First experience with portable high-performance geometry code on GPU

John Apostolakis[1], *Marilena Bandieramonte*[1], *Georgios Bitzes*[1], *Gabriele Cosmo*[1], *Johannes de Fine Licht*[1]*, *Laurent Duhem*[3], *Andrei Gheata*[1], *Guilherme Lima*[2], *Tatiana Nikitina*[1], *Sandro Wenzel*[1]†

[1]CERN, 1211 Geneve 23, Switzerland
[2]Fermilab, Kirk & Pine Rd, Batavia, IL 60510, USA
[3]Intel, 2200 Mission College Blvd., Santa Clara, CA 95054-1549, USA

DOI: http://dx.doi.org/10.3204/DESY-PROC-2014-05/18

The Geant-Vector prototype is an effort to address the demand for increased performance in HEP experiment simulation software. By reorganizing particle transport towards a vector-centric data layout, the projects aims for efficient use of SIMD instructions on modern CPUs, as well as co-processors such as GPUs and Xeon Phi.

The geometry is an important part of particle transport, consuming a significant fraction of processing time during simulation. A next generation geometry library must be adaptable to scalar, vector and accelerator/GPU platforms. To avoid the large potential effort going into developing and maintaining distinct solutions, we seek a single code base that can be utilized on multiple platforms and for various use cases.

We report on our solution: by employing C++ templating techniques we can adapt geometry operations to SIMD processing, thus enabling the development of abstract algorithms that compile to efficient, architecture-specific code. We introduce the concept of modular "backends", separating architecture specific code and geometrical algorithms, thereby allowing distinct implementation and validation. We present the templating, compilation and structural architecture employed to achieve this code generality. Benchmarks of methods for navigation in detector geometry will be shown, demonstrating abstracted algorithms compiled to scalar, SSE, AVX and GPU instructions. By allowing users to choose the most suitable type of processing unit for an application or potentially dispatch work dynamically at runtime, we can greatly increase hardware occupancy without inflating the code base. Finally the advantages and disadvantages of a generalized approach will be discussed.

1 Introduction

Recent evolution of hardware has seen a reawakened focus on vector processing, a concept that had otherwise lain dormant outside the world of video games while CPU clock speeds still rode Moore's law. Most notable is the surge of interest in general purpose GPU programming (GPGPU), but vector instruction sets have existed on convential CPUs since the introduction of *3DNow!* by AMD [3] and *SSE* by Intel [4] in 1998 and 1999, respectively.

*Presenting author (johannes.definelicht@cern.ch).
†Corresponding author (sandro.wenzel@cern.ch).

To improve performance on modern hardware, it is essential to make use of their vector processing capabilities to access memory efficiently, whether this means efficient cache access on CPU or efficient use of memory bandwidth on GPU.

1.1 GeantV

The Geant Vector prototype (GeantV) [1] was started as a feasibility study of the potential for adding vectorization to detector simulation programs, in order to address an increasing need for performance in HEP detector simulation software.

At the core of GeantV there is a scheduling engine orchestrating the progress of the simulation. This scheduler works on particle data packed in *structure of arrays* (SOA) form for cache-efficiency and to accommodate SIMD memory access [7]; particles that require similar treatment are grouped into *baskets*, which can be dispatched to the relevant components for vectorized treatment [5]).

The GeantV R&D effort targets multiple platforms for vectorization, including SIMD instructions on CPUs (SSE, AVX) and accelerators such as GPUs and the Xeon Phi (AVX512). Although sometimes limited by the state of compilers, the project pursues *platform independence* to avoid *vendor lock-in*.

1.2 VecGeom

Navigation and particle tracking in detector geometry is responsible for up to 30%–40% of time spent on particle transport in detector simulation (this fraction is highly experiment dependent) [2], thus being an important research topic for vectorization in the GeantV prototype. The main consumers of cycles are the four algorithms shown in Figure 1, which are implemented separately for each geometrical shape.

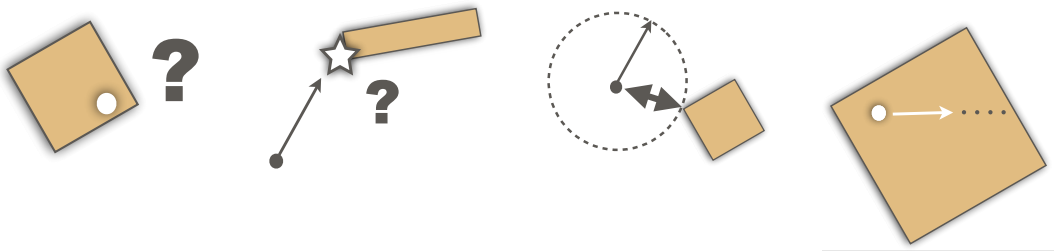
VecGeom introduces vectorization techniques in the implementation of a new geometry modeller. It is a continuation of the USolids [8] effort, a common library for geometrical primitives, unifying and enhancing the implementations that currently exist in ROOT [9] and Geant4 [10]. The scalar interfaces of VecGeom are therefore made to be compliant with the ones of USolids, allowing free interchangeability between the two. VecGeom and USolids are converging to a single code base.

To accommodate the scheduler of GeantV mentioned above, VecGeom exposes vector signature operating on baskets of particles. These signatures take SOA objects carrying information on the particles in the basket, instead of a single set of parameters. Since these interfaces come in addition to the scalar ones, there is an issue of signature multiplication requiring a lot of boilerplate code, which will be addressed in Section 2.3.

VecGeom does not rely on vectorization alone to achieve performance. In the spirit of USolids, the library employs the best available algorithms to solve the problems listed in Figure 1. This can mean picking and optimizing the best algorithm between existing ones in ROOT, Geant4 and USolids, or it can mean writing them from scratch if deemed relevant. When a new shape is implemented, performance is compared directly to that of the existing libraries. Algorithms are not accepted before their *scalar* version (in addition to the vectorized one) outperforms all existing implementations (more on different kernel versions follows in Section 2.1). After algorithmic optimization, the library uses a number of additional templating techniques for performance which are described in Section 2.1 and 2.4.

2 Templates for portability and performance

Templating is at the heart of VecGeom, being involved in every aspect of optimization. The philosophy is that whenever the parameters of a performance-critical component can be resolved



(a) Determine if point is inside a volume.

(b) Intersection check and distance to intersection with volume.

(c) Minimum bound on distance to volume boundary.

(d) Distance to exit current volume.

Figure 1: Primary problems solved by a geometry library during particle transport in simulation.

at compile-time, or even just if it has a finite set of configurations (see Section 2.4), templates will be used to generate specialized machine code.

2.1 Thinking in kernels

The approach taken by VecGeom to achieve portability and performance through templates is to separate performance-critical functionality into small plug-and-play *kernels*. Each kernel should have a single and well-defined purpose, delegating subtasks to separate kernels, promoting the sharing of kernels between algorithms. Kernels are built to operate on a single unit of input, meaning no explicit vectorization is typically done in them. Instead, vectorization can be conveyed by the *types* on which operations are performed (as this will often be a template parameter), and is described in Section 3. Logical constructs such as branches have to be abstracted to higher level functions in order to accommodate this. To ensure that kernels can indeed be shared between different contexts, kernels can live either as free C-like functions or as static methods of enclosing classes, but should never relate to an object. When an object-abstraction is useful, classes can instead implement static methods and call them from within their own methods.

2.2 Architecture abstraction

An important motivation for taking the kernel approach is *architecture abstraction*. VecGeom is a performance-oriented library, and as such aims at maximizing hardware utilization by offering to run scalar and SIMD code on CPU, and also run on accelerators. Rather than writing separate implementations for each target architecture and multiplying the required maintenance, VecGeom has opted for a generalized approach. The difference between architectures is isolated to memory management and the *types* that are operated on, with the latter being handled by the kernels. For this, the concept of *backends* was introduced. Backends represent an architecture to which kernels should compile, and are encapsulated in C++ trait classes. These classes contain the relevant types to operate on, as well as some static members, such as a boolean to toggle early returns. Backends are used as a template parameter of kernels that operate on the provided types, and by overloading higher level functions called on these types, the architecture specific instructions necessary to perform operations is decided. An example of a backend class is shown in Listing 1.

```
class VcBackend {
public:
  typedef Vc::double_v double_t;
  typedef Vc::int_v int_t;
  typedef Vc::double_m bool_t;
  static constexpr bool earlyReturns
       = true;
};
```

Listing 1: A C++ trait class representation of a backend, providing types to operate on and allowing other configuration options. Vc is used to wrap vector instruction sets.

```
template <class B>
typename B::double_t DistanceToPlane(
    Vector3D<typename B::double_t> const &point,
    Vector3D<typename B::double_t const &plane) {
  typename B::double_t result;
  result = point.dot(plane);
  // If behind the point is considered inside
  MaskedAssign(result < 0, 0, &result);
  return result;
}
```

Listing 2: Kernel operating on a backend class, such as the one shown in Listing 1.

2.3 Solving code multiplication

The introduction of the vector interfaces led to a *multiplication of signatures* necessary for a shape to implement, as each shape has to provide both the scalar and vector signatures. Because of the way kernels are designed, this quickly led to large amounts of boilerplate code that simply called different instantiations of the the same family of templates. These were not only cumbersome to write for each implemented shape, but also meant a lot of duplicate code to maintain in the future. To improve this, the *curiously recurring template pattern* (named by James Coplien [12]) is employed. This pattern allows a leaf class to inherit from a templated auxiliary class, passing its own type as a template parameter. The auxiliary class will then inherit from the appropriate base class and implement a number of methods calling static methods of the class passed as a template parameter. This can be either the inheriting class itself or a separate class entirely. Using the interface of VecGeom shapes, this meant that only a single templated method per algorithm had to be implemented for each shape. The boilerplate code used to call different configurations was collected in a single global auxiliary class. An example of this technique is provided in Listing 3.

2.4 Shape specialization

VecGeom defines a set of geometrical shapes. Some parameters describing the shapes have a significant impact on the algorithms: an angle configuration can eliminate the need for a trigonometric function, or a tube with no inner radius can just be treated as a cylinder. Creating separate shape classes for specific configurations would cause the number of primitives to explode. Instead, VecGeom takes advantage of this behind the scenes by using the concept of *shape specialization*. By having kernels template on one or more specialization parameters, blocks of code are tweaked or removed entirely from the implementation at compile time. This saves branch mispredictions and register allocation as compared to performing all branches at runtime. The concept is shown in Listing 4.

To avoid to expose end users to specialization, the instantiation of each configuration is done by a factory method. This limits specialization to a finite phase space, but hides the specifics through polymorphism, handing the user an optimized object behind the base primitive's interface. Specialization happens when shapes are *placed* into their frame of reference. Volumes instantiated by other means are not specialized, and will instead branch at runtime.

Figure 2 shows relative benchmarks between the same kernel compiled for three different scenarios: the unspecialized case where branches are taken at runtime, the specialized case

```
template <class B, class S>
class Helper : public B {
public:
  virtual double Distance(Vector3D<double> ...) const {
    // Implement virtual methods by calling the
    // implementation class methods
    S::template Distance<kScalar>(...);
  }
  virtual void Distance(SOA3D<double> ...,
                        double *distance) const {
    // Generate vector types from the input and loop
    // over the kernel
    S::template Distance<kVector>(...);
  }
};
```

Listing 3: Applying the curiously recurring template pattern to eliminate boilerplate code. A shape inheriting from the helper class only needs to provide a single kernel per algorithm, as the helper will use it to implement all the virtual function required by the interface.

```
template <bool hollow>
class TubeImplementation {
public:
  void Distance(...) {
    // Treat outer radius and
        both ends
    ...
    if (hollow) {
      // Treat inner radius
      ...
    }
  }
};
```

Listing 4: Change or remove blocks of code at compile-time. To avoid relying on the optimizer, an auxiliary function can also be introduced to explicitly specialize different cases of a template parameter.

where unused code has been optimized away at compile time, and the vectorized case treating four incoming rays in parallel using AVX instructions.

3 Running on multiple architectures

With the proposed kernel design, supporting multiple architecture means to provide the per-architecture backend trait class, and to write kernels in a generic fashion, letting types lead to the underlying instructions. Mathematical functions typically work by simple overloading on input types. Conditional statements need to be handled more carefully, as conditions evaluated from the input can have different values for each value in the vectors being operated on. This can be done using *masked assignments* and *boolean aggregation*: the overloaded masked assignment function uses an input condition to only assign input elements to a new value if the condition evaluates to true for the instance in question; the overloaded boolean aggregation functions reduce an input condition by aggregators *all*, *any* or *none*. When the input is scalar, these functions trivially assign the variable or return the conditional variable itself, respectively. An example of a kernel using a masked assignment is shown in Listing 2.

VecGeom uses the Vc [11] library to target CPU vector instructions. Vc provides types stored as packed, aligned memory, which are loaded into vector registers. Depending on the supported instruction set, this can compile to SSE, AVX or AVX512 intrinsics, operating on 2, 4 or 8 doubles at a time, respectively. Since vectorization by Vc is explicit, kernels instantiated with these types are guaranteed to use appropriate vector intrinsics. Figure 2 includes a benchmark for a kernel compiled to AVX intrinsics by operating on Vc types.

Supporting CUDA is fairly straightforward, as the VecGeom kernel architecture fits well into a GPU design. Kernel functions are simply annotated with `__host__` and `__device__` when compiling with nvcc, making them available to both CPU and GPU as needed. Additionally, the mask-based coding style described above means that branching is discouraged by design, showing how the generalized SIMD-oriented approach benefits multiple architectures.

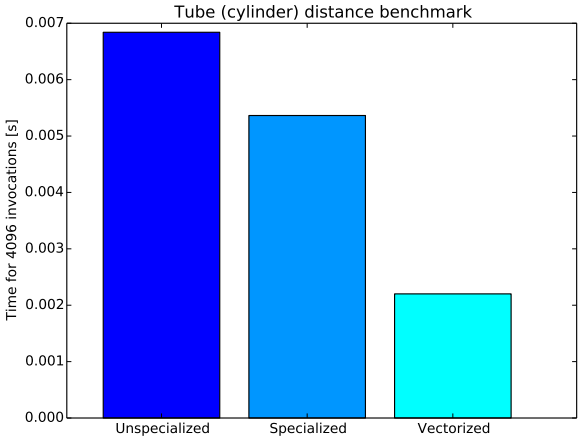


Figure 2: Relative benchmarks of the distance algorithm to a cylinder, which is the special case of a tube with no inner radius. All configurations are compiled from the same kernel. Vectorized kernel compiled to AVX instructions, treating four rays in parallel.

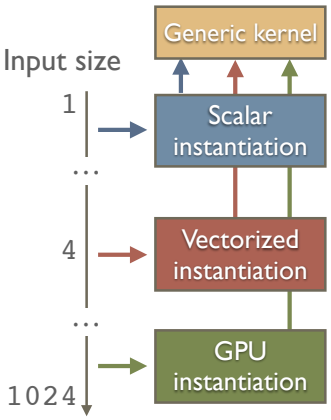


Figure 3: Methods targeting different architectures are instantiated from the same templated kernel.

Once kernels have been instantiated with different backends, it is easy to perform benchmarks such as the ones shown in Figure 4, comparing performance and scaling on different architectures. This can also be used for verification purposes: once an algorithm is validated on one architecture, other architectures can conveniently be validated against this result.

3.1 GPU support

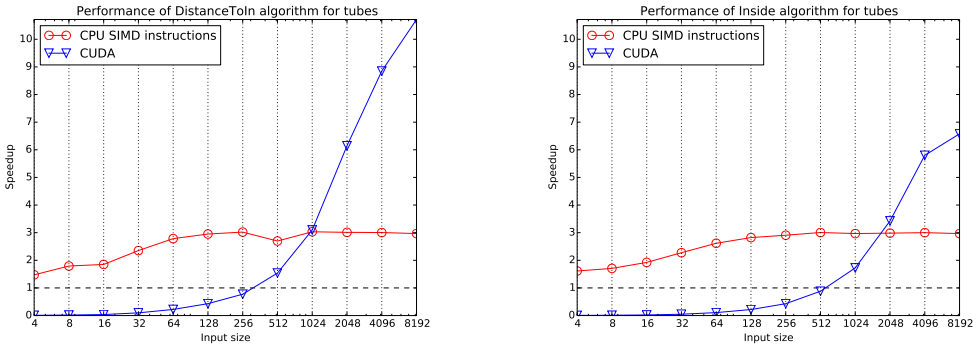
The library support for GPUs was first implemented in CUDA. Although this opposes the desire to avoid vendor lock-in, it was considered the more productive way forward given the current state of CUDA and OpenCL compilers.

The support offered is two-fold: shape primitives and their respective kernels are provided as device functions that can be run from user code, and memory synchronization capabilities are offered to copy geometries built in main memory to GPU memory. Together this offers full flexibility to integrate the GPU in user code, whether it be exclusively or as a target for offloading.

An effort to investigate the use of this approach with OpenCL has been significantly hampered by the lack of support for polymorphism in the current specification.

3.2 Feasibility of architecture independence

Several advantages of architecture independent kernels have been discussed, such as maintainability, hardware utilization and cross-architecture benefits of SIMD data structures. There are of course disadvantages to this approach, the most prominent being loss of important architecture specific optimizations by restricting kernels to use higher-level abstractions. This is recognized in VecGeom by allowing implementation of template specializations of a given backend for kernels or overwriting methods (that are otherwise implemented by a helper as



(a) Distance algorithm, 50 floating point operations per kernel execution.

(b) Inside algorithm, 10 floating point operations per kernel execution.

Figure 4: Relative benchmarks of generic kernels run on scalar, AVX and CUDA architectures on consumer-grade hardware (3.4 GHz Ivy Bridge Core i7 and GeForce GTX 680). Speedup with respect to the scalar instantiation of the same kernel shown as the black dashed line. The GPU needs considerably more floating point operations than the CPU to efficiently hide memory latency.

demonstrated in Section 2.3) in leaf classes. In this way, the code base can be extended to include specific implementations without sacrificing the benefits of having general kernels.

Since kernels often template on one or more parameters, and to encourage compiler optimizations in general, all kernels are implemented in header files. In performance intensive code, function calls (virtual function calls in particular) have proven to be a significant performance liability, amplified by the design that encourages high separation of functionality. Avoiding this comes with at least two (correlated) penalties, being compilation time and binary file size. Although this is typically an acceptable trade-off, the impact can be significant, and is monitored as development progresses to ensure a desirable balance.

4 Summary

Through the philosophy of small, reusable and general kernels, and the creation of abstract backends, VecGeom has achieved a small code base with a large degree of portability, supporting both CPU and accelerators. Architecture specific optimizations remain possible by using template specialization and method overwriting.

Benchmarks were presented that demonstrate benefits of vectorization achieved in a general fashion by employing the Vc library, and the gain in the performance obtain by specializing shapes at compile time to remove unused code (all configurations instantiated from the same templated kernel).

GPU support is offered through device instantiations of kernels and memory synchronization API, allowing exclusive or offloading use by an external scheduler.

References

[1] J. Apostolakis et al., *Rethinking particle transport in the many-core era towards GEANT 5*, 2012 J. Phys.: Conf. Ser. **396** 022014, doi:10.1088/1742-6596/396/2/022014.

[2] J. Apostolakis et al., *Vectorising the detector geometry to optimise particle transport*, 2014 J. Phys.: Conf. Ser. **513** 052038, doi:10.1088/1742-6596/513/5/052038.

[3] AMD, *3DNow! Technology Manual*, 21928G/0, Rev. G, March 2000.

[4] Intel Corporation, *Pentium III Processor for the PGA370 Socket at 500 MHz to 1.13 GHz*, 245264-08, Rev. 8, June 2001.

[5] A. Gheata, *Adaptative track scheduling to optimize concurrency and vectorization in GeantV*, Proc. ACAT 2014, forthcoming.

[6] Sandro Wenzel, *Towards a generic high performance geometry library for particle transport simulation*, Proc. ACAT 2014, forthcoming.

[7] Intel Corporation, *Programming Guidelines for Vectorization*, 2014.

[8] Marek Gayer et al., *New software library of geometrical primitives for modeling of solids used in Monte Carlo detector simulations*, 2012 J. Phys.: Conf. Ser. **396** 052035, doi:10.1088/1742-6596/396/5/052035.

[9] R. Brun and F. Rademakers, *ROOT: An object oriented data analysis framework*, Nucl. Instrum. Meth. A **389** (1997) 81.

[10] S. Agostinelli et al., *Geant4 - a simulation toolkit*, Nuclear Instruments and Methods **A** 506 (2003) 250-303.

[11] M. Kretz and V. Lindenstruth, *Vc: A C++ library for explicit vectorization*, Softw: Pract. Exper., 42: 14091430. doi: 10.1002/spe.1149, December 2011.

[12] J. Coplien, *Curiously Recurring Template Patterns*, C++ Report, February 1995.