VecGeom@GPU
- ongoing work -

25th Geant4 Collaboration Meeting

andrei.gheata@cern.ch

GPU-efficient geometry navigation

- Motivated by the need to have accelerator-friendly simulation
- Two main candidates
    - VecGeom: our in-house geometry modeler
        - ~1:1 mapping of scalar C++ CPU object model to GPU based on macros
        - CudaManager handling creating and populating GPU instances from CPU ones
        - Effective for making the GPU understand our CPU code, but unaware of device specificity: memory/cache hierarchies, parallelism models, resources
    - Optix: state of art proprietary ray-tracing software
        - Efficient library allowing efficient scheduling of user kernels (shaders) in a ray-driven pipeline
        - Ray-model intersection handled by hardware-accelerated BVH
        - Promising attempts for optical photon driven simulation (Optiks)

GPU-efficient geometry navigation

- Motivated by the need to have accelerator-friendly simulation
- Two main candidates
  - VecGeom: our in-house geometry modeler - at hand to try now
    - ~1:1 mapping of scalar C++ CPU object model to GPU based on macros
    - CudaManager handling creating and populating GPU instances from CPU ones
    - Effective for making the GPU understand our CPU code, but unaware of device specificity: memory/cache hierarchies, parallelism models, resources
  - Optix: state of art proprietary ray-tracing software - promising, but potentially more effort
    - Efficient library allowing efficient scheduling of user kernels (shaders) in a ray-driven pipeline
    - Ray-model intersection handled by hardware-accelerated BVH
    - Promising attempts for optical photon driven simulation (Optiks)

Feasibility study: a GPU raytracer demonstrator

- Import a geometry setup
- Implement a simple GPU-aware navigator
- Implement some simple “shader models”
- Write a demonstrator running on both CPU/GPU
    - Deal with all possible blockers along the way
- Implement few kernel scheduling scenarios
- Profile the code and understand bottlenecks
- Decide where to to go from here

Feasibility study: a GPU raytracer demonstrator

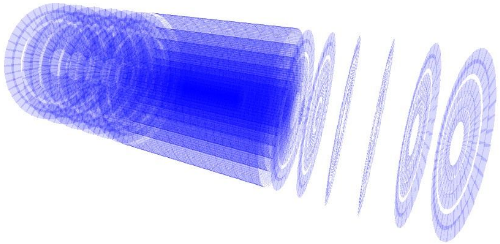
- Import a geometry setup - use our VecGeom GDML importer + trackML geom
- Implement a simple GPU-aware navigator - use a looper w/o optimizations
- Implement some simple “shader models” - specular reflection/ transparency
- Write a demonstrator running on both CPU/GPU
    - Deal with all possible blockers along the way - valid GPU object instances, memory/stack size
- Implement few kernel scheduling scenarios
- Profile the code and understand bottlenecks

} Sheffield GPU hackathon

- Decide where to to go from here - main R&D directions to achieve performance
- Co-developed with Guilherme A. as branch in VecGeom

Some blockers

- Getting the (non-trivial) CPU code to compile and run on GPU…
  - Handling allocations, object copying and synchronization, kernel scheduling
- Handling rays as we would do tracks in simulation
  - Storing large track states for all pixels of a large image requires lots of memory
  - CUDA Exception: Lane User Stack Overflow - deep stacks, abusing local variables, …
  - CUDA_EXCEPTION_5, Warp Out-of-range Address.
  - RaytraceBenchmark received signal CUDA_EXCEPTION_6, Warp Misaligned Address
- Got it working eventually...



Sheffield hackathon

- Organizers: NVidia + Sheffield University
- 8 teams (scientific areas) with few mentors each
- 3 weeks: general presentations, mentoring, work, support, meetings with experts, reports
- Got some insight on profiling tools usage: Nsight Systems & Nsight Compute
- Learned a lot, got useful contacts and links, understood performance bottlenecks

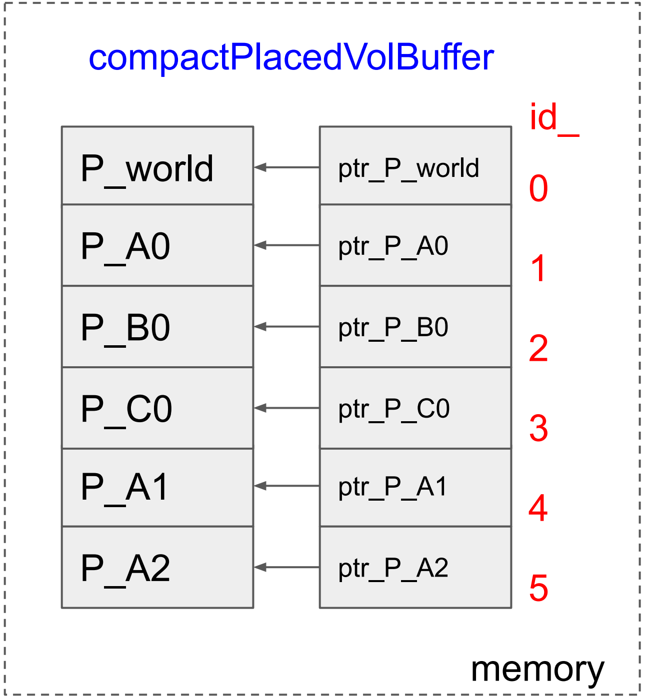
Sheffield hackathon takeaways

- Kernel scheduling should be done carefully, minimizing the need for synchronization to maximize occupancy
- Kernels of smaller size/complexity to be preferred to large ones, giving the opportunity to more cores to run concurrently
- Our scientific code produces high register pressure, overspilling to memory. Per-thread optimal settings for allocated registers is a compromise to be found per card type.
- Double precision is way too expensive on GPU
  - NVIDIA charging premium for double precision enabled cards
  - “Emulating FP64 with double-float arithmetic is conservatively 20x slower than native float arithmetic”

Minimizing GPU memory footprint

- GPU workflow = massive parallelism on tracks
- Handling a large number of track states - O(million) - concurrently is inevitable
- Geometry part of the state is considerable
    - Array of placed volumes indices in the geometry hierarchy
        - allowing global transformation computation & per-level navigation
        - Size ~ maximum geometry depth (15-20 for LHC setups)
- Ideally need two navigation states/track
    - Pre-step and post-step locations

Navigation state handling in VecGeom



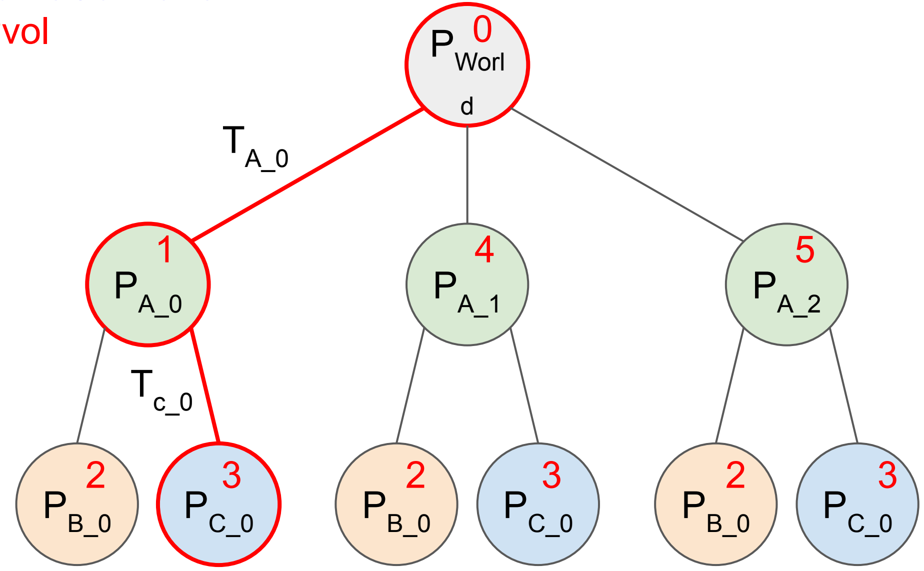
compactPlacedVolBuffer

		id_
P_world	ptr_P_world	0
P_A0	ptr_P_A0	1
P_B0	ptr_P_B0	2
P_C0	ptr_P_C0	3
P_A1	ptr_P_A1	4
P_A2	ptr_P_A2	5

memory

Index2PVolumeConverter

id_ ↔ ptr_Pvol



Sequential physical volume index stored as data member

Navigation history “levels”

NavigationState {fCurrentLevel = 3 / 3, fPath = {0, 1, 3}} :: TopTransform ($G_{013} = T_{A\_0} * T_{c\_0}$)

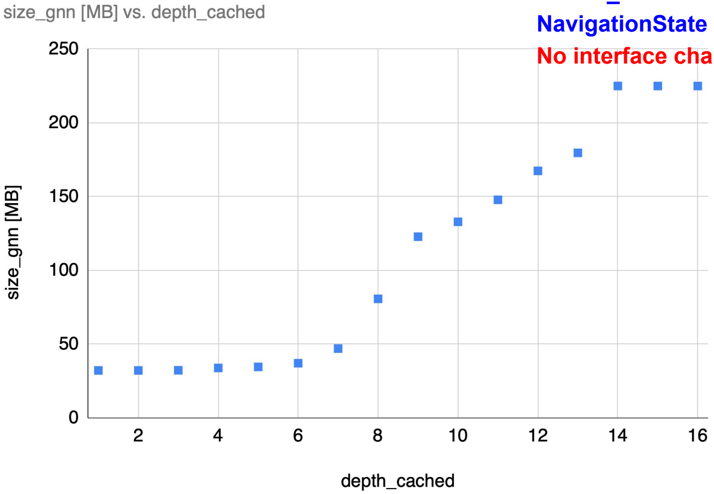
Tradeoff: state becoming an index in a global table

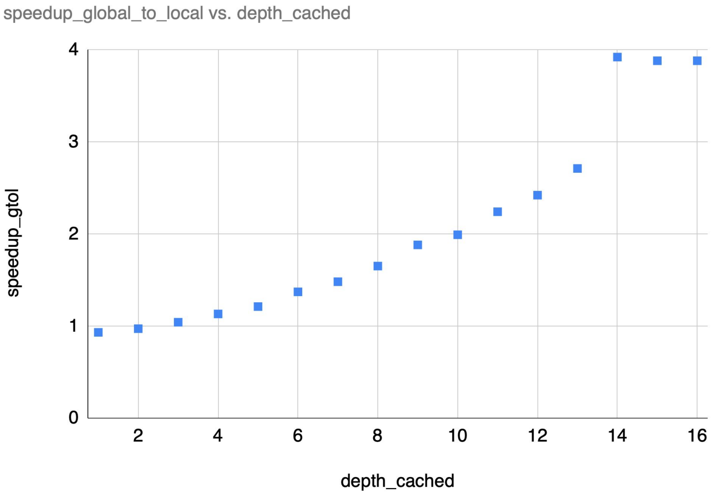
- The physical volumes can be enumerated and their info stored in a table
  - Track state becomes a 32 bits index in this table -> global navigation index
  - The table can become large for big geometry setups
  - Bonus: global transformations can be also cached down to a given depth -> speedup

-DUSE_NAVINDEX=ON

NavigationState becomes a type alias

No interface changes



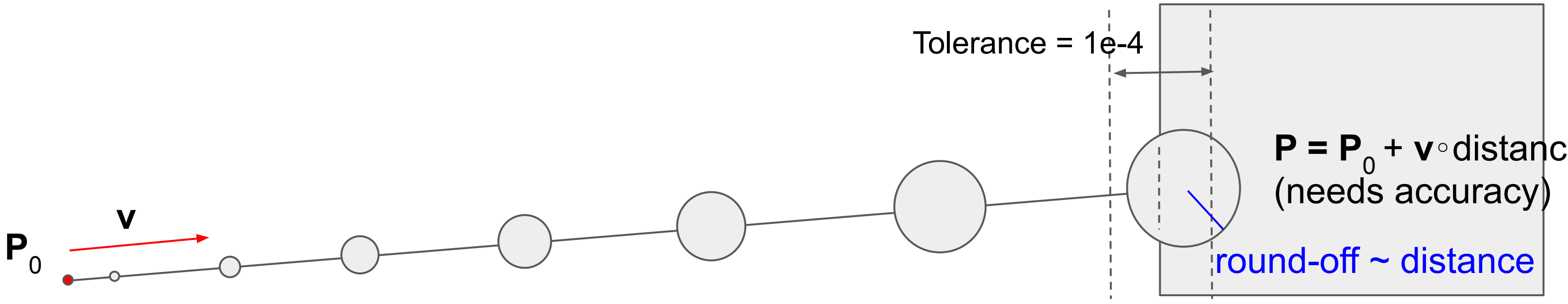


VecGeom in single precision mode

- Much to gain on GPU
  - Is single precision good enough for geometry navigation?
  - How difficult to implement in VecGeom?
- Many possible approaches - algorithms templated on the data type
  - Mixing single & double precision interfaces however difficult to maintain/validate
  - Simplest way to test: generalizing vecgeom::Precision as type alias, chosen at compilation time
- Had to touch most VecGeom classes, preserving interfaces
  - -DSINGLE_PRECISION=ON compiling OK
  - Changing numerical constants (such as kTolerance)
  - Many solids unit tests checking algorithms stability against propagation/boundary crossing are failing

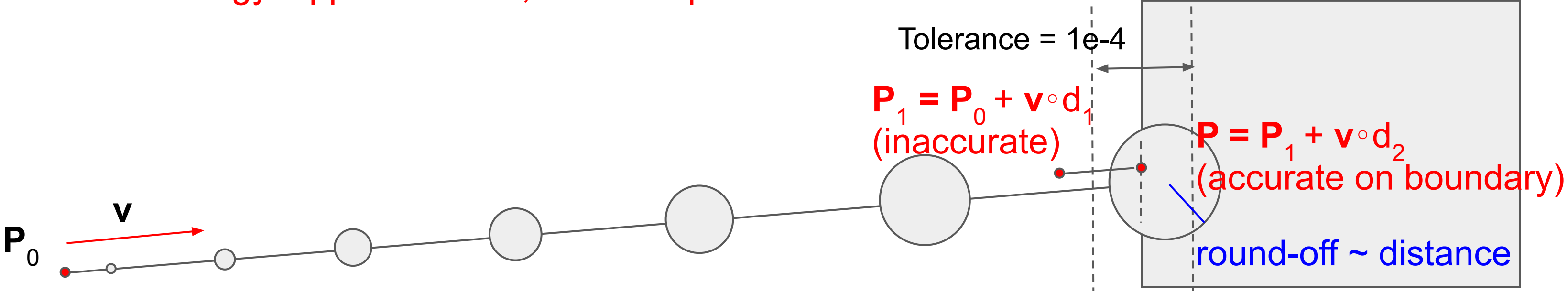
Rounding errors

- 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, arithmetic operations involving large numbers have large round-off errors
- Typical geometry example: rounding errors for propagated points



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, arithmetic operations involving large numbers have large round-off errors
- Typical geometry example: rounding errors for propagated points
  - Strategy: approach solid, then compute distance



Tolerance = 1e-4

$\mathbf{P}_1 = \mathbf{P}_0 + \mathbf{v} \circ d_1$ (inaccurate)

$\mathbf{P} = \mathbf{P}_1 + \mathbf{v} \circ d_2$ (accurate on boundary)

round-off ~ distance

Conclusions

- Possible to use VecGeom on GPU
    - A demonstrator ray-tracing utility using arbitrary geometry was developed
- Work started to make geometry efficient for simulation on GPU
    - Smaller navigation state caching transformation matrices
    - Single precision navigation
- Need for navigator class optimized for GPU
    - Using BVH or voxelization
- Most of these optimizations will become available in Geant4 with the native VecGeom navigation
    - A version of the global navigation index table with transformation caching could be implemented in Geant4 native as well