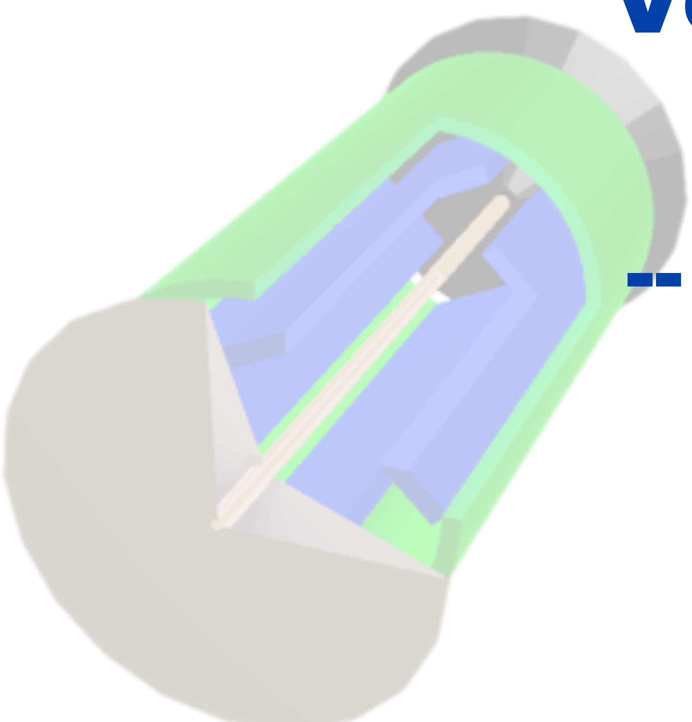
5th International Workshop on Future Challenges in Tracking and Trigger



Vectorizing the geometry library for simulation

-- experience and results from a prototype and future directions --

Sandro Wenzel / CERN-PH-SFT

(for the GPU simulation+ Geant-V prototypes)

CERN

intel EMEA High Performance and Throughput Computing

Fermilab

CERN openlab

Outline

Part I: Introduction

Very short intro to Geant-V

Part II: Prototype phase

A SIMD-vectorized geometry prototype: goals and lessons learned

Part III: VecGeom: current developments

Current developments: A generic high performance geometry library

Introduction and recap of status of many-particle vectorization prototype

with contributions from

Marilena Bandieramonte (University of Catania, Italy)
Georgios Bitzes (CERN Openlab)
Laurent Duhem (Intel)
Raman Sehgal (BARC, India)
Juan Valles (CERN summer student)

The Eight performance dimensions

- The "dimensions of performance"
  - Vectors (SIMD)
  - Instruction Pipelining
  - Instruction Level Parallelism (ILP)
  - Hardware threading
  - Clock frequency
  - Multi-core
  - Multi-socket
  - Multi-node

Micro-parallelism: gain in throughput and in time-to-solution

Gain in memory footprint and time-to-solution but not in throughput

Possibly running different jobs as we do now is the best solution

slide by F. Carminati

The Eight performance dimensions

- The "dimensions of performance"
  - Vectors (SIMD)
  - Instruction Pipelining
  - Instruction Level Parallelism (ILP)
  - Hardware threading
  - Clock frequency
  - Multi-core
  - Multi-socket
  - Multi-node

Micro-parallelism: gain in throughput and in time-to-solution

Gain in memory footprint and time-to-solution but not in throughput

used by Geant4-MT (event parall.)

Possibly running different jobs as we do now is the best solution

slide by F. Carminati

The Eight performance dimensions

- The "dimensions of performance"
  - Vectors (SIMD)
  - Instruction Pipelining
  - Instruction Level Parallelism (ILP)
  - Hardware threading
  - Clock frequency
  - Multi-core
  - Multi-socket
  - Multi-node

Micro-parallelism: gain in throughput and in time-to-solution

targeted by Geant-V (track parall.)

Gain in memory footprint and time-to-solution but not in throughput

used by Geant4-MT (event parall.)

Possibly running different jobs as we do now is the best solution

slide by F. Carminati

Key observation for Geant-V: Classical HEP transport is mostly local

- To make use of SIMD microparallelism we need “data” parallelism: multiple data on which to operate same instructions

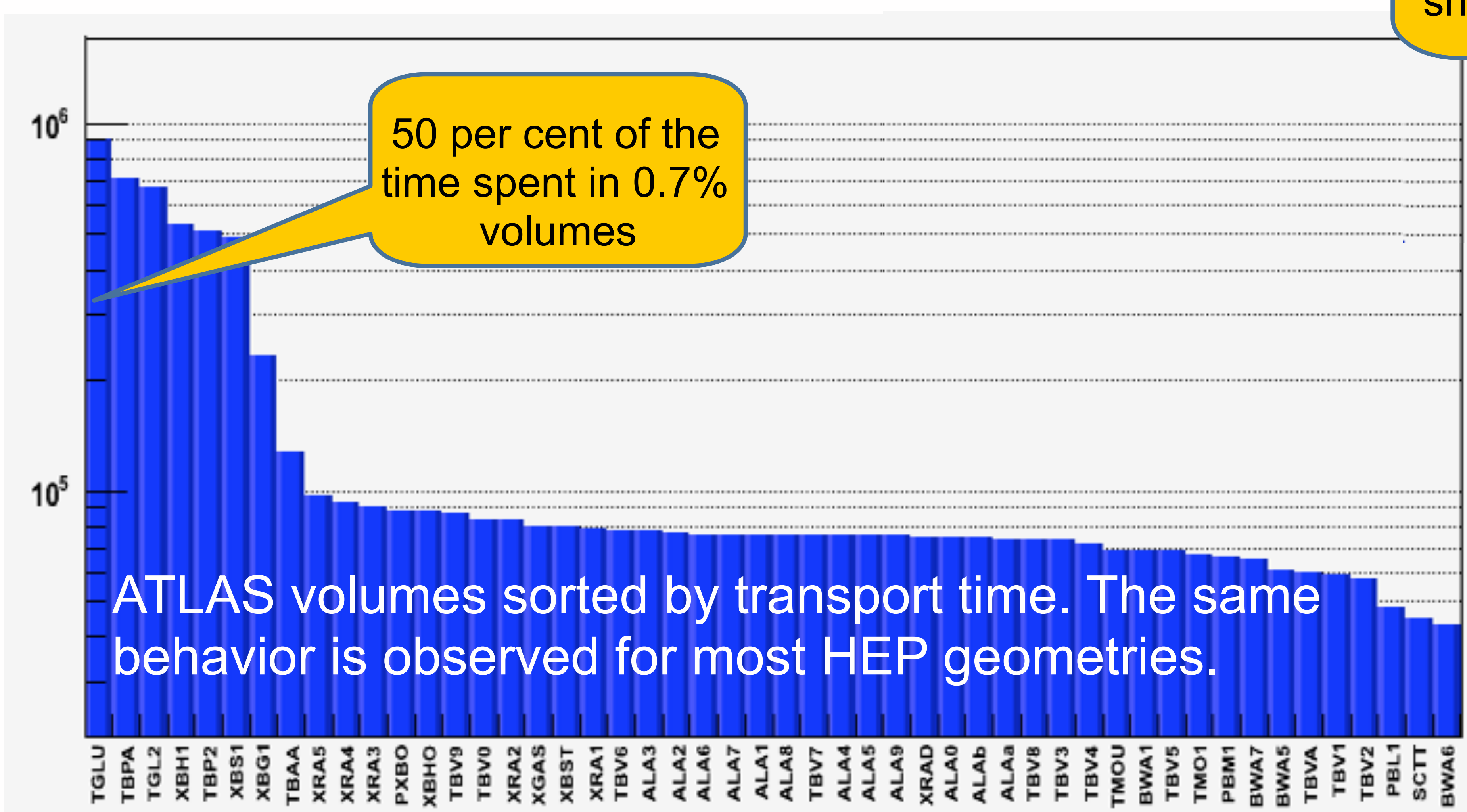
slide by F. Carminati

Key observation for Geant-V: Classical HEP transport is mostly local

- To make use of SIMD microparallelism we need “data” parallelism: multiple data on which to operate same instructions
- benchmarks have shown that in simulation 50 percent of CPU time is spent in small number of logical volumes of detector

- idea: interleave multiple events in simulation and group particles by logical volume **= basket of particles**

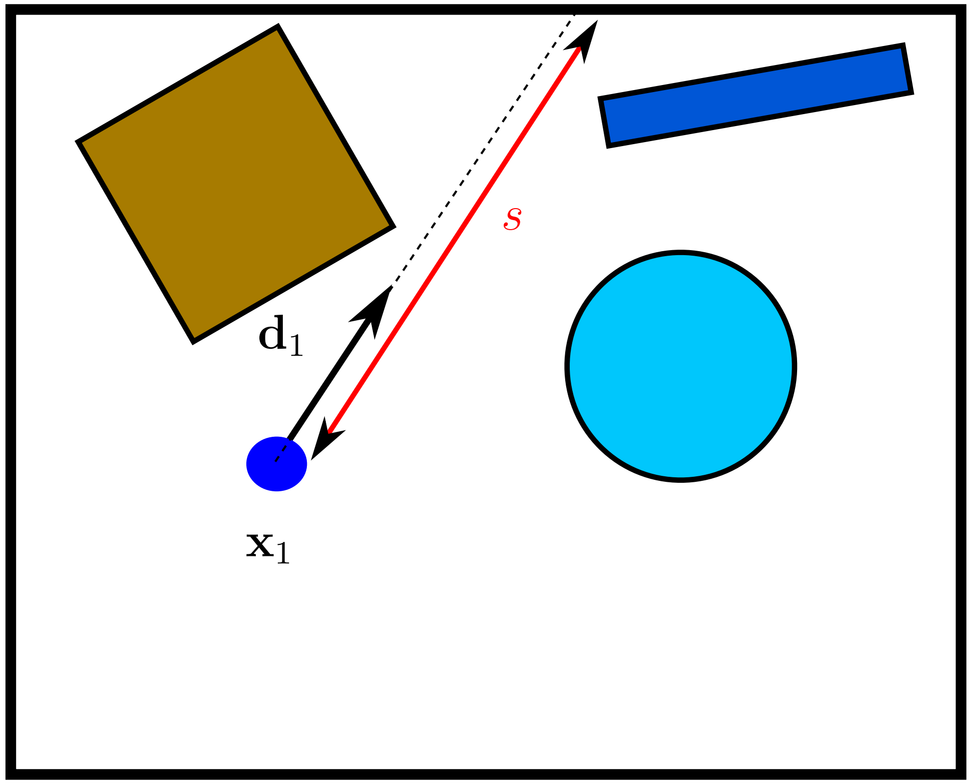
data parallelism in a logical volume; same geometry code; shared physics code



slide by F. Carminati

Vectorizing geometry: The problem statement

typical geometry task in particle tracking: **find next hitting boundary and get distance to it**

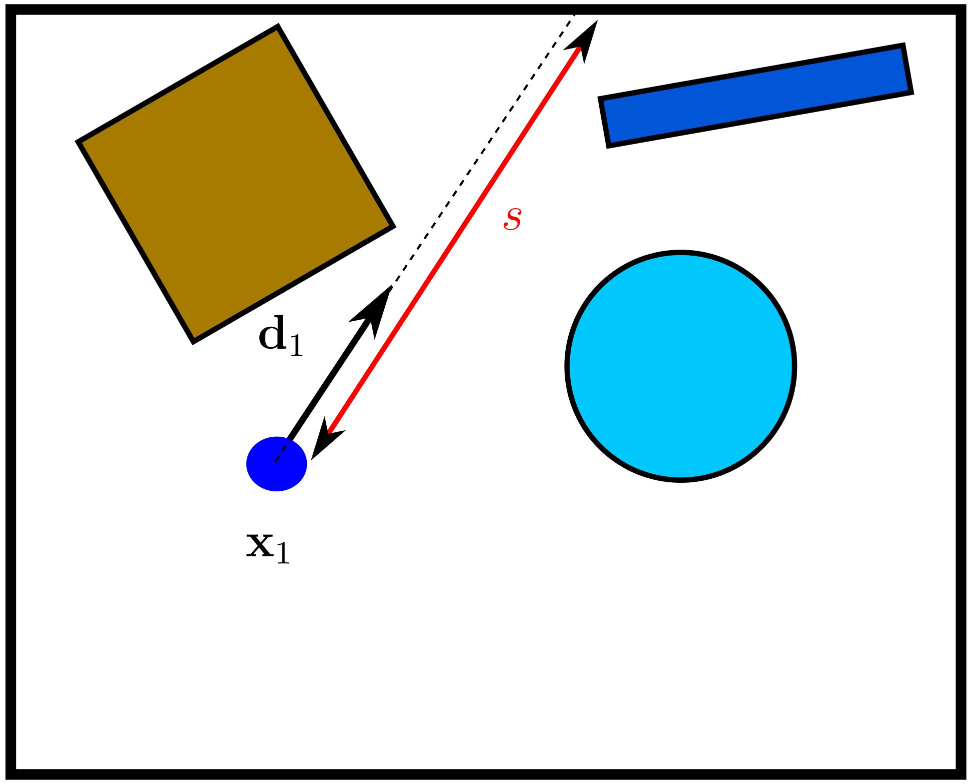


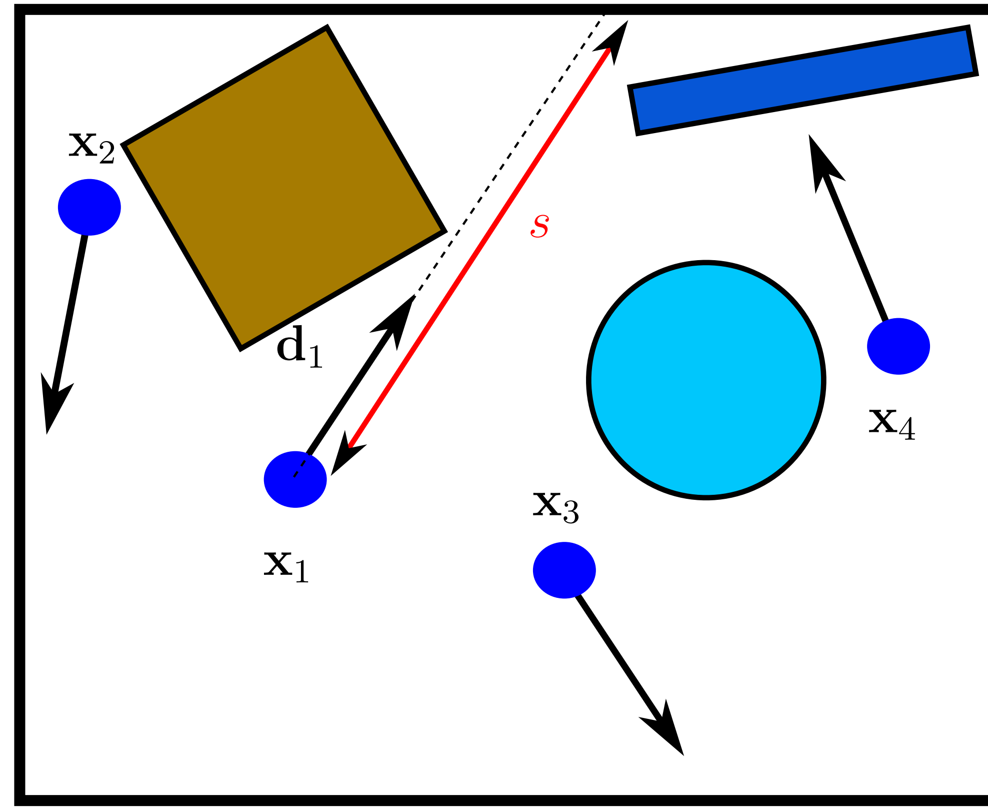
1 particle

functionality provided by existing code (Geant4, ROOT,...)

Vectorizing geometry: The problem statement

typical geometry task in particle tracking: **find next hitting boundary and get distance to it**





1 particle

functionality provided by existing code (Geant4, ROOT,...)

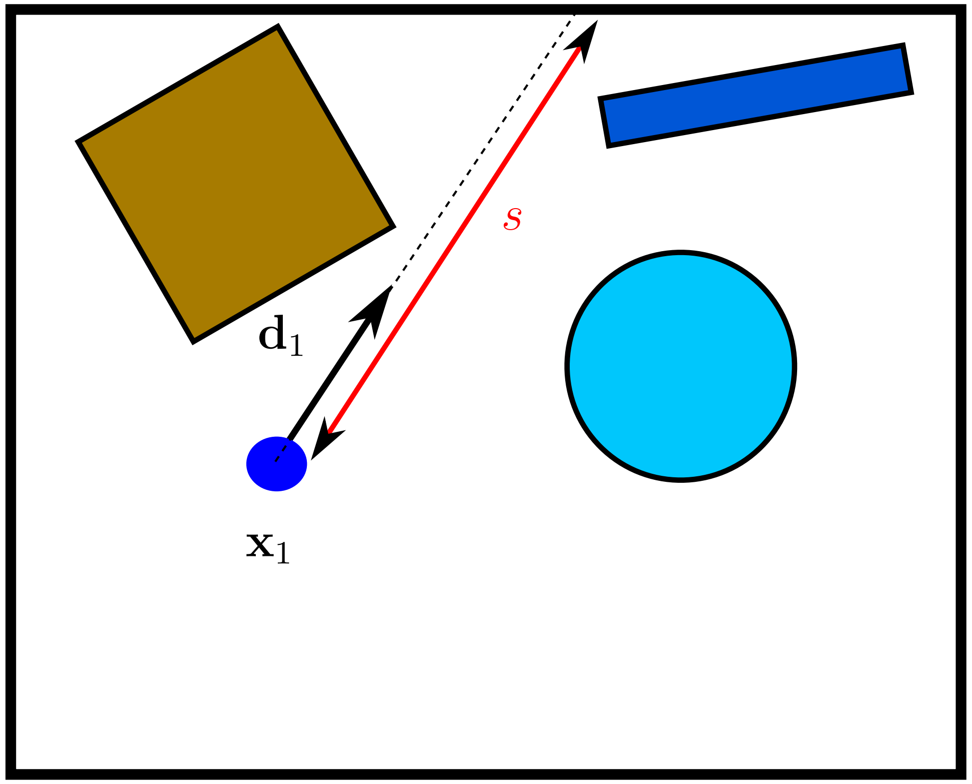
vectors of particles

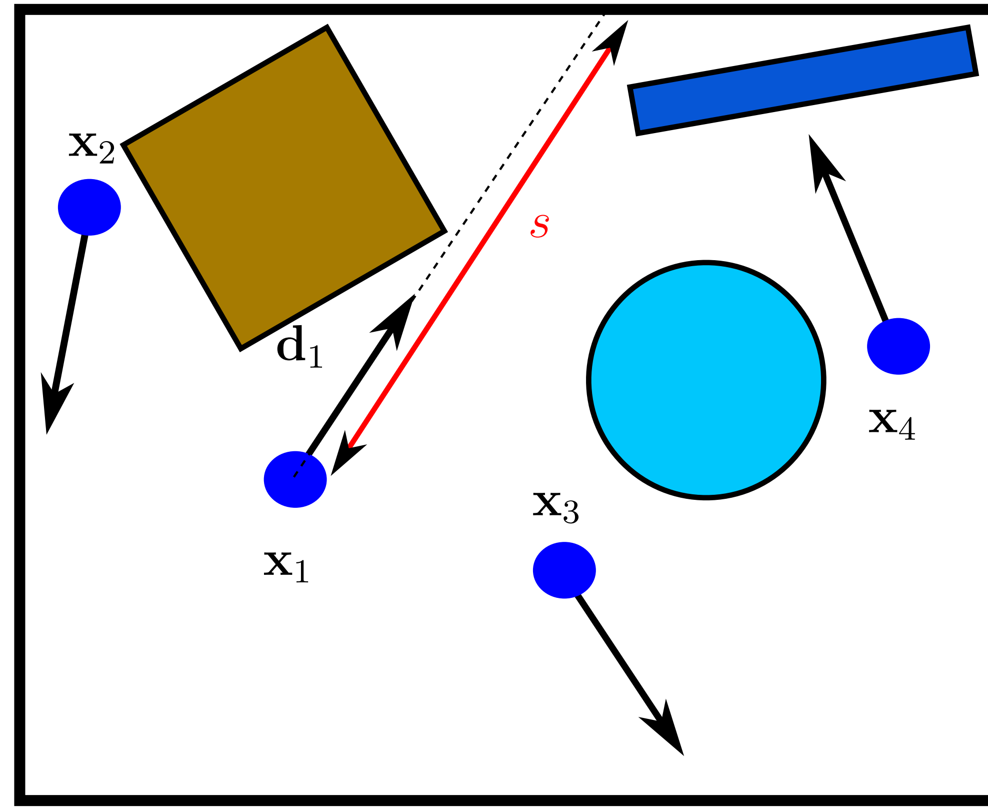
functionality targeted by future simulation approaches

aim for efficient utilization of current and future hardware

Vectorizing geometry: The problem statement

typical geometry task in particle tracking: **find next hitting boundary and get distance to it**





1 particle

functionality provided by existing code (Geant4, ROOT,...)

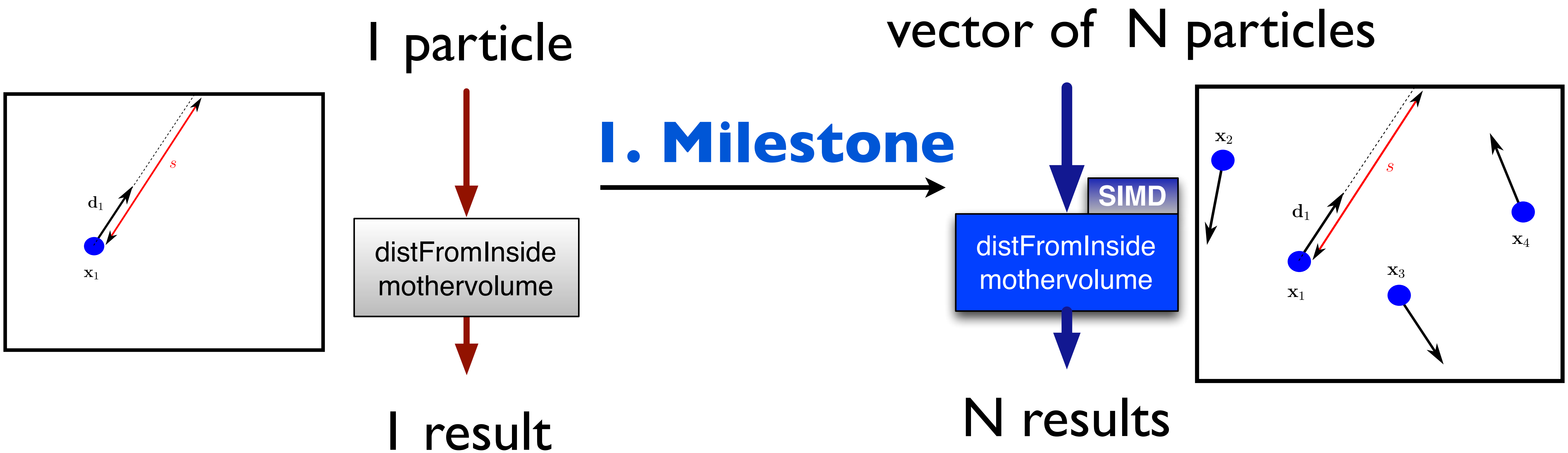
vectors of particles

functionality targeted by future simulation approaches

aim for efficient utilization of current and future hardware

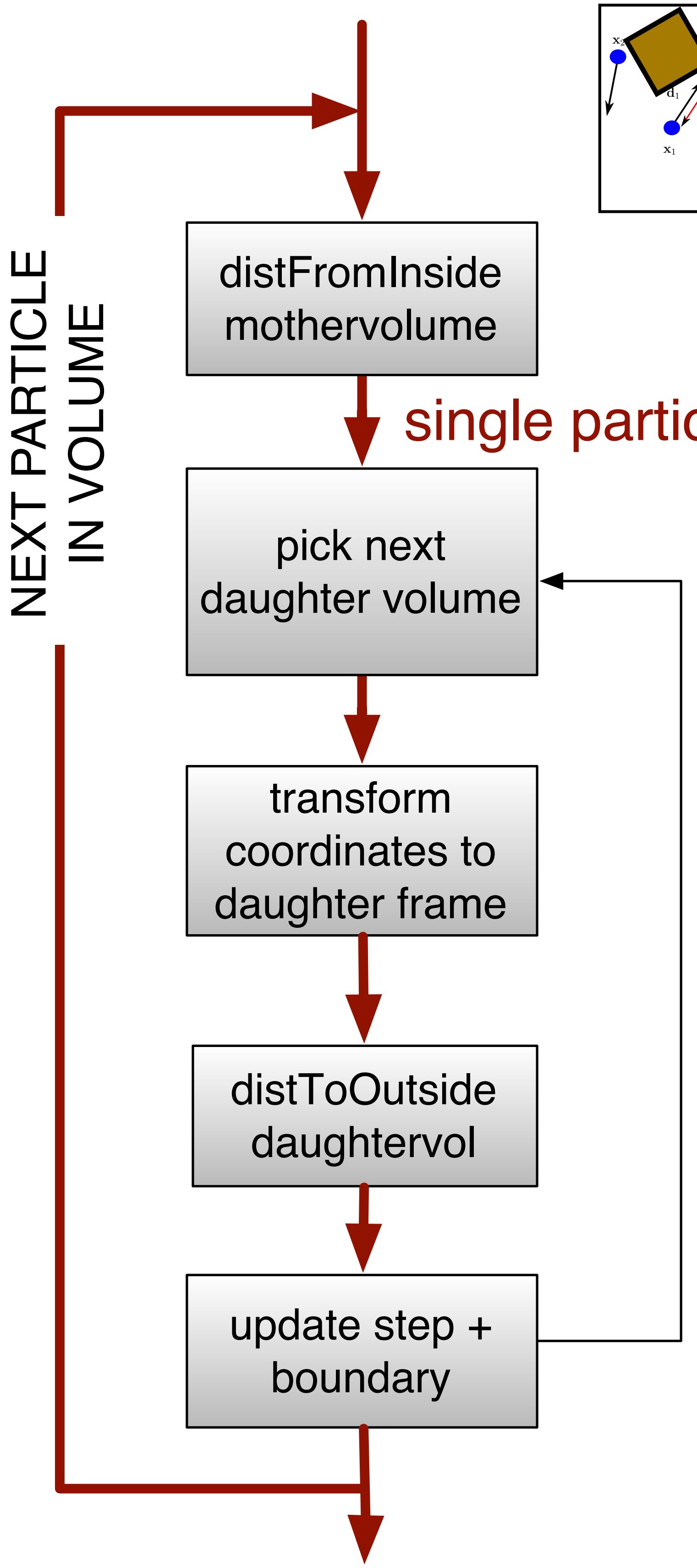
➡ **prototype study started ~04/2013**

1st Step: Vector Processing in Elementary Geometry Algorithms

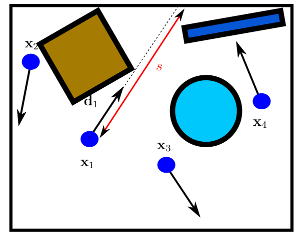


- Provide **new interfaces** to process baskets in **elementary** geometry algorithms
- make efficient use of baskets and try to use SIMD vector instructions wherever possible (**throughput optimization**)

2nd step: Vector processing in complex algorithms:

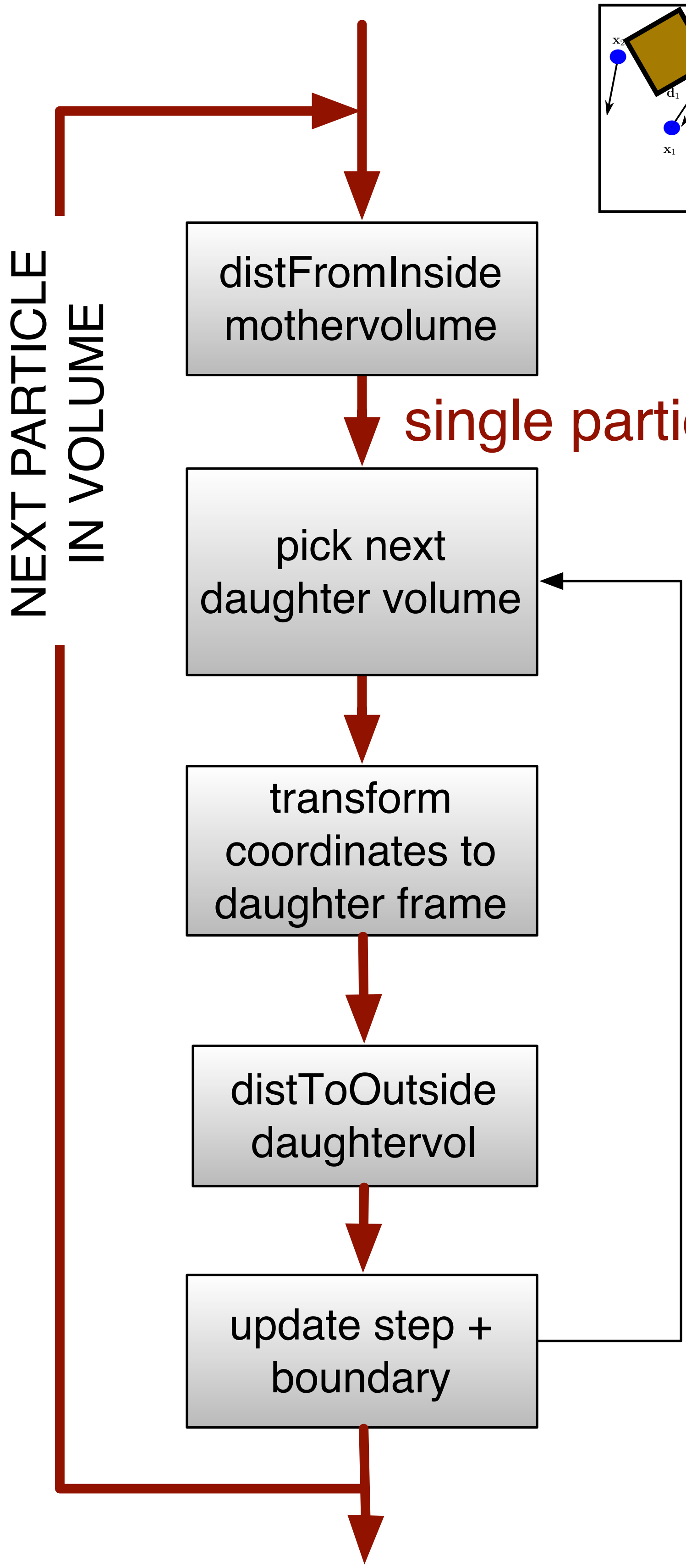


single particle flow

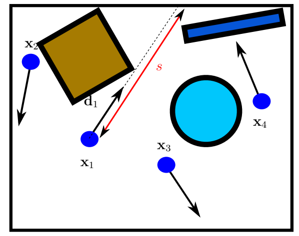


each particles undergoes a series of basic algorithms (with outer loop over particles)

2nd step: Vector processing in complex algorithms:



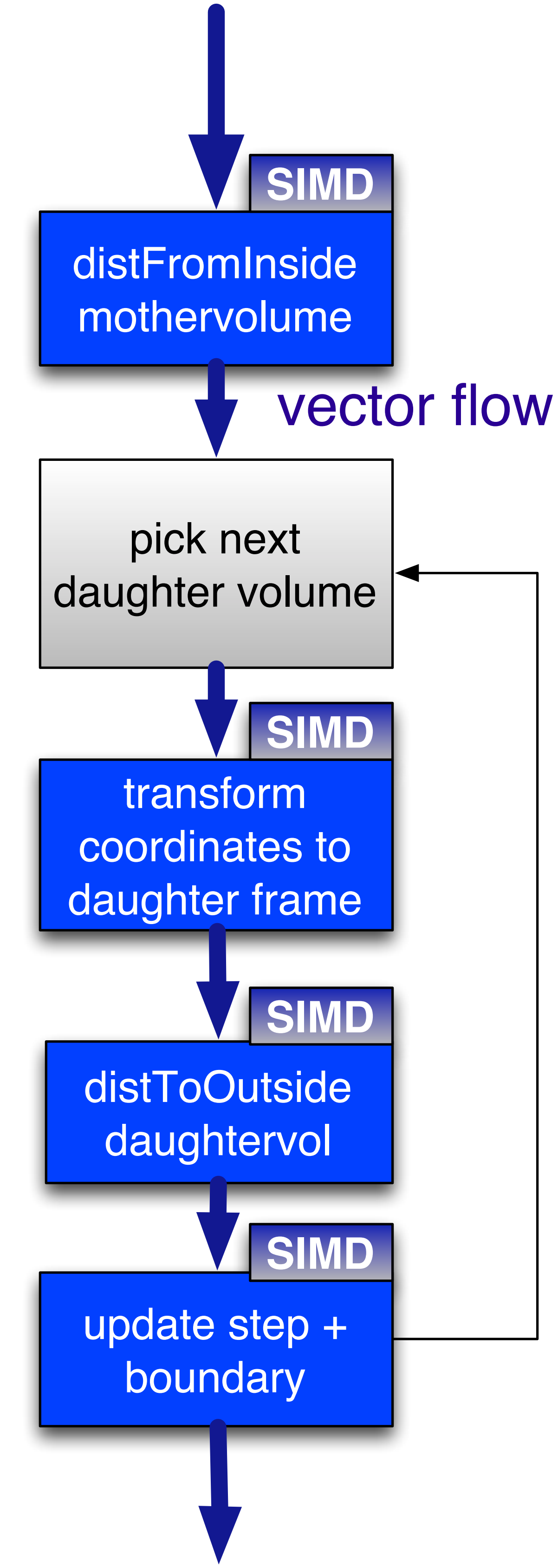
single particle flow



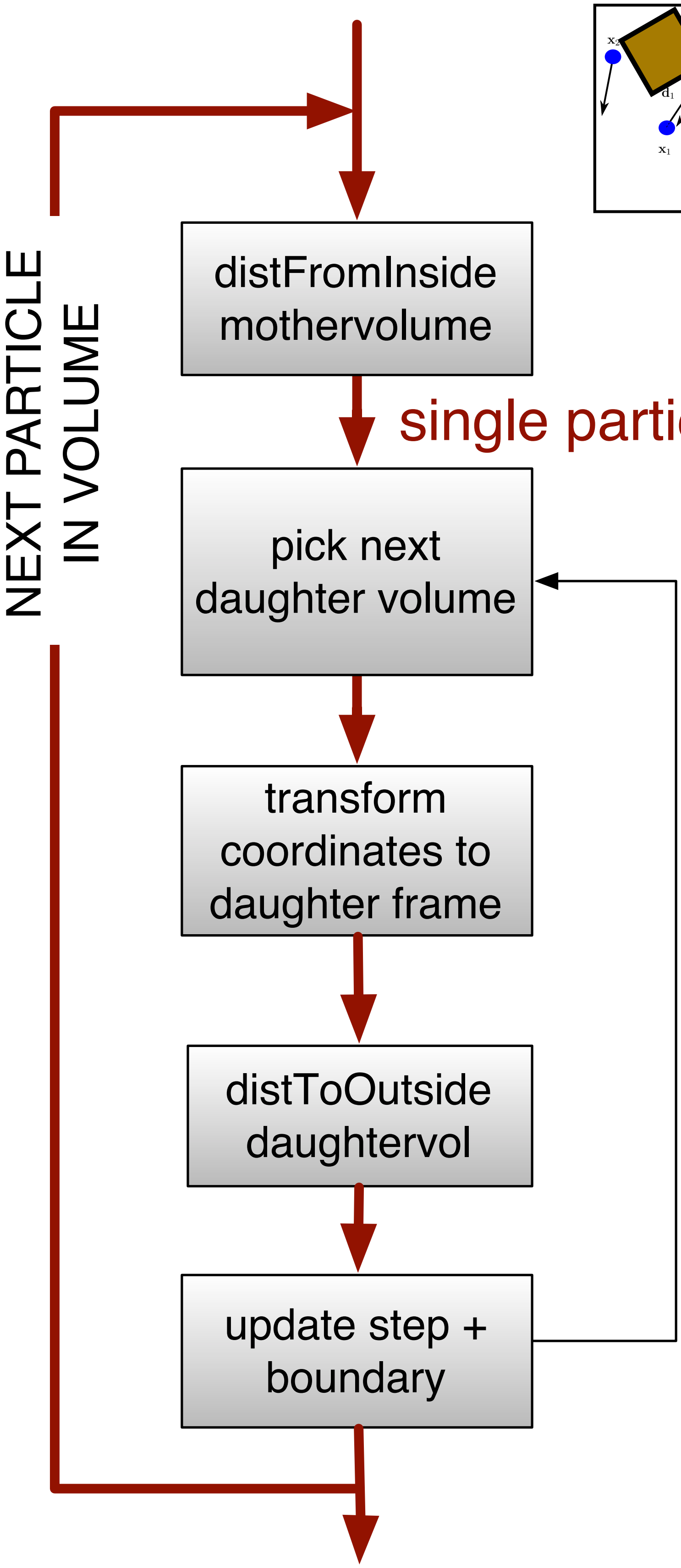
each particles undergoes a series of basic algorithms (with outer loop over particles)

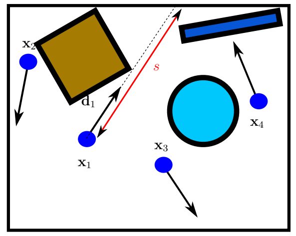
2nd step

Each algorithm takes a basket of particles and spits out vectors to the next algorithms



2nd step: Vector processing in complex algorithms:



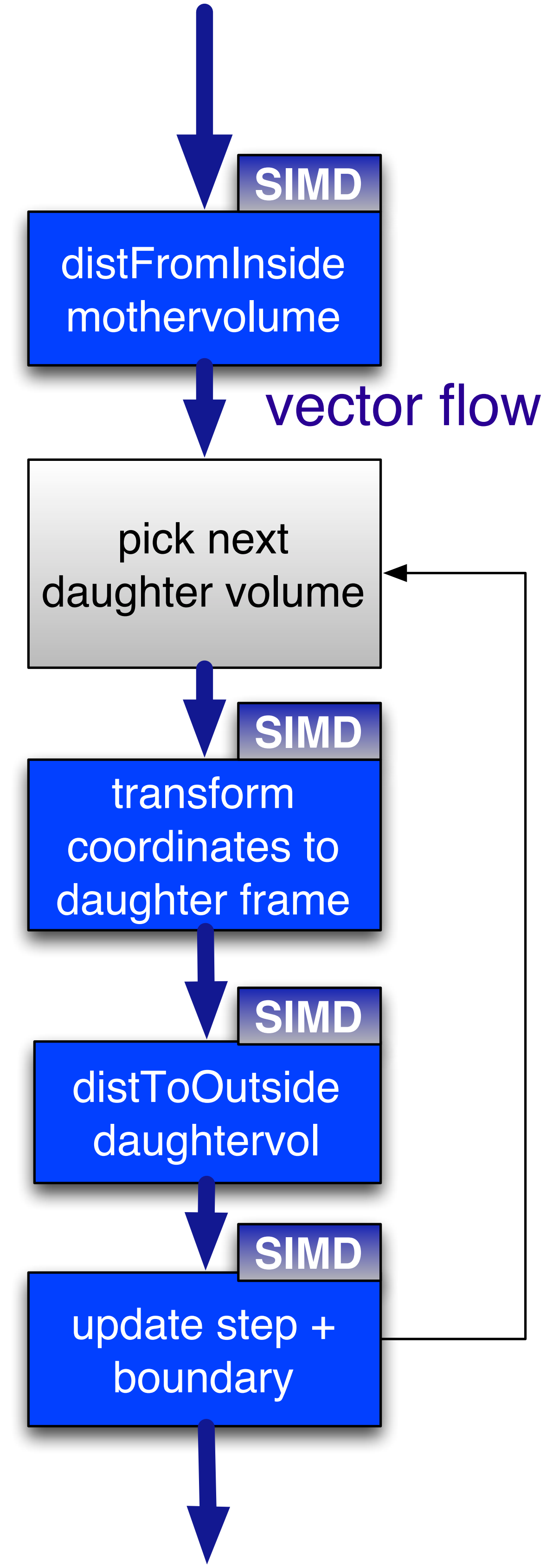


each particles undergoes a series of basic algorithms (with outer loop over particles)

2nd step

Each algorithm takes a basket of particles and spits out vectors to the next algorithms

- less function calls!
- SIMD (SSE, AVX) instructions
- better code locality (icache)



SIMD Vectorization Programming model

How to (particle) **vectorize existing code** (with many branches...) ?

Option A ("free lunch"):

put code into a loop and let the compiler do the work

- works in very few cases

SIMD Vectorization Programming model

How to (particle) **vectorize existing code** (with many branches...) ?

Option A ("free lunch"):

put code into a loop and let the compiler do the work

- works in very few cases

Option B ("convince the compiler"):

refactor the code to make it "auto-vectorizer" friendly

- might work but strongly compiler dependent

SIMD Vectorization Programming model

How to (particle) **vectorize existing code** (with many branches...) ?

Option A ("free lunch"):

put code into a loop and let the compiler do the work

- works in very few cases

Option B ("convince the compiler"):

refactor the code to make it "auto-vectorizer" friendly

- might work but strongly compiler dependent

Option C ("use SIMD library"):

refactor the code and perform explicit vectorization using a vectorization library

- always SIMD vectorizes, compiler independent
- excellent experience with the Vc library
- other libraries exist: VectorType (Agner Fog), Boost::SIMD, ...

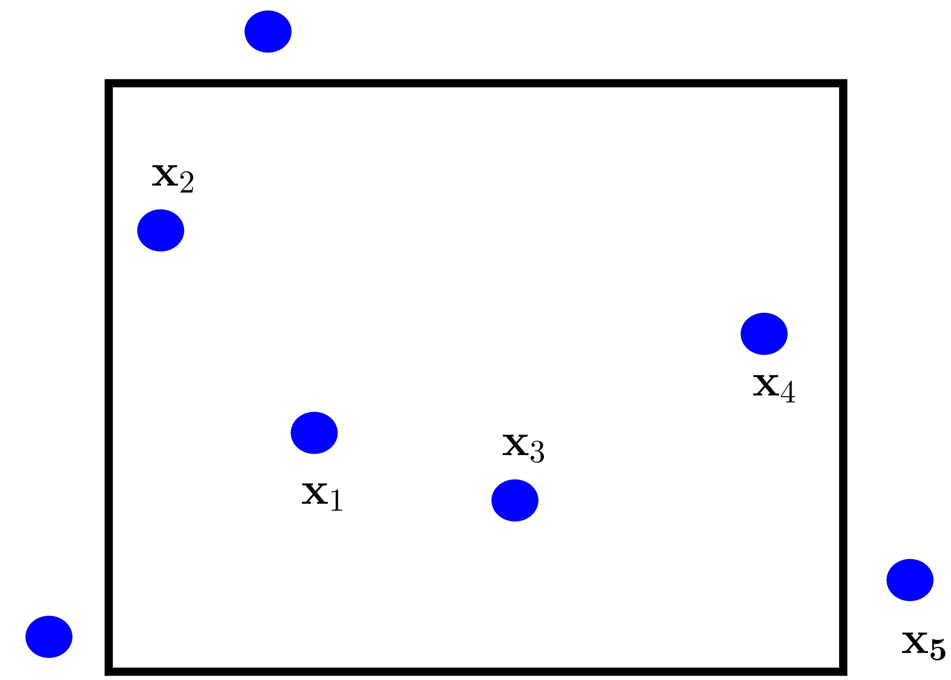
http://code.compeng.uni-frankfurt.de/projects/vc

```
// hello world example with Vc-SIMD types
Vc::Vector<double> a, b, c;
c=a+b;
```

“Option A: Free lunch vectorization”

- **starting point: some existing code** (here easy example)

```
bool contains( const double * point ){
    for( unsigned int dir=0; dir < 3; ++dir ){
        if( fabs (point[dir]-origin[dir]) > boxsize[dir] )
        return false;
    }
    return true;
}
```



- **provide vector-interface**, call basic/elemental function ... and hope that compiler autovectorizes ...

```
void contains_v( const double * point, bool * isin, int np ) {
    for( unsigned int k=0; k < np; ++k) {
        isin[k]=contains( &point[3*k] );
}}
```

Option B: convince the compiler

- massage/refactor original code to make the compiler autovectorize
  - copy scalar code to new function ("manual inline")
  - AOS - SOA conversion of data layout
  - early - return removal
  - manual loop unrolling

```
void contains_v_autovec( const P & points, bool *  isin, int np ){
  for (int k=0; k < np; ++k)
  {
      bool resultx=(fabs (point.coord[0][k]-origin[0]) > boxsize[0]);
      bool resulty=(fabs (point.coord[1][k]-origin[1]) > boxsize[1]);
      bool resultz=(fabs (point.coord[2][k]-origin[2]) > boxsize[2]);
      isin[k]=resultx & resulty & resultz;
}}
```

- **this is only version** that **autovectorizes uncondionally** with all compilers tested (icc 13, gcc 4.7/4.8)
- **uncondionally:** no pragmas or further platform/compiler dependent hints

Option C: Use vector library/classes

```
void contains_v_Vc( const P & points, bool *  isin, int np )
{
  for( int k=0; k < np; k+=Vc::double_v::Size)
  {
     Vc::double_m inside;
     inside  = (abs (Vc::double_v(point.coord[0][k])-origin[0]) < boxsize[0]);
     inside&= (abs (Vc::double_v(point.coord[1][k])-origin[1]) < boxsize[1]);
     inside&= (abs (Vc::double_v(point.coord[2][k])-origin[2]) < boxsize[2]);
     // write mask as boolean result
     for (int j=0;j<Vc::double_v::Size;++j){
        isin[k+j]=inside[j];
     }
}
```

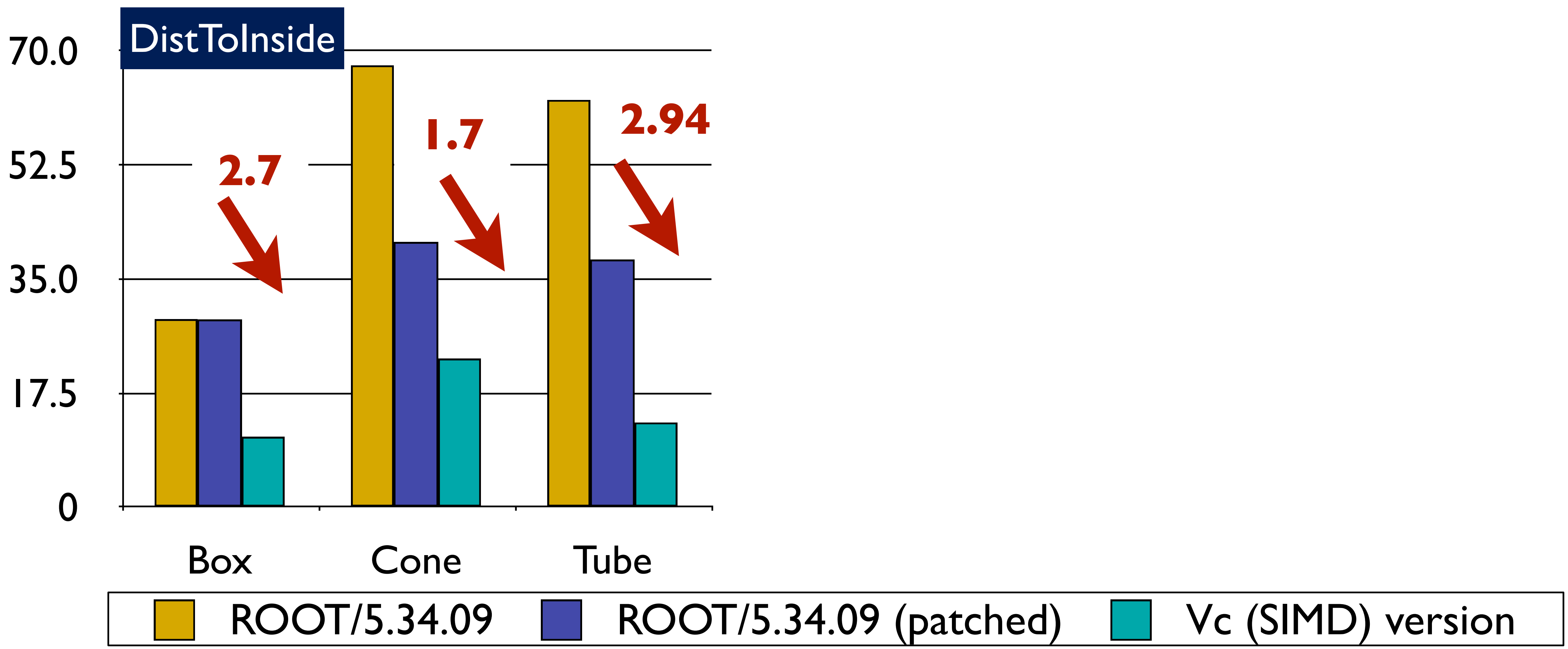
- almost same code as before using Vc library (see talk yesterday)
  - always vectorizes; don't have to convince compiler
  - excellent performance (automatically uses aligned data)
  - can mix vector context and scalar context (code)
  - given that we have to refactor code anyway, this is our implementation choice

Status of simple shape/algorithm investigations

- provided optimized code to simple shapes (box, tube, cone) for functions
  - **" DistToInside"**, **"DistToOutside", "Safety", "IsInside/Contains"**
  - here: using the ROOT shapes
  - For simple shapes the **performance gains match our expectations**

Status of simple shape/algorithm investigations

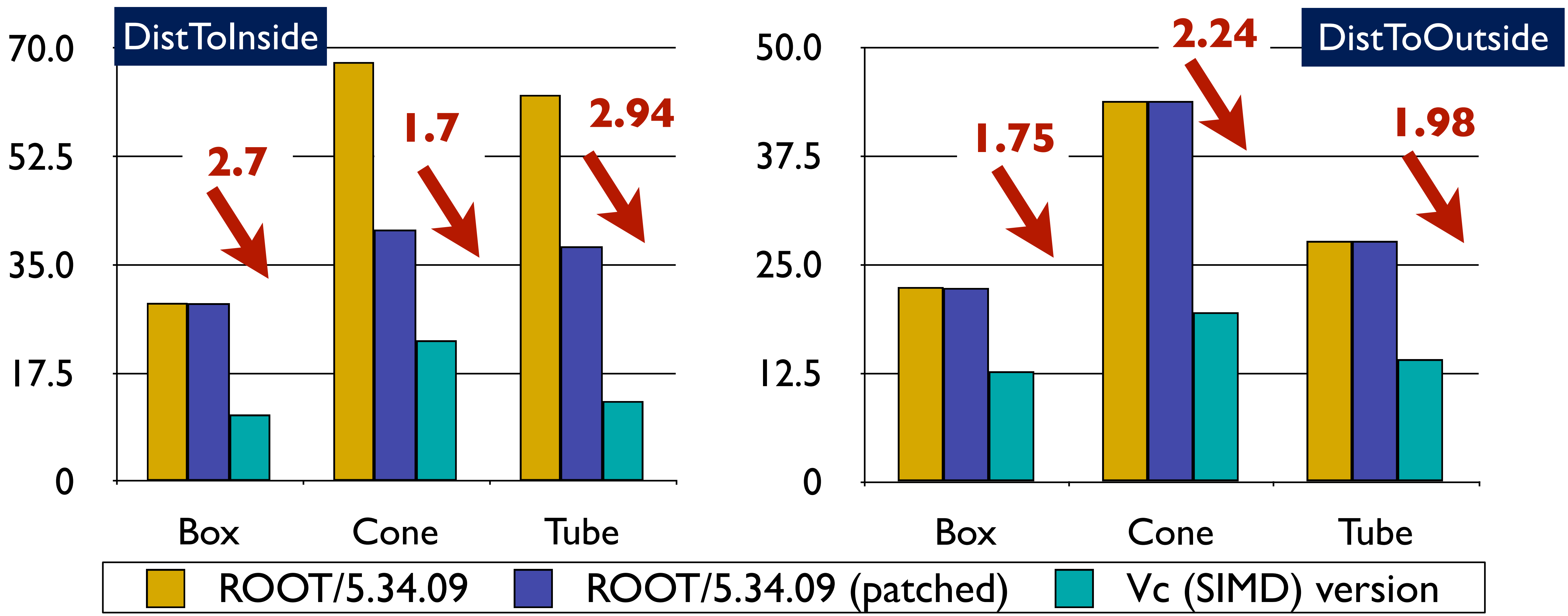
- provided optimized code to simple shapes (box, tube, cone) for functions
  - **" DistToInside", "DistToOutside", "Safety", "IsInside/Contains"**
  - here: using the ROOT shapes
  - For simple shapes the **performance gains match our expectations**



comparison of processing times for 1024 particles (AVX instructions), times in microseconds

Status of simple shape/algorithm investigations

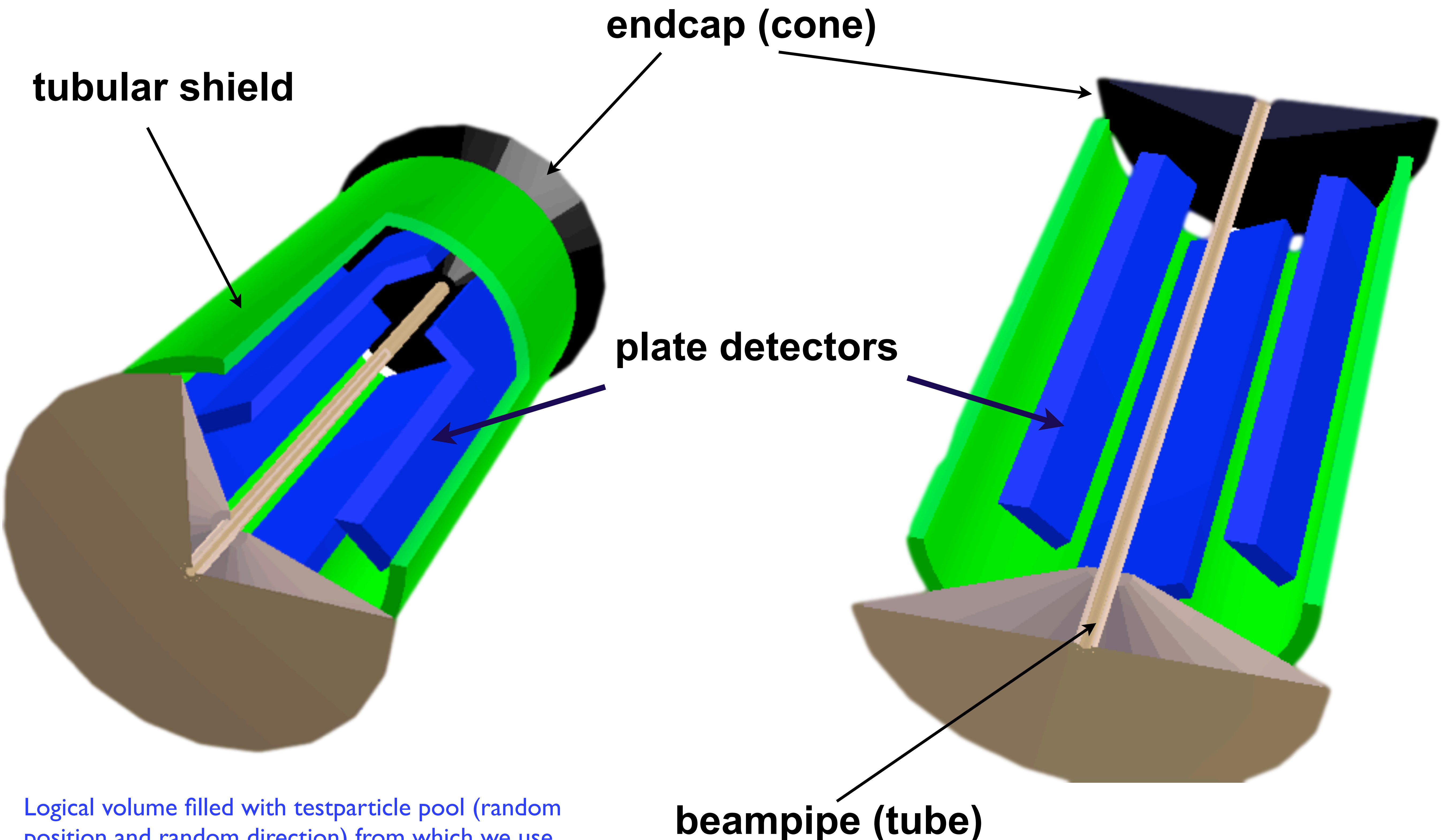
- provided optimized code to simple shapes (box, tube, cone) for functions
  - **" DistToInside", "DistToOutside", "Safety", "IsInside/Contains"**
  - here: using the ROOT shapes
  - For simple shapes the **performance gains match our expectations**



comparison of processing times for 1024 particles (AVX instructions), times in microseconds

Benchmark higher level navigation algorithm

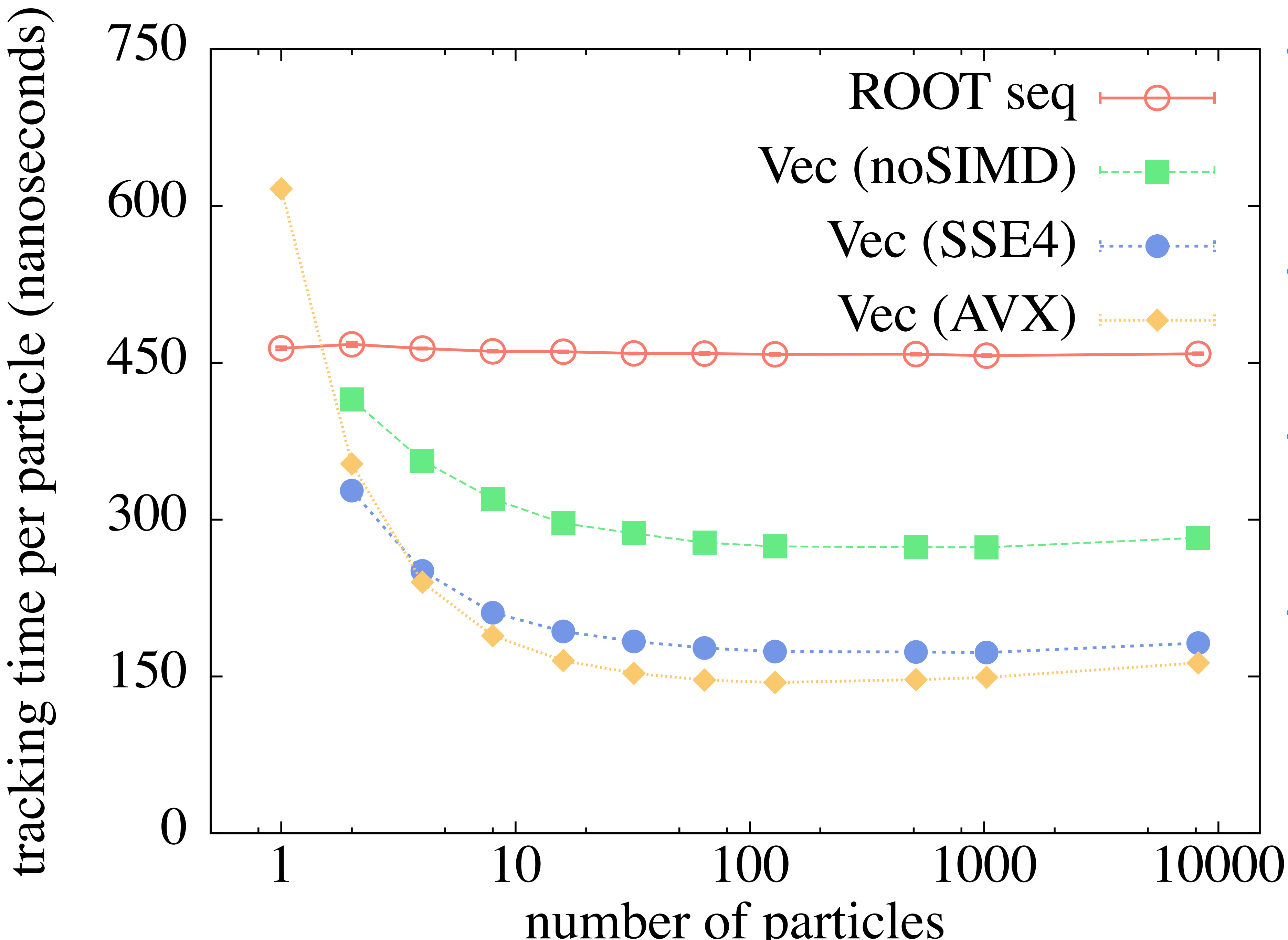
implemented a toy detector for a benchmark ("<u>not to easy; not too complex</u>"): 2 tubes, 4 plate detectors, 2 endcaps (cones), 1 tubular mother volume



Logical volume filled with testparticle pool (random position and random direction) from which we use a subset N for benchmarks (P repetitions)

Benchmark Results: Overall Runtime (CHEP13)

- time of processing/navigating N particles (P repetitions) using scalar algorithm (ROOT) versus vector version

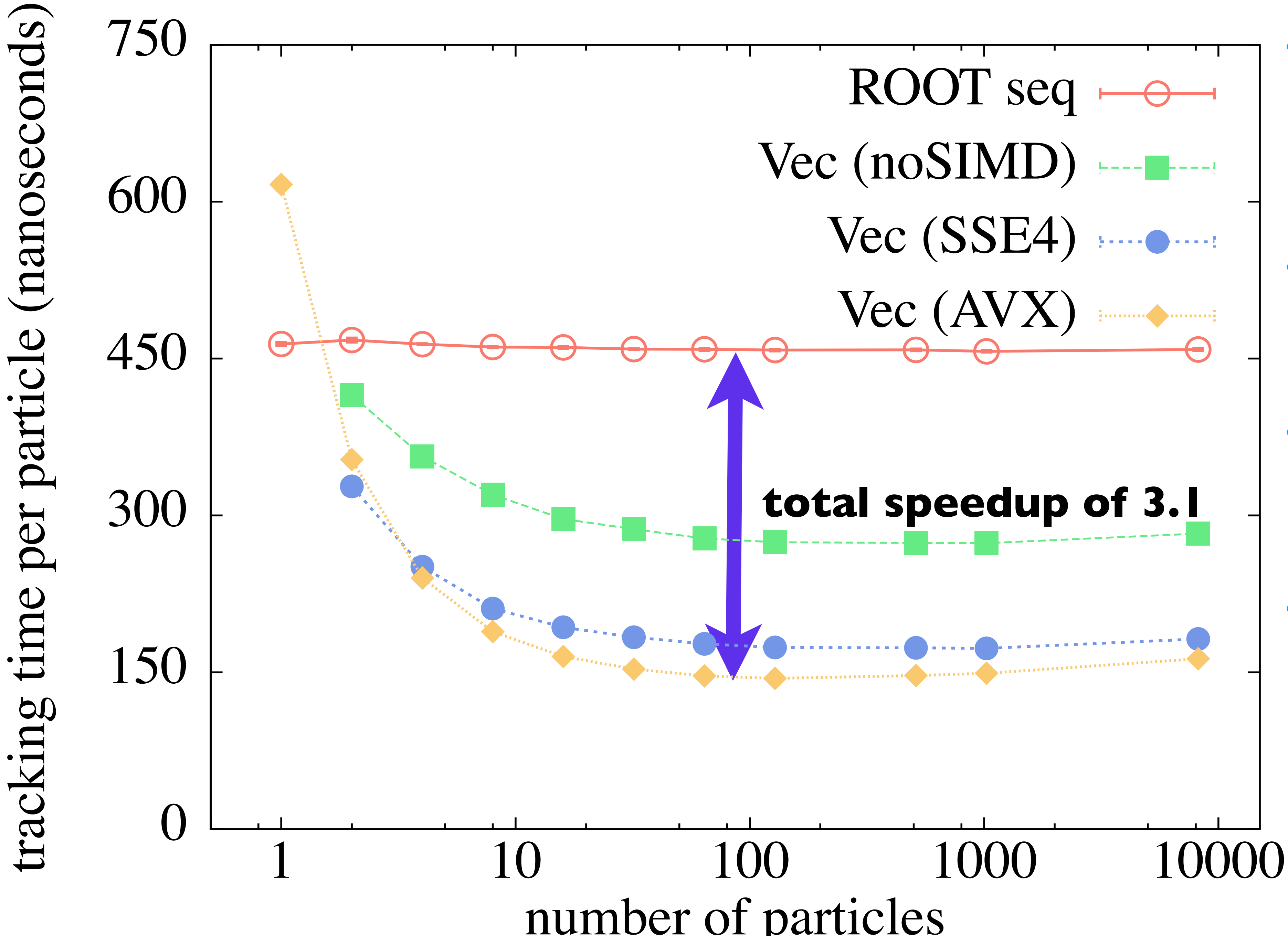


- free lunch gain due to treatment of baskets alone
- excellent speedup for SSE4 version
- some further gain with AVX
- already gain considerably for small N

CHEP13 paper: http://arxiv.org/pdf/1312.0816.pdf

Benchmark Results: Overall Runtime (CHEP13)

- time of processing/navigating N particles (P repetitions) using scalar algorithm (ROOT) versus vector version

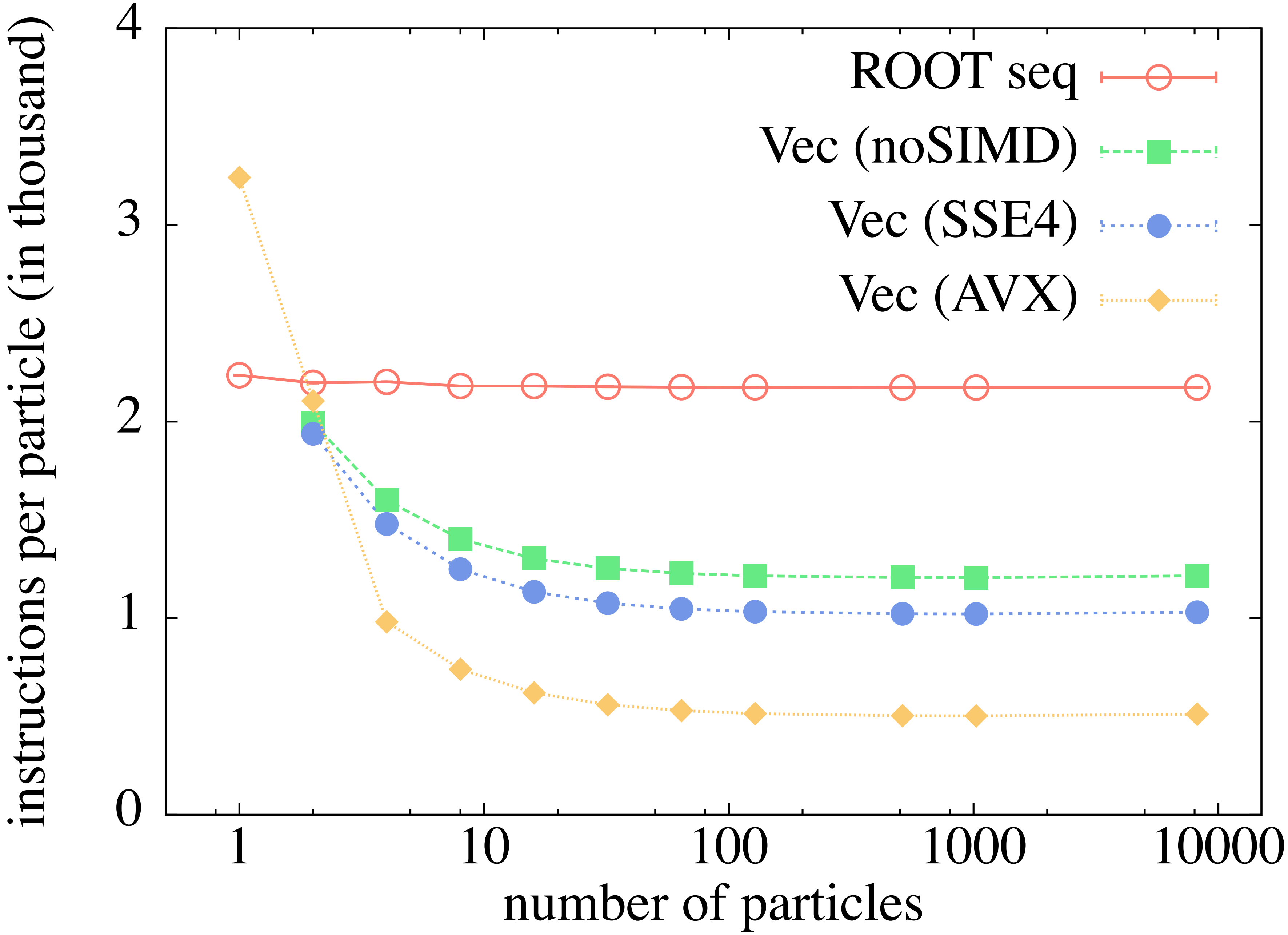


- free lunch gain due to treatment of baskets alone
- excellent speedup for SSE4 version
- some further gain with AVX
- already gain considerably for small N

CHEP13 paper: http://arxiv.org/pdf/1312.0816.pdf

Further Metrics: Executed Instructions

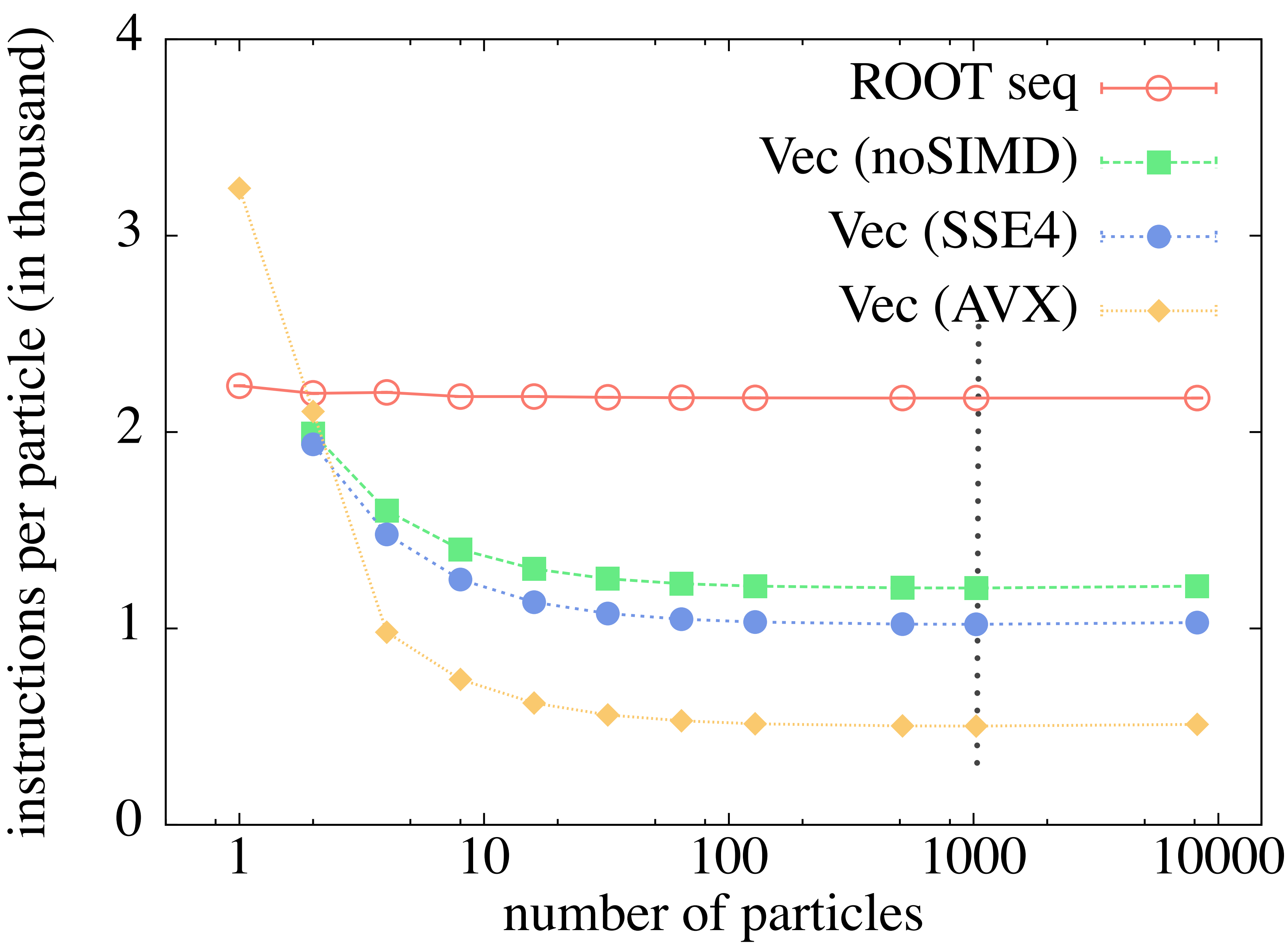
- investigate origin of speedup: study **hardware performance counters**; here number of instructions executed



- gain mainly **due to less instructions** (for the same work)

Further Metrics: Executed Instructions

* investigate origin of speedup: study **hardware performance counters**; here number of instructions executed



* gain mainly **due to less instructions** (for the same work)

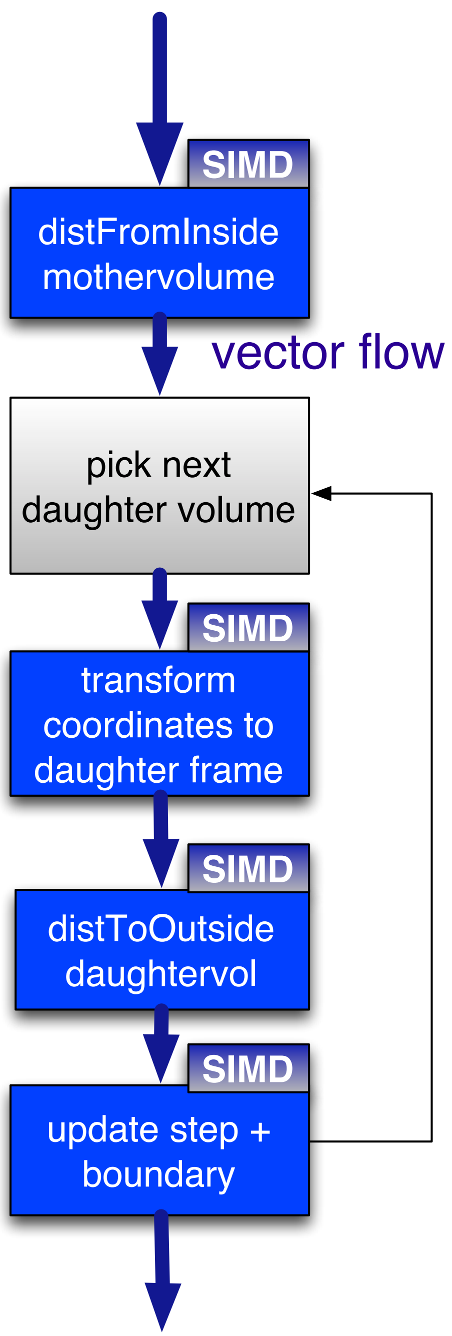
* detailed analysis (binary instrumentation) can give statistics, e.g.:

	ROOT	**Vec**
MOV	30%	15%
CALL	4%	0.4%
V..PD (SIMD instr)	5%	**55%**

comparison for N=1024 particles (AVX versus ROOT seq)

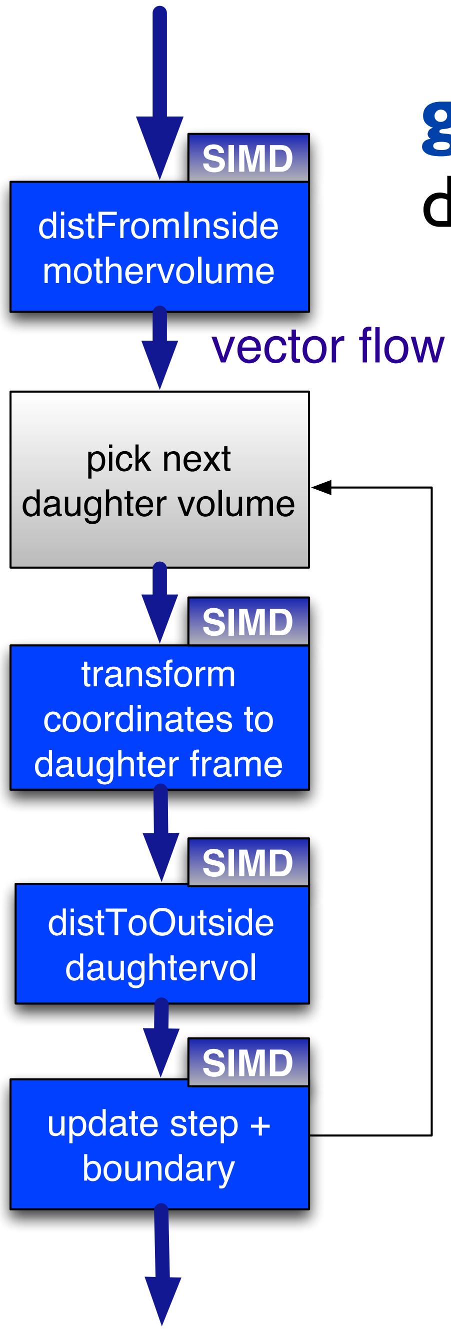
Current performance status (April 14)

since CHEP13, have improved the algorithms further



Current performance status (April 14)

since CHEP13, have improved the algorithms further



good overall performance gains for navigation algorithm (in toy detector with 4 boxes, 3 tubes, 2 cones) - compared to ROOT/5.34.17

	16 particles	1024 particles	SIMD MAX
Intel IvyBridge (AVX)	~2.8x	~4.0x	4x
Intel Haswell (AVX2)	~3.0x	~5.0x	4x
Intel Xeon-Phi (AVX512)	~4.1x	~4.8x	8x

Xeon-Phi and Haswell benchmarks by CERN Openlab (Georgios Bitzes)

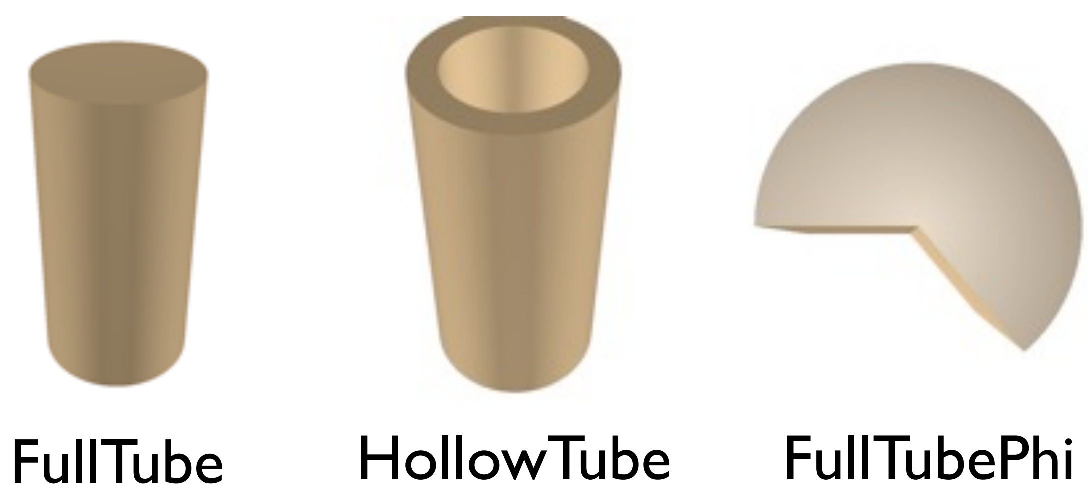
gcc 4.8; -O3 -funroll-loops -mavx; no FMA

Improving vectorization: C++ template techniques

"branches are the enemy of vectorization..."

a lot of branches in geometry code just distinguish between "static" properties of class instances

- general "tube solid" class distinguishes at runtime between "FullTube", "Hollow Tube" ...



Improving vectorization: C++ template techniques

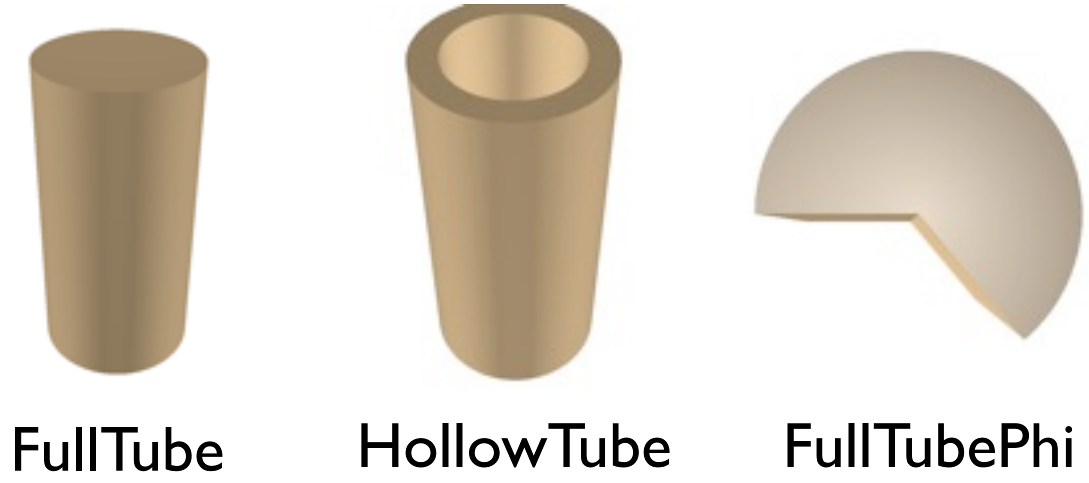
“branches are the enemy of vectorization...”

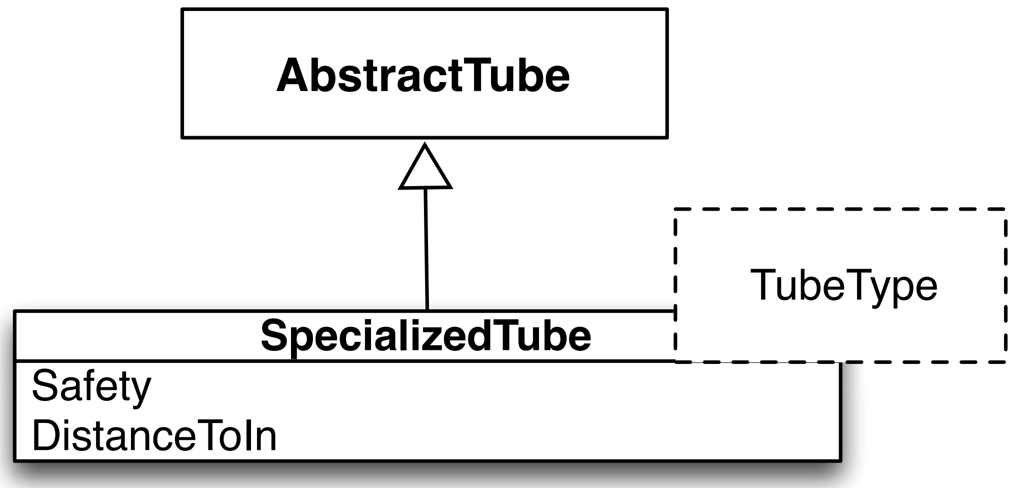
a lot of branches in geometry code just distinguish between “static” properties of class instances

- general “tube solid” class distinguishes at runtime between “FullTube”, “Hollow Tube” ...

we employ **template techniques** to:

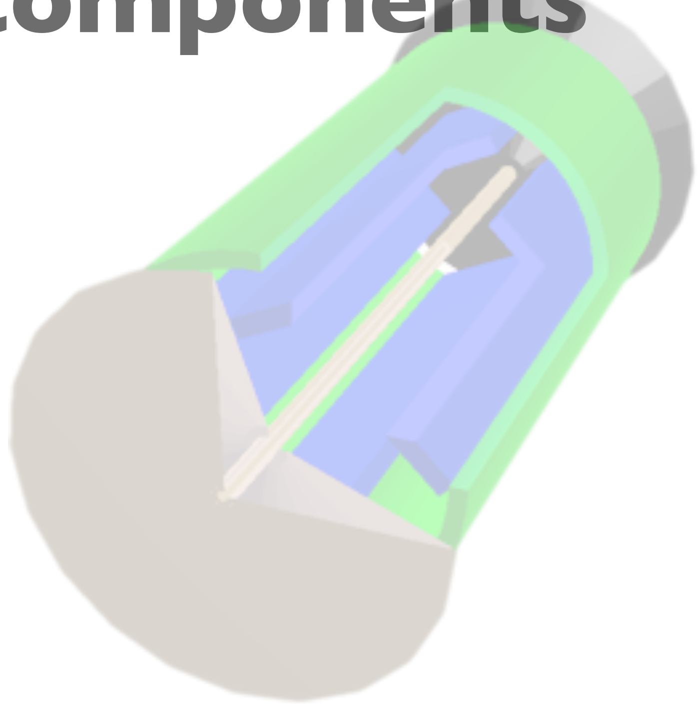
- evaluate and **reduce “static” branches at compile time**
- to **generate binary code specialized to concrete solid** instances
  - ➡ makes vectorization more efficient
  - ➡ allows better compiler optimizations in scalar code





Beyond the prototype: Towards a general high performance library for detector geometry

"vectorization everywhere"
"architecture abstraction"
"reusable generic components"



with contributions from

Georgios Bitzes (CERN Openlab)
Johannes De Fine Licht (CERN technical student)
Guilherme Lima (Fermilab)

Where do we go from here?

* It is now time to put these experiences/results into practice and provide a complete vectorized geometry library for simulation packages

Challenges from the software development perspective

- Lessons learned in small prototype
  - in prototype, had to refactor or rewrite code completely to achieve vectorization
  - vector code exists in addition to scalar code
- Should we follow same approach to port large existing code base in Geant4/ROOT/USolids geometry library?
  - maintenance nightmare
  - validation nightmare
- **Clearly the answer is no:** It would be nice to have code which can be used in both scalar and vector context (to large extentd)

Challenges continued

- How can we reuse the same code on the CPU + GPU?
  - the geometry library should be usable on different architectures
  - A vector friendly CPU functions is a good starting point for a kernel on the GPU; GPU could just reuse vector kernel in a different context
- How can we benefit from future advances in compiler technology (autovectorization)?
  - expressing algorithms with Vc often makes them suitable for autovectorization
  - we would like to stay flexible and possibly benefit from advances in this area
- How can we make code platform independent + vector implementation independent?
  - How can we play with other vector library implementations?
  - We'd like to use the best option available on a case by case basis (Vc, Boost::Simd, VectorClass (Agner Fog) as a function of performance and platform

Generic programming

- Generic programming with C++ templates provides the solution to all those problems
  - has been around for a long time and is among the few high-performance techniques of C++
  - not much used in HEP codes (at least not in simulation)
  - here, a very good option (inside a library implementation, almost not much user code) and probably almost without alternative
  - same approach as Vc (for instance) at a slightly higher level
- works very well with NVidia CUDA
- not (really) supported by pure OpenCL ...

A simple example for the generic approach

Example code for propagation of particles in a constant magnetic field ...

```
template<typename BaseDType, typename BaseIType>
void ConstBzFieldHelixStepper::DoStep(
            BaseDType const & x0, BaseDType const & y0, BaseDType const & z0,
            BaseDType const & dx0, BaseDType const & dy0, BaseDType const & dz0,
            BaseIType const & charge, BaseDType const & momentum, BaseDType const & step,
            BaseDType & x, BaseDType & y, BaseDType & z,
            BaseDType & dx, BaseDType & dy, BaseDType & dz
           ) const
  {
      const double kB2C_local = -0.299792458e-3;
      BaseDType dt = sqrt((dx0*dx0) + (dy0*dy0));
      BaseDType invnorm=1./dt;
      BaseDType R = momentum*dt/((kB2C_local*BaseDType(charge))*(fBz));
      BaseDType cosa= dx0*invnorm;
      BaseDType sina= dy0*invnorm;
      BaseDType helixgradient = dz0*invnorm*abs(R);

// some code omitted ...

       x = x0 + R*( -sina + cosphi*sina + sinphi*cosa ));
       y = y0 + R*( cosa  + sina*sinphi - cosphi*cosa ));
       z = z0 + helixgradient*phi;

       dx = dx0 * cosphi - sinphi * dy0;
       dy = dx0 * sinphi + cosphi * dy0;
       dz = dz0;
  }
```

A simple example for the generic approach

Example code for propagation of particles in a constant magnetic field ...

abstract types

```
template<typename BaseDType, typename BaseIType>
void ConstBzFieldHelixStepper::DoStep(
            BaseDType const & x0, BaseDType const & y0, BaseDType const & z0,
            BaseDType const & dx0, BaseDType const & dy0, BaseDType const & dz0,
            BaseIType const & charge, BaseDType const & momentum, BaseDType const & step,
            BaseDType & x, BaseDType & y, BaseDType & z,
            BaseDType & dx, BaseDType & dy, BaseDType & dz
           ) const
  {
      const double kB2C_local = -0.299792458e-3;
      BaseDType dt = sqrt((dx0*dx0) + (dy0*dy0));
      BaseDType invnorm=1./dt;
      BaseDType R = momentum*dt/((kB2C_local*BaseDType(charge))*(fBz));
      BaseDType cosa= dx0*invnorm;
      BaseDType sina= dy0*invnorm;
      BaseDType helixgradient = dz0*invnorm*abs(R);

// some code omitted ...

       x = x0 + R*( -sina + cosphi*sina + sinphi*cosa ));
       y = y0 + R*( cosa  + sina*sinphi - cosphi*cosa ));
       z = z0 + helixgradient*phi;

       dx = dx0 * cosphi - sinphi * dy0;
       dy = dx0 * sinphi + cosphi * dy0;
       dz = dz0;
  }
```

actual code read (almost) as usual

A simple example for the generic approach

Example code for propagation of particles in a constant magnetic field ...

```
template<typename BaseDType, typename BaseIType>
void ConstBzFieldHelixStepper::DoStep(
            BaseDType const & x0, BaseDType const & y0, BaseDType const & z0,
            BaseDType const & dx0, BaseDType const & dy0, BaseDType const & dz0,
            BaseIType const & charge, BaseDType const & momentum, BaseDType const & step,
            BaseDType & x, BaseDType & y, BaseDType & z,
            BaseDType & dx, BaseDType & dy, BaseDType & dz
           ) const
  {
      const double kB2C_local = -0.299792458e-3;
      BaseDType dt = sqrt((dx0*dx0) + (dy0*dy0));
      BaseDType invnorm=1./dt;
      BaseDType R = momentum*dt/((kB2C_local*BaseDType(charge))*(fBz));
      BaseDType cosa= dx0*invnorm;
      BaseDType sina= dy0*invnorm;
      BaseDType helixgradient = dz0*invnorm*abs(R);

// some code omitted ...

       x = x0 + R*( -sina + cosphi*sina + sinphi*cosa ));
       y = y0 + R*( cosa  + sina*sinphi - cosphi*cosa ));
       z = z0 + helixgradient*phi;

       dx = dx0 * cosphi - sinphi * dy0;
       dy = dx0 * sinphi + cosphi * dy0;
       dz = dz0;
  }
```

abstract types

actual code read (almost) as usual

Demonstrated use of this code in:

a) scalar sense

b) vectorization with Vc

c) autovectorization with Intel compiler

d) as the basis for a CUDA kernel

excellent for maintenance

“VecGeom”

- A project “VecGeom” was started to put those ideas into practice for the geometry
- merged with AIDA Unified Solids effort
- https://github.com/sawenzel/VecGeom.git
- current implementation status:
  - library abstraction layer to provide some abstractions on concepts that differ in various backends (masks, masked assignments, math functions, loopers)
  - generic templated implementations for few shapes (box, para, tube, cone)
  - geometry hierarchies on CPU and GPU
  - can be basis for GPU + Geant-V simulation prototypes (already used)
  - much reduced actual code base compared to previous situation with different versions for scalar and vector code

The prototype: summary

Goals

Performance

- optimized many particle treatment

Approach

SIMD

template techniques

algo + class review

- template class specialization / code generation

Implementation

Vc library

VecGeom : overview

Goals

Performance

- optimized many particle treatment
- optimized 1-particle functions
- optimized base types / containers

Abstraction

- SIMD abstraction
- CPU/GPU abstraction

Code reuse

- reusable components
- same code base for CPU/GPU where appropriate

Approach

SIMD

algo + class review

template techniques

- template class specialization / code generation
- generic programming

Implementation

Vc library | Cilk Plus | autovectorization | Boost::SIMD | ?

Summary

Part I:

promising SIMD results in geometry demonstrator

promoted use of vectorization in simulation codes

Part II:

promoted use of generic programming in HEP codes; working towards general high-performance geometry library that is

flexible,

portable,

performant,

maintainable due to reduced code size