Abstraction of the CMS track reconstruction software for heterogeneous computing: application to the CMS silicon strip clustering algorithm

Ričards Remess

Riga Technical University

August 24, 2023

Supervisors:

Mario Masciovecchio (University of California, San Diego)

Emmanouil Vourliotis (University of California, San Diego)

Abstract

The reconstruction of charged particles in the CMS tracking detector, during both the online and the offline collision event reconstruction, is crucial for the success of the CMS physics program. However, at increasing instantaneous luminosity, the required computing resources for an efficient and timely track reconstruction grow almost exponentially. For this reason, the CMS track reconstruction software needs to maximally exploit parallel computing on multi- and many-core CPUs as well as GPUs (heterogeneous computing). In this context, the ability to abstract the CMS track reconstruction software for heterogeneous computing allows not to depend on specific choices of computing architectures, at the present time as well as in the future. This project focused on the algorithm used to cluster charge deposits in the CMS silicon strip tracker, for both the online and the offline event reconstruction, with the goal to enable its execution on any CPU or GPU architecture, in view of the future usage of the Alpaka library.

Contents

1 Introduction

1.1 Reconstruction of charged particle tracks

Charged particle track reconstruction is a very important process for the Compact Muon Solenoid (CMS) experiment at the CERN Large Hadron Collider (LHC), and other high energy physics experiments. The primary goal of track reconstruction is to accurately determine the trajectories of charged particles based on signals ('hits') recorded by detectors. This process is crucial for identifying particles, understanding their properties, and studying the interactions that occur during collisions. Silicon strip clusterization is the process of identifying and grouping together charge deposits in adjacent silicon sensors (strips) that are traversed by charged particles, thus forming a 'cluster'.

1.2 The CMS software (CMSSW)

The CMS software (CMSSW) [1] is a software package consisting of different components used for the event reconstruction within the CMS experiment.

The CMSSW workflows used for the event reconstruction are controlled by `Python` configuration files. These files determine the order of the algorithms to be run, and their execution parameters. Modules (`C++` files) that are used to create and add new objects to the event, based on the existing information, are called 'producers'.

1.3 Track reconstruction using Graphic Processing Units

Compared to Central Processing Units (CPUs), Graphics Processing Units (GPUs) offer much better performance for tasks that are highly parallelizable, such as matrix operations. Until recent times, the software for track (or, more generally, event) reconstruction at CMS has been developed targeting CPU architectures. However, in certain cases the same software can run on modern GPUs with large performance benefits, arising not only from the higher performance of GPUs and the higher computational efficiency, but also from the ability to utilize both CPUs and GPUs simultaneously, in heterogeneous computing systems.

In this context, CMSSW modules are run on a CPU, which represents the 'Host' in the heterogeneous computing system. Within a module, it is then possible to invoke 'kernels', i.e., a set of operations that can run in parallel on independent computing threads. These are instead executed on the 'Device' of the heterogeneous computing system (i.e., a GPU in this case).

The silicon strip clustering is among the algorithms that have been recently ported to CUDA (Compute Unified Device Architecture) [2] in CMSSW, thus enabling their offloading to GPUs.

1.4 Abstraction of CMS track reconstruction software with Alpaka

CUDA is a programming interface designed for utilization of certain types of NVIDIA GPUs. It is proprietary and closed source, and designed to work for programming languages such as C, C++ and Fortran.

Instead, Alpaka [3] is a versatile `C++`, header-only library that enables the development of algorithms on accelerated computing architectures with ease and efficiency. It is designed to facilitate high-performance parallel computing on various platforms, including CPUs, GPUs and other accelerators. Alpaka streamlines the process of writing code that can fully exploit the parallel processing capabilities of modern hardware. This is achieved by abstracting the complexities of memory management, synchronization, and data movement, allowing programmers to focus on algorithm design and problem-solving.

One of Alpaka's key strengths is its adaptability. A codebase written in Alpaka supports a wide range of backends, allowing a single version of the code to transition between different computing architectures, without any modifications. This adaptability not only future-proofs code but also optimizes performance across diverse hardware platforms. For this reason, Alpaka is particularly suitable for large-scale, long-term experiments such as CMS.

1.5 Structure of Arrays (SoA)

In computer programming, especially when dealing with structured data, two common data layout approaches are 'Structure of Arrays' (SoA) and 'Array of Structures' (AoS). These approaches determine how data elements are stored in memory and are accessed.

In an AoS, data are organized as a collection of structures (or objects), where each structure represents a single entity and contains multiple fields, or attributes. All the attributes for a given entity are stored together in memory, and the structures for different entities are placed sequentially. An AoS layout, in C++, may look like:

```
struct Particle {
    float mass;
    float charge;
    float momentum;
    // Other attributes...
};
Particle particles[N];
```

In an SoA, data are instead organized so that each attribute of all entities is stored together in separate arrays. Each array holds the values of a specific attribute for all entities. For example, the previous AoS structure converted to an SoA structure would become:

```
float masses[N]
float charges[N]
float momenta[N]
//Other arrays for different attributes...
```

The main reason to switch from AoS to SoA in the context of GPUs is that SoA ensures minimal memory access operations by placing homologous variables from each structure instance into cache aligned columns in memory. This allows data to be coalesced as shown in Fig. 1, which minimizes the required memory access overhead.

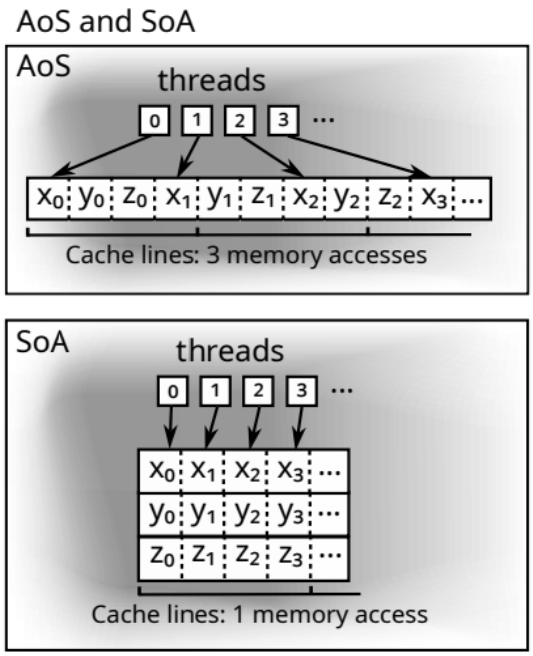

Figure 1: AoS and SoA memory layout structure, and potential access pattern. [4]

1.6 SoA implementation in CMS software

A package [4] is available in CMSSW that simplifies the creation of SoA arrays, their views (i.e., objects with pointers to data), and buffer management. Fig. 2 shows how an SoA array is initialized using the `GENERATE_SOA_LAYOUT` macro. In this example, the macro creates an array with multiple columns of type double named x, y, z. It also creates a scalar value named r, and an eigen-column matrix, denoted as m. Using the SoA layout provides a common access structure for differing data formats on both CPU and GPU, and it facilitates and unifies the interface between them.

```
namespace portabletest {
  // the typedef is needed because commas confuse macros
  using Matrix = Eigen::Matrix<double, 3, 6>;

  // SoA layout with x, y, z, id, m fields
  GENERATE_SOA_LAYOUT(TestSoALayout,
                      // columns: one value per element
                      SOA_COLUMN(double, x),
                      SOA_COLUMN(double, y),
                      SOA_COLUMN(double, z),
                      SOA_COLUMN(int32_t, id),
                      // scalars: one value for the whole structure
                      SOA_SCALAR(double, r),
                      // Eigen columns
                      SOA_EIGEN_COLUMN(Matrix, m))
  using TestSoA = TestSoALayout<>;
}  // namespace portabletest
```

Figure 2: Example of generation of an SoA layout [4].

2 Abstraction of the CMS strip clustering algorithm

The goal of this project is to migrate the SoA data formats involved in the CMS strip clustering algorithm to a layout like the one described above and depicted in Fig. 2. This is the first, crucial step for the migration of the full strip clustering algorithm from CUDA to Alpaka, thus making it architecture agnostic.

The data flow between the CPU and GPU is sketched in Fig. 3. Each block represents a data format, while each arrow denotes the module used to transition between them. In the modules involved in data format from CPU to GPU and vice versa (producers [1] and [2]), corresponding kernels are also run (kernels [1] and [2], respectively).

The migration of the SoA data formats entailed that the CMSSW `CUDADataFormats` and `RecoLocalTracker` packages had to be updated to use such an SoA layout:

- A header file, `SiStripUtilities.h` (similar to Figure 3) which defines an SoA layout generated with the `GENERATE_SOA_LAYOUT` macro was introduced. This layout serves as a template for the Device and Host data formats.
- The header file containing the definitions of CUDA layouts, `SiStripClustersCUDA.h`, was replaced by two separate ones, where the Device and Host instances of the SoA layout are defined: `SiStripClustersSoADevice.h` and `SiStripClustersSoAHost.h`.
- The newly added classes were included in the relevant dictionaries, `Buildfile.xml`, `classes.xml`, and `classes.h`, in order to instruct the compiler.

The `SiStripClustersSoADevice` class is the class that stores the SoA layout on the Device. However, only the portable device collection in this class is stored on the device, while every other variable is stored on the Host.

Full details about this software development activity can be found in Ref. [5].

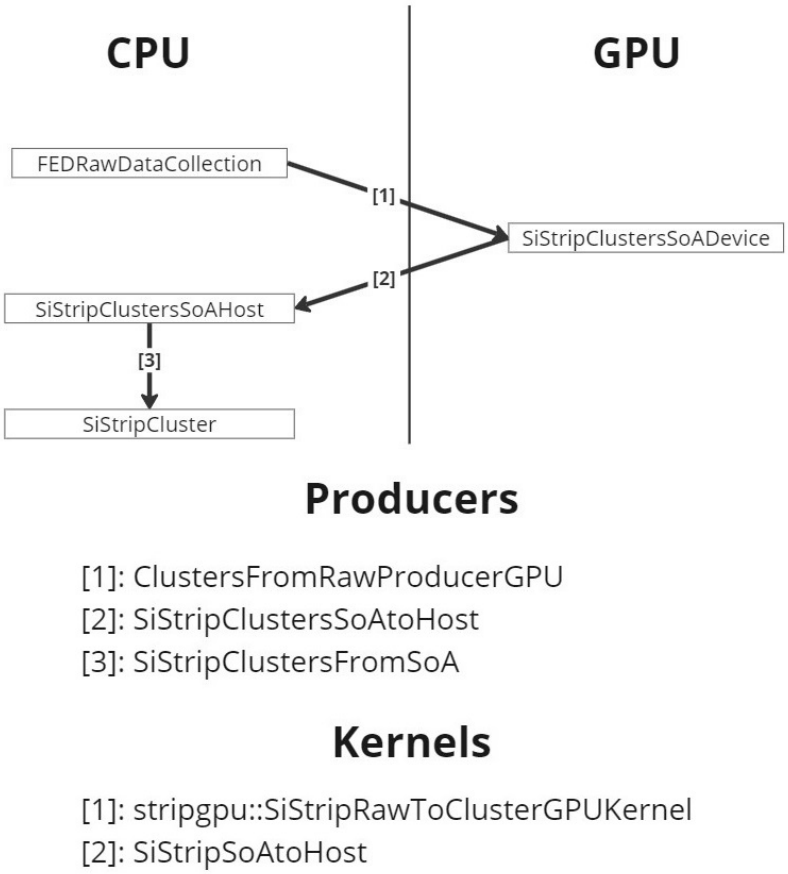

Figure 3: Diagram representation of data objects on CPU and GPU, and modules that control the data flow from one to another.

2.1 Evaluation of the algorithm performance: the setup

In order to validate the update, the performance obtained with the original version of the algorithm was compared to that obtained after the update itself. No difference in performance is expected.

The track reconstruction task was run in simulated events, on a computing node where a GPU was available, by either enabling the offloading of the strip clustering algorithm to GPU, or not. The output of the reconstruction was then analyzed, comparing the results obtained with the original version of the software to those obtained after this work.

The full list of commands used to produce such a comparison is provided in Appendix A.

3 Results

The track reconstruction physics performance is evaluated in terms of track reconstruction *efficiency* and *fake rate*. It is measured in a simulated $t\bar{t}$ sample with superimposed pileup events. The number of pileup events is sampled from minimum bias events with Poisson mean uniformly distributed from 55 to 75. The CMS detector conditions are simulated with no failure and account for the residual radiation damage due to the LHC Run 2 operations.

The reconstructed tracks are selected with the 'high purity' quality flag [6]. A reconstructed track is considered associated to a simulated particle if more than 75% of its hits have been originated from such simulated particle. If this is not the case, the reconstructed track is considered as the result of a random combination of hits and marked as a misidenti-

fied ('fake') track. Only simulated tracks from the 'signal vertex' (hard scattering) are used in the efficiency computation. The tracking efficiency is defined as the fraction of simulated tracks associated to at least one reconstructed track. All simulated tracks from any vertex (including pileup vertices) are used in the fake rate computation. The tracking fake rate is defined as the fraction of misidentified reconstructed tracks.

Fig. 4 (Fig. 5) shows the tracking efficiency and fake rate as obtained when the strip clustering algorithm is run on CPU (GPU), before and after the development described in this note, as a function of the track transverse momentum (p_{T}). For efficiency (fake rate), the p_{T} of the matched simulated particle (reconstructed track) is used. As expected, the performance after the development described in this note is identical to that before.

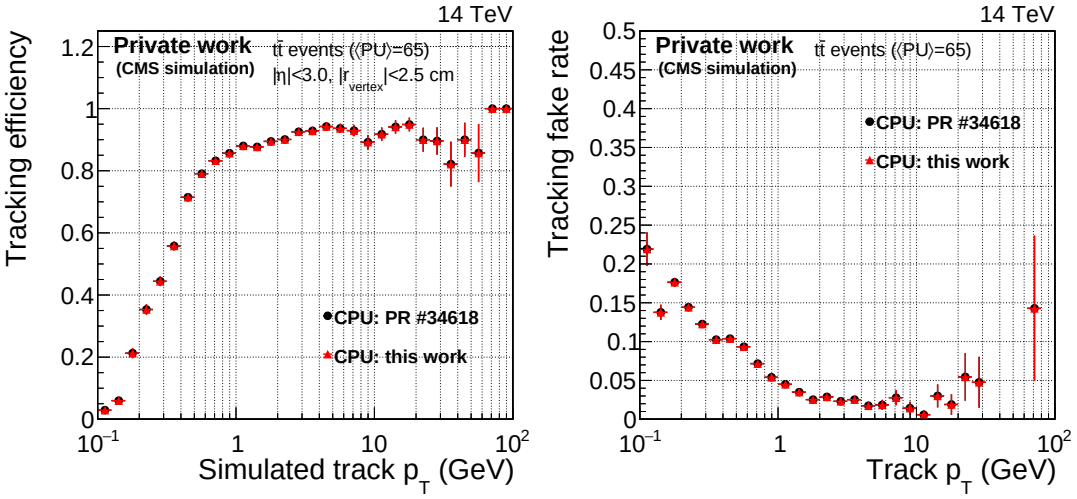

Figure 4: The tracking (left) efficiency and (right) fake rate are shown as a function of the simulated and reconstructed track transverse momentum, p_{T}, respectively, when the strip clustering algorithm is run on CPU. The performances obtained with the baseline version of the CMS software and after this work are compared, showing perfect agreement.

Comparisons of the efficiency and fake rate obtained on CPU and GPU are shown in Fig. 6, as a function of the track p_{T}. As expected, the CPU and GPU implementations show nearly identical performances, with negligible differences due to the non-deterministic nature of the GPU parallel execution. On GPU, because parallel local tasks are executed in a non-deterministic order, the truncation of the strip clusters can lead to the selection of different strips in each cluster with respect to the CPU implementation. The truncation of large strip clusters is a feature of the algorithm, and it has been shown to retain tracking efficiency with a reduced fake rate.

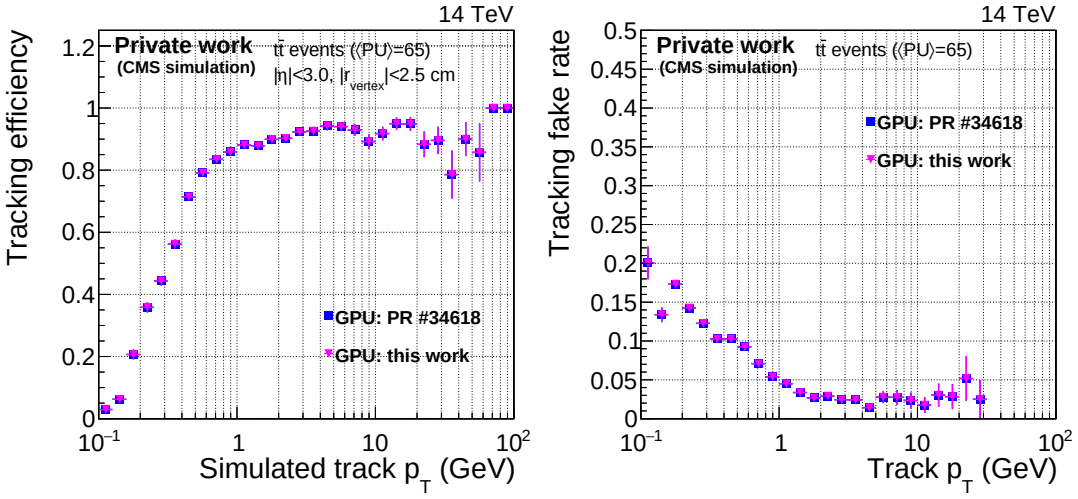

Figure 5: The tracking (left) efficiency and (right) fake rate are shown as a function of the simulated and reconstructed track transverse momentum, p_{T}, respectively, when the strip clustering algorithm is run on GPU. The performances obtained with the baseline version of the CMS software and after this work are compared, showing perfect agreement.

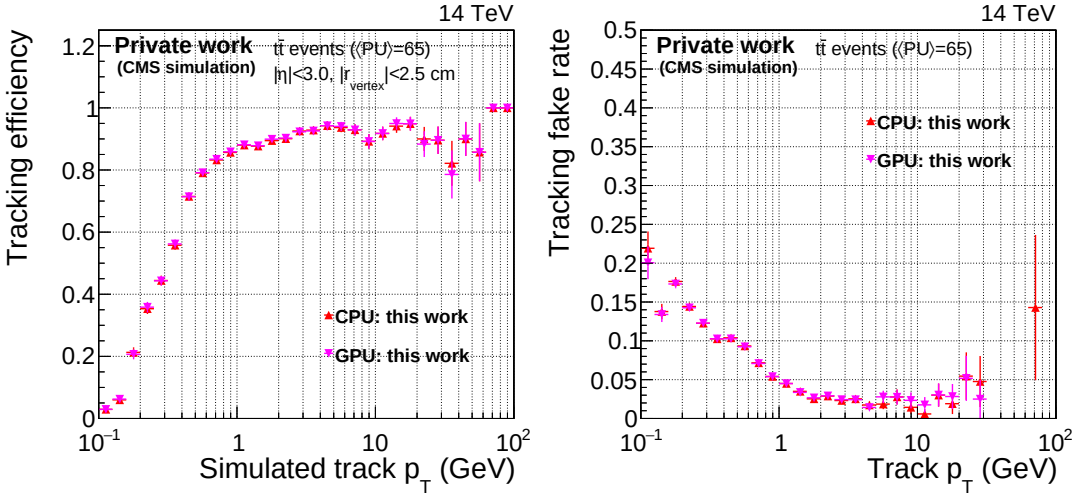

Figure 6: The tracking (left) efficiency and (right) fake rate are shown as a function of the simulated and reconstructed track transverse momentum, p_{T}, respectively. The performances obtained after this work when the strip clustering algorithm on CPU and on GPU are compared, showing nearly perfect agreement.

4 Summary and outlook

A report of a CERN Summer Student project was presented, aimed at the abstraction of the CMS track reconstruction software for heterogeneous computing, with a focus on the algorithm used to cluster charge deposits in the CMS silicon strip tracker. The ultimate goal of this work and its anticipated further development is to enable the execution of the strip clustering algorithm on any CPU or GPU architecture, exploiting the prospective usage of the Alpaka library in the CMS software (CMSSW).

In this work, the generalization of the `SiStripCluster` data format in CMSSW was successfully accomplished and validated. Further developments are anticipated that will build on these results: other data formats will be updated to use 'Structure of Arrays' (SoA) layouts, together with the GPU kernels that consume or produce those formats. After this update will be accomplished, the full migration of the strip clustering algorithm to Alpaka will be achieved.

References

[1] CMS Collaboration, *CMS software GitHub repository* (2023).

[2] NVIDIA, *NVIDIA CUDA home page* (2023).

[3] Alpaka Group, *Alpaka GitHub repository* (2023).

[4] Cano, E. and Bocci, A. (on behalf of the CMS Collaboration), *Implementation of Generic SoA Data Structures in the CMS Software.*, poster at 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research (2022).

[5] Remess, R., Masciovecchio, M., and Vourliotis, E., *GitHub: commit with development* (2023).

[6] CMS Collaboration, *Description and performance of track and primary-vertex reconstruction with the CMS tracker*, JINST 9 (2014) 10 (2014).

A List of CMSSW commands

The full workflow from code to performance comparisons entails multiple steps:

- code development;
- compilation/building (`scram build`, within CMSSW environment);
- run track reconstruction, and possibly tracking validation;
- harvesting of validation DQM output.

Once the harvested DQM files for both reference and target scenarios are produced, it is possible to proceed to performance comparisons.

For track reconstruction and validation on the CPU the following command was used:

```
cmsDriver.py step3 -s RAW2DIGI,RECO:
    reconstruction_trackingOnly,VALIDATION:
    @trackingOnlyValidation,DQM:@trackingOnlyDQM --
    conditions auto:phase1_2022_realistic --datatier DQMIO -
    n 100 --eventcontent DQM --geometry DB:Extended --era
    Run3 --filein <input filename, GEN-SIM-DIGI-RAW format>
    --fileout <output filename, DQM format> --mc --customise
     RecoLocalTracker/SiStripClusterizer/
    customizeStripClustersFromRaw.
    customizeStripClustersFromRaw
```

For track reconstruction and validation on the GPU the following command was used:

```
cmsDriver.py step3 -s RAW2DIGI,RECO:
    reconstruction_trackingOnly,VALIDATION:
    @trackingOnlyValidation,DQM:@trackingOnlyDQM --
    conditions auto:phase1_2022_realistic --datatier DQMIO -
    n 100 --eventcontent DQM --geometry DB:Extended --era
    Run3 --filein <input filename, GEN-SIM-DIGI-RAW format>
    --fileout <output filename, DQM format> --mc --customise
     RecoLocalTracker/SiStripClusterizer/
    customizeStripClustersFromRaw.
    customizeStripClustersFromRaw --procModifiers gpu,
    gpuValidation
```

For harvesting DQM outputs:

```
cmsDriver.py step4 -s HARVESTING:@trackingOnlyValidation+
    @trackingOnlyDQM --conditions auto:phase1_2022_realistic
     --mc --geometry DB:Extended --scenario pp --filetype
    DQM --era Run3 -n -1 --filein <input filename, DQM
    format>
```

where `<input filename, DQM format>` is the output of the previous step.

To generate plots comparing the results of different versions:

```
makeTrackValidationPlots.py -o <output directory> --
    extended <reference file> <target file>
```

where `<reference file>` and `<target file>` are the outputs of the DQM harvesting step for reference and target versions, respectively.