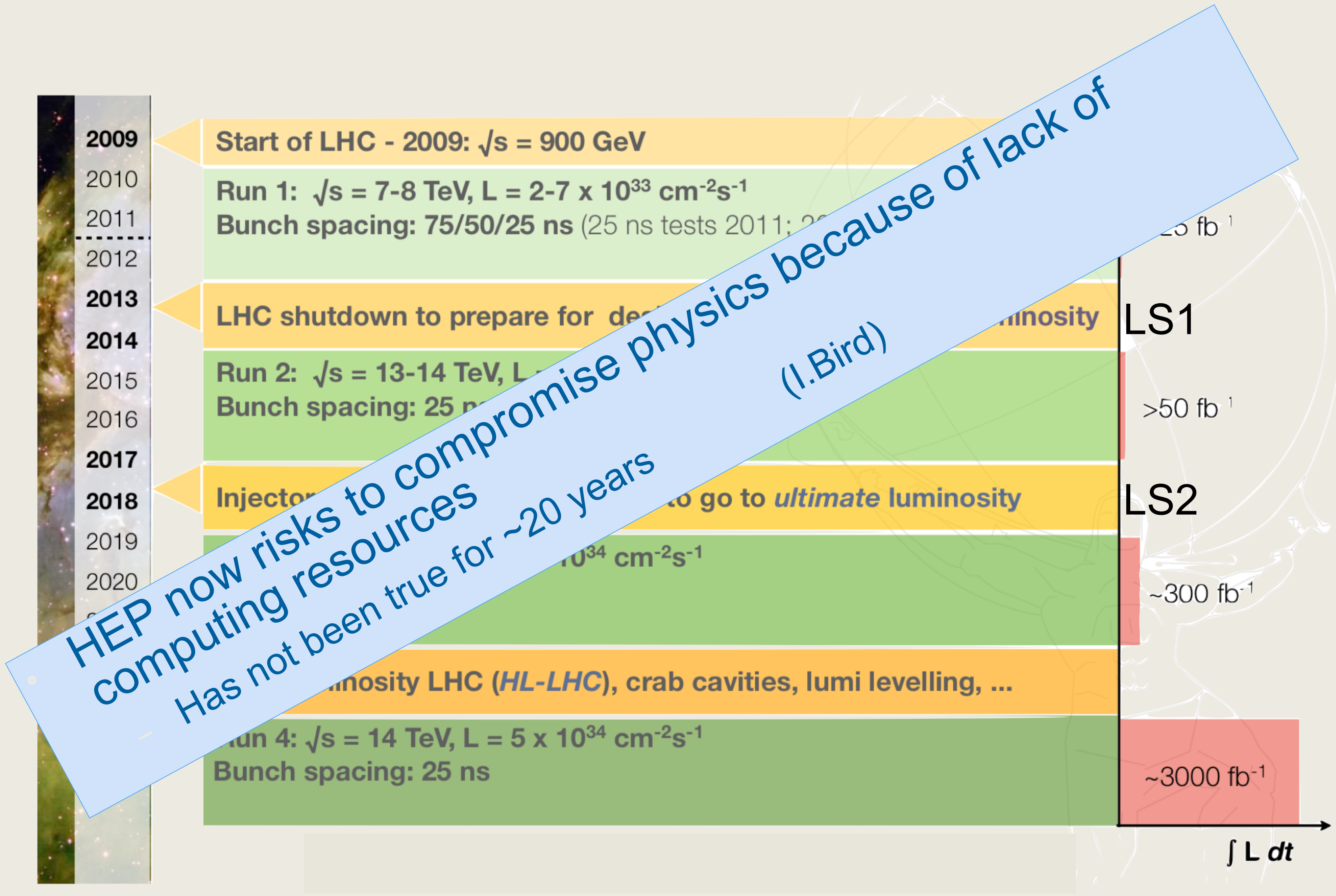
Simulation Vector Prototype

Philippe Canal / FNAL
assembled with
F. Carminati, A. Gheata, S. Wenzel / CERN
S.Y. Jun/ FNAL

Outline

- Overview
- Scheduler
- Geometry
- Physics Processes

A luminous future for HEP...



2009 Start of LHC - 2009: √s = 900 GeV

2010
2011
2012
Run 1: √s = 7-8 TeV, L = 2-7 x 10^{33} cm^{-2}s^{-1}
Bunch spacing: 75/50/25 ns (25 ns tests 2011; 2

2013
2014
LHC shutdown to prepare for de

2015
2016
2017
Run 2: √s = 13-14 TeV, L
Bunch spacing: 25 n

2018
Injector to go to *ultimate* luminosity

2019
2020
10^{34} cm^{-2}s^{-1}

inosity LHC (*HL-LHC*), crab cavities, lumi levelling, ...

un 4: √s = 14 TeV, L = 5 x 10^{34} cm^{-2}s^{-1}
Bunch spacing: 25 ns

25 fb^{-1}

LS1

>50 fb^{-1}

LS2

~300 fb^{-1}

~3000 fb^{-1}

∫ L *dt*

HEP now risks to compromise physics because of lack of computing resources

Has not been true for ~20 years

(I.Bird)

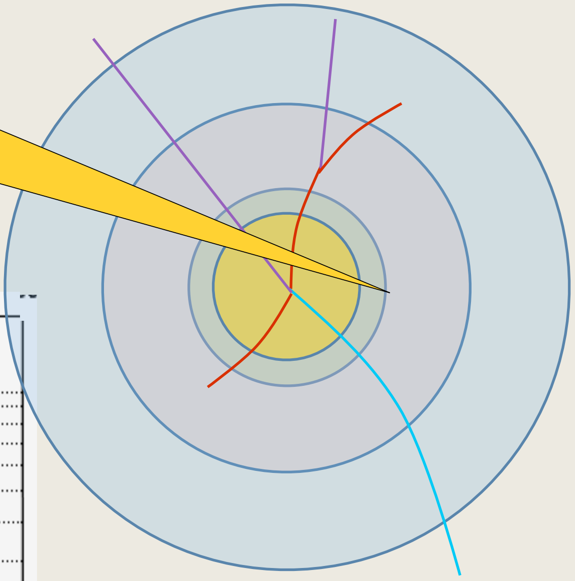
A fresh look at the Simulation

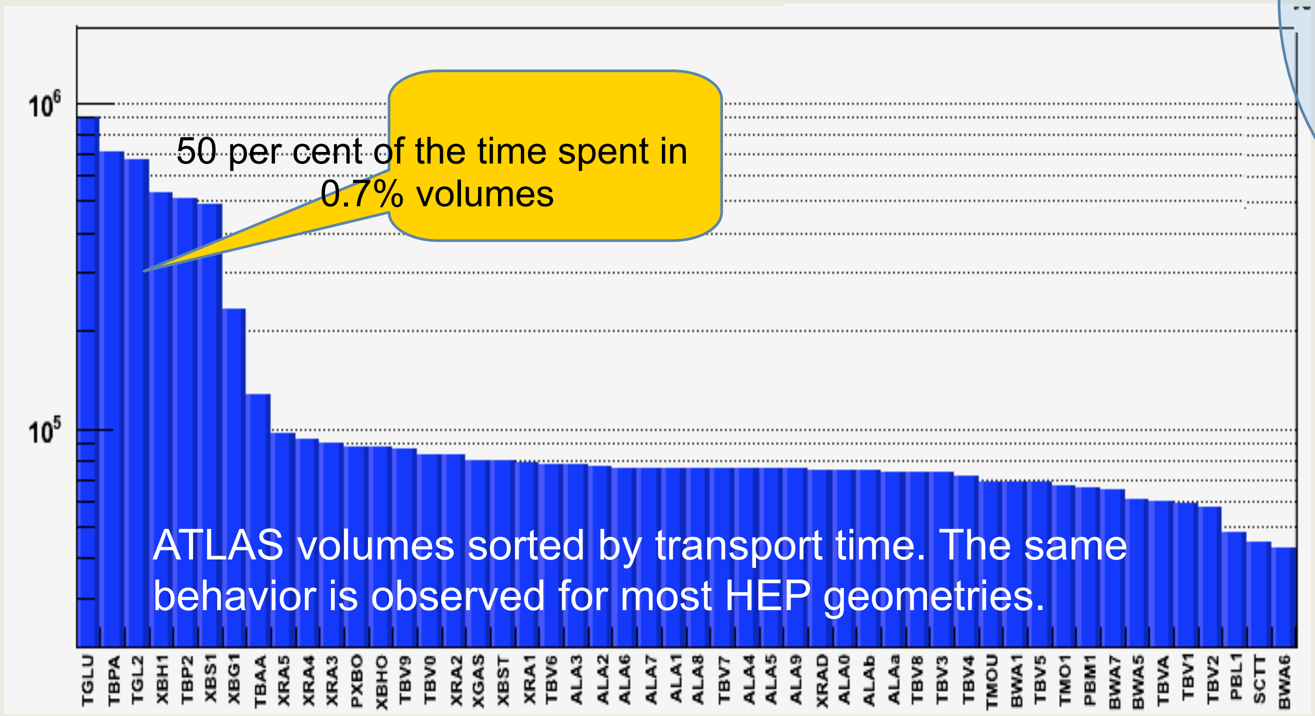
- More than a factor 10 increase expected in the simulation needs in the next few years!
- The most CPU-bound and time-consuming application in HEP with large room for speed-up
  - Largely experiment independent
  - Precision depends on (the inverse of the sqrt of) the number of events
- Grand strategy
  - Explore opportunities with no constraints from existing code
  - Expose the parallelism at all levels, from coarse granularity to micro-parallelism
  - Integrate slow and fast simulation to optimise both in the same framework
  - Explore if-and-how existing physics code (GEANT4) can be optimized in this framework
- Improvements (in geometry for instance) and techniques are expected to feed back into other HEP applications

Classical HEP transport is mostly local !

- Event- or event track-level parallelism will better use resources but won’t improve these points

- Geometry navigation (local)
- Material – X-section tables
- Particle type - physics processes



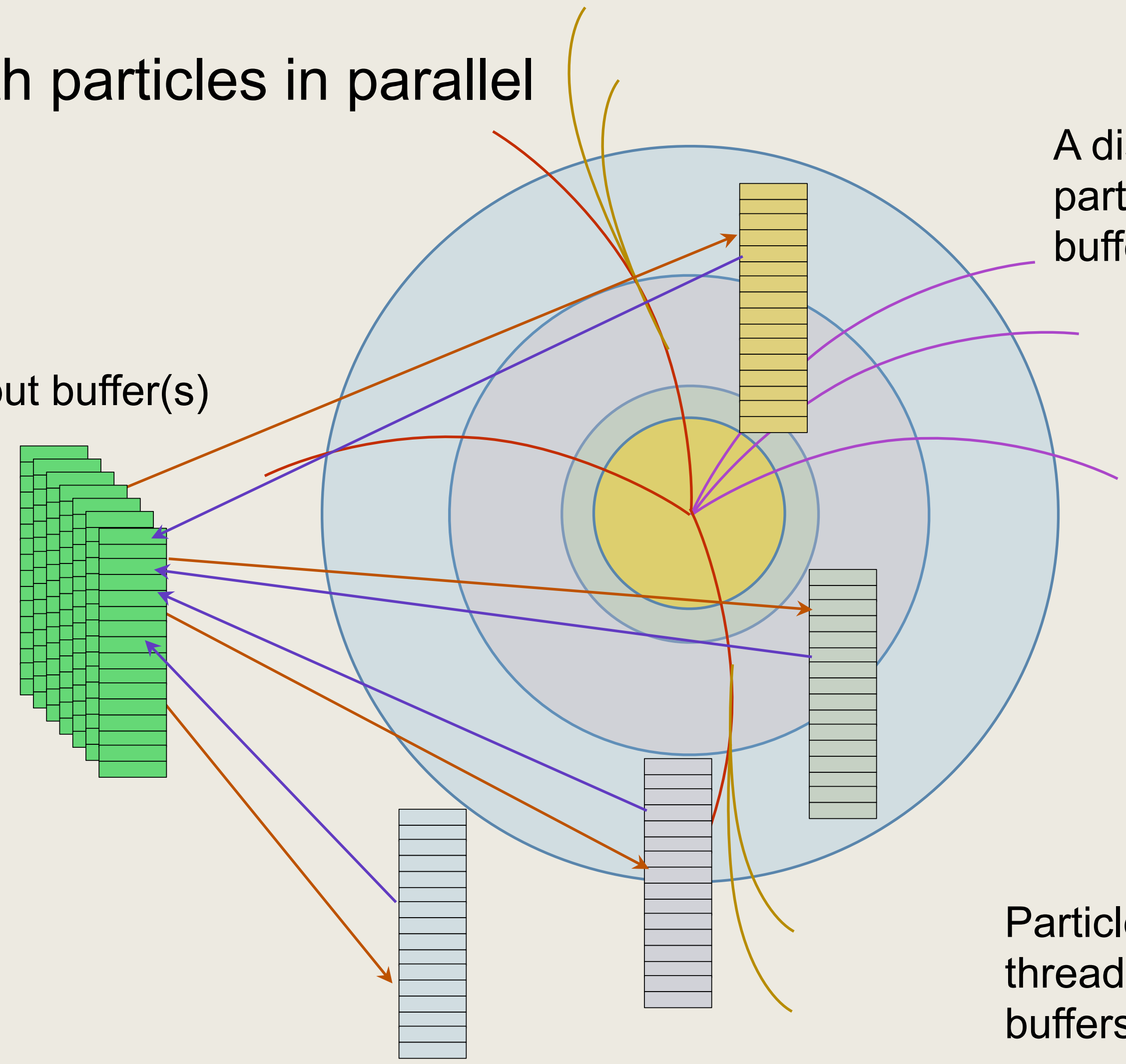


ATLAS volumes sorted by transport time. The same behavior is observed for most HEP geometries.

- Navigating very large data structures
- Cache misses, No locality
- OO abused: very deep instruction stack
- Existing code very inefficient (0.6-0.8 IPC)

Introduce "basketised" transport

Deal with particles in parallel



A dispatcher thread puts particles back into transport buffers

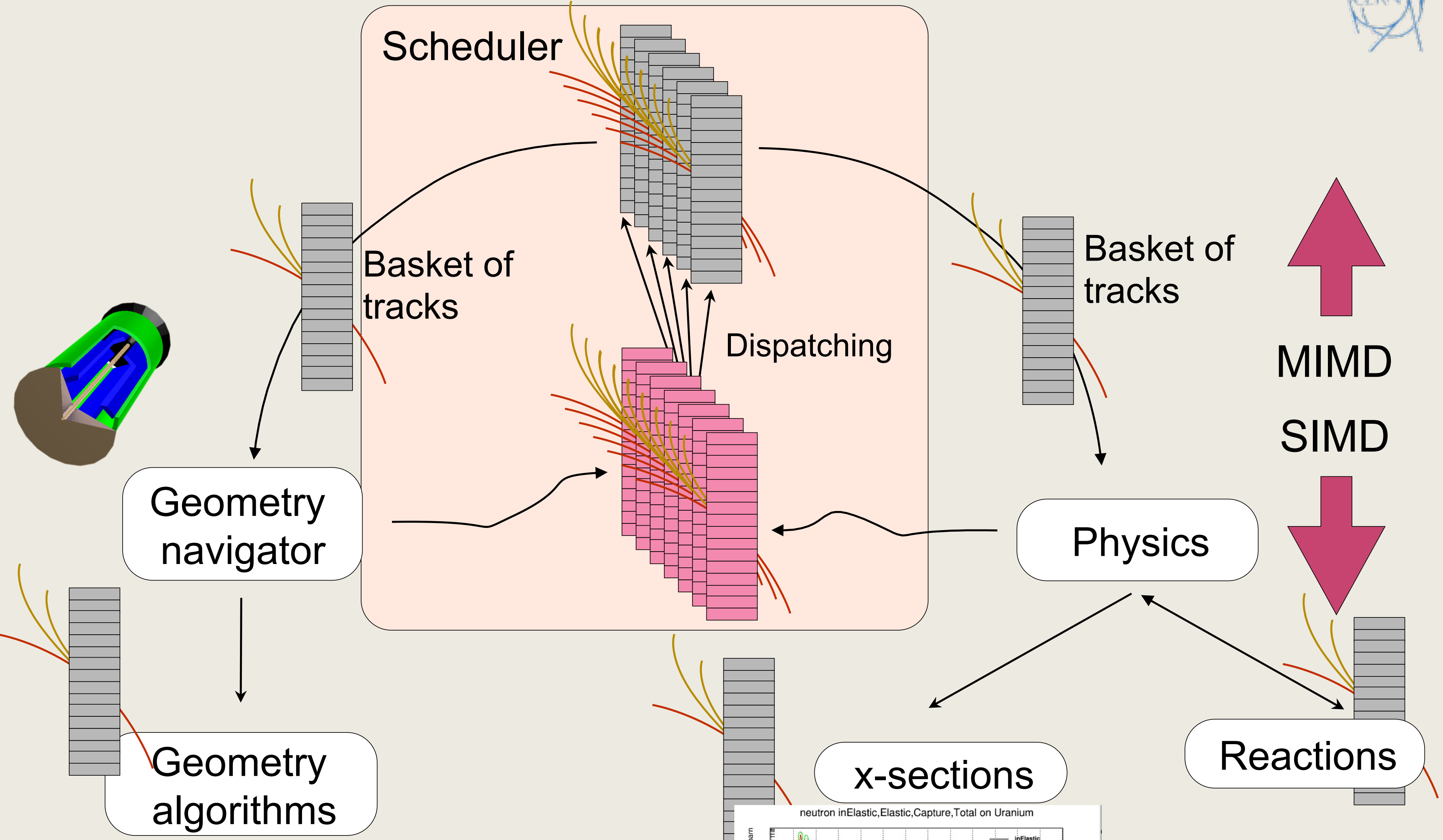
Everything happens asynchronously and in parallel

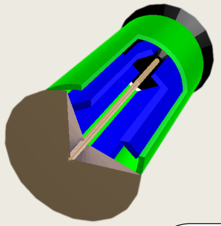
The challenge is to minimise locks

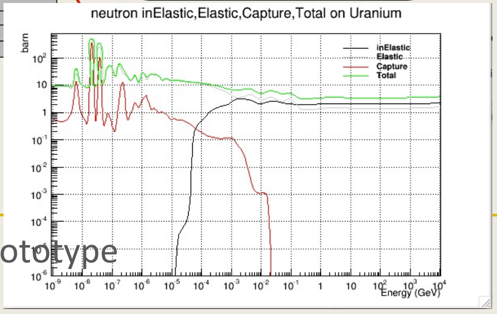
Keep long vectors

Avoid memory explosion

Particles are transported per thread and put in output buffers


Scheduler
Basket of tracks
Basket of tracks
Dispatching
MIMD
SIMD
Geometry navigator
Physics
Geometry algorithms
x-sections
Reactions




neutron inElastic,Elastic,Capture,Total on Uranium
barn
inElastic
Elastic
Capture
Total
Energy (GeV)

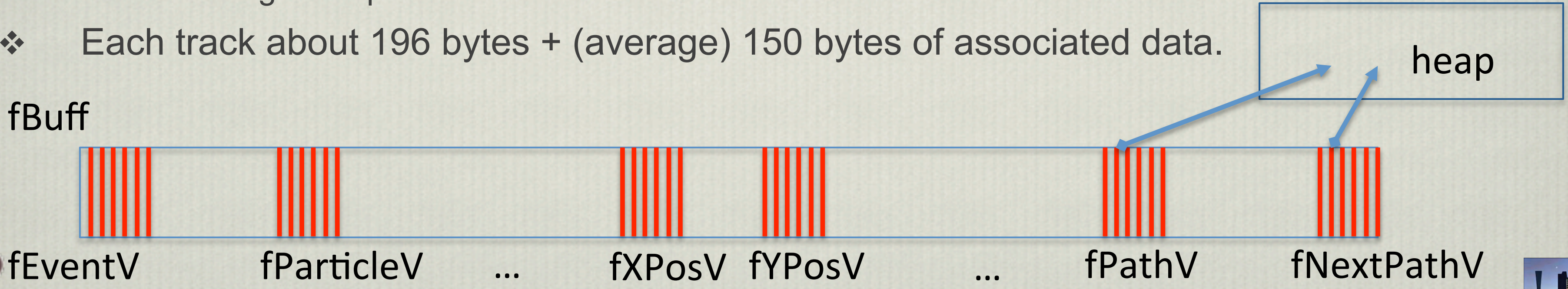
Scheduler

Outlook

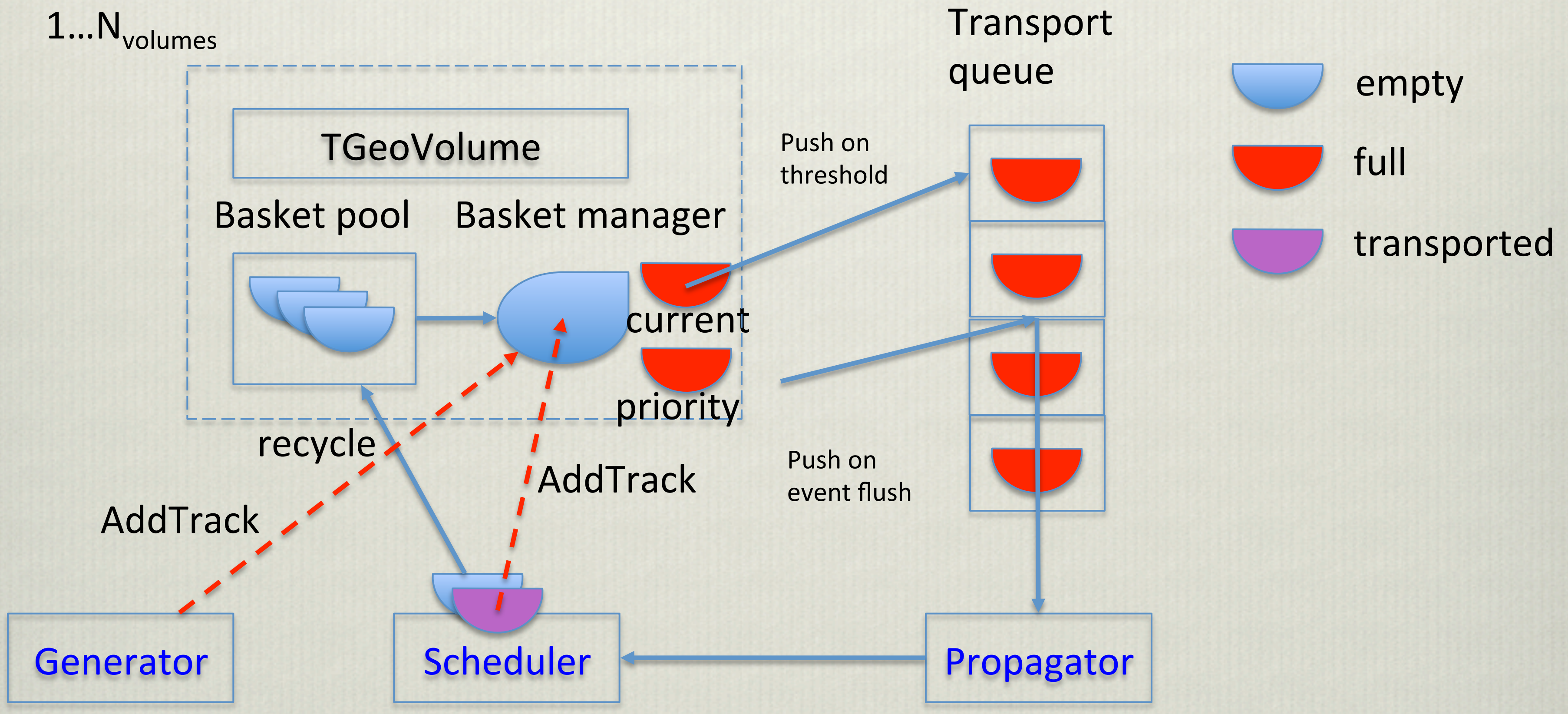
- Event structure and containers
- Baskets and data queues
- Basket managers (per LV)
- Transport (physics and geometry) and track phases
- Scheduler class and scheduling thread
- Scheduling policies and multithreading
  - Connection to vector geometry
  - Connection to physics
  - Connection to GPU prototype
- Monitoring

Baskets of tracks

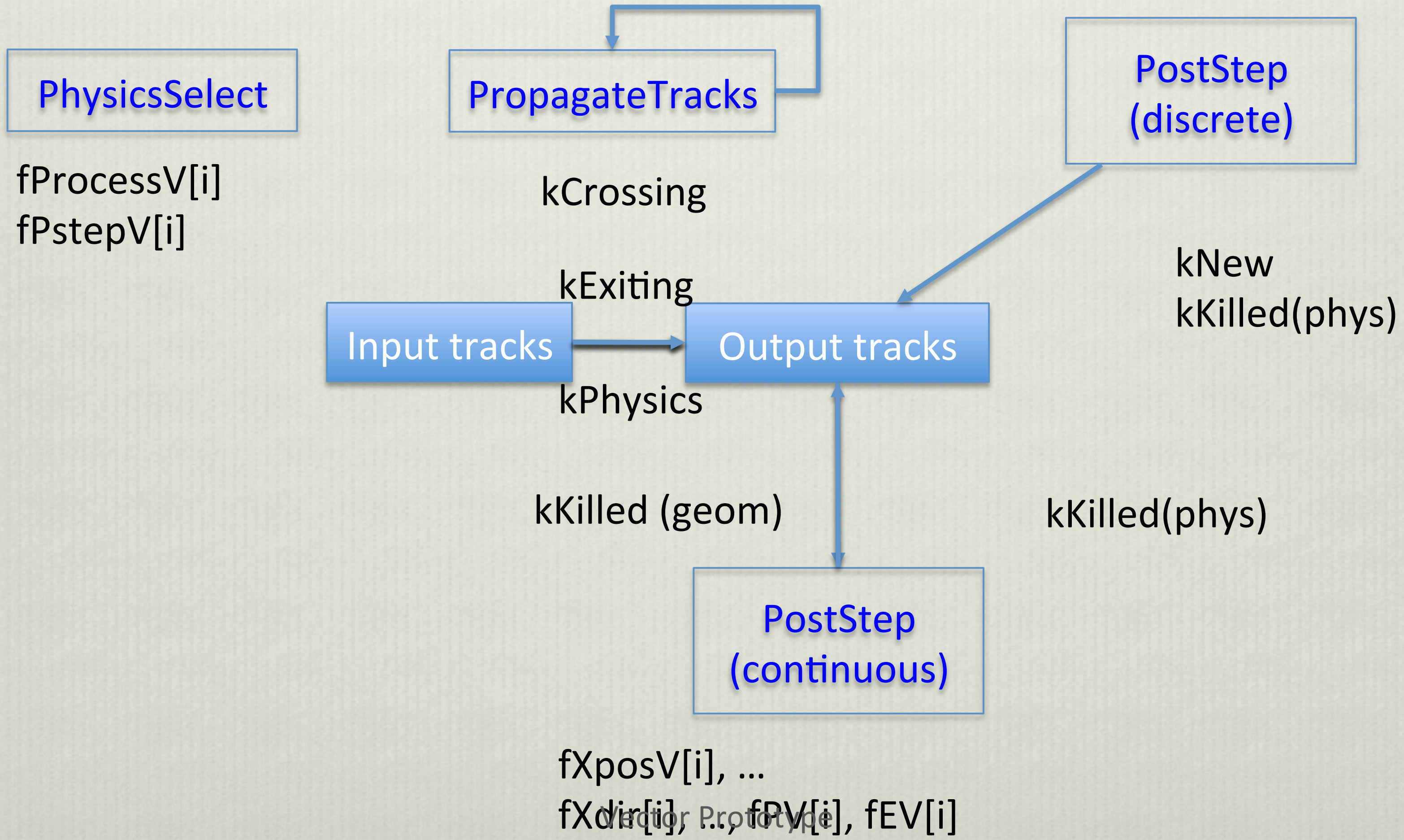
- Unit of work for the transport thread
  - Containing only tracks in a single logical volume
- Current implementation: basket = input and output GeantTrack_v
- GeantTrack_v:
  - SOA matching GeantTrack using internal memory management for vector performance
  - Buffer management: allocate, copy, resize, import and export and remove GeantTrack
  - Management of holes (i.e. tracks that finished transport in the current propagation cycle)
    - compact Vector when not efficient-> compact tracks (using bit container)
  - Sorting by track status, needed to vectorize different propagation stages
    - E.g. transport of neutral tracks
- Each track about 196 bytes + (average) 150 bytes of associated data.



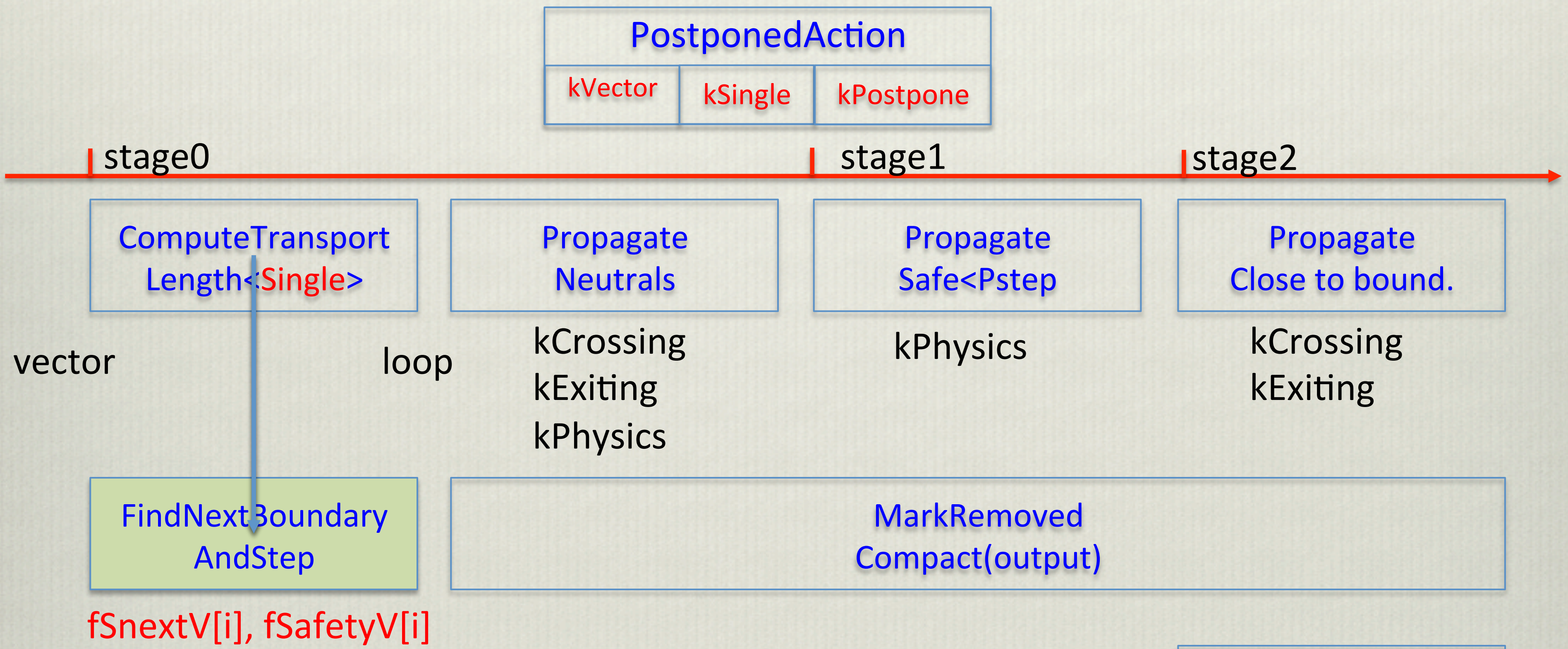
Basket lifecycle

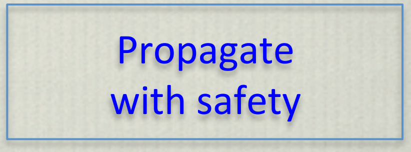


Basket transport



PropagateTracks





- kVector – continue in vector mode
- kSingle – call PropagateTracksSingle at the given stage
- kPostpone – copy remaining tracks to output
- MarkRemoved + Compact – compact holes and copy these tracks to the output

Scheduler

- Pulls transported baskets, dispatches tracks to basket managers per volume
- Single thread, one scheduler/ multiple thread, multiple schedulers
  - TBB task approach, to be investigated after understanding and tuning the scheduler with real physics
- Applying policies for:
- Workload balancing
  - Divide the work evenly to scale with number of workers
  - Queue control: garbage collection on work queue depletion
  - Improvement: schedule physics as separate task (process selection and discrete processes post-step)
- Memory management
  - Not active currently, the idea it to trigger hit/digits collection and memory cleanup on thresholds
- Keep large vectors
  - Raise transportability thresholds per volume
  - Postpone sparse tracks when not in garbage collection mode
- Trigger single track mode when vectorization gives just overhead

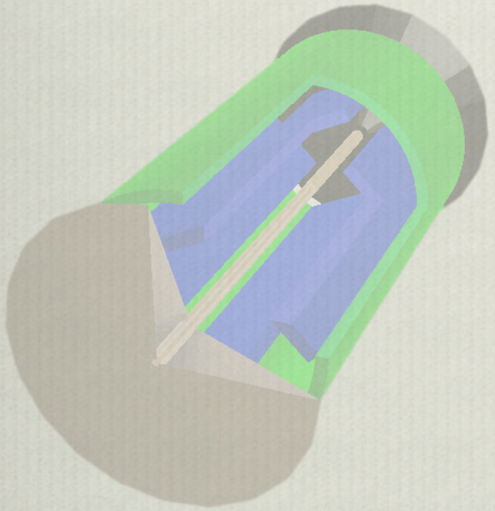
GPU Connector to an External Scheduler

- GPU connector is an interface to the Vector Prototype
- Challenges
  - different geometry implementation – need to translate location and history information back and forth
  - difference in data layout
  - only a subset of particle can be handled
  - (ideal) bucket size very different from CPU
  - try to maximize kernel coherence
- Implementation
  - stage particles in a set of buckets
    - list and type of bucket is customizable, one idea is to buckets based on particle/energy that have a common (sub)set of likely to apply physics.
    - within this baskets the particles are placed in order/group given by the VP
  - delay the start of a kernel/task until it has enough data or has not received any new data in a while
  - to maximize overlap uploads are started for a task after handling a CPU basket

Monitoring

- Internals of track dynamics
  - Track counters in different phases, efficiency to prioritize events
- Basket dynamics
  - Number of baskets, size per volume, transportability threshold
  - Vector size
- Memory monitoring
- Multithreading efficiency
  - Locks and waits analysis, concurrency
- New class GeantTrackStat
  - Used if GEANT_DEBUG=1
  - Track counters for number of tracks/steps per event, read in the different track phases
- Separate monitoring thread with graphics to be done

Vectorizing and optimizing detector geometry classes



Sandro Wenzel

Geometry - Outline

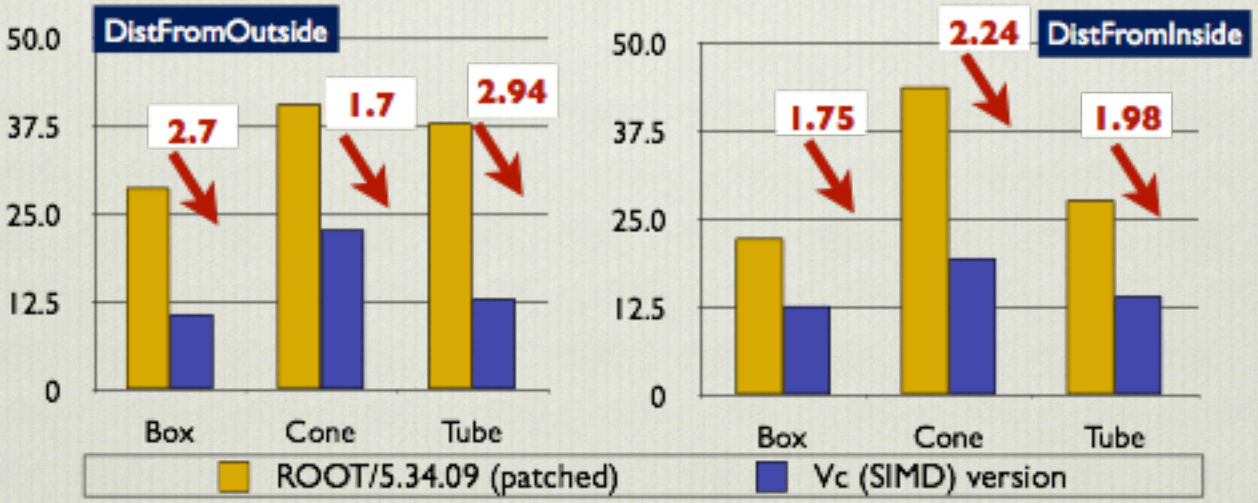
- First Results
- Challenges on the path to continue
- Arguments for template based techniques in future geometry development
  - Template class specialization for performance increase / better vectorization (this talk)
  - Template techniques for code generality (future talk)

focus on ideas rather than

many performance numbers

First Results

- ❖ Activity since spring 2013 focused on studying feasibility of vectorizing (primitive and higher-level) geometry algorithms for the Vector and GPU simulation prototypes
- ❖ Demonstrated for a couple of shapes (box, tube, cone) that this is very possible indeed with good performance gains

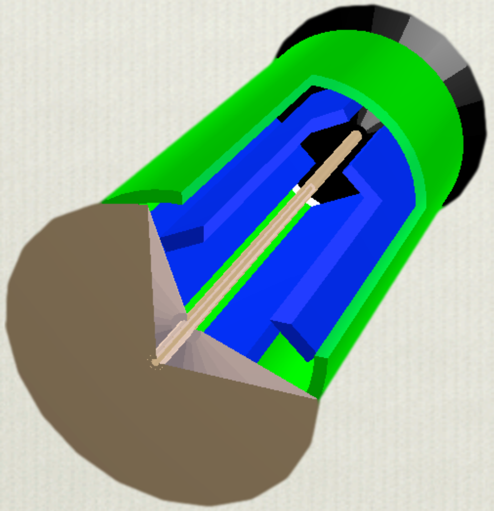


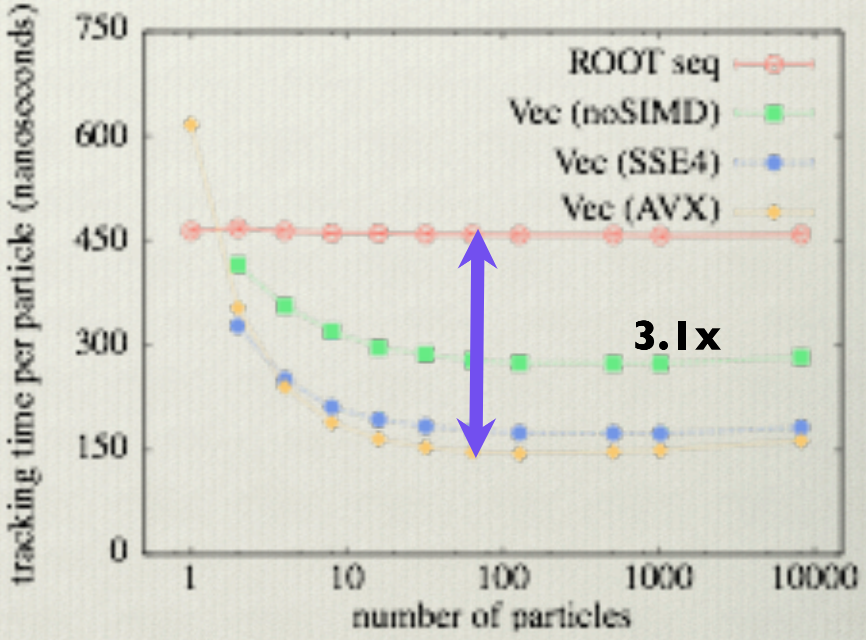
- ❖ This came at the cost of totally rewriting the routines to make them vector friendly
- ❖ Adopted programming model: Vc library, Intel Cilk Plus Array notation

First Results

- **higher-level vector performance benchmark:**
  - (simplified) navigation of vectors of particles in a simplified detector with daughter shapes

max SIMD speedup of 3.1





- **How much better can we do?**
  - profiling@Intel: very good already; maybe try to reduce unnecessary operations (reduce branches; floating point ops)
  - much of the ideas here are based on this original advice

Further goals / Challenges

- Start a systematic effort to produce a "prototype ready" vectorized geometry library for both CPU and GPU.
  - provide a library with vector interfaces for important geometry function.
  - provide a library targeting the CPU + CUDA at the same time
  - achieve best performance

- **Main challenges ahead**:
  - current code does not serve for SIMD vectorization or SIMT -- **there are often too many branch levels** (see for instance tube::distanceToIn in Geant4/Usolids)
  - hence, **total code rewrite necessary**
  - **complete revalidation necessary**

challenges continued ... / implications

- Targeting different backends and instructions sets (vector, GPU, scalar) sounds like a lot of code repetition if we continue to code the way it was done in the past
  - will be a nightmare for maintenance and testing

- We should hence (these points are related)
  - write code which is **generic**
    - functions which work with scalar or vector arguments
  - **reuse code** as much as possible **without performance loss**
    - example: many kernels for tube / cone / polycone are shared and should be written only once (without function calls)
    - write code which is **composable from smaller "codelets"**

taken together these requirements points to C++ templates

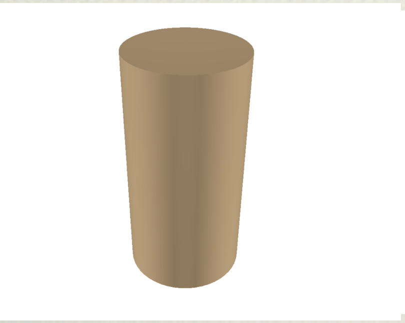
- A **templated library** is perfect to achieve/increase performance:
  - template class specialization allows to produce very optimized code for particular shapes / matrices, etc.

- A **templated library** is a good approach to solve the general challenges presented:
  - one can write generic code easily with template functions
  - one automatically writes easily reusable("inlineable") code since templates usually requires coding in header files
  - can solve the problem of different backends (CPU/GPU)

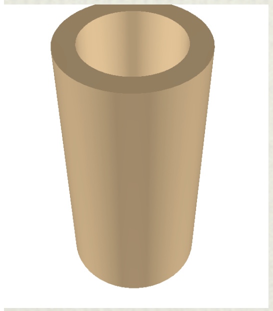
Example of template class specializations

Motivation for class specialization -- reduction of branches --

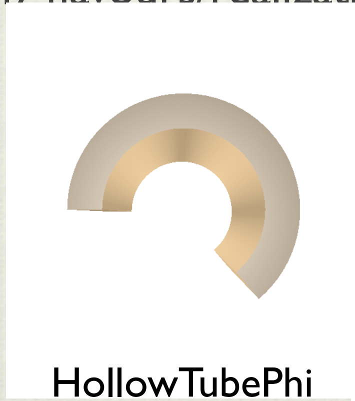
❖ shape primitives come in many flavours/realizations (here for tube)


FullTube

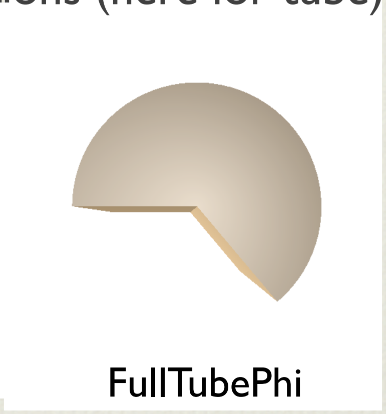
15%


HollowTube

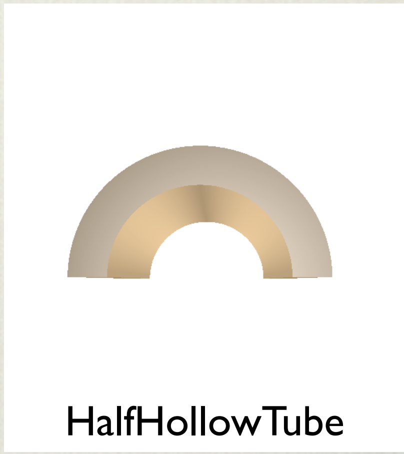
10%


HollowTubePhi

5%


FullTubePhi

68%


HalfHollowTube

few

statistics generated from Atlas, CMS, ALICE, LHCB geometries (ftp://root.cern.ch/root/geometries.tar.gz)

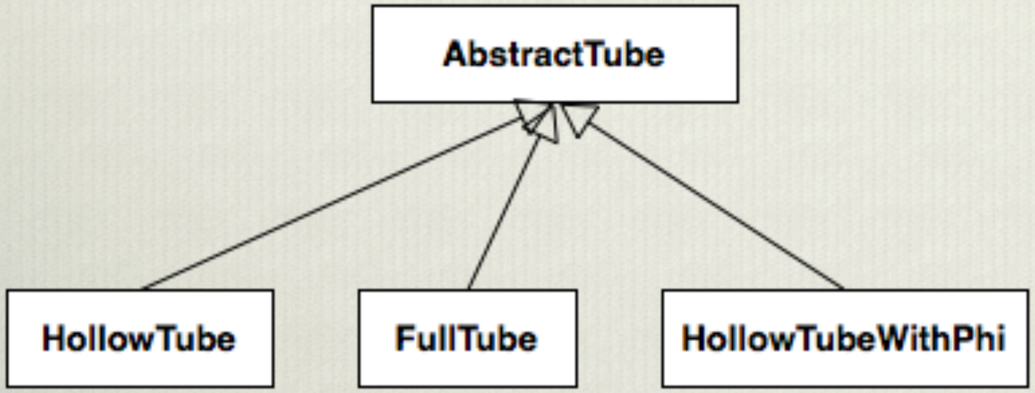
❖ in reality current libraries (USolid,Root) **implement one or few generic tube classes** -- mainly to have few code lines to maintain

❖ a lot of the branches (if statements) are static in the sense that they test properties of the tube instance ("**if I am hollow then; else** ")

❖ **such static branches reduce performance** (we will see by how much)

possibilities to make algorithms more specialized

- a way to get rid of many branches would be to introduce a separate class for each important tube realization
  - **canonical approach:** solution with handwritten separate classes

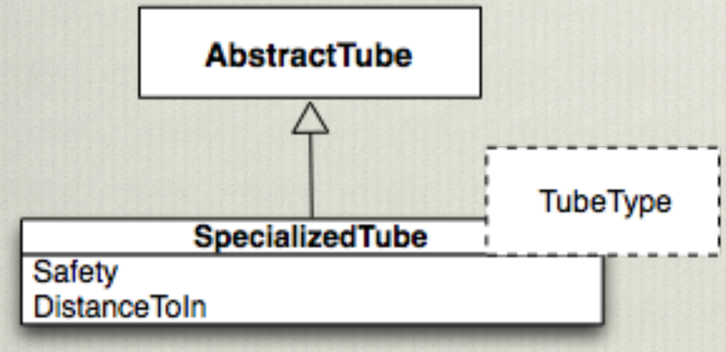


AbstractTube *t = new FullTube();

performance

code repetition

- **alternative idea:** solution with templated classes



AbstractTube *t = new SpecializedTube<FullTube>();

performance

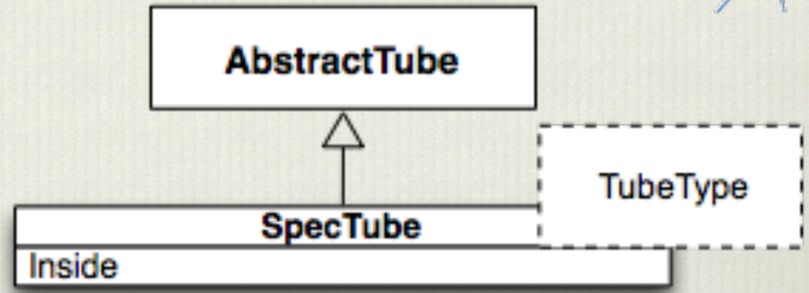
(almost) no code repetition

user does not even need to care about special classes / should use factory methods

AbstractTube *t = GeoManager::CreateTube(...);

common code - many realizations

```
template<typename TubeType>
class
SpecTube{
 //  ...
 bool Inside( Vector3D const & ) const;
 //...
};
```



❖ sharing code between classes with compile-time branches (scalar toy example)

```
template<typename TubeType>
bool SpecTube<TubeType>::Inside( Vector3D const & x) const
{
   // checkContainedZ
   if( std::abs(x.z) > fdZ ) return false;

   // checkContainmentR
   double r2 = x.x*x.x + x.y*x.y;
   if( r2 > fRmaxSqr ) return false;

   if ( TubeType::NeedsRminTreatment )
   {
      if( r2 < fRminSqr ) return false;
   }

   if ( TubeType::NeedsPhiTreatment )
   {
      // some code
   }
   return true;
}
```

we can express "**static**" **ifs** as **compile-time if statements (e.g. via const properties of TubeType)**

gets optimized away if a certain TubeType does not need this code

compiler creates different binary code for different TubeTypes

Different example for class specialization -- reduction of floating point operations --

- next to branch reduction; can find many examples where specializing code can be beneficial to save many floating point operations
- example: coordinate transformations between coordinate systems of different shapes
  - known to consume a considerable time (in simple geometries) -- Laurent Duhem@Intel
  - advice: reduce the number of useless multiplications
- often coordinate transformations are treated as a generic "4x4 matrix times a vector" operation (some exceptions in ROOT)

GeneralTransformation ← treating every transformation by general code means ~9 multiplications + ~9 additions per cartesian point

Some performance evaluation for tube

- ❖ with template approach have now **vectorized all realizations of tubes in one go** (previously only simple tubes)

- ❖ speedup of calculating distances of 1024 particles to a placed tube in a world volume (with a high hit rate of 80%)

- ❖ ratio of runtime for vector kernels: **non-templated / templated**

FullTube	~1.15
HollowTubeWithPhi	~1.16
HalfHollowTube	~1.24

benefit from templating the tube (first estimate - this might be depend on many circumstances + parameters)

- ❖ **some preliminary speedups compared to USolids scalar**

HollowTubeWithPhi	~2.7
HalfHollowTube	~2.6

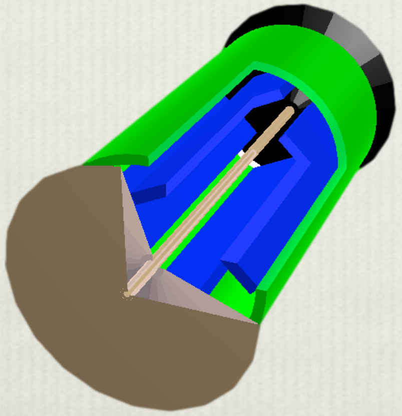
benefit from vectorizing + templating the tube (on AVX)

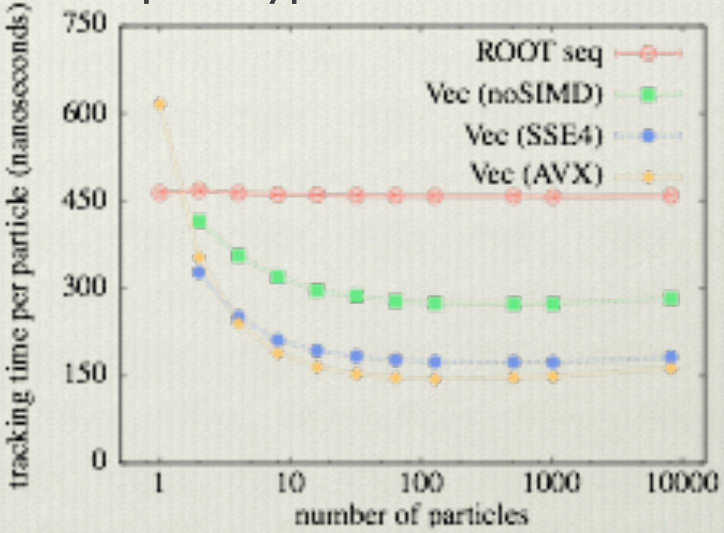
⇨ these SIMD speedups match our expectations

Benchmark revisited

- an initial version of templated vectorized geometry has been finished (shape + coordinate transform specialization) https://github.com/sawenzel/VecGeom.git

- able to readdress CHEP13 benchmark with this new prototype





old status: max speedup = 3.1

new status: **relative performance increase by ~30% (seen for 16, 64, 1024 particles)**

⇨ new status: max speedup ~ 4

- **the template technology gives the extra kick to vectorization !!**

some important implications

this is nice, but...

- unavoidable facts (on the negative side):
  - templates require a rethinking of how we implement a geometry library
  - one needs to code a lot in header files which will stress the compilers
  - currently this is an incompatible programming style compared to existing libraries (USolids, ROOT)
  - the binary code size increases (a lot) - need to study negative impact of this
  - some implications for users unavoidable (avoid new operator in favour of factories ...)

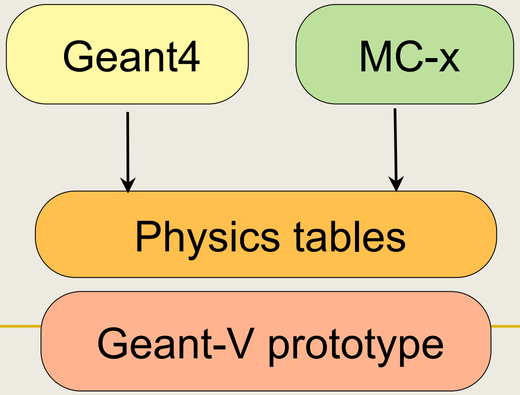
on the other hand...

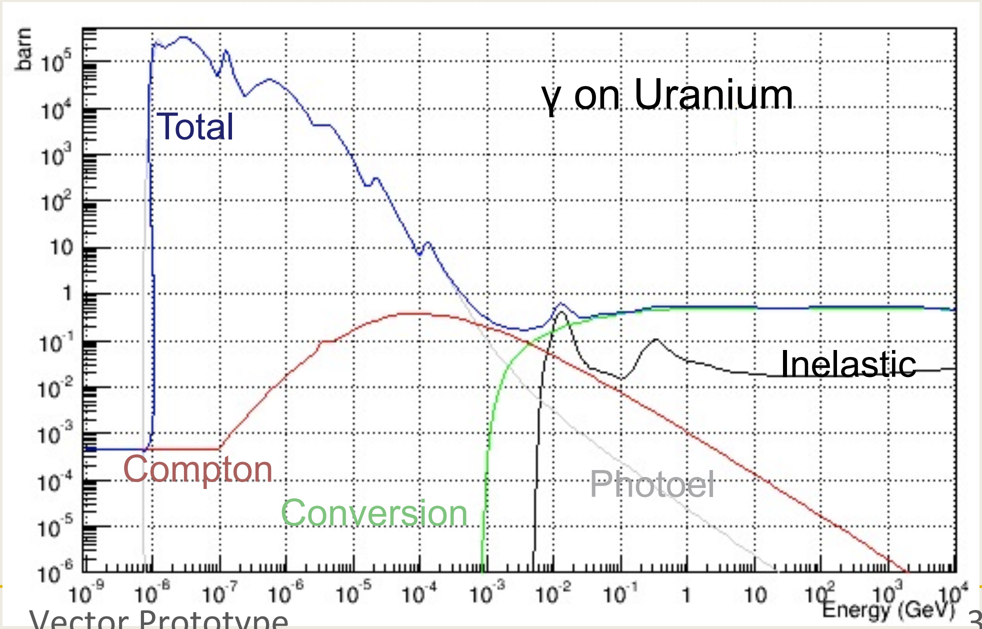
- coding in header files has many positive side effects:
  - code can be shared much simpler between different backends/languages such as C++/CPU and CUDA/GPU
  - code can be reused much simpler in different algorithms (by inlining)

Physics Processes

Physics

- A lightweight physics for realistic shower development
  - Select the major mechanisms
    - Bremsstrahlung, e+ annihilation, Compton, Decay, Delta ray, Elastic hadron, Inelastic hadron, Pair production, Photoelectric, Capture + dE/dx & MS
  - Tabulate all x-secs (100 bins -> 90MB)
  - Generate (10-50) final states (300kB per final state & element)
- It will not be good Geant4, but but it could be the seed of a fast simulation option
- Independent from the MonteCarlo that actually generates the tables



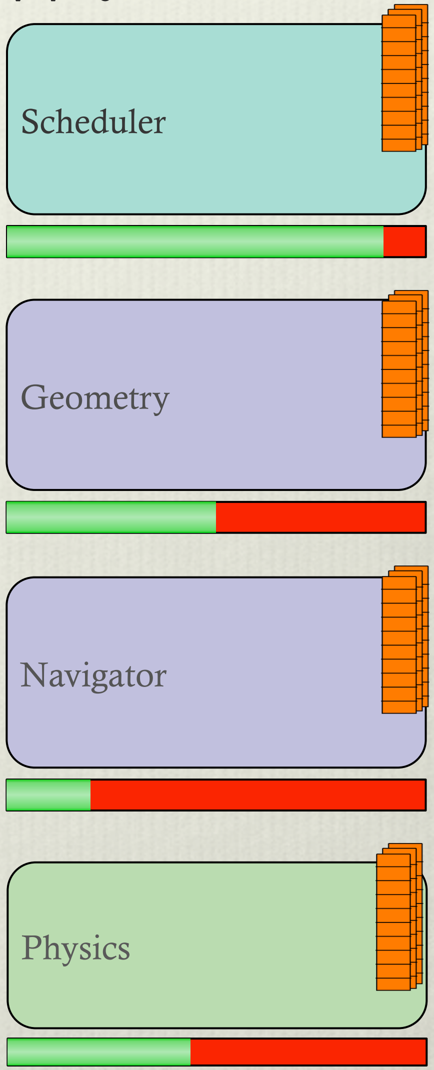


Testing - benchmarking

- Same Physics code must work on CPU and GPU
  - Use C++ template techniques to vectorize platform independent code.
- Standard benchmark Geant4 - Vector Prototype
  - Will have both tabulated physics and vectorized physics ported to Geant4
  - Then can test equivalent physics for each geometry in both framework for both speed and validity.
- Simple “physics” benchmark for Vector Prototype
  - We decided to use something like geant4_vmc/examples/E03 because is a simple calorimeter
  - The idea is to replace the had part with the prototype x-sec

Where are we now?

- Scheduler
  - The new version, hopefully improved of the scheduler has been committed and we are testing it
- Geometry
  - The proof or principle that we can achieve large speedups (3-5+) is there, however a lot of work lays ahead
- Navigator
  - "Percolating" vectors through the navigator is challenging. We have a simplified navigator that achieves that, but more work is needed here
- Physics
  - Can generate x-secs and final states and sample them; starting work on vectorized physics.



Summary

- HEP needs all the cycles it can obtain, nowadays this means using parallelism and SIMD
- Simulation is the ideal primary target for investigation for its relative experiment independence and its importance in the use of computing resources
- The Geant Vector project aims at demonstrating substantial speedup (3-5+) on modern architectures
- The work is done in close collaboration with the stakeholders and with Geant4

2014 Milestones

Milestone – April 11th

- Setup: simple benchmark: ~Ex03 only boxes
- G4 with 'tabulated' physics
- Connect tabulated physics with Vec prot.
- Port Brems to Vector prot, and use also in G4 with tabulated
- Develop USolid and UGeom to be able to run Ex03 in Vector prototype
- Robust scheduler

Field Propagation

- Extend Vector I/F for Field Propagation
  - Important for realistic CPU
  - Depends on other objectives, resources
- Decision point: February 14th

ASAP after April

- Move to Geant4 10.0
- Nightly build system

- Both are
  - Recommended or desirable for April
  - Necessary for July

Milestone 2 – end July

- Magnetic Field (may be earlier)
- Intermediate Detector: 3-5 solids all Vector
- CMS(v 2008): 10 solids - Top 5 vector
- Vector Compton process
  - including first pass of abstraction
- Testing all combos (3 geom, VP/VPT/G4T/G4TV/G4V/G4)
  - Check VP=G4TV, VPT=G4T, G4V=G4

Glossary

- VP= Vec Prototype with Max Vector Procs
- VPT= Vec Prototype with Tab Procs only
- G4T= G4 with Tabulated Procs ‘only’
- G4TV= G4 w/ max Vec Procs, rest Tab procs
- G4V= G4 replacing only Vec Procs
- G4 = Original Geant4

Vector Physics

- Target: create first version of generic code for Vector and GPU
  - similar to approach of Sandro/Johannes
  - separated from G4

GPU

- Get in sync between GPU and Vector
- Principle shadow developments
  - UGeom
  - Tabulated Physics
- Navigation - 'Lock-step' inquiries to Solid Type (November ?)

MIC

- Expect it to work efficiently if GPU runs well enough
- Seek person (Laurent?)
  - to test April prototype, check efficiency
  - follow development.

November

- Complete EM Physics (for a Phys list)
  - As close as possible to Std EM Physics
  - One process (e.g. MSc) with 2 models in Energy
- Full set of Primitive Shapes
  - Composites (importance in CMS?)
- Voxelisation?
- Results for MIC

Backup Slides

Team

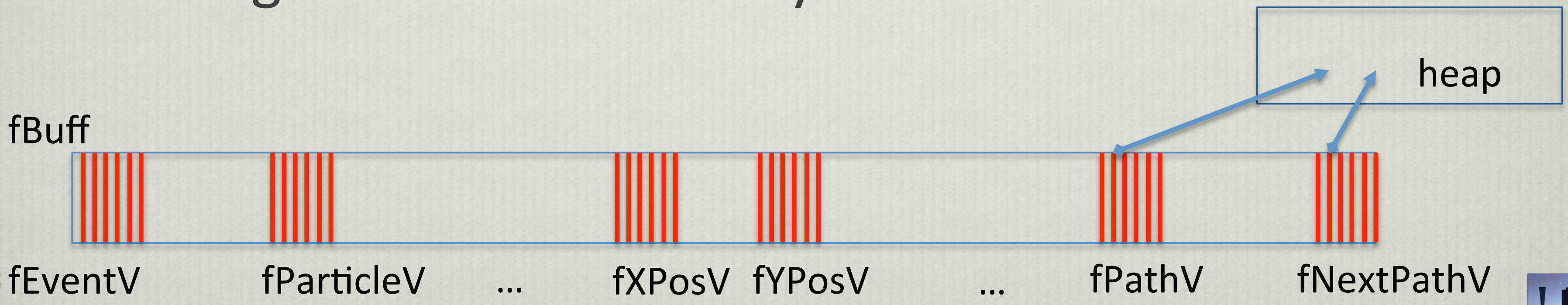
- Andrei (30%), Fed(50%), John(40%), Johannes(100%), Mihaly(100%), Sandro(100%), Georgios(50%*0.5), Tatiana(25%+), doctoral student (100% >March) = 5.5 FTE
- Philippe(30%), Soon(50%), Guilherme(100%), Physics-List-X(20%) = 2.0 FTE
- Marilena(?5%), Raman(?100% >June)
- Laurent (?) (~10%)

GeantTrack

- Track identifiers: event, slot (memory management), track ID, PDG, code
- Particle identifiers: PDG, GeantV code, charge, mass, species
- Kinematics: position, direction, momentum, energy
- Status: status, N steps, N null steps, boundary flag, pending flag
- Geometry/physics context: process, proposed step, current step, distance to boundary, safety, current path, next path
- sizeof(GeantTrack) = 192 bytes + 2*sizeof(TGeoBranchArray) = 192+2*48+depth*4+16 = **344 bytes** in average
- Can this be reduced? Size influences memory requirements AND CPU overhead for reshuffling operations in vector mode.

Track vectorizable container

- Track data format not used directly by the transport – only used to import tracks from generators/ processes
  - Track data imported into GeantTrack_v
- SOA matching GeantTrack using internal memory management for vector performance
  - Single resizable memory block

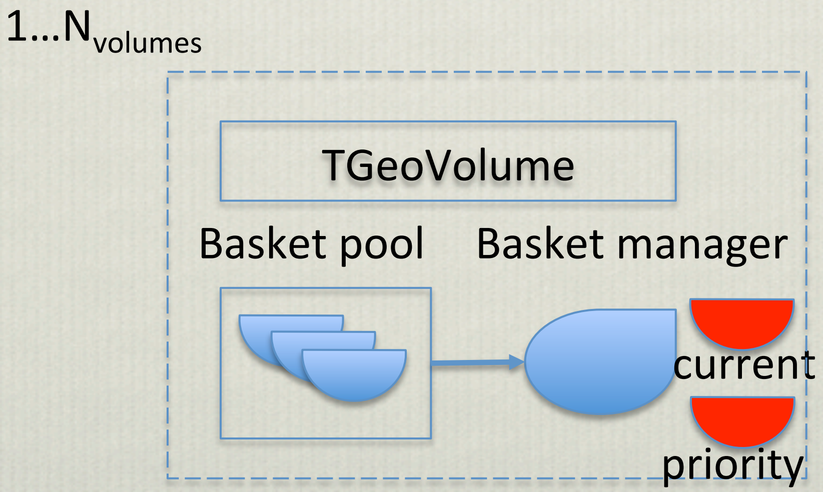


GeantTrack_v

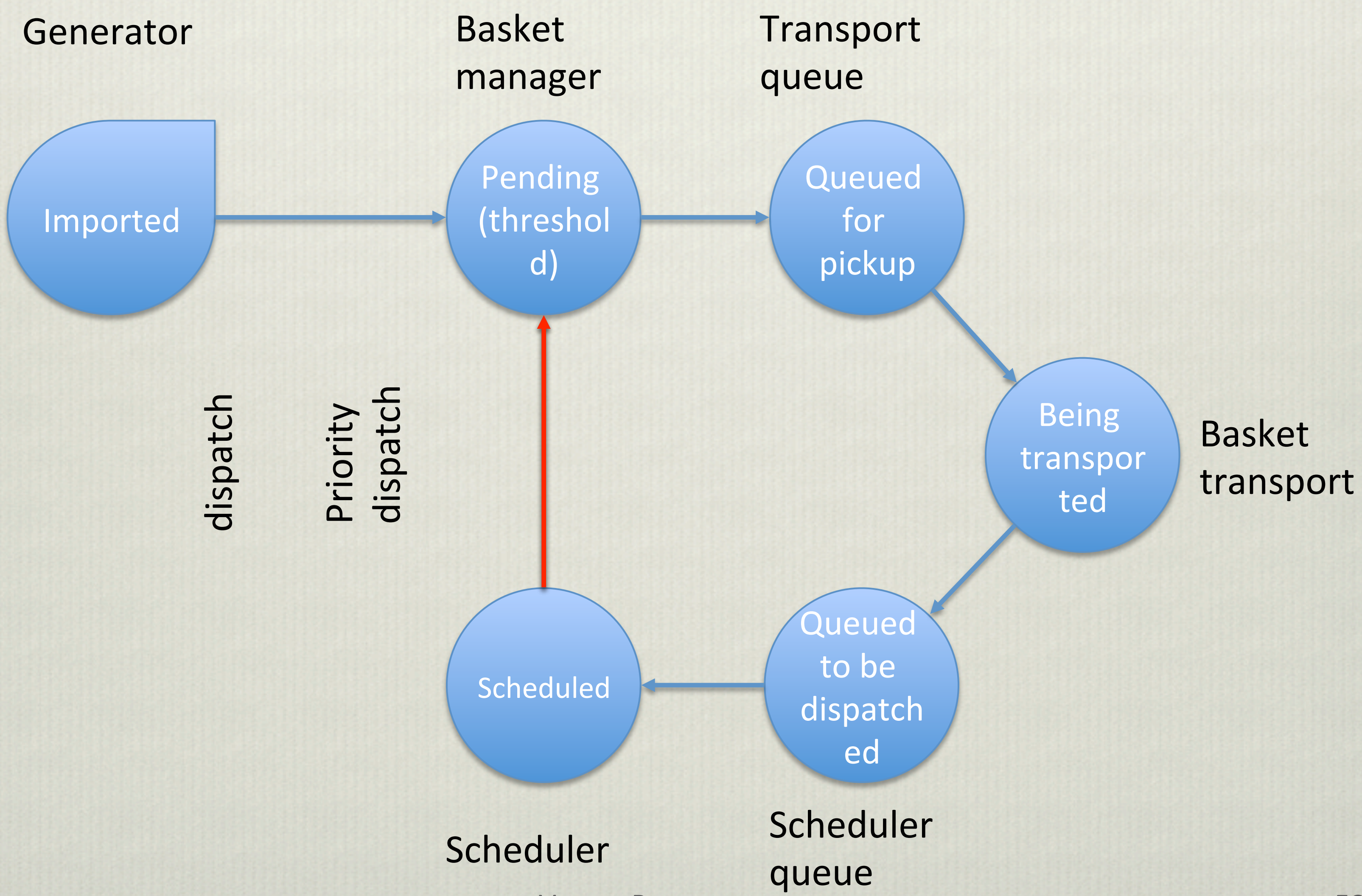
- Buffer management: allocate, copy, resize
- Import tracks from GeantTrack or GeantTrack_v
  - And track removal
- Management of holes (i.e. tracks that finished transport in the current propagation cycle)
  - Vector not efficient-> compact tracks
    - Hole finding algorithm based on TBits + memory copy overhead
- Sorting by track status, needed to vectorize different propagation stages
  - E.g. transport of neutral tracks

Basket managers

- One basket manager per volume
  - Receiving tracks entering the volume from generator or scheduler
  - Accessed by scheduler only
- Pool of empty baskets, one current basket + one basket for prioritized tracks
- Lock-free access for unique scheduler (only one thread can add tracks)
- Transportability threshold per manager
  - If threshold reached when adding tracks, the current basket is pushed in the work queue and replaced from the pool. Tracks added with the priority flag go to the priority basket which gets pushed to the priority side of the queue
  - Threshold(vol) = Ntracks_in_flight(vol)/2N_threads rounded to %4 (min 4, max 256)



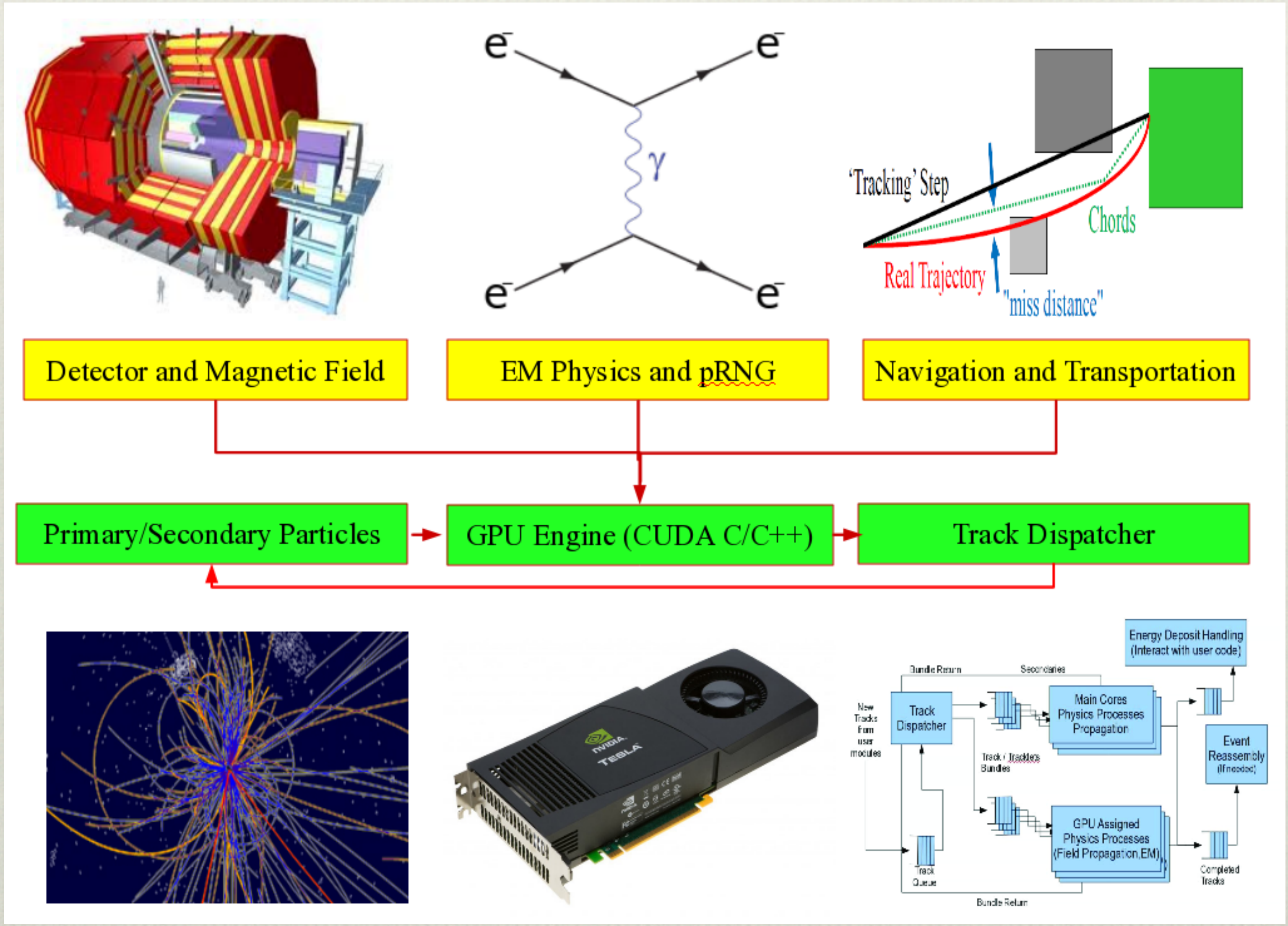
Track stages



Connection to physics & geometry

- Currently trivial approach to physics, have to interface to the new physics code
  - Process selection based on total x-sec
  - Redo process interface for actions (along and post-step)
- Connect to vectorized navigator
  - Even limited to simple setups, we need to understand gains and overheads + tuning
- Connect scheduler to GPU transport
  - Using a manager thread to take and transport baskets from the main CPU queue
  - We have to understand if there are extra requirements
  - Can be done for both geometry and physics baskets

Overview of key components



e-
e-
γ
e-
e-
'Tracking' Step
Chords
Real Trajectory
"miss distance"
Detector and Magnetic Field
EM Physics and pRNG
Navigation and Transportation
Primary/Secondary Particles
GPU Engine (CUDA C/C++)
Track Dispatcher
Energy Deposit Handling (Interact with user code)
Bundle Return
Secondaries
New Tracks from user modules
Track Dispatcher
Main Cores Physics Processes Propagation
Track / Tracklets Bundles
Event Reassembly (If needed)
GPU Assigned Physics Processes (Field Propagation,EM)
Track Queue
Completed Tracks
Bundle Return
NVIDIA TESLA

Reminder, Motivation

Explore possibilities to recast particle simulation so that it takes advantage from all performance dimensions/technologies

Dimension I ("sharing data") : multithreading/multicore

In HEP, mainly to reduce memory footprint

⇨ Geant4 Release 10!

Dimension II ("throughput increase") : incore micro-parallelism or vectorization

⇨ Currently often not exploited because requires "parallel data" to work on

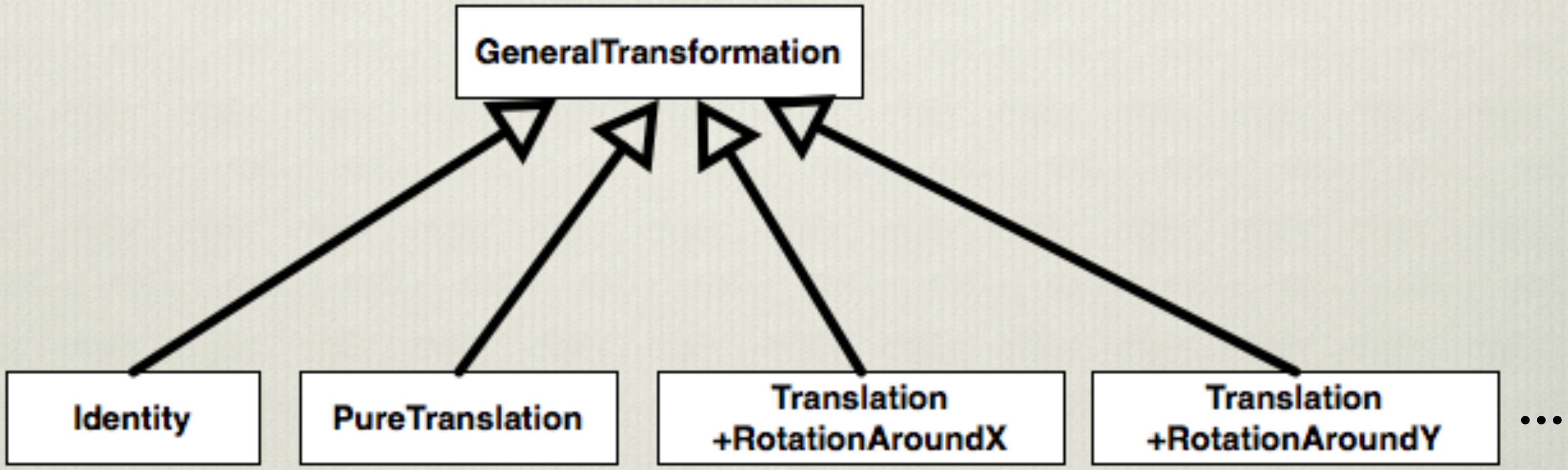
Research projects (GPU prototype and Geant-Vector Prototype) have started targeting beyond dimension I:

parallel data ("baskets") = particles from different events grouped by logical volumes

specializing coordinate transformations

- How many of those floating point operations are actually relevant?

- Let' s have a look at what important transformations are actually used:
  statistics generated from ATLAS,CMS,ALICE, LHCB geometries (ftp://root.cern.ch/root/geometries.tar.gz)



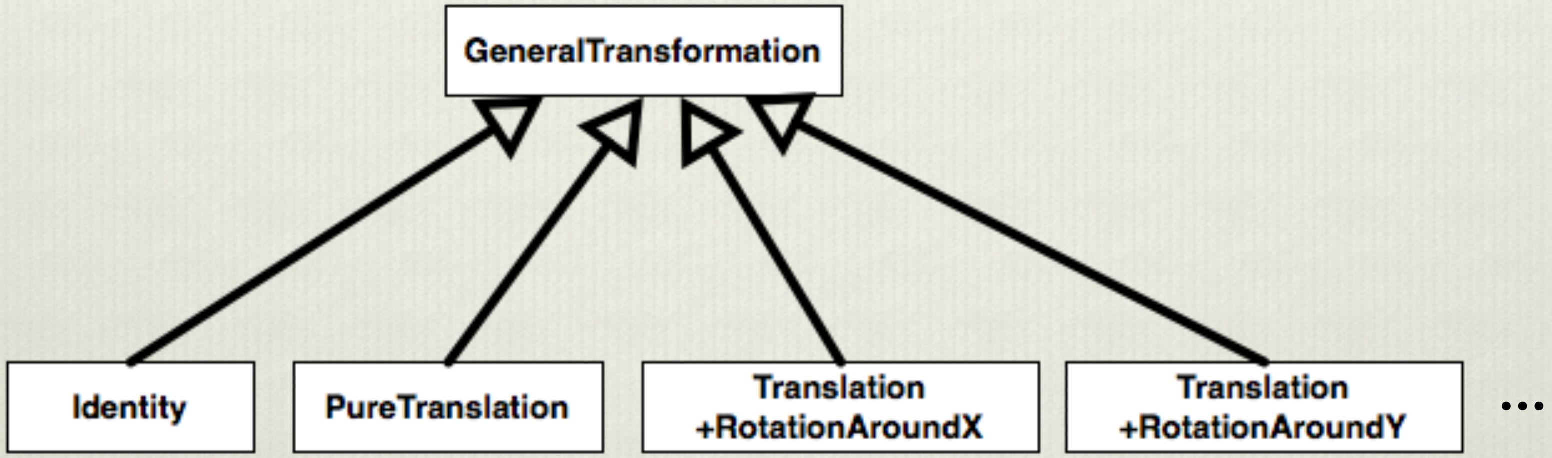
17% **28%** **~50% of all transformations are a translation + <u>very simple</u> rotation**

- **looking still closer, one realizes: ~85% of all matrices would actually require <=3 multiplications, <=3 additions**

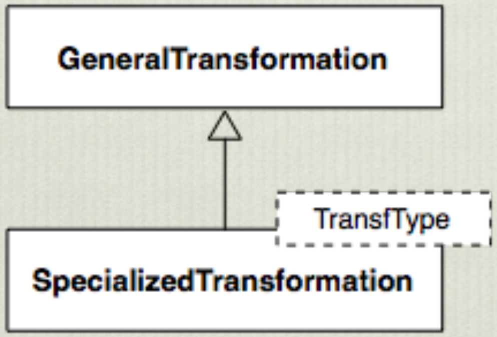
- for vectors of particles this adds up to a considerable saving in floating point ops

Specializing Coordinate Transformations

* We should have specialized coordinate transformations !



* As before we can generate them using a template class



* A **factory** takes care to produce right instance

```
GeneralTransformation *t = GeoManager::CreateTransformation( ... );
```

Statistics Tubes

counting atlas, cms, alice, lhcb, babar

(taken from root files; probably a bit out of date)

* fulltube rmin = 0; 3826374
* hollowtube rmin >0; 2692417
* phitube rmin=0; 17475959
* phitube rmin>0; 1405601

statistics matrices

counting atlas, cms, alice, lhcb, babar

- Identity transformation: 8.6 million; percent of total: 17%
- only translation: 14.1 million; percent total: 28%
- only rotation: 0.8 million; percent total: 1.6%
- combi matrices: 27 million; percent total: 54%
- 20 million rotation matrices have 6 zeros !!
- 4.3 million rotation matrices have 4 zeros !!
- total number: 5.05 e7 matrices

Acknowledgements

Thanks to:

- Geant-V / GPU team
- Laurent.Duhem@Intel for discussions leading to the present ideas
- Johannes De Fine Licht (implementing a lot of the template ideas)

First prototype available at:

https://github.com/sawenzel/VecGeom.git

Summary/Outlook

Summary

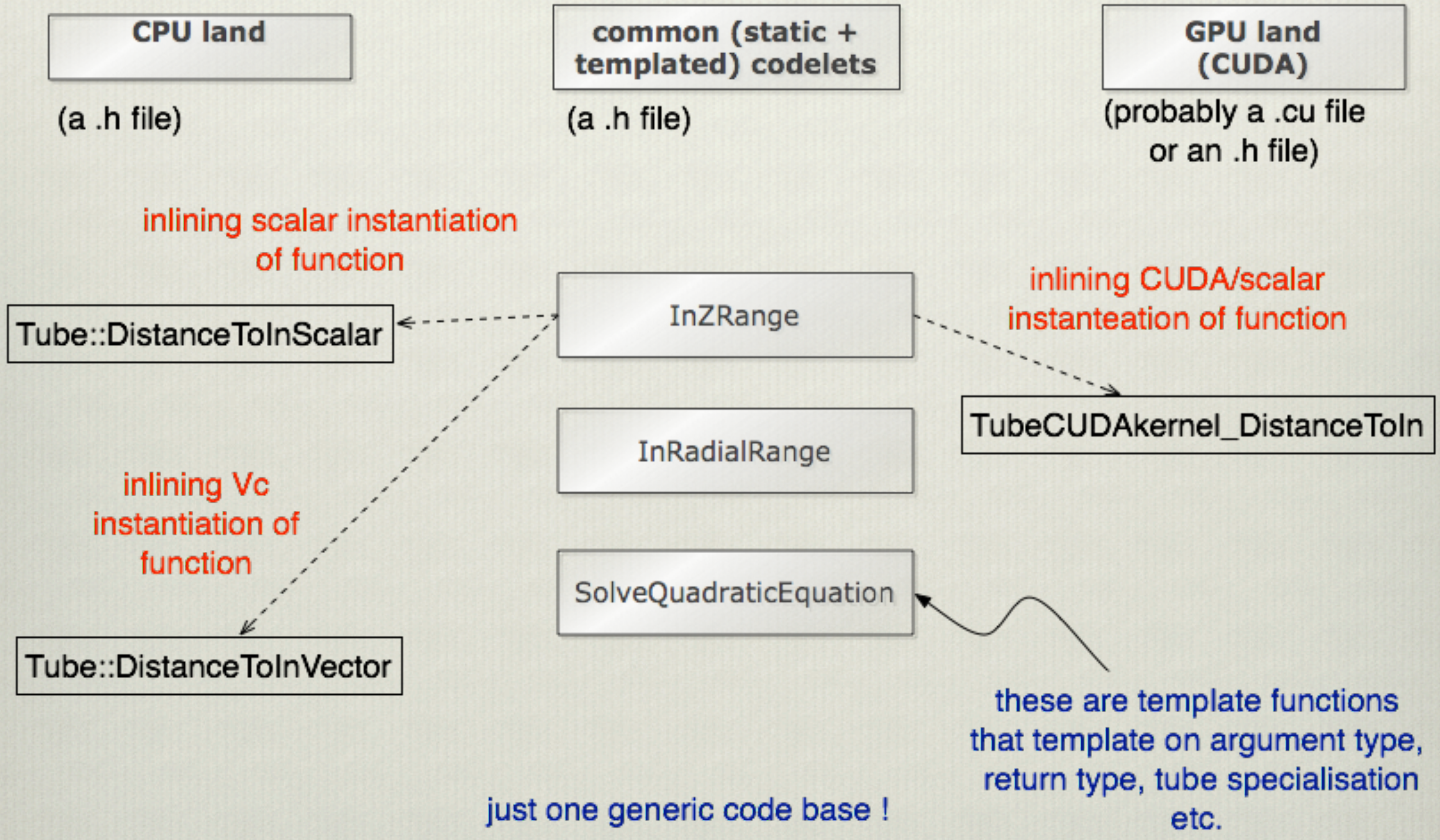
- status and challenges of vectorized geometry
- discussed motivation for using template techniques
- concentrated here on benefits of template specialization for performance
  - generation of specialized classes without code duplication
  - reduction of static branches leading to better compiler optimization and more efficient vectorization
  - avoiding unnecessary floating point operations
- overall 30% gain in our standard (simple) benchmark

Outlook

- code generality between scalar and vector code
- sharing code between CPU and GPU $\Rightarrow$ upcoming talk by Johannes De Fine Licht
- April milestone for Geant-V / GPU prototype

Backup slides

Towards a common CPU / CUDA code base


CPU land
(a .h file)
common (static + templated) codelets
(a .h file)
GPU land (CUDA)
(probably a .cu file or an .h file)
inlining scalar instantiation of function
Tube::DistanceToInScalar
InZRange
inlining CUDA/scalar instanteation of function
TubeCUDAkernel_DistanceToIn
InRadialRange
inlining Vc instantiation of function
SolveQuadraticEquation
Tube::DistanceToInVector
these are template functions that template on argument type, return type, tube specialisation etc.
just one generic code base !

Notes on benchmark conditions

- System: Ivybridge iCore7 (4 core, not hyperthreaded (can read out 8hardware performance counters))
- Compiler: gcc4.7.2 (compile flags -O2 -unroll-loops -ffast-math -mavx)
- OS: slc6
- Vc version: 0.73
- benchmarks usually run on empty system with cpu pinning (taskset -c)

- benchmarks use preallocated pool of testdata, in which we take out N particles for processing. Repeat this P times. For repetitions distinguish between random access of N particles (higher cache impact) or sequential access in datapool (as shown here)
- benchmarks shown use NxP=const to time an overall similar amount of work

Physics Specific work

- Many issues opened in Jira about physics interactions
- "SFT-private" version of G4 created
- Opportunity to verify x-sections as extracted against x-sections as sampled and data
  - An immediate issue with ionisation x-section
- Some specific problems in the sampling code
  - The activation of the capture mechanism causes the code to crash
  - The sampling of the multiple scattering angle is problematic, as it clearly gives wrong results. It would be important to see whether this can be done by the SampDisOne routine that is already sampling the other interaction

Targets

- By the end of the year we will "glue" the different pieces together
  - And hopefully demonstrate the speedup potential of MT, locality and SIMD
- Measure the evolution of the memory footprint and the performance of the code at least in terms of hardware counters
- Absolute performance measurements will be harder
  - Difficult compare apples to apples
  - Probably we need to develop dedicated benchmarks
- Compare physics performance with full MC's
- We are working closely with Geant4 for the physics tables
- Once the prototyping phase over, we will have to sit down with the stakeholders and decide how to proceed from there