Application of Portable Parallelization Strategies for GPUs on track reconstruction kernels

Martin Kwok, Matti Kortelainen, Giuseppe Cerati, Alexei Strelchenko, Oliver Gutsche

Fermi National Accelerator Laboratory, Batavia, IL, USA

E-mail: kkwok@fnal.gov

Abstract. Utilizing the computational power of GPUs is one of the key ingredients to meet the computing challenges presented to the next generation of High-Energy Physics (HEP) experiments. Unlike CPUs, developing software for GPUs often involves using architecture-specific programming languages promoted by the GPU vendors and hence limits the platform that the code can run on. Various portability solutions have been developed to achieve portable, performant software across different GPU vendors. Given the rapid evolution of these portability solutions, an early adoption of them in simple HEP testbed applications will help us understand the strengths and weaknesses of respective approaches.

We apply several portability solutions, including Alpaka, Kokkos, SYCL and std::execution::par, on kernels for track propagation extracted from the mkFit project. We report on the development experience of the same application with different portability solutions, as well as their performance on GPUs, measured as the throughput of the kernels, from different manufacturers such as NVIDIA, AMD and Intel.

1. Introduction

Heterogeneous computing is one of the key components to meet the computing challenge of next generation of HEP experiments. Taking HL-LHC upgrade as an example, which aims at collecting 10 times more data than the LHC has collected in approximately the same number of operating years. The required total CPU hours to support the ambitious science goal of HL-LHC is projected to exceed the amount that can be obtained with a flat budget increase of 10-20% [1]. Given the discrepancy, the usage of compute accelerators, such as GPUs, could provide the additional computing power to meet the resource demand.

However, it is far from a trivial task for the HEP community to adopt the use of compute accelerators, with millions line of code already written for x86 CPU architecture. Converting domain-specific CPU algorithms into efficient kernels that can take advantage of the massive parallelism in GPU is a task that requires significant development resources. Typical HEP experiments require hundreds of such kernels with no dominant hot-spots, each may use custom data objects, and need to be developed and maintained by domain experts and executed on hundreds of different computing sites.

Given the above constrains, it is necessary for the adoption of compute accelerators in HEP to be portable across multiple accelerator platforms with minimal code changes while maintaining good performance. Achieving performance portability will avoid duplication of development efforts, maintenance of multiple code base, and gain access to more computing resources that uses different accelerators. In fact, the recently deployed and planned exa-scale HPC systems around the world, summarized in Table. 1, have a diverse set of CPU/GPU

architectures and vendor providers. The proliferation of architecture choices of HPC systems is likely to continue in their pursue for best computing performance. These exa-scale systems could provide the additional computational resources for meeting future computing challenges, if the HEP software could make use of them.

HPC system	Location	CPU	GPU	Peak Flop/s	Year
Perlmutter, NERSC	U.S.	AMD(x86)	Nvidia	94 PFlop/s	2020
Aurora, ANL	U.S.	Intel(x86)	Intel	>1 ExaFlop/s	2023
Frontier, ORNL	U.S.	AMD(x86)	AMD	1.69 ExaFlop/s	2021
El Capitan, LLNL	U.S.	AMD(x86)	AMD	1.5 ExaFlop/s	2023
Leonardo, Cineca	Italy	Intel(x86)	Nvidia	256 PFlop/s	2021
LUMI, CSC	Finland	AMD(x86)	AMD	429 PFlop/s	2021
Alps, CSCS	Switzerland	Nvidia(arm)	Nvidia	4.7 PFlop/s	2020

Table 1. Example list of recently deployed/planned major HPC systems and the CPU/GPU vendor selected. Peak flop taken from the TOP500 November 2022 list if available.

There has been active efforts, coming from both industry and academia, to develop portable parallelization solutions, which are summarized in Figure. 1. Many of the solutions are rapidly changing in timescales of a month, for new features, better compiler supports or new backends. Several different approaches are being attempted among these solutions, including using compiler pragmas (OpenMP/OpenACC), C++ libraries (Alpaka [2], Kokkos [3, 4]) and language extension (SYCL, std::par). Each approach inherits certain advantages and disadvantages, which may have very different implications if a HEP experiment wants to adopt it. In this work, we will examine the performance of Kokkos, SYCL, Alpaka and std::par on different GPU backends, using an example test-bed application in the HEP context.

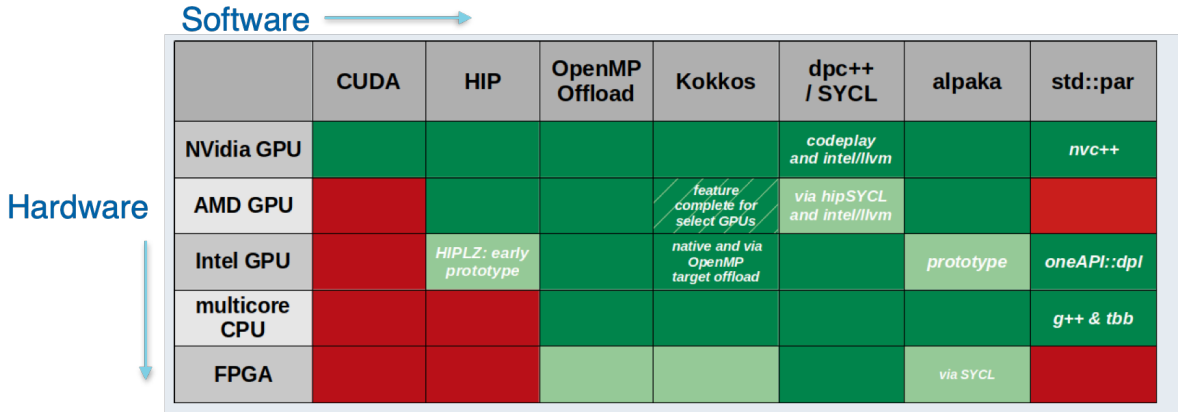

Figure 1. Summary of hardware supports for different portability solutions, as of Oct 2022. Green indicates officially supported, red indicates un-supported, while light-green indicates under-development.

2. The propagation-to-r (`p2r`) program

Reconstruction of tracks of charged particles is one of the most computational intensive task in collider experiments such as ATLAS and CMS at the LHC, which makes it the prime targets for parallelization investigations. We developed a standalone mini-application, called propagation-to-r (`p2r`) [5], which performs the core math of parallelized track reconstructions. The kernel aims at building charged particle tracks in the radial direction under a magnetic field from detector hits, which involves propagating the track states and performing Kalman

updates after the propagation. The kernels are implemented based on a more realistic application, called `mkFit` [6], which performs vectorized CPU track fitting. Together with an analogous project, propagation-to-z (`p2z`)[7], the two programs form the backbone of track fitting kernels.

The `p2r` program uses a simplified program workflow, which processes a fixed number of events (`nevts`) with the same number of tracks in each events (`ntrks`). A fixed set of input track parameters is smeared randomly and then used for every tracks. All track computations are implemented in a single GPU kernel. The input data are structured as an array-of-structure-of-array (AOSOA). The total number of tracks to process equals to `ntrks` $\times$ `nevts`, in which the tracks in each event are grouped into batches of size `bsize`. The structure of array (SOA) structure that contains a batch of tracks is called MPTRK. Figure 2 shows the data structure used in the p2r program.

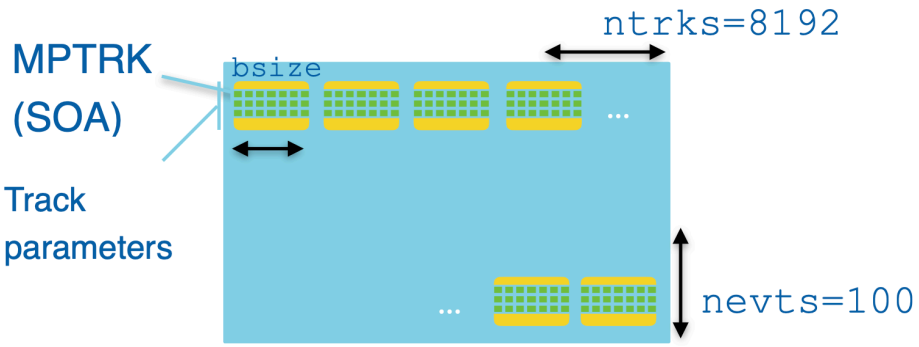

Figure 2. Illustration of the data structure used in the `p2r` program.

3. Overview of the explored portable programming models

In this section, we give a brief overview for each of the portable programming models tested in this work.

3.1. Alpaka

Alpaka [2] is a single-source, header-only C++ parallelization library. Initially, it started out as a thin abstraction layer over CUDA, and continues to have similar API structures to CUDA. The general approach is to construct an abstraction layer on top of CUDA concepts, such as work division, memory operations, to provide portability. In particular, compute kernels are templated with an accelerator object class, which provides easy switching of accelerator back-ends at compile time.

3.2. Kokkos

Similar to Alpaka, Kokkos [4, 3] is also a single source C++ template library. One major difference with Alpaka is that Kokkos aims to be more descriptive on the parallelism, rather than prescriptive. With a more descriptive model, developers are asked to express the algorithm in general parallel programming concepts, which are then mapped to hardware by the Kokkos framework. Therefore, users do not explicitly map the loop iterations to threads, which allows more flexible support for hardware that is not a close match of GPU-centric programming model. Kokkos also provides a data structure for multidimensional arrays, `Kokkos::View`, to handle efficient data layout for both GPU and CPU.

3.3. SYCL

SYCL [8] is a *specification* for single-source C++ programming model for heterogeneous computing, for which different compilers or libraries can make concrete implementations. One major compiler that supports SYCL is DPC++ [9], which is developed by Intel to support Intel's accelerator hardware, including GPUs and FPGAs. `OpenSYCL`(former hipSYCL) is another recent, open-sourced effort to support NVIDIA, AMD and Intel GPUs via SYCL. In general, SYCL compilers are under more active developments. Both Alpaka and Kokkos are developing SYCL backends to support Intel hardware.

3.4. std::par

The C++ ISO standard also plans to extend the existing support to parallelization algorithms. The basic functionalities such as `std::execution` and `std::for_each` has been supported since C++17 [10], with more supporting features for concurrency, such as `std::atomic<T>` and `std::atomic_ref<T>`, integrated in C++20 [11]. Some of the limitations with the `std` includes the lack of explicit device memory management (i.e. only support unified shared memory), async operations and the lack of support for user-defined kernel launch parameters. Although compiler supports are still in early development, NVIDIA is supporting `std::par` algorithms via a closed-source compiler(`nvc++`) while Intel's GPU can be supported by using oneAPI libraries.

4. Porting experience

In this section, we summarize the porting experience of `p2r` with different portability technologies. Our reference implementation is vectorized on CPU, and parallelized with Intel Threading Building Blocks (oneTBB). We first completed the implementation of a CUDA version, from which we determined the operating parameters via profiling. Then we converted the CUDA version of `p2r` to different portability solutions. For most solutions, this step involved changing the corresponding API for data handling and kernel launching, while the core kernel code largely remained the same. A considerable amount of time was also spent on configuration of the software stack for compilation and build options, which was often longer for AMD and Intel GPU backends and could sometimes require support from the developers of the portability layer. Table 2 shows different versions of compilers used in this work. Last but not least, we note that the majority portion of development time was spent on profiling to understand if a specific implementation is converted optimally.

A more detailed comparison on the advantages/disadvantages of each portability solutions can be found in here [12].

	CUDA	HIP	Alpaka (v0.9.0)	Kokkos (3.6.1)	SYCL	std::par
NVIDIA GPU	cuda/11.6.2	N/A	cuda/11.6.2 gcc/9.2.0 nvcc	cuda/11.6.2 gcc/8.2.0 nvcc	cuda/11.6.2 intel/llvm-sycl	nvc++/22.7
AMD GPU	N/A	rocm/5.2.0	rocm/5.1.3 gcc/9.2.0 hipcc	rocm/5.1.3 gcc/9.2.0 hipcc	rocm/5.1.3 intel/llvm-sycl	N/A

Table 2. Summary of compiler versions used for different GPU backends in this work. The llvm-sycl branch is compiled with the commit [13]. We note that HIP could be compiled for NVIDIA GPUs as well and that oneAPI toolkit recently included a plugin that enables std::par to be used on AMD GPUs, however, both cases are not tested in this work.

5. Measurement and results

Measurements were performed on the computing nodes with a NVIDIA A-100 GPU and AMD MI-100 GPU in the Joint Laboratory for System Evaluation (JLSE) hosted at the Argonne National Laboratory. Table 3 summarizes some key performance specifications of the two GPUs.

Vendor	Model	Peak FP64 flops	Peak Memory bandwidth	N cores
AMD	MI-100	11.5 T flops	1.2 TB/s	7680
NVidia	A-100	9.7 T flops	1.9 TB/s	6912

Table 3. Comparison of key performances specifications for GPUs used in the measurements.

Different implementations of the `p2r` program were compiled to execute on the two GPUs, using the same operation parameters. Each kernel corresponds to the computation of 4 million tracks. The metric for comparison is the overall track processing throughput of the kernel, which is defined as the number of processed tracks divided by the duration of the kernel. Time required for data transfer between the host and device is excluded. Before each measurement, two warm-up runs are executed to reach a more stable hardware condition for computation. The average of 10 measurements, and the corresponding standard deviations, is reported in Figure 3 for each technology. The throughput obtained from portability technologies are compared as a fraction of the throughput reached by the platform-native implementation.

On both A100 and MI-100 GPUs, Alpaka and Kokkos achieved much better performance than SYCL and std::par. In particular for the A100 GPU, Alpaka and Kokkos's performance is very close to the native CUDA version, whereas SYCL and std::par are approximately factor of 10 and 2 slower, respectively. We note, however, that both Alpaka and Kokkos suffer from a $\sim$ 40% slow down if the kernel launch parameters, such as number of threads per block and number of blocks per grid, are left to be determined by the automatic heuristic of the portability layer. Results shown in Figure 3 are obtained by explicitly choosing the same launch parameters and register per thread values as in the native CUDA version.

On MI-100, Alpaka version achieved 23% higher throughput than the native HIP implementation. While it is possible that the portability layer could introduce better optimization than the native implementation, further profiling work is required to confirm the cause for the better performance observed in this case.

The poor performance of the SYCL implementation on both tested GPUs is not fully understood. Initial profiling results show orders of magnitude more instructions were executed in the kernel, inducing much more memory traffic and hence latency. Beyond understanding the results on GPU, we plan on performing further studies with CPU multi-core backends of these portability technologies, as we envisioned that an efficient CPU backend remains critically important for most HEP applications in the near future.

6. Conclusion

In summary, we implemented a track propagation application with four different portability technologies, Alpaka, Kokkos, SYCL, and std::par, and measured their performance on data center-grade NVIDIA and AMD GPUs. This is an early investigation of implementing HEP-specific algorithms using the latest portability technologies, which could play an important role for GPU usage for HEP experiment in the near future. Together with other test-bed applications investigated by the HEP-CCE, we hope this can inform the HEP community on the advantages and potential issues on the relevant portability technologies.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Office of High Energy Physics, High Energy Physics Center for Computational Excellence (HEP-

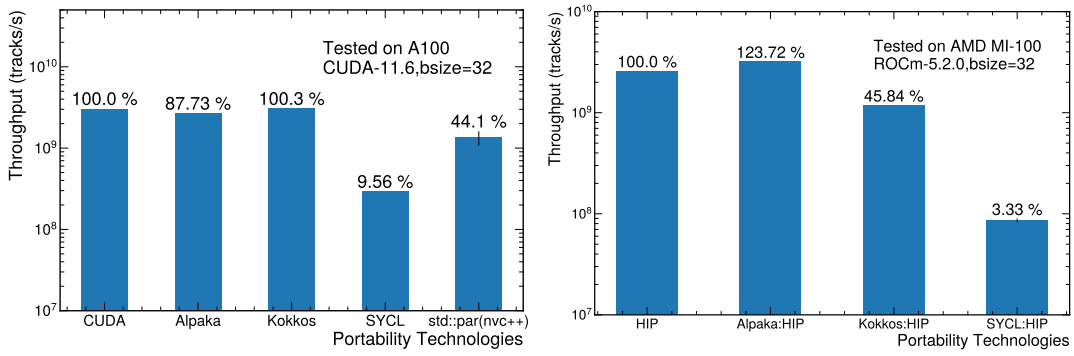

Figure 3. Overall kernel throughput for different portability technologies on measured on a NVIDIA A-100 GPU(left) and a AMD MI-100 GPU(right) on JLSE. The average result of 10 measurements is reported as the nominal values, and the corresponding standard deviations is used as uncertainties, which is typically 2 orders of magnitude smaller than the average value.

CCE) at Fermi National Accelerator Laboratory under B&R KA2401045. This work was also supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of High Energy Physics, Scientific Discovery through Advanced Computing (SciDAC) program.

We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

References

[1] CMS collaboration 2022 CMS Phase-2 Computing Model: Update Document Tech. rep. CERN Geneva URL https://cds.cern.ch/record/2815292
[2] Matthes A, Widera R, Zenker E, Worpitz B, Huebl A and Bussmann M 2017 (*Preprint* 1706.10086) URL https://arxiv.org/abs/1706.10086
[3] Trott C R, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, Gayatri R, Harvey E, Hollman D S, Ibanez D, Liber N, Madsen J, Miles J, Poliakoff D, Powell A, Rajamanickam S, Simberg M, Sunderland D, Turcksin B and Wilke J 2022 *IEEE Transactions on Parallel and Distributed Systems* **33** 805–817
[4] Edwards H C, Trott C R and Sunderland D 2014 *Journal of Parallel and Distributed Computing* **74** 3202 – 3216 ISSN 0743-7315 domain-Specific Languages and High-Level Frameworks for High-Performance Computing URL http://www.sciencedirect.com/science/article/pii/S0743731514001257
[5] The p2r program https://github.com/cerati/p2r-tests
[6] Lantz S, McDermott K, Reid M, Riley D, Wittich P, Berkman S, Cerati G, Kortelainen M, Hall A R, Elmer P, Wang B, Giannini L, Krutelyov V, Masciovecchio M, Tadel M, Würthwein F, Yagil A, Gravelle B and Norris B 2020 *Journal of Instrumentation* **15** P09030–P09030 URL https://doi.org/10.1088/1748-0221/15/09/P09030
[7] The p2z program https://github.com/cerati/p2z-tests
[8] The Khoronos SYCL Working Group 2021 *SYCL 2020 Specification (revision 2)*
[9] The dpc++ compiler https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html
[10] ISO. 2017. ISO/IEC 14882:2017: Programming languages — C++.
[11] ISO. 2020. ISO/IEC 14882:2020: Programming languages — C++
[12] Evaluating portable parallelization strategies for heterogeneous architectures https://indi.to/k7Bsk
[13] Intel llvm/sycl branch https://github.com/intel/llvm/tree/70c2dc6dcf73f645248aa7c70c8cefdabf37e9b7