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)
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

- Gain in memory footprint and time-to-solution but not in throughput
- Micro-parallelism: gain in throughput and in time-to-solution

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

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

Targeted by Geant-V
(track parall.)

Used by Geant4-MT
(event parall.)

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

50 per cent of the time spent in 0.7% volumes

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

data parallelism in a logical volume; same geometry code; shared physics code
Vectorizing geometry: The problem statement

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

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

1 particle
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**

![Diagram of particle tracking](image1)

1 particle

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

![Diagram of particle tracking](image2)

vectors of particles

functionality targeted by future simulation approaches

aim for efficient utilization of current and future hardware

➡ prototype study started ~04/2013

Sandro Wenzel

5th International Workshop on Future Challenges in Tracking and Trigger
1st Step: Vector Processing in Elementary Geometry Algorithms

1. Milestone

- 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:

- distFromInside mothervolume
- single particle flow
- pick next daughter volume
- transform coordinates to daughter frame
- distToOutside daughtervol
- update step + boundary

each particle undergoes a series of basic algorithms (with outer loop over particles)
2nd step: Vector processing in complex algorithms:

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

NEXT PARTICLE IN VOLUME

- distFromInside mothervolume
- single particle flow
- pick next daughter volume
- transform coordinates to daughter frame
- distToOutside daughtervol
- update step + boundary

vector flow

Each algorithm takes a basket of particles and spits out vectors to the next algorithms.
2nd step: Vector processing in complex algorithms:

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

- distFromInside motherVolume
- pick next daughter volume
- transform coordinates to daughter frame
- distToOutside daughterVol
- update step + boundary

2nd step

Single particle flow

- SIMD
- vector flow

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
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

```cpp
// 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)

```cpp
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 ...

```cpp
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 auto-vectorize
- copy scalar code to new function ("manual inline")
- AOS - SOA conversion of data layout
- early - return removal
- manual loop unrolling

```c
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 auto-vectorizes unconditionally with all compilers tested (icc 13, gcc 4.7/4.8)
* unconditionally: no pragmas or further platform/compiler dependent hints
Option C: Use vector library/classes

```cpp
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**
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**

<table>
<thead>
<tr>
<th>Shape</th>
<th>ROOT/5.34.09</th>
<th>ROOT/5.34.09 (patched)</th>
<th>Vc (SIMD) version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box</td>
<td></td>
<td>2.7</td>
<td>1.7</td>
</tr>
<tr>
<td>Cone</td>
<td>2.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tube</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Shape</th>
<th>DistToInside</th>
<th>DistToOutside</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box</td>
<td>2.7</td>
<td>1.75</td>
</tr>
<tr>
<td>Cone</td>
<td>1.7</td>
<td>2.24</td>
</tr>
<tr>
<td>Tube</td>
<td>2.94</td>
<td>1.98</td>
</tr>
</tbody>
</table>

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

- ROOT/5.34.09
- ROOT/5.34.09 (patched)
- Vc (SIMD) version

Sandro Wenzel
5th International Workshop on Future Challenges in Tracking and Trigger
Benchmark higher level navigation algorithm

implemented a toy detector for a benchmark ("not to easy; not too complex"): 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

![Graph showing benchmark results]

- 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

**Table 1.** Dynamic instruction count statistics for instructions executed in the actual algorithm of Figure 1. Shown are what fractions of the total instruction number are taken by simple memory moves (MOV), call instructions (CALL), SIMD instructions (ALL SIMD) as well as arithmetic SIMD instructions (ARITHM SIMD). These numbers are obtained for 16 particles and a comparison is done between the four algorithmic versions mentioned in the text. Some absolute numbers are given in [brackets](http://arxiv.org/pdf/1312.0816.pdf) (with an arbitrary but consistent scale).

Benchmark Results: Overall Runtime (CHEP13)

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

### Figure 3

Results from the benchmark comparing the scalar and sequential algorithm with a vector-oriented algorithm and various degrees of usage of SIMD instructions (from scalar fallback to AVX).

- (a) Comparison of the runtime per particle showing a speedup factor of roughly 3 comparing the original version to the AVX code.
- (b) Comparison of actual instructions executed per particle.

### Table 1

Dynamic instruction count statistics for instructions executed in the actual algorithm of Figure 1. Shown are what fractions of the total instruction number are taken by simple memory moves (MOV), call instructions (CALL), SIMD instructions (ALL SIMD) as well as arithmetic SIMD instructions (ARITHM SIMD). These numbers are obtained for 16 particles and a comparison is done between the four algorithmic versions mentioned in the text. Some absolute numbers are given in [brackets](http://arxiv.org/pdf/1312.0816.pdf) (with an arbitrary but consistent scale).

- Instruction (type) ROOT seq Vec (noSIMD) Vec (SSE4) Vec (AVX)
- ALL [17554373] [10608655] [9230632] [5024520]
- MOV 0 [5211457] 0 [1227198] 0 [819813]
- CALL 0 [634933] 0 [24601] 0 [24601] 0 [24601]
- ALL SIMD 0 [17554373] [9230632] [5024520]
- ARITHM SIMD 0 [17554373] [9230632] [5024520]

We can track the origin of the gains somewhat in analyzing the dynamic instruction mix actually executed in the benchmark. Some important numbers obtained from this analysis are summarized in Table 1. Essentially, it can be seen that going from a scalar to vector interface allows to reduce the number of function call instructions accompanied by a massive reduction in simple memory moves (such as to save registers on the stack). Further, when introducing SIMD optimisations with Vec, the overall number of instructions further shrinks and the CPU vector unit is used to a much higher degree. Using the hardware performance counters, we have also confirmed that the number of instruction cache misses is considerably reduced due to better code locality of using the vectorised interfaces. We expect this effect to become even more important with more complex algorithms. It should be noted that the current numbers are a first optimistic results. As such they define a new baseline for possible next iterations of optimisations, which we should mention that these number may be influenced by other factors that were optimized in the porting process; for instance: removal of check whether to calculate safety within some functions.
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).

![Graph showing number of instructions per particle vs. number of particles for different vector and sequential algorithms.](image)

Table 1.

<table>
<thead>
<tr>
<th>Instruction (type)</th>
<th>ROOT seq</th>
<th>Vec (noSIMD)</th>
<th>Vec (SSE4)</th>
<th>Vec (AVX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>17554373</td>
<td>10608655</td>
<td>9230632</td>
<td>5024520</td>
</tr>
<tr>
<td>MOV</td>
<td>0.296</td>
<td>0.116</td>
<td>0.132</td>
<td>0.163</td>
</tr>
<tr>
<td>CALL</td>
<td>0.0036</td>
<td>0.0023</td>
<td>0.0026</td>
<td>0.0048</td>
</tr>
<tr>
<td>ALL SIMD</td>
<td>0.043</td>
<td>0.188</td>
<td>0.641</td>
<td>0.57</td>
</tr>
<tr>
<td>ARITHM SIMD</td>
<td>0.023</td>
<td>0.039</td>
<td>0.289</td>
<td>0.30</td>
</tr>
</tbody>
</table>

We can track the origin of the gains somewhat in analyzing the dynamic instruction mix actually executed in the benchmark. Some important numbers obtained from this analysis are summarized in Table 1. Essentially, it can be seen that going from a scalar to vector interface allows to reduce the number of function call instructions accompanied by a massive reduction in simple memory moves (such as to save registers on the stack). Further, when introducing SIMD optimisations with Vec, the overall number of instructions further shrinks and the CPU vector unit is used to a much higher degree. Using the hardware performance counters, we have also confirmed that the number of instruction cache misses is considerably reduced due to better code locality. We expect this effect to become even more important with more complex algorithms. It should be noted that the current numbers are a first optimistic results. As such they define a new baseline for possible next iterations of optimisations, which we should mention that these numbers may be influenced by other factors that were optimized in the porting process; for instance: removal of check whether to calculate safety within some functions.
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.:

<table>
<thead>
<tr>
<th>Instruction (type)</th>
<th>ROOT seq</th>
<th>Vec (noSIMD)</th>
<th>Vec (SSE4)</th>
<th>Vec (AVX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>30%</td>
<td>15%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>4%</td>
<td>0.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V..PD (SIMD instr)</td>
<td>5%</td>
<td>55%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Dynamic instruction count statistics for instructions executed in the actual algorithm of Figure 1. Shown are what fractions of the total instruction number are taken by simple memory moves (MOV), call instructions (CALL), SIMD instructions (ALL SIMD) as well as arithmetic SIMD instructions (ARITHM SIMD). These numbers are obtained for 16 particles and a comparison is done between the four algorithmic versions mentioned in the text. Some absolute numbers are given in [brackets] (with an arbitrary but consistent scale).

Figure 3. Results from the benchmark comparing the scalar and sequential algorithm with a vector-oriented algorithm and various degrees of usage of SIMD instructions (from scalar fallback to AVX). (a) Comparison of the runtime per particle showing a speedup factor of roughly 3 comparing the original version to the AVX code. (b) Comparison of actual instructions executed per particle.

We can track the origin of the gains somewhat in analyzing the dynamic instruction mix actually executed in the benchmark. Some important numbers obtained from this analysis are summarized in Table 1. Essentially, it can be seen that going from a scalar to vector interface allows to reduce the number of function call instructions accompanied by a massive reduction in simple memory moves (such as to save registers on the stack). Further, when introducing SIMD optimisations with VC, the overall number of instructions further shrinks and the CPU vector unit is used to a much higher degree. Using the hardware performance counters, we have also confirmed that the number of instruction cache misses is considerably reduced due to better code locality of using the vectorised interfaces. We expect this effect to become even more important with more complex algorithms. It should be noted that the current numbers are a first optimistic results. As such they define a new baseline for possible next iterations of optimisations, which we should mention that these numbers may be influenced by other factors that were optimized in the porting process; for instance: removal of check whether to calculate safety within some functions.
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

<table>
<thead>
<tr>
<th></th>
<th>16 particles</th>
<th>1024 particles</th>
<th>SIMD MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel IvyBridge (AVX)</td>
<td>~2.8x</td>
<td>~4.0x</td>
<td>4x</td>
</tr>
<tr>
<td>Intel Haswell (AVX2)</td>
<td>~3.0x</td>
<td>~5.0x</td>
<td>4x</td>
</tr>
<tr>
<td>Intel Xeon-Phi (AVX512)</td>
<td>~4.1x</td>
<td>~4.8x</td>
<td>8x</td>
</tr>
</tbody>
</table>

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 extent)
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 ...

```cpp
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 ...

```cpp
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 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

(aalmost) as usual
A simple example for the generic approach

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

```
#include <cmath>

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

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

easy to maintain for different applications

Sandro Wenzel, CERN-PH-SFT 5th International Workshop on Future Challenges in Tracking and Trigger
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
**Performance**

- optimized many particle treatment

**Goals**

**Approach**

- SIMD
- template techniques
  - 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

template techniques
- template class specialization/code generation

algo + class review

generic programming

Implementation

Vc library

Cilk Plus

auto-vectorization

Boost::SIMD

Sandro Wenzel, CERN-PH-SFT

5th International Workshop on Future Challenges in Tracking and Trigger
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