FPGA Cluster algorithm within CMSSW

Summer student end report

Max A. Bensink

Supervisors: Olena Karacheban, Cristina Oropeza Barrera, Stefano Mersi

Abstract

This study aims to evaluate the performance of the BRIL clustering algorithm in handling data from a particle detector, with a specific focus on its linearity across varying levels of pileup (concurrent collisions). The algorithm's performance is also compared with an existing High-Level Trigger (HLT) algorithm. Datasets with varying pileups, from 0.5 to 200, are analysed using ROOT modules. Metrics such as hit maps, mean cluster counts, and cluster sizes are considered. Results indicate that the BRIL algorithm maintains its linearity up to high pileup levels and tends to identify slightly more clusters than the HLT algorithm, likely due to the absence of hit-filtering in BRIL. The study suggests potential improvements such as implementing thresholding and removing single-hit clusters for better computational efficiency. Future work could include a more detailed cluster-to-cluster comparison with HLT and testing the algorithm at extremely high pileups to explore its upper operational limits.

Table of Contents

1. Introduction

The Compact Muon Solenoid (CMS) Inner Tracker Endcap Pixel Detector plays a key role in luminosity measurements during Phase 2 of the Large Hadron Collider (LHC). While a basic clustering algorithm has already been implemented in the Field-Programmable Gate Array (FPGA), further fine-tuning is required for optimal performance. The scope of this project is to integrate this clustering algorithm into the CMS software (CMSSW) simulations, thereby identifying any issues within the algorithm and determining its overall performance.

In the context of particle accelerators like the LHC, luminosity is a critical performance indicator. It quantifies the number of particles colliding within a specific area and time frame. Luminosity is formally defined as the number of events per cross-sectional area per unit time, typically expressed in units such as $cm^{-2}s^{-1}$.

Any detector which provides count linear to pileup can be used as luminometer. Silicon pixel detectors are among the types of detectors that can be used for this purpose. They are composed of a grid of pixel cells that are sensitive to charged particles passing through them. Each activated pixel generates an electric signal that can be counted.

Clusters are a group of neighbouring pixels that were activated by a single particle, and determining the luminosity can be done by counting this number of clusters within a Silicon Pixel Sensor. It is crucial to determine the number of clusters instead of counting the number of activated pixels, as single hit counting is more sensitive to noise.

To address this challenge, the Beam Radiation, Instrumentation, and Luminosity (BRIL) project has developed an FPGA-based clustering algorithm. This algorithm processes data from pixel readout chips and computes the clusters detected within those pixels.

At the moment, testing this algorithm is time-consuming due to the complexities involved in programming and testing FPGA software. By implementing this algorithm within CMSSW, challenges such as these can be mitigated, particularly as the code will be written in C++, making it easier to understand and debug.

2. The BRIL clustering algorithm

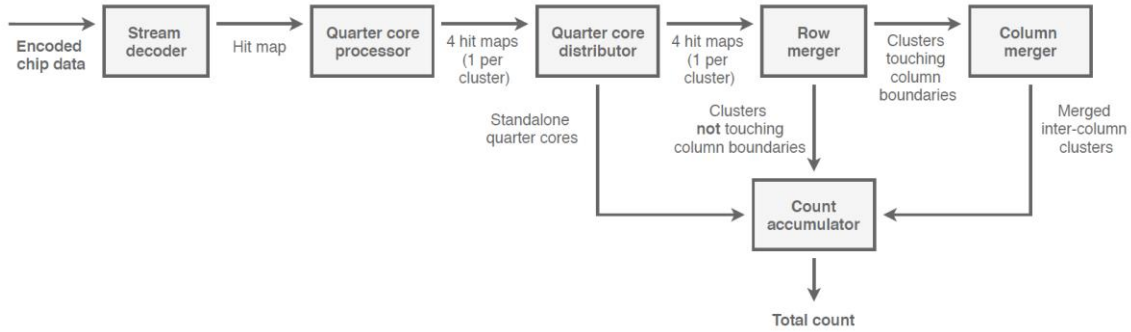


Figure 1: FPGA based algorithm individual processes. The data received from the chip must first be decoded. 4 consecutive processes are then used for computing the clusters[1].

The clustering algorithm is designed as a series of parallel processes, each tasked with examining a different part of the incoming data (see Figure 1)[1]. The data is sent in sets of 16 pixels called “Quartercores”. The arrangement of these pixels can either be a 2x8 matrix or a 4x4 matrix, depending on the width and height of a single pixel (see appendix A). The readout chip is agnostic to pixel size and pixel matrix size [2]. This means that the reconstruction of the pixel data must be done in the algorithm.

2.1. The Stream decoder

The data sent by the Pixel Readout Chip (ROC) is transmitted and contains a set of information that is needed to reconstruct the data from the pixel sensor. The data is sent one column at a time. Within each column, the information from each row of Quartercores is sent in sequence before moving on to the next column. The following information is transmitted per Quartercore:

- The row and column of the Quartercore.
- A compressed binary tree holding the information of the activated pixels (also called hits).
- A flag for if the Quartercore is a neighbour of the previous sent Quartercore.
- A flag for if the Quartercore is last within that column.
- A flag for if the Quartercore is the last within the transmitted data for the ROC (meaning last in event).
- The Time Over Threshold (TOT) value of the hits.

This data is decoded and then placed into a data structure which is then sent to the Quartercore processor.

2.2. The Quartercore processor

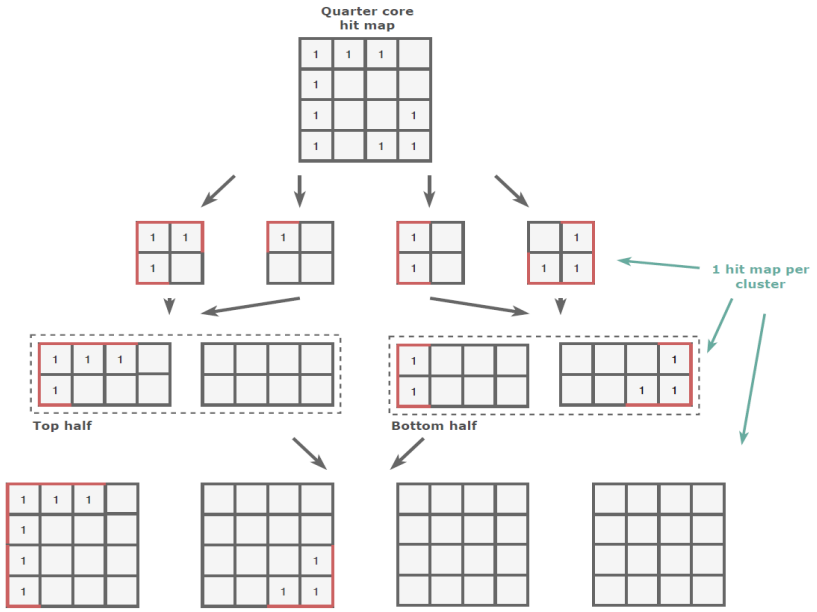


Figure 2: Quarter core processor steps[1].

This part of the algorithm identifies individual clusters that can be found within a single Quartercore. The process consists of three steps (visualised in Figure 2):

1. The procedure initiates by dividing the Quartercore into four 2x2 matrices.
2. It subsequently examines the upper and lower sections to determine if there are any hits on the inner pixels of both the left and right 2x2 matrices. If hits are detected, they are combined, resulting in a 4x2 matrix.
3. The final step mirrors the second step, but this time it involves the inner pixels at the top and bottom of the 4x2 matrices. If so, the matrices are combined into a 4x4 matrix.

The identified clusters are then added to the Quartercore dataset which is then sent to the distributor.

2.3. The Quartercore distributor

The distributor's decision relies on two key sets of information:

- Details about the hits within the Quartercore, specifically whether any hits touch the outer edges of the Quartercore.
- Metadata provided by the ROC, such as whether the current Quartercore is adjacent to the previous one, is the last in its column, or is the last in the entire set of data for that event.

Based on the identified clusters within each Quartercore, the Quartercores can take two paths:

- The number of clusters will be summed and then added to a counter.
- Alternatively, it may pass through a row and column merger to evaluate if clusters from different Quartercores should be combined.

Taking these factors into account, the distributor determines whether the data from the Quartercore should proceed for further analysis or can be sent directly to the accumulator for counting.

2.4. The row merger

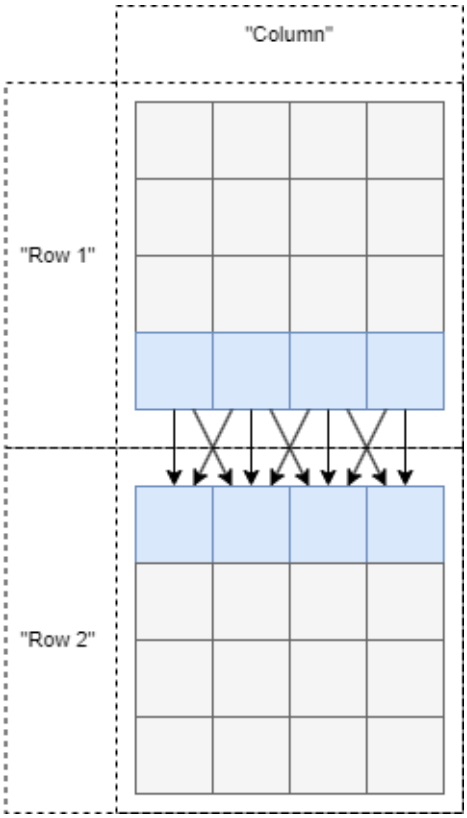


Figure 3: which pixels are checked for touching in the row merger. The algorithm also checks if there are pixels touching from just the corners.

When two neighbouring Quartercores are sent to the row merger, the process initiates by examining the first Quartercore for clusters that have contact with clusters from the second Quartercore (shown in Figure 3). When a touching pair of clusters is identified, the two clusters are merged into a unified cluster. This unified cluster is then directed to a secondary processing buffer, where it can undergo comparison with other clusters from the same Quartercore.

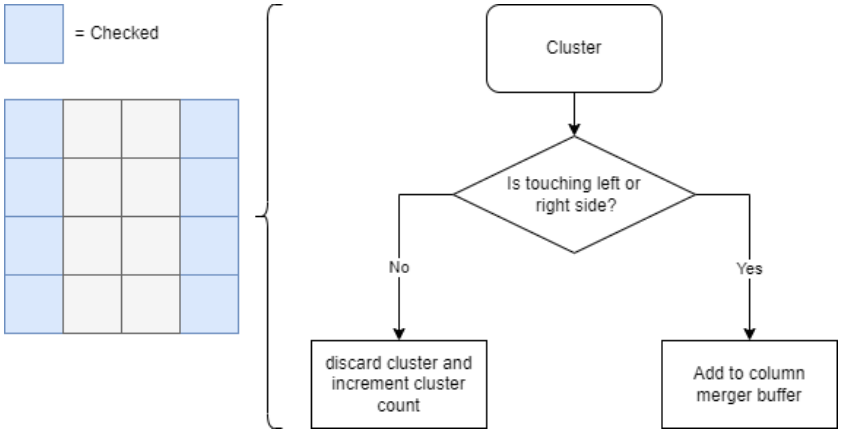


Figure 4: Example of what pixels within a cluster are checked after row merging has finished.

After completing all possible cluster mergers, each cluster undergoes one of two actions (visualised in Figure 4):

- If a cluster isn't touching the left or right side of the Quartercore, the cluster is discarded and the counter for clusters found within the row merger is incremented.
- If further processing is necessary, the clusters are forwarded to the column merger stage.

2.5. The column merger

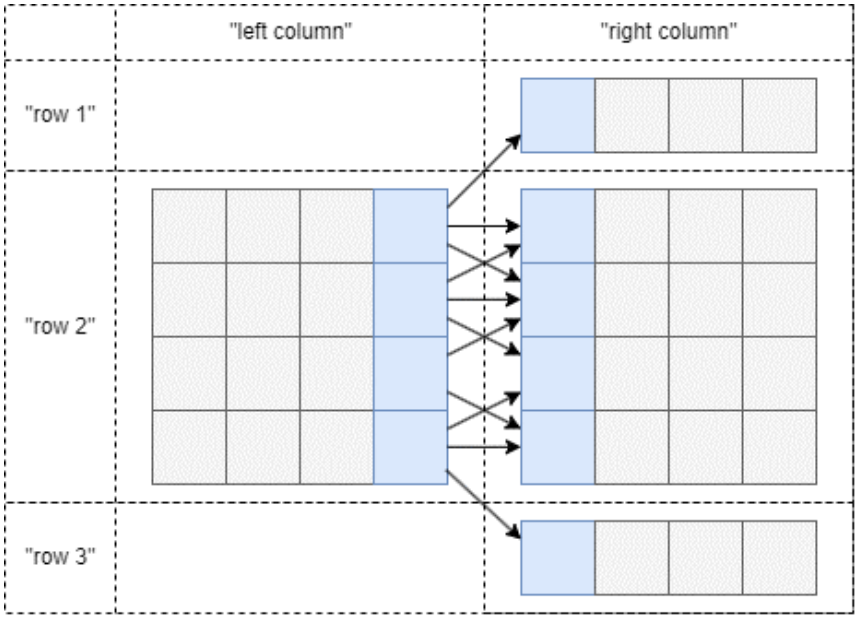


Figure 5: Clusters from the left column are compared to clusters from the right column. Here, if there are any pixels (marked blue) touching each other, they will be merged. Clusters one position higher or lower are also checked for pixels touching the corners.

This process is fairly similar to the row merger. The same steps are taken for the clusters but then done for merging on the left and right side of the clusters. Here, when 2 columns are received from the row merger, it starts iterating per cluster in the left column if there are any touching clusters – even the corners of the clusters – in the right column (see Figure 5). When a neighbouring cluster is found, the resulting merged cluster is then stored within the

right column buffer. This process is then looped over for every cluster within the left column.

If a cluster from the left column doesn't touch any in the right column, it is discarded. The counter for clusters found within the column merger is then incremented.

2.6. The accumulator

This part of the algorithm checks whether any of the previously explained processes is still running. The processing will be done once a Quartercore with the “is last in event” bit turned on has been processed.

Once this Quartercore comes through, the accumulator will sum up the counters from the distributor, row merger and column merger. This value is then stored for data analysis as this is proportional to the pileup of the event, therefore can be used for the luminosity measurement.

The accumulator also stores additional data like the warnings and errors given by the processes. This is critical for identifying data corruption within the design of the algorithm. These warnings and errors are the following:

1. Buffer overflow errors for:
    - Stream decoders’ input and output buffer
    - Quartercore processors’ input and stage 3 buffer
    - Row buffers’ input and processing buffer
    - Column buffers’ input and processing buffer
2. Stream decoder bitstream error
    - This error is raised when the encoded bitstream cannot be decoded.

2.7. Benchmarking of the BRIL clustering algorithm

A CMS standard clustering algorithm, known as the High-Level Trigger (HLT) algorithm, is used to evaluate the performance of the BRIL clustering algorithm. Unlike BRIL, the HLT algorithm incorporates ADC information during the cluster formation process. As a result, minor variations in cluster counts between the two algorithms are to be expected, although the identified clusters should largely overlap.

3. Implementing the algorithm in C++

The CPU based clustering algorithm must be identical to the clustering algorithm used for the hit data in the FPGA. Simply converting the code from VHDL to C++ is not easily done as they are both used for different use cases. Both use their own architecture to their own benefit [3]. Listed below are some key differences:

Hardware vs software logic

FPGAs are hardware devices that allow to create custom digital circuits by configuring the logic gates and interconnections on the chip. Using an FPGA involves designing digital circuits using Hardware Description Languages (HDLs) like Verilog or Very (High Speed integrated Circuit) Hardware Description Language (VHDL).

CPUs have a standard structure that can be controlled using a set of instructions (x86 or ARM for example). These instructions tell the processor how to handle certain data within memory such as addition, subtraction, multiplication et cetera. Higher level programming languages like C++, python and Java can then be used to handle these instructions in a readable and manageable format.

Customization

FPGAs excel in parallel processing and can be highly customised to perform specific tasks efficiently. It's possible to create multiple hardware components that operate concurrently and optimize the design for your specific application.

CPUs are optimised for sequential processing of instructions. This makes it easier to debug the data in between processing steps. CPUs can also utilise multiple cores to parallelise data processing but is still limited to the CPUs per core performance and instructions are still processed sequentially.

3.1. From parallel to sequential

In the C++ implementation of the clustering algorithm, the parallel tasks managed by the FPGA will be executed one after the other in a sequential manner. This implies that, for a single pixel frame, each task will operate over the entire frame. The result of each task will be stored in an array, and once the previous task completes processing all the data, the array's contents will be transmitted to the next task. This approach avoids the complexities of programming multiple threads that require intercommunication.

3.2. Retrieving hit data from CMSSW

For testing and verifying the algorithm, hit data from the CMSSW simulation is utilized. In this simulation, proton-proton collisions are modelled, and the products of these collisions are processed through the geometry of the CMS detectors. Using the GEANT toolkit, the pixel detectors' response to the particle trajectories is then subsequently calculated and stored.

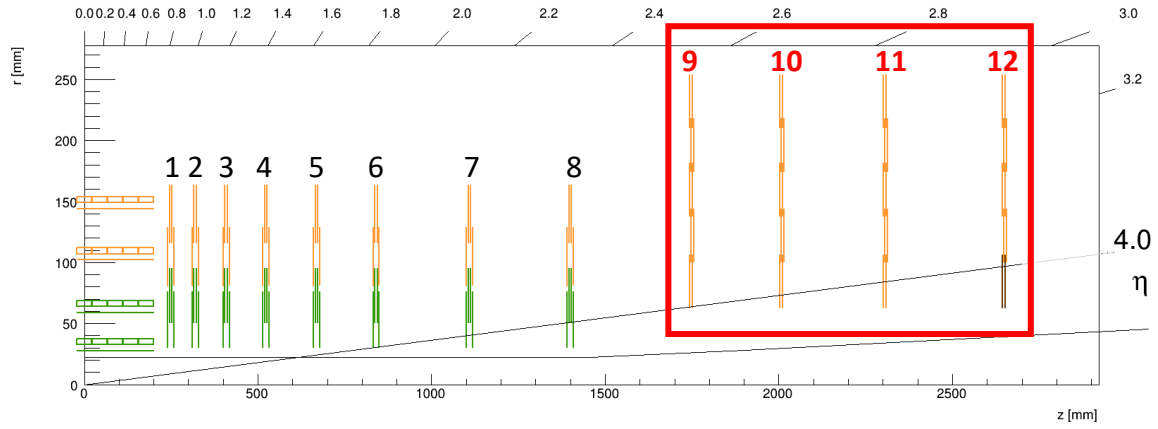


Figure 6: +z side of Inner tracker with the Tracker Endcap Pixel Detector (TEPX) highlighted.

This report is based on the simulation of hits that were found within the Tracker Endcap Pixel Detectors (TEPX). These consist of a set of 4 disks on the end of +z and -z within the Inner Tracker (see Figure 6) [1]. Each disk is built from 176 modules to cover this entire area within the CMS detector.

Using the PixelDigi class within CMSSW [4], hit information from each individual TEPX module can be extracted. So be able to separate the modules from each other, extra information is added about the modules position, such as:

- End of the CMS (+z and -z).
- Disk on that side (9 to 12).
- Ring on that disk (1 to 5).
- Layer on that disk (front or back).
- Module index within that layer.

Hardcoded IDs for extracting the layer of the module

In the current version of CMSSW (CMSSW_13_0_10), there is no direct way to extract the layer of where the module is located. This information is however held within the ID of the module. The modules are grouped per front and back layer, meaning they can be checked if an ID falls in the range of the front layer modules or back layer modules. These ranges are hardcoded into a plugin to allow the code to identify which layer the module is from [5].

3.3. Storing information from algorithm

Clusters found in the CMS SW simulation by the clustering algorithm will be exported and made available in root files. Key information about the clusters such as the position, dimensions, and number of hits within the cluster will be stored by the clustering algorithm for data analysis.

3.4. Software warnings

Some warnings were created to identify issues within the C++ version of the BRIL clustering algorithm. These exist out of the following:

Row merger

1. Processor buffer not empty:
    - This warning is called when the algorithm tries to go to the next column with data while there are still clusters to be processed.
2. Already used:
    - This warning is called when the algorithm tries to merge 2 clusters that were already merged before.
3. Not found:
    - Not found means that the algorithm found a cluster to merge from another Quartercore, but cannon find it back in the buffer.

Column merger

1. Not adjacent:
    - This means that the algorithm tries to merge two clusters that are one or more columns away from each other, meaning that they cannot be merged.
2. Already used:
    - Same as row merger.
3. Not found:
    - Same as row merger.

3.5. Starting up the simulation

The plugin requires access to cernbox for accessing the pre-generated hit data and storing it in the user's cernbox drive. Because of this, it's recommended to run the simulation from LXPLUS or any other computer that has direct access to cernbox.

The git repository can be cloned into the "src" folder of the CMSSW installation. When this is done, the code first needs to be compiled. Afterwards, the following command can be run from the console:

```
1. cmsRun itbrilclusterexporter/python/ITBRILClusterExporter.py \
2.         xrdredirector=file: \
3.         dataset=PU_75.0_D91_cbarrera \
4.         output=output_file_PU_75.0_cbarrera.root
```

This command will generate the cluster data from a pileup (PU) of 75. Changing the dataset will allow the user to test the clustering algorithm on different pileups[6].

4. Analysing the data

4.1. Pileups used for analysis

Datasets have been provided for determining the clusters with the BRIL clustering algorithm. A different pileup is used for each file, where a higher pileup corresponds to a higher number of proton-proton collisions. This is done to understand how the algorithm will respond to an increasing number of particles flying through the detector and activating more pixels on each TEPX module.

The datasets contain data for pileups of 0.5, 1.0, 1.5, 2.0, 10.0, 30.0, 50.0, 75.0, 100, 120, 140, 160, 180 and 200. Here, pileups 0.5 till 2.0 contain 10'000 events whereas the others contain 1'000.

4.2. Plotting the data

A set of ROOT modules have been made for plotting different information about the clusters [7]. These can be loaded into the ROOT Command Line Interface (CLI). A function is added called get_tree_data to automatically extract the tree objects from the root files. The following graphs were made to plot the data:

- A hit map plot where the hits and clusters of a single module are visualised (Figure 13).
- Distribution of clusters reconstructed per event per ring (Figure 7)
- The mean cluster count per ring for one disk as a function of pileup (Figure 9).
- The mean cluster size as a function of pileup (Figure 12).
- The fraction of clusters with more than one hit per pileup (Figure 17).
- Ratio of found clusters between custom BRIL clustering algorithm and CMS standard HLT clustering algorithm (Figure 15).

More detailed instructions for the reproducibility of the plots are provided on Gitlab [7].

5. Results

Testing the performance of the BRIL clustering algorithm is done in 2 ways:

- Determining the linearity of the algorithm as a function of pileup.
- Comparing the BRIL clustering algorithm to the HLT clustering algorithm.

All the results are made from the cluster data of disk 9 in the Inner Tracker (see Figure 6).

5.1. Linearity of the clustering algorithm

Average number of clusters per ring instead of per module

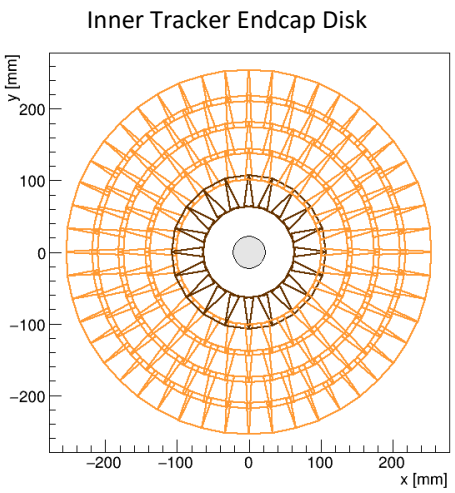


Figure 8: A single disk of TEPX-modules. The ring is composed out of 5 concentric rings with these modules.

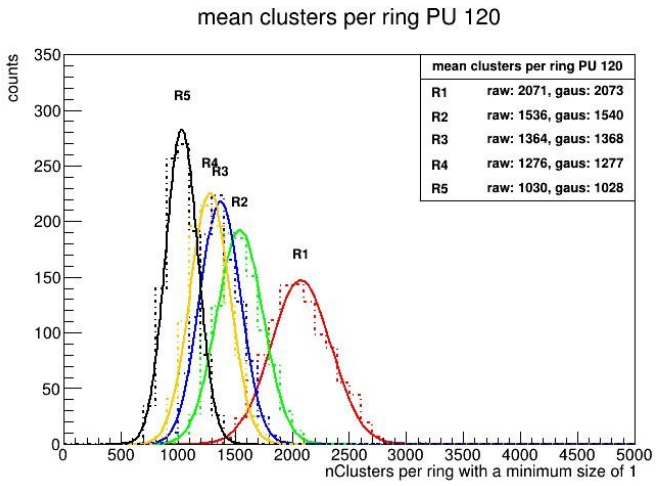


Figure 7: A plot of showing the mean number of clusters found per ring. the data shows that the inner most ring (R1) has a higher occupancy than the most outer ring (R5).

In the CMS Inner Tracker, each disk is composed of five concentric rings, with Ring 1 being the smallest and closest to the interaction point and Ring 5 being the largest and farthest away (see Figure 8). Due to their varying distances from the interaction point, these rings experience different rates of particle interactions. Specifically, Ring 1, being closer to the interaction point, is likely to record more particle interactions than Ring 5. The likelihood of interactions diminishes for the outer rings, not only due to the increased distance but also because of the larger angle diverging from the trajectory of particles emanating from the interaction point (as can be seen in Figure 7).

For this reason, determining the average number of clusters will be done per ring as opposed to per module.

Mean number of clusters per ring per pileup

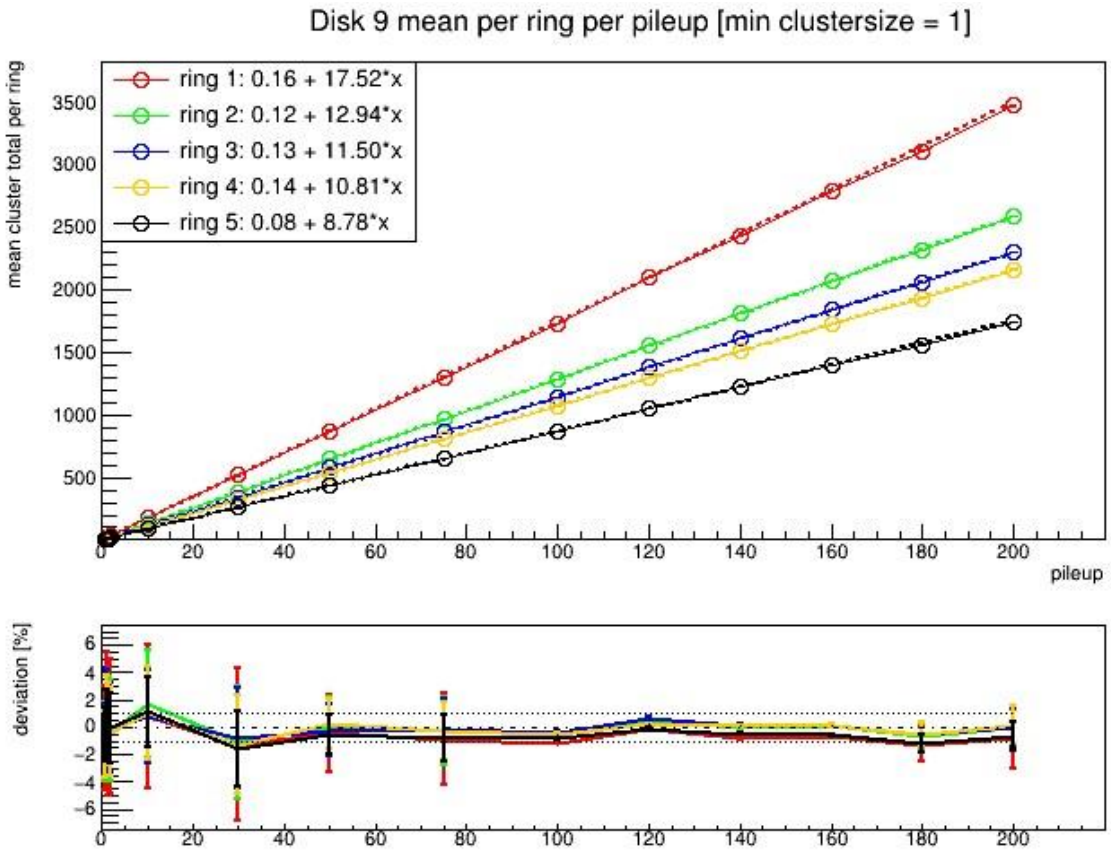


Figure 9: mean cluster count per ring per pileup. The lower graph shows the deviation of the raw data compared to the fit done over the data from pileups 0.5 to 2.0.

The linearity of the algorithm can be determined by calculating the average number of clusters per ring per pileup. Figure 9 shows the mean cluster count per ring plotted as a function of pileup. A Linear fit is then added at low pileups (below pileup of 2, as normally calibration of the detector is done in this low pileup range) and extrapolated to high pileup values. Deviation from this linear fit is the measure of non-linearity of the algorithm. In Figure 9, the deviation of each measurement point from the linear fit is shown in the lower part of the plot. As can be seen from the plot, the number of clusters grows linear with pileup. The deviation graph shows that the raw data can have an error of around +/-1.5% from the fit.

Linearity when counting bigger clusters

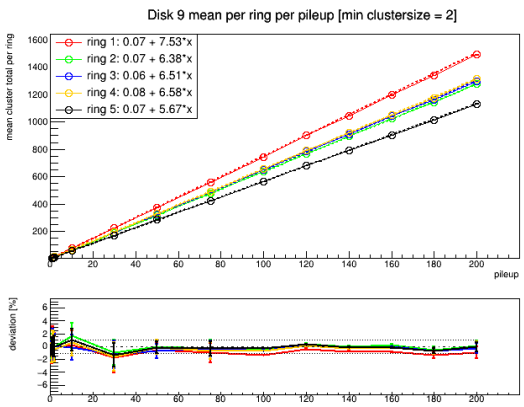


Figure 10: mean cluster count per pileup with minimum cluster size is 2

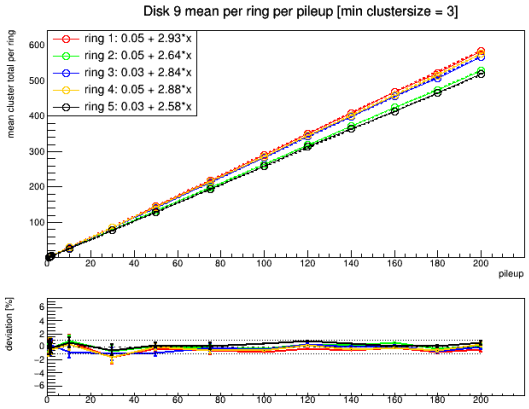


Figure 11: mean cluster count per pileup with minimum cluster size 3

Figure 10 and Figure 11 show plots of the mean cluster count per pileup where the minimum size of the clusters has to be at least 2 or 3 pixels. The deviation from the fit seems to become less when looking at bigger clusters. When choosing a minimum clusters. Further study can be done for what cluster size range could be used for improving the linearity of the data used for determining the luminosity.

Mean cluster size per pileup

In case of hight occupancy at high pileup, clusters can start merging. This would result in the increase of the average cluster size as a function of PU. This must be considered and tested.

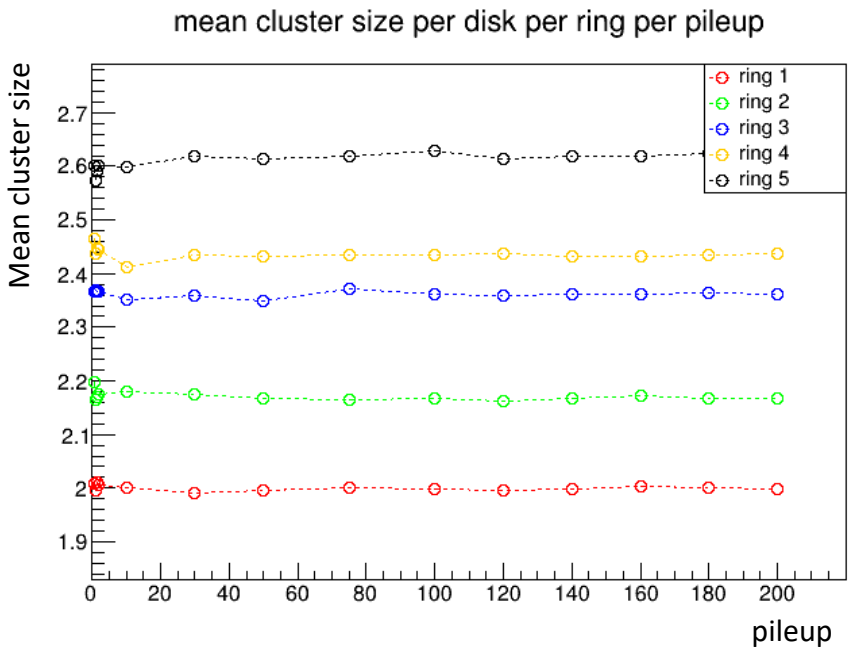


Figure 12: mean cluster size per ring per pileup.

As shown in Figure 12, the average cluster size is constant for pileups 0.5 to 200. This means that for the current maximum pileup, occupancy is still low and cluster merging does not have an impact.

This can also be seen within the hit map of a single module within the detector:

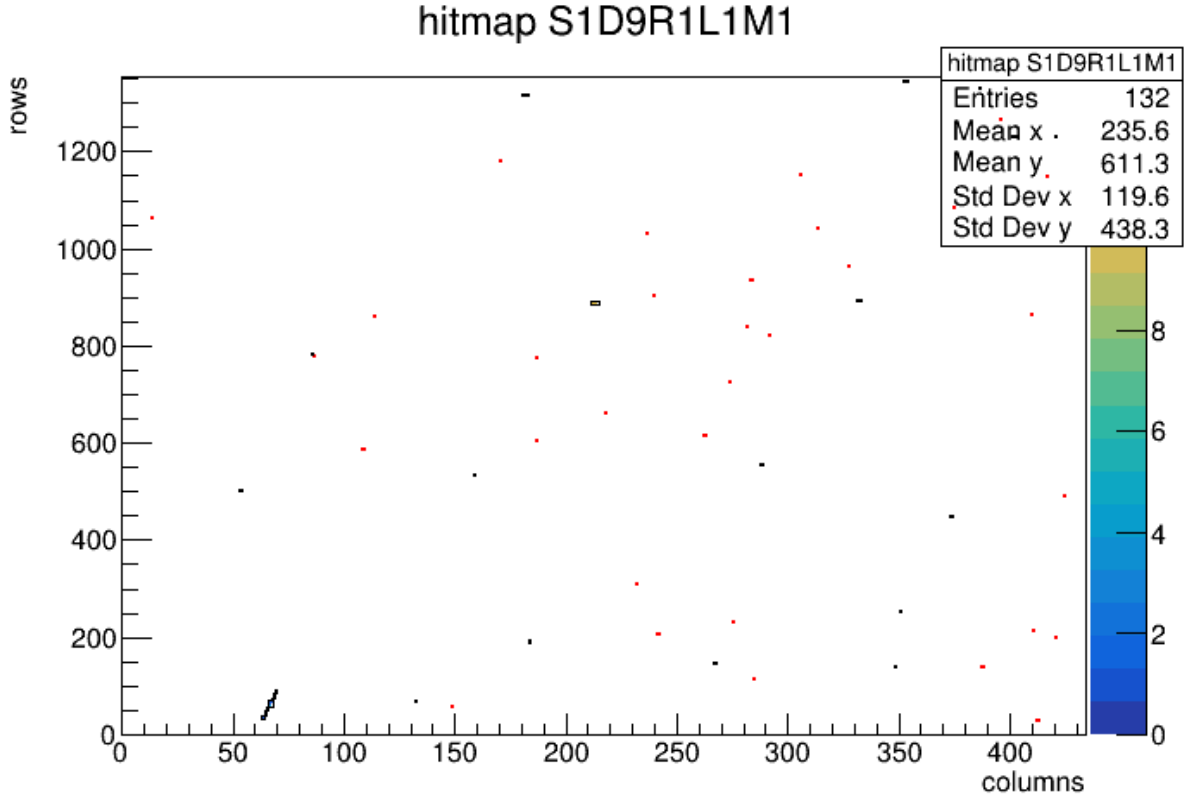


Figure 13: hit map of module one (M1) within disk 9 (D9) on the first layer (L1) of ring one (R1) on the left side (S1) using a pileup of 200.

The hit map shown in Figure 13 was generated for a pileup of 200. The occupancy on the module shows to be not high enough for unwanted cluster merging.

5.2. Performance compared to the HLT algorithm.

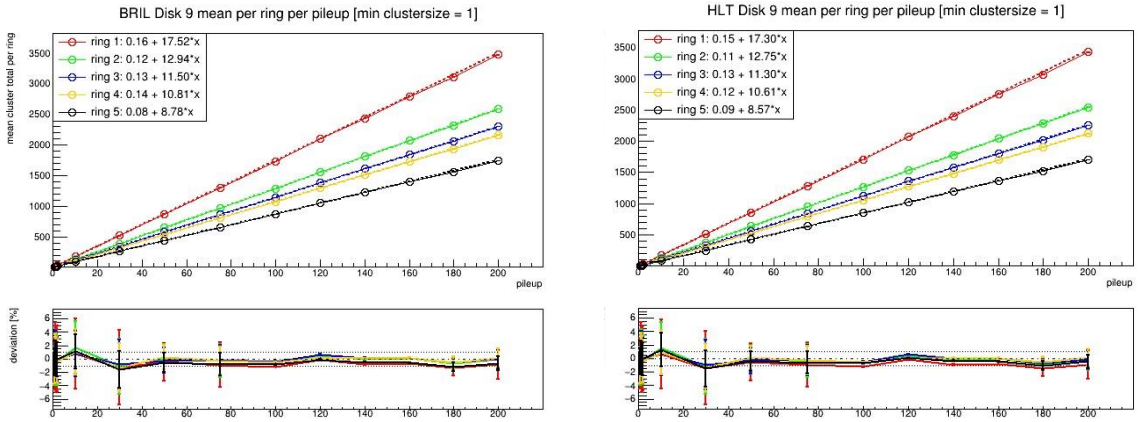


Figure 14: plot of mean cluster per ring per pileup for both HLT and BRIL algorithm.

By plotting the same graphs for HLT as was done for BRIL (shown in Figure 14), it is possible to identify small differences between the number of clusters found by both algorithms. Both algorithms show very similar results for the data and the fit. There is however a small difference in the constants used for the fits. This means that both algorithms identified a different number of clusters per pileup.

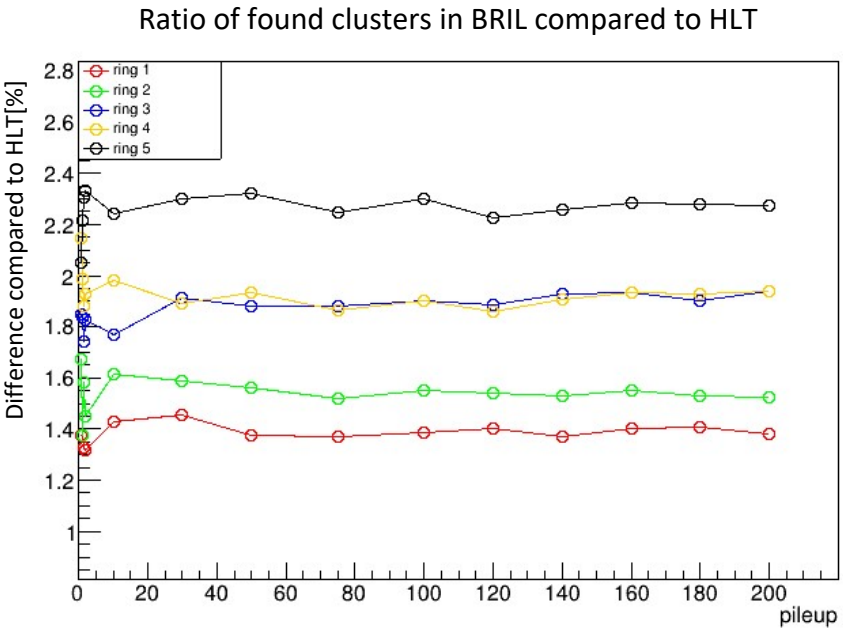


Figure 15: ratio of clusters found by BRIL algorithm in comparison to HLT algorithm.

In Figure 15, the BRIL algorithm on average finds between 1.3% to 2.4% more clusters than the HLT algorithm. In the graph, the HLT algorithm seems to find less clusters for each ring. The same can also be seen when looking at the distribution of the found clusters by both algorithms for a single pileup.

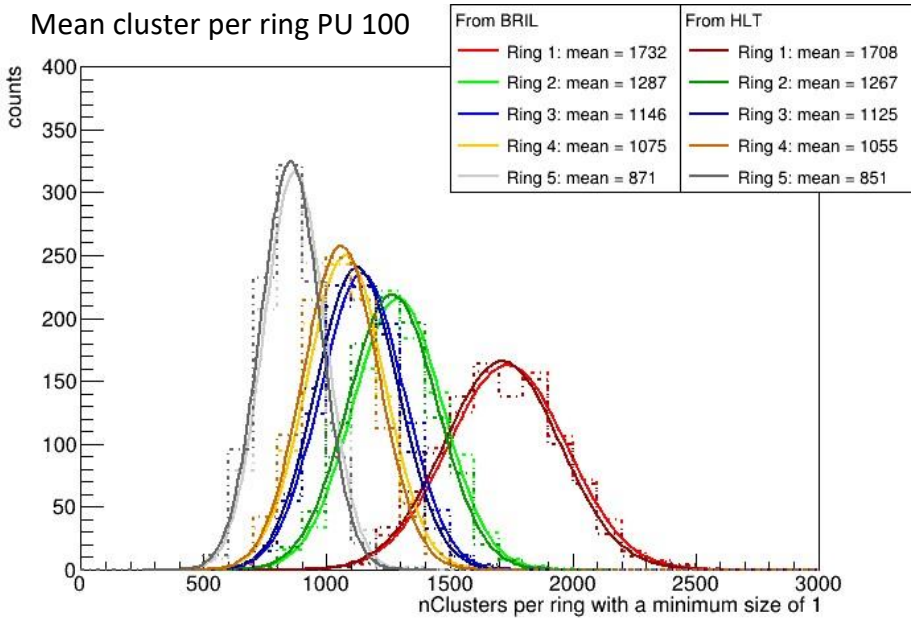


Figure 16: HLT and BRIL plot for mean cluster per ring for PU 100

In Figure 16, the mean cluster count per ring for PU 100 is plotted for both HLT and BRIL. The small shift in average cluster count can be seen per ring. This difference in cluster count is however expected, as the HLT algorithm implements a threshold over the found hits [8]. This threshold removes hits from the data with a low ADC value, meaning that there will be less clusters formed by the algorithm.

There is continuation of this study: per cluster check, where individual cluster differences of complicated shape can be compared between two algorithms.

5.3. High occupancy of single hit clusters

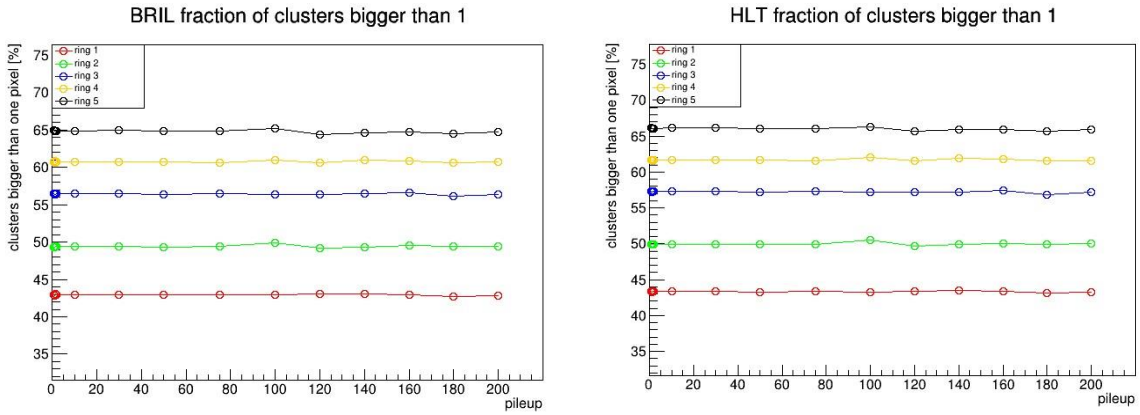


Figure 17: percentage of clusters bigger than 1 hit per pileup for both HLT and BRIL

Single hit clusters are clusters that only consist out of one pixel. In Figure 17 it is shown that the percentage of single hit clusters is constant over all pileups for both the HLT and the BRIL algorithm. in will be useful to check what is the fraction of these single hit clusters, which have low ADC count. These are most probably noise and removing low amplitude single hit clusters will increase stability of the algorithm.

5.4. Split cluster formation in HLT

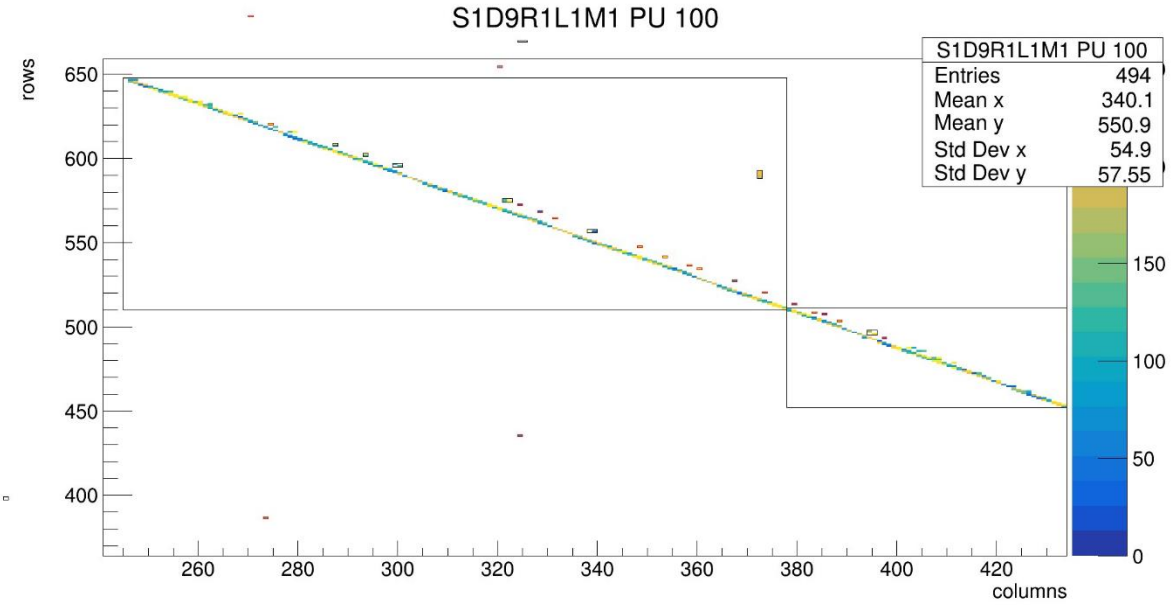


Figure 18: In this hit map generated using the HLT algorithm, what initially appears as two distinct clusters is, upon closer inspection, revealed to be a single, elongated trace that has been divided into separate clusters.

While looking through the hit maps of the HLT algorithm, a strange anomaly was found where a large cluster was split into 2 smaller clusters (see Figure 18). As of now it is unknown why this happens, but a hypothesis is that the algorithm has an internal

hardcoded limit for the size of the clusters. If this is a feature from the algorithm it should kept in mind when comparing it to the BRIL clustering algorithm.

6. Conclusion

The task of this project was to implement a C++ version of an FPGA-based clustering algorithm and to study the linearity of the algorithm as a function of PU. The results show that the BRIL algorithm is linear up to a PU of 200 (expected PU in LHC Phase 2).

The BRIL and HLT algorithms show similar results, but a small difference in results is observed due to an ADC threshold applied for clustering done within the HLT algorithm. As a continuation of this study, I proposed a function which could be used to compare individual clusters between BRIL and HLT algorithm to deepen the understanding of the results. It would also be interesting to implement the TOT variable in the BRIL algorithm, thus allowing the algorithm to remove hits with a too low value (thresholding).

Another addition to the algorithm could be to remove single hit clusters with low ADC count from the dataset. Fraction of these low amplitude single hit clusters needs to be evaluated beforehand.

7. References

[1] "The Phase-2 Upgrade of the CMS Beam Radiation, Instrumentation, and Luminosity Detectors Technical Design Report CMS Collaboration," 2021, Accessed: Sep. 03, 2023. [Online]. Available: https://cds.cern.ch/record/2759074/files/CMS-TDR-023.pdf

[2] "The RD53B-CMS Pixel Readout Chip Manual."

[3] "FPGA vs. GPU vs. CPU: Which Is the Best Choice for Your Application? - RAYPCB." https://www.raypcb.com/fpga-vs-gpu-vs-cpu (accessed Sep. 03, 2023).

[4] "PixelDigi.h." https://github.com/cms-sw/cmssw/blob/master/DataFormats/SiPixelDigi/interface/PixelDigi.h (accessed Sep. 03, 2023).

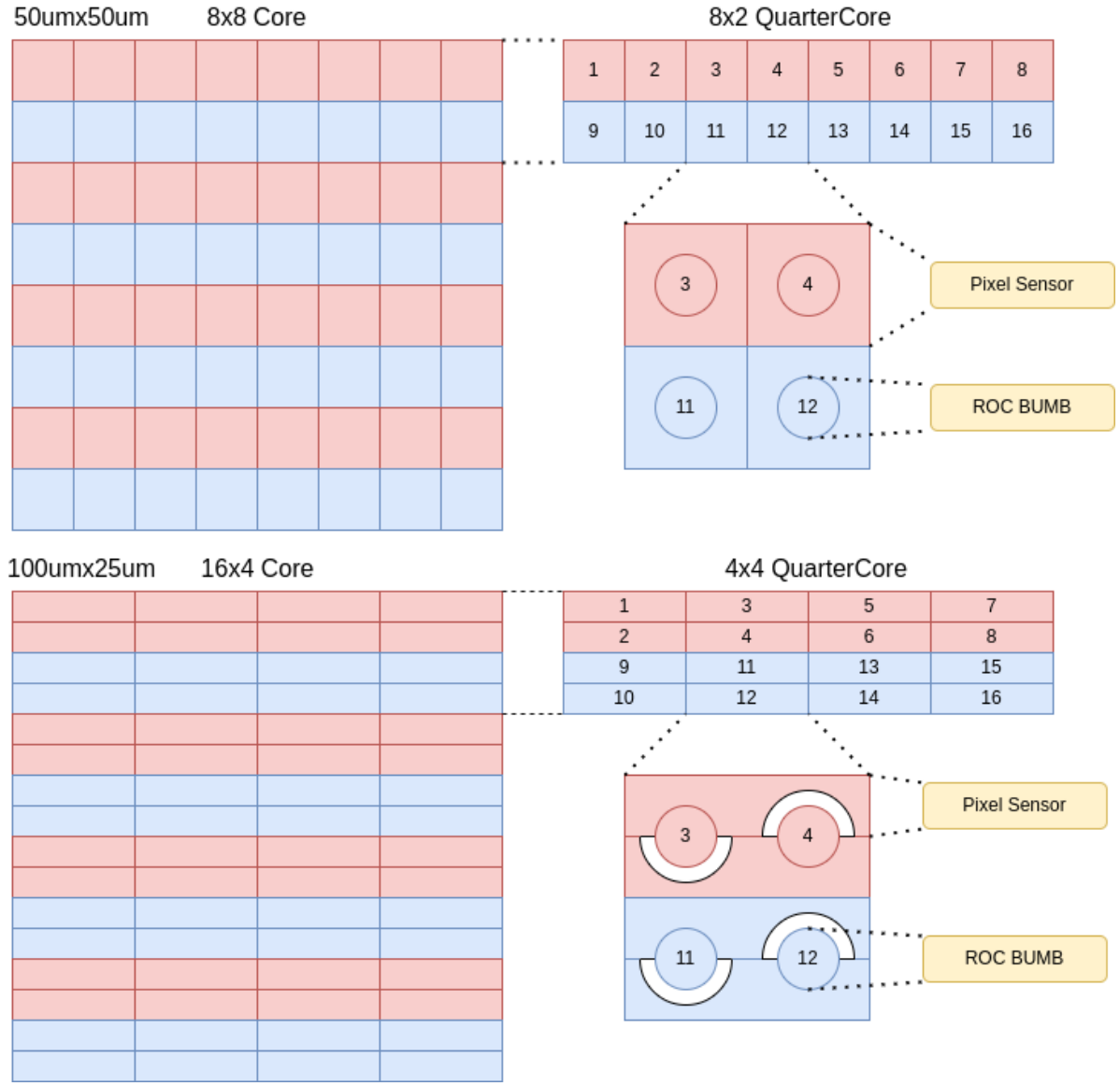
[5] M. Bensink, "ITBRILClusterExporter hardcoded ID ranges." https://gitlab.cern.ch/mbensink/bril_it_analysis/-/blob/master/itbrilclusterexporter/plugins/ITBRILClusterExporter.cc#L132 (accessed Sep. 03, 2023).

[6] M. A. Bensink, "BRIL IT Analysis CMSSW plugins," 2023. https://gitlab.cern.ch/mbensink/bril_it_analysis (accessed Sep. 04, 2023).

[7] M. A. Bensink, "BRIL IT Analysis ROOT modules," 2023. https://gitlab.cern.ch/mbensink/it_root_modules/-/tree/master (accessed Sep. 04, 2023).

[8] "SiPixelClusterizer - cms-sw/cmssw." https://github.com/cms-sw/cmssw/blob/master/RecoLocalTracker/SiPixelClusterizer/doc/SiPixelClusterizer.doc (accessed Sep. 05, 2023).

Appendix A: Pixel dimensions and readout chip



The diagrams above show how the different dimensions of the pixels impact the way these would be bonded to the readout chip. The 50um x 50um pixels would be readout line by line, whereas the 100x25um pixels are readout per 2 rows. Here, every second pixel readout is from the second row (or vice versa depending on the orientation of the pixels).