Vectorizing and optimizing detector geometry classes

-- benefits and opportunities from template techniques --

R&D!

Sandro Wenzel / CERN-PH-SFT

(for the GPU simulation+ Geant-V prototypes)

concurrency forum, 29.1.2014

building on previous talks in this forum (5.6.13 + 9.10.13)

Outline

- short reminder of what we are doing
- status of effort so far
- 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)

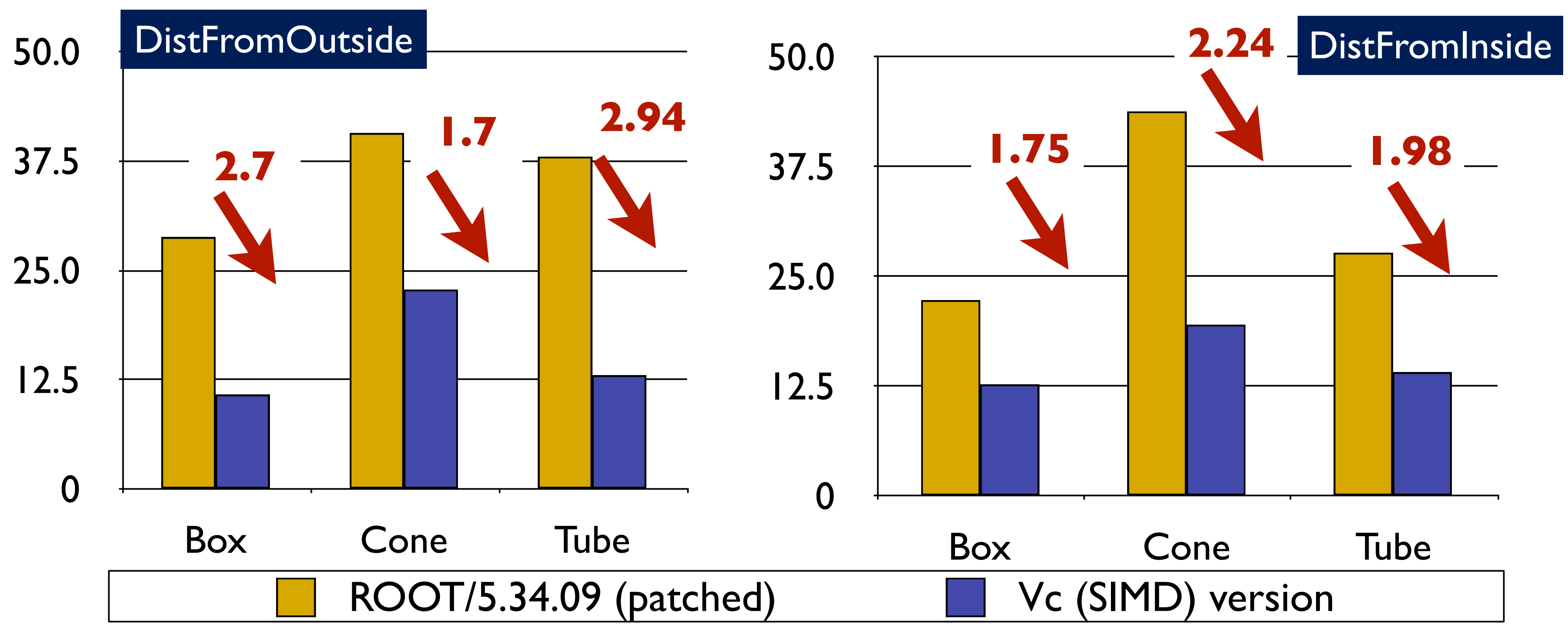
focus on ideas rather than

many performance numbers

Status Reminder (CHEP13)

- activity since spring 2013 focused on studying feasibility of vectorizing (primitive and higher-level) geometry algorithms for the Geant-V and GPU simulation prototypes

- demonstrated for a couple of shapes (box, tube, cone) that this is very possible indeed with good performance gains

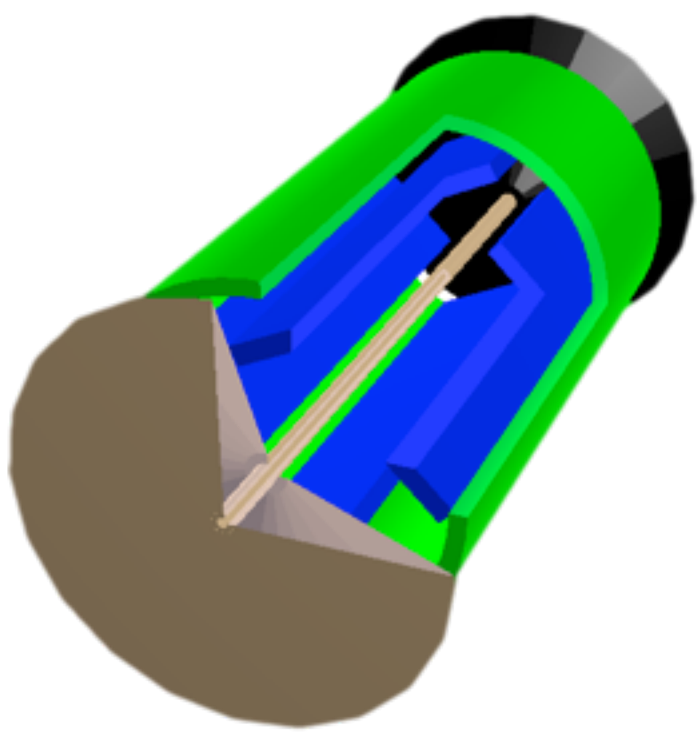


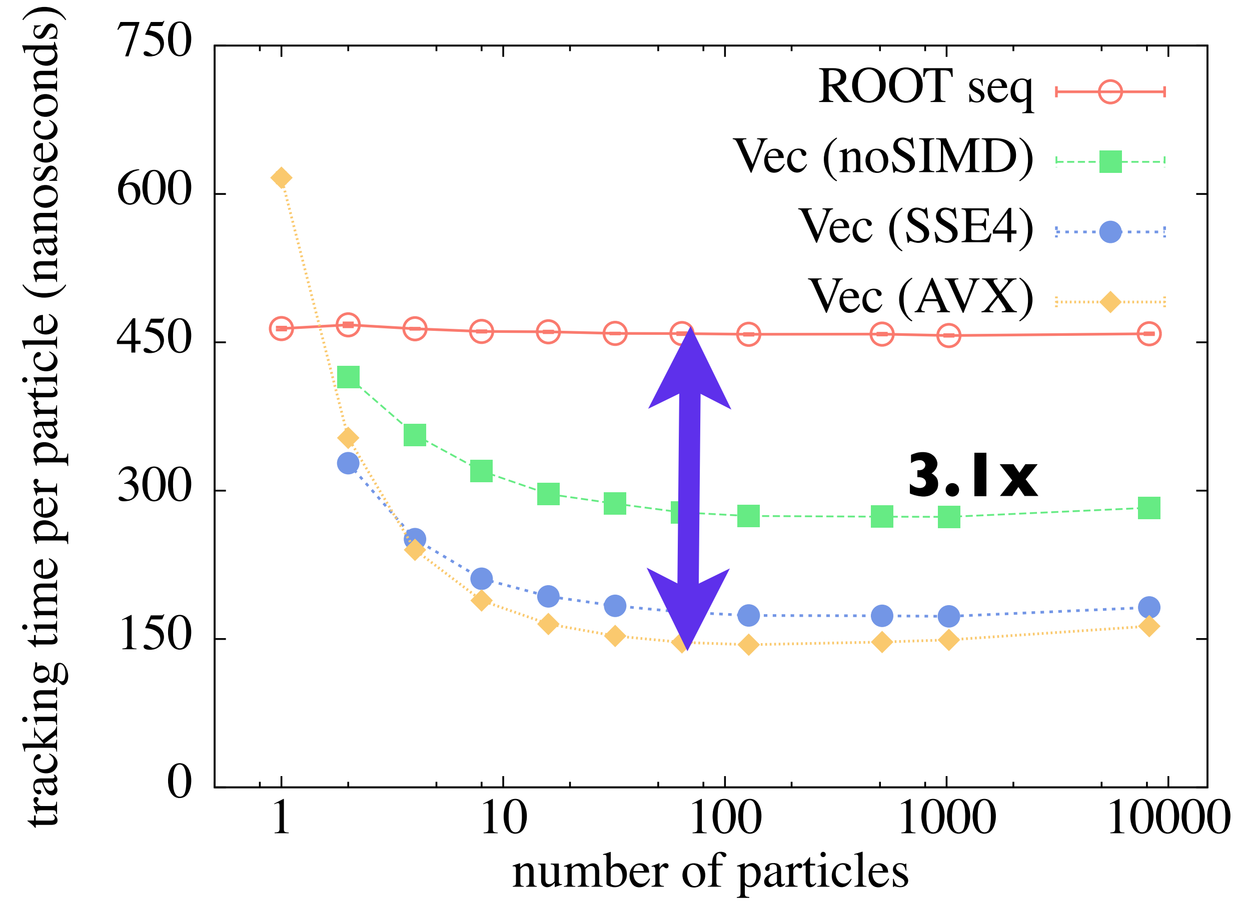
- this came at the cost of totally rewriting the routines to make them vector friendly

- adopted programming model: Vc library, Intel Cilk Plus Array notation

CHEP13 higher-level vector performance benchmark:

- (simplified) navigation of vectors of particles in a simplified detector with daughter shapes
  - CHEP13: **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

- we should also now start a systematic effort to produce a "prototype ready" vectorized geometry library for both the Geant-V and GPU-prototypes
  - provide a library with vector interfaces for important geometry funct.
  - provide a library targeting the CPU + CUDA at the same time
  - achieve best performance
- **main challenges ahead** (from my point of view):
  - 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** (regardless of starting point: ROOT or USolids)
  - **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 remind me ... of 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.

    ⇨ focus of this talk

- 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)

    ⇨ focus of another talk

any alternatives??

Benefit 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

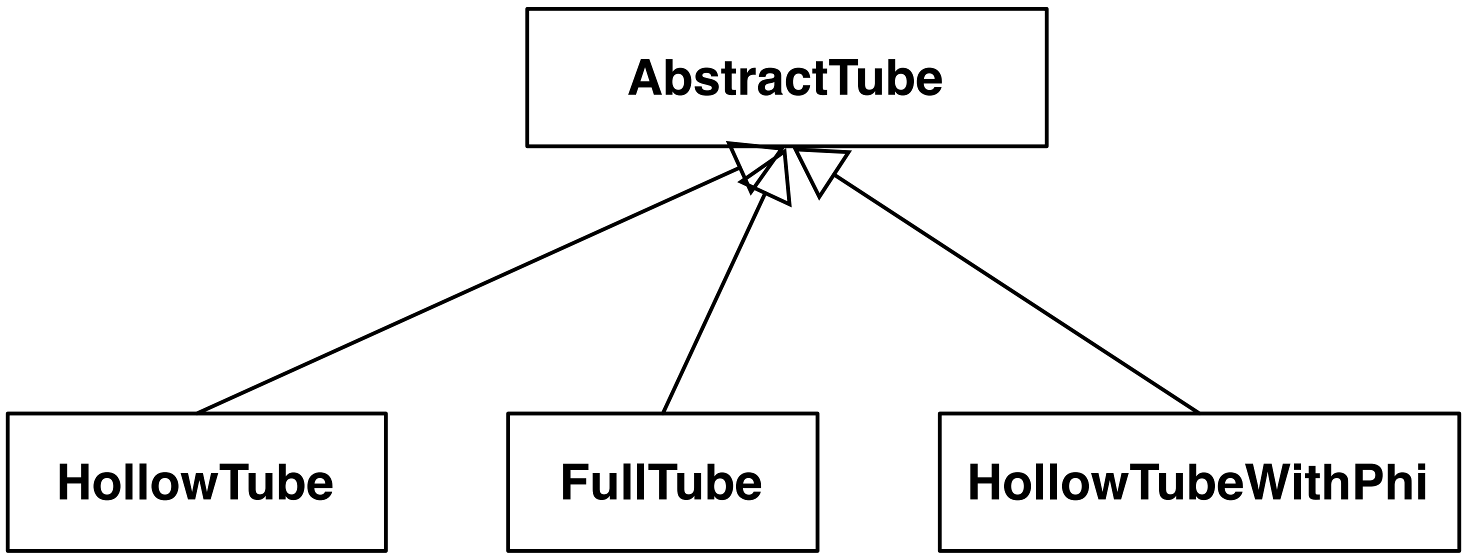
few

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 <u>handwritten separate classes</u>

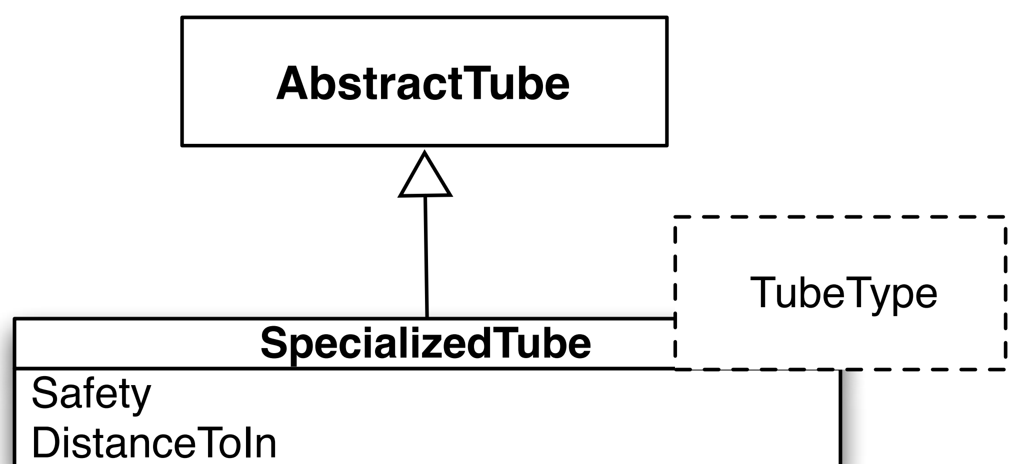



```
AbstractTube *t = new FullTube();
```

performance 

code repetition 

- **alternative idea:** solution with <u>templated classes</u>




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

performance 

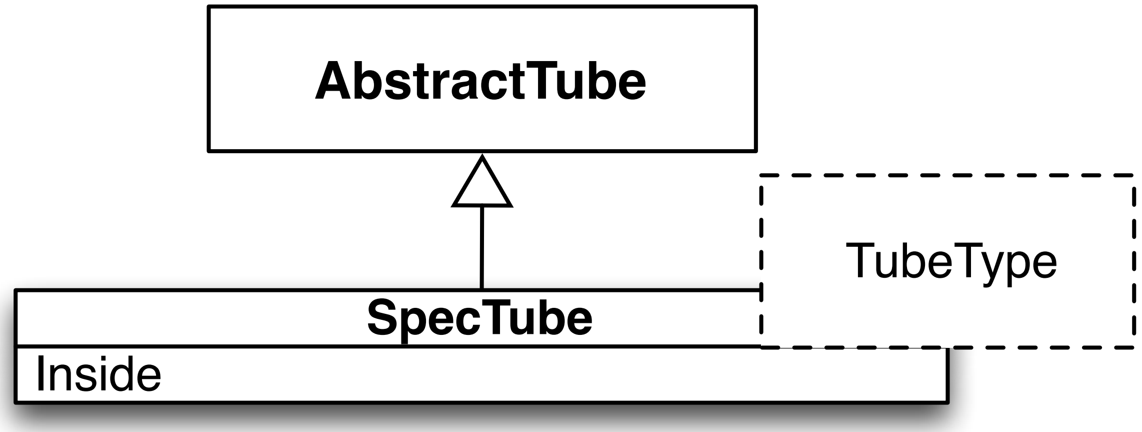
(almost) no code repetition 

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 --

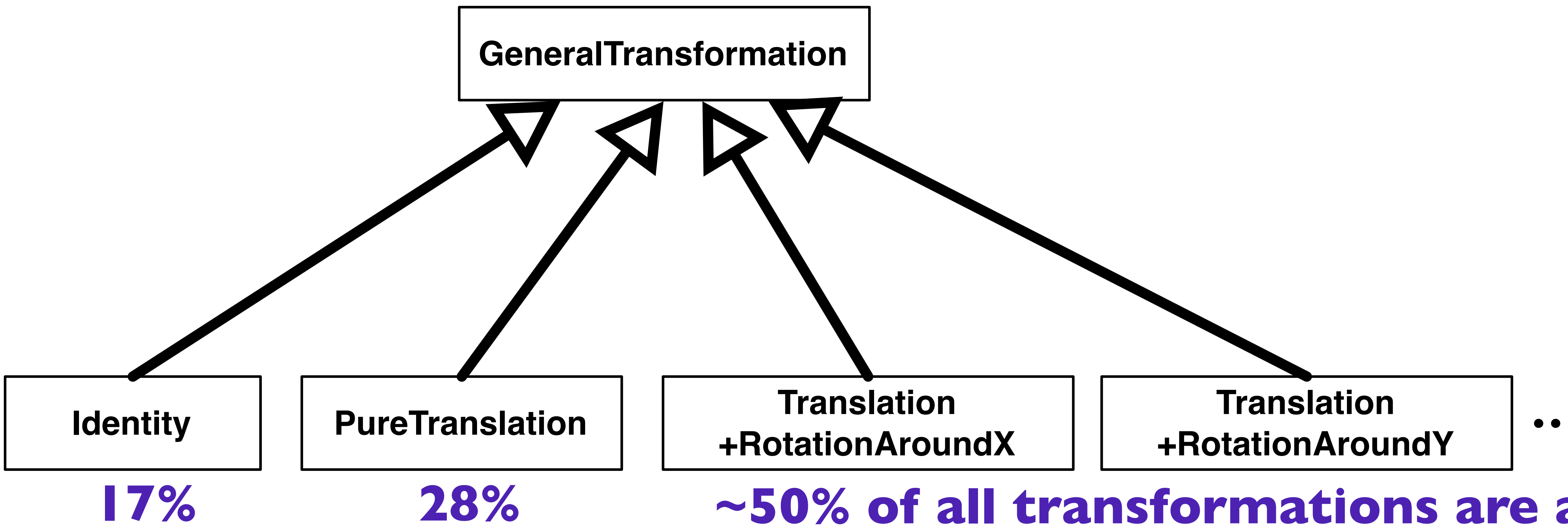
- next to branch reduction; can find many examples where specializing code can be beneficial to save many floating point operations
- example: coordinate transformations between coordinate systems of different shapes
  - known to consume a considerable time (in simple geometries) -- Laurent Duhem@Intel
  - 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

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)

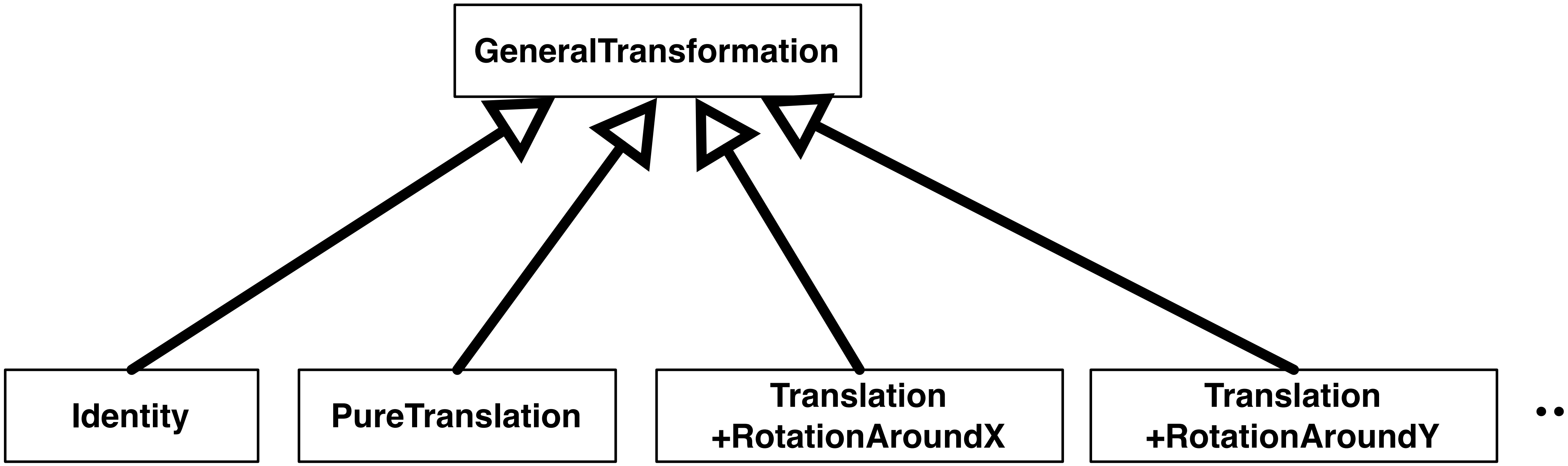


~50% of all transformations are a translation + very simple rotation

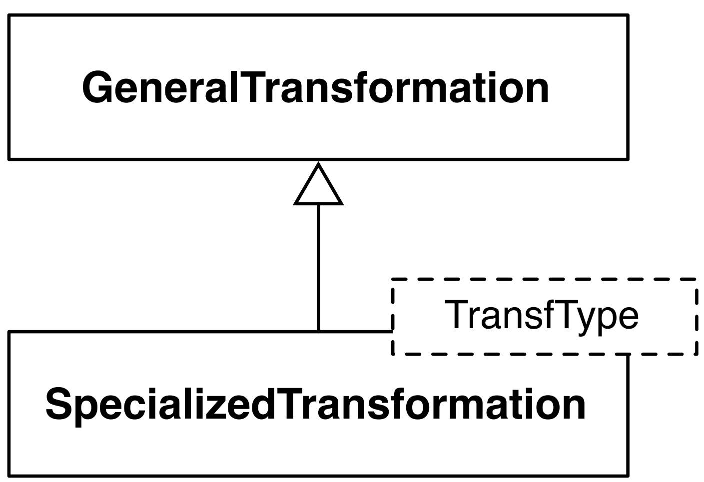
- **looking still closer, one realizes: ~85% of all matrices would actually require <=3 multiplications, <=3 additions**
- for vectors of particles this adds up to a considerable saving in floating point ops

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(...);

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**

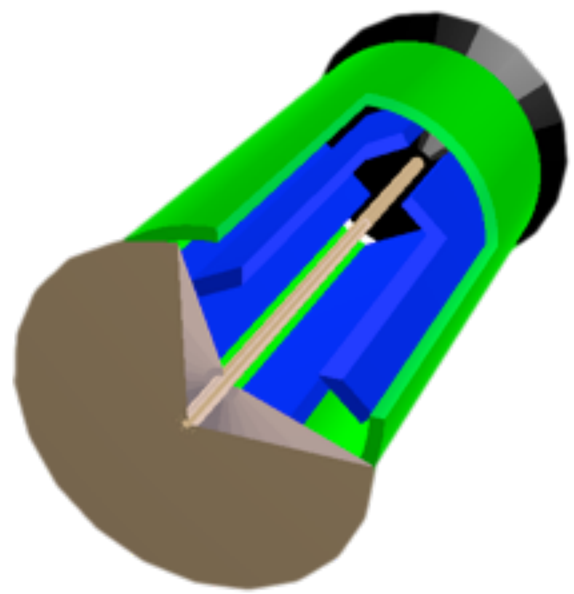
HollowTubeWithPhi	~2.7
HalfHollowTube	~2.6

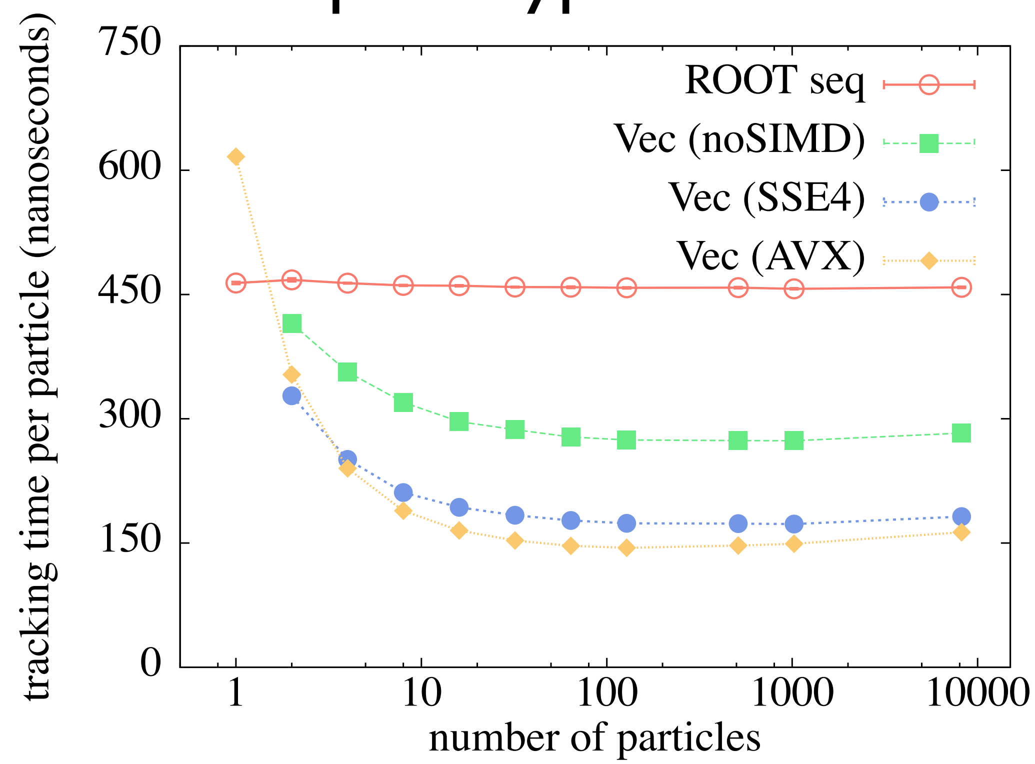
benefit from vectorizing + templating the tube (on AVX)

→ these SIMD speedups match our expectations

CHEP13 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 !!**

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)

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 → upcoming talk by Johannes De Fine Licht
- April milestone for Geant-V / GPU prototype

Acknowledgements

Thanks to:

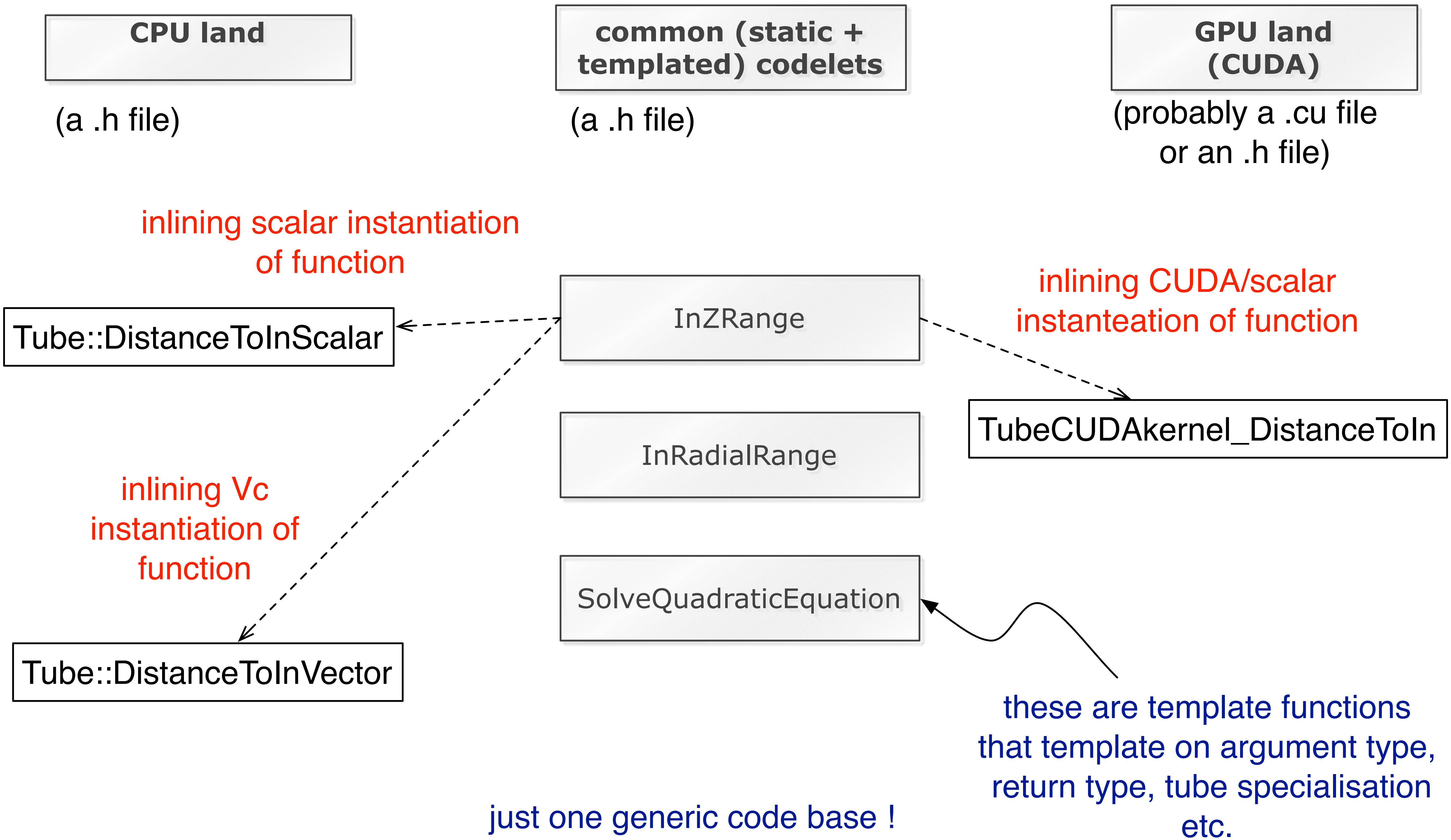
- Geant-V / GPU team
- Laurent.Duhem@Intel for discussions leading to the present ideas
- Johannes De Fine Licht (implementing a lot of the template ideas)

First prototype available at:

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

Backup slides

Towards a common CPU / CUDA 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