Towards a high performance geometry library for particle-detector simulations

J Apostolakis[1], M Bandieramonte[2], G Bitzes[1], R Brun[1], P Canal[3], F Carminati[1], G Cosmo[1], J C De Fine Licht[1], L Duhem[4], V D Elvira[3], A Gheata[1], S Y Jun[3], G Lima[3], T Nikitina[1], M Novak[1], R Sehgal[5], O Shadura[6] and S Wenzel[1†]

[1]European Organization for Nuclear Research (CERN) - Geneva, Switzerland
[2]University of Catania and INAF (IT)
[3]Fermi National Accelerator Laboratory (US)
[4]Intel Corporation
[5]Bhabha Atomic Research Center (IN)
[6]National Technical University of Ukraine, Kyiv Politechnic Institute
[†]corresponding author

E-mail: sandro.wenzel@cern.ch

Abstract. Thread-parallelisation and single-instruction multiple data (SIMD) "vectorisation" of software components in HEP computing has become a necessity to fully benefit from current and future computing hardware. In this context, the Geant-Vector/GPU simulation project aims to re-engineer current software for the simulation of the passage of particles through detectors in order to increase the overall event throughput. As one of the core modules in this area, the geometry library plays a central role and vectorising its algorithms will be one of the cornerstones towards achieving good CPU performance. Here, we report on the progress made in vectorising the shape primitives, as well as in applying new C++ template based optimisations of existing code available in the Geant4, ROOT or USolids geometry libraries. We will focus on a presentation of our software development approach that aims to provide optimised code for all use cases of the library (e.g., single particle and many-particle APIs) and to support different architectures (CPU and GPU) while keeping the code base small, manageable and maintainable. We report on a generic and templated C++ geometry library as a continuation of the AIDA USolids project. The experience gained with these developments will be beneficial to other parts of the simulation software, such as for the optimisation of the physics library, and possibly to other parts of the experiment software stack, such as reconstruction and analysis.

1. Introduction to VecGeom

Targeting the efficient use of vector- and in general SIMD microparallelism is becoming a necessity to fully exploit the capabilities of current and future computing hardware. This is mainly due to SIMD (single instructions multiple data) vector units becoming wider while the CPU clock rate is stalling at a more or less constant value. Unfortunately, only few existing software packages can directly benefit from this trend in hardware development (e.g., via a simple recompile statement) because in most cases the data is not organised in memory to be able to benefit from SIMD vector units, or worse there is no "vector" data to operate on in the first place. Fortunately, in particle-detector simulation we are in the first situation as there is

an inherent "event-level" and "track-level" parallelism thanks to which we can group data in "data vectors". However, this requires a very substantial software development effort to recast existing simulation packages into a vector-friendly form. The Geant-Vector project (Geant-V) took up this challenge two years ago [1, 2]. The key idea of this R&D project is to group data from different events into vector data containers (named baskets) according to some locality criteria. One of the natural criteria is geometric locality – particles in the same (logical) volume are grouped into baskets, but other criteria might be applied as described in [3].

The concept of baskets can be seen as one of the necessary steps in simulation to benefit from vectorisation. A second, equally important, condition is to extend the interfaces and functionality of the major components of the simulation framework to provide an efficient basket processing. First of all, this implies the provision of new vector interfaces or API to the core functionality. Furthermore, it means the use of "horizontal" SIMD vectorisation (i.e., vectorisation over different particles) when processing the baskets.

One of the principal components of particle-detector simulation is the geometry modeler library. It provides the basic shape primitives used as building blocks to describe a detector and it offers basic geometry "ray-tracing" functionality required for the propagation of particles throughout the detector geometry model. The most important geometry tasks are a) collision detection between particles (rays) and shape primitives, b) determination if particles are inside or outside shapes, c) calculation of minimal ("safe") distances to shape primitives and d) the determination of the distance to the exit point when leaving a shape primitive. Currently, a majority of software stacks in HEP uses one of the two wide spread libraries in form of the Geant4 [4] and ROOT(TGeo) [5] geometry modelers. Recently, the AIDA USolids project was started to converge to a single code base for the geometrical primitives in order to achieve better long-term maintainability of geometry algorithms and to foster further development of new code under a common umbrella to the benefit of both Geant4 and ROOT [6].

The present work, however, was motivated from the fact that none of these packages was designed with the requirement to process baskets, nor do any of those packages support compiling and executing geometry algorithms on a GPU. Hence, we are in a situation where those new features have to be developed almost from scratch. While this implies a major software development effort, it also brings with it the chance to modernise existing code, for example, by taking advantage of template features in C++ to increase performance and to write more generic and reusable code.

After our initial SIMD prototyping studies presented at CHEP2013 [7], our development called VecGeom (for vectorised geometry) is now merged with the AIDA USolids project [6] and seen as its evolution, as far as the geometric primitive code is concerned. Furthermore, VecGeom also offers geometry modeler components to actually describe hierarchical detectors and to perform scalar and vector navigation in these detectors. These parts go beyond the initial scope of the USolids project.

The following sections shortly report on the current progress of VecGeom starting with an overview of our software development pattern and a report of the implementation- and performance status thereafter.

2. Software engineering challenges

There are two major challenges to overcome in this project which are related to achieving good performance as well as good code maintainability. The first is the problem of how to achieve a reliable (multi-particle) SIMD vectorisation of our algorithms. In the past this often required platform specific code using assembly language or compiler intrinsics. However, in recent years this challenge is actually becoming a rather manageable task thanks to the availability of new high-level C++ (templated) wrapper libraries which hide away the complexity of SIMD hardware-instructions by providing user-friendly C++ types and operators [8–10].

Using such libraries of wrapper types, writing vectorised code is *often* no different to writing scalar code apart from using slightly different variable types (apart from some different treatment of conditional execution etc.). This is in particular true for multi-particle vectorisation for which we currently employ the Vc library [9], chosen for it's ease of use and the very good performance obtained in our initial vectorisation study [7].

The second challenge is related to software maintenance due to potential code duplication. Our two primary goals of providing new multi-particle vectorised geometry algorithms, as well as porting the geometry code to CUDA (or other GPU languages), in addition to the existing scalar code, could potentially lead to a code multiplication problem. This brings the danger of a significantly increased maintenance effort in the long term. To illustrate this, note that a brute force approach of just extending the code base in addition to already existing scalar code would imply having roughly 200 new functions to maintain for the approximately 20 supported primitive solids (box, sphere, tube,...).

Clearly, we do not want to follow this brute-force path. Rather, we make particularly extensive use of the generic programming pattern, possible in C++ with template classes and template functions. These template functions are then used to compile into specialised binary code for the scalar interface, the multi-particle interface, or the GPU kernel, as illustrated in Fig. 1. We refer the reader to our git repository for real code examples [11].

This approach is motivated by the following arguments:

- Scalar-particle and multi-particle code is very similar in many cases. A generic description of the algorithms in form of C++ template functions, using scalar or vector types as template parameters, is therefore able to describe both the scalar and multi-particle algorithms of the geometry library.
- Whenever the algorithms are not similar, either the explicit template specialisation approach, or the use of C++ traits and policy techniques [12, 13] which permit the specialisation of certain code regions, can be used.
- GPGPU computing follows the SIMD paradigm, so whenever we have developed a multi-particle algorithm, this should be the basis for a well performing GPU kernel.
- The CUDA nvcc compiler is now fully C++11 compliant having no problems to understand templates and is able to compile standard C++ code into GPU device kernels.[1]

3. Implementation status, new features and performance examples

3.1. Primitive solid types

Following our initial vectorisation demonstrator that started mid 2013, our program of work consisted in refactoring the core geometry algorithms for the solid primitives using generic templated algorithms and we based our development mostly on the algorithms available in USolids [6].

This process involves a major development effort but the benefits are already clearly visible. Currently, we have achieved generic implementations for the following shape primitives used in detector construction: box, sphere, torus, tube(segments), cone(segments), trapezoids, symmetric trapezoids (trds), parallelepiped, paraboloid, hyperboloid. In all these cases, we are able to generate well performing SIMD kernels for the treatment of baskets and in all the above mentioned cases the same generic code is used for the scalar API with improved or equivalent performance compared to the original USolids code. Moreover, the same shape-primitives are now trivially available for detector construction on the GPU.

[1] The situation with OpenCL is not as good but more advanced standard and compilers (for example SYCL) are coming up which hopefully will make OpenCL accessible to us without the need to develop separate OpenCL code.

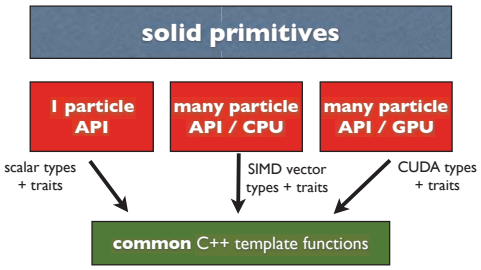

Figure 1. Diagram illustrating our software development approach. Templated C++ functions are written using abstract types and abstracted properties (traits). The different API then instantiate the template functions with concrete types and properties and the compiler produces different binary code in each case. In case of the multi-particle API, we currently inject SIMD vector types coming from the Vc library.

We should emphasise that next to these benefits, the process of refactoring the algorithms also has given us the opportunity to introduce some other optimisation or new constructs to the geometry library. Notably, these are:

- Improved algorithms, in particular via the elimination of many trigonometric functions often found in Geant4/ROOT algorithms.
- The use of template class specialisation to create specialised geometry objects from generic templates. Template shape specialisation is particularly useful for solid classes that can assume many different realisations. Traditionally, for example, the tube and cone shapes are each modeled by one C++ class but they can come in many different variations: hollow tubes and cones, full tubes and cones or hollow tubes/cones with a cut. This implies the presence of many conditional branches in the code. Template shapes specialisation helps avoiding these branches at runtime while keeping just one C++ implementation for all shape variations.

Taking all points together, our development effort has allowed us to maintain roughly the current code size in terms of the number of lines of code, to provide SIMD vectorised (or "GPUised") algorithms for multi-particle treatment *and* to increase the performance of the traditional scalar algorithms at the same time. This is for instance demonstrated in Fig. 2 which shows benchmark data for the case of the tube-segment shape which is one of the most important shapes in detector construction. From the point of view of the Geant-V project, the most important number is the speedup (generally between 2 and 3) due to the SIMD vectorised treatment of baskets which is promising. We see similar SIMD improvements for all our currently implemented shapes. Note that detailed benchmarks on the GPU are not yet included in this plot as this work is still in progress.

3.2. Other geometry components

Beyond generic implementations at the shape level, VecGeom also includes a first portable implementation of a geometry navigator which is able to process single particles as well as baskets of particles on CPU and GPU. As a foundation layer to this, we provide an in memory representation of the detector which is synchronisable (copyable) between CPU and GPU. As a result, VecGeom can today serve as a minimalistic[2] replacement of the complete Geant4 or ROOT/TGeo geometry modellers. Indeed, we are able to do full particle-simulations in the

[2] In the sense that we do yet provide sophisticated navigation algorithms making use of spatial optimisations often used in computer graphics and game engines.

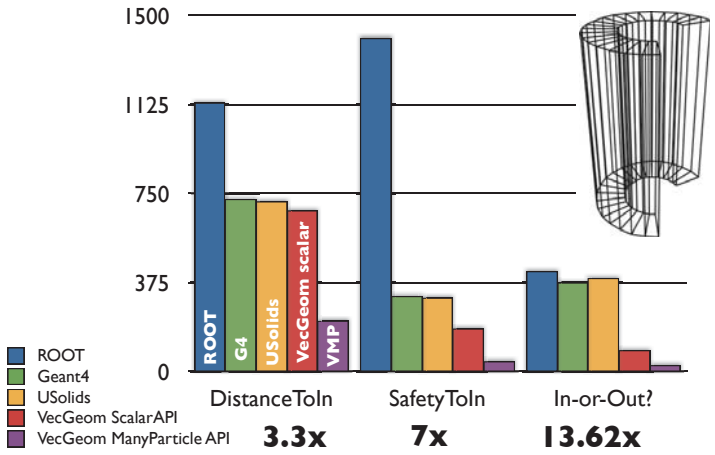

Figure 2. Performance numbers (CPU time in arbitrary units) for a tube-segment shape for three of the most essential algorithms from the Geant4, ROOT, USolids and the new generic VecGeom implementations. The two important features here are improved scalar performances of VecGeom over USolids (from better algorithms and template specialisation techniques) as well as excellent SIMD vectorisation results for the multi-particle treatment. The bold numbers indicate the total speedup as compared to the current implementation of the USolids library. The benchmark was performed using 1000 particles; We used gcc-4.7, ROOT5.34.18 and Geant4-10p01 in their release builds. The benchmark machine is a Intel(R) Core(TM) i7-3770 CPU@3.40GHz using the AVX instruction set.

Geant-V prototype either using TGeo or VecGeom. In a very simple detector simulation (based on the original Geant4 example N03, using only boxes), the total runtime improvement when using VecGeom instead of TGeo was measured to be roughly 40% (see [3] for more information). This is a first very promising number which is mainly due to a SIMD treatment of baskets in the geometry part of the simulation. This implies that together with a future vectorised treatment of the physics routines we hope to achieve considerable better overall performance gains. Of course, it is not possible to extrapolate this first result to realistic detector setups but we shall be able to provide those comparisons, together with the ability to represent more complex geometries, in the near future.

3.3. Ongoing work and outlook

We are moving towards generic implementations of the more complicated shape primitives often used in HEP experiments. These are notably the polyhedra and polycone shapes. Due to their complex nature it might not be evident nor possible to achieve good SIMD vectorisation for the many-particle API. In this case, a good strategy seems to be to apply internal vectorisation to speed up the original algorithms due to the presence of internal loops over regular structures in those shapes. We have already achieved promising results for the polyhedra and generic trapezoid shapes.

Another area addresses complex shapes in form of boolean solids which follow the constructive solid geometry (CSG) approach to build more complicated solids out of primitive constituents. Here we made first prototypes replacing the current virtual function based CSG approach with a templated (policy like) approach allowing the compiler to perform better code optimisation for particular boolean solid instances. Concrete results will be presented elsewhere.

4. Discussion and Summary

The present paper describes our effort to improve the performance of geometry libraries in detector simulations and to enable them for the new use cases in the context of multi-particle processing on CPU and GPU. We have made considerable progress towards this goal and we see that all our new implementations are currently faster (or equally fast) compared to all previous implementations, that they vectorise to the expected level and that they all use the same generic code implementation. We believe that a complete fully vectorised and otherwise optimised geometry library can be achieved soon.

One of the remaining challenges is how the vectorised solid algorithms can be used in the context of very complex detectors where optimised spatial tree-structures (or voxelisation) are used in the navigator.

Beyond the development in detector geometries, the present project has allowed us to gain invaluable experience in vectorising algorithms as well as in addressing software and portability issues, that will – without a doubt – help to tackle other developments in the Geant-V project and elsewhere.

References

[1] Apostolakis J, Brun R, Carminati F and Gheata A 2012 *Journal of Physics: Conference Series* **396** 022014

[2] Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2014 *Journal of Physics Conference Series* **523** 012004

[3] Apostolakis J, *et al.* Adaptative track scheduling to optimize concurrency and vectorization in geantv (ACAT2014; this proceedings)

[4] Allison J, *et al.* 2006 *Nuclear Science, IEEE Transactions on* **53** 270–278 ISSN 0018-9499

[5] Gheata A URL ftp://root.cern.ch/root/doc/18Geometry.pdf

[6] Gayer M, Apostolakis J, Cosmo G, Gheata A, Guyader J M and Nikitina T 2012 *Journal of Physics: Conference Series* **396** 052035

[7] Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2014 *Journal of Physics Conference Series* **513** 052038

[8] Esterie P, Gaunard M, Falcou J, Lapresté J T and Rozoy B 2012 Boost.simd: generic programming for portable simdization *PACT* ed Yew P C, Cho S, DeRose L and Lilja D J (ACM) p 431 ISBN 978-1-4503-1182-3

[9] Kretz M and Lindenstruth V 2012 *Software: Practice and Experience* **42** 1409

[10] Fog A Vcl, c++ vector class library URL http://www.agner.org/optimize/vectorclass.pdf

[11] URL http://git.cern.ch/pub/VecGeom

[12] Myers N June 1995 "a new and useful template technique: Traits"

[13] Alexandrescu A 2001 *Modern C++ Design* (Addison-Wesley)