Evaluating Performance Portability with the CMS Heterogeneous Pixel Reconstruction code

Nikolaos Andriotis[1], *Andrea* Bocci[2], *Eric* Cano[2], *Laura* Cappelli[3], *Tony* Di Pilato[4,5], *Luca* Ferragina[6], *Gabrielle* Hugo[2], *Matti J.* Kortelainen[7,*], *Martin* Kwok[7], *Juan Jose* Olivera Loyola[8], *Felice* Pantaleo[2], *Aurora* Perego[9], *Wahid* Redjeb[2,10] (on behalf of the CMS Collaboration), *Mark* Dewing[11], and *Julien* Esseiva[12]

[1]Barcelona Supercomputing Center, Spain
[2]CERN, Geneva, Switzerland
[3]INFN Bologna, Italy
[4]Center for Advanced Systems Understanding (CASUS), Görlitz, Germany
[5]University of Geneva, Switzerland
[6]University of Bologna, Italy
[7]Fermi National Accelerator Laboratory, Batavia, IL, USA
[8]Institute of Technology and Higher Studies of Monterrey, Mexico
[9]University of Milano Bicocca, Italy
[10]RWTH Aachen University, Germany
[11]Argonne National Laboratory, Lemont, IL, USA
[12]Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Abstract. In the past years the landscape of tools for expressing parallel algorithms in a portable way across various compute accelerators has continued to evolve significantly. There are many technologies on the market that provide portability between CPU, GPUs from several vendors, and in some cases even FPGAs. These technologies include C++ libraries such as Alpaka and Kokkos, compiler directives such as OpenMP, the SYCL open specification that can be implemented as a library or in a compiler, and standard C++ where the compiler is solely responsible for the offloading. Given this developing landscape, users have to choose the technology that best fits their applications and constraints. For example, in the CMS experiment the experience so far in heterogeneous reconstruction algorithms suggests that the full application contains a large number of relatively short computational kernels and memory transfer operations. In this work we use a stand-alone version of the CMS heterogeneous pixel reconstruction code as a realistic use case of HEP reconstruction software that is capable of leveraging GPUs effectively. We summarize the experience of porting this code base from CUDA to Alpaka, Kokkos, SYCL, std::par, and OpenMP offloading. We compare the event processing throughput achieved by each version on NVIDIA and AMD GPUs as well as on a CPU, and compare those to what a native version of the code achieves on each platform.

[*]e-mail: matti@fnal.gov

1 Introduction

CMS [1, 2] started to utilize Graphics Processing Units (GPUs) in its High-Level Trigger (HLT) at the beginning of the LHC Run 3 [3]. The GPU integration into the CMS' data processing framework, CMSSW, was done directly with CUDA, and therefore only NVIDIA GPUs can be utilized. GPU vendors tend to provide their own APIs that also differ from programming the CPU. Developing and maintaining multiple versions of the physics algorithms would not be sustainable, particularly in large code bases that will be used for tens of years.

Over the past several years, many technologies aiming to provide full portability between CPUs and GPUs have emerged to ease the development and maintenance of heterogeneous applications. CMS had decided to adopt Alpaka [4–6] for the needs for LHC Run3 [7], and also to continue the exploration of other portability solytions for the long term. In this work we explore the applicability of Alpaka, Kokkos [8], SYCL [9], std::par [10], and OpenMP offloading [11] for portability across NVIDIA and AMD GPUs as well as the CPU, using the CMS heterogeneous pixel reconstruction [12] as a testbed for a realistic set of HEP reconstruction algorithms, focusing on the performance of each version.

2 CMS Heterogeneous Pixel Reconstruction

The CMS Heterogeneous Pixel Reconstruction was first developed with CUDA. The chain of about 40 GPU kernels takes the raw data of the CMS pixel detector as an input (about 250 kB/event), along with the beamspot parameters and necessary calibration data, and produces pixel tracks and vertices. The kernels are organized in 5 framework modules, depicted in Figure 1, that are scheduled by a simple oneTBB [13] based framework that mimics the relevant concurrency behaviors of CMSSW [14, 15].

The standalone setup [16] includes binary data files for the raw pixel detector data from 1000 simulated top quark pair production events from CMS Open Data [17], with an average of 50 superimposed pileup collisions with a center-of-mass energy of 13 TeV, using design conditions corresponding to the 2018 CMS detector. All of the data are read into the memory at the job startup to exclude I/O from the event processing throughput measurement.

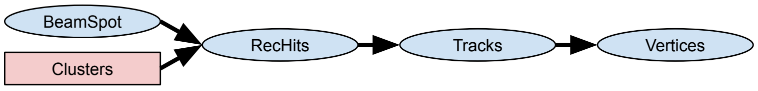


Figure 1: Directed acyclic graph of the framework modules in the heterogeneous pixel reconstruction. The arrows denote the data flow between the modules. The Clusters module (red rectangle) is the only one that transfers data from the device to the host, while the other modules (blue oval) do not.

3 Evaluated performance portability technologies

Alpaka is a thin, header-only templated C++ library that provides a similar abstraction level to CUDA. The supported backends in include CPU Serial, OpenMP 2, `std::thread`, oneTBB, CUDA, HIP, and SYCL as experimental. In our experience, Alpaka has been flexible to work with, for example one can build and run a single application that supports multiple GPU backends.

Kokkos is also a templated C++ library, whose abstraction level is higher than CUDA and aims to provide a descriptive programming model. The supported backends include CPU Serial, OpenMP, POSIX threads, CUDA, HIP, HPX, OpenMP target offloading, and SYCL as experimental. Kokkos' programming model provides parallel algorithms such as prefix scan, reduction, and sorting, as well as multidimensional array with customizable layout that was the precursor to `std::mdspan`. Kokkos' approach for its runtime library imposes constraints how the application code needs to be built.

SYCL is a specification by the Khronos group for a general C++ API to program heterogeneous devices. Notable implementations include Intel's oneAPI DPC++ [18] and open-source LLVM, and Open SYCL[1] [19]. In this work only the Intel's oneAPI and LLVM compilers were tested. SYCL provides similar concepts as CUDA, but also adds some higher-level algorithms.

std::par refers to the parallel algorithms in the C++ standard library. Notable implementations include NVIDIA HPC SDK for NVIDIA GPUs, Intel's openAPI oneDPL library for various devices, and e.g. GCC provides multithreaded implementation with oneTBB. The abstraction level is much higher than with Alpaka, Kokkos, or SYCL, and it could provide a low barrier for using GPUs in a new code base.

OpenMP is a compiler pragma based approach that has a long history that started as providing fork-join model for parallelizing loops, and has grown to include e.g. task-based concurrency model and offloading to compute accelerators.

4 Performance comparison and porting experience

4.1 Alpaka and Kokkos

The experiences of porting the CUDA version of the pixel reconstruction to Alpaka and Kokkos have been reported earlier in Refs. [20] and [21], respectively. The performance measurements reported here were done using the resources of the Joint Laboratory for System Evaluation at Argonne National Laboratory. All measurements were run for about 5 minutes, were repeated 4 times, and the average and standard deviation of the measurments are reported.

4.1.1 CPU

The CPU backends were run on two machines, one had 2 sockects of Intel Xeon Platinum 8176 CPU (Skylake microarchitecture) with 28 cores and 56 threads per socket; the other had 1 socket of AMD EPYC 7532 CPU (Milan microarchitecture) with 32 cores and threads.

For the CPU serial backends the throughput and peak memory of the full node was measured by running N processes of M threads such that $N \times M$ equals to the number of the hardware threads of the node. Here "serial" refers to each algorithm being run serially, and multiple copies of the algorithms are run concurrently, each copy processing data from separate events. In these tests the number of such algorithm copies, also referred to as concurrent events, was set the same as the number of threads per process.

The event processing throughput and the peak Resident Set Size (RSS) are shown in Figure 2. The direct and Alpaka serial versions give similar throughput, except the performance of the direct serial decreases dramatically when the per-process thread count goes beyond 28

[1]Formerly known as hipSYCL, and currently going through another name change

threads. The throughput of the Kokkos serial version does not scale at all, because Kokkos' serial backend has an explicit lock that prevents its efficient concurrent use. Both Kokkos and Alpaka versions use significantly less memory than the direct serial version.

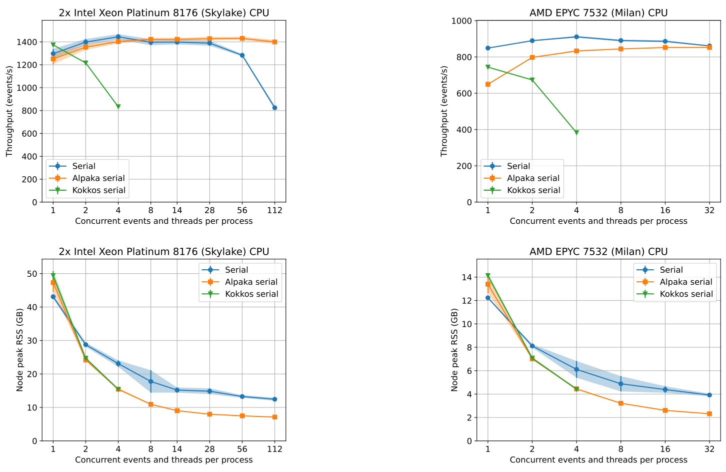


Figure 2: Comparison of the event processing throughput (top row) and peak RSS (bottom row)of direct CPU version, and the serial backends of the Alpaka and Kokkos versions of the heterogeneous pixel reconstruction as a function of the number of the concurrent events and threads per process on dual-socket Intel Xeon Platinum 8176 CPU (left) and AMD EPYC 7532 CPU (right). The number of processes was set such that all the hardware thread slots were utilized. In these tests only the events were processed in parallel, and the algorithm execution was serial. The reported throughput and peak RSS are the sum over all processes.

The behavior of parallelizing also the algorithms was tested with Alpaka's TBB backend and Kokkos' POSIX threads backend. In these tests the node had only one process, and the process had one event in flight. The event processing throughput is shown in Figure 3. In both cases parallelizing the algorithms only decreases the throughput, implying that for these algorithms on a CPU processing data through multiple copies of the algorithms concurrently is more useful than intra-algorithm parallelism.

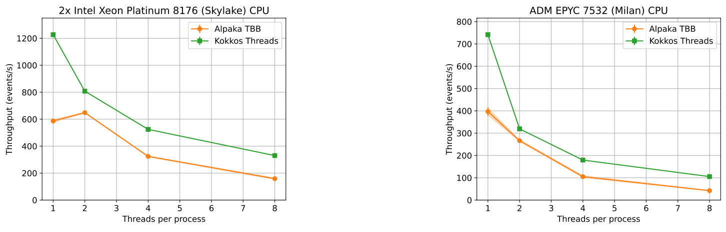
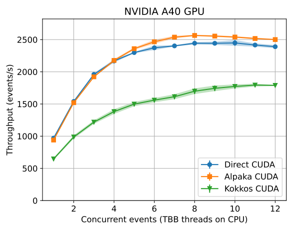


Figure 3: Comparison of the event processing throughput of Alpaka version with TBB backend, and Kokkos version with Threads backend, of the heterogeneous pixel reconstruction as a function of the number of threads on dual-socket Intel Xeon Platinum 8176 CPU (left) and AMD EPYC 7532 CPU (right). These tests had one event in flight, and each algorithm is parallelized.

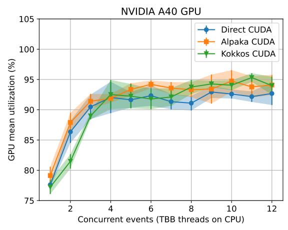
4.1.2 NVIDIA GPUs

The CUDA backends were tested on machines with NVIDIA A40 and A100 GPUs. The performance of a single process utilizing a single GPU was measured by increasing the number of concurrent events and CPU threads, and keeping the compute node otherwise empty. The event processing throughput, mean GPU utilization, peak GPU memory usage, CPU utilization, and peak host RSS are shown in Figure 4 for the A40 GPU. The mean GPU utilization was measured by recording the GPU utilization reported by `nvidia-smi` every 15 seconds and taking the average. The CPU utilization was measured by measuring the CPU and wall clock time of the event processing loop, and accounting for the expected level of host-side parallelism i.e. concurrent events. The A40 and A100 GPUs performed similarly, with A40 giving slightly higher throughput than A100, and therefore only the A40 results are shown for brevity.

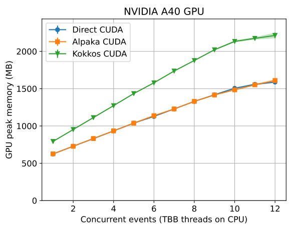
The direct CUDA version and Alpaka version's CUDA backend show similar event processing throughput, with Alpaka giving slightly higher throughput for more than 5 concurrent events than direct CUDA. Kokkos version's CUDA backend yields significantly lower throughput than direct CUDA or Alpaka versions. All the versions result in similar GPU utilization. Kokkos version uses significantly more GPU memory compared to the direct CUDA and Alpaka versions. The Kokkos version uses more CPU than direct CUDA and Alpaka versions for more than 5 concurrent events, and Alpaka version uses slightly more host memory than the direct CUDA and Kokkos versions.



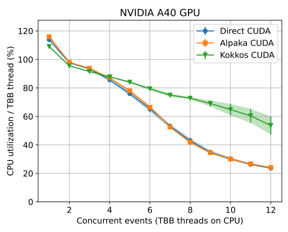
(a)



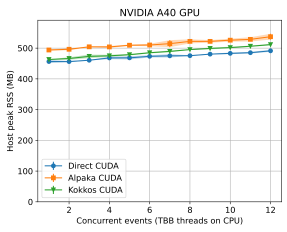
(b)



(c)



(d)



(e)

Figure 4: Comparison of the event processing throughput (a), mean GPU utilization (b), peak GPU memory usage (c), CPU utilization (d), and peak host RSS (e) of direct CUDA version, and the CUDA backends of the Alpaka and Kokkos versions of the heterogeneous pixel reconstruction as a function of the number of concurrent events and oneTBB worker threads on the CPU on NVIDIA A40 GPU. One process was run on an otherwise empty compute node.

4.1.3 *AMD GPUs*

The HIP backends were tested on machines with AMD MI250 and MI100 GPUs. As with CUDA backends, the performance of a single process utilizing a single GPU was measured by increasing the number of concurrent events and CPU threads, and keeping the compute node otherwise empty. The event processing throughput, peak host RSS, and host CPU utilization are shown in Figure 5 for the AMD MI100 GPU. The MI250 GPU shows quantitatively similar behavior, and is omitted for brevity.

The Alpaka version gives higher throughput than the direct HIP version for more than 3 concurrent events. The direct HIP version shows strange behavior for 3 concurrent events, which is reproducible but not understood. The throughput of the Kokkos version is much worse than the others. All versions use similar amount of host memory, and conversely to the NVIDIA GPUs, Kokkos uses less CPU resources compared to direct HIP and Alpaka versions.

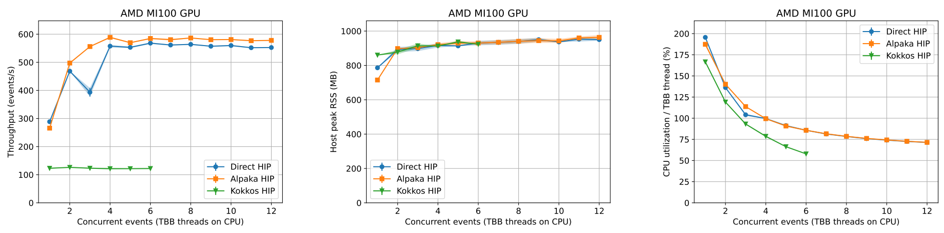


Figure 5: Comparison of the event processing throughput (left), peak host RSS (middle), and host CPU utilization (right) of the direct HIP version, and the HIP backends of the Alpaka and Kokkos versions of the heterogeneous pixel reconstruction as a function of the number of the concurrent events and CPU threads on AMD MI100 GPU. A single process was ran on the compute node, that was empty from other activity.

4.2 SYCL

The development of the SYCL version revealed many bugs in the Intel LLVM compiler, such as with collective operations on the CPU and with block shared variables. We were not able to replicate the setup that would result in a working executable on other machines. We also did not succeed in compiling the code for AMD GPUs. The performance measurements were done on a machine with AMD Ryzen 5900x CPU (Zen 3 microarchitecture) with 12 cores and 24 threads, and NVIDIA GTX 1080 GPU (Pascal microarchitecture). The event processing throughput on both CPU and GPU are shown in 6. On both hardware the direct serial or CUDA version gives significantly higher throughput than the SYCL version.

4.3 std::par

The std::par version of the pixel reconstruction is complete, but we were unable to test the full application because of compiler bugs leading to crashes. We found the high abstraction level to be a hurdle for converting a large and optimized CUDA application, that was easier to map to Alpaka, Kokkos, or SYCL than to std::par. The higher level of abstraction means that we have to part away with specific low-level features, asynchronous execution, add extra kernel launch, or significantly change the threading model that can not be expressed with the current C++ standard algorithms API.

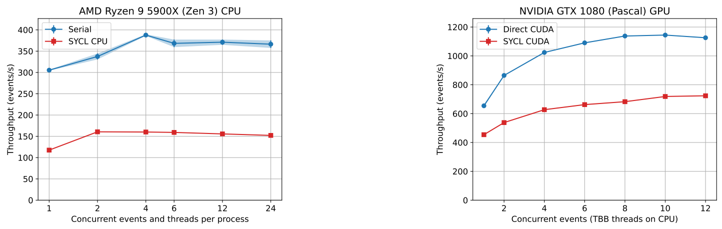


Figure 6: Comparison of the event processing throughput of the direct CPU (left) and CUDA (right) versions and the SYCL version of the heterogeneous pixel reconstruction as a function of the number of concurrent events and CPU threads on AMD Ryzen 5900x CPU (left) and on AMD GTX 1080 GPU (right). In the CPU tests (left) the the number of processes was set such that all the hardware threads were utilized, whereas in the GPU tests (right) one process was ran and the node was otherwise empty.

4.4 OpenMP

We ported some invidual kernels of the pixel reconstruction to OpenMP target offloading. We found that the OpenMP target offload can be used in conjunction with non-OpenMP-based multithreading, such as oneTBB. The porting work was done with LLVM compiler (versions 15, 16, and master branch) targeting NVIDIA and AMD GPU backends. Many problems in the compiler were encountered and reported, and some of them were fixed very quickly.

The individual OpenMP kernels were tested also with other compilers. With NVIDIA HPC SDK the kernels compile, but fail to run. With AMD compilers (AOMP, AFAR) the compiler crashes. The Intel oneAPI compiler (`icpx`) compiles the code, but was not pursued further.

Preliminary look on performance of some of the individual kernels with NVIDIA Nsight Systems shows that the OpenMP kernels are slower than corresponding CUDA kernels, and the OpenMP version shows much more data movement compared to direct CUDA version.

5 Conclusions

In this work we have compared the performance of direct, Alpaka, Kokkos, and SYCL versions of the CMS heterogeneous pixel reconstruction on x86 CPUs, and on NVIDIA and AMD GPUs. Overall Alpaka was found to yield comparable, or in some cases better, performance than the direct CPU, CUDA, and HIP versions. Alpaka was also found to be easiest to work with in this codebase, as it is flexible and adds only little constraints on top of the vendor APIs.

Challenges with Kokkos include the CPU Serial backend not supporting concurrent instances that prevents efficient concurrent processing of events, and overheads compared to direct implementations. We encountered lots of compilation problems with SYCL (Intel oneAPI and LLVM implementations), std::par (NVIDIA HPC SDK implementation), and OpenMP (various implementations). The main concerns with SYCL are overheads, with std::par the apparent necessity for many more kernels, and with OpenMP the added data movements. These hurdles suggest that the portability technologies are not yet mature enough to be used in production in large scale highly-concurrent applications comprised of vast number of relatively short computational kernels.

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-CCE) at Argonne National

Laboratory, Fermi National Accelerator Laboratory, and Lawrence Berkeley National Laboratory under B&R KA2401045. 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, JINST **3**, S08004 (2008)

[2] CMS Collaboration, *Development of the CMS detector for the CERN LHC Run 3* (2023), `2309.05466`

[3] G. Parida, these proceedings

[4] B. Worpitz, *Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures* (2015)

[5] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W.E. Nagel, M. Bussmann, *Alpaka - An Abstraction Library for Parallel Kernel Acceleration* (IEEE Computer Society, 2016), `1602.08477`

[6] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, M. Bussmann, *Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library* (2017), `1706.10086`

[7] A. Bocci, these proceedings

[8] C.R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez et al., IEEE Transactions on Parallel and Distributed Systems **33**, 805 (2022)

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

[10] ISO/IEC 14882:2020, *Programming languages - C++* (2020)

[11] OpenMP Architecture Review Board, *OpenMP Application Programming Interface, version 5.1* (2020)

[12] A. Bocci, V. Innocente, M. Kortelainen, F. Pantaleo, M. Rovere, Front. Big. Data **3**, 601728 (2020), `2008.13461`

[13] *oneAPI Threading Building Blocks*, https://github.com/oneapi-src/oneTBB (2023), accessed: 2023-08-18

[14] C.D. Jones, L. Contreras, P. Gartung, D. Hufnagel, L. Sexton-Kennedy, J. Phys.: Conf. Series **664**, 072026 (2015)

[15] A. Bocci, D. Dagenhart, V. Innocente, C. Jones, M. Kortelainen, F. Pantaleo, M. Rovere, EPJ Web Conf. **245**, 05009 (2020)

[16] *Standalone Patatrack pixel tracking*, https://github.com/cms-patatrack/pixeltrack-standalone/ (2021), accessed: 2023-08-18

[17] CMS Collaboration, *TTToHadronic_TuneCP5_13TeV-powheg-pythia8 in FEVT-DEBUGHLT format for 2018 collision data. CERN Open Data Portal.*, doi:10.7483/OPENDATA.CMS.GOB0.0LEW (2019)

[18] *Intel oneAPI DPC++/C++ compiler*, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compiler.html (2023), accessed: 2023-08-18

[19] A. Alpay, B. Soproni, H. Wünsche, V. Heuveline, *Exploring the possibility of a hipSYCL-based implementation of oneAPI*, in *IWOCL'22: International Workshop on OpenCL* (Association for Computing Machinery, New York, NY, USA, 2022)

[20] A. Bocci, A. Czirkos, A.D. Pilato, F. Pantaleo, G. Hugo, M. Kortelainen, W. Redjeb, J. Phys.: Conf. Ser. **2438**, 012058 (2023)

[21] M.J. Kortelainen, M. Kwok, T. Childers, A. Strelchenko, Y. Wang, EPJ Web Conf. **251**, 03034 (2021)