Navigation, field integration and track parameter transport through detectors using GPUs and CPUs within the ACTS R&D project

A Salzburger[1], J Niermann[1,2], B Yeo[3,4], A Krasznahorkay[1] and S N Swatman[1,5]

[1] CERN, 1211, Geneva, Switzerland
[2] II. Physikalisches Institut, Georg-August-Universität Göttingen, 37073, Göttingen, Germany
[3] Department of Physics, University of California, CA 94720, Berkeley, USA
[4] Lawrence Berkeley National Laboratory, CA 94720, Berkeley, USA
[5] University of Amsterdam, 1012 WX, Amsterdam, The Netherlands

E-mail: `andreas.salzburger@cern.ch` `joana.niermann@cern.ch` `beom.ki.yeo@cern.ch` `attila.krasznahorkay@cern.ch` `stephen.nicholas.swatman@cern.ch`

Abstract. In order to tackle the amount of data that is expected to be produced in the high luminosity era of the *Large Hadron Collider* (LHC) and future experiments, the rate at which charged particle tracks can be reconstructed must increase significantly. Massively parallel GPGPU architectures are promising candidates to fulfil this requirement. Current state-of-the-art reconstruction software is highly optimized for deployment on CPU architectures and, in many cases, cannot be used on hardware accelerators without undergoing extensive adaptations or a reduction of the problem scope. The ACTS community currently hosts multiple R&D projects investigating the implementation of a complete track reconstruction chain demonstrator that runs on CPU and GPU devices. One of these projects is DETRAY, which provides a GPU-friendly detector geometry description and track state propagation, including material effects. It uses the VECMEM library for memory management and the COVFIE library for a detailed magnetic field map. Within DETRAY, track parameters and their associated covariances can be propagated through a detector geometry with a magnetic field by applying an adaptive Runge–Kutta–Nyström algorithm. It uses a single-source implementation that has been shown to run on the host, as well as on a GPU device using CUDA.

1. Track reconstruction in the ACTS R&D project

A Common Tracking Software (ACTS) [1, 2, 3] provides a toolkit of algorithms for track reconstruction that are written in modern C++, encompassing all steps of the tracking chain, from hit clustering to track fitting. An accurate track reconstruction depends on an exact description of the detector geometry, which is provided by the ACTS *tracking geometry*. The tracking geometry grants access to the precise positioning and shape of the sensitive and passive detector elements, while at the same time reducing complexity to save computing resources. The detailed structures of a detector geometry are abstracted to simpler detector element surfaces for a fast navigation onto which an averaged material is mapped. ACTS can build a tracking geometry from various sources via dedicated geometry plugins, like the *ROOT Geometry Package* (*TGeo*) [4] and *DD4hep* [5].

Relying on e.g. a polymorphic geometry description and a vector-of-vector data layout has made the adoption of ACTS tracking algorithms on hardware accelerators difficult. Furthermore, some C++ Standard Library functionality is not available when working with backends like CUDA [6] or SYCL [7]. In a dedicated study, several R&D projects were launched to investigate the adaptation of the ACTS track reconstruction pipeline to device [8, 9, 10, 11, 12, 13]. The algorithmic code, i.e. clustering, seeding, track finding and fitting, is part of the TRACCC project. The DETRAY project provides the tracking geometry and track state propagation. The magnetic field description that is needed in DETRAY is implemented using the COVFIE library. Last but not least, the VECMEM and ALGEBRA-PLUGINS projects supply memory management and linear algebra functionality to the other projects.

2. The detray Project - Overview

The DETRAY project provides a header-only library that contains a geometry description capable of modelling an ACTS tracking geometry to full detail. A track state parametrization can be propagated in a magnetic field through the tracking geometry together with its covariance, including material effects. The DETRAY codebase is fully heterogeneous, so that any class can be instantiated and used in both host and device code. In the current propagation model, where threads are directly matched onto tracks, the entire propagation loop can thus be expressed in a single kernel of less than 100 lines of code.

The library comes with many native geometric shapes, for example, rectangles, cylinders or discs. These types are fully compile-time polymorphic and are managed in tuple-of-vector based containers with *visitor* functionality, i.e. a given callable is checked against every tuple entry using a type identifier index. Custom shapes can be added by re-compiling the central *detector* object. The detector holds the tracking geometry as well as the magnetic field data and performs the data movement to device if required. The underlying data structure of the detector is based on container indices and does not make use of pointers. A given detector is therefore relocatable in memory.

The memory management of the vector-like data in the detector is handled by the VECMEM library. All constructors are allocator-aware and take a polymorphic memory resource that determines the allocation scheme, see [11] for more information. Furthermore, all classes are templated on the container types, so that e.g. the `std::vector` based host-side container can be switched against the `vecmem::device_vector` type in kernel code. To construct the non-owning device-side detector, a *view* is generated from the host instance and passed to the kernel. This view is given to the corresponding constructor of the device-side detector, which unpacks it and hands the sub-views of its data members to their respective constructors until the full type is assembled. Afterwards, the new device-side detector instance can be used in the same way as host-side.

3. Tracking Geometry Implementation

The geometry description in DETRAY follows the ACTS tracking geometry model, which is surface based and describes volumes as an aggregation of boundary surfaces. To avoid virtual function calls in DETRAY, the shapes of the surface primitives have been implemented as lightweight data structures that define all properties needed to characterize a particular local geometry. They contain shape specific type definitions and methods, such as e.g. the local coordinate system, the type of intersection algorithm to be used with the shape, as well as the number of boundary values and the respective boundary check method. The coordinate system transformations and intersection algorithms are implemented as separate functions so that they can be re-used between shapes. For example, a `rectangle2D` shape will have two half lengths as boundaries in a two dimensional local Cartesian coordinate system and will use a line-plane intersection algorithm.

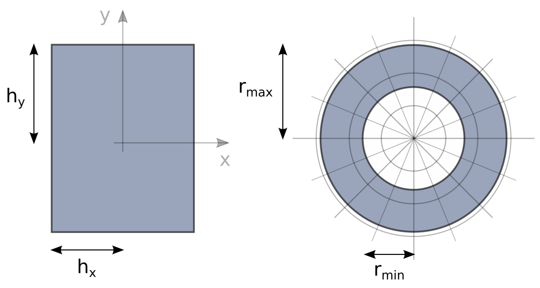

Figure 1. A mask of rectangular (left) and ring (right) shape. The boundary values are defined in the respective local coordinate system.

The shape types can be given as a template argument to any class that exhibits a geometrical shape, which can then adopt the relevant definitions and methods. This allows for an easy extension of DETRAY with new shapes while avoiding inheritance. An important example are the surface masks, which define the extent of a surface primitive of a given shape in its local coordinate system and thus hold all shape related data of a surface instance, see Figure 1. Other examples are the building of the grid data structures or, in future, the bounding volume class.

Since the mask corresponding to each distinct surface primitive is of a different type and cannot be referenced by a polymorphic base class pointer, the detector needs to hold them in separate vector containers which are bundled in a tuple. Calling shape specific behaviour of a surface then commences through functions that are given to a *visitor* method of the mask tuple container. The same container pattern is applied for the surface material since there are currently two types of material available, which abstract the detailed distribution of material in the tracking geometry from the full geometry description. These are homogeneous *slabs* or *rods* (depending on the surface shape) of material of a given thickness, radiation and interaction length. The material description itself is configurable in many material specific parameters, such as e.g. atomic number or molar density. A number of predefined materials are available, which can additionally be composed into material mixtures.

Surfaces are placed in the detector space by affine transformations, which are kept in a transform store container as part of the detector. The primary responsibility of the transform store will be to identify the correct transform for a given surface and geometric context, so that alignment conditions data can be handled in a thread-safe way. Considering the transform store design in conjunction with the combinatorics of different shape and material types, the surface class in DETRAY has been reduced to a descriptor structure that merely keeps the type identifiers and indices of the surface components in the detector (tuple-) data stores. These surface descriptors are lightweight and are designed to allow for a straightforward implementation of surface instancing.

4. Navigation model

In DETRAY, volumes primarily subdivide the detector into navigation domains. Every volume is described by its boundary surfaces, called portals, and provides an access method to the sensitive and passive surfaces that it contains. The portal surfaces link a volume to its neighbours by referencing a number of masks, each of which standing for a shared boundary surface. Non-portal surfaces always link back to their mother volume. Empty gap volumes are placed in between the sensor layer volumes, so that the navigation flow can be described as moving through a succession of volumes via the portal links. Within a given volume, the surfaces will be provided by an acceleration data structure, which is defined by every volume individually and performs a neighborhood lookup around the track state position to reduce the navigation to nearby surfaces.

The *navigator* class is built around a surface candidate cache which is filled by ray intersection from the surfaces returned by the volume accelerator data structure. The assumption is, that on a small scale the track is straight enough to be approximated by a tangential ray. The benefit of this approach is that ray-surface intersections are well studied and efficient to compute. Given the distance to the closest surface candidate, the magnetic field integration can commence in the next step to advance the track state position.

The navigator exposes a number of methods to the propagation that update the candidate cache in different ways in an effort to minimize the amount of intersections and sorting that needs to be done in a given propagation step. These methods are governed by a *trust level* flag which can be set by the other participants in the propagation in case (re-)navigation is required:

- ***Full Trust:*** The track state has not changed since the last call to the navigator, therefore do nothing.
- ***High Trust:*** Only update the distance to the current next candidate surface. This assumes that the candidates and their sorting in the cache are still consistent with the track state.
- ***Fair Trust:*** Update all candidates and resort the cache. This assumes that the track state has not moved out of the current local neighbourhood of surfaces, yet.
- ***No Trust:*** (Re-)initialize the navigation in a given volume. Also called during a navigation volume switch.

5. Field Integration

The track state parametrization contains eight values $(x, y, z, t, d_x, d_y, d_z, q/p)$, with d the normalized track direction, if described in the global detector frame and six values $(\text{loc}_0, \text{loc}_1, \varphi, \theta, q/p, t)$ when expressed in the local coordinate system of a surface. In order to advance the track state through the detector, the equation of motion of a moving charged particle in the detector's magnetic field has to be solved. Since, in general, the field is inhomogeneous, no closed algebraic solution for the particle trajectory exists. In this case, the field integration has to be done numerically. In lockstep with the track position, the track parameter covariance matrix is updated in order to transport the covariances from their initial estimate at the beginning of the track propagation to their values at the final surface, see e.g. [17]. This is done by transforming the initial covariance matrix with a Jacobian that represents the full parameter error propagation. In particular, during the field integration step, this means that the Jacobian corresponding to the field integration needs to be calculated.

Advancing the track state position and updating the corresponding transport Jacobian are the tasks of the *stepper* class, which is called after the navigator during the propagation. It applies a fourth order Runge–Kutta–Nyström (RKN) algorithm with an adaptive step size [18, 19, 20] to ensure a sufficiently low integration error. The maximal path length in a given step is determined by the navigator as the straight-line distance to the next surface on the trajectory. Consequently, the adaptive RKN algorithm is allowed to only reduce the step size by running the step size adjustment uniquely when the integration error exceeds a given tolerance, in which case the step size scale factor is guaranteed to remain smaller than one. This results in a greedy stepping algorithm that does not need to (re-) check the navigation step size constraint.

The magnetic field vector at the sampling positions of the RKN algorithm are provided by the COVFIE library. Currently available in DETRAY is a homogeneous magnetic field, which is mainly used for testing and validation. A magnetic field map that is read from file and uses e.g. a nearest neighbour interpolation to get a precise field vector at an arbitrary position is about to be added as well.

6. Track State Propagation
Both stepper and navigator are called inside the propagation loop, which is run by the *propagator* class until the track state either leaves the detector world space or one of a number of predefined termination criteria is reached. The propagator can in principle be equipped with different stepper and navigator implementations, depending on the problem setting, as well as a variable number of so-called *actors*. Actors are a concept that has been adopted from ACTS and are used to extend the propagator with various functionality, such as the aforementioned abort criteria. The actor chain is assembled at compile-time by giving each actor type as template parameter to the chain. The actor states are strapped into a tuple of references from which each individual state is retrieved when an actor is called. Actors may *observe* other actors and will thus be handed the subject's state in addition to its own on every call. Furthermore, an observing actor can itself be subject to other observing actors, resulting in a compile-time, depth-first call tree of propagation actions. The actor mechanism allows for an extendable and highly flexible expression of the propagation loop.

For example, the transport of the track state covariance matrix [17] is handled by a dedicated actor. Starting from a track state that is bound to a surface, it will determine the Jacobian to transform the covariance to the global detector coordinate frame, then add the transport Jacobian obtained from the stepper and, following that, the Jacobian for the local coordinate transformation of the next surfaces is calculated. This process is repeated until the final, full Jacobian for the track can be assembled. In between, whenever the track is propagated to a surface, the covariance matrix needs an additional update with respect to the surface material. This is handled by the *material interactor*, which adds terms corresponding to multiple scattering and energy loss effects according to Bethe-Bloch. Other examples for actors are the *path limit aborter*, which prevents infinite propagation, the *random scatterer* used in the simulation or, in future, the Kalman Filter that is due to be added in TRACCC.

7. Summary and Outlook
In the DETRAY testbed detector environment, which is based on the pixel component of the ACTS generic detector [21], track parameter and error propagation was demonstrated in a single-source codebase. The propagation model follows the ACTS design, using a tracking geometry based navigation, an adaptive fourth order Runge-Kutta-Nyström algorithm to perform the field integration, as well as a propagation loop including compile-time pluggable actors.

The code has been tested using a homogeneous magnetic field on a CPU and a CUDA backend. In a very preliminary benchmark of the full propagation loop with 50 repetitions, see Figure 2, the performance was competitive between the multithreaded host and the device when running a large number of tracks. While the device propagation performance is virtually identical for the Eigen3 and custom `std::array`-based linear algebra implementations, and might hence be dominated by other effects, the difference in the host-side propagation performance between the two is not currently understood.

Though there is no clear gain visible for the device-side over the host-side propagation, yet, the current implementation comes with a number of caveats. For example, all data are held in CUDA unified memory, geometry acceleration data structures and detector-based memory optimization, like surface instancing or sorting, are not yet deployed and impacts from thread divergence are expected on the device-side as well. In the current parallelization model, one track is assigned per thread in both the host and device implementation and propagated to the end by the same thread. This model is straightforward to implement in a single-source code design, but also likely results in thread divergence when e.g. one track encounters a surface and needs to execute special code, e.g. the covariance updates, while all other tracks are still running the field integration step. The benchmarks are consequently under further investigation.

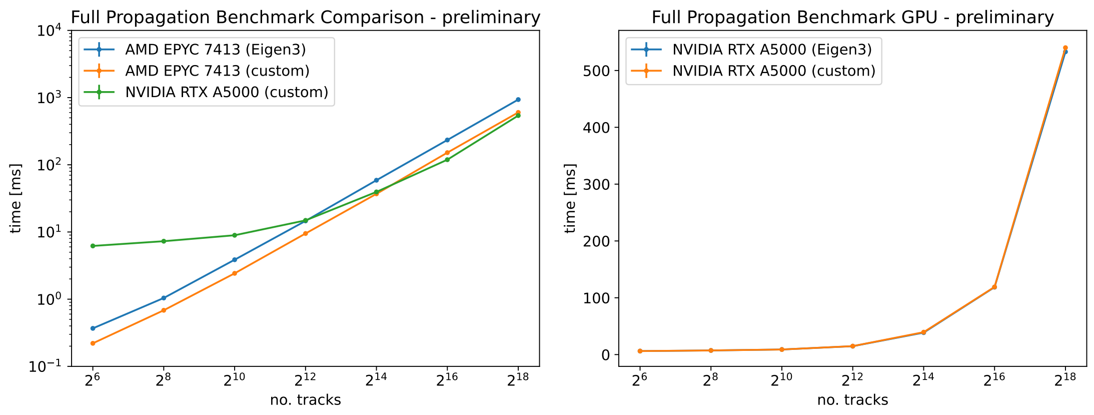

Figure 2. Benchmarks of the full propagation loop in single precision on an AMD EPYC 7413 24-Core (48 threads, using openMP [22]) host processor and an NVIDIA RTX A5000 device for two linear algebra implementations [13].

Work is on-going to read ACTS tracking geometries into DETRAY, thus allowing to use existing detector descriptions in DETRAY without further adaptation. This also includes geometry acceleration data structures and more detailed material mapping, neither of which are deployed in DETRAY, yet. A detailed physics validation and performance benchmarking study will follow.

Acknowledgments
This work benefited from support by the CERN Strategic R&D Programme on Technologies for Future Experiments (CERN-OPEN-2018-006) [23].

This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 05E18CHA).

This work was funded by the NSF under Cooperative Agreement OAC-1836650.

References
[1] Salzburger A *et al.* 2022 ACTS Project GitHub URL `https://github.com/acts-project/acts`
[2] Ai X *et al.* 2022 *Computing and Software for Big Science* **6** 8
[3] Ai X, Mania G, Gray H M, Kuhn M and Styles N 2021 *Computing and Software for Big Science* **5** 20
[4] Brun R, Gheata A and Gheata M 2003 *Nucl. Instrum. Methods Phys. Res., Sect. A* **502** 676–680
[5] Petrič M, Frank M, Gaede F, Lu S, Nikiforou N and Sailer A 2017 *J. Phys.: Conf. Ser.* **898** 042015
[6] NVIDIA 2022 URL `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`
[7] Reyes R and Lomüller V 2016 *Parallel Computing: On the Road to Exascale* pp 673–682
[8] Salzburger A, Niermann J, Yeo B and Krasznahorkay A 2023 *J. Phys.: Conf. Ser.* **2438** 012026
[9] Salzburger A *et al.* 2020 GitHub URL `https://github.com/acts-project/detray`
[10] Krasznahorkay A *et al.* 2020 GitHub URL `https://github.com/acts-project/traccc`
[11] Swatman S N 2022 GitHub URL `https://github.com/acts-project/covfie`
[12] Swatman S N, Krasznahorkay A and Gessinger P 2023 *J. Phys.: Conf. Ser.* **2438** 012050
[13] Krasznahorkay A *et al.* 2020 GitHub URL `https://github.com/acts-project/algebra-plugins`
[14] Brun R and Rademakers F 1997 *Nucl. Instrum. Methods Phys. Res., Sect. A* **389** 81–86
[15] Guennebaud G, Jacob B *et al.* 2010 Eigen v3 URL `http://eigen.tuxfamily.org`
[16] Bell N and Hoberock J 2012-12 pp 359–371 GPU Computing Gems jade ed ISBN 9780123859631
[17] Lund E, Bugge L, Gavrilenko I and Strandlie A 2009 *J. Instrum.* **4** P04016
[18] Myrheim J *et al.* 1979 *Nucl. Instrum. Methods* **160** 43–48
[19] Frühwirth R 1987 *Nucl. Instrum. Methods Phys. Res., Sect. A* **262** 444–450
[20] Lund E, Bugge L, Gavrilenko I and Strandlie A 2009 *J. Instrum.* **4** P04001
[21] Kiehn M *et al.* 2019 *EPJ Web Conf.* **214** 06037
[22] Dagum L and Menon R 1998 *Comput Sci Eng* **5** 46–55
[23] Aleksa M *et al.* 2018-12 Tech. rep. CERN CERN-OPEN-2018-006