Computing and Software for Big Science (2025) 9:13
https://doi.org/10.1007/s41781-025-00143-6

RESEARCH

Performance Portability of the Particle Tracking Algorithm Using SYCL

Bartosz Soból[1,2] · Michael Papenbrock[3] · Tobias Stockmanns[4] · Grzegorz Korcyl[1]

Received: 20 December 2024 / Accepted: 18 June 2025
© The Author(s) 2025

Abstract
With fast advancements in detector and software technologies used in large-scale physics experiments, the requirements for the performance of computing systems used for both online and offline data processing have grown drastically. The industry offers a wide range of hardware devices to select from when designing such systems, but in turn, this imposes a technical challenge. To achieve efficient device utilization, deep knowledge of a particular technology is required. While one can achieve a high optimization level, it yields a hard-to-maintain codebase with limited upgradeability or capability to migrate to other platforms. It becomes a significant challenge, especially if available human resources are limited. In this paper, we present the application of the SYCL heterogeneous programming model that can help overcome those drawbacks. By the introduction of an abstraction layer, the source code is decoupled from the computing device architecture, and the developer can select the compilation target. The same codebase can, therefore, be executed on any supported hardware platform. We use the particle track reconstruction algorithm developed for the Forward Tracker in the $\overline{\text{P}}$ANDA experiment to demonstrate portability between various computing architectures and performance evaluation of the solution.

Keywords High-performance computing · Heterogeneous computing · Computing architectures · Particle tracking · Parallel computing

Introduction

Motivation

Large-scale particle physics experiments have become the pinnacle of technological advancements. Modern sensors register events with great precision and at high data rates. The complexity of particle physics interactions under interest often necessitates in-depth analysis and data selection. This cannot be achieved with low-level hardware signals and requires the processing of the complete data stream from detector components. Hence, the concept of continuous readout is more commonly implemented. The sensors with the readout electronics are transformed into free-streaming data sources, which places a high demand on the computing infrastructure.

One of the prominent examples is the ALICE experiment at the Large Hadron Collider facility. A raw data stream of up to 3TB/s is transferred to the computing cluster where the event reconstruction and selection happen on CPU and GPU-equipped nodes [1] using O^2[2] software platform. A comparable approach will be utilized for the CBM [3] and

✉ Bartosz Soból
bartosz.sobol@doctoral.uj.edu.pl

Michael Papenbrock
michael.papenbrock@physics.uu.se

Tobias Stockmanns
t.stockmanns@fz-juelich.de

Grzegorz Korcyl
grzegorz.korcyl@uj.edu.pl

1 Department of Information Technologies, Jagiellonian University, Prof. Łojasiewicza 11, Kraków, Poland

2 Doctoral School of Exact and Natural Sciences, Jagiellonian University, Prof. Łojasiewicza 11, Kraków, Poland

3 Department of Physics and Astronomy; Nuclear Physics, Ångströmlaboratoriet, Uppsala University, Regementsvägen 10, Uppsala, Sweden

4 Institut für Kernphysik 1, Forschungszentrum Jülich GmbH, Wilhelm-Johnen-Str., D-52428 Jülich, Germany

Published online: 01 July 2025

$\overline{\text{P}}$ANDA [4] detector systems currently under construction at FAIR [5, 6].

The examples cited above demonstrate a clear shift in modern detector systems' design and implementation of data processing mechanisms. Instead of designing dedicated electronics, often based on FPGA devices for real-time feature extraction, regular and commercially available hardware is deployed. Consequently, the challenge becomes developing software that will make the most out of the available hardware resources, which are often heterogeneous, including CPU, GPU, and FPGA accelerator cards. Compute nodes must execute an processing pipeline on a chunk of raw data to reconstruct a physical event and decide whether it should be transferred to storage or dropped. The software must be highly optimized and the available hardware features must be exploited to sustain the high data throughput. This in turn requires building a complex software stack using multiple toolchains, often vendor-specific, making it difficult to develop, maintain, upgrade, and reuse.

In this paper, we demonstrate the application of the SYCL heterogeneous programming model [7], which enables the development of a single-source, modern C++ software solution with the concept of kernels - functions that can be delegated to any supported computing device. We have developed the particle tracking algorithm for the Forward Tracker in the $\overline{\text{P}}$ANDA experiment [8, 9] based on the track road method [10] and assessed its performance on a wide range of hardware platforms. The same source code has been compiled and successfully executed on various CPUs and GPUs. The collected measurements allow us to estimate the overhead from the SYCL mechanisms, the performance of SYCL implementations, hardware architectures, and their combinations.

Outline

Numerous detector subsystems of the PANDA apparatus, including the Forward Tracker detector, are being developed and tested in-beam and in other already running experiment installations. A straw detector based on hardware developed for the Forward Tracker was successfully incorporated at the HADES experiment [11]. Simulation and many algorithms are also established, and data rates in the expected experiment environment are well-defined. The track reconstruction algorithm discussed includes matching track segments in magnetic field and non-magnetic field regions, and is therefore highly relevant for other experiments with straw-based tracking detectors.

The paper is structured as follows: after a general introduction and motivation, we describe the particular case of the gaseous straw detector - the Forward Tracker stations of the $\overline{\text{P}}$ANDA experiment. It is followed by a detailed overview of the particle track reconstruction algorithm and its implementation written in C++ using the SYCL programming model, where its key concepts are employed to migrate efficiently between specific hardware platforms. The key part consists of the evaluation of multiple computing platforms, including CPU and GPU devices. The results are summarized, and the conclusions are formulated in the final part of this contribution.

Background

Particle Tracking

Particle tracking is a fundamental technique to determine the momentum and charge of particles produced by a particle beam interacting with a target or another beam. Various detector technologies register the position of passing particles, ranging from fine-grained pixels of silicon arrays to straw tubes filled with gas. In all cases, a particle leaves information about its trajectory by inducing electrical signals in detector channels, which are then registered by the digitization and readout electronics.

The track reconstruction process usually consists of three primary stages: (i) raw data filtration and preparation; (ii) pattern recognition procedure that selects from all registered signals only those groups that become track candidates, characterized by specific spatial and temporal relationships; and (iii) track fitting that fits a straight line or a curve (in case a magnetic field is present) to the previously selected detector signals. The computing power required to process possible track candidates effectively grows significantly with the number of detector channels and the beam particle interaction rate in the target. Let us consider the ATLAS Inner Detector, which has nearly one billion electronic channels. During Run 2 (13 TeV), the number of hits registered during a single bunch crossing every 25 ns in proton–proton collisions ranged between 30k and 60k, [12].

Although the first step in data processing is closely linked to a specific detector and its readout system and cannot be generalized, the remaining two steps utilize a variety of standard algorithms. These algorithms range from primitive data sorting and sequential search to methods such as Nearest Neighbour, Kalman filtering, and, more recently, Graph Neural Networks [13–15].

The selection of algorithms is also driven by the location and specification of the computing platform where the track reconstruction is to be performed in the complete data processing pipeline. In the real-time regime, realization close to the detector hardware utilizes FPGA resources where non-iterative algorithms are favorable. On the other hand, reconstructing tracks offline on CPU-GPU platforms allows the implementation of more complex numeric algorithms.

Each approach necessitates a distinct implementation and a high level of optimizations, often utilizing various programming languages, models, techniques, and vendor-specific tools. Research on which solution would best suit a particular use case and its requirements would be facilitated by a single-source implementation and its evaluation on various computing platforms.

Heterogeneous Programming Models

Traditionally, every vendor provides a set of APIs and programming models dedicated to supporting its platforms exclusively. That leads to a situation when developing and maintaining large accelerated codebases that are compatible with all major platforms from leading vendors becomes a considerable challenge, especially for scientific projects where resources are usually limited.

To address this issue, universal or heterogeneous programming models were developed. Their primary objective is to provide mechanisms and abstractions that enable the compilation and execution of a single code implementation of a given algorithm on different devices. Some of the most influential ones are OpenCL [16], SYCL [7], OpenMP [17], and Kokkos [18, 19].

Most are based on a host-kernel model widely known from NVIDIA CUDA [20]. One exception is OpenMP, a directive-based model initially designed only for CPU parallelization. In newer versions of OpenMP, major GPU vendors added the ability to offload computations to GPU accelerators.

Khronos SYCL

This paper will focus on the SYCL programming model, its capabilities, and performance portability between compilers and hardware platforms. SYCL is a high-level programming model maintained by the Khronos Group and aims to support various hardware platforms from different vendors in heterogeneous computing systems. The main goal of SYCL is to provide a base API ensuring code portability so that a single application can make use of different hardware targets like CPUs, GPUs, or FPGAs, either simultaneously or depending on underlying computing system specifications. It was originally designed as a single-source C++-based abstraction layer over OpenCL, but has since evolved into an independent standard.

Programming Model Overview

SYCL APIs are based on standard C++17 without language extensions and use a classic approach with device sections located in separate procedures - kernels. Kernels are provided to the SYCL device queue as C++ functors (functions, lambdas, or callable objects). Host and device code (kernels and auxiliary functions) form a single shared codebase. The SYCL compiler is responsible for identifying device code and compiling it to the format consumable by the target device.

The default memory management in SYCL is based on the buffer-accessor model with *buffer* object responsible for handling underlying memory and data ownership. The data can be accessed using *accessor* objects with predeclared read-write permission. Using accessor properties, SYCL runtime can create an acyclic dependency graph of kernels and perform implicit scheduling of data transfers between the host and accelerators. Since SYCL 2020 exists also an alternative pointer-based memory management mechanism called Unified Shared Memory (or USM), which is similar to solutions known from other kernel-based programming models like CUDA and enables explicit memory management.

Implementations

Similarly to the C++ standard, SYCL specification and Khronos Group do not provide a reference compiler. Hence, the actual support for SYCL on different platforms depends on its voluntary implementers. Currently, two implementations of the latest SYCL specification can be considered mature or production-ready: AdaptiveCpp (formerly hipSYCL) [21] and Intel oneAPI with Data Parallel C++ (or DPC++) compiler [22]. The first one is developed and maintained by the research team based at Heidelberg University. Leveraging existing, native toolchains, it can support virtually all major CPU and GPU architectures from vendors like AMD, NVIDIA, and Intel, including the most recent Intel Ponte Vecchio GPU accelerators. The latter is based on the LLVM platform and, as a part of the oneAPI project, is the primary platform for software development targeting Intel hardware. It implements standard SYCL API and provides extensions allowing software optimizations for specialized hardware like Intel FPGA devices. While it is maintained and supported mainly by Intel, with a set of plugins developed by Codeplay, its applicability can be extended to AMD and NVIDIA platforms.

Using the two SYCL implementations mentioned above, it was possible to evaluate the software on various hardware platforms and test not only its ability to compile and execute, but also the code and performance portability between platforms and compilers.

Existing, Hardware-Accelerated Software Frameworks

Efforts of the community towards the development of generalized data processing software frameworks, including

hardware acceleration, have been made, for instance, in [23] for the ALICE and [24] for the LHCb experiments. They both target providing fast processing pipelines for online feature extraction and data selection with acceleration on GPUs at their cores. Their software support is provided by independent, in-house developed layers of abstraction through C++ macros and library constructs. Therefore, two groups operating within one research facility have distinct implementations of a similar mechanism, which makes it difficult to reuse the codebase and to efficiently employ the resources. Such an approach is also limited to the existing and supported APIs (currently CUDA and HIP [25]) where the introduction of new types of devices, such as FPGA or RISC-V, would require additions to the abstraction layer, which can be a time-consuming and error-prone process.

Application of a framework that standardizes access to peripheral computing devices of various architectures, APIs, and tools would help avoid reinventing the wheel, reduce the complexity of the core, and allow developers to focus on algorithmics rather than infrastructure. An interesting argument in support of a standardized framework is the introduction of Intel GPU devices, whose software support is via OneAPI, which, in fact, is based on one of the SYCL implementations, DPC++.

Particle Tracking in $\overline{\text{P}}$ANDA

$\overline{\text{P}}$ANDA Experiment and the Forward Tracker

The $\overline{\text{P}}$ANDA experiment, schematically shown in Fig. 1, has been designed to study the field of hadron physics with an antiproton beam at momenta between 1.5 and 15 GeV/c interacting with an internal hydrogen target. The primary goal of the research is to investigate charmonium and open charm mesons, search for gluonic excitations, and spectroscopy of both charmed and strange bosons [26].

The Forward Tracker (abbr. FT) is a gaseous, straw-tube detector designed to measure the trajectory of charged particles deflected by a magnetic dipole field. It consists of 6 stations arranged in pairs and located before (FT12: stations FT1 and FT2), inside (FT34), and after a dipole magnet (FT56). Each station is composed of 4 double layers of straws, with the first and last one oriented vertically, while the inner layers are tilted by $+5^\circ$ and -5° accordingly for *XOY* plane determination [27]. A simplified, pictorial representation of the straws arrangement is presented in Fig. 2. The total number of straws and data channels is equal to 12224.

Due to the complex topologies of the events of interest requiring diversified selection criteria and high interaction rates, the Data Acquisition System (abbreviated as DAQ) has been designed to implement the continuous readout concept without a low-level hardware trigger [28]. Each detector subsystem operates autonomously and provides detector-specific analog signal handling, digitization, filtration, zero-suppression, and exposes a preprocessed data package to a common DAQ network. The network transfers the data packages from all subsystems to the computing facility for high-level processing. In order to assemble data packages from autonomous subsystems, a precise synchronization system is also provided by the common network [29].

Such a system and consequently data organization is ideal for exploring accelerated computing techniques. Various

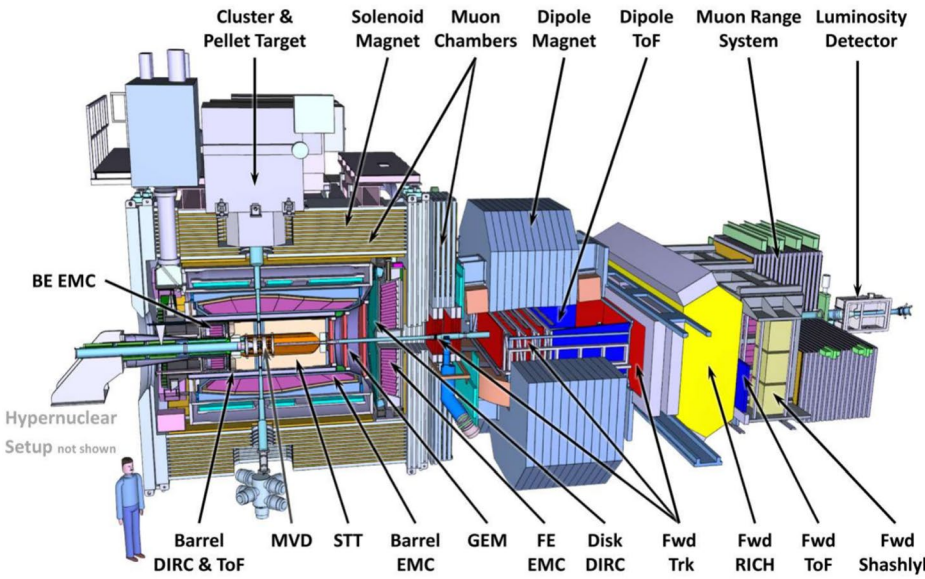


Fig. 1 Complete $\overline{\text{P}}$ANDA detector setup. Forward Tracker stations FT12, FT34 and FT56 shown as *Fwd Trk* from left to right

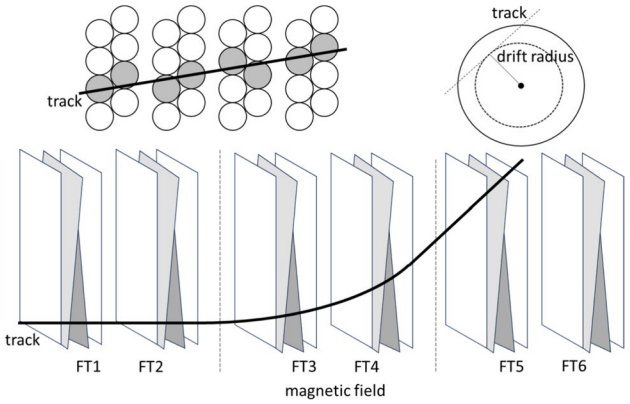


Fig. 2 Arrangement of the straws in the Forward Tracker. Each station is composed of four double layers (top left), with inclined inner layers (center, light, and dark grey). Highlighted straws (top left) respond to an exemplary track passing through the detector and bent by the magnetic field. The single straw response is presented on top, right

detector subsystems require distinct algorithms that can be executed in parallel and use the unique features of heterogeneous platforms depending on the given algorithm characteristics.

Particle Tracking Algorithm

Every interaction between charged particles passing through the detector and its elements is registered and collected during operation. These interactions, called hits, are then assigned to a specific particle path, and finally, each path is described analytically. The algorithm takes as an input a list of hits $H(S, r)$, where S is an activated straw and r is a drift radius, i.e., the shortest distance between the passing particle and the straws' anode wire (Fig. 2 top, right). In the following description, we assume all hits making up an input to the algorithm belong to a single event, defined by a dedicated algorithm

In the case of the PANDA Forward Tracker, the track reconstruction algorithm consists of three main parts: for particle tracks in the field-free region in front and after the dipole magnet (FT12 and FT56), for particles in the magnetic field (FT34), and for matching track segments from different tracking stations.

Tracking in the Field-Free Regions

The first step of the algorithm is the linear track segment reconstruction for free particles outside of the magnetic field in FT12 and FT56. At this stage, all possible track candidates are identified and then filtered for duplicates. It starts with the reconstruction of projections of the tracks onto the *ZOX* (horizontal) using hits from vertical layers of straws.

Initially, all pairs of hits (H_1, H_2) are selected, such that H_1 and H_2 come from one of the first two and the last two vertical layers in a given tracking station. In FT12, an additional constraint to the hit pairs is applied so that only pairs forming a line that can pass through the collision point are considered. The equation of a straight line l passing through centers of S_1 and S_2 is determined for each pair. Then the procedure looks for activated straws S_i for which the distance $d(S_i, l) <= max(r_1, r_2) + R$, where R is the radius of the straw tube. If multiple straws exist which fulfill the above criteria for the same layer, the one with a smaller value of d is selected for further processing. When the number of selected straws is greater than six (an ideal track would engage eight straws, but the detector efficiency is considered), reconstruction proceeds to the next step, and a pair of circles $O_1(S_1, r_1)$ and $O_2(S_2, r_2)$ with centers in coordinates of S_1 and S_2 anode wires locations, is constructed. Then, equations of four possible lines tangent to O_1 and O_2 are calculated. For each line, the algorithm checks if the line passes through at least six of the previously determined straws S_i and if such lines exist, the one with the smallest sum of distances from hits is considered a track candidate.

This technique may lead to the creation of additional, duplicated track candidates. Therefore, the duplicate elimination procedure is conducted on the list of track candidates determined in the previous steps. A track candidate is considered a duplicate of another track if they overlap in at least three layers, and in the rest of the layers, hit straws are neighbors.

The set of such track candidates, containing only projections onto the *ZOX* plane and with duplicates filtered out, is then passed to the next step to obtain the second projection – onto the vertical *ZOY* plane. To determine the *ZOY* projection, first, the auxiliary plane $Z'OY$ is defined as a plane containing the already reconstructed *ZOX* projection and perpendicular to the *ZOX* plane. Then, all hits originating in inclined straw layers are identified, and intersections $P_k(z_k, y_k)$ of hit straw anode wires with the $Z'OY$ plane are calculated. For each such intersection, two auxiliary points $P_1(z_k, y_k + cc)$ and $P_2(z_k, y_k - cc)$, where $cc = r\sqrt{1 + a^2}$ with a, r being a slope of the straw and hits drift radius, respectively, are needed. Finally, a set of inclined straw hits with two auxiliary points on $Z'OY$ each are sent to the procedure similar to the one used for the *ZOX* plane. Four lines are constructed for each pair of auxiliary points from the outer inclined layers, and the distances from the inclined straws to the lines are calculated. From tangent lines consisting of at least six straws, the one with the smallest sum of distances is selected as a projection of the track candidate onto the *ZOY* plane.

The above procedure is conducted independently for both tracking stations outside the magnetic field.

Tracking Inside the Magnetic Field

After leaving the first tracking station, FT12, particles passing through the PANDA Forward Tracker enter the magnetic field of the dipole magnet, which bends the particle path in the *ZOX* plane to a circular shape. The second tracking station, FT34, is located inside the dipole magnet.

For the *ZOX* plane inside the magnetic field, the track reconstruction algorithm has to obtain the equation of a circle describing the particle path. For this purpose, hits from vertical straws are again used in addition to the tracks reconstructed in the FT12 tracking station. For each such track and each hit straw $S(z, x)$ from one of the last two layers of the FT34, two circles are constructed that are both tangent to the FT12; the first is passing through the point $P_1(z, x + r)$ and the second through $P_2(z, x - r)$. For both circles, the closest reasonable hits, i.e., such that the track lays inside the hit straw, in each layer are found. Finally, the circle with a smaller mean of distances from the circle to hits is accepted to the track as the extension to the FT12 track into the FT34.

ZOY projections of tracks in the magnetic field are assumed to be straight lines. So, for every track with a *ZOX* projection of the FT34 extension adequately reconstructed, the equation of the *ZOY* projection of a track from the FT12 station is used. The procedure looks for the closest hit in each inclined layer in FT34, and if the distance from the extension of the FT12 track to the selected hit is reasonable, the hit is accepted to the track.

Matching Track Segments from FT1234 with FT56 Extensions

As tracks in FT34 are reconstructed as extensions of FT12, only existing FT1234 tracks have to be matched with their FT56 continuations if such exists. As some lower momentum particles may escape the detector before reaching FT56, not all FT1234 tracks have their FT56 extensions. For every FT56 track, distances to all FT1234 tracks at a particular z coordinate at the end of the magnetic field are calculated on both *ZOX* and *ZOY* planes. The FT56 part is considered an extension of the FT1234 track with the smallest sum of distances. It is ensured that each FT1234 part is matched with at most one FT56 track segment.

Tracking Accuracy

The correctness and accuracy of the discussed algorithm were evaluated using multiple datasets of simulated data. To evaluate the algorithm in different conditions (signal densities and particle path shapes), we prepared simulations of 1, 3, 5, and 8 muons per event with energies of 0.55, 2.55, and 5.55GeV emitted at the polar angle in the range from 0° to 15°. Such a range of parameters has been selected to reflect experiment conditions and produce both simpler and harder to resolve events (track overlapping).

Figure 3 shows the ratio of correctly reconstructed tracks in each analyzed set. We considered a track to be correctly reconstructed if it was reconstructed at all detector stations, and at least 80% of reconstructed hits occurred in the same straws that belonged to the original simulated track. The percentage of correctly reconstructed tracks was determined in relation to reconstructible simulated tracks, i.e., those that passed through all detector stations. The achieved accuracy is above 90% in all except for lower energy eight muon cases. Overall results are on par with expectations and comply with requirements [10].

Accelerated Implementation of Particle Tracking

The SYCL implementation of the track reconstruction algorithm consists of seven kernels: six main kernels corresponding to the consecutive steps of the algorithm and one auxiliary kernel (Rewrite kernel). Besides logical algorithm stages that have their reflection in individual

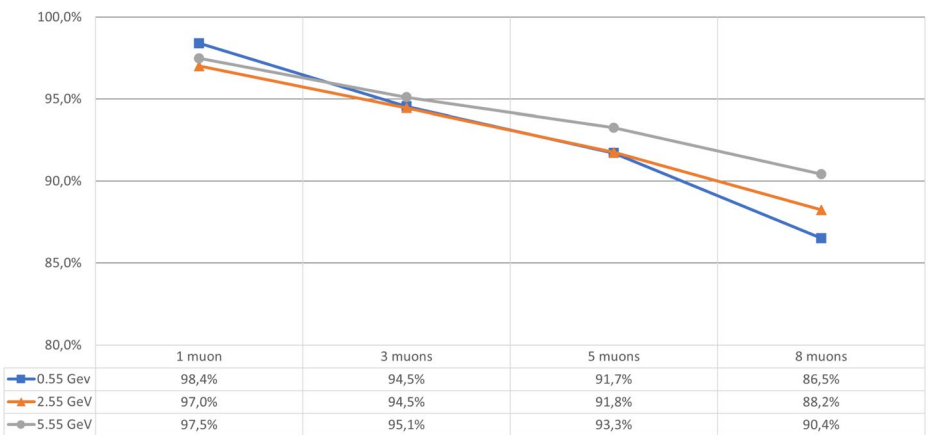


Fig. 3 Tracking accuracy, $\theta \in (0°, 15°)$, full tracks

kernels, the whole track reconstruction pipeline can be divided into three stages (FT12, FT34, and FT56) named after the detector sections from which they require input data. Stages FT12 and FT56 are independent and potentially can be executed in parallel. The complete reconstruction pipeline involves 11 kernel submissions—two times four kernels for linear track segments (FT12 and FT56), two kernels for circular track segments (FT34), and one Match kernel. The subsequent pipeline stages and kernels involved are presented in Fig. 4.

All kernels are submitted to the single SYCL queue. We observed no measurable performance improvement achieved using separate queues for independent algorithm parts. On CPU devices, multiple queue execution is generally not expected to improve performance significantly. At the same time, recent versions of leading SYCL implementations can allocate multiple GPU device streams under the same SYCL queue and implicitly cover many cases when independent kernels or data transfer can be executed in parallel or out-of-order.

Parallelization

The SYCL implementation started as a low-effort port from a single-threaded C++ code. Several optimizations were later applied to improve the performance of accelerated code. Some of the most notable changes were made concerning parallelization granularity. In the first naive approach, all kernels were parallelized on the level of events, meaning that each kernel instance (or GPU thread) realized all computation needed to process a single event. This solution resulted in large kernels, low GPU occupancy, and non-local memory access from each thread. In subsequent iterations, more and more granular parallelization was introduced. Finally, the best performance was achieved using a combination of kernels parallelized on either hit-pair, track, or event level.

The first kernel, LineZOX, is parallelized on the hit-pair level, with each pair of outermost hits processed in the separate work-item. The kernel uses the mechanism of SYCL nd_ranges, which enables fine-grained control over the work-group size and distribution of work-item between

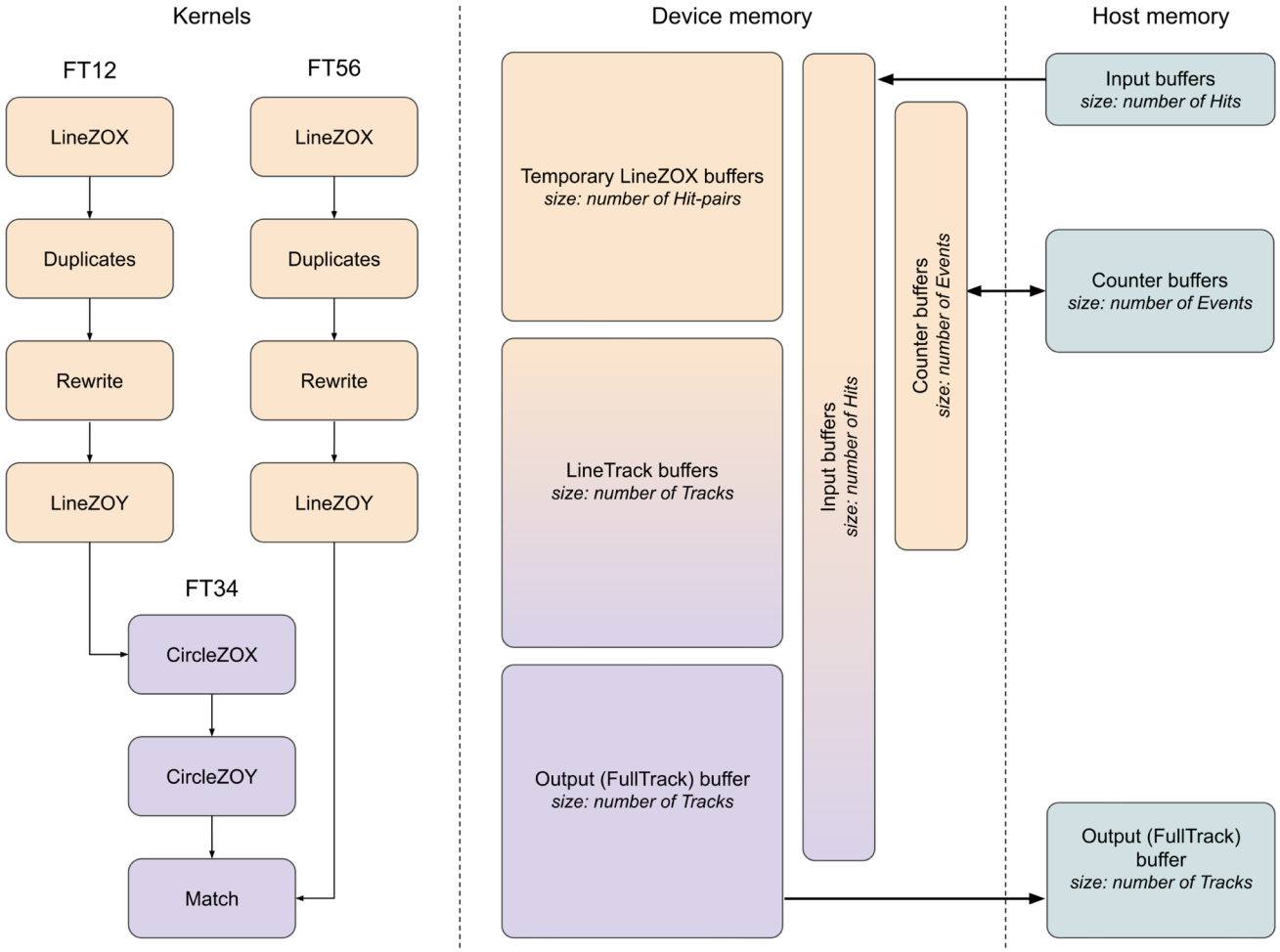


Fig. 4 Kernel data dependency and data flow graph

Table 1 Kernel parallelization levels

Kernel	Parallelization level	Mechanism
LineZOX	Hit pair	sycl::nd_range
Duplicates	Event	sycl::range
Rewrite		
LineZOY	Track	
CircleZOX		
CircleZOY		
Match	Event	

work-groups. Each work-group is responsible for processing all hit-pairs belonging to the single event, and the work-group size can be adjusted to best fit the GPU architecture and average event size. On the other hand, the Duplicates, Rewrite, and Match kernels still operate on the event level, performing all processing for a single event in a dedicated work-item. Those are relatively short and light kernels, so lower granularity doesn't have a meaningful impact on performance. The rest of the implemented kernels (LineZOY, CircleZOX, CircleZOY) are parallelized on the track level. In this case, work-items are responsible for a single reconstruction step of precisely one track candidate. Kernels other than LineZOX do not use features of nd_range-based parallelism and are based on more straightforward regular ranges. A list of kernels with their properties is presented in Table 1.

Data Management and Rewrite Kernel

For data management, the mechanism of SYCL buffers was used. To minimize time spent on data movement, we organized the pipeline so that the intermediate data need to be accessed on the host side only once: after the Duplicates step, so two times during the complete algorithm execution. The transferred data contain information about the number of track candidates reconstructed for each of the processed events and is used to allocate buffers for further steps of precisely the required size. All other data movements happen for the input and final output data (fully reconstructed particle tracks). The schematic of the data flow is shown in Fig. 4.

The output buffer of the LineZOX step needs to fit all theoretically possible track candidates and, hence, is allocated with a surplus and is usually sparse. The auxiliary kernel (Rewrite kernel) copies track candidates remaining after duplicates removal from the original location to the final smaller output buffer. As LineZOX and Duplicates kernels have $O(n^2)$ memory complexity, this additional step significantly reduces the algorithms' memory footprint and the amount of data transferred at the end of the pipeline.

Other performance considerations included the memory layout for input data. In the preprocessing stage, we split input hits into buckets that later serve as inputs for different algorithm steps. The input data for each event were initially wrapped in a dedicated data class object during the preprocessing. In the SYCL context, such a data structure was required to have a fixed size for all events from the input dataset, which led to inefficient use of memory as events have varying sizes. The memory model with flat buffers was introduced to overcome the issue. Input data for all events are stored continuously in large buffers of hits, and additional small buffers contain offsets for each event.

Performance Evaluation

Measurement Procedure

As experimental dataset, we used a collection of Monte Carlo simulations of the PANDA detector generated using the PandaRoot framework [30]. As the benchmarking dataset, we used muon simulations described in 3.3. The complete dataset consisted of 28 simulation files (or subsets) of 10,000 events each, resulting in a total of 280 thousand simulated events. In order to test the performance on input data larger than simulated 10000 events, events from each subset were multiplied by reusing events, forming 2, 4, 8, and up to 64 times larger datasets.

Measurements were conducted on various CPU and GPU-based platforms. We used two 64-core AMD EPYC processors of different generations and 24-core Intel Xeon units for CPU platforms. GPU targets included two NVIDIA accelerators and two AMD Instinct devices. Additional test runs were also carried out using Intel Ponte Vecchio GPU and AMD Alveo U280 FPGA accelerator. Table 2 contains specific information about the processing units used.

Table 2 Experimental platforms, hardware setup, theoretical maximum performance for CPU [32] and GPU [33]

Platform	Device	Type	Units[1]	FP32 th. peak perf. [TFlops]
Cascade	Intel Xeon Platinum 8268	CPU	24	nd
Rome	AMD EPYC 7742		64	2.3
Milan	AMD EPYC 7763		64	2.5
Volta	NVIDIA V100	GPU	80	15.7
Ampere	NVIDIA A100		108	19.5
CDNA	AMD Instinct MI100		120	23.1
CDNA2	AMD Instinct MI250X		220	47.9
Ponte Vecchio	Intel GPU Max 1100		56	22.2
Alveo	AMD Alveo U280	FPGA	n/a	n/a

CPU—cores; NVIDIA GPUs—streaming multiprocessors, AMD GPUs—compute units; Intel GPUs—X^e-cores

Table 3 Experimental platforms, software setup

Platform	AdaptiveCpp	DPC++
Cascade	23.10/LLVM 16	2024.0
Rome		
Milan		
Volta	23.10/CUDA 11.8/LLVM 16	2024.0/CUDA 11.8
Ampere		
CDNA	23.10/ROCm 5.4/LLVM 16	2024.0/ROCm 5.4
CDNA2		
Ponte Vecchio	23.10/LLVM 16	2023.2
Alveo	n/a	triSYCL@3272498

All tests were performed using two major SYCL implementations: AdaptiveCpp and Intel's oneAPI (DPC++). For the AdaptiveCpp runs, the SYCL code was compiled using clang-based compilation flows for all targets. In order to enable DPC++ support on AMD and NVIDIA GPU platforms, we used Codeplay oneAPI plugins and ahead-of-time compilation targets for specific GPU architectures. For DPC++ compilation on CPU platforms, we used the ahead-of-time *x*86_64 universal target. Finally, the Alveo platform was targeted using the experimental fork of DPC++, adding AMD FPGA support [31]. Software configurations and specific SYCL compiler versions used on each platform are shown in Table 3.

The track reconstruction pipeline was executed on each platform using our experimental dataset in batches of different sizes from 10 to 640 thousand events, measuring the average time to process a single event. The number of threads used for the primary measurements was set to the number of physical cores on all CPU platforms. Because of the underlying system configurations, using two logical threads per physical core was possible only on the Milan platform. Additional runs were conducted on the Rome and Milan platforms to analyze the performance on different thread counts.

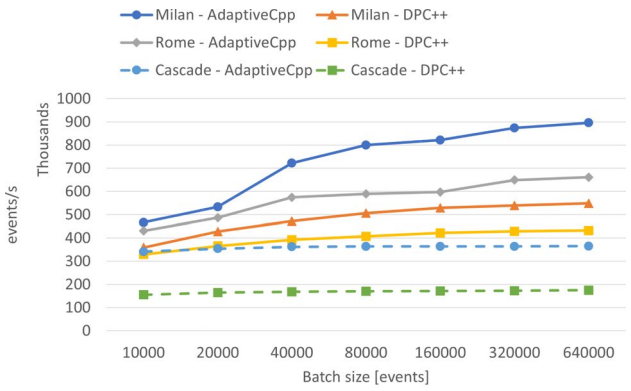
The performance of SYCL implementation was finally compared against the original single-threaded C++ version and native CUDA and HIP implementations.

Performance Results

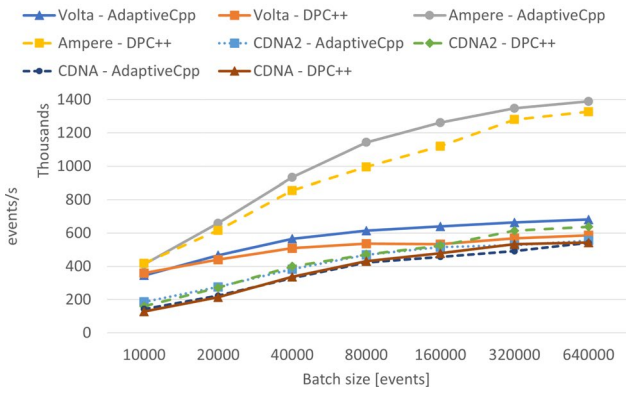
CPU Performance

The performance of CPU platforms was analyzed in terms of the maximum throughput achieved for the entire processor and throughput normalized to a single core.

Figure 5a shows the throughput results (events per second) obtained on each tested CPU platform for various input data batch sizes using all available processor cores (one thread per physical core). The obtained results indicate an



(a) CPU



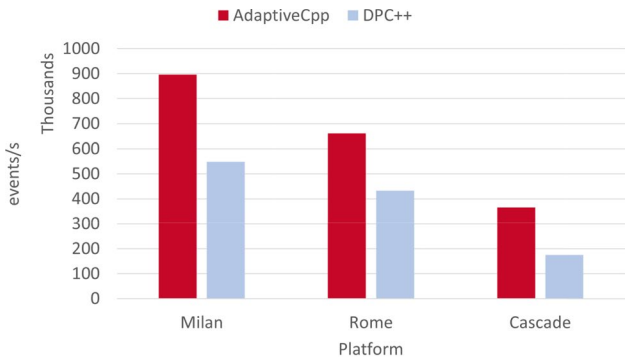
(b) GPU

Fig. 5 Throughput changes under different batch sizes

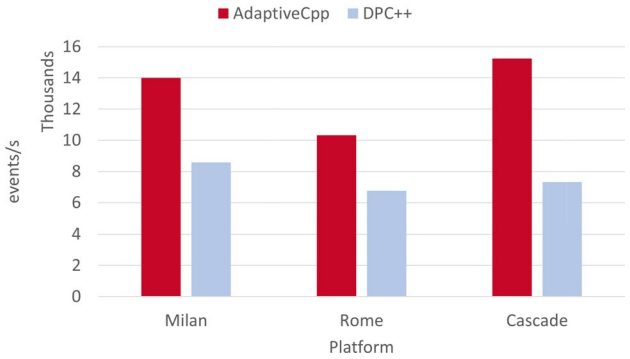
increase in the efficiency of calculations for larger data packages, most visible on the Milan platform. This is consistent with the expectation of a decreasing impact of the SYCL runtime overhead on the overall computation time. The highest throughput was achieved for a data package of 640 thousand events. The results for this batch size are shown in detail in Fig. 6a. Figure 6b shows these results normalized to one physical core for each CPU chip.

The Rome and Milan platforms are based on two generations (Zen 2 and Zen 3 architectures) of high-end 64-core AMD processors. We see performance gains between generations of around 35% for medium to large batches and tinier differences for smaller batch sizes. The increase for larger batch sizes does reflect the differences observed for these systems in synthetic CPU benchmarks [32].

The Cascade platform is based on a 24-core processor, so its lower overall performance is understandable. However, it has the highest throughput when normalized to a single core despite being the oldest of analyzed CPU platforms. As the total throughput on the Cascade platform was measured using a lower number of threads, the reason for such results might be a non-ideal algorithm speedup with the number of cores resulting from imperfect parallelization. An additional factor

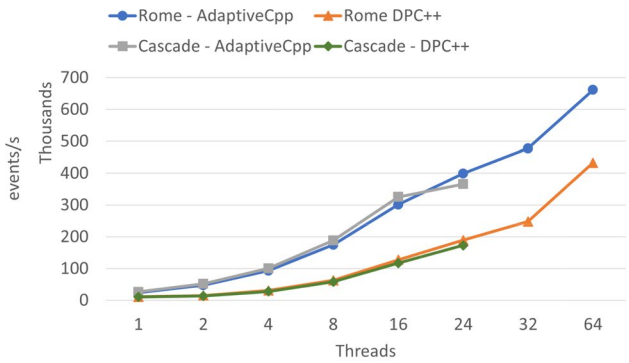


(a) All physical cores, single thread per core

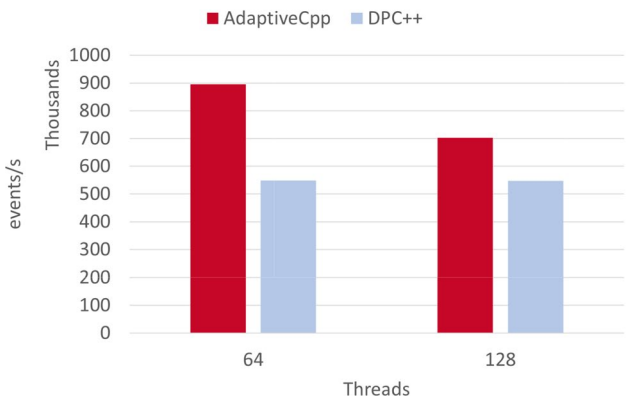


(b) Single core, single thread (normalized)

Fig. 6 CPU throughput



(a) Rome, Cascade



(b) Milan

Fig. 7 CPU speedup

that may affect per-core performance is the memory bandwidth. The Cascade platform is based on 6-channel DDR4-2933 memory with a maximum theoretical bandwidth per core that is 83% higher than in the case of AMD platforms equipped with 8-channel DDR4-3200 memory.

The data on the algorithm's speedup with the number of threads used for the Rome and Cascade platforms in Fig. 7a show similar performance scalability up to 24 threads (cores) on both platforms. These results indicate that memory bandwidth does not significantly influence normalized performance on the Cascade platform, which is, in fact, primarily affected by non-ideal algorithm acceleration and different core counts on Intel and AMD platforms. The above data can be used to estimate the algorithm's parallelization portion p according to Amdahl's law [34]:

$$S_p(s) = \frac{1}{(1-p) + \frac{p}{s}},$$

where s is a speedup of parallel sections, and $S_p(s)$ is the maximum theoretical speedup of the whole task under the given value of p. Assuming s equals the number of threads used, we use the previous speedup results and non-linear global optimization to determine $p_{acpp} = 0.98$ and $p_{dpc++} = 0.99$ for AdaptiveCpp and DPC++ runs on the Rome platform. The fitted curves are shown in Fig. 8 alongside the original speedup data.

Hence, in our case, DPC++ can deliver similar speedup capabilities to AdaptiveCpp while achieving significantly lower overall performance. The reason could be different overheads from backend runtimes (OpenCL in DPC++ and OpenMP in AdaptiveCpp), which can become apparent in the case of multiple short kernel calls.

To investigate the speedup aspect further, we checked how both SYCL implementations would behave on the Milan platform with two logical threads per one physical core enabled utilizing a full EPYC 7763 CPU. Figure 7b shows almost identical throughput achieved using 64 and 128 threads in DPC++ runs and a significant decline in the performance of the AdaptiveCpp version.

Native Single-Threaded CPU Performance

To investigate the implementation's effectiveness using SYCL further, we compared its performance to the original single-threaded C++ implementation. Figure 9 shows the throughput achieved on the Rome and Cascade

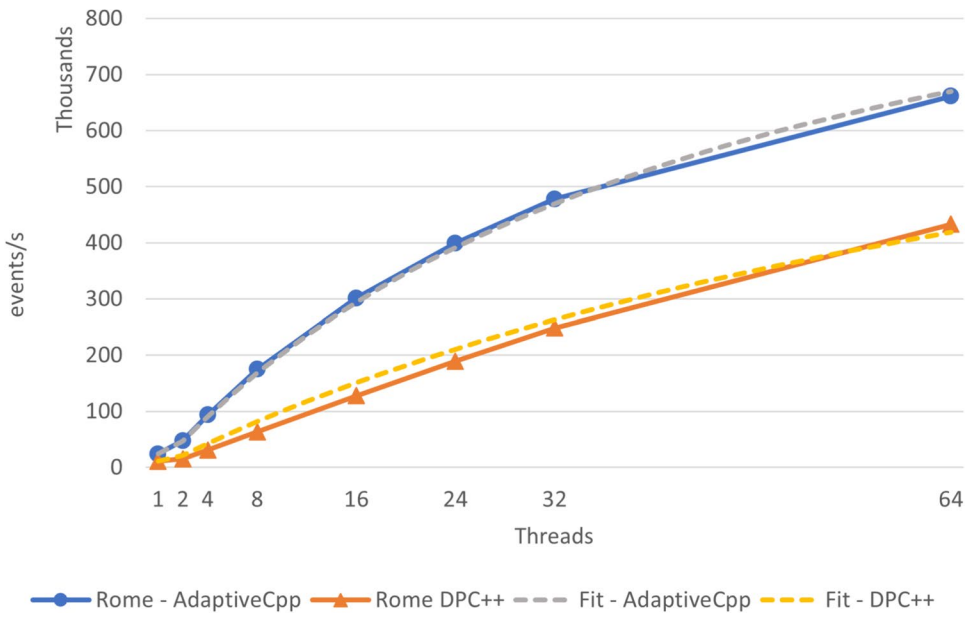


Fig. 8 CPU speedup, Rome platform, Amdahl's law fit

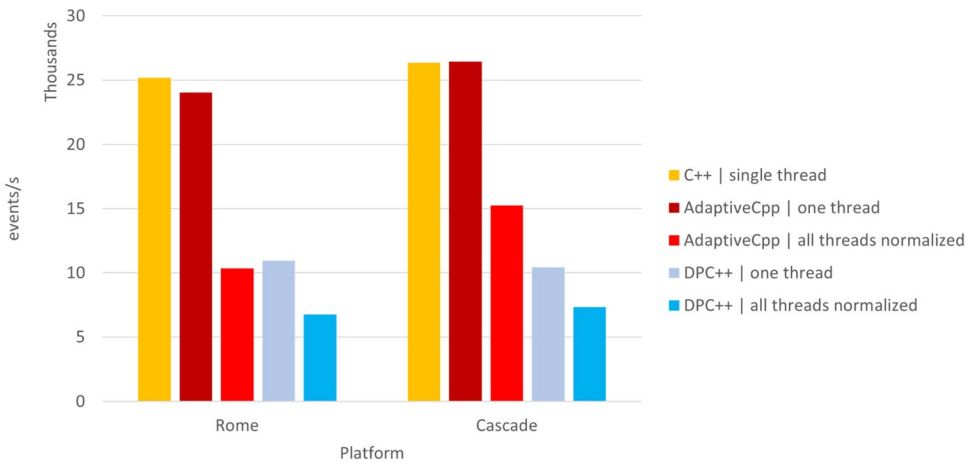


Fig. 9 Throughput per thread

platforms by the pure C++ and SYCL implementations. Additionally, the previously mentioned data on the total performance of the SYCL implementation normalized to one thread are included in that figure.

In the case of SYCL implementation, much higher efficiency was achieved using the AdaptiveCpp compiler. In this case, the performance of a single computational thread is similar to the pure C++ version. The DPC++ compiler performs worse, consistent with previous data. The normalized throughput data again show the impact of the non-ideal speedup of the algorithm with the number of threads.

It is worth mentioning that both of the SYCL compilers used (AdaptiveCpp and DPC++) use dedicated threads to execute kernels. Therefore, even though the actual calculations are performed in one thread, the application may use additional resources.

GPU Performance

When evaluating performance on GPU-based platforms, a procedure analogous to one previously used for CPU platforms was conducted. Visible in Fig. 5b, the boost in throughput occurring as the size of the input data packet increases is even more pronounced than for the CPU platforms, especially on the Ampere platform. Again, the highest throughput was achieved with a data package of 640 thousand events. Figure 10 shows the throughput results on GPU platforms for data batches of 640 thousand events in the context of the previously studied CPU results.

There are visible differences in performance between generations of NVIDIA (Volta, Ampere) platforms. In the case of AMD-based platforms (CDNA, CDNA2), these differences are significantly lower and do not reflect theoretical

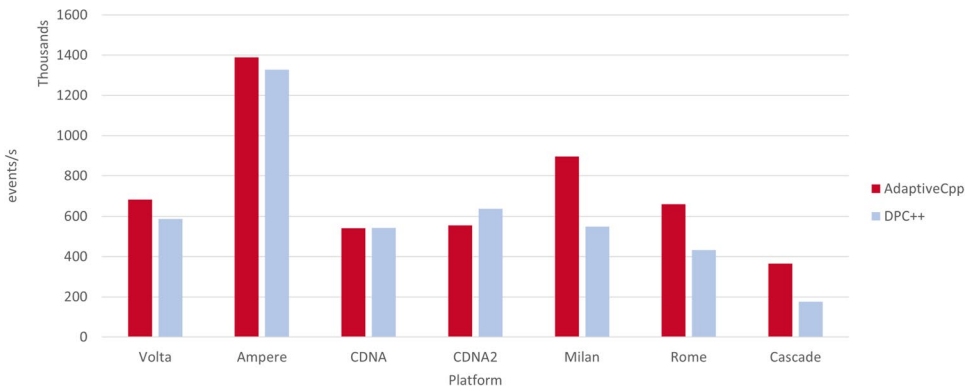


Fig. 10 Throughput

differences in performance between generations. Also, the performance of AMD platforms, especially CDNA2, is lower than expected compared to NVIDIA platforms (Ampere). Profiling data show that differences between CDNA2 and Ampere come mostly from kernel execution times rather than data transfers.

It is also worth noting that in the case of platforms based on NVIDIA systems, the performance obtained with the AdaptiveCpp compiler is relevantly higher than in analogous runs with DPC++. The situation is different on AMD platforms where DPC++ performs similarly (CDNA) or better (CDNA2) than AdaptiveCpp. Please note that both SYCL implementations (AdaptiveCpp and DPC++) rely on existing CUDA and HIP support in the upstream LLVM toolchain.

Native GPU Performance

In addition to analyzing the performance of SYCL implementations on various GPU platforms, it is also important to compare performance with implementations in programming models native to specific platforms—in this case, CUDA, and HIP.

When preparing such a comparison, it is essential for the native versions to be designed as a close port of the SYCL version. At the same time, applying best practices for aspects such as configuring computing grid parameters, which are done manually in both CUDA and HIP, is essential. This allowed us to compare the capabilities of SYCL compilers with their native counterparts and, at the same time, minimize the impact of differences in CUDA/HIP runtime management on the overall performance of a given implementation.

Figure 11 shows a comparison of the throughput achieved by the native versions with previous results for the SYCL implementation with the AdaptiveCpp and DPC++ compilers. As one can see, in most cases, the performance of the native versions is very close to that achieved by SYCL. Only on the CDNA platform the performance was significantly higher compared to SYCL.

Further fine-tuning of native versions for specific GPU architectures could likely result in additional throughput gains. However, a similar situation could occur in the case of SYCL implementation and, for example, wider use of the nd_ranges mechanism with manual management of the computing grid. However, it was a conscious decision of the authors to use simpler API variants except when there was a strong indication that moving to a lower level of abstraction could result in significant performance improvements, as with the LineZOX kernel.

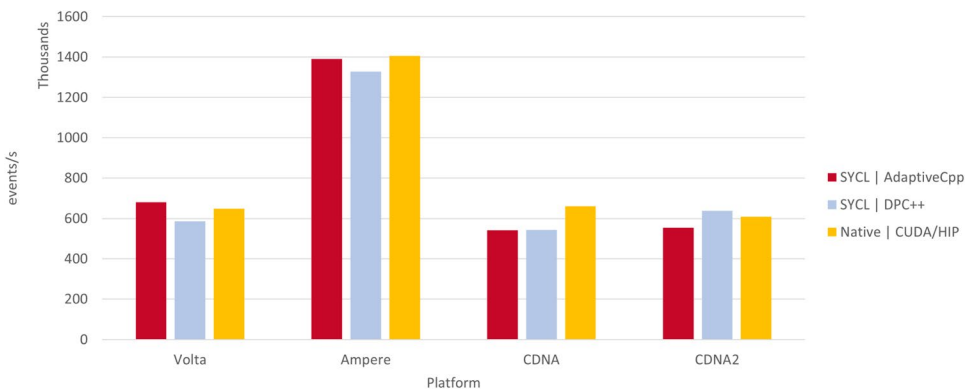


Fig. 11 Native GPU throughput

Other Platforms

For the record, we have also successfully compiled and run the solution on the Intel Ponte Vecchio GPU and AMD Alveo U280 FPGA accelerator cards. Due to the terms and conditions of the Intel Developer Cloud, we were not allowed to present the performance results of the first one.

To successfully compile for the Alveo platform, due to toolchain limitations, we had to use a simplified version of the LineZOX kernel, which uses basic SYCL ranges (sycl::range) instead of ND-range. As the execution on FPGA is not parallel in the sense that it is on GPUs and CPUs, this change did not have a negative impact on performance. Additionally, to make placing and routing compilation steps possible, it was necessary to distribute SYCL buffers across multiple memory banks (DDR and HBM). The successful compilation of the complex SYCL code for the FPGA platform is an achievement of its own. The measured performance of Alveo U280 was two to three orders of magnitude worse compared to other platforms, as FPGAs can outperform other architectures in streamlined and pipelined applications, which was not the case with the tracking algorithm. This exercise allowed us, however, to evaluate the current support of the AMD FPGAs in SYCL. Basic kernel generation (C++ to HDL) works correctly utilizing the Vitis HLS compiler, which is already well-established in this area. Unlike with other platforms, on an FPGA accelerator, the user can define kernel-to-host connectivity and paths for the memory buffers (e.g., transferring through external DDR, internal HBM, directly to the kernel, and more). Those decisions have a tremendous impact on the overall solution performance but require manual implementation of specific constructs [35]. This step, however, imposes modifications to the original source code and, therefore, denies the single codebase for any platform idea behind SYCL. It is important to note that without those optimizations, one can still generate a working solution with callable kernels connected through a single port of an external DDR memory bank, which greatly limits achievable performance.

Conclusions

We have successfully demonstrated the portability of a single codebase between a wide range of computing platforms using the SYCL heterogeneous programming model on the example of the particle tracking algorithm for the Forward Tracker in the $\overline{\text{P}}$ANDA experiment. In-depth performance evaluation allows us to draw conclusions not only about the hardware platforms but SYCL implementations as well and can already serve as a reference for finding the best combination for a particular use case. Below is a summary of the main takeaways:

- Complex, C++ codebase, without any vendor-specific modifications, can be efficiently compiled and executed on all supported devices.
- SYCL overhead impact on performance becomes negligible with the batch size.
- Solution under evaluation manifests expected scalability, which shows no negative effect of SYCL on performance
- Selection of the SYCL implementation has a major impact on overall performance:
  - For both AMD and Intel CPU targets, AdaptiveCPP performs better than DPC++. Unintuitively, DPC++ (developed by Intel) for Intel CPUs can produce less performant solutions even by a factor of 2 compared to AdaptiveCPP.
  - For AMD and NVIDIA GPUs, the differences between SYCL implementations are not pronounced.
- Support for NVIDIA GPUs (CUDA backend) by both SYCL implementations produces better results than for the AMD GPUs (ROCm backend) when their theoretical peak performance is considered.

The development of the project allowed us to define a set of general considerations when designing software with SYCL. These concepts, however, do not differ from those in most parallel computing frameworks and can be narrowed down to three main points:

- When targeting GPU architectures, it is favorable to design kernels processing a larger number of minimalistic work-items than a smaller number of complex tasks.
- If a kernel consists of a large loop over input elements, it should be converted to a range call over work-items whenever possible.
- Host-device memory transfers should be limited to a minimum (favorably fetching input data and storing final results).

Software development has become an inherent part of large-scale physics experiments. From early simulations up to the measurement data analysis, the codebase is often developed by various groups, using various tools and techniques. With the increase in complexity of these processes, it becomes more and more important to efficiently develop code that delivers performant, reliable, reusable, and future-proof solutions. By introducing a standardized abstraction layer and the host-kernel model, SYCL can significantly help scientific collaborations without sacrificing performance.

Acknowledgements Computational resources have been provided by the ACC Cyfronet at The University of Science and Technology in Cracow, The Green IT Cube at GSI Helmholtzzentrum für Schwerionenforschung and the Facility for Antiproton and Ion Research (FAIR)

in Darmstadt. The project could be realized thanks to the support from the AMD University Program and their donations.

Author Contributions All authors contributed to the study conception and design. BS has implemented and evaluated the software components. GK has provided technical and substantive support. All authors have prepared the first draft of the manuscript.

Funding No funding was received to assist with the preparation of this manuscript.

Data Availability The datasets generated during and/or analyzed during the current study are available from the corresponding author upon reasonable request.

Declarations

Competing Interests The authors have no competing interests to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

References

1. The ALICE Collaboration: Technical Design Report for the Upgrade of the Online-Offline Computing System (2015). https://cds.cern.ch/record/2011297 . Accessed 24 Jun 2025.
2. Eulisse G, Rohr D (2024) The O2 software framework and GPU usage in ALICE online and offline reconstruction in Run 3. EPJ Web Conf 295:05022. https://doi.org/10.1051/epjconf/202429505022
3. Agarwal K (2023) The Compressed Baryonic Matter (CBM) experiment at FAIR - physics, status and prospects. Phys Scr. https://doi.org/10.1088/1402-4896/acbca7
4. Belias A, For the PANDA Collaboration (2023) Overview of the PANDA detector design at FAIR. Int J Mod Phys Conf Ser 51:2360001. https://doi.org/10.1142/S2010194523600017
5. Gutbrod H (2005) International Facility for Antiproton and Ion Research (FAIR) at GSI, Darmstadt. Nuclear Phys A. https://doi.org/10.1016/j.nuclphysa.2005.02.137
6. Lehmann I (2024) Status and Plans at GSI/FAIR. https://indico.global/event/2309/contributions/31748/. Accessed 24 Jun 2025.
7. SYCL Specification. https://www.khronos.org/sycl/. Accessed 10 Jan 2023.
8. The PANDA Collaboration (2021) The PANDA experiment: phase one physics at FAIR. Eur Phys J A. https://doi.org/10.1140/epja/s10050-021-00475-y
9. The PANDA Collaboration (2009) Physics performance report for PANDA: strong interaction studies with antiprotons. https://arxiv.org/abs/0903.3905 [hep-ex]. Accessed 24 Jun 2025.
10. Płażek J, Smyrski J, Salabura P et al (2021) Track finding algorithm using the track road method for the PANDA forward tracker. J Comput Sci. https://doi.org/10.1016/j.jocs.2021.101448
11. Adamczewski-Musch J et al (2021) Production and electromagnetic decay of hyperons: a feasibility study with HADES as a phase-0 experiment at FAIR. Eur Phys J A 57(4):138. https://doi.org/10.1140/epja/s10050-021-00388-w
12. Pérez D (2022) Improved track reconstruction for prompt and long-lived particles in ATLAS for the LHC Run 3. PoS ICHEP2022, 1078 https://doi.org/10.22323/1.414.1078
13. Ai X, Gray H, Salzburger A et al (2023) A non-linear Kalman filter for track parameters estimation in high energy physics. Nucl Instrum Methods Phys Res Sect A Accel Spectrom Detect Assoc Equip 1049:168041. https://doi.org/10.1016/j.nima.2023.168041
14. Esmail W, Stockmanns T, Ritman J (2019) Machine Learning for Track Finding at PANDA https://arxiv.org/abs/1910.07191. Accessed 24 Jun 2025.
15. Strandlie A, Frűhwirth 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
16. Stone J, Gohara D, Shi G (2010) Opencl: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12:66–72. https://doi.org/10.1109/MCSE.2010.69
17. 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
18. Trott CR, Lebrun-Grandié D, Arndt D et al (2022) Kokkos 3: programming model extensions for the exascale era. IEEE Trans Parallel Distrib Syst 33(4):805–817. https://doi.org/10.1109/TPDS.2021.3097283
19. Poenaru A, Mcintosh-Smith S, Lin T (2021) A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application. In: High Performance Computing - 36th International Conference, ISC High Performance 2021, Proceedings. 332–350. https://doi.org/10.1007/978-3-030-78713-4
20. CUDA. https://developer.nvidia.com/cuda-toolkit. Accessed 20 Jan 2025
21. Alpay A (2020) SYCL Beyond OpenCL: The Architecture, Current State and Future Direction of hipSYCL. In: IWOCL '20: Proceedings of the International Workshop on OpenCL. https://doi.org/10.1145/3388333.3388658
22. Intel® oneAPI DPC++/C++ Compiler. https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2025-0/overview.html. Accessed 20 Jan 2025
23. Rohr D, For the ALICE Collaboration (2021) Usage of GPUs in ALICE online and offline processing during LHC Run 3. EPJ Web Conf. https://doi.org/10.1051/epjconf/202125104026
24. Aaij R, Albrecht J, Belous M et al (2020) Allen: a high-level trigger on GPUs for LHCb. Comput Softw Big Sci. https://doi.org/10.1007/s41781-020-00039-7
25. HIP. https://rocm.docs.amd.com/projects/HIP/en/latest/. Accessed 20 Jan 2025
26. Brinkmann KT, Gianotti P, Lehmann I (2006) Exploring the mysteries of strong interactions: the PANDA experiment. Nucl Phys News 16:15–18. https://doi.org/10.1080/10506890600579868
27. Smyrski J et al (2018) Pressure stabilized straw tube modules for the PANDA forward tracker. JINST 13:P06009. https://doi.org/10.1088/1748-0221/13/06/P06009
28. Korcyl K, Konorov I, Kúhn W (2012) Modeling event building architecture for the triggerless data acquisition system for panda experiment at the hesr facility at fair/gsi. J Phys Conf Ser 396(1):012027. https://doi.org/10.1088/1742-6596/396/1/012027

29. Malige A, Korcyl G, Firlej M et al (2022) Real-time data processing pipeline for trigger readout board-based data acquisition systems. IEEE Trans Nucl Sci 69(7):1765–1772. https://doi.org/10.1109/TNS.2022.3186157
30. Spataro S, For the PANDA collaboration (2011) The PandaRoot framework for simulation, reconstruction and analysis. J Phys Conf Ser. https://doi.org/10.1088/1742-6596/331/3/032031
31. triSYCL DPC++. https://github.com/triSYCL/sycl. Accessed 10 Jan 2023
32. CPU Benchmarks. https://www.cpubenchmark.net/. Accessed 12 Jan 2023
33. GPU Specs Database. https://www.techpowerup.com/gpu-specs/. Accessed 12 Jan 2023
34. Amdahl G (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. AFIPS '67 (Spring). 483–485. https://doi.org/10.1145/1465482.1465560
35. Korcyl G, Korcyl P (2020) Towards lattice quantum chromodynamics on FPGA devices. Comput Phys Commun 249:107029. https://doi.org/10.1016/j.cpc.2019.107029

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.