An empirical performance-portability evaluation for Lorentz Vectors computations via SYCL

Monica Dessole[1], **Jolly Chen**[1,2] **Axel Naumann**[1]

[1]CERN, Geneva, Switzerland
[2]University of Amsterdam, Amsterdam, The Netherlands

E-mail: monica.dessole@cern.ch

Abstract. In recent years, we have seen a rapid increase in the variety of computational architectures, featuring GPUs from multiple vendors, a trend that will likely continue in the future with the rise of possibly new accelerators. The High Energy Physics (HEP) community employs a wide variety of algorithms for accelerators which are mostly vendor-specific, but there is a compelling demand to expand the target capabilities of these tools via single-source cross-platform performance-portable abstraction layers, such as SYCL. In this work, we present GenVectorX, a SYCL-based multi-platform extension of the GenVector package of ROOT, that provides classes and functionalities to represent and manipulate particle events. This tool is intended for general usage, but it specifically targets HEP experiments data processing. Moreover, we discuss results showing that the SYCL-based implementation exhibits comparable performance and scalability as the CUDA implementation when targeting NVIDIA GPUs.

1 Introduction

High Energy Physics (HEP) research is characterised by the need for processing and analysing huge amounts of particle collision data coming from the accelerators. ROOT [3] is a popular tool for storing, analyzing and visualizing physics data regarding particle collisions. These collision events are expressed as operations on particles, represented as 4-dimensional time-space vectors, also known as Lorentz Vectors. Within ROOT, the GenVector package contains classes for specialized vectors in 2, 3, and 4 dimensions, and their operations, providing models and capabilities tailored to HEP analysis. The largest source of such data is the Large Hadron Collider (LHC), hosted at CERN in Switzerland, which since its start has reached peaks of 1 PB/s of data generated from physics events that need to be stored and accessed by the physics community to be analysed.

In this scenario, in which scientists need to run their analysis on computing facilities exhibiting different hardware configurations and composition, performance portability plays a crucial role. Such considerations motivate the need for developing performance portable software tailored to the HEP use case. Performance portability of scientific computing applications is gaining importance as hardware becomes more heterogeneous. While NVIDIA GPUs are nowadays standard co-processors, other vendors' GPU architectures are emerging in the High Performance Computing (HPC) scenario, see e.g. LUMI HPC system which is based on AMD GPUs and Aurora HPC system on Intel(R) GPUs. Moreover, the landscape of multi-core CPUs and alternative accelerators is also diversifying, see e.g. FPGAs and the RISC-V initiative [11]. Porting scientific codes to each new platform via hardware-specific programming languages is not sustainable as it costs valuable human time for development and maintenance. Performance portability frameworks overcome this issue by introducing an abstraction layer that can map

Single Instruction Multiple Data (SIMD) parallel models across different hardware platforms. This concept has been successfully demonstrated, for example, by libraries like Kokkos [4], Alpaka [18], as well as SYCL [9], providing abstractions to enable heterogeneous device programming. In this work, we focus on SYCL because of its performance efficiency and portability on different devices, such as CPUs, GPUs, and FPGAs. A number of general purpose studies have detailed the process of migrating from CUDA to SYCL, see [12, 16, 2, 5, 7, 17]. However, to the best of our knowledge, none of them has been focusing the porting of a fundamental HEP library, whose benefits could reflect on a wide variety of related software.

In this paper, we present GenVectorX, an extension of GenVector for execution on NVIDIA GPUs through a native CUDA as well as other platforms via a SYCL implementation of the Lorentz Vector classes that facilitate computations with physical vectors. We compare the performance of one of the most common operations involving Lorentz Vectors on multiple platforms. We carry out an extensive test campaign on NVIDIA GPUs, with particular focus on the performance gap between native CUDA and SYCL code execution. Focusing on an invariant mass computation problem, we study scaling and demonstrate that our SYCL implementation can reach performance portability. This paper is organized as follows. Section 2 details the porting procedure and provides quantitative results about code divergence. Performance analysis results are presented in Section 3, and Section 4 concludes this treatise.

2 Migration process

2.1 Computational models

SYCL is a Khronos Group language standard that enables code for heterogeneous and offload processors, to be written using modern ISO C++, providing APIs and abstractions for finding devices (e.g. CPUs, GPUs, FPGAs) on which code can be executed, and to manage data resources and code execution on those devices. The SYCL kernel consists of the main computational kernel, which can be expressed as a C++ lambda function or as a functor object, the argument values associated with the kernel, and the parameters defining index range. Similarly to the CUDA execution model, the SYCL index hierarchy also consists of a 1, 2, or 3-dimensional grid of work-items, corresponding to the single execution threads. These work-items are grouped into equal sized thread groups called work-groups. Beside the main global memory, SYCL defines a local memory which is shared among all the work-items within a work-group and which can be exploited to enhance data reuse. Depending on the implementation and the hardware availability, this shared local memory can be mapped into different physical memories. Furthermore, the work-items within a work-group can be synchronized with the use of barriers, but synchronization across different work-groups is not possible. The SYCL standard defines two abstractions for declaring and accessing data on devices with different memory contexts: buffers and Unified Shared Memory (USM) pointers. With buffers, the data management is handled entirely by the SYCL runtime and access to the underlying data is permitted via accessors. Buffers are accessible from both the host and the device. The user needs to create an accessor that defines the type of access (read-only, write-only, or read-write), which is information that the runtime uses to determine the data dependencies and necessary memory transfers. For USM pointers, there are three different allocation types: host, device, and shared. These pointers allow for direct dereferencing and fine-grained control over the ownership, but the data dependencies and copy operations need to be handled by the user.

2.2 SYCL and CUDA extentions

With C++ being the programming language of choice, almost all classes and methods can be ported to SYCL and CUDA in a straightforward manner by providing macros and wrapper functions that encapsulate the differences between the host code for the two programming models. The main necessary changes involve mathematical functions. In fact, in SYCL the OpenCL math functions are available in the namespace `sycl::` on host and device with the same precision guarantees as defined in the OpenCL 1.2 specification document [8] for host and device. In GenVectorX, all basic mathematical functions belong to the namespace `ROOT::Experimental::` and they are are compiled and guarded according to a macro definition. Thus, when SYCL is enabled, the mathematical functions corresponds to their SYCL equivalent. Similarly, when CUDA is enabled, all mathematical functions are defined according to their CUDA equivalent. Unlike SYCL, where classes and methods does not need to be decorated to be called on the device, CUDA requires the use of both `__host__` and `__device__` decorators, indicating that the classes and methods can be called by both the host and the device. In order to maintain a single source for CUDA, SYCL and standard CPU, we use instead decorators defined as macros which are defined as blank whenever CUDA is disabled.

The strategies described so far allow to have a single source code defining classes and functionalities to manipulate physicals vectors that can be used in code running on both host and device. Such object

```
template <class Scalar, class LVector>
__global__ void InvariantMassesKernel
(LVector *v1, LVector *v2, Scalar *m,
    size_t N)
{
    int id = blockDim.x * blockIdx.x +
        threadIdx.x;
    if (id < N)
    {
        LVector w = v1[id] + v2[id];
        m[id] = w.mass();
    }
}
```

Figure 1: Invariant Masses CUDA Kernel

```
template <class Scalar, class Vector>
class InvariantMassesKernel
{
public:
    InvariantMassesKernel
    (LVector *v1, LVector *v2, Scalar *m,
         size_t n)
        : d_v1(v1), d_v2(v2), d_m(m), N(n
            ) {}

    void operator()(sycl::nd_item<1> item
        ) const
    {
        size_t id = item.get_global_id().
            get(0);
        if (id < N)
        {
            LVector w = d_v1[id] + d_v2[
                id];
            d_m[id] = w.mass();
        }
    }

private:
    LVector d_v1;
    LVector d_v2;
    Scalar d_m;
    size_t N;
};
```

Figure 2: Invariant Masses SYCL function object

can be included in user defined code targeting a CUDA or SYCL device. In order to provide user-ready functionalities, we define higher level kernel functions to carry our most common computations in High energy physics analysis. Such functions are defined as `__global__` CUDA kernels and, for what concerns SYCL, as function objects. As an example, we detail a function kernel that will be used later in the performance analysis, namely the Invariant Masses kernel. The `InvariantMasses` function returns the invariant mass of two particles expressed in any 4-dimensional coordinate system. Figures 1 and 2 show the CUDA kernel and the SYCL object function definitions, respectively. It is worth noticing that both kernels can handle a template Lorentz Vector object that exposes mass computation via the method `m()`.

2.3 Code Similarity

In this paper, our analysis focuses on the impact of manual code specialization upon developers with regards to code maintenance. We measure this impact using a version of the code similarity metric [10, 13] which represents the average pairwise distance between source codes used to target different platforms. For the sake of clarity, let us formally introduce the following terminology: a *problem* is an input to application, with a correctness test and observable performance; an *application* is a collection of software that can run problems on one or more platforms; a *platform* is a collection of hardware and software on which an application runs problems. Code similarity is calculated as:

$$CS(a,p,H) = 1 - \binom{|H|}{2}^{-1} \sum_{(i,j)\in H\times H} d_{i,j}(p,a) \tag{1}$$

where $d_{i,j}(p,a)$ represents the distance between the source code required to solve problem p using application a on platforms i and j (from platform set H). As done in [16, 14], we adopt the Jaccard distance.

Table 1 shows code similarity of GenvectorX in both its CUDA and SYCL implementations against GenVector, its pure C++ counterpart. Both implementations show a very high similarity score, proving

to have low impact on maintenance. However, SYCL implementation is still preferable for its ability of targeting multiple backends.

Table 1: Code Similarity against pure C++ code

Similarity	Platform	Problem
0.9694	CUDA	Invariant Masses
0.9715	SYCL	Invariant Masses

3 Performance analysis

3.1 Experimental setup

We compare the GenVectorX library on three different computing environment:

1. NVIDIA GeForce RTX 3060 using CUDA 12.2
2. NVIDIA L4 using CUDA 12.3
3. NVIDIA A100 40GB PCIe using CUDA 12.2
4. AMD MI250X using ROCm 5.3.3

In the following section we conduct experiments aimed at showing whether performance portability is achieved or not, focusing on the Invariant Masses example. We compare two different implementations of SYCL compilers, i.e. Intel(R) oneAPI Toolkit [6] and AdaptiveCPP [1]. To determine how effective SYCL is in optimizing data transfers, we select two strategies to compare: buffers with implicit transfers via accessors (BUF) and USM device pointers with explicit transfers (PTR). Moreover, we investigate the root causes of observed performance gaps.

3.2 Scaling: Kernel Execution Time

We evaluate the performance portability of SYCL by evaluating its scaling and comparing it against that of CUDA. Results are shown in Figures 3, 4, 5 and 6. Each figure shows the execution time against the input size. Results are shown for native CUDA, for Intel(R) OneAPI compiler with the use of device pointers and USM, as well as for AdaptiveCPP compiler with the use of USM pointers and buffers. Only the computational kernel evaluating Invariant Masses is measured. For every measurement, the timing is evaluated as the best execution time out of three runs executed sequentially. Results do not highlight significant differences between USM pointers and buffers. CUDA outperforms SYCL, whatever compiler is adopted. AdaptiveCPP is outperformed by OneAPI implementation.

3.3 NVIDIA GPUs: Total Execution Time Breakdown

In order to better appreciate the difference between execution timing of the different implementations, the code executed on NVIDIA GPUs has been profiled with NVIDIA NSight profiler. Results are shown in Figures 7, 8 and 9. The graph plots the time spent in kernels, memory operations, and CUDA API calls separately. Since the CUDA API calls for the two SYCL implementations are named differently, we also combined these into 5 categories for clarity: event, kernel, memory, module, and stream operations. The CUDA memory operations refer to memory transfer time, while the CUDA memory API calls refer to the time spent on setting up the memory operations. The CUDA memory API calls include (de)allocations and copying. Note that the CUDA API calls are executed by the CPU, while the kernels are executed on the GPU, so there is some overlap in runtime that is not illustrated. It is interesting to point out how the performance gap on RTX3060 and L4 (Figures 7, 8) seems to be caused by the kernel execution itself.

4 Conclusion

In this work, we have described our efforts for creating GenVectorX, a SYCL and CUDA version of GenVector, a C++ package providing classes and functionality to particles involved in collisions in High Energy Physics research. This package is part of the ROOT library, a tool for storing, analyzing and visualizing physics data regarding particle collisions adopted by physicists from all over the world. With

Figure 3: RTX3060

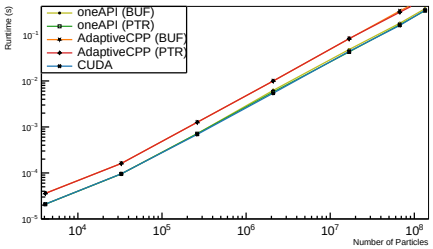


Figure 4: L4

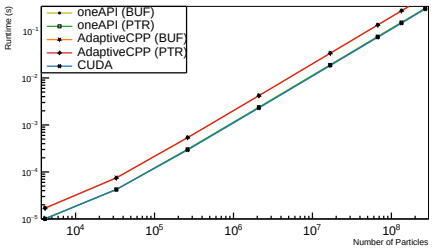


Figure 5: A100

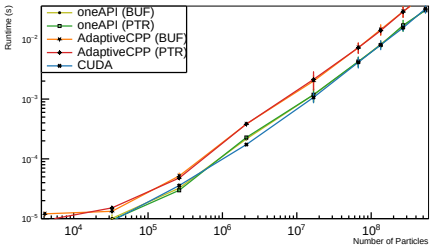


Figure 6: MI250X

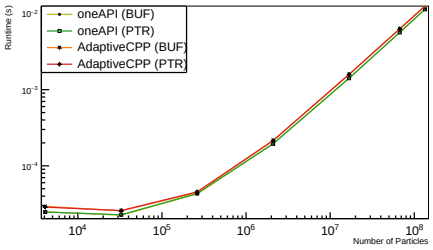
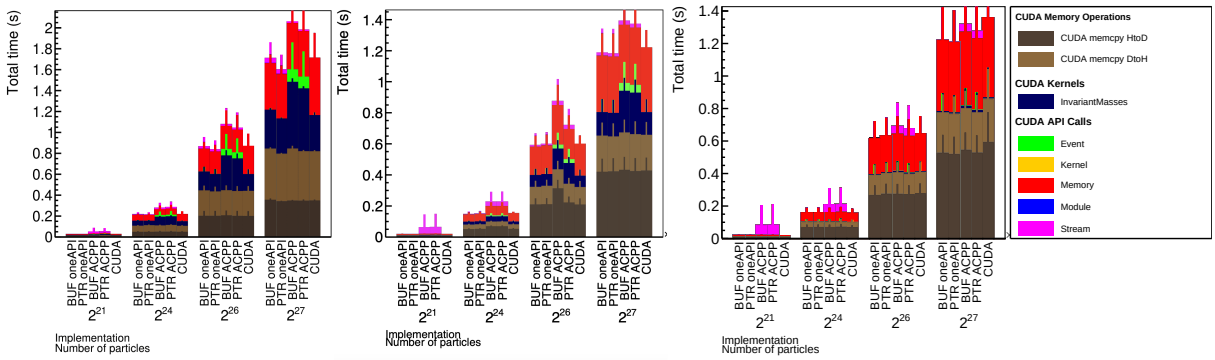


Figure 7: RTX3060

Figure 8: L4

Figure 9: A100



this project, we have explored the potentiality of SYCL as performance portable framework to migrate and modernize the fundamental GenVector package. We have carried out an extensive test campaign, showing that SYCL can achieve competitive performance with respect to CUDA. Furthermore, we have compared two main different implementations of SYCL compiler, namely AdaptiveCPP and Intel(R) OneAPI, showing that it is possible to achieve comparable performance with these tools. It is worth to remark that the use of two compiler has been particularly helpful in the context of code debugging. Moreover, SYCL has confirmed its performance portability capacity with near-one code similarity. These primary results encourage us to pursue the porting project to a production-ready level, which will next target the integration among the other packages in the ROOT library, in particular with user-familiar tools such as ROOTDataFrame (RDF) [15].

References

[1] A. Alpay, B. Soproni, H. Wünsche, and V. Heuveline. Exploring the possibility of a hipsycl-based implementation of oneapi. In *International Workshop on OpenCL*, IWOCL'22, New York, NY, USA, 2022. Association for Computing Machinery.

[2] A. Bagusetty, A. Panyala, G. Brown, and J. Kirk. Towards cross-platform portability of coupled-cluster methods with perturbative triples using sycl. In *2022 IEEE/ACM International Workshop*

on Performance, Portability and Productivity in HPC (P3HPC), pages 81–88, 2022.

[3] R. Brun and F. Rademakers. Root — an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1):81–86, 1997. New Computing Techniques in Physics Research V.

[4] H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[5] S. Christgau and T. Steinke. Porting a legacy cuda stencil code to oneapi. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*, pages 359 – 367, 2020.

[6] I. Corporation. oneapi base toolkit, Accessed on 28 November 2023.

[7] M. Costanzo, E. Rucci, C. García-Sánchez, M. Naiouf, and M. Prieto-Matías. Migrating cuda to oneapi: A smith-waterman case study. In I. Rojas, O. Valenzuela, F. Rojas, L. J. Herrera, and F. Ortuño, editors, *Bioinformatics and Biomedical Engineering*, pages 103–116, Cham, 2022. Springer International Publishing.

[8] T. K. O. W. Group. The opencl specification, version 1.2.19, 2012.

[9] T. K. S. W. Group. Sycl 2020 specification (rev. 8), 2023.

[10] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey. Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 24–36, 2018.

[11] M. A. Islam and K. Kise. Resource-efficient risc-v vector extension architecture for fpga-based accelerators. In *Proceedings of the 13th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART '23, page 78–85, New York, NY, USA, 2023. Association for Computing Machinery.

[12] P. Nguyen, P. Nayak, and H. Anzt. Porting batched iterative solvers onto intel gpus with sycl. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1048–1058, New York, NY, USA, 2023. Association for Computing Machinery.

[13] S. Pennycook, J. Sewall, and V. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, 2019.

[14] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith. Navigating performance, portability, and productivity. *Computing in Science & Engineering*, 23(5):28–38, 2021.

[15] D. Piparo, P. Canal, E. Guiraud, X. Valls Pla, G. Ganis, G. Amadio, A. Naumann, and E. Tejedor Saavedra. RDataFrame: Easy Parallel ROOT Analysis at 100 Threads. *EPJ Web Conf.*, 214:06029, 2019.

[16] E. M. Rangel, S. J. Pennycook, A. Pope, N. Frontiere, Z. Ma, and V. Madananth. A performance-portable sycl implementation of crk-hacc for exascale. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1114–1125, New York, NY, USA, 2023. Association for Computing Machinery.

[17] L. Solis-Vasquez, E. Mascarenhas, and A. Koch. Experiences migrating cuda to sycl: A molecular docking case study. In *Proceedings of the 2023 International Workshop on OpenCL*, IWOCL '23, New York, NY, USA, 2023. Association for Computing Machinery.

[18] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann. Alpaka - an abstraction library for parallel kernel acceleration, May 2016.