Summer Student Project Report

Prototype for GPU Port of the Clustering Algorithm for the Inner Tracking System of ALICE

Author: Nikolaus Dräger
Supervisors: Matteo Concas, David Rohr

July 2023 - August 2023

Abstract

The clustering algorithm is a key step in the preprocessing of ALICE's Inner Tracking System (ITS) detector data. It not only compresses the data for enhanced storage efficiency but also plays a vital role in the reconstruction of a particle's trajectory. To date, this computational task has been executed on the CPU. In this report, we introduce a prototype that transitions the clustering to a GPU architecture. Preliminary benchmarks yield promising results and demonstrate the feasibility of GPU-based clustering for the ITS, setting the stage for continued research and development in this direction.

1 Introduction

The clustering algorithm plays a crucial role in preprocessing the data from ALICE's Inner Tracking System. Beyond just compressing the data for more efficient storage, information about these clusters is essential for the computation of trajectories of particles in a subsequent step. The preprocessing of ITS detector data therefore hinges on the capabilities of such a clustering algorithm. Historically anchored to CPU processing, there is a growing need to explore alternatives that might offer increased speeds. ALICE is well experienced with porting other algorithms to the GPU and employs GPUs in the online event reconstruction. Thus, the infrastructure is available and the use of GPUs for more algorithms is a next logical step. In this light, our work introduces a prototype for a clustering algorithm that runs on the GPU. First, preliminary benchmarks yield great kernel execution times and motivate further study of such options.

The report proceeds as follows: Section 2 delves into the ITS and clustering algorithm. Section 3 discusses the current CPU implementation. In Section 4, we compare GPU and CPU architectures. Section 5 details the GPU-based clustering algorithm, followed by preliminary benchmark results in Section 6. Finally, Section 7 concludes and outlines future directions.

2 The Inner Tracking System and Clustering

The Inner Tracking System of ALICE consists of 24,120 silicon pixel sensors arranged in barrel-like arrays of different radii as depicted in Figure 1. Each of those sensors features a resolution of 512 × 1024 [1, 2].

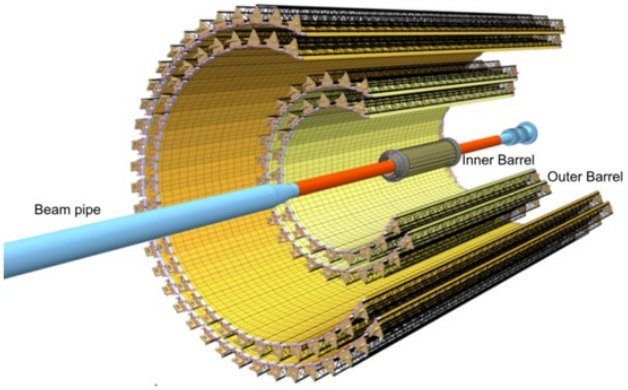

FIGURE 1: Illustration of the Inner Tracking System of ALICE [3].

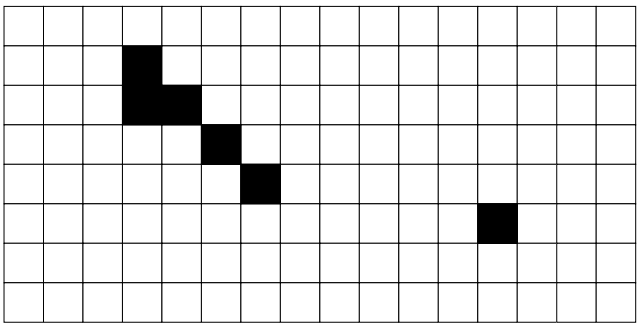

FIGURE 2: Low resolution model of a silicon pixel detectors. White pixels are in the standard state, black pixels are fired by charged particles passing through the sensor.

When a charged particle passes through a sensor, nearby pix-

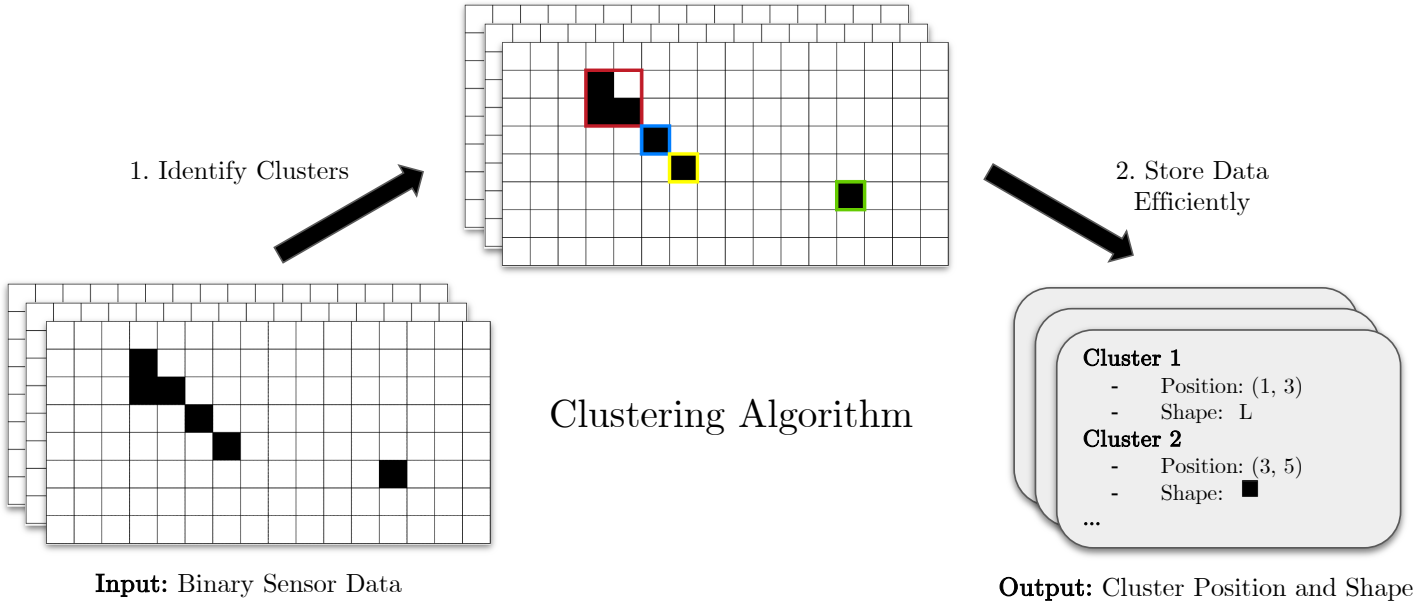

FIGURE 3: Illustration of the clustering algorithm. First, clusters are identified. In a next step the clusters found are stored in a more efficient format. It should be noted that, depending on the implementation, the red, blue, and yellow clusters could alternatively be treated as a single cluster. Ultimately, multiple heuristics for clustering are compared, and the one providing the best performance in event reconstruction is employed.

els fire and register a signal. This is depicted in Figure 2 where we, for simplicity, assume a lower resolution model of a sensor. The black pixels are the ones that are fired while the white pixels are in their default state and do not register a signal.

The silicon pixel sensor is a binary sensor meaning that a pixel either registers a hit (fires) or does not, without quantifying intermediate steps or "gray levels". Since it is undesirable to store all fired pixels for every sensor, we introduce a preprocessing step for clustering that identifies adjacent pixels that are likely to be fired by the charge of the same particle and proceeds to only store the position and the shape of a cluster together with the sensor's chip id. Not only does this enable a means of compression and thus significantly reduces the memory needed to store the information of the many sensors in the array, it also represents a necessary step in the further reconstruction of the particles' trajectories.

3 Current Solution using CPU

In the current version of O2, ALICE's software framework for the reconstruction, calibration and simulation of detector data, clusters are computed on the CPU [4]. The chips can be processed independently of each other indicating a simple strategy for parallelizing this problem. A thread pool is installed where a currently unoccupied thread is able to process the next chip. The processing stops once all chips have been processed. This method is illustrated in Figure 4.

4 Comparing GPUs to CPUs

Like shown in Figure 5, a modern CPU usually consists of multiple cores. Each core features a control unit and a very fast L1 cache. Together, these cores and their caches form a cache hierarchy within the CPU. Modern CPUs such as the AMD EPYC 9754 feature up to 128 physical cores [5].

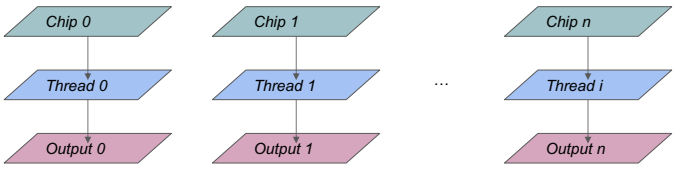

FIGURE 4: Thread pooling approach for processing chips on the CPU.

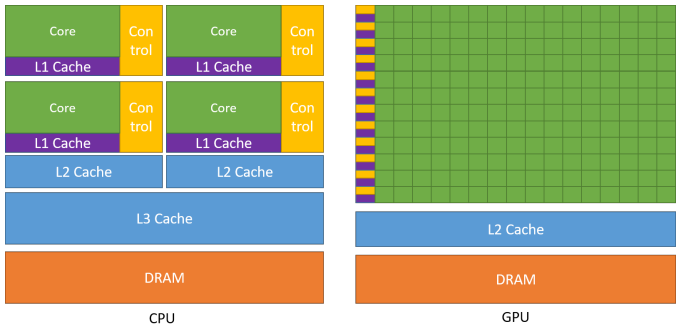

FIGURE 5: An illustration of a typical modern CPU architecture compared to a GPU architecture [7].

GPUs feature a vastly different architecture. They encompass a significantly larger number of cores. A modern GPU like the Nvidia RTX 6000 Ada, for instance, features core counts of 18,176 [6]. However, on the GPU groups of threads are bundled into what is called a "warp". Typically, on modern GPUs a warp consists of 32 to 64 threads. The threads within a warp follow a Single Instruction, Multiple Thread (SIMT) execution model where each thread in a warp executes the same instruction on a different part of the data.

While GPUs are not as versatile as CPUs in terms of the wide range of tasks they can efficiently handle - largely due to their design emphasizing parallelism - they perform well in scenarios benefiting from such parallel processing. The advantage

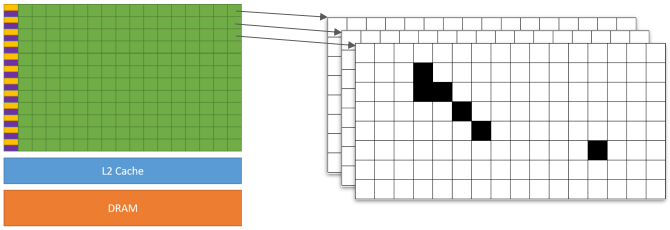

FIGURE 6: To exploit the high number of cores, each thread will be assigned a chip (or a subregion thereof) to process.

of this parallelism often translates into notable performance boosts, especially for tasks like graphics rendering and specific computational workloads.
Subsequent sections will provide a detailed exploration of our methodology for computations on the GPU.

5 Methodology

In a first step, we want to exploit the high core count of the GPU architecture in an analogous way as the current CPU version. Chips in the ITS can be processed independently of each other. We can assign one GPU thread to each chip. Such a problem is called "embarassingly parallel". Figure 6 shows the utilization of the GPU cores in this setting.

5.1 Isolation of the Problem

Before starting the actual implementation, we have to set some assumptions and boundaries to the problem. For the first prototype, to keep the code simple, several features of the original CPU version have been excluded and assumptions were made:

1. **There is no cluster size at which a cluster needs to be split for storing.**

   In the CPU implementation that is currently used in production, very large clusters will be split before storing.

2. **Pixels will not be masked.**

   In the production version, pixels can be masked and will simply be ignored.

3. **There is an efficient algorithm that precomputes smaller, populated subregions for a chip.**

   This assumption will be explained in more detail in the following sections.

4. **All tests are benchmarked with data from *pp* MC simulations of a time frame with 200 events at an interaction rate of 500 kHz.**

5.2 Extracting Regions

Before the implementation using one GPU thread per chip, let us first take a look at some approximate stats in our simulated data.

Number of chips	24,120
Number of fired chips per ROF	~100-350
Number of clusters per ROF	~175
Size of a cluster	~1-30 pixels

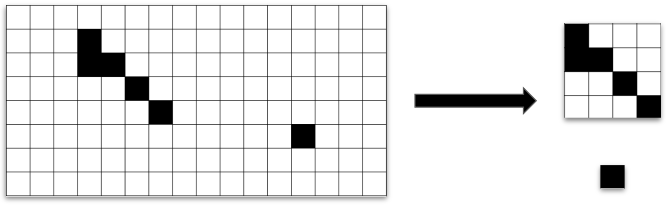

FIGURE 7: Extraction of populated regions on the chip as a means to improve the performance of the kernel.

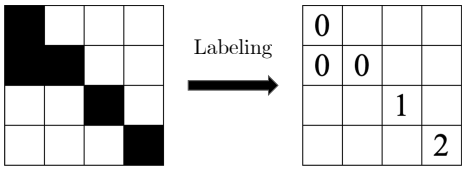

FIGURE 8: Input and output of the labeling algorithm.

From these numbers we can infer that the fired pixels are populating the sensor sparsely. This indicates that we might be able to accelerate the computation of the clusters using a preprocessing step.
We chose to implement this step in a way such that bounding boxes that are populated will be passed to the actual kernel while regions without any fired pixels will be left away. This is abstractly illustrated in Figure 7.
In this refined implementation, each thread is responsible for processing an individual extracted region instead of an entire chip, thereby reducing the processing time required per entity.

5.3 Union-Find algorithm for cluster detection

To compute the clusters, we employed a simple union-find algorithm. The algorithm iterates over all the pixels in a region and for each pixel looks at the top and the left pixel. By the design of the algorithm, both of these adjacent pixels will already have been processed. This also means that if these pixels are fired, then they are guaranteed to be part of a cluster. The following pseudocode highlights the steps that are then taken in the three different scenarios:

```
Iterate over all pixels p in a region:
    If p is fired:
        If p has no fired top of left
            ↪ neighbor:
             Create a new cluster
        If p has exactly one fired
            ↪ neighbor (either top or left
            ↪ ):
             Attach p to that cluster
        If p has both a top and a left
            ↪ neighbor:
             Merge the clusters and attach
                ↪ p
```

Figure 8 shows the input and the output of this cluster detection labeling algorithm. In a subsequent step, these labels have to be converted such that each cluster will be represented by a list of pixels and their corresponding coordinates on the chip. This information is then passed forward to a seperate compression algorithm that is currently executed on the CPU, defining the scope of what the labeling and cluster

	GPU		CPU	
	Mean (ms)	SD (ms)	Mean (ms)	SD (ms)
Region Extraction	very slow ($\sim 10^5$)	-		
Labeling	**167.30**	**13.06**		
1. Transfer to GPU Memory	142.00	13.01		
1a. `malloc` operations (174.9 MB)	73.82	12.12	***36.10***	***0.99***
1b. `memcpy` operations (5.6 MB)	1.01	0.01		
1c. `memset` operations (166.8 MB)	0.24	0.05		
2. ***Kernel Execution***	***1.60***	***0.22***		
3. Transfer to Host Memory	24.10	0.32		

TABLE 1: Mean and standard deviation of execution times for distinct stages of a clustering algorithm performed on both GPU and CPU. The GPU algorithm is divided into the region extraction preprocessing step and the labeling step. The labeling step is further decomposed into three phases: transferring data to the device memory, executing the kernel, and transferring data back to the host.

finding algorithm handles.

6 Benchmarks

We conducted preliminary benchmarks on 887 ROFs, yielding a total of 150,766 extracted regions, utilizing an Intel Core i7-8700 CPU and an NVIDIA GeForce RTX 2080 GPU. Benchmarks were repeated 10 times to compute both the mean and the standard deviation of the execution times. The resulting execution times are summarized in Table 1.
The region extraction currently bottlenecks the GPU computation. This was however expected as the extraction algorithm was not the focus of this work. The kernel execution itself, ignoring the transfer of data to and from the device memory, shows very promising performance that surpasses the CPU computation.
In our benchmarks, we observed that the data transfer times to and from GPU memory are suboptimal, which, at present, detracts from the overall performance benefits of using the GPU. These transfer times, when combined with the execution times, currently result in the GPU version underperforming compared to the CPU version. However, we believe there is substantial room for improvement to eventually take advantage of the rapid kernel execution time.

7 Conclusion

We developed a first prototype for the clustering algorithm on the GPU for the ITS2. The algorithm is integrated into the O2 framework and can be plugged into the current workflow. The algorithm shows promising performance with the option for optimization and fine-tuning to enhance the execution time.
The current implementation requires a preprocessing step for the extraction of regions. Further questions on this preprocessing step such as if it is required at all for an enhanced performance on the GPU and how this algorithm can be optimized requires a dedicated study and development.
Given that the current implementation acts as a prototype or proof of concept, several factors should be evaluated in further development:

- Kernel execution parameters such as the number of blocks and number of threads per block need to be tuned.
- The union-find algorithm might want to be extended to allow for 8-connectedness, for the possibility to identify non-adjacent pixels as one cluster and for integrating the aforementioned ability to mask certain pixels.
- In real application, some of the assumptions need revision. For example, large clusters will have to be split before compressing and storing.

Looking forward, future versions might consider extending the port to a larger fraction of the code base, including the cluster compression and storage.

Acknowledgements

I would like to thank my supervisors, Matteo and David, for their invaluable support during my project. I had the opportunity to learn extensively about C++ and GPU programming from them.
I would also like to thank the Summer Student Team for organizing such a rewarding program, complemented by the fantastic people I got to meet this summer.

References

[1] G. Aglieri Rinella. "Developments of stitched monolithic pixel sensors towards the ALICE ITS3". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1049 (2023), p. 168018. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2023.168018. URL: https://www.sciencedirect.com/science/article/pii/S0168900223000086.

[2] A. Szczepankiewicz. "Readout of the upgraded ALICE-ITS". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 824 (2016). Frontier Detectors for Frontier Physics: Proceedings of the 13th Pisa Meeting on Advanced Detectors, pp. 465–469. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2015.10.056. URL: https://www.sciencedirect.com/science/article/pii/S0168900215012681.

[3] *ALICE Collaboration — alice-collaboration.web.cern.ch.* https://alice-collaboration.web.cern.ch/menu_proj_items/its. [Accessed 24-08-2023].

[4] *GitHub - AliceO2Group/AliceO2: O2 software project for the ALICE experiment at CERN — github.com.* https://github.com/AliceO2Group/AliceO2. [Accessed 25-08-2023].

[5] Inc. Advanced Micro Devices. *AMD EPYC 9004 Series Processors Datasheet*. [Accessed 25-08-2023]. 2023. URL: https://www.amd.com/system/files/documents/epyc-9004-series-processors-data-sheet.pdf.

[6] NVIDIA Corporation. *NVIDIA RTX 6000 Ada Datasheet*. [Accessed 25-08-2023]. 2023. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/proviz-print-rtx6000-datasheet-web-2504660.pdf.

[7] *CUDA C++ Programming Guide — docs.nvidia.com.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed 24-08-2023].