FERMILAB-CONF-23-080-CMS

Performance portability for the CMS Reconstruction with Alpaka

Andrea Bocci[1], Angela Czirkos[1], Antonio Di Pilato[4], Felice Pantaleo[1], Gabrielle Hugo[1], Matti Kortelainen[2], Wahid Redjeb[1,3], on behalf of the CMS collaboration

[1]CERN, European Organization for Nuclear Research, Meyrin, Switzerland
[2]Fermilab, Fermi National Accelerator Laboratory, Batavia, IL, USA
[3]RWTH Aachen University, III. Physikalisches Institut A, Aachen, Germany
[4]CASUS, Center for Advanced Systems Understanding, Görlitz, Germany

E-mail: `wahid.redjeb@cern.ch`

Abstract. For CMS, Heterogeneous Computing is a powerful tool to face the computational challenges posed by the upgrades of the LHC, and will be used in production at the High Level Trigger during Run 3. In principle, to offload the computational work on non-CPU resources, while retaining their performance, different implementations of the same code are required. This would introduce code-duplication which is not sustainable in terms of maintainability and testability of the software. Performance portability libraries allow to write code once and run it on different architectures with close-to-native performance. The CMS experiment is evaluating performance portability libraries for the near term future.

1. Introduction

The increasing luminosity of the Large Hadron Collider (LHC) in Run 3[1], and subsequently in the High Luminosity Phase of the LHC, will significantly increase the event description complexity due to the higher number of simultaneous proton-proton collisions known as pileup. These new conditions will pose significant challenges for the CMS [2] data taking and data processing framework, and substantial improvements are needed to fulfill the computational timing and memory requirements of the CMS online and offline infrastructures.

Heterogeneous Computing is becoming more and more popular due to the advent of diverse computing resources such as Graphics Processing Units (GPUs) and Field Programmable Gate Array (FPGAs), and can be exploited in computing farms to achieve a better time, cost, and energy to solution. In particular, the CMS experiment is including GPUs in the trigger farms for the Run 3 data taking, and several workflows are being implemented to run on GPUs, exploiting parallelism. Currently, the Pixel Tracks and Vertices reconstruction, the calorimeters local reconstruction, and the Particle Flow clustering [3] are all reconstruction algorithms that will run at the CMS High Level Trigger [4] on GPUs during the Run 3 data taking.

Several architectures are now available: AMD, ARM, IBM Power, and Intel, for the CPU side, and AMD, Intel, and NVIDIA for GPUs. The CMS experiment needs to support this whole range to achieve better performance at a lower cost. However, specialized implementations of

[1] Third data-taking period, beginning in 2022 and ending in 2025 [1]

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

the same code for each architecture would introduce a lot of code duplication, and would make the framework hard to maintain, test, and validate. The CMS experiment is exploring solutions to achieve *performance portability*, exploiting frameworks and libraries that allow writing a single source code that can be compiled and executed on different architectures achieving close-to-native performance. This approach avoids the need of maintaining and testing several implementations of the same algorithm.

Performance portability libraries are a possible solution to achieve performance portability. The CMS collaboration has chosen the Alpaka [5] performance portability library as a solution for the Run 3 data taking. To evaluate the feasibility of the adoption of Alpaka for the CMS reconstruction, in terms of performance, ease of use, and integration, the pixel tracks and vertices reconstruction modules have been developed using this library.

2. Abstraction Library for Parallel Kernel Acceleration - Alpaka

Alpaka [5] is a header-only C++ library that allows performance portability across several computing architectures. To achieve that, Alpaka abstracts the underlying backend implementation and levels of parallelism, defining a hierarchical redundant parallelism model, similar to the one implemented in CUDA.

Currently, Alpaka supports several backends: the serial execution on the host CPU, the parallel execution on the host CPU, through C++ threads, Boost.Fiber [6] and Threading Building Blocks (TBB) [7]. It Supports OpenMP [8] and openACC [9] for the parallel execution on the host CPU and supported GPUs, and the parallel execution on AMD and NVIDIA GPUs with the corresponding HIP [10] and CUDA [11] backends. An experimental SYCL [12] backend is available for Intel and Xilinx devices. Moreover, the Alpaka C++ interface allows to define the user-defined representation of accelerators as C++ entities.

The single source code obtained via Alpaka can be compiled for a single or multiple backends, building an application that supports different platforms and accelerator types in a single binary, allowing the user to choose the one to use at the run-time.

To achieve portability across multiple devices, the memory management implemented by Alpaka is uniform for the different platforms. The memory allocation is performed via smart pointers to avoid memory leaks and manual freeing. Alpaka does not optimize the memory operations between devices, but data are stored in *data-structure agnostic* buffers that support copies between devices. Thus, the user needs to take care of data distribution between devices.

The model adopted by Alpaka, enables the separation of the parallelization strategy from the algorithm. The algorithm is defined by a *kernel* function that is executed by threads. The parallelization strategy is described by the *Accelerator* and the *Work Division.* The accelerator defines the acceleration strategy by mapping the parallelization levels to the hardware. It is important to mention that the Alpaka model exploits parallelism and memory hierarchies on a node at all the levels available in current hardware, and ignores the unsupported levels, allowing the user to program all the hardware types in the same way.

The abstraction introduced by the Alpaka library relies on the definition of constraints a certain type has to fulfill to be usable with the template functions the library provides. These constraints, called concepts in C++, allow defining algorithms that use different objects and types. The Alpaka interface separates the different concepts using namespaces, obtaining an ergonomic programming model quite simple to use.

3. CMS Patatrack Pixel Tracks and Vertices Reconstruction in Alpaka

Alpaka has been tested porting algorithms from the CMS Software (CMSSW), with the goal of obtaining a single source code that can be executed on several devices and with different parallelization strategies, with computing performance close to the native ones. In order to understand the strengths and weaknesses of the library, an exploratory study has been performed porting to Alpaka the Patatrack Pixel Tracks and Vertices reconstruction algorithms [13, 14, 15]. These algorithms perform the track and vertices reconstruction on GPUs, starting from the raw data of the CMS Pixel detector. The first step of the track reconstruction is the *local reconstruction*, where the digitized information of every detector module and of every pixel is unpacked and interpreted in parallel, to build the final *digis* collection. The neighboring digis are then clustered using an iterative process that produces *clusters*. The clusters' shapes are used to determine the *hit position*. To obtain the final track, clusters are linked together to form n-tuplets, that are eventually fitted to obtain the track parameters. Several steps are needed to produce the n-tuplets. First, hits belonging to adjacent pairs of pixel detector layers are connected together, to create the *doublets*. Several selection criteria are applied to check the compatibility and form a doublet; this step is performed in parallel by different threads, starting from the outer hits. Afterward, doublets that share a common hit are tested for compatibility to create a *triplet*. Moreover, all the doublets that have an inner hit on the first pixel layer are marked as root doublets, as well as doublets starting from the second pixel layer without inner neighbors. The root doublets are used as starting point for a Depth-First-Search (DFS) to connect doublets and form the n-tuplets. During the doublets creation, the *Fishbone* [14] mechanism is used to clean possible duplicates, solving the ambiguities by merging the overlapping doublets. The multiple-scattering aware Broken Line fit [14, 16] is used to fit the n-tuplets and obtain the track parameters, to produce the final track object. To reconstruct the vertices, the pixel tracks obtained are clustered together along the beam axis.

The whole process happens on the GPU, and to optimize the coalesced memory access, data are stored in *Structures-of-Arrays*. Eventually, the final pixel tracks and vertices collection can be copied back to the CPU and can be converted to the legacy data format (*Arrays-of-Structures*) on-demand.

To test the Alpaka library, a standalone version of this sequence of algorithms has been prepared, with a lightweight version of the CMSSW framework [17, 18], to emulate the whole infrastructure and understand how to interface Alpaka with CMSSW. This reconstruction module counts tens of small kernels that run one after the other, contrary to standard HPC applications that have few big kernels that run for a long time. Moreover, the algorithms make use of several instructions that may work differently in Alpaka: threads barriers, atomics, etc. Thus, this study allows testing Alpaka on multiple aspects.

4. Computing Performance

The whole pixel tracks and vertices reconstruction modules have been successfully ported to Alpaka, supporting the CPU serial and GPU CUDA backends, but additional backends can be easily added. The physics performance obtained are the same with respect to the native implementation. The computing performance has been tested on a machine equipped with a dual-socket *AMD EPYC 7543* 32-Core, 64 threads, and two *NVIDIA Tesla T4* GPUs. The application has been executed on one GPU, pinning a single CPU socket.

Multiple events can be executed in parallel by different CPU threads on different EDM streams [18], performing asynchronous operations on the same GPU, thus, the performance are evaluated in terms of throughput (events per second). Each stream processes 10k events, and for the GPU implementations, the transfer of the results back to the host is not considered.

Figure 1 shows the results obtained by compiling Alpaka for the CPU serial backend. It shows a good scaling with the number of concurrent CPU threads. The current GPU native version of the reconstruction module has two main optimizations: the stream-ordered asynchronous memory operations, introduced in CUDA 11.2, that allow ordering the memory allocations and deallocations within a certain stream and avoid global synchronisations, and a custom caching allocator that exploits memory pools to reduce the cost of the API calls [17]. At the time of writing, the stream-ordered memory operations are also implemented in Alpaka, while a custom

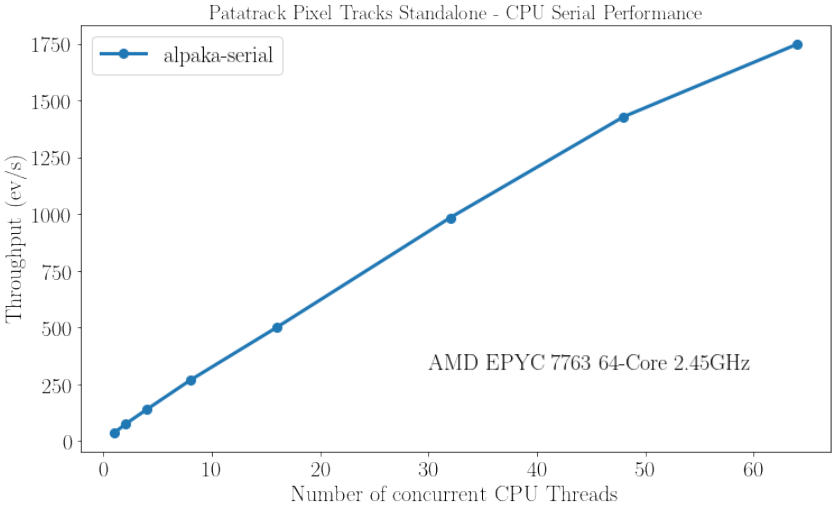

Figure 1. Results of the Alpaka version compiled for the CPU serial backend, varying the number of concurrent CPU threads

caching allocator is currently under development.
Figure 2 shows the results obtained and the differences between the various implementations The introduction of the stream-ordered asynchronous memory operations increases the overall performance, avoiding the need of synchronizing the entire device, restricting the lifetime of the memory operations to the specific stream, eliminating in this way the synchronization of the GPU work submitted in other streams. The caching allocator optimization for the native CUDA version shows higher performance, making use of memory pools that can be re-used within the same event, reducing the number of API calls needed to allocate the memory. Regarding the version without any optimizations, the Alpaka performance are very close to the native-implementation one. Comparing the results with the stream-ordered memory operations, Alpaka shows lower throughput values until 10 concurrent CPU threads. The Alpaka implementation of the stream-ordered memory operations has been developed during this study, and more efforts are ongoing to obtain performance closer to the native ones. Considering the advantages introduced by Alpaka, being able to compile the application for different backends, and considering that there is still room for improvement to enhance the run-time performance obtained using the Alpaka abstraction layer, the results obtained are very promising.

5. Conclusion and Future Perspectives

The CMS experiment is investigating solutions to adopt Heterogeneous Computing in its computing farm, to face the challenges that will be posed by the future upgrades of the LHC. To exploit multiple architectures and avoid code duplication, performance portability frameworks are a solution to obtain a single source code that can be compiled and executed on multiple architectures, making the framework more maintainable and testable in the long term. The Alpaka library is currently the best solution to achieve that. The demonstrator described in this work has shown how the different implementations can be built from a single code base and can run on multiple back-ends, with performance comparable to the native one. Not only the computing performance but also the simple usage of the Alpaka interface, make the library a good solution that fulfills the CMS computing requirements. Given the results obtained with this study, Alpaka has been selected as the official Portability-layer for the deployment at the CMS High Level Trigger for the Run 3 data acquisition.

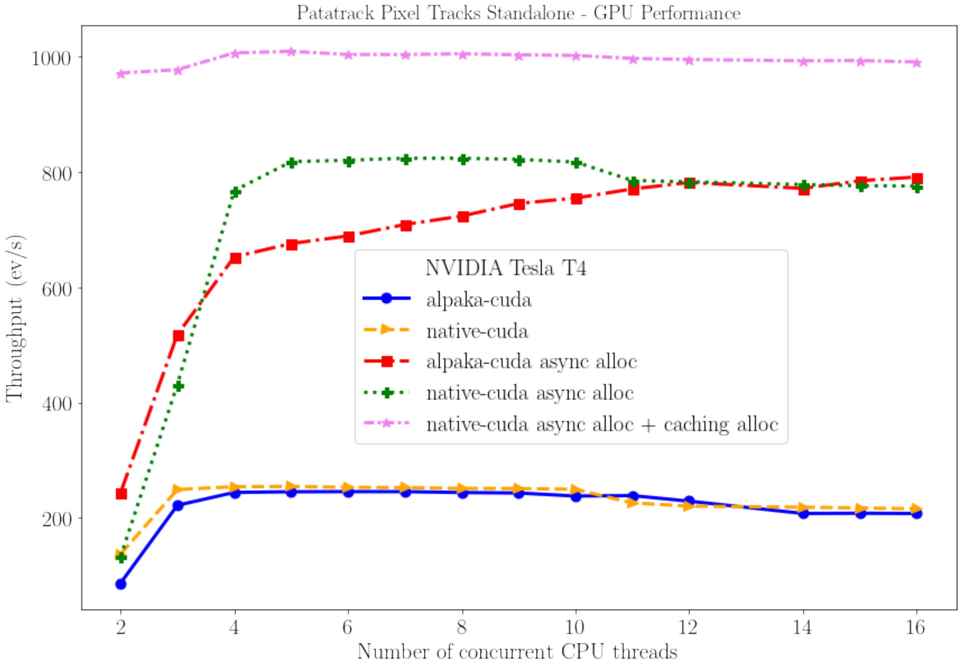

Figure 2. Comparison between the Alpaka version compiled for the GPU CUDA backend and the corresponding native version, running on an NVIDIA T4 GPU and varying the number of concurrent CPU threads. The blue and orange lines correspond respectively to the Alpaka and native-CUDA versions without any optimization. The green and red lines correspond respectively to the Alpaka and native-CUDA versions with the stream-ordered asynchronous memory operations. The pink line corresponds to the native-CUDA version with the stream-ordered memory operations and the custom caching allocator.

The Alpaka version of the pixel tracks and vertices reconstruction will be included in the CMS software, also providing the opportunity to port other algorithms with this library.
A rich R&D program is ongoing in CMS, including further investigations of other performance portability layers aiming Run 4 and beyond.

Acknowledgements

The work of M. Kortelainen 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-CCE) at Argonne National Laboratory, Fermi National Accelerator Laboratory, and Lawrence Berkeley National Laboratory under B&R KA2401045.
The work of W. Redjeb was supported by the Innovative Digital Technologies for Research on Universe and Matter (ErUM IDT) program of the German Federal Ministry of Education and Research.

References

[1] 2022 LHC Time Schedule `https://lhc-commissioning.web.cern.ch/schedule/LHC-long-term.htm` [Online; accessed 10-May-2022]

[2] 2006 CMS Physics: Technical Design Report Volume 1: Detector Performance and Software Tech. rep.

[3] 2017 *Journal of Instrumentation* **12** P10003–P10003

[4] 2007 CMS High Level Trigger Tech. rep. CERN Geneva revised version submitted on 2007-10-19 16:57:09 URL `https://cds.cern.ch/record/1043242`

[5] Zenker E, Worpitz B, Widera R, Huebl A, Juckeland G, Knüpfer A, Nagel W E and Bussmann M 2016 *CoRR* **abs/1602.08477** (*Preprint* 1602.08477) URL `http://arxiv.org/abs/1602.08477`

[6] Boost C++ Libraries URL `http://www.boost.org/`

[7] Intel Threading Building Blocks URL `https://software.intel.com/en-us/intel-tbb`

[8] OpenMP Application Program Interface URL `http://www.openmp.org`

[9] The OpenACC Application Programming Interface, version 3.1 (2020) URL `https://www.openacc.org/`

[10] *HIP Programming Guide v4.5*

[11] *CUDA C Programming manual*

[12] The Khoronos SYCL Working Group, SYCL 2020 Specification (revision 2) (2021)

[13] Collaboration C 2021 The Phase-2 Upgrade of the CMS Data Acquisition and High Level Trigger Tech. rep. CERN Geneva URL `https://cds.cern.ch/record/2759072`

[14] Bocci A, Kortelainen M, Innocente V, Pantaleo F and Rovere M 2020 *Front. Big Data* **3** 601728. 12 p (*Preprint* 2008.13461) URL `http://cds.cern.ch/record/2744911`

[15] 2022 Patatrack pixeltracks URL `https://doi.org/10.5281/zenodo.6552773`

[16] Blobel V 2006 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **566** 14–17 ISSN 0168-9002 tIME 2005 URL `https://www.sciencedirect.com/science/article/pii/S0168900206007996`

[17] **245** ISSN 2100-014X

[18] Jones C D, Paterno M F, Kowalkowski J, Sexton-Kennedy L and Tanenbaum W 2006 The new cms event data model and framework