Computing and Software for Big Science (2021) 5:20
https://doi.org/10.1007/s41781-021-00065-z

ORIGINAL ARTICLE

A GPU-Based Kalman Filter for Track Fitting

Xiaocong Ai[1] · Georgiana Mania[1,2] · Heather M. Gray[3,4] · Michael Kuhn[5] · Nicholas Styles[1]

Received: 15 April 2021 / Accepted: 13 September 2021 / Published online: 5 October 2021

Abstract
Computing centres, including those used to process High-Energy Physics data and simulations, are increasingly providing significant fractions of their computing resources through hardware architectures other than x86 CPUs, with GPUs being a common alternative. GPUs can provide excellent computational performance at a good price point for tasks that can be suitably parallelized. Charged particle (track) reconstruction is a computationally expensive component of HEP data reconstruction, and thus needs to use available resources in an efficient way. In this paper, an implementation of Kalman filter-based track fitting using CUDA and running on GPUs is presented. This utilizes the ACTS (A Common Tracking Software) toolkit; an open source and experiment-independent toolkit for track reconstruction. The implementation details and parallelization approach are described, along with the specific challenges for such an implementation. Detailed performance benchmarking results are discussed, which show encouraging performance gains over a CPU-based implementation for representative configurations. Finally, a perspective on the challenges and future directions for these studies is outlined. These include more complex and realistic scenarios which can be studied, and anticipated developments to software frameworks and standards which may open up possibilities for greater flexibility and improved performance.

Keywords Particle tracking · Track fitting · Parallelization · GPU · CUDA · OpenMP

Introduction

The reconstruction of the trajectories of charged particles for High-Energy Physics (HEP) experiments is a very computationally demanding task, which is performed both when selecting events in real time with the online *trigger* and during the subsequent high-precision offline reconstruction of events for physics analysis. The most commonly used techniques are adaptive methods based on the Kalman filter [1, 2], which account for the trajectories of charged particles in magnetic fields and the energy loss of charged particles in the detector material. See Ref. [3] for a review. As the execution time of such algorithms explodes combinatorially with the number of charged particles, the advent of the upgrade to the Large Hadron Collider (LHC), the High-Luminosity LHC (HL-LHC), portends an even greater challenge, with events containing up to 10,000 tracks.

For many years, HEP has been relying on Moore's Law [4], the observation that the number of transistors on an integrated circuit doubles approximately every two years. As the circuits have begun to approach intrinsic limits in terms of density and power, Moore's Law has begun to slow, further complicating potential performance improvements [5]. In addition, other computing architectures have become increasingly powerful and hence popular, such as graphical processing units (GPUs) and field programmable gate arrays (FPGAs). Therefore there has been a shift towards achieving speed improvements by adding additional cores, particularly at high-performance computing centers. These many-core systems require highly parallel code to be fully exploited, requiring additional knowledge from software developers. Moreover, much of the existing code for high-energy physics experiments is not well-suited to such architectures and

✉ Xiaocong Ai
xiaocong.ai@desy.de

[1] Deutsches Elektronen-Synchrotron, Hamburg, Germany

[2] Informatics Department, University of Hamburg, Hamburg, Germany

[3] Physics Department, University of California, Berkeley, CA, USA

[4] Physics Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA

[5] Faculty of Computer Science, Otto von Guericke University Magdeburg, Magdeburg, Germany

hence requires significant development and adaptation to be able to exploit them.

Porting algorithms to GPUs typically requires specialized code redesign and optimization, but performance gains through vectorization using Single Instruction Multiple Data (SIMD) instructions, and parallelization using many-core CPU architectures often require less significant changes to the code base. Several HEP experiments have leveraged the power of many-core systems for real-time online and/or offline track reconstruction [6–10]. These studies have demonstrated good scalability of the throughput of events per second with the number of CPU cores. GPU-accelerated track reconstruction has also been studied. For example, both the ALICE [11] and LHCb [12] experiments at the LHC have proposed a GPU-based High-Level Trigger (HLT) to handle the much increased data rate expected during Run 3 of LHC [13–15]. In particular, LHCb has implemented a fully GPU-based high-throughput HLT framework, which processes a data rate of up to 40 Tbit/s using approximately 500 GPUs [15]. In these studies, ALICE and LHCb used a simplified or parameterized Kalman filter for track fitting for maximum speed with some impact on track resolution compared to offline track reconstruction using a full Kalman filter. The level of resolution loss is either acceptable for the online identification of interesting events for further offline analysis [15] or is recovered through dedicated optimization of the HLT tracking algorithms [13]. Initial studies of porting a full Kalman filter to GPUs can be found in Ref. [7]. Other track finding algorithms such as the Cellular Automaton and Hough Transforms have also been studied on GPUs [16, 17]. GPUs are also used for accelerating other steps of online event processing at HEP experiments, e.g. cluster finding [15, 18], vertex reconstruction [19] and event selection [20] and Ref. [21] presents a recent review of applications of GPUs for online event processing in HEP. The trend is generally towards bringing the full reconstruction chain to GPUs in order to minimize the penalties from intermediate data transfer between host and GPUs (see Ref. [15, 18]). Beyond HEP, GPU-accelerated Kalman filtering has been explored for a range of applications [22, 23]. However, these use cases tend to focus on much larger (up to three orders of magnitude) matrix sizes than are typical in HEP applications, and so the direct applicability is limited.

We present a proof-of-concept of a full Kalman filter algorithm on GPUs utilizing A Common Tracking Software (ACTS) [24–28], which provides a toolkit of algorithms for track reconstruction within a generic, framework- and experiment-independent software package. Detailed studies of the physics and technical performance are presented for two different GPU architectures and compared to performance on CPUs. In particular, we identify and discuss the key challenges in the implementation and highlight future directions towards the development of an even more performant full Kalman filter algorithm.

The Kalman Filter and ACTS

Track reconstruction is typically a multi-stage procedure, wherein candidates can be rejected at each stage. This approach allows high reconstruction efficiency and purity to be achieved in the final output collection, while reducing the overhead from processing unwanted candidates further than necessary. It starts from measurements (deposited energy in sensitive elements of the detector) and combines them in various configurations (including appropriate calibrations at various stages) to form plausible candidate trajectories. Accurate estimations of the parameters which define the mathematical form used to describe the trajectory are then made.

After any required pre-processing of the raw measurements, a typical first step is *Seeding*, in which small sets of compatible measurements are grouped using simple criteria and an initial trajectory estimate made. Seeds passing requirements can then be used as the basis for *Track Finding*, in which additional compatible measurements are added to the trajectory through the detector. Once the full set of measurements for the trajectory is obtained, a *Track Fitting* step can be performed, to precisely estimate the parameters and their covariance.

A commonly used approach and important tool in many track reconstruction applications is the Kalman filter [1, 2]. Developed in the late 1950s, the initial application of the Kalman filter procedure was in ballistics [29], where it allowed telemetry data for the heading and acceleration of the projectile to be combined with information on its location. The generalized procedure, in which measurements are combined with predictions based on an underlying model, results in state estimates more precise than either measurements or predictions alone, and has since been very widely used in many fields.

Within track reconstruction, a typical Kalman filter *step* would proceed as follows (see Fig. 1):

1. An initial estimate of the track state (i.e. helix parameters) at a given position is taken as the starting point.
2. This track state is propagated according to the track model on to the next *Measurement Surface* (i.e. the reference plane of a sensitive detector), providing a prediction of track state on this surface.
3. The prediction is combined with the measurement at this surface, if present, either through a weighted average

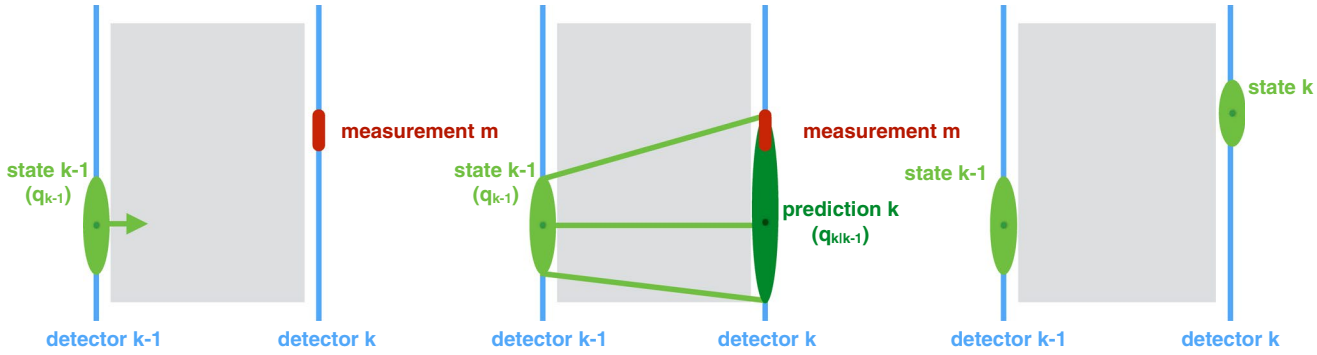

Fig. 1 Illustration of the different steps of the Kalman filter using a simplified detector model consisting of only two layers. (Left) The track state at the $(k-1)$th surface is indicated with the light green ellipse, and a measurement on the kth surface is indicated in red. (Center) A prediction is made for the track state on the kth surface and the size of the prediction is indicated with the dark green ellipse. (Right) The track state on the kth surface is updated by including the measurement on the kth surface

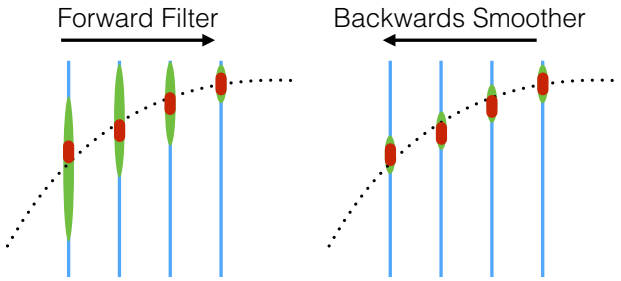

Fig. 2 Illustration of a forward filter and a backwards smoother on a simplified four layer detector geometry. The red points indicate the measurements and their uncertainties on each layer. The green points indicate the predictions. The predictions from the forward filter (left) are obtained when the filter is run from left to right. The predictions from the backwards filter are obtained during a second pass of the filter when it is run from right to left

or the so-called *Gain Matrix* formalism forming a new track state which is used to update the initial estimate.

4. This new estimate is then used for further Kalman steps, up to the end of the trajectory.

The Kalman procedure has the property that the next state estimate can be determined from the one preceding it. While this is a useful property in many cases, as it requires no 'history' to be stored, it has the consequence that only the final state contains the full information about all the steps preceding it, and therefore the best possible precision. To allow the prior track states to benefit from this information (e.g. to allow a χ^2 quality metric to be defined based on measurement residuals), an additional stage is needed. This *Smoother* stage can be performed using one of two approaches: either using essentially the same procedure as the forward Kalman filter but in reverse direction as illustrated in Fig. 2 or using the Rauch–Tung–Striebel (RTS) smoother [30] formalism with the stored Jacobians between states calculated during the forward Kalman filter steps. The latter approach does not require a second propagation of the track parameters and is therefore expected to have better timing performance.

ACTS has its origins in the track reconstruction algorithms used by the ATLAS experiment [31]. In addition to a tracking toolkit, ACTS also includes a fast simulation package. The ACTS code is designed to be inherently thread-safe to support parallel code execution and the data structures are vectorised. The implementation has been designed to be fully agnostic to detection technologies, detector design, and software frameworks so that it can be used by a range of experiments. The Eigen library [32] is used for algebra operations. In addition, ACTS is designed to be an R&D platform for the development of new algorithms and the porting of existing algorithms to new hardware platforms. See Ref.[27, 28] for further details.

While various representations of trajectories are possible, in this paper we will focus on helical trajectories of charged particles in a solenoidal magnetic field described using the following parameters:

- Two parameters loc_0 and loc_1 describing the spatial coordinates represented in the local frame of the measurement plane. In the special case of describing the parameters at a perigee surface[1], these become the transverse and longitudinal impact parameters d_0 and z_0 respectively.
- The polar and azimuthal angles of the particle momentum vector direction, ϕ and θ, at that point.
- A curvature parameter, expressed as the ratio of charge to momentum $\frac{q}{p}$.
- The time t.

[1] A surface defined at the point of closest approach to a reference point, for example the nominal interaction point in a particle collider.

Both the backwards-propagation and the RTS Kalman smoothing approaches are available within ACTS. The latter approach is used for the performance studies in this paper.

Parallelization and Offloading Techniques

There is a wide range of tools and frameworks available that can improve the runtime performance of scientific code via parallelization and offloading. Two of the most widely used frameworks are Open Multi-Processing (OpenMP) [33] and Compute Unified Device Architecture (CUDA) [34]. While the former traditionally allows parallelization on CPU systems via multiple threads, the latter is used to offload parts of the code to massively parallel Nvidia GPUs.

OpenMP

OpenMP is a compiler-based high-level approach for thread parallelization on shared memory architectures. One of its outstanding features is that it is very easy to use and does not require knowledge of threading and operating system internals [35]. It is available for C, C++ and Fortran, some of the most-widely used programming languages for scientific computing. OpenMP achieves its simplicity by being integrated into the compiler, which facilitates the parallelization of applications. Compiler support is widely available, which allows it to be used on personal computers as well as supercomputers. Applications are annotated with so-called *pragmas*, and turned into parallel code by the compiler; for instance, a simple `#pragma omp parallel for` pragma instructs the compiler to parallelize the subsequent *for* loop. OpenMP takes care of thread management and scheduling as well as data decomposition, which allows developers to focus on the problem they want to solve.

One of OpenMP's drawbacks has been its focus on CPU-based parallelism. However, it has recently been extended with improved offloading functionality that allows the compiler to offload certain parts of an application to accelerators such as GPUs and FPGAs. Consequently, OpenMP now can target both CPUs and GPUs, which offers better portability than vendor-specific approaches such as CUDA [36].

CUDA

CUDA is a parallel computing platform and application programming interface introduced by Nvidia for their line of GPUs [37]. It allows highly parallel GPUs to be used for general-purpose computations such as those common in high-energy physics. It is available for a range of programming languages, including C, C++, and Fortran. Wrappers are also available for additional programming languages, such as Python, R, Julia and many others.

GPUs are very specialized processing units and feature a high number of computing cores, which can be leveraged for scientific computations. Programs are offloaded to the GPUs in the form of so-called *compute kernels*, that is, single functions and their associated data. A kernel executes in parallel across a set of threads, which can use per-thread registers. Moreover, threads are aggregated into so-called warps that are executed concurrently. Several of these warps can be grouped into a thread block, which has access to a fast region of shared memory that all threads within the block can access. Finally, thread blocks can be combined into grids by the programmer. Thread blocks in a grid can only share data via global memory. Details of the CUDA programming model can be found in Ref. [38].

CUDA extends existing languages and requires dedicated compilers (`nvcc` for C/C++ and `nvfortran` for Fortran). While it allows for the optimal use of Nvidia GPUs, it is not portable and cannot be used for GPUs produced by other vendors. However, there have been attempts to provide abstraction layers or conversion tools for other approaches to be able to run CUDA code via OpenCL [39, 40]. There are also a variety of libraries that automatically offload compute-intensive operations to the GPUs. Examples include libraries such as cuBLAS for linear algebra and cuFFT for fast-fourier transforms [41].

Competing approaches including OpenACC [42] and OpenCL are available but have not been as widely adopted so far. For the parallel GPU implementation presented in this paper, we have chosen CUDA because it is the de-facto standard for GPU-accelerated code and is widely supported. Our attempts to develop a GPU-accelerated solution using OpenMP were not successful so far due to offloading support in OpenMP still being in an early stage of development.

GPU Implementation

In this section, implementation details of the Kalman filter used to run track fitting on both CPUs and GPUs are presented. The code can be found in Ref. [43].

Parallelization Strategy

As discussed in Sect. 2, track fitting is the step in the track reconstruction chain that precisely estimates the reconstructed track parameters and the associated covariance matrices. If track fitting is performed sequentially in a single event, the execution time will increase almost linearly with increasing track multiplicity. However, the dependence of the track fitting execution time on the number of tracks weakens if the track fitting can be parallelized. The

implementation of a track-level parallel strategy is straightforward, since the track fitting for each reconstructed track is completely independent. In addition, the algorithm can be parallelized *within* a track fit, i.e. intra-track parallelization. Possible gains come from the matrix operations, e.g. the transportation of the track parameters in a magnetic field and the Kalman filter update and smoothing, which are computationally expensive and have to be repeated for all the propagation steps and measurements by a total of up to $\mathcal{O}(10)$ times per track. However, in practice only very limited intra-track parallelization for those matrix operations can be achieved by using multiple threads, because the sizes of the matrices in one operation are usually relatively small. For example, the largest size matrix operated on in a single track fit in ACTS is the covariance matrix of the track parameters represented in the global coordinate system, which is of size 8×8.

This paper discusses both parallelization strategies for track fitting on GPUs:

1 Track-level parallelization: Track fitting for different tracks is executed in parallel using different CUDA threads (or blocks if further intra-track parallelization is used).
2 Intra-track parallelization: The matrix operations involved in a single track fit are parallelized as much as possible using multiple threads within a single CUDA block. In this case, the block shared memory is used for the objects relevant with one track fit.

The transportation of the track parameters and their associated covariance matrices in a magnetic field requires a numerical solution to the equation of particle motion. The adaptive Runge-Kutta-Nyström [44] method is used to transport the track parameters in ACTS. When extrapolating the track parameters from one measurement point to the next, the covariance of the track parameters is updated with the transport Jacobian between the measurement points. Because the track parameters are represented in the local coordinate frame of the detector, the transform Jacobians between local and global track parameters at the two measurement points have to be applied. If the fitting is performed using one CUDA block per track, the matrix multiplication for the covariance transport can be parallelized using multiple threads.

CUDA Considerations and Limitations

Various CUDA programming requirements have consequences for the problem-specific factors that shape the parallelization strategies, and thus have an impact on the final implementation. The most challenging ones are detailed next.

Polymorphism

Virtual functions cannot be called inside a CUDA kernel unless the objects are constructed there. The Curiously Recurring Template Pattern (CRTP) is a C++ design pattern that emulates the behaviour of dynamic polymorphism through having a base class which is a template specialization for the derived class itself.

The ACTS Kalman filter is designed to be independent of the detector's tracking geometry, which could contain surfaces of different concrete types for different tracking detectors. To realize this design pattern on accelerators, the surfaces are implemented with CRTP, instantiated outside of the Kalman filter, and fed to the algorithm. CRTP is successfully used to define the surfaces as shown in the code sample in Listing 1.

```
template <typename Derived>
inline const typename Derived::SurfaceBoundsType
    *Surface ::bounds() const {
  return static_cast<const Derived *>(this)->bounds();
}
```

Listing 1 Function definition in Surface base class using CRTP

Thread Memory Limitations

The amount of memory available for a thread, which includes the stack frame size and the maximum number of registers, is automatically configured by the CUDA runtime environment based on device properties including the total amount of shared memory and the cache sizes, and also on the number of parallel threads per execution block. Because the GPU's performance gain is based on the ability to run thousands of threads in parallel, this limits considerably the amount of memory available per thread compared to the CPU. This memory limitation is a major concern for recursive functions, which have to be reimplemented using an iterative approach.

Limited Support for Linear Algebra Libraries

In the Kalman smoothing, the track parameter covariance matrix needs to be inverted to calculate the gain matrix. While the Eigen-based matrix class has a member function that returns the inverse matrix, this method is not supported in the GPU kernels. Also CUDA 10.0 discontinued the support for invoking cuBLAS functions from within the device kernels through cublas device routine [45]. Since at the moment neither of these two linear algebra

libraries provides a solution for our scenario, a customized matrix inverter is implemented whose performance impact is discussed in Sect. 5.

Precision and Rounding

Heterogeneous resources produce slightly different results due to different approaches to floating-point arithmetic and rounding. CPUs typically promote float operands to doubles when possible, perform the operations in double precision and then truncate the result to simple precision. Moreover, the x86 floating-point units use extended double-precision registers (80-bit), while CUDA limits the register sizes to 32-bit and 64-bit as described by the IEEE standard 754-2008 [46, 47].

To mitigate these effects, the customized matrix inverter is always configured to perform the algebra operations using double precision. Floating-point precision for other algebra operations is used for benchmarking the performance, and the comparison between the results with float and double precision is discussed in Sect. 5.3.

Data Structure and Transfer

Detector geometry information is a fundamental component for track parameter propagation and the integration of material effects during the track reconstruction. Because track reconstruction with a realistic detector description as used in full detector simulation requires significant computational resources, a simplified detector geometry, the so-called *tracking geometry*, is used during the track reconstruction in ACTS for fast navigation and extrapolation of tracks. The basic geometrical component of the tracking geometry in ACTS is the surface. The surface object carries information about its geometrical orientation, shape and boundary, the material approximated from a full detector geometry, and a unique hierarchical geometry identifier.

Magnetic fields are used for measuring the momentum of charged particles at high-energy physics experiments. When a charged particle passes through a magnetic field, it bends with the degree of bending inversely proportional to its momentum. In this paper, a constant magnetic field represented as an Eigen matrix of size 3×1 is used for the performance studies presented. It should be noted that this represents a significant simplification with respect to an inhomogeneous magnetic field, as found in many experiments, for which position-specific field information may need to be stored and retrieved, i.e. more memory might be required. The description of inhomogeneous or nonparametric magnetic fields however is also possible within ACTS.

The ACTS tracking Event Data Model (EDM) includes classes to describe tracking objects such as measurements and track parameters, which are represented with surfaces. The EDM for the track state is designed for the Kalman filter. It consists of a measurement, a predicted track parameter, a filtered track parameter and a smoothed track parameter, all located on a surface. Figure 3 shows a track state on a surface. During the construction of the detector geometry, a unique geometrical identifier is assigned to each detector surface. By storing the geometrical identifier in the tracking EDM, information about the geometry can be shared between the measurements and track parameters, and between the CPU and GPU.

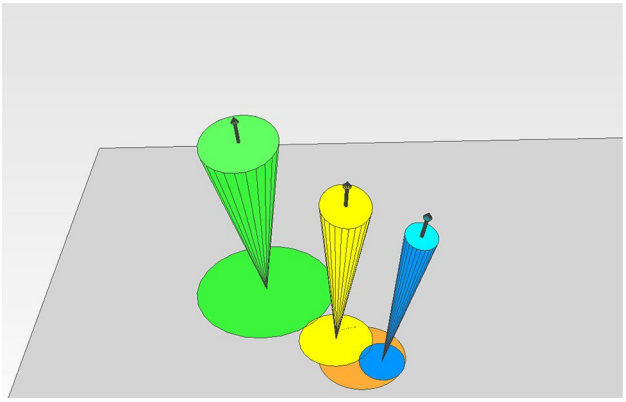

Fig. 3 Demonstration of a track state with a measurement (orange), a predicted track parameter (green), a filtered track parameter (yellow) and a smoothed track parameter (blue) on a plane surface (gray). The covariance matrix of the local coordinates for both the measurement and the fitted track parameters is represented by an ellipse. The momentum direction of the fitted track parameters is represented by an arrow with its covariance matrix represented by a cone oriented in the direction of the arrow

The Kalman filter, detector geometry and magnetic field are global data shared by different tracks during the track reconstruction, and are, therefore, stored in the kernel global memory. In addition, four sets of track-specific data are required to execute the track fitting kernel on the GPU:

1. Input measurements of the trajectory.
2. Starting track parameters to steer the track parameters propagation.
3. Track fitting configurations, e.g. a target surface to extract the fitted track parameters.
4. Fitted results including the fitting status, a collection of track states on the trajectory and the fitted track parameters on the target surface.

Track-specific data are allocated on the pinned memory on the host, i.e. page-locked memory, and this memory is allocated contiguously for each track. The number of detector surfaces intersected by the particle varies with the kinematics of the particle and the detector layout, i.e. the data loads

Table 1 The size (in bytes) of track-specific memory for a single track

Data type	Size (B)
Input measurements	280
Seeding parameters	168
Fitting configurations	144
Fitting status	1
Fitted states	8480
Fitted track parameters	216
Total	9289

are different between tracks. Managing this load imbalance would require dedicated memory management and task scheduling strategies. In this paper, the detector surfaces are therefore constructed to be boundless[2] to guarantee that the same number of surfaces are traversed by the tracks. Each surface object requires approximately 120 bytes of memory. In addition to that, the size of memory allocated and transferred for different track-specific data for a detector with 10 plane surfaces is summarized in Table 1. Considering the relatively large size of the track-specific data, and the fact that they are accessed only once during the track fitting, those data are also stored in the kernel global memory.

Performance Evaluation

The software performance is studied using a simple telescope-like detector geometry with 10 planar surfaces perpendicular to the global x axis and placed equidistantly with 30 mm between two adjacent planes. Realistic HEP track reconstruction applications typically involve a more complicated detector geometry. A constant magnetic field of 2 T along the global z axis is used. Samples containing a single muon per event are used for the performance evaluation. Muons are used for this study because they interact only minimally with the detector material and thus high-quality track fits are expected. The muons have a transverse momentum uniformly distributed between 1 and 10 GeV with both the azimuthal angle and polar angle fixed to zero.

The Fatras fast simulation engine [48] within the ACTS toolkit is used to generate simulated hits of the single muons on the detector surfaces. Figure 4 illustrates the simulated hits on the detector surfaces for a sample of 10,000 single muons. The dense tracking environment at the HL-LHC is not expected to exceed 10,000 tracks per event.

[2] Boundless surfaces always have an intersection with a track as long as the track is not parallel to the surface.

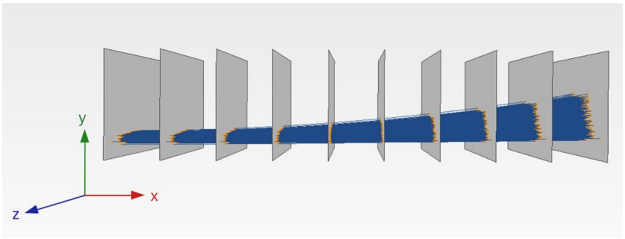

Fig. 4 The detector configuration used to study the performance. It consists of 10 identical planar surfaces (gray planes) perpendicular to the x-axis. The trajectories of 10,000 simulated single muons (blue lines) and the associated simulated hits on the detector surfaces (orange dots) are indicated

As the pattern recognition algorithms required to find track candidates from measurements are beyond the scope of this paper, the known trajectories of the simulated particles are used for the fitting in place of track candidates provided by a pattern recognition step. The measurements corresponding to the track candidates are obtained by smearing the positions of the simulated hits with Gaussian distributions to model detector resolution effects. A resolution of 50 μm in both the x and y dimensions of the detector, representative of the resolution of current pixel detectors at the LHC, is used. The initial set of track parameters for the track fit is based on the simulated particle vertex and momentum, smeared by Gaussian noise.

Hardware and Software Environment

Computing nodes of two supercomputers are used for running the performance tests:

1. Cori Intel Xeon Haswell node, Cori Intel Xeon Phi Knight's Landing (KNL) node, and Cori GPU node at the National Energy Research Scientific Computing Center (NERSC) [49].
2. Intel Xeon Skylake (SL) node and ATLAS-GPU01 node at the National Analysis Facility (NAF) at DESY.

Only GPUs from the Nvidia Tesla family are studied. All nodes use the CentOS 7 operating system and all GPUs are using CUDA version 10.2.89. Tables 2 and 3 show the detailed hardware and software configurations of the systems for CPUs and GPUs respectively.

These machines cover a wide range of architectures, with Cori-Haswell-CPU representing a standard compute node with two processors and a moderate amount of cores (for a total of 64 threads), while Cori-KNL-CPU features a higher core count due to the Xeon Phi's particular architecture (for a total of 272 threads). Moreover, NAF-P100-GPU and NAF-V100-GPU allow two different GPU generations

Table 2 CPU configurations

Sys.	Model name	S×C×T	Clock rate (GHz)	Mem. (GB)
1	Intel Xeon E5-2698 v3 (Cori-Haswell-CPU)	2×16×2	2.30	128
1	Intel Xeon Phi 7250 (Cori-KNL-CPU)	1×68×4	1.40	96
2	Intel Xeon Gold 5115 (NAF-SL-CPU)	2×10×2	2.40	376

The Sys. column specifies whether the CPUs are used in the NERSC (1) or NAF (2) system. The S×C×T column represents the number of sockets (S), cores per socket (C) and threads per core (T)

Table 3 GPU configurations

Sys.	GPU	FP32 cores	FP64 cores	Clock rate (GHz)	Mem. (GB)
1	GV100-SXM2 (Cori-V100-GPU)	5120	2560	1.53	16
2	GP100-PCIe (NAF-P100-GPU)	3584	1792	1.48	16
2	GV100-SXM2 (NAF-V100-GPU)	5120	2560	1.53	32

The Sys. column specifies whether the GPUs are used in the NERSC (1) or NAF (2) system. FP32 and FP64 columns denote the numbers of floating point compute units for single and double precision arithmetic operations, respectively

to be compared, with NAF-P100-GPU being from the older Pascal architecture and NAF-V100-GPU belonging to the newer Volta architecture. The NAF-P100-GPU is connected through a Peripheral Component Interconnect Express (PCIe) serial connector while the NAF-V100-GPU uses the SXM2 connector, a multi-line serial connector that provides both Nvidia NVLink and PCIe connectivity [50]. Cori-V100-GPU is identical to NAF-V100-GPU except for the lower amount of main memory. Each of the GPUs contains multiple Streaming Multiprocessors (SMs), which are similar to CPUs. However, each GPU is equipped with a large number of SMs, specifically, up to 60 SMs for the P100 and up to 84 SMs for the V100. Each SM contains many CUDA Cores, which execute compute kernels in the form of threads (64 single precision/32 double precision cores per SM for both P100 and V100). Each warp consists of 32 threads, with each warp running on one SM and each SM being able to execute up to 64 warps simultaneously. Due to a large number of total cores, it is important to distribute work across a sufficient number of warps. This allows the dedicated warp schedulers to achieve maximum utilization by keeping the cores busy with instructions. While on P100 all threads share a single program counter (and therefore have to execute the same instruction at the same time), V100 manages execution state per-thread, allowing more independence.

Tracking Performance

The resolution of the Kalman filter-based track fit is validated by calculating the pulls of the track parameters as follows:

$$v_{\text{pull}} = \frac{v_{\text{fit}} - v_{\text{truth}}}{\sigma_{\text{v}}}, \tag{1}$$

where v_{fit} and σ_{v} are the value and uncertainty of the fitted track parameter respectively, and v_{truth} is the corresponding parameter for the simulated particle.

The pull distributions of the six fitted perigee track parameters for a simulated sample of 10,000 single muons obtained from Cori-V100-GPU are shown in Fig. 5. The pull distributions have means compatible with zero and widths compatible with one, which demonstrates that the track parameters and their uncertainties are estimated correctly by the track fit.

Computing Performance

The timing performance of the track fitting for different number of tracks on various computing architectures and configurations is measured. Ten tests are run for each measurement. The mean time of the ten tests is taken as the measurement result, and square root of the average of their squared deviations from the mean is taken as the measurement error.

The baseline tests are performed using only track-level parallelization, i.e. without intra-track parallelization using shared memory. CUDA supports user configuration of the runtime properties of the GPU kernels, e.g. grid size and block size, and the launching of multiple kernels with multiple CUDA streams. Furthermore, the number of registers per thread can be controlled via the CUDA `__launch_bounds__` qualifier when the kernel is defined. Unless explicitly specified, the tests are performed using one CUDA stream with 255 registers per thread as default

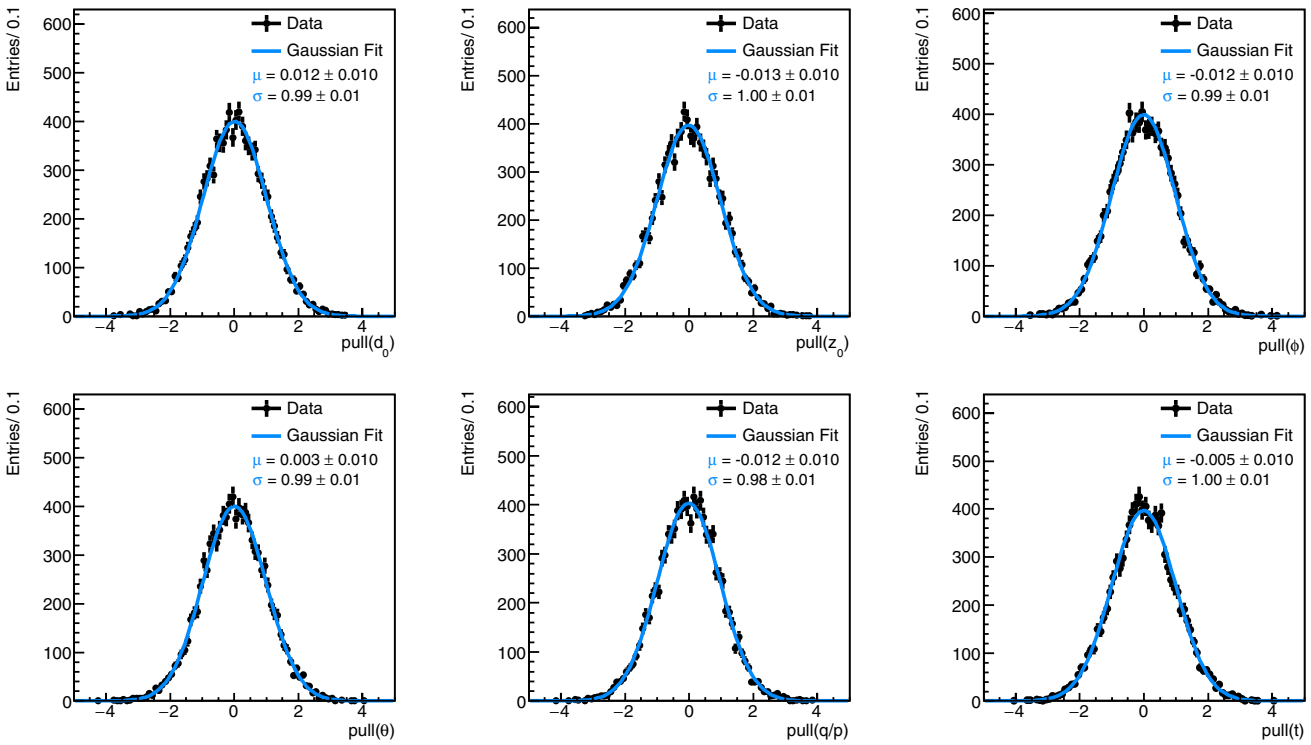

Fig. 5 The distributions of the pull values of the fitted perigee track parameters, $(d_0, z_0, \phi, \theta, \frac{q}{p}, t)$, for a sample of 10,000 single muons obtained from Cori-V100-GPU. The black dots are the pull values, and the blue lines are Gaussian fits to the distributions

configuration [50]. Moreover, we choose a grid size of 5120×1 to match the number of processing units on the V100 GPU, and a block size of 8×8×1, which is also the largest size of the matrix dealt with by the Kalman filter in ACTS. Performance with intra-track parallelization and different CUDA configurations are also studied for comparison.

All the tests use single-precision arithmetic operations with the exception of (a) the matrix inversion algorithm required by the smoother as detailed in Sect. 4.2.3, and (b) the explicit scenario which compares the different precisions described in Sect. 5.3.4.

A singularity container with the executable and the required dependencies used to produce the results presented here is accessible in Ref. [43].

Performance of the Custom Matrix Inversion Algorithm

Because the Eigen-based matrix inversion algorithm used by ACTS cannot be called inside CUDA kernels, a custom algorithm for matrix inversion implemented for this purpose is used. Measurements are performed to compare our custom implementation to the Eigen-based implementation on the CPUs of both systems. As shown in Fig. 6, it adds additional time to the fitting when the number of tracks exceeds 100. While the Eigen-based implementation is significantly faster by a constant factor when using only one thread, this effect is much less pronounced when using as many threads. Improving the performance of the custom matrix inversion operations on GPUs to match

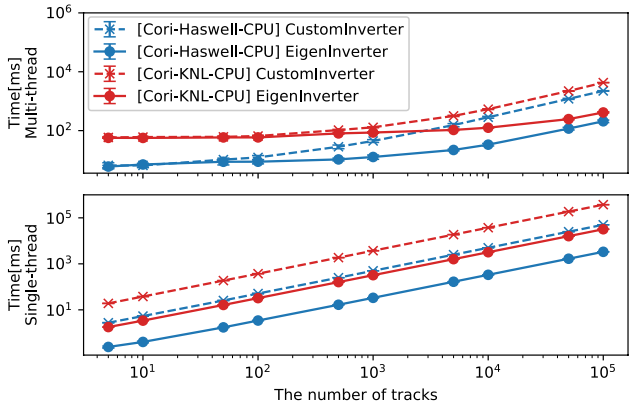

Fig. 6 The fitting time as a function of the number of tracks with different matrix inversion algorithms on Cori-Haswell-CPU (dashed blue for custom matrix inversion, and solid blue for Eigen-based matrix inversion) and Cori-KNL-CPU (dashed red for custom matrix inversion, and solid red for Eigen-based matrix inversion). The top panel shows the results with 60 and 250 threads on Cori-Haswell-CPU and Cori-KNL-CPU, respectively, and the bottom panel shows the results with a single thread

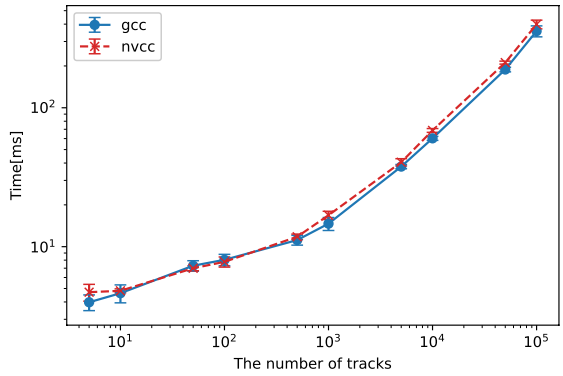

Fig. 7 The fitting time as a function of the number of tracks on NAF-SL-CPU using 60 threads with Eigen-based matrix inversion obtained with the `gcc` (solid blue) and `nvcc` (dashed red) compiled executables, respectively

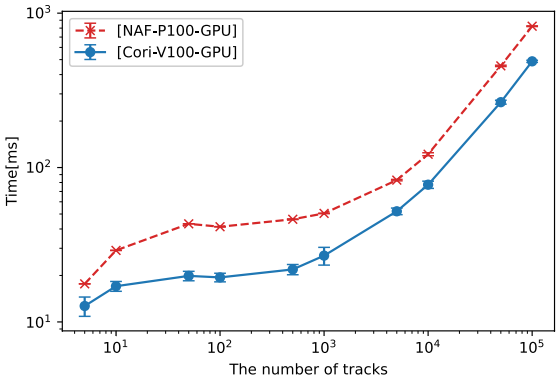

Fig. 8 The fitting time as a function of the number of tracks on NAF-P100-GPU (dashed red) and Cori-V100-GPU (solid blue)

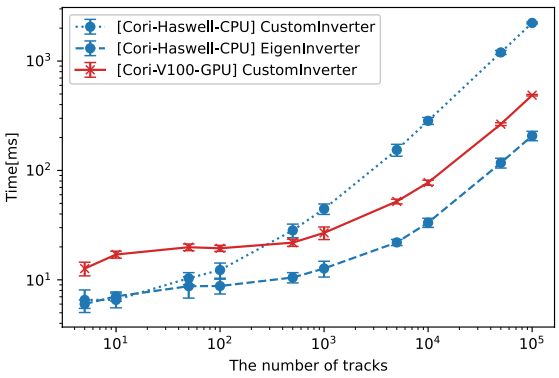

Fig. 9 The fitting time as a function of the number of tracks on Cori-Haswell-CPU using 60 threads with Eigen-based matrix inversion (dashed blue) and custom matrix inversion (dotted blue), and on Cori-V100-GPU (solid red)

specialized linear algebra libraries would be expected to further improve the performance.

Performance of CUDA Code on CPU

The GPU-based track fitting program compiled with `nvcc` can also run on CPUs (although the CUDA driver and CUDA runtime need to be accessible to successfully allocate page-locked memory on the host). In this case, the track-level parallelization is achieved by using OpenMP threads and the host-device model including memory and execution offloading is bypassed. The `nvcc` compiler uses the host compiler (`gcc` in this case) to generate the executable. This approach ensures comparable execution time when the number of tracks is small but it induces a small performance penalty (between 4 % and 18 %) compared with the standard C++ implementation compiled with OpenMP support when the number of tracks exceeds 1000, as shown in Fig. 7.

Nevertheless, it demonstrates the potential for single-source code targeting heterogeneous hardware resources. This is especially important for large and long-running software projects that might be used on different hardware architectures. However, this still requires the code to be written using CUDA and implies the portability limitations discussed in Sect. 3.2.

Performance Comparison Between GPU Architectures

Performance of the Kalman filter-based track fitting on the P100 and V100 Nvidia Tesla cards, is compared. Figure 8 shows that fitting time on NAF-P100-GPU is more than a factor of two longer than on Cori-V100-GPU, therefore the following tests will focus on the V100 due to its superior performance. These performance differences are expected to a certain extent because the P100 and the V100 come from different hardware generations, with the V100 generally featuring more cores, higher clock rates and an improved interconnect.

Performance Comparison Between CPU and GPU

Figure 9 compares the fitting time on Cori-V100-GPU with that on Cori-Haswell-CPU using different approaches to the matrix inversion. As mentioned previously, only the custom matrix inversion is possible with Cori-V100-GPU. When considering the custom matrix inverter, Cori-V100-GPU displays superior performance compared to Cori-Haswell-CPU when the number of tracks exceeds 1000. However, the Eigen-based implementation on CPUs still outperforms our custom inverter on GPUs, demonstrating that it is important to not only consider the potential benefits from porting the actual code to GPUs but to also take supporting libraries into account.

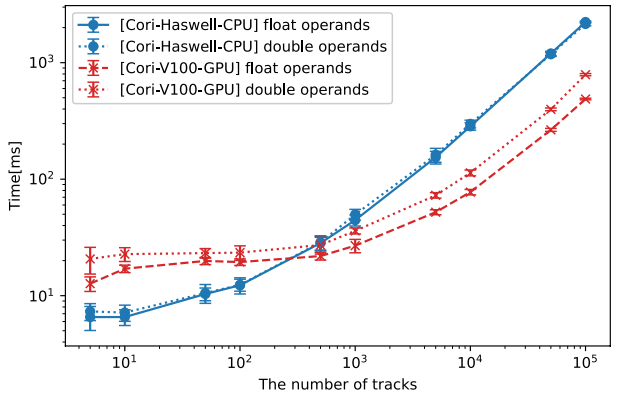

Fig. 10 The fitting time as a function of the number of tracks using `float` and `double` operands on Cori-Haswell-CPU (solid blue for float, and dotted blue for double) and Cori-V100-GPU (dashed red for float, and dotted blue for double). The tests are performed using the custom matrix inversion and the ones executed on Cori-Haswell-CPU are using 60 OpenMP threads

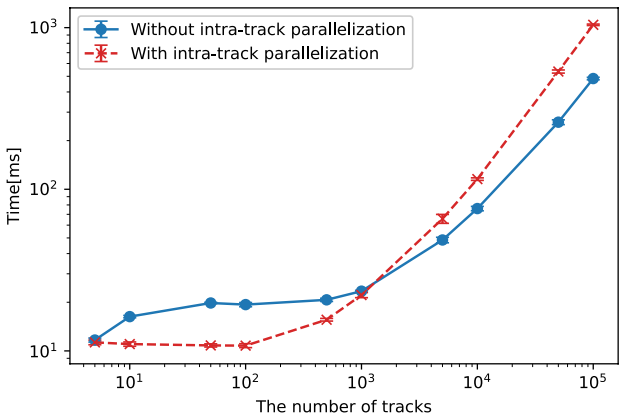

Fig. 11 The fitting time as a function of the number of tracks with linear grid of size 100,000×1 with (dashed red) or without (solid blue) intra-track parallelization on Cori-V100-GPU

Our matrix inversion algorithm performs all the operations in double precision regardless of the operands' types. Figure 10 shows that there is little variation in performance between the different operands' types: when running the code on Cori-Haswell-CPU, there is virtually no performance difference for more than 1000 tracks, while double operands are slightly slower on Cori-V100-GPU.

As a consequence of parallel execution, the fitting time per track varies inversely with the number of tracks for all the considered platforms. In particular, for an HL-LHC scenario with up to 10,000 tracks, both the current prototype on GPUs and the Eigen-based implementation on CPUs show a fitting time per track in the order of microseconds.

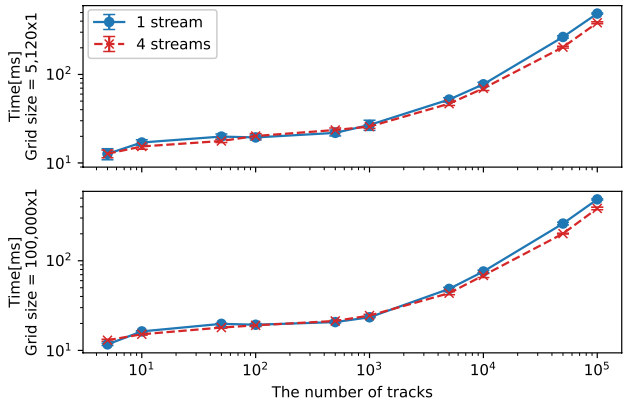

Fig. 12 The fitting time as a function of the number of tracks with linear grids of sizes 5120×1 (top) and 100,000×1 (bottom), with one (solid blue) or four (dashed red) streams per device on Cori-V100-GPU

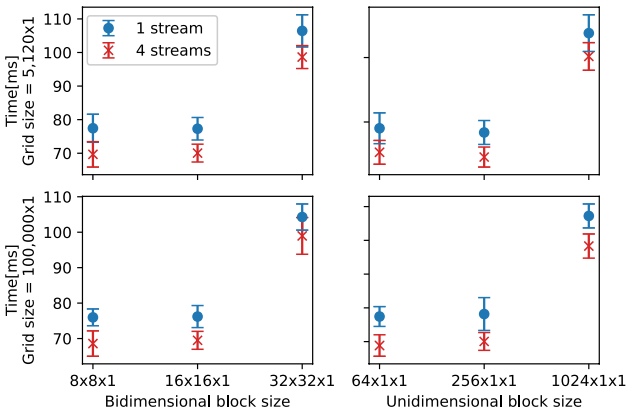

Fig. 13 The fitting time for 10,000 tracks as a function of bidimensional (left) or unidimensional (right) block sizes with linear grids of sizes 5120×1 (top) and 100,000×1 (bottom), with one (blue) or four (red) streams per device on Cori-V100-GPU

Performance with Different GPU Configurations

The impact of usage of intra-track parallelization with shared memory, and variations in number of streams per device, grid size and block size as well as the number of fitted tracks are investigated and the most important results are discussed next.

Figure 11 shows that performance gain could be achieved by using intra-track parallelization with shared memory when the number of tracks is below 1000. However, when the number of tracks exceeds 1000, using intra-track parallelization results in a performance penalty due to limitation of the available shared memory and resident threads per SM.

Figure 12 shows that no significant performance gain is obtained from using more streams per device.

Figure 13 shows the required wall clock time with different grid and block sizes for 10,000 tracks. Note that when

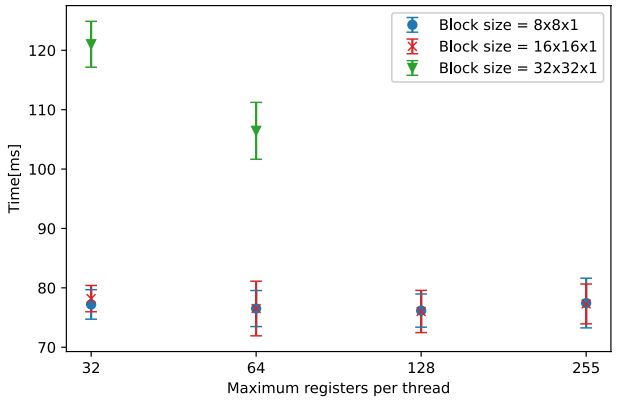

Fig. 14 The fitting time for 10,000 tracks using various number of registers per thread and block sizes on Cori-V100-GPU node. The circle blue, cross red and triangle green represent block sizes of 8×8×1, 16×16×1 and 32×32×1, respectively. When there are 1024 threads per block, a maximum of 64 registers per thread is allowed

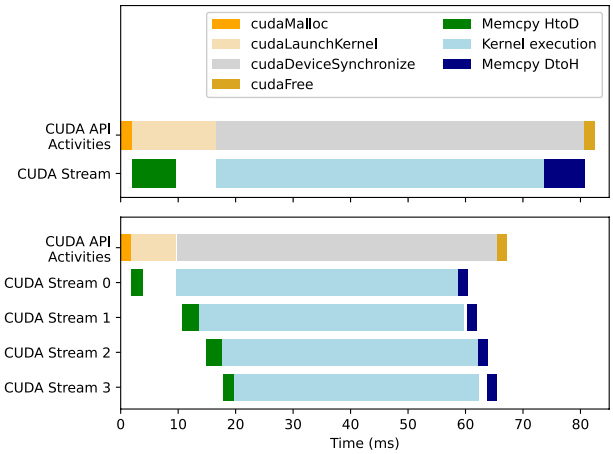

Fig. 15 The timeline of CUDA activities for offloading track fitting for 10,000 tracks to Cori-V100-GPU with one stream (top) and four streams (bottom). The starting times of memory allocation on GPU are taken as the 0 point of the timelines

there are 1024 threads per block, the CUDA `__launch_bounds__` must be specified with the maximum number of threads per block no less than 1024 to avoid "too many resources requested for launch" errors. This results in a maximum of 64 registers per thread at Cori-V100-GPU with 65,536 32-bit registers per SM. While the performance differences between the two grid sizes are negligible, larger block sizes increase the runtime by up to a factor of 1.5. These results show that it is important to choose block sizes that are appropriate for the underlying hardware, i.e. when the overall workload is not large enough to saturate all the SMs on the GPU, a relatively large block size could lead to further imbalance of the workload distributed to the SMs and hence compromise the performance.

The number of registers per thread can be reduced with the goal of running more threads per block without exceeding the hardware limits (i.e. the maximum number of blocks per SM). Figure 14 shows how a variation in the number of registers per thread affects the overall performance for a track fitting workload of 10,000 tracks. The performance varies little with the number of registers per thread. This is because the number of resident threads on the SMs are not increased by reducing the number of registers per thread when the overall workload is small. Increasing the block size can enforce at least the same amount of threads as the block size resident on some SMs, but this could lead to unwanted performance compromise due to inefficient GPU utilization. See Sect. 6 for further discussion.

The different components of the fitting time are analysed using Nsight Systems, one of Nvidia's new performance analysis tools [51]. Figure 15 shows the timeline of those CUDA API activities required for GPU offloading, including the memory allocation on GPU, kernel launching, device synchronization and memory deallocation on the GPU,

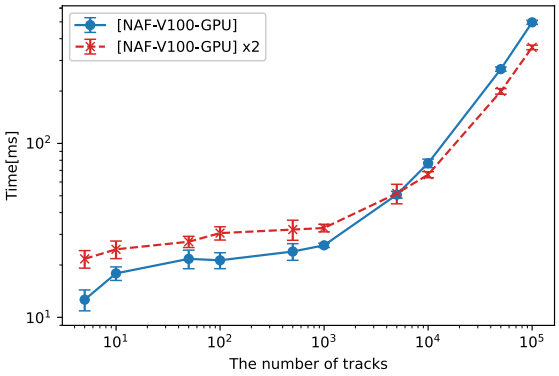

Fig. 16 The fitting time as a function of the number of tracks executed in one stream per device with linear grid of size 5120×1 and block size of 8×8×1, when using one NAF-V100-GPU (solid blue) and two NAF-V100-GPU in parallel (dashed red)

and the timeline of memory transfer and kernel execution in either one CUDA stream or multiple streams for 10,000 tracks. When using only a single stream, kernel execution accounts for roughly 70% of the total runtime, and memory transfer accounts for roughly 17% with significant impact on the performance. The performance gain from overlapping the data transfer and kernel execution with multiple streams is limited by the memory transfer time. A dedicated CUDA synchronization method might improve this.

Discussion

The timing performance studies in Sect. 5 use a single GPU. Potential gains in the timing performance by utilizing multiple GPUs and the GPU occupancy (which could have

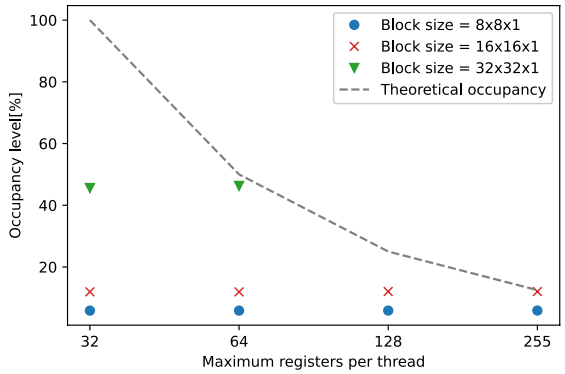

Fig. 17 Warp occupancy levels using various number of registers per thread and block sizes for fitting 10,000 tracks executed in one stream with linear grid of size 5120×1 on Cori-V100-GPU. The black dashed line is the theoretical warp occupancy, and the circle blue, cross red and triangle green represent the achieved warp occupancy with block sizes of 8×8×1, 16×16×1 and 32×32×1, respectively. When there are 1024 threads per block, a maximum of 64 registers per thread is allowed

an impact on the timing performance) are investigated in addition.

While the performance of the GPU-based Kalman filter was tested on one GPU, the implementation can also run on multiple GPUs in parallel. The prototype uses a team of threads on the host, each one fitting the trajectory of a subset of tracks on a different GPU. Despite better problem size scaling, the multi-device solution has a slightly larger overall execution time for smaller numbers of tracks, as shown in Fig. 16. Communication via Message Passing Interface (MPI) is required to fully exploit the parallelism by resolving synchronization overhead between the GPUs.

In addition, the latest versions of the Nvidia HPC Software Development Kit (SDK) provide new tools and libraries designed to maximize performance by optimizing memory transfers and scaling to multiple devices while targeting heterogeneous resources [52]. Additionally, various studies regarding vendor-agnostic offloading approaches show promising results based on standard APIs and/or open-source, non-proprietary solutions [53, 54]. These would be very interesting to explore in future iterations.

The warp occupancy, defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM (e.g. 64 warps per SM on the V100), is analyzed using a different Nvidia performance tool: Nsight Compute [51]. Figure 17 shows the dependency of the theoretical warp occupancy and the achieved warp occupancy on the number of registers per thread and block size for a track fitting workload of 10,000 tracks. The theoretical occupancy is 100% when the number of registers per thread is no larger than 32 and decreases when more registers are required for one thread, hence fewer threads are active. Since the maximum number of thread blocks per SM is 32 on the V100, the block size is not a limiting factor to theoretical occupancy as long as it is no less than 64.

The achieved occupancy is well below the theoretical one, in particular when the block size is small. The reason is that an average of only 119 tracks are distributed to each SM of Cori-V100-GPU, providing a total workload of 10,000 tracks. Therefore, only 2 blocks are resident on the SM when the block size is 8×8×1 (128 threads), and at most 1 resident block when the block size is 16×16×1 (256 threads) or 32×32×1 (1024 threads). These correspond to an occupancy of 6.25%, 12.5%, and 50%, respectively. In this case, reducing the number of registers per thread has no impact on the warp occupancy. Event-level parallelization is an effective approach to increase the workload, and hence improve the SM warp occupancy. This can be achieved by an offloading pattern with a fully contained chain of tracking modules that run on GPU requiring minimum data transfer between CPU and GPU [18]. The workload also needs to be accounted for when analyzing the impact of SM warp occupancy on the performance. For instance, better warp occupancy does not necessarily correspond to better timing performance, as shown in Fig. 14, for the particular track fitting workload of 10,000 tracks studied here. Further discrepancy between the achieved occupancy and the theoretical one arises from the imbalanced track fitting workload both within blocks and across blocks due to different momentum hence propagation paths between tracks. In this paper, the workload imbalances were already controlled by using a homogeneous detector geometry for all the tracks. For a realistic detector, the concept of tracking regions as presented in Ref. [8, 9] can be used for the parallelization of track reconstruction. In the case of track finding, there is additional workload imbalance from the selection of compatible measurements from a pool of non-static number of measurements on a detector surface and possibly splitting the track propagation into multiple branches if there are more than one compatible measurement found. One possible approach to suppress the workload imbalance level would be to group the tracks based on their kinematic properties so that one group of tracks will encounter the same segmented detector region, and assign different groups of tracks to different grids or blocks. See Ref. [8, 9] for further discussion.

Conclusion

The reconstruction of charged particle trajectories for current and future high-energy physics experiments is a significant computational challenge. New approaches are needed to cope with the dramatically increased event complexity and rates and with the movement away from x86 architectures. The Kalman filter algorithm is the mainstay

of current track reconstruction strategies. We presented a proof-of-concept implementation of a full Kalman filter track fitting algorithm using ACTS on two different Nvidia GPU architectures using a simplified detector geometry and a constant magnetic field. We have performed studies of its physics and technical performance and compared this to results using CPUs with a particular focus on the limitations observed. Ideas for improvements for future implementations were discussed.

As existing fast matrix inversion algorithms cannot run on GPUs, we developed a custom prototype matrix inversion algorithm, which does not match the CPU performance of highly-optimized state-of-the-art algorithms. When controlling for the matrix inversion algorithm, worse performance for low track multiplicity is obtained with the GPUs compared to the CPUs and the performance is improved by a factor of up to 4.6 with respect to the CPUs for events with more than 1000 tracks. Significant performance differences are shown between the different GPU architectures.

Parallelization within the track fit was implemented and performance gain was observed with a relatively low track multiplicity. The performance dependence on GPU configurations was also studied. The performance was largely independent of the grid size and did not change when using multiple kernels. Memory transfer and other overhead can account for up to 30% of the total run time for the track fit.

The typical HL-LHC track multiplicity of 10,000 tracks per event is a relatively small workload for GPUs. For events with 10,000 tracks, performance gain by up to a factor of 1.5 is achieved by using a smaller block size. The small workload is also the main limiting factor in the achieved occupancy on the GPU.

We have compared different methods for the Kalman filter implementation and studied the dependence on the GPU configuration. We have identified limitations of the approach and highlighted areas for future work directions.

Specifically, an evaluation of alternative approaches for GPU offloading, especially those provided by vendor-agnostic interfaces such as OpenMP, can be expected to result in improved performance portability. Moreover, further improvements to the GPU-based matrix inversion algorithm can be expected to bring its performance closer to existing CPU implementations.

Acknowledgements We would like to thank Dr. Attila Krasznahorkay (CERN), Dr. Charles Leggett (Lawrence Berkeley National Laboratory) and Dr. Andreas Salzburger (CERN) for their careful reading of the manuscript and their helpful comments and suggestions. This work used the grid computational resources operated at Deutsches Elektronen-Synchrotron (DESY), Hamburg, Germany and at the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We acknowledge the support by the National Science Foundation (NSF) and Data Science in Hamburg - HELMHOLTZ Graduate School for the Structure of Matter (DASHH).

Funding Open Access funding enabled and organized by Projekt DEAL. This work was funded by the NSF under Cooperative Agreement OAC-1836650, and supported by DASHH with the Grant-No. HIDSS-0002.

Availability of data and materials Not applicable. No associated data except for code.

Code availability The code used for this research (including a Singularity container for reproducibility) is available open source [43].

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Billoir P (1984) Track fitting with multiple scattering: a new method. Nucl Instrum Meth A 225:352–366. https://doi.org/10.1016/0167-5087(84)90274-6
2. Fruhwirth R (1987) Application of Kalman filtering to track and vertex fitting. Nucl Instrum Meth A 262:444–450. https://doi.org/10.1016/0168-9002(87)90887-4
3. Strandlie A, Frühwirth R (2010) Track and vertex reconstruction: from classical to adaptive methods. Rev Mod Phys 82:1419–1458. https://doi.org/10.1103/RevModPhys.82.1419
4. Moore GE (1965) Cramming more components onto integrated circuits. Electronics 38:8. https://doi.org/10.1109/JPROC.1998.658762
5. Shalf J (2020) The future of computing beyond Moore's Law. Philos Trans Roy Soc A 378:2166. https://doi.org/10.1098/rsta.2019.0061
6. Cerati G et al (2014) Traditional tracking with Kalman filter on parallel architectures. https://arxiv.org/abs/1409.8213
7. Cerati G et al (2017) Parallelized Kalman-filter-based reconstruction of particle tracks on many-core processors and GPUs. EPJ Web Conf. 150, 00006 (2017). https://doi.org/10.1051/epjconf/201715000006
8. Cerati G et al (2020) Reconstruction of charged particle tracks in realistic detector geometry using a vectorized and parallelized kalman filter algorithm. EPJ Web Conf 245:02013. https://doi.org/10.1051/epjconf/202024502013

9. Lantz S et al (2020) Speeding up particle track reconstruction using a parallel Kalman filter algorithm. J Instrum 15(09):P09030. https://doi.org/10.1088/1748-0221/15/09/p09030
10. Kisel I (2018) for CBM Collaboration Event topology reconstruction in the CBM experiment. J Phys Conf Ser 1070:012015. https://doi.org/10.1088/1742-6596/1070/1/012015
11. ALICE Collaboration (2008) The ALICE experiment at the CERN LHC. J Instrum 3(8):S08002. https://doi.org/10.1088/1748-0221/3/08/s08002
12. LHCb Collaboration (2008) The LHCb detector at the LHC. J Instrum 3(8):S08005. https://doi.org/10.1088/1748-0221/3/08/s08005
13. Rohr D, Gorbunov S, Schmidt MO, Shahoyan R (2018) Track reconstruction in the ALICE TPC using GPUs for LHC Run 3. https://arxiv.org/abs/1811.11481
14. Rohr D, Gorbunov S, Ole Marten S, Shahoyan R (2019) GPU-based online track reconstruction for the ALICE TPC in run 3 with continuous read-out. EPJ Web Conf 214:01050. https://doi.org/10.1051/epjconf/201921401050
15. Aaij R et al (2020) Allen: a high-level trigger on GPUs for LHCb. Comput Softw Big Sci 4(1):7. https://doi.org/10.1007/s41781-020-00039-7
16. Funke D, Hauth T, Innocente V, Quast G, Sanders P, Schieferdecker D (2014) Parallel track reconstruction in CMS using the cellular automaton approach. J Phys Conf Ser 513(5):052010. https://doi.org/10.1088/1742-6596/513/5/052010
17. Rinaldi L, Belgiovine M, Sipio RD, Gabrielli A, Negrini M, Semeria F, Sidoti A, Tupputi SA, Villa M (2015) GPGPU for track finding in high energy physics. https://arxiv.org/abs/1507.03074
18. Bocci A, Kortelainen M, Innocente V, Pantaleo F, Rovere M (2020) Heterogeneous reconstruction of tracks and primary vertices with the CMS pixel tracker. https://arxiv.org/abs/2008.13461
19. vom Bruch D (2017) Online data reduction using track and vertex reconstruction on GPUs for the Mu3e experiment. EPJ Web Conf 150:00013. https://doi.org/10.1051/epjconf/201715000013
20. Sen P, Singhal V (2015) Event selection for MUCH of CBM experiment using GPU computing. In: 2015 Annual IEEE India conference (INDICON), pp 1–5. https://doi.org/10.1109/INDICON.2015.7443569
21. vom Bruch D (2020) Real-time data processing with GPUs in high energy physics. J Instrum 15(06):C06010. https://doi.org/10.1088/1748-0221/15/06/c06010
22. Huang MY, Wei SC, Huang B, Chang YL (2011) Accelerating the Kalman Filter on a GPU. In: 2011 IEEE 17th international conference on parallel and distributed systems, pp 1016–1020 (2011). https://doi.org/10.1109/ICPADS.2011.153
23. Xu D, Xiao Z, Li D, Wu F (2016) Optimization of parallel algorithm for Kalman filter on CPU-GPU heterogeneous system. In: 2016 12th international conference on natural computation, fuzzy systems and knowledge discovery (ICNC-FSKD), pp 2165–2172. https://doi.org/10.1109/FSKD.2016.7603516
24. Gumpert C, Salzburger A, Kiehn M, Hrdinka J, Calace N (2017) ACTS: from ATLAS software towards a common track reconstruction software. Tech. Rep. ATL-SOFT-PROC-2017-030. 4, CERN, Geneva (2017). https://doi.org/10.1088/1742-6596/898/4/042011
25. Ai X (2019) Acts: a common tracking software. In: Meeting of the division of particles and fields of the American Physical Society. https://arxiv.org/abs/1910.03128
26. Gessinger P, Grasland H, Gray H, Kiehn M, Klimpel F, Langenberg R, Salzburger A, Schlag B, Zhang J, Ai X (2020) The Acts project: track reconstruction software for HL-LHC and beyond. EPJ Web Conf 245:10003. https://doi.org/10.1051/epjconf/202024510003
27. Ai X (2020) Tracking with a common tracking software. https://arxiv.org/abs/2007.01239
28. Ai X, Allaire C, Calace N, Czirkos A, Ene I, Elsing M, Farkas R, Gagnon LG, Garg R, Gessinger P, Grasland H, Gray HM, Gumpert C, Hrdinka J, Huth B, Kiehn M, Klimpel F, Krasznahorkay A, Langenberg R, Leggett C, Niermann J, Osborn JD, Salzburger A, Schlag B, Tompkins L, Yamazaki T, Yeo B, Zhang J, Mania G, Kolbinger B, Moyse E, Rousseau D (2021) A common tracking software project. https://arxiv.org/abs/2106.13593
29. Kalman RE (1960) A new approach to linear filtering and prediction problems. J Basic Eng 82(1):35–45. https://doi.org/10.1115/1.3662552
30. Rauch HE, Tung F, Striebel CT (1965) Maximum likelihood estimates of linear dynamic systems. AIAA J 3(8):1445–1450. https://doi.org/10.2514/3.3166
31. ATLAS Collaboration (2008) The ATLAS Experiment at the CERN large Hadron Collider. JINST 3 (S08003):437. https://cds.cern.ch/record/1129811. Also published by CERN Geneva in 2010
32. Guennebaud G, Jacob B et al (2010) Eigen v3. http://eigen.tuxfamily.org
33. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55. https://doi.org/10.1109/99.660313
34. CUDA Toolkit Documentation (2021) https://docs.nvidia.com/cuda/index.html. Accessed 4 February 2021
35. Clark D (1998) OpenMP: a parallel standard for the masses. IEEE Concurr 6(1):10–12. https://doi.org/10.1109/4434.656771
36. Daley CS, Ahmed H, Williams S, Wright NJ (2020) A case study of porting HPGMG from CUDA to OpenMP target offload. In: Milfeld K, de Supinski BR, Koesterke L, Klinkenberg J (eds) OpenMP: portable multi-level parallelism on modern systems - 16th international workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings. Lecture notes in computer science, vol 12295, pp 37–51. Springer. https://doi.org/10.1007/978-3-030-58144-2_3
37. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. ACM Queue 6(2):40–53. https://doi.org/10.1145/1365490.1365500
38. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Online. Accessed 4 February 2021
39. Du P, Weber R, Luszczek P, Tomov S, Peterson GD, Dongarra JJ (2012) From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming. Parallel Comput 38(8):391–407. https://doi.org/10.1016/j.parco.2011.10.002
40. Babej M, Jääskeläinen P (2020) HIPCL: tool for porting CUDA applications to advanced OpenCL platforms through HIP. In: McIntosh-Smith S (ed) IWOCL '20: international workshop on OpenCL, ACM, Munich, Germany, April 27–29, 2020, pp 18:1–18:3. https://doi.org/10.1145/3388333.3388641
41. Fatica M (2008) CUDA toolkit and libraries. In: 2008 IEEE hot chips 20 symposium (HCS), pp 1–22. https://doi.org/10.1109/HOTCHIPS.2008.7476520
42. Herdman JA, Gaudin WP, Perks O, Beckingsale DA, Mallinson AC, Jarvis SA (2014) Achieving portability and performance through OpenACC. In: Chandrasekaran S, Foertter FS, Hernandez OR (eds) Proceedings of the first workshop on accelerator programming using directives, WACCPD '14, New Orleans, Louisiana, USA, November 16–21, pp 19–26. IEEE Computer Society. https://doi.org/10.1109/WACCPD.2014.10
43. Ai X, Mania G, Gray HM, Kuhn M, Styles N (2021) gpuKalmanFitter: v2.0. https://doi.org/10.5281/zenodo.4693389
44. Myrheim J, Bugge L (1979) A fast Runge-Kutta method for fitting tracks in a magnetic field. Nucl Instrum Meth 160(1), 43–48. https://doi.org/10.1016/0029-554X(79)90163-0

45. NVIDIA CUDA Toolkit v10.0.130 Release notes. https://docs.nvidia.com/cuda/archive/10.0/cuda-toolkit-release-notes/index.html#deprecated-features. Online. Accessed 4 February 2021
46. IEEE 754-2008 - IEEE standard for floating-point arithmetic (2008). https://standards.ieee.org/standard/754-2008.html
47. CUDA toolkit documentation - floating point and IEEE 754. https://docs.nvidia.com/cuda/floating-point/index.html. Online. Accessed 4 February 2021
48. Edmonds K, Fleischmann S, Lenz T, Magass C, Mechnich J, Salzburger A (2008) The fast ATLAS Track Simulation (FATRAS). Tech. Rep. ATL-SOFT-PUB-2008-001. ATL-COM-SOFT-2008-002, CERN, Geneva. https://cds.cern.ch/record/1091969
49. NERSC Cori System Specification. https://docs.nersc.gov/systems/cori/#system-specification. Online. Accessed 4 February 2021
50. NVIDIA Tesla V100 GPU Architecture (2017). https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Online. Accessed 4 February 2021
51. Knobloch M, Mohr B (2020) Tools for GPU computing debugging and performance analysis of heterogenous HPC applications. Supercomput Front Innov 7(1):91–111. https://doi.org/10.14529/jsfi200105
52. NVIDIA HPC Software Development Kit. https://developer.nvidia.com/hpc-sdk. Online. Accessed 4 February 2021
53. Deakin T, Poenaru A, Lin T, McIntosh-Smith S (2020) Tracking performance portability on the Yellow Brick Road to Exascale. In: 2020 IEEE/acm international workshop on performance, portability and productivity in HPC (P3HPC), pp 1–13. https://doi.org/10.1109/P3HPC51967.2020.00006
54. Gayatri R, Yang C, Kurth T, Deslippe J (2018) A case study for performance portability using OpenMP 4.5. In: Chandrasekaran S, Juckeland G, Wienke S (eds) Accelerator programming using directives—5th international workshop, WACCPD 2018, Dallas, TX, USA, November 11–17, 2018, Proceedings, Lecture notes in computer science, vol 11381, pp 75–95. Springer. https://doi.org/10.1007/978-3-030-12274-4_4