Adoption of the alpaka performance portability library in the CMS software

Andrea Bocci[1], Antonio Di Pilato[1,2], Eric Cano[1], Felice Pantaleo[1], Gabrielle Hugo[1], Marco Rovere[1], Matti Kortelainen[3], Shahzad Malik Muzaffar[1], Vincenzo Innocente[1], Wahid Redjeb[1,4], on behalf of the CMS collaboration

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

E-mail: andrea.bocci@cern.ch

Abstract. To achieve better computational efficiency and exploit a wider range of computing resources, the CMS software framework (CMSSW) has been extended to offload part of the physics reconstruction to NVIDIA GPUs, while the support for AMD and Intel GPUs is under development. To avoid the need to write, validate and maintain a separate implementation of the reconstruction algorithms for each back-end, CMS decided to adopt a performance portability framework. After evaluating different alternatives, it was decided to adopt Alpaka as the solution for Run-3. Alpaka (Abstraction Library for Parallel Kernel Acceleration) is a header-only C++ library that provides performance portability across different back-ends, abstracting the underlying levels of parallelism. It supports serial and parallel execution on CPUs, and extremely parallel execution on GPUs. This contribution will show how Alpaka is used inside CMSSW to write a single code base; to use different tool-chains to build the code for each supported back-end, and link them into a single application; and to select the best back-end at runtime. It will highlight how the alpaka-based implementation achieves near-native performance, and will conclude discussing the plans to support additional back-ends.

1. Introduction

A portion of the CMS High Level Trigger (HLT) reconstruction has been rewritten to run on GPUs using the CUDA framework, enabling over 40% of the runtime to be offloaded to GPUs during the 2022 data taking, as shown in Figure 1. The deployment of a GPU-equipped HLT farm resulted in a 70% increase in event processing throughput, 50% better performance per kilowatt, and 20% improved performance per cost[1]. This GPU-based reconstruction has been validated offline on GPU-equipped nodes at CMS Tier-2s, deployed in production since the start of LHC Run-3 data taking in 2022, and optimized throughout the year. Current efforts focus on writing additional algorithms to leverage GPUs, including particle flow clustering, full primary vertex reconstruction, electron seeding, and other related tasks.

To avoid duplicating the effort of writing, maintaining and validating the reconstruction algorithms for each back-end, CMS explored various performance portability options[2, 3]. The exploration was based on the Standalone Patatrack Pixel Tracking application[4], that

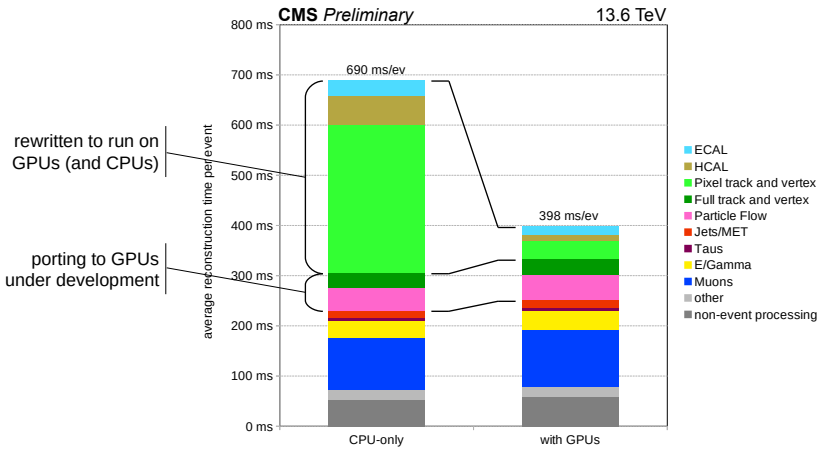

Figure 1. Average processing time per event for the CMS online reconstruction, measured on a production HLT node equipped with two AMD EPYC 7763 Milan CPUs and two NVIDIA Tesla T4 GPUs. The measurements were performed on proton-proton events collected in October 2022 with an average pileup of 56 collisions, using 8 concurrent jobs, each with 32 CPU threads, and processing 24 concurrent events. The horizontal bands represent the time fractions spent in different categories of reconstruction algorithms. The left stacked bar displays the average processing time when using only CPUs, while the right stacked bar shows the time when utilizing both CPUs and GPUs.

implements part of the CMS reconstruction algorithms in a stand-alone environment, easier to port to different back-ends. The Alpaka library was ultimately selected due to its high performance, very close to and sometimes surpassing that of native implementations, it support for GPUs from different vendors and generations within a single application, and its ease of integration into the CMS build system.

2. The Alpaka library

The Alpaka library[5, 6, 7, 8] is a C++17 header-only abstraction library designed to simplify accelerator development and enhance performance portability across various accelerators. It supports a wide range of devices, including CPUs and GPUs from NVIDIA and AMD, and offers multiple accelerator back-end options like CUDA, HIP/ROCm, OpenMP, `std::thread`, and serial execution. Support for Intel GPUs and FPGAs based on SYCL and oneAPI is under development[9]. Alpaka enables developers to write a single kernel implementation using function objects with a specific interface, eliminating the need for specialized CUDA, OpenMP, or custom threading code. The library allows mixing accelerator back-ends within a device queue, and runtime decision-making for kernel execution. The abstraction approach mirrors the CUDA grid-blocks-threads strategy, requiring algorithms to be organised into kernels executing on a multi-dimensional grid of uniform work items. Threads are organized in blocks, with parallel execution within each block, efficient synchronisation primitives and fast shared memory for interaction. Block execution order is unspecified and depends on the accelerator used, allowing optimal adaptation to the available hardware. Integration in an existing project is straightforward, using either CMake or Unix Makefiles.

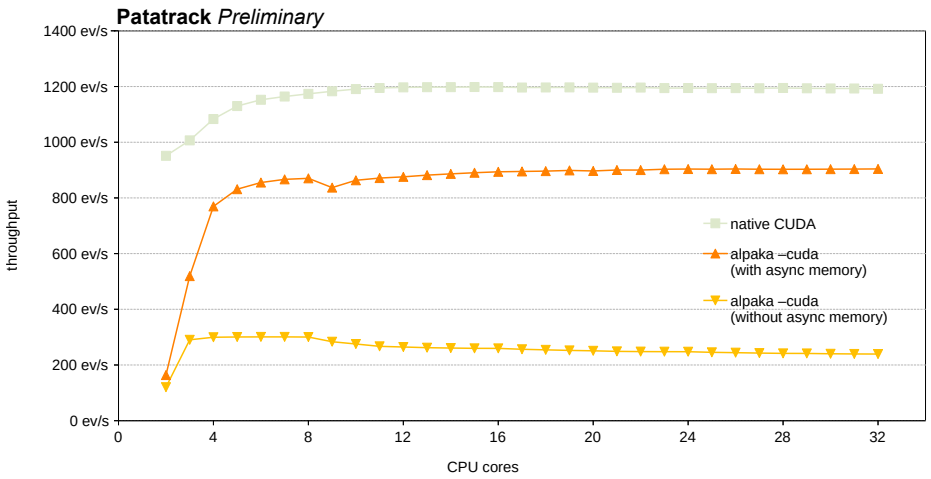

Figure 2. Average processing time per event for the Alpaka-based Patatrack pixel track reconstruction application running on an NVIDIA Tesla T4 GPU, plotted as a function of the number of CPU threads used and concurrent events being processed. The downward yellow triangles indicate the performance without using stream-ordered memory operations, while the upward orange triangles indicate the performance after implementing stream-ordered memory operations in Alpaka and using them in the application. For reference, the light green squares indicate the performance of the native CUDA application.

3. Performance

The reconstruction code in CMS exhibits patterns that differ significantly from those typically found in HPC code. The data analyzed consists of numerous independent events, with processing times for each event ranging from milliseconds to seconds. To exploit multi-core CPUs and GPUs, multiple events are processed simultaneously, necessitating the use of parallel "streams" or queues. Event processing involves numerous medium-sized memory allocations, copies, transfers, and the execution of dozens of distinct kernels. This results in an application that rapidly cycles through host and device memory and launches thousands of kernels per second.

The data processing throughput of such an application can be substantially improved by reducing the interaction with the accelerator runtime, employing a caching layer for the GPU-related resources: device global memory, pinned host global memory used for faster data transfers, CUDA streams and events used for synchronizing operations. As CMSSW has been successfully using this approach with native CUDA, it was essential to provide a similar functionality while using Alpaka. During the evaluation of the Alpaka library, CMS developers integrated the more general functionalities into the library itself, while re-implementing the CMS-specific features on top of Alpaka.

Figure 2 shows the impact of implementing stream-ordered memory operations in Alpaka. These operations are natively available for the CUDA (version 11.2 or newer) and HIP/ROCm (version 5.4 or newer) back-ends, and are emulated for the CPU back-end. Their introduction removed the main bottleneck for the Standalone Patatrack Pixel Tracking application, leading to a performance gain of 270% on an NVIDIA Tesla T4 GPU.

Figure 3 shows the impact of implementing a "caching allocator" on top of the Alpaka memory operations. Caching and reusing the host and GPU memory buffers in the user code instead of releasing and them to the back-end and reallocating them reduced the number of synchronisation points in the application, improving the overall performance by 25%.

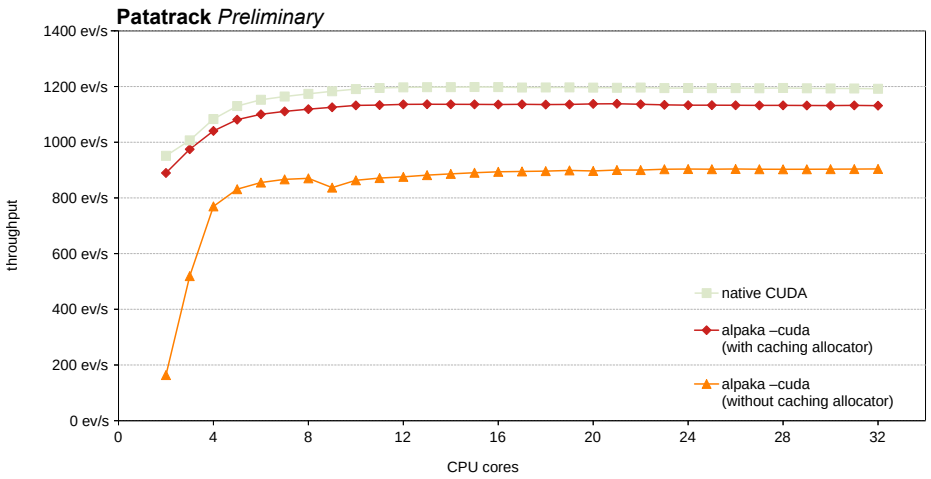

Figure 3. Average processing time per event for the Alpaka-based Patatrack pixel track reconstruction application running on an NVIDIA Tesla T4 GPU, plotted as a function of the number of CPU threads used and concurrent events being processed. The upward orange triangles indicate the performance without using a caching memory allocator, while the red diamonds indicate the performance after implementing a caching allocator for host and device memory on top of Alpaka and using them in the application. For reference, the light green squares indicate the performance of the native CUDA application.

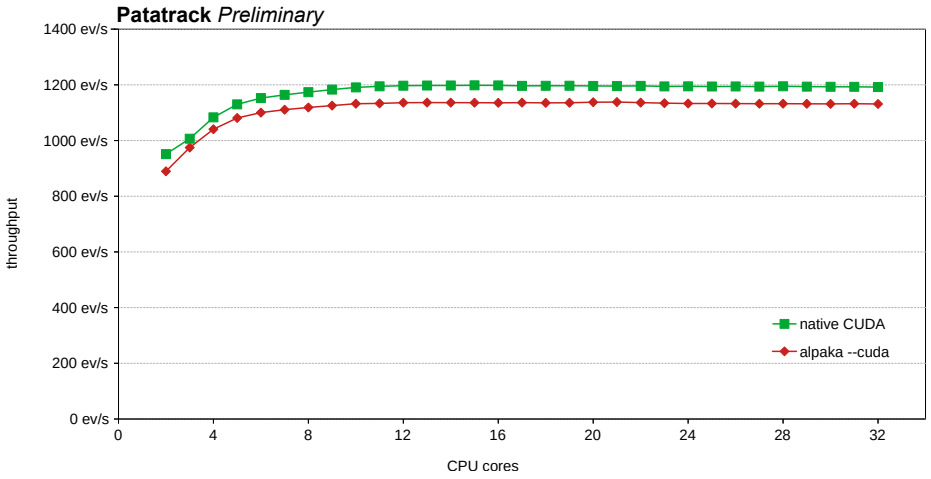

Figure 4. Average processing time per event for the Patatrack pixel track reconstruction application running on an NVIDIA Tesla T4 GPU, plotted as a function of the number of CPU threads used and concurrent events being processed. The red diamonds indicate the performance of the Alpaka-based application, while the green squares indicate the performance of the native CUDA application.

Finally, Figures 4 and 5 compare the performance of the native GPU application and its native CPU port with that of the Alpaka version. Running on an NVIDIA Tesla T4 GPU, the

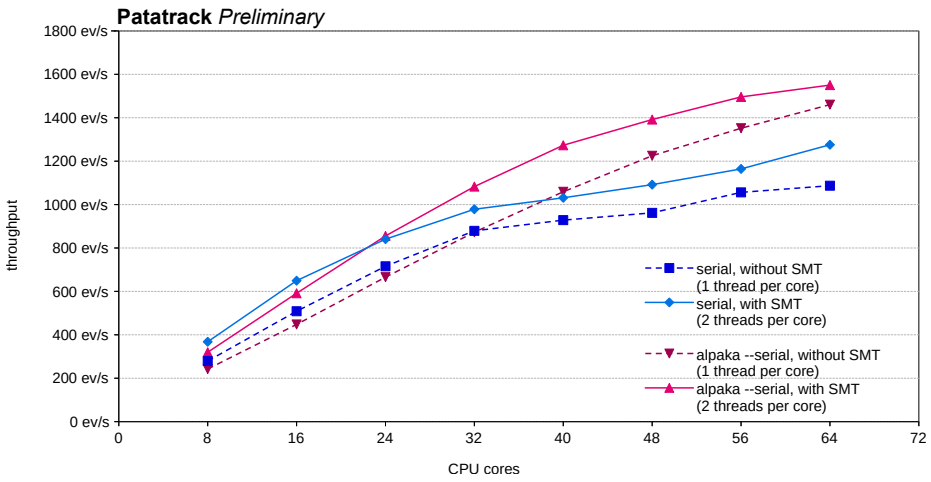

Figure 5. Average processing time per event for the Patatrack pixel track reconstruction application running on an AMD EPYC Milan 7763 CPU with 64 cores, shown as a function of the number of physical CPU cores used and concurrent events being processed. Dashed lines indicate the performance using a single thread per physical core, while solid lines demonstrate the impact of employing two threads per physical core with Simultaneous Multi-Threading (SMT) enabled. The purple downward triangles and pink upward triangles indicate the performance of the Alpaka-based application, whereas the blue squares and light blue diamonds represent the performance of the native CPU implementation.

Alpaka version achieves a performance better than 95% of the native one. On an AMD EPYC Milan 7763 CPU with 64 cores the performance of the Alpaka version is significantly better than that of the native implementation.

4. Adoption in CMSSW

The build system of CMSSW was extended to integrate Alpaka and automatically build packages for all the supported back-ends, using either the device specific compiler or simply linking to the device runtime, and producing a distinct shared library per back-end. A new `alpaka/` subdirectory was added under the existing ones such as `interface/`, `src/`, and `plugins/`. The source files under these subdirectories are compiled multiple times, once per back-end, with the active back-end identified by preprocessor macros. The files with the standard `.cc` extension are compiled by the host compiler (*e.g.* `g++`), while those with the newly introduced `.dev.cc` extension are compiled by the back-end specific device compiler (*e.g.* `nvcc -x cu` or `hipcc`).

Figure 6 shows the directory structure of two packages written using Alpaka. In the example, the files under `HeterogeneousCore/AlpakaTest/plugins/alpaka/` undergo multiple compilations: once for the CPU back-end, once for the CUDA back-end, and once for the HIP/ROCm back-end. The file `TestAlgo.dev.cc` is compiled for each back-end by a different compiler: `g++` for the CPU back-end, `nvcc -x cu` for the CUDA back-end, and `hipcc` for the HIP/ROCm back-end. In contrast, the other `.cc` files are always compiled using `g++`.

Finally, all the compiled files are linked with the back-end specific runtime libraries, and built into a separate shared library per back-end. This allows loading only the runtime libraries that are actually used by the application: for example, running on a machine without GPUs does not require loading any of the CUDA or ROCm libraries.

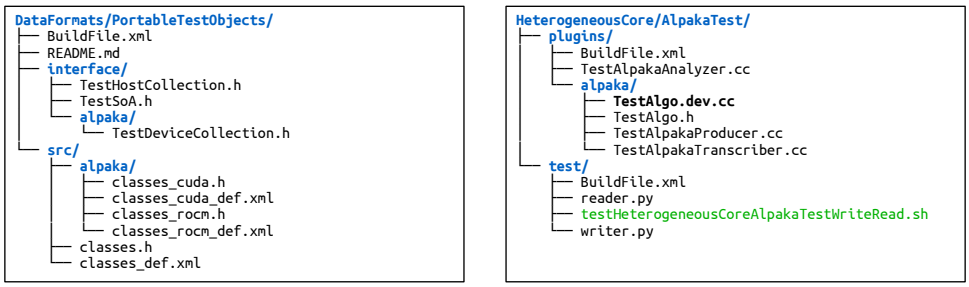

Figure 6. Directory structure for two CMSSW packages that use the Alpaka library. Source files under an `alpaka/` subdirectory are compiled multiple times, once per back-end, using either the standard host compiler or a back-end specific compiler. `.cc` files use the former, while `.dev.cc` files (in bold in the right box) use the latter.

5. Conclusions and future work

With the adoption of Alpaka as the *performance portability* solution for Run-3, CMS will be able to use a single code base to leverage leading accelerator technologies, such as NVIDIA CUDA and AMD ROCm, while automatically providing a CPU-only version. This approach paves the way for enhancements along two orthogonal directions. Within CMS, various groups are porting the existing CUDA algorithms to Alpaka, and developing new parallel algorithms directly using Alpaka. At the same time, CMS developers are closely collaborating with the Alpaka group to expand the back-ends that CMSSW can run on, and enhance the performance of the Alpaka library itself.

Acknowledgements

The authors would like to acknowledge the assistance of OpenAI's ChatGPT, which is based on the GPT-3.5 and GPT-4 models[10], for editing and enhancing sections of this manuscript. ChatGPT's AI language generation capabilities provided valuable input in refining the text and improving its clarity.

References

[1] Huwiler M 2023 *these proceedings*
[2] Kortelainen M J, Kwok M, (on behalf of the CMS Collaboration), Childers T, Strelchenko A and Wang Y 2021 *EPJ Web Conf.* **251** 03034 URL `https://doi.org/10.1051/epjconf/202125103034`
[3] Bocci A, Czirkos A, Di Pilato A, Pantaleo F, Hugo G, Kortelainen M, Redjeb W and on behalf of the CMS collaboration 2023 *Journal of Physics: Conference Series* **2438** 012058 URL `https://dx.doi.org/10.1088/1742-6596/2438/1/012058`
[4] Standalone patatrack pixel tracking URL `https://github.com/cms-patatrack/pixeltrack-standalone/`
[5] Alpaka: Abstraction library for parallel kernel acceleration URL `https://github.com/alpaka-group/alpaka`
[6] Worpitz B 2015 *Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures* Master thesis Technische Universität Dresden URL `http://dx.doi.org/10.5281/zenodo.49768`
[7] Zenker E, Worpitz B, Widera R, Huebl A, Juckeland G, Knüpfer A, Nagel W E and Bussmann M 2016 (IEEE Computer Society) (*Preprint* `1602.08477`) URL `http://arxiv.org/abs/1602.08477`
[8] Matthes A, Widera R, Zenker E, Worpitz B, Huebl A and Bussmann M 2017 (*Preprint* `1706.10086`) URL `https://arxiv.org/abs/1706.10086`
[9] Stephan J, Bastrakov S, Di Pilato A, Ehrig S, Gruber B M, Vyskočil J, Widera R and Bussmann M 2023 *these proceedings*
[10] OpenAI 2023 Gpt-4 technical report (*Preprint* `2303.08774`)