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

Monica Dessole, Jolly Chen, Axel Naumann

ROOT team,
EP SFT, CERN

13th March 2024

1. Background
2. Migrating from CUDA to SYCL
3. Numerical Experiments
4. Conclusions and future work

Heterogeneous Computing and Portability

Current hardware accelerators in modern (pre-)exascale supercomputers

Machine	GPU vendors	language
Frontier	AMD	ROCm
Aurora	Intel	oneAPI
LUMI	AMD	ROCm
Polaris	NVIDIA	CUDA
Perlmutter	NVIDIA	CUDA

Two Types of Portability:

Functional Portability

The ability for a single code to run anywhere.

Performance Portability

The ability for a single code to run well anywhere.

Performance Portability Frameworks

Figure: Hardware support of portability layers[1]

	CUDA	HIP	OpenMP Offload	Kokkos	dpc++ / SYCL	alpaka	std::par
NVidia GPU							*nvc++*
AMD GPU				*feature complete for select GPUs*		*hip 4.0.1 / clang*	
Intel GPU		*CHIP-SPV early prototype*		*native and via OpenMP target offload*		*prototype*	*oneAPI::dpl*
multicore CPU							*g++ & tbb*
FPGA						*via SYCL*	

[1] Mohammad Atif et al. "Evaluating Portable Parallelization Strategies for Heterogeneous Architectures in High Energy Physics". In: *arXiv preprint* (2023).

GenVector Package

GenVector is a large package ($\sim$ 11k lines) within ROOT enabling fundamental operations for HEP analysis

- 2, 3 and 4 dimensional physical vectors
- coordinate systems: Cartesian, Polar, and so forth
- user can specify the underlying scalar type, say single or double precision floats
- more advanced operations: rotations, Lorentz and Poincare transformations

GenVectorX

Extend GenVector to parallel execution on NVIDIA GPUs via CUDA and other backends via SYCL, while retaining performance execution and user friendly API.

SYCL Computational Model

SYCL is a cross-platform abstraction layer that for a “single-source” code using modern ISO C++. Two notable implementations:

- Intel(R) oneAPI Toolkit
- AdaptiveCPP[2] (before openSYCL/hipSYCL)

Memory management

- buffers and accessors (BUF), handled entirely by the SYCL runtime
- unified shared memory pointers (PTR), explicitly defined by the user

Kernel definition

- lambda functions, optionally named
- named function objects, same functionality as any C++ function object

[2] Aksel Alpay et al. “Exploring the Possibility of a HipSYCL-Based Implementation of OneAPI”. In: *International Workshop on OpenCL*. IWOCL’22. ACM, 2022.

From CUDA to SYCL[3]

Figure: Index Hierarchy

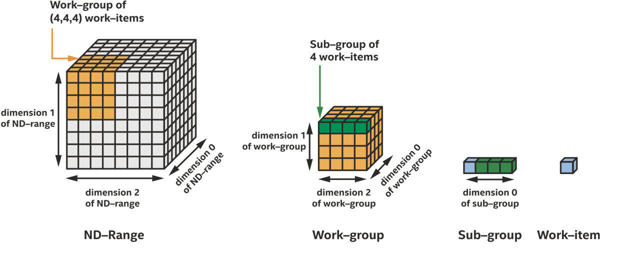


Table: Execution model

CUDA	SYCL
Thread	work-item
Warp	sub-group
Block	work-group
Grid	ND-range

[3] James Reinders et al. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. 2021.

Example: Invariant Masses

Figure: SYCL function object

```
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: CUDA Kernel

```
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();
    }
}
```

Code Similarity

Given a problem and a set of platforms (here CUDA, SYCL and CPU), code similarity is a measure of similarity between code bases that takes value in $[0, 1]$.

Table: Code Similarity against pure C++ code

Similarity	Platform	Problem
0.9694	CUDA	Invariant Masses
0.9715	SYCL	Invariant Masses

High similarity $\implies$ only specialize small regions of code with CUDA and SYCL
The effort in maintaining SYCL code is comparable with CUDA code, but

CUDA $\implies$ NVIDIA GPUs

SYCL $\implies$ multiple backends

Experimental Setting

Four computing environments:

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

Questions:

- Can we achieve performance portability? On which platforms?
- If any difference is found, can we identify its root cause?
- Is there any difference between SYCL memory management strategies?

Scaling: Kernel Execution Time

Figure: RTX3060

Figure: L4

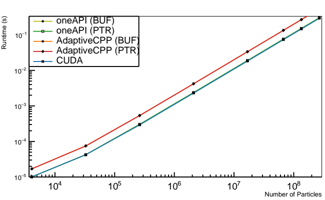


Figure: A100

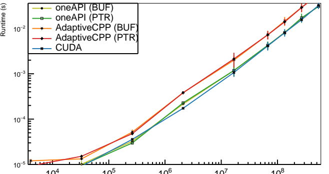
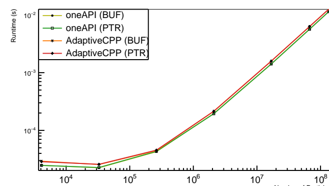


Figure: MI250X



NVIDIA GPUs: Total Execution Time Breakdown

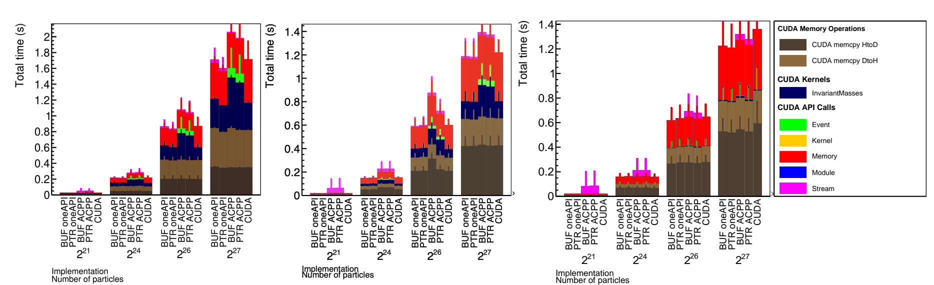


Figure: RTX3060

Figure: L4

Figure: A100

What we did so far

- We detailed the migration to both SYCL and CUDA of a large, complex, C++ code base, providing guidance and insights regarding the analogies and differences between the two frameworks for other developers interested in migrating their own codes.
- We evaluated code divergence of GenVectorX, to estimate the benefits in maintaining a single source code without specializing regions of code for specific targets.
- We empirically showed performance portability of the migrated SYCL code on different platforms and architectures.

What needs to be done

- Integration within ROOT is work in progress.
- We will carry out performance evaluation on other platforms and architectures, namely Intel GPUs and Risc-V.
- We will extend GenVectorX to tackle jagged arrays.
- We will implement SYCL support in cling for just-in-time compilation.
- Open problem: dealing with filtering and skimming.

More infos:

- Code is available of GitHub: `https://github.com/root-project/genvectorx`
- Preprint: `https://arxiv.org/abs/2312.02756`

Thank you for listening!

References I

Alpay, Aksel et al. “Exploring the Possibility of a HipSYCL-Based Implementation of OneAPI”. In: *International Workshop on OpenCL*. IWOCL’22. ACM, 2022.

Atif, Mohammad et al. “Evaluating Portable Parallelization Strategies for Heterogeneous Architectures in High Energy Physics”. In: *arXiv preprint* (2023).

Harrell, Stephen Lien et al. “Effective Performance Portability”. In: *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2018, pp. 24–36.

Reinders, James et al. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. 2021.

Code Similarity: Evaluation

Code similarity is calculated as[4]

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

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). We adopt the Jaccard distance defined as $d_{i,j}(p,a) = 1 - s_{i,j}(p,a)$, where

$$s_{i,j}(p,a) = \left| \frac{c_i(a,p) \cap c_j(a,p)}{c_i(a,p) \cup c_j(a,p)} \right|. \tag{2}$$

Here, c_i and c_j represent the set of source lines required to compile application a and execute problem p for platforms i and j, respectively.

[4] Stephen Lien Harrell et al. "Effective Performance Portability". In: *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2018, pp. 24–36.

Hardware Specs

The main features of the GPUs that are relevant in our case are summarized here:

Table: GPUs specification

	RTX3060	L4	A100	MI250X
Vendor	NVIDIA	NVIDIA	NVIDIA	AMD
Architecture	Ampere	Ada Lovelace	Ampere	CDNA2
CUDA Capability	86	89	80	-
Global Memory (GB)	12	24	40	128
Peak FP64 (TFLOPs)	6.3	15.5	9.7	47.8
Peak Bandwidth (GB/s)	360.0	300.0	1555.0	3200.0