Porting CMS Heterogeneous Pixel Reconstruction to Kokkos

Taylor Childers[1], *Matti J.* Kortelainen[2,*], *Martin* Kwok[2], *Alexei* Strelchenko[2], and *Yunsong* Wang[3]

[1]Argonne National Laboratory, Lemont, IL, USA
[2]Fermi National Accelerator Laboratory, Batavia, IL, USA
[3]Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Abstract. Programming for a diverse set of compute accelerators in addition to the CPU is a challenge. Maintaining separate source code for each architecture would require lots of effort, and development of new algorithms would be daunting if it had to be repeated many times. Fortunately there are several portability technologies on the market such as Alpaka, Kokkos, and SYCL. These technologies aim to improve the developer productivity by making it possible to use the same source code for many different architectures. In this paper we use heterogeneous pixel reconstruction code from the CMS experiment at the CERNL LHC as a realistic use case of a GPU-targeting HEP reconstruction software, and report experience from prototyping a portable version of it using Kokkos. The development was done in a standalone program that attempts to model many of the complexities of a HEP data processing framework such as CMSSW. We also compare the achieved event processing throughput to the original CUDA code and a CPU version of it.

1 Introduction

Graphics processing units (GPUs) are being used in scientific computing because of their cost and power efficiency in solving data-parallel problems. Currently each GPU vendor provides their own APIs and programming models, that also differ from the programming of the CPU. There are, however, similarities in these GPU programming models, and in many cases the code for very core pieces of algorithms can be shared between the CPU and the GPUs, but the surrounding code arranging the data and calling the algorithms has to differ. In multi-million line code bases that have many custom algorithms and have to be maintained for tens of years, such duplication of code would require significant development and maintenance effort, and be error prone to maintain.

Over several years, many technologies for fully portable code between CPUs and compute accelerators have emerged to ease the development and maintenance burden of heterogeneous applications. These technologies include C++ libraries, such as Alpaka [1–3], Kokkos [4], and RAJA [5, 6]; SYCL [7] that can be implemented as libraries such as triSYCL [8] and hipSYCL [9] or as specific compilers such as ComputeCpp [10] by Codeplay and DPC++ [11] by Intel; compiler pragma based solutions such as OpenMP [12] and

*e-mail: matti@fnal.gov

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.

OpenACC [13]; or as standard C++ itself via parallel STL where the compiler is solely responsible for generating necessary code for the offloading.

In this work we explore the applicability of Kokkos for portability across CPU and GPUs using the Patatrack heterogeneous pixel reconstruction workflow [14] from the CMS experiment [15] at the CERN LHC [16] as a use case for a set of realistic HEP reconstruction algorithms that are able to effectively utilize a GPU. The work was done in the context of the DOE HEP Center for Computational Excellent (HEP-CCE). We look into not only the porting of the algorithms, but also the implications of integrating such an approach into a HEP data processing software.

We mimic the setup of the CMS data processing software, CMSSW [17]. CMSSW is multi-threaded [18–20] using the Intel Threading Building Blocks (TBB) [21], and the current plan for direct same-node compute accelerators is to build code for all supported accelerators in the same release build, express all possibilities in the configuration, and decide at runtime what code exactly to run based on hardware availability [22, 23]. We are looking for a single-source solution that would provide portability at least across CPU and GPUs, would be relatively easy to program with by HEP physicists, would provide adequate performance on all relevant platforms, and would require the least amount of change in the CMSSW building and data processing model. It is unlikely that all these goals would be met by a single technology, and therefore it is necessary to learn the details in all these aspects to find the best compromise.

This paper is organized as follows. The technical aspects of the Patatrack pixel reconstruction are described in Section 2. A brief introduction of Kokkos is given in Section 3. The experience of porting the original CUDA application into Kokkos is reported in Section 4. In Kokkos' nomenclature a place that runs code is called an *execution space*. We have tested Serial, Threads, CUDA, and HIP execution spaces of Kokkos, and we focus on several aspects in how Kokkos would fit into a framework like CMSSW. We have measured the event processing throughput of the Kokkos version's CPU and CUDA execution spaces, and compare those to direct CPU and CUDA implementations in Section 5. Conclusions are given in Section 6.

2 Patatrack Heterogeneous Pixel Reconstruction

The Patatrack pixel reconstruction pioneered offloading algorithms to NVIDIA GPUs with direct CUDA programming within CMSSW. The offloaded chain of reconstruction algorithms takes the raw data of the CMS pixel detector as an input, along with the beamspot parameters and necessary calibration data, and produces pixel tracks and vertices. CMSSW schedules algorithms as units that are called *modules*. The pixel reconstruction algorithms are organized in five modules, depicted in Figure 1, that communicate the intermediate data in the GPU memory through the CMSSW event data. The BeamSpot module only transfers the beamspot data into the GPU. The Clusters module transfers the raw data to the GPU, unpacks them, calibrates the individual pixels, and clusters the pixels on each detector module. The RecHits module estimates the 3D position of each cluster forming hits. The Tracks module forms n-tuplets from the hits, and fits the hit n-tuplets to obtain track parameters. The Vertices module forms vertices from the pixel tracks. There are further modules to optionally transfer the tracks and vertices to the CPU, and to convert the Structure-of-Array (SoA) data structures to the data formats used by downstream algorithms in CMSSW, but those are not considered in this work and therefore not shown in Figure 1.

In order to explore code portability technologies, the CUDA code of the Patatrack pixel reconstruction was extracted from CMSSW into a standalone program [24]. The separation from CMSSW gives us freedom to modify the compilers, build rules, external libraries, and

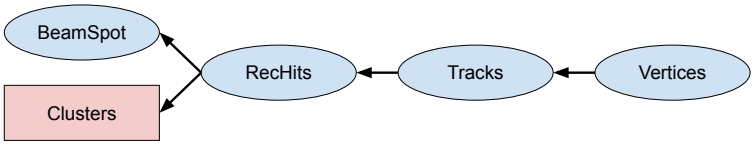

Figure 1. Directed acyclic graph of the framework modules in the Patatrack pixel reconstruction. The arrows denote the data dependencies of the modules, e.g. RecHits module depends on data produced by BeamSpot and Clusters modules. The Clusters module (red rectangle) is the only one that transfer data from the device to the host and uses External Worker synchronization mechanism, while the other modules (blue oval) do not.

code organization that would be more laborious to achieve in the full CMSSW software stack. The standalone program was crafted to mimic several aspects of CMSSW, including similar organization of code into shared libraries, plugin libraries that are loaded dynamically based on run-time information, and a simple framework that uses TBB for multi-threading. From the CMSSW framework concurrency features this simple framework includes only event loop based on TBB tasks, processing of multiple events concurrently, and processing of independent modules concurrently for the same event. There is only a single module type of each module having a separate instance for each concurrent event, and the *External Worker* concept [23] is included in order to use the CPU threads to do other work while the GPU is running the offloaded work. The CMSSW tools to use CUDA runtime directly in the modules [23] are also included.

The standalone setup includes a binary data file that contains raw pixel detector data from 1000 simulated top quark pair production events from CMS Open Data [25], 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, about 250 MB, are read into the memory at the job startup to exclude I/O from the throughput measurement. The necessary pixel detector conditions data are also stored in binary files, and read into the memory at the start of the job. The data processing throughput is calculated by measuring the time spent in the event processing, and dividing the number of processed events with that time. For each event, the object holding the raw data for that event is copied once from the aforementioned memory buffer to another object owned by the event data structure. The event processing time includes the time taken by this copy operation.

3 Kokkos

Kokkos is a programming model and a C++ library for writing performance portable applications. At the time of writing the latest version of Kokkos is 3.3.1, and it supports several execution spaces. An algorithm can be run serially on the host CPU via a *host serial* execution space, or it can be parallelized with one of two *host parallel* execution spaces that are OpenMP and (POSIX) Threads. An algorithm can also be offloaded to compute accelerators with *device parallel* execution spaces. NVIDIA GPUs can be used with CUDA or HPX execution spaces, and AMD GPUs can be used with HIP execution space. There are also OpenMP-Target and SYCL 2020 execution spaces that can support various platforms depending on the underlying toolchain. Currently all other device parallel execution spaces than CUDA are experimental. In this work we have tested Serial, Threads, CUDA, and HIP execution spaces.

```
// declarations of variables
constexpr uint32_t MaxNumModules;
constexpr uint32_t maxHitsInModule();
Kokkos::View<uint32_t const*, Kokkos::CudaSpace> cluStart;
Kokkos::View<uint32_t*, Kokkos::CudaSpace> moduleStart;

Kokkos::parallel_for(
  Kokkos::RangePolicy<Kokkos::Cuda>(0, MaxNumModules)),
  KOKKOS_LAMBDA(const int index) {
    moduleStart(index + 1) = std::min(maxHitsInModule(), cluStart(index));
});
```

Figure 2. A simplified example of using `RangePolicy` policy with `parallel_for`. The initialization of the declared variables is omitted for brevity. In this example the execution and memory space template argument are spelled out explicitly. If the compile-time defaults for those suffice, the explicit template arguments can be left out. Corresponding CUDA program is shown in Figure 3.

Kokkos makes use of a runtime library. The library can have the Serial, one host parallel, and one device parallel execution space enabled at the same time, and this set is chosen at the library build configuration time. In addition, at least for CUDA execution space, one library can support only GPUs that have the same major compute capability number. For example, one library can support Volta (compute capability 7.0) and Turing (7.5) GPUs, but not Volta and Pascal (6.0) GPUs. In the code the execution space to be used can be chosen at compile time with template arguments. If the execution space is not specified explicitly, the most advanced execution space available in the library is used, i.e. device parallel execution space is preferred over host parallel execution space, which is preferred over the Serial execution space. Currently Kokkos supports only one device per process.

Kokkos provides high-level interface for parallel operations. These include `parallel_for` for a for-loop of independent iterations, `parallel_scan` for a prefix scan, and `parallel_reduce` for a reduction. Parallel operations can be nested with some restrictions. The details of the iteration are controlled with a policy. A `RangePolicy` can be used for a 1-dimensional range where all elements of the range can be processed independently. An example of `parallel_for` with `RangePolicy` is shown in Figure 2 and a corresponding CUDA version in Figure 3. An `MDRangePolicy` extendes the concept of the 1-dimensional `RangePolicy` to many, up to 6, dimensions. A `TeamPolicy` introduces a *league* of *teams* that consist of *threads*[1]. Threads in a team can use a common scratch memory space, and can synchronize within the team with a barrier. In addition, Kokkos has some support for tasks and graphs, that are not explored in this work.

As well as parallel operations, Kokkos provides a datastructure for multi-dimensional array, `Kokkos::View`. It is reference counted and behaves like `std::shared_ptr`, and can be passed to device functions by value. A major feature of the `Kokkos::View` is that its memory layout can be controlled with template arguments, and the default layout depends on the memory space. In addition, intents for the memory can be expressed with additional template arguments, for example specifying random-access constant data enables seamless use of CUDA texture caches. Data transfers between the host and the device are done explicitly.

[1]The *league* corresponds to *grid* in CUDA, and *team* corresponds to *block*.

```
// declarations of used variables
constexpr uint32_t MaxNumModules;
constexpr uint32_t maxHitsInModule();

__global__
void fillHitsModuleStart(uint32_t const* cluStart, uint32_t* moduleStart) {
  for(int i = threadIdx.x, iend = MaxNumModules; i < iend; i += blockDim.x) {
    moduleStart[i + 1] = std::min(maxHitsInModule(), cluStart[i]);
  }
}

uint32_t const* cluStart_;
uint32_t* moduleStart_;
fillHitsModuleStart<<<1, 1024>>>(cluStart_, moduleStart_);
```

Figure 3. CUDA version of the simplified example expressed in Kokkos in Figure 2. The initialization of the declared variables is omitted for brevity.

4 Porting experience

4.1 Impact on building

The current plan to support compute accelerators in CMSSW software stack is to build code for all supported accelerators, and choose the exact version to be run at runtime [22]. The various constraints of the Kokkos runtime library, described in Section 3, make it challenging to deploy in this manner. A single runtime library supporting only one device parallel execution space, and only one CUDA major architecture or CPU vector architecture, would, in this plan, imply the need to build many versions of the runtime library. The correct version would have to be loaded dynamically based on the available hardware. In this work we used exactly one runtime library at a time.

Every source file that includes any Kokkos header must be built with a compiler that is capable of compiling the code for all the enabled execution spaces, even if the source file would not use any Kokkos functionality. For example, if the Kokkos runtime library was built with CUDA execution space enabled, all source files including Kokkos headers must be compiled with a CUDA capable compiler.

Kokkos provides an integration with the CMake build system. In this work, however, we used CMake only to build the Kokkos runtime library itself, and we used a plain Makefile to build the application code. We did this because CMSSW uses the SCRAM build system [26], and therefore we'd have to understand the exact build rules in order to implement those for SCRAM.

The inability of `nvcc` to link device code from shared objects imposed severe constraints on how the Kokkos runtime library had to be built. We were able to use the runtime library built as a dynamic library with `RangePolicy`, but with the first use of `TeamPolicy` that approach lead to link errors from `nvcc`. The only build setup we managed to get to work was to build the Kokkos runtime library as a static library without support for relocatable device code, but with position-independent code for the host (`-fPIC`) to be able to link the static library with dynamic libraries of the application. This setup implies that CUDA separate compilation model can not be used, and therefore each source file must contain all device code called from that file, either directly or via including other files. Also, CUDA dynamic parallelism can not be used.

With the HIP execuion space we were able to use a dynamic Kokkos runtime library, and in fact were not able to get a static build to work with the HIP compiler.

4.2 Impact on code

As mentioned in Section 3, the Kokkos execution space is chosen at compile time. A choice done at runtime would be a much better fit in the current plans of using compute accelerators in CMSSW. We implemented the capability of choosing the execution space at runtime by building each source file containing Kokkos code once for each execution space and using namespaces to guarantee different symbols for each execution space.

Conversion of CUDA kernel calls to Kokkos parallel operations was mostly straightforward. Kokkos provides a parallel scan and sort, and therefore we decided to use those instead of trying to port the implementations of scan and radix sort device functions in the direct CUDA version. The code uses team-wide scan, but before version 3.3, Kokkos provided only league-wide scan. Before updating to Kokkos 3.3 we used the league-wide scan with two additional kernels to post-process the league-wide result to be equivalent to a team-wide scan. Kokkos' parallel sort function can be called only from the host code, which meant that we had to split all the CUDA kernels that called the device-side sort function into two kernels, and call the Kokkos' host-side sort function in between. Finding out the proper and efficient way to transform the CUDA code to use the Kokkos' scan and sort APIs was the most time consuming single effort.

For hierarchical parallelism, or thread teams, we found that the number of threads in a team is not exactly portable. The Serial execution space requires it to be exactly one, Threads execution space can use at most the number of CPU threads, and CUDA execution space has the same limitations as CUDA itself. This disparity can be largely mitigated by specifying the number of threads as `Kokkos::AUTO()`, that leaves the decision of the number of threads to Kokkos.

We found `Kokkos::View` to be useful by providing a unified interface for memory allocation, and smart pointer semantics for managing the ownership of the memory block. Also the ability to avoid an additional memory allocation in code that transfers data from host to device for CPU-only execution spaces is a plus. The more advanced features like multiple dimensions and the layout control are not needed in this code, where nearly all arrays have only one dimension. The only exception is the track covariance matrix, but we did not try to transform the Eigen-based implementation in the original CUDA into multidimensional `Kokkos::View`. In this code a SoA abstraction would be much more useful than multidimensional array, and we do not see how `Kokkos::View` would help in crafting SoA data structures.

In the first Kokkos version we found that about 80 % overall kernel runtime was spent in `Kokkos::View` initialization. In this code the first operation for all device memory is a write either by a memory copy from the host memory, or by a computation done in a kernel. Therefore all the initialization done by default is unnecessary, and avoiding that with `Kokkos::ViewAllocateWithoutInitializing` argument to `Kokkos::View` constructor improved the event processing throughput by almost a factor of 3.

At the time of writing, we have not been able to successfully run the full application with the HIP execution space. A test application that uses the same build and dynamic library infrastructure works well, but is not complex-enough to give meaningful insights into the performance.

Furthermore, we have not yet managed to run the application with multiple concurrent events with Serial or CUDA execution spaces. The Threads execution space explicitly prevents calls from more than one thread, even if the calls would come at different times. Despite of the Threads execution space being uninteresting to be used in the context of CMSSW, we have included it as a comparison point in the performance measurements in Section 5 to show how a parallelization strategy different from concurrent events would perform.

5 Performance comparison

The performance tests were done on GPU nodes of the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). A Cori GPU node has two sockets with Intel Xeon Gold 6148 ("Skylake") processors, each with 20 cores and 2 threads per core, and eight NVIDIA V100 GPUs. For this work we used only one CPU socket, to avoid the impact of non-uniform memory access (NUMA), and one GPU. In all tests the threads were pinned to a single socket. Each job was run for approximately 5 minutes, processing the set of 1000 individual events for an integer number of times, and repeated 8 times on random nodes of the GPU cluster. The code was compiled with GCC 8.3.0, and `nvcc` from CUDA 11.1.

In order to minimize the impact of the CPU frequency scaling the CPU programs were tested by running another program on the background with as many threads as needed to fill all the 40 hardware threads of the socket. Table 1 shows the throughput of the Kokkos version with Serial and Threads execution spaces, and of the direct CPU version with 1 and 40 threads. The Kokkos version processes one event at a time, and with the Threads execution space each Kokkos parallel operation is parallelized with the same number of threads. The direct CPU version, on the other hand, is parallelized by processing multiple events concurrently, one event per thread. While comparing the multi-threaded throughput of these two approaches is not exactly fair, it does show what can be achieved with a single process using the different approaches.

The results in Table 1 show that the intra-event parallelization scales poorly, whereas parallelizing over events gives much better throughput and scales well. We have not concluded yet why the direct CPU version gives 1.5 times better throughput than the Kokkos version with Serial execution space.

The programs using CUDA were tested without any background activity on the CPU. Table 2 shows the throughput of the Kokkos version with CUDA execution space, and of the direct CUDA version. The direct CUDA version can process data from multiple events concurrent with CUDA streams, and this approach helps to get 2.5 times higher throughput from the V100 GPU than when processing one event at a time. With a single event in flight, the memory pool, based on the `CachingDeviceAllocator` of the CUB [27] library, helps to increase the throughput by 4.5 times compared to using raw CUDA memory allocations.

Table 1. Comparison of the event processing throughput between the Kokkos version of the program using Serial and Threads execution spaces and the CPU version implemented from the original CUDA version through a simple translation header. In all cases all the threads were pinned to a single CPU socket (Intel Xeon Gold 6148) that has 20 cores and 2 threads per core. Each test ran about 5 minutes, and CPU-heavy threads from a background process were used to fill all the 40 hardware threads of the socket. The work in the CPU version is parallelized by processing as many events concurrently as the number of threads the job uses without any intra-event parallelization, whereas in the Kokkos version there is only one event in flight, and all parallelization is within the data of that event. For the Kokkos version with Threads execution space the maximum throughput from a scan from 1 to 20 threads is reported. The reported uncertainty corresponds to sample standard deviation of 8 trials.

Test case	Throughput (events/s)
CPU version, 1 thread	13.5 ± 0.2
Kokkos version, Serial execution space	8.5 ± 0.2
CPU version, 40 threads	539 ± 9
Kokkos version, Threads execution space, peak (18 threads)	28 ± 1

Table 2. Comparison of the event processing throughput between the Kokkos version of the program using CUDA execution space and the original CUDA version. In all cases the CPU threads were pinned to a single CPU socket, and used one NVIDIA V100 GPU. Each test ran about 5 minutes, and the machine was free from other activity. The CUDA version can process data from multiple events concurrently using many CPU threads and CUDA streams, and uses a memory pool to amortize the cost of raw CUDA memory allocations. The maximum throughput from a scan from 1 to 20 concurrent events is reported for the CUDA version. In order to compare to the current state of the Kokkos version, the CUDA version was tested also with 1 concurrent event and disabling the use of the memory pool. The reported uncertainty corresponds to sample standard deviation of 8 trials.

Test case	Throughput (events/s)
CUDA version, peak (9 concurrent events and CPU threads)	1840 ± 20
CUDA version, 1 concurrent event	720 ± 20
CUDA version, 1 concurrent event, memory pool disabled	159 ± 1
Kokkos version, CUDA execution space	115.7 ± 0.3

The Kokkos version with the CUDA execution space reaches about 70 % of the throughput of the direct CUDA version when run on a single concurrent event and disabling the use of the memory pool. Profiling indicates that various overheads e.g. in the `Kokkos::View` are the main cause for the performance difference. From Table 2 it is also clear that the kind of data processing done in this application benefits greatly from a memory pool, and from processing multiple events concurrently.

6 Conclusions

We have ported the Patatrack heterogeneous pixel reconstruction code from CUDA to Kokkos. In our experience Kokkos provides an API that is at a higher level than CUDA, and would be easier to develop new algorithms by physicists that are not necessarily experts in programming. We have achieved almost full portability between CPU, CUDA, and HIP, even if work still continues to understand runtime failures of the HIP execution space version of the code. This analysis shows that Kokkos can give 70 % of native CUDA performance in a simplified setup without either a memory pool or concurrent events. If similar performance proportion can be achieved also in a more realistic setup, it may be worth using a portable framework to reduce person power in maintaining a code base despite the loss of compute performance.

Our impression is that Kokkos would work well for a project that compiles the code separately for each target architecture, does not rely much on shared libraries, uses CMake as the build system, and does not rely on concurrent work outside of Kokkos. CMSSW doing all these in the opposite way implies that integrating the current version of Kokkos into the current data processing model of CMSSW would be challenging to do without sacrificing application performance. It is not, however, clear to us at this time to what extent these challenges are caused by design choices in Kokkos, or by the nature of the portability problem itself.

More work is needed to complete the study with Kokkos. In addition, comparisons to other portability technologies are planned within the HEP-CCE.

Acknowledgements

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. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE- AC02-05CH11231.

References

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

[2] 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

[3] 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

[4] H.C. Edwards, C.R. Trott, D. Sunderland, Journal of Parallel and Distributed Computing **74**, 3202 (2014)

[5] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R.W. Scogland, *RAJA: Portable Performance for Large-Scale Scientific Applications* (2019), IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), p. 71

[6] *RAJA Performance Portability Layer*, https://github.com/LLNL/RAJA (2021), accessed: 2021-02-07

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

[8] *trisycl*, https://github.com/triSYCL/triSYCL (2021), accessed: 2021-02-07

[9] A. Alpay, V. Heuveline, *SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL*, in *Proceedings of the International Workshop on OpenCL* (Association for Computing Machinery, New York, NY, USA, 2020), IWOCL '20, https://doi.org/10.1145/3388333.3388658

[10] *ComputeCpp*, https://developer.codeplay.com/products/computecpp/ce/home (2021), accessed: 2021-02-07

[11] *Intel oneAPI DPC++/C++ compiler*, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compiler.html (2021), accessed: 2021-02-07

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

[13] OpenACC-Stadnrad.org, *The OpenACC Application Programming Interface, version 3.1* (2020)

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

[15] CMS Collaboration, JINST **3**, S08004 (2008)

[16] L. Evans, P. Bryant, JINST **3**, S08001 (2008)

[17] C.D. Jones, M. Paterno, J. Kowalkowski, L. Sexton-Kennedy, W. Tanenbaum, *The New CMS Event Data Model and Framework*, in *Proceedings of International Conference on Computing in High Energy and Nuclear Physics (CHEP06)* (2006)

[18] C.D. Jones, E. Sexton-Kennedy, J. Phys.: Conf. Series **513**, 022034 (2014)

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

[20] C.D. Jones, J. Phys.: Conf. Series **898**, 042008 (2017)

[21] *oneAPI Threading Building Blocks*, `https://github.com/oneapi-src/oneTBB` (2021), accessed: 2021-02-07

[22] CMS Offline Software and Computing, *Evolution of the CMS computing model towards phase-2* (2021), CMS-NOTE-2021-001, `https://cds.cern.ch/record/2751565`

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

[24] *Standalone Patatrack pixel tracking*, `https://github.com/cms-patatrack/pixeltrack-standalone/` (2021), accessed: 2021-02-07

[25] 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)

[26] J.P. Wellisch, C. Williams, S. Ashby, *SCRAM: Software configuration and management for the LHC Computing Grid project*, in *Proceedings of International Conference on Computing in High Energy and Nuclear Physics (CHEP03)* (2003), p. TUJP001, `cs/0306014`

[27] *CUB*, `https://nvlabs.github.io/cub/` (2021), accessed: 2021-02-07