Accelerating navigation in the VecGeom geometry modeller

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 072032 doi:10.1088/1742-6596/898/7/072032

Accelerating navigation in the VecGeom geometry modeller

Sandro Wenzel[1*] **and Yang Zhang**[2+] ***for the VecGeom Developers***

[1] CERN, Route de Meyrin, 1211 Geneva, Switzerland

[2] Department of Informatics, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany

E-mail: [*]sandro.wenzel@cern.ch; [+]yang.zhang@student.kit.edu

Abstract. The VecGeom geometry library is a relatively recent effort aiming to provide a modern and high performance geometry service for particle detector simulation in hierarchical detector geometries common to HEP experiments. One of its principal targets is the efficient use of vector SIMD hardware instructions to accelerate geometry calculations for single track as well as multi-track queries.

Previously, excellent performance improvements compared to Geant4/ROOT could be reported for elementary geometry algorithms at the level of single shape queries. In this contribution, we will focus on the higher level navigation algorithms in VecGeom, which are the most important components as seen from the simulation engines. We will first report on our R&D effort and developments to implement SIMD enhanced data structures to speed up the well-known "voxelised" navigation algorithms, ubiquitously used for particle tracing in complex detector modules consisting of many daughter parts.

Second, we will discuss complementary new approaches to improve navigation algorithms in HEP. These ideas are based on a systematic exploitation of static properties of the detector layout as well as automatic code generation and specialisation of the C++ navigator classes. Such specialisations reduce the overhead of generic- or virtual function based algorithms and enhance the effectiveness of the SIMD vector units.

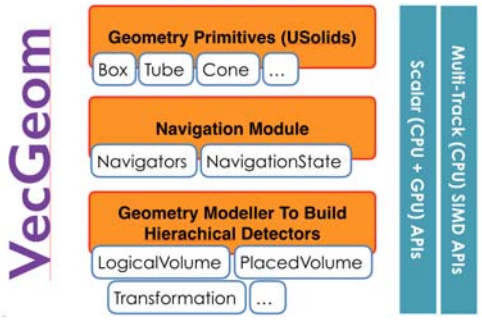
These novel approaches go well beyond the existing solutions available in Geant4 or TGeo/ROOT, achieve a significantly superior performance, and might be of interest for a wide range of simulation backends (GeantV, Geant4). We exemplify this with concrete benchmarks for the CMS and ALICE detectors.

1. Introduction

Being able to transport particles in a complex three-dimensional world is one of the cornerstones of particle detector simulation frameworks. Much like in many gaming or ray tracing applications, the geometry engine is responsible for calculating various distances between the transported particle and detector components as well as to provide collision detection algorithms.

VecGeom [1] is a C++ geometry modeller for particle detector simulation frameworks which was developed from scratch with the target to modernise and optimise geometry routines, in particular by making use of single instruction multiple data (SIMD) vectorisation. It is based on a model of hierarchies of (CGS) volumes following the logic of what is or was available in Geant3, Geant4 [2, 3] and TGeo/ROOT [4, 5].

VecGeom's development first started as a component of the GeantV simulation project (see, e.g., Refs. [6–8]) to satisfy the need for a many-track API needed by the GeantV engine and



(1) transform global particle coordinates to local system of scene
(2) determine distance leaving this volume (scene)
(3) query for set of candidate daughter volumes to probe for intersection
(4) test candidate set for intersection distances
(5) optionally relocate across boundaries

Figure 1: (left) Component overview of VecGeom. (right) Main algorithmic steps in navigation

which was not offered by any existing solution. Today, VecGeom is a standalone library which is used by GeantV, but with the capability to serve simulations using the Geant4 engine or the TGeo/ROOT framework. Indeed, VecGeom is a consequential evolution of the Unified Solids (USolids) [9] project on the shape algorithms, which was started to foster more common and modular code shared between Geant4 and TGeo. Today, the USolids and VecGeom effort have joined forces and constitute a common project. Figure 1 (left) gives a rough overview over the principal components of VecGeom, which are: (a) algorithms for geometric primitives, (b) higher level navigation components doing hit detection and querying the location of tracks in the detector as well as (c) structures to model and describe the layout of a detector.

A lot of the effort of the VecGeom project has so far gone into development of algorithms for basic geometric entities within component (a) in order to enable multi-particle operations using SIMD processing or to improve on existing code. This work has been presented previously [10, 11] and continues to be improved and extended constantly.

The present proceeding focuses on our recent developments done for the navigation components (b). In section 2, we will describe the work undertaken to achieve scalable and SIMD enabled collision detection and location queries. In section 3, complementary ideas are presented for speeding up navigation algorithms, based on the idea of complete code specialisation.

2. SIMD accelerated navigation

2.1. General navigation algorithms

The purpose of the navigation module is to provide algorithms to locate particles in hierarchical detectors and to trace them in a scene of geometric objects. A scene can be thought of as the set of objects which can be seen by a particle at any moment. A detector consists of many different modules or scenes and whenever a particles crosses a material boundary it enters into a different scene. Using the terms of TGeo or Geant4 geometry, a scene is associated with a logical volume. A logical volume has a material, an associated geometric primitive describing its boundary as well as daughter volumes making up its content.

Using such definitions, the main tasks of a navigation module are

(i) To determine in which scene a particle is, given its global coordinates
(ii) To tell which object will be hit next, given the ray (i.e., the position and the current straight line direction) of a particle
(iii) Related to (i) and (ii): To determine the next scene after crossing the next boundary

In this work we will concentrate mostly on task (ii) and show how it can be accelerated using SIMD techniques. The typical algorithmic pipeline for this task is shown in figure 1 (right). We will focus here on a description to speed up tracing single particles in a complex geometry which is applicable to both GeantV and Geant4.

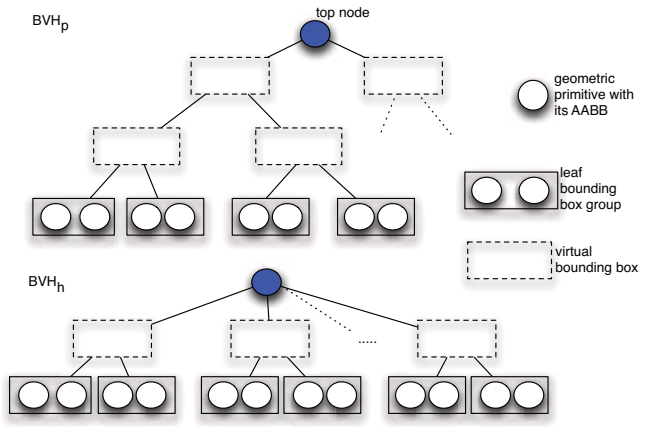


Figure 2: Data structures used for SIMD filtering of hit candidates. A (part of a) regular and balanced bounding box tree structure is shown in the upper half while a tree with a fixed depth but variable number of first-level children is shown at the bottom. Both bounding volume hierarchies contain leaf nodes that group aligned bounding boxes (solid boxes) of objects (circles) in a scene. Leaf nodes are grouped into bigger boxes (dashed boxes) by using clustering.

2.2. Introduction to SIMD-tree traversal

The simplest hit detection algorithms, just iterating over the complete list of objects and testing each in turn, have linear running time with respect to the number of objects in a scene and are hence not computationally efficient. Via the use of helper spatial data structures, this linear time bottleneck can be overcome (see, e.g., Ref. [12] for an overview). The accelerating helper structure can be queried quickly and commonly in logarithmic time for a set of candidate objects which are subsequently tested for intersection. Both Geant4 and TGeo implement such "voxelisation" techniques but neither of them uses CPU SIMD instruction sets as an acceleration.

Our goal of speeding up the filtering of the candidate set using vector instructions, puts a couple of constraints on the spatial data structure. Notably, one may argue that traversal of completely irregularly shaped hierarchies is not well suited due to a lot of branching and bad load balancing. To gain most from SIMD, all vector lanes should be kept busy at a maximum of times. On the other hand, well-balanced and regularly formed tree structures, in which during traversal each node query is doing the same work on all vector lanes, could be well suited for SIMD acceleration. Regular trees (or hierarchies) of bounding volumes (BVH) (see Ref. [12]) seem to be a natural choice for this: By virtue of vectorisation instructions we are able to quickly calculate the intersection between a ray and a fixed number of bounding boxes during tree traversal and hence decide quickly with which tree branch to proceed. In other words, the idea is to gain from SIMD acceleration by calculating the distance to a bunch of bounding boxes in parallel, whereas a good overall scaling property comes from a hierarchical organisation of bounding boxes enclosed by larger bounding boxes. This idea has also been described in the ray tracing literature in various contexts and variants [13–15].

2.3. Concrete data structure approach

Without loss of generality, we have concentrated our efforts on treating axis-aligned bounding boxes (AABBs) as the basic building blocks. The key computational kernel is hence `ray.computeDistance(group_of_AABBs)` [16] which is easily vectorised for multiple boxes using platform independent vectorisation libraries such as Vc [17] or our higher level abstraction called VecCore [18]. AABBs have the advantage to avoid expensive coordinate transformations and floating point divisions. However, in certain circumstances, more tightly placed (arbitrarily rotated) bounding boxes might be of advantage. The above kernel can easily be generalised to this situation.

Starting from the basic geometry primitives in a scene, we first calculate the AABB of each object. The resulting boxes then serve as input for constructing hierarchical bounding box structures. In the course of this work, we have notably played with two concrete structures, as

displayed in figure 2, which we call pure and hybrid BVH, respectively. The first, BVH_p, is a regular tree structure with fixed number of children per node and arbitrary depth whereas the second is a tree of fixed depth but arbitrary children for the root node. As this is some sort of hybrid between a tree and a flat list, we call it BVH_h.

The idea in both cases is to cluster the leaf AABBs into bigger bounding boxes by a proximity criterion. This process is repeated iteratively resulting in a tree of bounding volumes, taking into account the constraint we impose on the overall structure. Possible mechanisms to achieve this are described further below. The vector length S of the SIMD registers (for example $S = 4$ for `float` on SSE4.2 and $S = 8$ for AVX) influences directly the tree structure. In case of BVH_p, the number of children for each node in the tree is (in principle) exactly S, thus leading to quadtrees, octrees, etc. depending on the SIMD architecture. The depth of the tree is determined by the number of objects. The benefit of BVH_p is that it scales very well leading to the least possible number of collision tests (on average). This structure is best for geometries with very many objects (such as millions of triangles in ray tracing).

On the other hand, BVH_p is a rather non-local data structure with many indirections and might be outperformed by other (more cache-friendly) structures in case of fewer number of objects. In HEP, typically not more than O(1000) objects – and often much less – are present in a scene which motivated us to also look into more data-local approaches. The hybrid hierarchy is based on this idea. Here, the depth of the tree is fixed to two, which allows us to map all the tree nodes into a consecutive array without relying on additional data structures for navigation. This memory layout is very cache-friendly, while retaining some form of hierarchical scaling benefit.

Depending on the complexity of the scene, VecGeom can decide to use either BVH_p or BVH_h (or just a linear search). In our experience so far, BVH_h almost always performed better than BVH_p for the use cases encountered.

2.4. BVH building techniques

We now discuss two possible algorithms to perform the tree building. In the top-down implementation of both BVHs we face the problem of splitting the set of volumes belonging to the current node into as many sub-clusters as there are child nodes, such that each group will be assigned to a new child node. We used two approaches to carry out clustering, one of them is a variation of the k-means algorithm. The algorithm has one parameter k which is the number of clusters. For BVH_p we cluster the volumes of the current node each time a node gets extended, where $\text{k} = S$. For BVH_h we only need to cluster once on all volumes of the geometry, where $\text{k} = \lceil \#\text{allvolumes}/S \rceil$. Given N volumes to cluster, both algorithms guarantee that in the end at least $\text{k} - 1$ clusters are filled up to their maximum capacity, i.e. having $\kappa = \lceil N/\text{k} \rceil$ elements each.

2.4.1. Clustering: k-means variation First, we carry out the classic k-means algorithm on the set of volumes with positions $V = \{v_1, \ldots, v_N\}$, where we initialise the cluster centers with random volume positions. Let's denote the final clusters with $C = \{c_1, \ldots, c_k\}$ and the cluster centers with $M = \{m_1, \ldots, m_k\}$. After the clustering step, we conduct an equalisation on the clusters to ensure $\text{k} - 1$ clusters have κ elements each. To this end, we first sort the clusters by descending cluster size and then iterate over these. As long as there is a cluster with more volumes than it can hold, we pick the volume furthest away from the cluster center and move it to the closest cluster which has an available spot.

2.4.2. Clustering: around furthest volume This algorithm aims to maximise the separation and minimise the intersection of child bounding boxes by clustering the furthest volumes first. From the set of remaining volumes that have not been assigned to any cluster yet, we choose

Algorithm 1 Cluster Equalisation

Input: $\mathbf{C}, \mathbf{M}, \kappa$
Output: equalised clusters $\mathbf{C}$
$\mathbf{k} := |\mathbf{C}|$
for $i := 0$ to $\mathbf{k}$ **do**
  sort($\mathbf{C}$) by decreasing size
  while $|\mathbf{c}_i| > \kappa$ **do**
    $\mathbf{c}_i^* \longleftarrow$ furthest element in $\mathbf{c}_i$ to $\mathbf{m}_i$
    $l \longleftarrow \underset{z}{\text{argmin}} \|\mathbf{m}_z - \mathbf{c}_i^*\|$ where $|\mathbf{c}_z| < \kappa$
    move($\mathbf{c}_i^*, \mathbf{c}_i, \mathbf{c}_l$) // move $\mathbf{c}_i^*$ from cluster i to l
  end while
  updateCenters($\mathbf{C}$) // recalculate cluster centers
end for

Algorithm 2 Cluster around furthest volume

Input: $\mathbf{V}, \kappa, \mathbf{k}$
Output: equalised clusters $\mathbf{C}$
  initialise $\mathbf{C}$ with $\mathbf{k}$ empty sets
$N := |\mathbf{V}|$
  $\text{mean}_r = \sum_{i=1}^{N} \frac{\mathbf{v}_i}{N}$ // mean of unassigned volumes
for $i := 0$ to $\mathbf{k}$ **do**
  $\text{mean}_c := 0$ // mean of i-th cluster
  while $|\mathbf{c}_i| < \kappa$ **do**
    if $|\mathbf{c}_i| == 0$ **then**
      $\text{index} = \underset{j}{\text{argmax}} \|\mathbf{v}_j - \text{mean}_r\|$ // index of furthest volume from remaining volumes
    else
      $\text{index} = \underset{j}{\text{argmin}} \|\mathbf{v}_j - \text{mean}_c\|$ // index of closest volume to the cluster center
    end if
    $\mathbf{V} \longleftarrow \mathbf{V} \backslash \mathbf{v}_{index}$
    updateMean($\mathbf{V}$, mean_r)
    $\mathbf{c}_i \longleftarrow \mathbf{c}_i \cup \mathbf{v}_{index}$
    updateMean($\mathbf{c}_i$, mean_c)
  end while
end for

the furthest one from the mean center of these volumes and create a new cluster at its position. The new cluster is then filled up to its maximum capacity with the closest volumes. After each addition the cluster center will be updated to the mean of its current elements.

2.5. Evaluation

In order to evaluate our implementation, we have picked a couple of complex and representative detector volumes (scenes) from the ALICE and CMS detector descriptions. Within each of those volumes, we have generated half a million random rays. For these rays we then perform the navigation pipeline of figure 1 (right) corresponding to navigation task iii and measure the time to process all rays. This task is done using both the Geant4 (`10.2.1`) and TGeo (`ROOT v6.06.8`) engines as well as using VecGeom (VG), using BVH_h, for two different ISA (SSE4.2 and AVX2). The evaluation has been performed with one thread on an otherwise idle `Intel(R)-Core(TM) i7-5930K` running `CERN CentOS7`. The compiler was `gcc4.8.5`. VecGeom tag `W40-16` was compiled in its release mode. Table 1 shows the resulting timings and clearly demonstrates that SIMD enabled navigation performs consistently faster than Geant4 and TGeo. Moreover, one can clearly observe the benefit coming from the SIMD treatment because a clear timing improvement is noticed by increasing the SIMD vector from 4 elements (SSE4.2) to 8 (AVX2). Note that it cannot be expected to achieve perfect scaling with the vector width as the SIMD-tree

Table 1: Timings (in seconds) to process all test rays for a list of complex detector volumes. The worst timing is shown in red while the best in blue. VecGeom's SIMD enabled navigation performs consistently better than any existing solutions.

Volume	#daughters	Geant4	TGeo	VG (SSE4.2)	VG (AVX2)
ALIC (ALICE)	65	0.74	1.07	0.30	0.23
TPC_Drift (ALICE)	641	14	2.2	1.2	0.9
MBWheel_1N (CMS)	789	0.84	1.09	0.49	0.35

query only represents one algorithmic part of the problem.

3. Specialised navigators through code generation

So far, we have focused on SIMD acceleration of the navigation pipeline. In this section, we change directions and address some completely orthogonal ideas to speed up the navigation algorithms. We would like to discuss the question "What other ways exist to make the navigation system in typical HEP applications faster?".

Working on this question is motivated by at least two goals: (a) Minimising the overall runtime of treating particles in the geometry module and (b) Maximising the number of CPU cycles that can be treated using the SIMD paradigm. The latter is equivalent to reducing algorithmic parts which are hard to vectorise (for example because of many branches).

The basic observation that we try to exploit here is the fact that HEP detectors are pretty *static* objects. Often they don't change much during a long period of simulation runs or data taking. Hence, the tracing of particles is repeated over and over again in almost identical geometric setups. In addition, in many applications only a few geometry scenes are dominating the CPU budget.

In this context, wouldn't it make sense to construct highly-optimised navigation algorithms for each of the dominating scenes? By design, VecGeom already provides this possibility because it allows to attach specialised navigator instances – implementing well-defined navigation interfaces – to logical volumes/scenes.

By "highly-optimised" we mean exploiting static information inside the algorithm itself. On the one hand, such specific information could, in some cases, be used within generic algorithms via some form of tabulation. On the other extreme, the information could be hard-coded into a specialised algorithm directly. Some trade-off might finally be best suited. For now we have opted to follow the direction of complete code specialisation in order to investigate what is possible and how much performance one can ultimately gain by giving the compiler as much information as possible.

We have identified at least the following areas where static information can be exploited:

Fast touchable to index hashing At any moment in time, particles have a concrete location in the detector. As the detector in VecGeom is described as a tree structure of placed volumes, the local reference frame of the particle with respect to the global detector frame is determined by a unique path/branch on the geometry tree (see "touchable" concept in Geant4). This path information is carried around in the state of the simulated particle. The path is essentially a *variable* length list of indices and it determines the global-local coordinate transformations which are needed during navigation. It is highly desirable to quickly lookup these transformations based on the path (instead of recalculating it frequently). The most efficient lookup would be to calculate a linear index, given a path, and fetch the transformation from a contiguous vector. For the most important logical volumes, a static analysis will be able to produce optimal path to index conversion lookup

structures. In particular, static analysis will do a dimensional reduction of the path state space and will figure out the relevant indices that influence the mapping.

Exploiting transformation structure Another optimisation for coordinate transformation is the exploitation of the structure of the transformation matrices. For example, static analysis will be able to tell if some coefficients are zero or one. Such information can directly be put into the code improving the floating point throughput.

Reduction of virtual functions Reducing virtual function calls will enable the compiler to perform additional optimisation. Static analysis of a scene is able to tell which exact geometric primitives (boxes, tubes, cones) are being queried during collision detection. This can directly be encoded into the C++ sources instead of letting the runtime figure this out through dynamic polymorphism. (Note that this is usually not an issue for ray tracing platforms which mostly only treat triangles anyway).

Optimised relocation When a particle moves across a material boundary it will obtain a new path state which needs to be calculated. Although this calculation can benefit from SIMD acceleration similar to section 2, it is still quite expensive. Static information such as which volumes share boundaries with which other volume would definitely help constraining the search space needing to be traversed and are hence a valuable optimisation. This technique is known to be used in other packages (e.g., DAGMC [19]) and is more straightforward to apply when the geometry is based on flat surface models coming from CAD systems. In our case, determining touching properties is relatively complicated and needs sampling techniques.

It is clear that performing the code specialisation related to the above ideas cannot be done by hand. We have hence developed a prototypic tool, which – given a detector description (in form of a TGeo ROOT file) and the logical volume name – emits an optimised navigator algorithm in C++. Our tool implements all of the aforementioned ideas in a first form. The resulting code can be compiled into a shared library that can be loaded as a plugin to the simulation. It is even conceivable that during a running simulation itself, important logical volumes are detected, specialised code constructed, compiled and hooked back into the simulation. However, more often than doing this just-in-time compilation, users might do a separate short simulation, construct the optimised code and use it in longer production runs. Table 2 presents a few preliminary numbers from a first evaluation of the tool. We use the same procedure as in section 2.5, but focus on ALICE detector volumes which were measured to be very important in terms of the number of steps done within them (from typical Pb-Pb collisions). Moreover, these volumes are rather simple and they do not contain lots of daughter objects. It is expected in these cases, that specialisation would give us the best improvements, although many other parameters (hierarchical depth of volume, complexity of scene in neighbouring volumes) have

Table 2: Navigation timing measurements for ALICE detector volumes, known to be important CPU consumers. Timings represent seconds to process 0.5M tracks as in section 2.5. Next to the times obtained with Geant4 and TGeo, VecGeom timings are shown for both the normal as well as the code-specialised navigator. The additional benefit of code specialisation (factor VGnorm/VGspec) is indicated in square brackets.

VolumeName	#daughters	Geant4	TGeo	VG norm	VG spec
ZNST	4	0.24	0.28	0.10	0.06 [**x1.6**]
voRB243CuTube	0	0.16	0.24	0.10	0.06 [**x1.6**]
AFaGraphiteCone	1	0.74	0.36	0.11	0.03 [**x3.6**]

a strong influence too. We observe significant accelerations (here between factors ≈ 1.6 and ≈ 3.6) due to code specialisation, and conclude that this technique can be extremely valuable and promising to further accelerate VecGeom and HEP simulation in general. Turning this study into production is from our perspective strongly encouraged, and of course not limited to the above list of ideas.

Note that in the context of GeantV, the technique using specialised navigators already allowed to obtain significant SIMD efficiency when treating baskets of particles in a (toy) simulation [20].

4. Summary and outlook

We presented two orthogonal advances in the VecGeom library related to the navigation component. The first one is related to making use of the SIMD paradigm in hierarchical data structures to quickly query hit candidate objects. It was shown that this leads to clear improvements over existing voxelisation approaches in Geant4 and TGeo. The second idea is related to automatic code specialisation and generation of navigator algorithms based on the idea to exploit static properties of HEP detectors.

It is planned to extend the SIMD BVH techniques to other areas in VecGeom, such as to the tessellated solid or multi-union geometric primitives.

References

[1] The VecGeom repository URL `https://gitlab.cern.ch/VecGeom/VecGeom`
[2] Agostinelli S *et al* 2003 *Nucl. Instr. Meth.* A **506** 250
[3] Allison J *et al* 2016 *Nucl. Instr. Meth.* A **835** 186
[4] Brun R and Rademakers F 1997 *Nucl. Instr. Meth.* A **389** 81
[5] Antcheva I *et al* 2009 *Computer Physics Communications* **180** 2499
[6] Apostolakis J *et al* 2015 *J. Phys.: Conf. Ser.* **608** 012003
[7] Amadio G *et al* 2015 *J. Phys.: Conf. Ser.* **664** 072006
[8] Amadio G *et al* 2016 *J. Phys.: Conf. Ser.* **762** 012019
[9] Gayer M, Apostolakis J, Cosmo G, Gheata A, Guyader J M and Nikitina T 2012 *J. Phys.: Conf. Ser.* **396** 052035
[10] Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2014 *J. Phys.: Conf. Ser.* **513** 052038
[11] Apostolakis J *et al* 2015 *J. Phys.: Conf. Ser.* **608** 012023
[12] Ericson C 2004 *Real-Time Collision Detection* (Boca Raton, FL, USA: CRC Press, Inc.) ISBN 1558607323, 9781558607323
[13] Embree - fast ray tracing kernels URL `https://embree.github.io/`
[14] Wald I, Woop S, Benthin C, Johnson G S and Ernst M 2014 *ACM Trans. Graph.* **33** 143:1–143:8
[15] Dammertz H, Hanika J and Keller A 2008 *Computer Graphics Forum* **27** 1225
[16] Williams A, Barrus S, Morley R K and Shirley P 2005 An efficient and robust ray-box intersection algorithm *ACM SIGGRAPH 2005 Courses* SIGGRAPH '05 (New York, NY, USA: ACM)
[17] Kretz M and Lindenstruth V 2012 *Software: Practice and Experience* **42** 1409–1430
[18] The VecCore repository URL `https://gitlab.cern.ch/VecGeom/VecCore`
[19] DAGMC URL `svalinn.github.io/DAGMC/`
[20] Gheata A, Amadio G, Bianchi C, Carminati F, Vallecorsa S and Wenzel S 2015 Supercomputing 2015 : Intel-Booth poster presentation