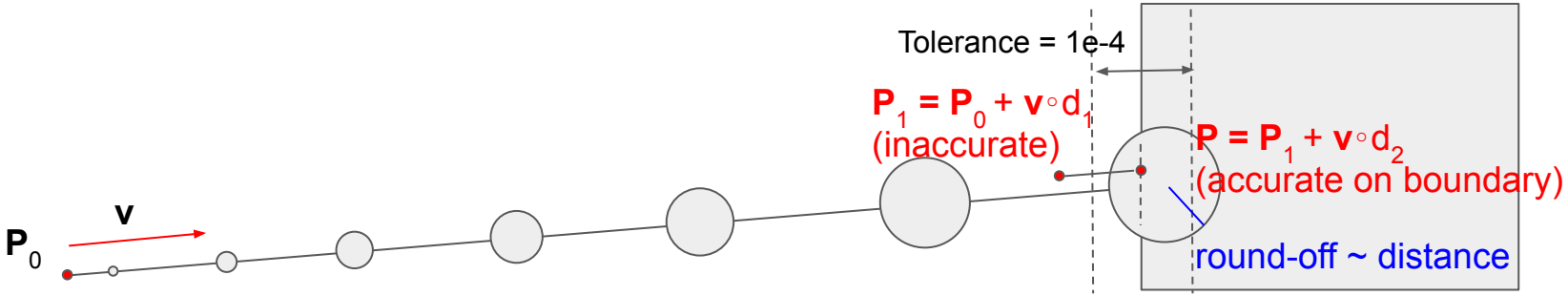
Single precision use in geometry

- Double precision currently has high cost on GPU
  - x32 less operations per clock than single precision on consumer hardware, x2 on high-performance cards
- Is single precision good enough for geometry navigation?
  - Intrinsically yes, but are algorithms stable to rounding errors?
  - To be tested extensively
- Implemented by generalizing *vecgeom::Precision* as type alias, chosen at compilation time (SINGLE_PRECISION=ON)
  - Changing numerical constants (such as the boundary tolerance)
- Several solids unit tests checking algorithms stability against propagation/boundary crossing are failing

andrei.gheata@cern.ch

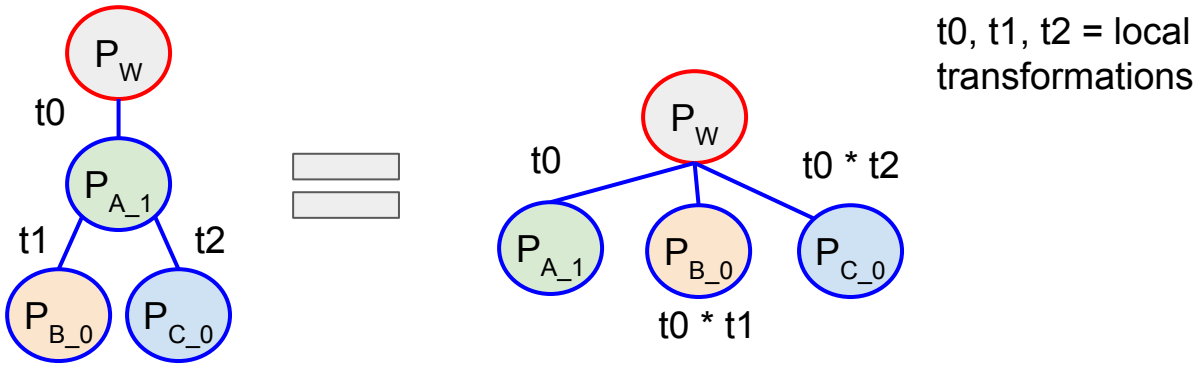
Algorithm stability to rounding errors

- Floating point representation in single precision: 23 bits mantissa + 8 bits exponent + 1 bit sign (vs. 52+11+1 for double precision)
  - As exponent grows, the last rounded significant digit represents a larger (absolute) number
  - As consequence, absolute round-off errors increase for larger numbers, affecting algorithms making cuts based on absolute values
- Typical geometry example: rounding errors for propagated points
  - Strategy: approach solid, then compute distance



GPU-friendly geometry transformations

- Geometry model - un-balanced tree
  - Searches/reductions time highly dependent on state (position/direction
  - Un-balanced computation cost for concurrent queries
- Balancing the queries by:
  - Navigation optimizations per volume, reducing the candidates as logN
  - **Flattening** the hierarchy and using a BVH acceleration structure (same approach used by graphics engines)



andrei.gheata@cern.ch

GPU-friendly geometry transformations

- The support for multiple primitive shape types makes code highly divergent
  - Particles in the same warp may talk to different solids
  - Grouping particle per volume comes with large overheads (GeantV)
- Transforming shapes into polyhedral approximations
  - To understand limitations (e.g. geometry max size, errors)
  - Automatic transformation may become non-trivial for complex setups
  - May generate overlaps
- Third party solutions (NVidia Optix) can be interesting
  - As shown by the previous talk, for steering the GPU simulation application
  - It would be interesting to access components such as the hardware-accelerated BVH in a library-like approach, or their ray scheduling engine

andrei.gheata@cern.ch

Conclusions

- Effort for optimizing VecGeom for GPU workloads ongoing
- First steps already started, this work is one of the 2021 priorities
- The CPU navigation optimizations are in general not suitable for GPU
  - We want to investigate alternative optimization strategies, favoring massive parallelism and minimizing kernel divergence