Towards a high performance geometry library for particle-detector simulation

Sandro Wenzel / CERN-PH-SFT

In collaboration with: J. Apostolakis (CERN), M. Bandieramonte (University of Catania, IT), G. Bitzes (CERN), R. Brun (CERN), P. Canal (Fermilab), F. Carminati (CERN), G. Cosmo (CERN), J. De Fine Licht (CERN), L. Duhem (Intel), D. Elvira (Fermilab), A. Gheata (CERN), S. Yung Jun (Fermilab), G. Lima (Fermilab), T. Nikitina (CERN), M. Novak (CERN), R. Sehgal (Bhabha Atomic Research Centre), O. Shadura (CERN)
Geometry in simulation

• geometry tasks are a major consumer of CPU cycles in detector simulation

• most of time spent in interaction with **shape primitives** which make up a detector

CMS detector: boxes, trapezoids, tubes, cones, ..., polycones, ...

• For shape primitives, a geometry library offers an API to ...

  - in or out?
  - collision detection and distance to enter object
  - minimal(safe) distance to object
  - distance to leave object
Part I: Geometry in simulation
- review of ROOT, Geant4, USolids packages
- the need to go beyond current implementations
- software challenges

Part II: Introducing “VecGeom”
- overview
- performance and status update

Part III: Some details on generic programming approach
- shared scalar/vector (CUDA) kernels
Geometry/Solid - Packages

very widespread in HEP, medical physics, ...

GEANT4
geometry modeler
~1994-

ROOT/TGeo
~2002-

AIDA USOLIDS
~2010-

EU/AIDA funded effort to merge
the libraries (on shape level):
• merge code base
• pick best implementation
• improve performance
• increase code quality
• increase long term maintainability

experiments using virtual
Monte Carlo framework
(ALICE, FAIR) + ...
very widespread in HEP, medical physics, ...

**GEANT4**

geometry modeler

~1994-

**ROOT/TGeo**

~2002-

**AIDA USOLIDS**

~2010-

EU/AIDA funded effort to merge the libraries **on shape level**:  
- merge code base  
- pick best implementation  
- improve performance  
- increase code quality  
- increase long term maintainability

experiments using virtual Monte Carlo framework (ALICE, FAIR) + ...

**improvements:**  
- new polycone (~8x faster than Geant4/Root)

**completely new features:**  
- multi-union, tesselated solids
New needs/beyond USolids

- USolids made a big step forward improving shape primitive code
- experiments are able to see the benefits now; USolids can be used in Geant4 simulations today! PLEASE TRY !!
New needs/beyond USolids

- USolids made a big step forward improving shape primitive code
- experiments are able to see the benefits now; USolids can be used in Geant4 simulations today! PLEASE TRY !!

but: **new needs/requirements** not yet addressed by current implementations

- no interfaces to process many particles at once
- no use of external/internal **SIMD vectorization**
- no use of **HPC features of C++ ("templates")** which could further improve performance
- (no library support **on GPU**)
Targeting vectorization

- vector instructions getting more important; vector registers becoming wider
- these instructions have to be used to efficiently use compute architecture; need to have “vector” data on which we apply the same tasks
Targeting vectorization

- vector instructions getting more important; vector registers becoming wider
- these instructions have to be used to efficiently use compute architecture; need to have “vector” data on which we apply the same tasks

outer vectorization

“parallel” collision detection

makes “future” code faster
Targeting vectorization

- Vector instructions getting more important; vector registers becoming wider.
- These instructions have to be used to efficiently use compute architecture; need to have “vector” data on which we apply the same tasks.

**Outer vectorization**

- “Parallel” collision detection makes “future” code faster.

**Internal vectorization**

- Internal loop over lateral planes for distance calc.
- Vectorization of inner loops; not common in shape code; but feasible for a couple of shapes (trapezoid).
- Beneficial for current simulations.
Software challenges implied by goals

• How do we achieve **reliable** vectorization on CPU??
Software challenges implied by goals

• How do we achieve **reliable** vectorization on CPU ??

• How do we **cope with the multiplication of interfaces** ...?

<table>
<thead>
<tr>
<th>Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x,y,z)</td>
</tr>
<tr>
<td>double DistanceToIn( 1 particle )</td>
</tr>
<tr>
<td>double* DistanceToIn( many particles )</td>
</tr>
<tr>
<td>bool Contains ( 1 particle )</td>
</tr>
<tr>
<td>bool* Contains ( many particles )</td>
</tr>
<tr>
<td>double SafetyToIn( 1 particle )</td>
</tr>
<tr>
<td>double* SafetyToIn( many particles )</td>
</tr>
<tr>
<td>double DistanceToOut ( 1 particle )</td>
</tr>
<tr>
<td>double* DistanceToOut( many particles )</td>
</tr>
</tbody>
</table>

>4 new functions per solid

\(~20\) primitive solids

\(~100\) new functions to maintain ( not including CUDA yet ... )
Approach to target software challenge

- solid primitives
- 1 particle API
  - common C++ template functions
- many particle API
- template C++ programming solves code multiplication issue
Approach to target software challenge

- reliable efficient SIMD vectorization achieved by using vector libraries (e.g., Vc) providing C++ approach to explicit vectorization
  
http://code.compeng.uni-frankfurt.de/projects/vc

- template C++ programming solves code multiplication issue
Approach to target software challenge

- **solid primitives**

  - **1 particle API**
    - scalar types

  - **many particle API**
    - vector types

- **common C++ template functions**

  - reliable efficient SIMD vectorization achieved by using vector libraries (e.g., Vc) providing C++ approach to explicit vectorization
    - [http://code.compeng.uni-frankfurt.de/projects/vc](http://code.compeng.uni-frankfurt.de/projects/vc)

  - template C++ programming solves code multiplication issue

- **nothing here is specific to geometry !!!**
“VecGeom”

- geometry primitive code development is now seen as long-term **evolution of USolids**
- already developed back-to-back with USolids; sharing a repository; same interfaces

- started as feasibility study of vectorization in geometry
- now “evolved” to project addressing all goals and challenges presented

**GEANT4 geometry modeler**

~1994-

**AIDA USOLIDS**

~2002-

**ROOT/TGeo**

~2010-

codename “VecGeom”

~2013-
Part II: Status + Performance
Performance case study: the tube segment

- most used/important shape primitive
- also integral part of complex shapes: polycone
- extremely important to be as fast as we can
Performance case study: the tube segment

- most used/important shape primitive
- also integral part of complex shapes: polycone
- extremely important to be as fast as we can

![Graph showing performance comparison of different libraries and APIs](image)


gcc 4.7; -O3 -funroll-loops -mavx; no FMA; Geant4 10 (Release); Root 5.34.18 (Release); benchmark with 1000 particles
Performance case study: the tube segment

- **most used/important** shape primitive
- also **integral part** of complex shapes: polycone
- extremely **important to be as fast as we can**

![Performance chart]

**improved scalar performance**
- improved algorithms (avoid atan2)
- template shape specialization

---

gcc 4.7; -O3 -funroll-loops -mavx; no FMA; Geant4 10 (Release); Root 5.34.18 (Release); benchmark with 1000 particles
Performance case study: the tube segment

- **most used/important** shape primitive
- **also integral part** of complex shapes: polycone
- **extremely important to be as fast as we can**

![Graph showing performance comparison]

**improved scalar performance**
- improved algorithms (avoid atan2)
- template shape specialization

**excellent SIMD vector performance**

<table>
<thead>
<tr>
<th></th>
<th>DistanceToIn</th>
<th>SafetyToIn</th>
<th>In-or-Out?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROOT</td>
<td>3.3x</td>
<td>7x</td>
<td>13.62x</td>
</tr>
<tr>
<td>Geant4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USolids</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VecGeom ScalarAPI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VecGeom ManyParticle API</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GCC 4.7; -O3 -funroll-loops -mavx; no FMA; Geant4 10 (Release); Root 5.34.18 (Release); benchmark with 1000 particles
Solid/shape implementation status; performance

timings for collision detection for various primitives

timing points form a polygon per library

Sandro Wenzel, ACAT2014
Solid/shape implementation status; performance

timings for collision detection for various primitives

timing points form a polygon per library

smaller area = better library performance
going complex...

- boolean solids are an important element in detector construction (subtraction solid, union solid)
- Geant4+Root offer construction of such objects based on a solid base class and virtual functions

```
SubtractionSolid( AbstractShape * left, AbstractShape * right );
```
going complex...

- boolean solids are an important element in detector construction (subtraction solid, union solid)
- Geant4+Root offer construction of such objects based on a solid base class and virtual functions

```cpp
SubtractionSolid( AbstractShape * left, AbstractShape * right );
```

- now offer advanced way to combine shapes (ala stl)

```cpp
template <typename LeftSolid, typename RightSolid>
class TSubtractionSolid
{
    TSubtractionSolid( LeftSolid * left, RightSolid * right );
};
```

- compiler can produce optimized code for any combination of primitive shapes ("template-shape specialization")
- no virtual function calls
- vectorization comes from reusing vector functions of components
going complex (condt)

- performance example for a subtraction solid “box minus tubesegment” (in CMS detector)

SIMD/ROOT speedup: \(8x\)  
SIMD/Geant4 speedup: \(6.6x\)
"VecGeom" and Geant-V

(templated/specialized) solid primitives

1 particle API

many particle API targeting SIMD vectorization

target use

common C++ template functions

detector description

functionality to create hierarchies of volumes = detector on CPU + GPU

detector navigation

Scalar navigation

Vector navigation

Geant-V / GPU prototype need additional library components to fully use vectorized shapes:
• shape hierarchies on CPU + GPU
• vector navigator
“VecGeom” in action

- Geant-Vector prototype can run complete first particle-detector simulations using VecGeom (or with ROOT/TGeo)
- Measured a total simulation runtime improvement of 40% going from ROOT/TGeo to VecGeom for small example

ExN03 example

- Should be able to simulate with CMS detector soonish....
Part III: Some details on programming approach
achieving shared scalar / vector code

double distance( double );

Vc::double_v distance( Vc::double_v );

remember...

common C++ template functions
achieving shared scalar / vector code

remember...

double distance( double );

Vc::double_v distance( Vc::double_v );

template<class Backend>
Backend::double_t
common_distance_function( Backend::double_t input )
{
  // complicated code implementing this function
  // using abstract types that Backend provides
}
achieving shared scalar / vector code

remember...

1 particle
API
targeting
SIMD
vectorization

many
particle
API

common
C++
template functions

double distance( double );

Vc::double_v distance( Vc::double_v );

template<class Backend>
Backend::double_t
common_distance_function( Backend::double_t input )
{
    // complicated code implementing this function
    // using abstract types that Backend provides
}

• “Backend” is a (trait) struct encapsulating standard types/properties for “scalar, vector, CUDA” programming; makes information injection into template function easy

struct ScalarBackend
{
    typedef double double_t;
    typedef bool   bool_t;
    static const bool IsScalar=true;
    static const bool IsSIMD=false;
};

struct VectorBackend
{
    typedef Vc::double_v double_t;
    typedef Vc::double_m bool_t;
    static const bool IsScalar=false;
    static const bool IsSIMD=true;
};

attention: this is not valid C++ code; need an additional “typename” before Backend
shared scalar-vector code: example

• toy example: calculate distance of particles to a Point represented by class Point with members (fX,fY,fZ)

• Point class offers 2 “distance” interfaces inlining same template function

<table>
<thead>
<tr>
<th>Point</th>
<th>double Distance(Vector3D&lt;double&gt; a)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>return DistanceKernel&lt;ScalarBackend&gt;(a);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Point</th>
<th>double_v Distance(Vector3D&lt;double_v&gt; a)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>return DistanceKernel&lt;VectorBackend&gt;(a);</td>
</tr>
</tbody>
</table>

attention: this is not valid C++ code; need an additional "typename" before Backend
shared scalar-vector code: example

- toy example: calculate distance of particles to a Point represented by class Point with members (fX,fY,fZ)
- Point class offers 2 “distance” interfaces inlining same template function

```
template<typename Backend>
inline __attribute__((always_inline))
Backend::double Point::DistanceKernel(const Vector3D<Backend::double_t> & point)
{
    Backend::double_t xp = fX - point.x();
    Backend::double_t yp = fY - point.y();
    Backend::double_t zp = fZ - point.z();
    // might have some Backend specific code
    if( Backend::IsScalar )
    {
        // we are able to diverge the code paths between different backends
        return Sqrt(xp*xp + yp*yp + zp*zp);
    }
}
```

```
Point::Distance(Vector3D<double> a)
{
    return DistanceKernel<ScalarBackend>(a);
}
```

```
Point::Distance(Vector3D<double_v> a)
{
    return DistanceKernel<VectorBackend>(a);
}
```

produces solid SIMD code

```
Vc::double_v
Point::Distance(Vector3D<Vc::double_v> a)
{
    return DistanceKernel<VectorBackend>(a);
}
```

attention: this is not valid C++ code; need an additional "typename" before Backend
Summary

- VecGeom is a detector geometry library which:
  - is **fast**
  - offers **vectorized** multi-particle treatment
  - follows **generic programming approach** to reduce code size
  - (supports CUDA and GPU)
- development model could be extended to other components of Geant-V prototype
Backup
Shape specialization by example

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

- If statements (“branches”) in **generic** code can be compiled away

```cpp
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**” if statements 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 `TubeType`es