Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

CLUEstering: a high-performance density-based clustering library for scientific computing

Supervisor:
Prof. Daniele Bonacorsi

Co-supervisors:
Prof. Francesco Giacomini
Dr. Felice Pantaleo, CERN
Dr. Wahid Redjeb, CERN

Submitted by:
Simone Balducci

Academic Year 2023/2024

Abstract

Clustering is a computational technique that aims at classifying objects based on their similarity, and is widely used in many branches of science nowadays, for instance in image segmentation, medical imaging, study of complex systems, machine learning techniques and high-energy physics.

As the amount of data collected in every field of research increases, techniques like clustering will have to deal with an increasing amount of data, which will keep increasing faster than the rate at which the hardware is evolving. This requires to find new ways to handle this data as efficiently as possible.

In the last decades, parallel processors like GPUs and FPGA have risen in popularity, thanks to their ability to perform complex calculations very efficiently by executing a large number of operations in parallel.

The purpose of this thesis is to develop a general-purpose clustering library based on the CLUE algorithm [1], a highly parallel density-based clustering algorithm used for the local reconstruction of hits in the high-granularity calorimeters of the CMS detector at CERN. CLUEstering is developed using the Alpaka library, a C++ performance portability library that allows to write code that runs on many types of modern processors with near-native efficiency and without any code duplication. The library is developed with a Python interface to the C++ backend, in order to make it easier to use and appeal to a wider range of users.

In the end the library was tested on selected datasets in order to assess the quality of its reconstruction and benchmark its performance. Also, to show its generality it was applied to two modern problems from two separate areas of science: *vertex reconstruction* in high-energy physics and *stars detection* from PSF images in astronomy.

Contents

Listings

List of Tables

List of Figures

List of Algorithms

Chapter 1

Clustering

Clustering is a computational technique used to group objects from a larger set based on some kind of similarity. It is used in many branches of science, ranging from machine learning and data mining to bioinformatics, high-energy physics, statistics, image analysis and more.

Clustering is a particular type of classification. As shown in Figure 1.1, classification techniques can be divided into several classes [2]:

- exclusive or non-exclusive: in an exclusive classification, each object belongs to one and only one class, meaning that the classes are non-overlapping.
- intrinsic or extrinsic: in intrinsic classifications, also called *unsupervised learning*, a proximity matrix is used to group the data. Extrinsic classifications on the other hand also use labels as well as the proximity matrix.
- hierarchical, partitional and density-based: hierarchical clustering algorithms build a hierarchy of clusters by iteratively merging or splitting them based on their similarity; partitional algorithms divide the data points in a pre-defined number of partitions in order to optimize an objective function, like minimizing the intra-cluster distance or maximizing the inter-cluster distance; finally, density-based algorithms construct the clsuters by identifying the regions where the density of points is higher.

1.1 Density-based algorithms

Density-based clustering algorithms assume that the points belonging to each cluster are drawn from a specific probability distribution, and the entire dataset is the combination of such distributions. The aim of this class of algorithms is to identify the distribution parameters for each of them.

The idea is to grow a given cluster as long as the density in the neighborhood exceeds some threshold. This means that in order to have a cluster, a neighborhood of a given radius has to contain at least a minimum number of points.

Often with these methods the point distributions are assumed to be multivariate gaussians for numeric data and multi-nominals for nominal data. A common solution is to

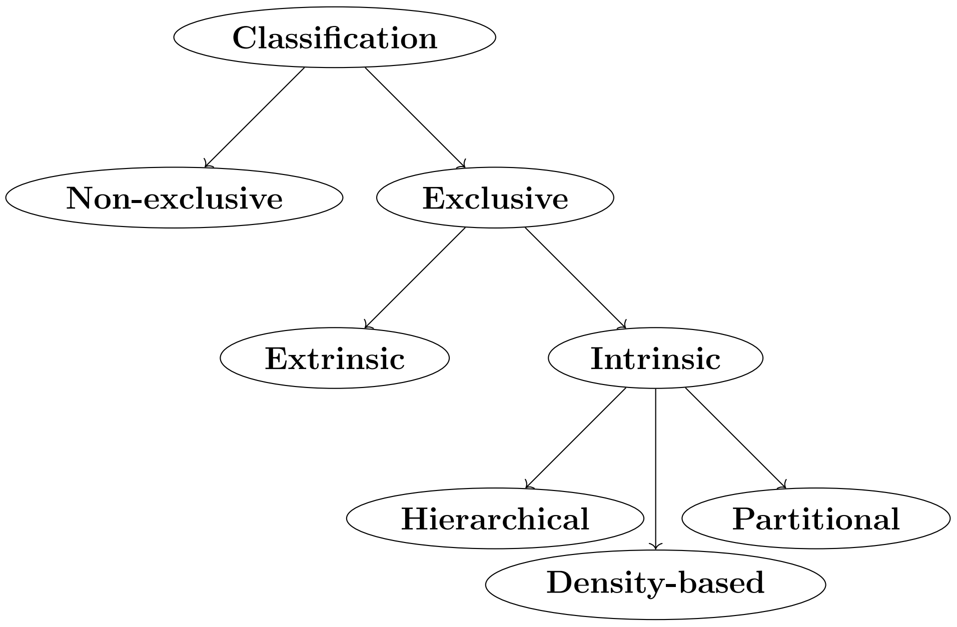


Figure 1.1: Tree illustrating the different types of techniques for classification.

use the maximum likelihood principle [1] to find the clustering structures and parameters that maximize the probability of reproducing the given data.

1.1.1 The DBSCAN algorithm

DBSCAN [3] is one of the most well-known density-based clustering algorithms. It takes two parameters, `eps` and `minPts`, representing respectively the radius of a circle around a point where the neighbors will be searched, and the minimum number of points that a cluster can contain.

The algorithm iterates over a set of points and for each point calls the `ExpandCluster` function. This function queries the surroundings of the point to find its neighbors and checks the number of points inside this region: if it is smaller than `minPts`, then the point is marked as noise and the algorithm moves on to the next, whereas if it's larger than `minPts`, then the cluster index in increased, and the neighbors of the point, as well as the points in their ε-neighbourhood are checked and if they are unclassified or previously marked as noise their cluster index is set.

The region query can be implemented efficiently using an `R*-tree`[2] [3]. In a database of n points the height of said tree is in the worst case $O(\log(n))$. Since the ε-neighborhoods are expected to be small with respect to the size of the whole clustering space, the run-time complexity of a region query is $O(\log(n))$, and since there is one query region for each of the n points, the time complexity of the whole algorithm is $O(n \log(n))$, provided

[1]The *maximum likelihood* method is a technique used for parameter estimation of a probability distribution. It states that the best estimate for the true value of a set of parameters is that for which the probability of reproducing the given experimental data is the highest.

[2]An `R-tree` is a type of tree data structure used for indexing multi-dimensional information. The ordering of higher-dimensional data is done by placing the objects into *minimum bounding hyper-rectangles (MBR)*.

[3]An `R*-tree` is a variant of an R-tree which is specifically used for indexing spatial information.

that the `eps` parameter is chosen in a meaningful way, and the space complexity is $O(n)$.

1.2 Hierarchical clustering algorithms

Hierarchical clustering algorithms construct the clusters by recursively partitioning the data points in a top-down or bottom-up approach [4]. They can be divided in two groups:

- agglomerative hierarchical clustering: each point represents a cluster, and this clusters are progressively merged until the desired structure is obtained.
- divisive hierarchical clustering: all the points initially belong to the same cluster, which is then divided in sub-clusters recursively, and the process continues until the desired structure is obtained.

The merging or division of clusters is performed with some kind of metric representing the similarity of the points belonging to the same clusters. Depending on the method used to quantify such similarity, hierarchical clustering algorithms can be further divided into three classes:

- single-link clustering: the distance between two clusters is equal to the shortest distance from any member of one clusters to any member of the other. Thus, if each point has a quantity representing similarity, the similarity between two clusters is equal to the greatest similarity from any member of one cluster to any member of the other.
- complete-link clustering: the distance between two clusters is equal to the longest distance from any member of one cluster to any member of the other.
- average-link clustering: the distance between two clusters is equal to the average distance from any member of one cluster to any member of the other.

1.2.1 HDBSCAN

HDBSCAN is an extension of the DBSCAN algorithm that converts it into a hierarchical clustering algorithm. The main goals of HDBSCAN are to produce good clusters even in datasets with clusters of varying density, and to be robust to outliers, corrupt data and noise.

The first step of the algorithms consists of transforming the space by defining distance metrics that are robust to noise. As a first, inexpensive, estimate of density the distance to the k-th nearest neighbour is used, which can be easily calculated using the distance matrix or by querying data using `kd-trees` and computing it with the metric of choice. For a point x this is called the *core distance*, $d_{core}(x)$. Then, to spread apart points with low density, i.e., high core distance, a new metric is defined, called *mutual reachability distance* $d_{mreach-k}$

$$d_{mreach-k}(x, y) = \max\{d_{core}(x), d_{core}(y), d(x, y)\}$$

where $d(x, y)$ is the distance between points x and y in the original metric. In this way, if one of the points is in a sparse region its core distance is going to be higher than the distance from the other point, and the two points are not linked. This prevents outliers from getting included to a larger cluster, with the risk of acting as a bridge and merging two well-separated clusters.

Using this new metric, the algorithm can start linking the points in dense regions: it constructs a graph where the nodes are the points and the weight of the edges is the mutual reachability distance. Then, the edges with weight below a threshold value are dropped, and the threshold is progressively decreased, which results in disconnecting the graph into connected components. The problem with this method is that it's computationally expensive, scaling as $O(n^2)$ in time, where n is the number of points. The solution is to use algorithms for finding the *minimum-spanning tree* [4] of a graph, which can be computed efficiently using Prim's algorithm. With the minimum-spanning tree, the hierarchy of connected components can be computed, from which the flat clusters are obtained by a cut given by the *minimum cluster size* parameter.

1.3 Partitional algorithms

1.3.1 k-Means

K-means clustering is a very popular partitional clustering algorithm [5]. It takes the expected number of clusters as a parameter and iteratively partitions the dataset using the distance of each point from the centroid of each cluster as a metric.

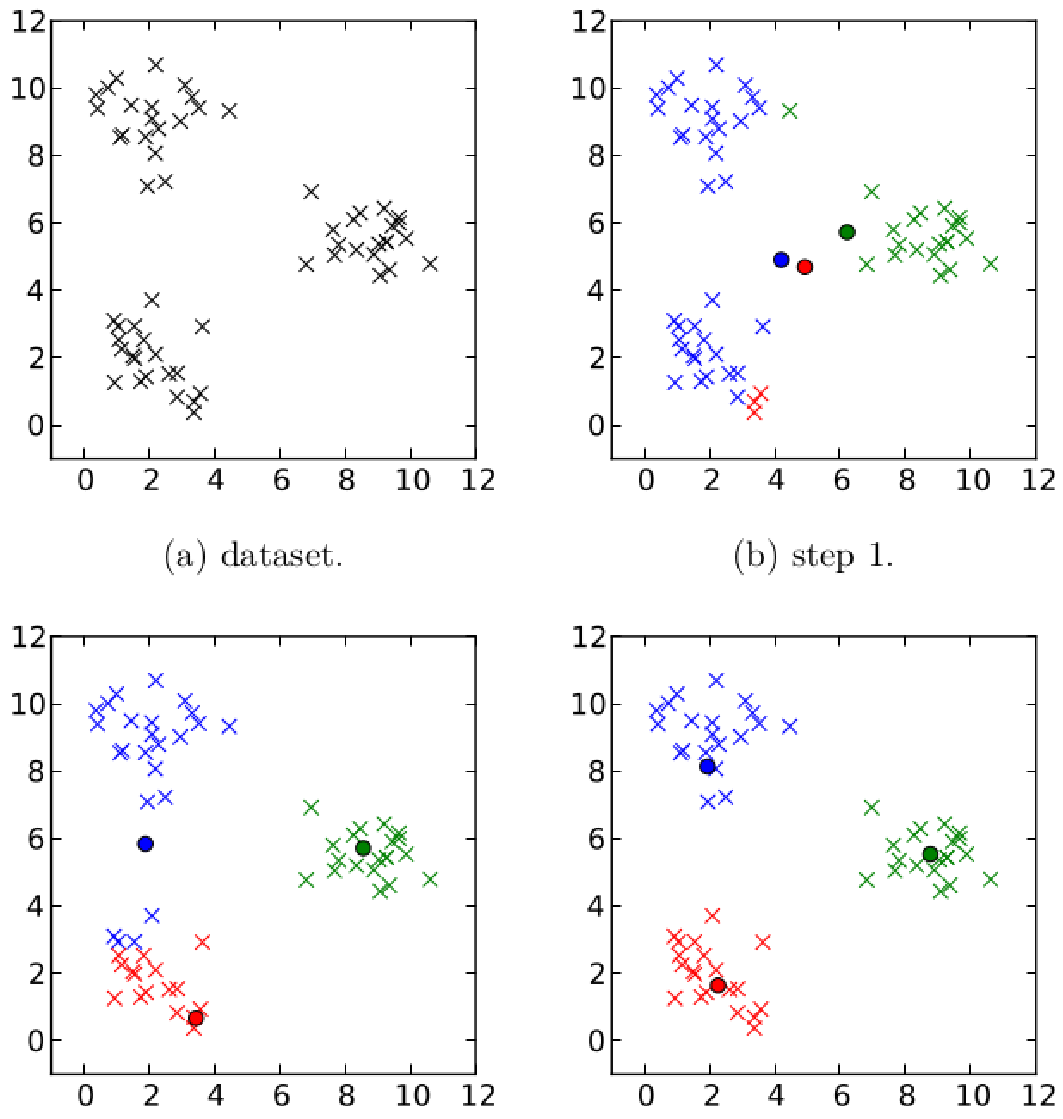
The algorithm, described in Algorithm 1, starts by taking the number of clusters k and choosing k points as initial centroids. Then, each point is assigned to the closest centroid. At this point, the centroids of each cluster are calculated again, and these steps are repeated until the convergence criterion is met. A typical convergence criterion is that at most 1% of the points change their cluster ownership. The steps of the algorithm are shown in Figure 1.2

Algorithm 1 Overview of the K-means algorithm

```
1. randomly choose k points as cluster centroids
while convergence criterion is not met do
    2. calculate the distance of each point to the centroids
    3. assign each point to the closest centroid
    4. re-calculate the centroid of each cluster
end while
```

The major factors that can impact the performance of the K-means algorithms are: choosing the initial centroids and estimating the number of clusters K. For choosing the initial centroids, the simplest and most widely used method in literature is that proposed by MacQueen, which consists of choosing the seeds at random.

[4]The minimum-spanning tree of a graph is a sequence of edges of a connected, weighted graph that connects all the vertices without any cycles and with the minimum possible total edge weight, meaning that the sum of the edges weights is as small as possible.



(a) dataset.

(b) step 1.

(c) step 2.

(d) step 3.

Figure 1.2: Illustration of the steps for the K-means clustering algorithm. First it loads the dataset (a) and generates K seeds and assigns each point to the nearest cluster (b). Then the last step is repeated iteratively (c) until the convergence criterion is met and the clustering results don't change noticeably (d).

1.4 Metrics for evaluating cluster quality

In order to assess the quality of the clusters reconstructed by an algorithm it's important to select the right metrics.

1.4.1 Homogeneity score

Homogeneity measures the similarity of the points inside a cluster [6]. Its value is defined between 0 and 1, where 1 indicates that all the points in the same cluster have the same label, and this is true for all the clusters.

The homogeneity score h is defined through *Shannon's entropy* H:

$$\begin{aligned} H(C|K) &= -\sum_{c,k} \frac{n_{ck}}{N} \log\left(\frac{n_{ck}}{n_k}\right) \\ H(C) &= -\sum_{c} \frac{\sum_k n_{ck}}{N} \log\left(\frac{\sum_k n_{ck}}{N}\right) \\ H(K) &= -\sum_{k} \frac{\sum_c n_{ck}}{N} \log\left(\frac{\sum_c n_{ck}}{N}\right) \end{aligned} \tag{1.1}$$

where C and K are the sets of truth labels and cluster labels, n_{ck} represents the number of points with label c assigned to cluster k, n_k the total number of points in the cluster k and N the total number of points in the dataset. The homogeneity score is then defined as:

$$h = 1 - \frac{H(C|K)}{H(C)} \tag{1.2}$$

An example of a perfectly homogeneous clustering can be seen in Figure 1.3: each of the clusters only contains points with the same label. However, this clustering is not perfect, since there are points with the same labels which are assigned to different clusters, which is not the expected result. This leads to the definition of another metric, the completeness score (Section 1.4.2).

1.4.2 Completeness score

Completeness measures how much similar points are matched to the same clusters [6]. Recalling the definition of Shannon entropy from Eq. 1.1, the completeness score is defined as:

$$c = 1 - \frac{H(K|C)}{H(K)} \tag{1.3}$$

A completeness score equal to 1 indicates that all the points with the same label get assigned to the same cluster. In Figure 1.4 can be seen an example of complete clustering: all the points with the same label are assigned to the same cluster. However the clusters are not homogeneous. This in addition to the considerations about the homogeneity score leads to the introduction of another metric which combines the two, the mutual information score (Section 1.4.3).

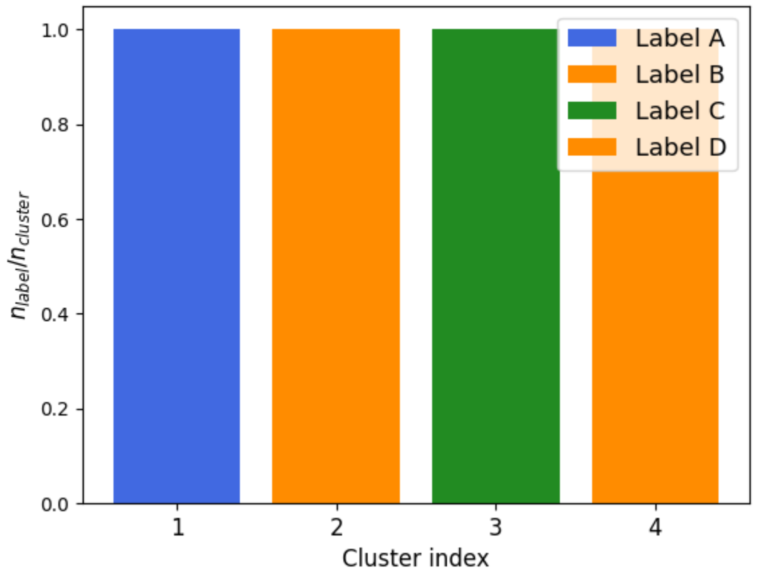


Figure 1.3: Example of perfectly homogeneous clustering. Note however how the clustering is not complete, since points with the same label can be found across different clusters.

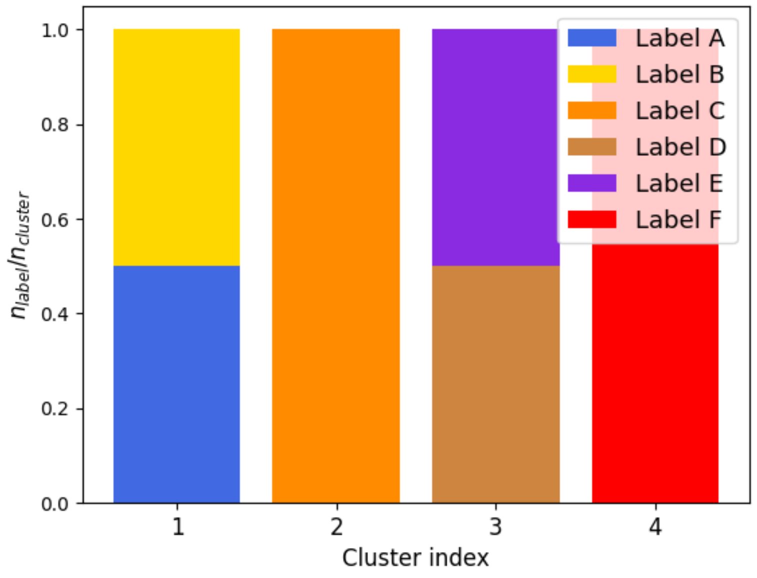


Figure 1.4: Example of perfectly complete clustering. Note however how the clustering is not homogeneous, since in the same cluster can be found points with different labels.

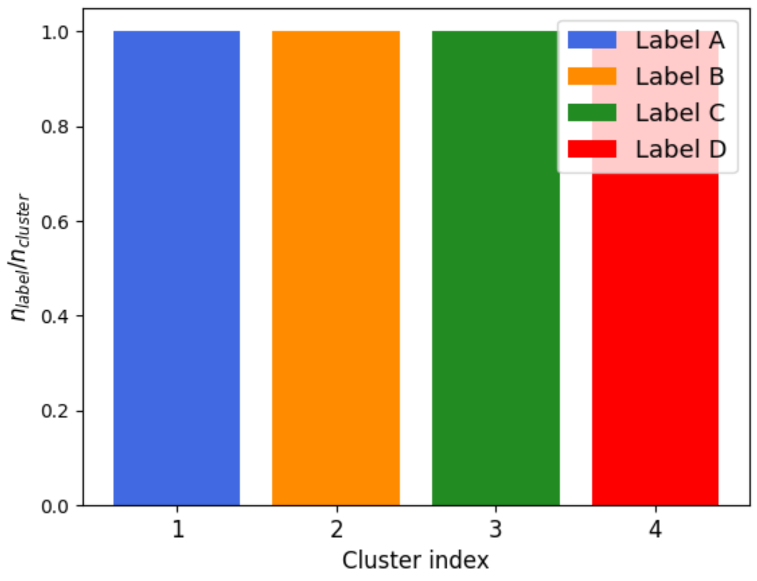


Figure 1.5: Example of clustering with both perfect homogeneity and completeness scores. The clusters only contain points with a single label and all the points with the same label are found in the same cluster.

1.4.3 Mutual information score

While both the homogeneity and the completeness scores give meaningful measures of the quality of a cluster analysis, the optimal solution outcome would be to have clusters which are both homogeneous and complete. To this scope the *Normalized Mutal Information score* [6] was introduced, which is defined as:

$$\text{NMI} = 2\frac{h \cdot c}{h + c} \tag{1.4}$$

where h is the homogeneity score and c is the completeness score. Figure 1.5 shows an example of a clustering that is both complete and perfectly homogeneous: all the points in a cluster have the same label and all the points with the same label are associated with the same cluster.

1.4.4 Silhouette method

The silhouette method [7] provides a numeric value indicating how similar a data point is to the other points in the same cluster compared to the other clusters.

Assuming that the data points have been divided into k clusters let

$$a(i) = \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} d(i, j) \tag{1.5}$$

be a variable representing the mean distance of point i from all the other points in the same cluster, and let

$$b(i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j) \tag{1.6}$$

be the minimum mean distance of point i from all the points in the other clusters. A small value of $a(i)$ indicates that the point i is well matched to the cluster C_I, whereas a high value of $b(i)$ indicates that it would be a better match for the neighbouring clusters.

The silhouette value $s(i)$ is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{1.7}$$

for clusters with $|C_I| > 1$ and $s(i) = 0$ for $|C_I| = 1$. Thus, the silhouette value is defined in the $[-1, 1]$ range. A value $s(i)$ equal to 1 indicates that the point is appropriately clustered. In contrast, a value equal to -1 indicates that the point would be a better match for the neighbouring clusters, and a value of 0 suggests that the point is in the middle of two clusters. The mean value of s for all the points of a cluster indicates how packed together they are, and the mean over the entire dataset indicates how well the data was clustered. It is also important to analyze the single s values for each cluster, because if the number of clusters is incorrect, some of them could have a noticeably lower silhouette value than the rest. This is possible especially for algorithms such as k-means.

In Figure 1.6 is shown an example of the silhouette method applied to a simple dataset containing four blobs.

1.4.5 Dunn index

The Dunn index is another metric for evaluating the results of a cluster analysis based on the structure of the dataset. Its goal is to identify clusters that are both compact, i.e. with a small variance in the distribution of points, and well separated, meaning that the distance between two clusters is much larger than their variance.

The Dunn index is defined as

$$D_m = \frac{\min\limits_{1 \leq i \leq j \leq m} \delta(C_i, C_j)}{\max\limits_{1 \leq k \leq m} \Delta_k} \tag{1.8}$$

where $\delta(C_i, C_j)$ is the distance between clusters i and j and Δ_k is the cluster diameter, calculated as the maximum within cluster distance:

$$\Delta_i^{within} = \max_{x,y \in C_i} d(x, y) \tag{1.9}$$

In alternative to the maximum within cluster distance, which is the method for calculating the diameter of a cluster originally proposed by Dunn, it can also be computed as the mean distance between all the pairs inside the clusters or the mean distance between all the points and the cluster centroid:

$$\Delta_i^{pairs} = \frac{2}{|C_i|(|C_i| - 1)} \sum_{x,y \in C_i, x \neq y} d(x, y) \tag{1.10}$$

$$\Delta_i^{centroid} = \frac{1}{|C_i|} \sum_{x \in C_i} d(x, \mu) \tag{1.11}$$

where μ is the cluster centroid.

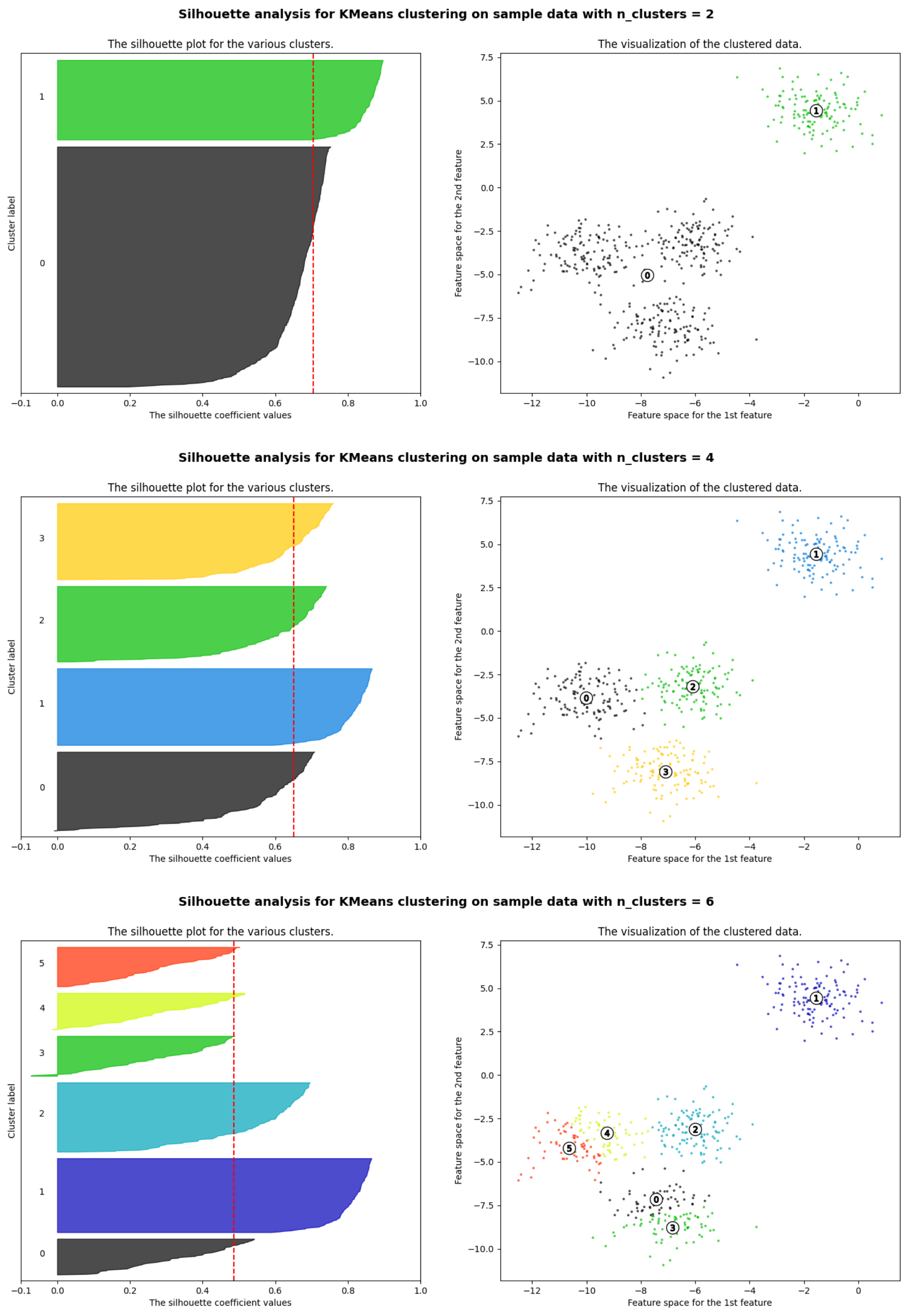


Figure 1.6: Illustration of the silhouette method for a simple dataset containing four blobs.

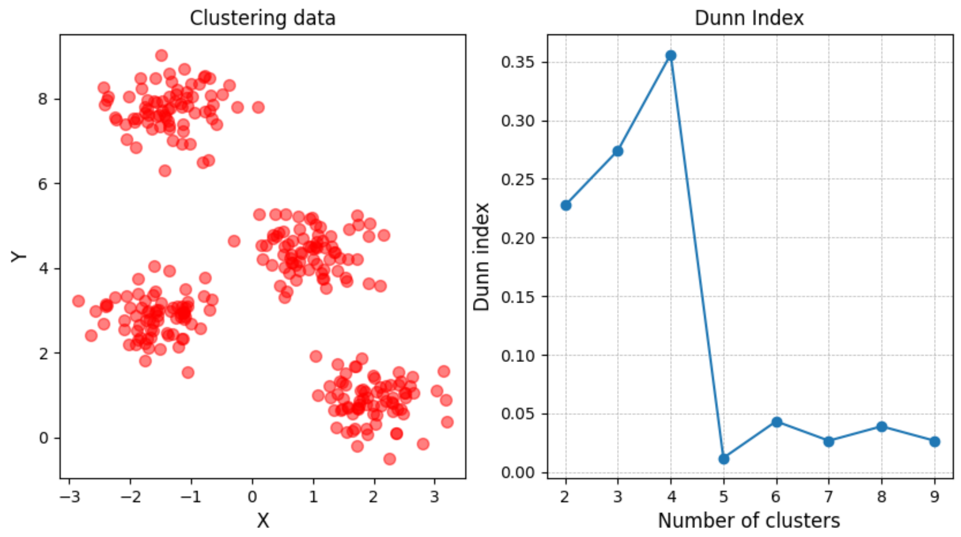


Figure 1.7: Dunn index for a dataset containing 4 clusters, clustered using K-means.

One of the drawbacks of the Dunn index with respect to the Silhouette method is its computational cost. Figure 1.7 shows the values of the Dunn index for a dataset of four blobs clustered with k-Means, showing a clear global maximum for $k = 4$, indicating that the parameter produces the most compact and well-separated clusters.

1.4.6 Davies-Bouldin index

The Davies-Boulding index is another popular metric for evaluating cluster quality. It measures how well-separated and compact the clusters are.

$$S_i = \left(\frac{1}{|C_i|} \sum_{x \in C_i} d(x, \mu)^q \right)^{\frac{1}{q}} \tag{1.12}$$

Let R_{ij} be a measure of how well the data was clustered, defined as

$$R_{ij} = \frac{S_i + S_j}{M_{ij}} \tag{1.13}$$

so the Davies-Bouldin index is defined as:

$$\text{DB} = \frac{1}{N} \sum_i \max_{j \neq i} R_{ij} \tag{1.14}$$

This index can be used by sampling its value for varying values of the clustering algorithm's parameters, and the parameter which results in the smallest index value indicates the best cluster. The main limitation of the Davies-Bouldin index are that it is sensitive to outliers, because they can significantly impact the average distances.

Figure 1.8 shows an example of use of the Davies-Bouldin index with the k-Means algorithm: the trend shows a clear minimum for $k = 4$, which indicates that that parameter produces the most compact and well separated clusters.

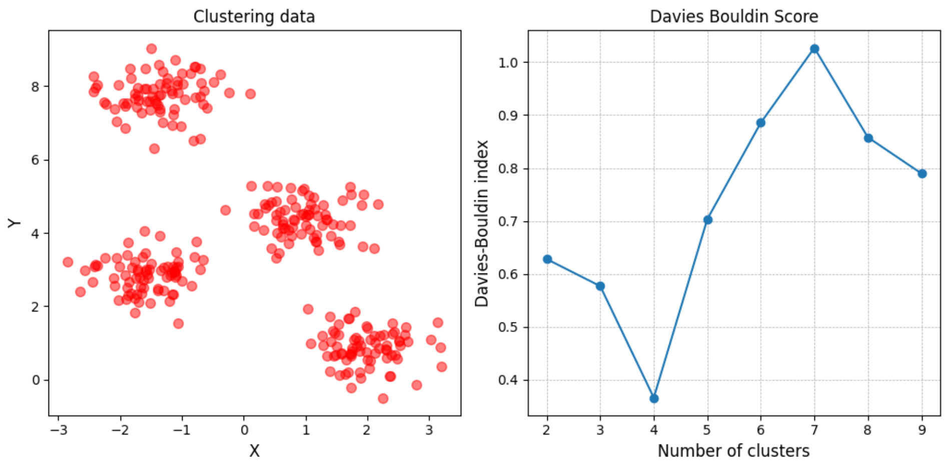


Figure 1.8: Davies-Bouldin index for a dataset containing 4 clusters, clustered using K-means. The values of the index have a clear minimum for value 4, indicating that it's the correct number of clusters.

Chapter 2

Parallel and Heterogeneous computing

2.1 Notions in computer architectures

The most basic computer architecture was presented by John Von Neumann et al. in 1945 and is known as *Von Neumann architecture*. In this design, illustrated in Figure 2.1, a computer is composed by:

- A processing unit, or processor, containing an arithmetic logic unit (ALU) and registers.
- A control unit, containing an instruction register and a program counter.
- Memory to store data and instructions.
- Devices for handling input and output.

In Von Neumann's architectures, operations are carried out following the *Fetch-Decode-Execute* cycle, which is shown in Figure 2.2. This also constitutes the main limitation of this architecture, which is usually referred to as *Von Neumann's bottleneck*: fetching instructions and executing operations on data can never be done at the same time, because they share the same channel (bus).

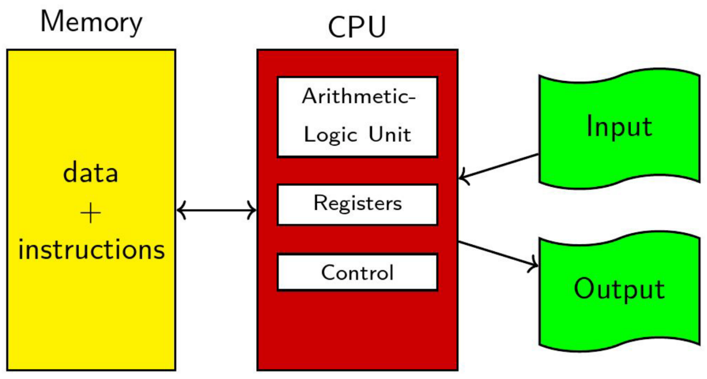


Figure 2.1: Illustration of Von Neumann's architecture [8].

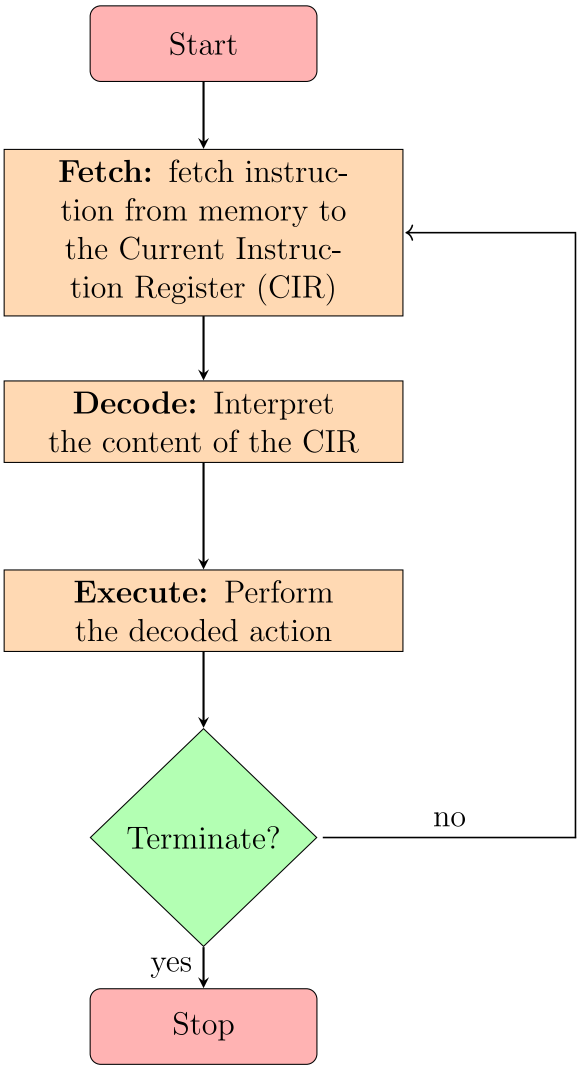

Start
Fetch: fetch instruction from memory to the Current Instruction Register (CIR)
Decode: Interpret the content of the CIR
Execute: Perform the decoded action
Terminate?
no
yes
Stop

Figure 2.2: Von Neumann architecture's fetch-decode-execute cycle.

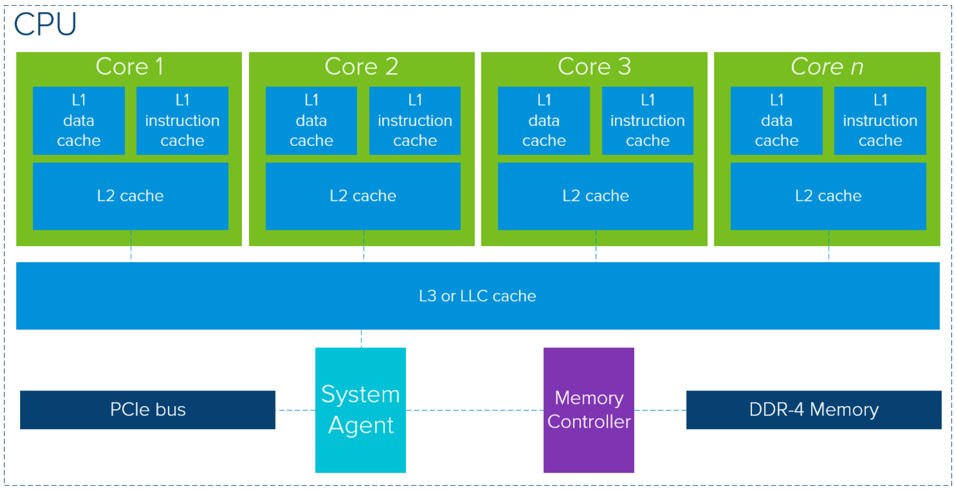


Figure 2.3: Illustration of a CPU architecture. Notice the different sizes of the L1, L2 and L3 cache, as well as the number and sizes of the cores

A typical server CPU (Figure 2.3) is made up by several cores, each contaning a control unit, an arithmetic logic unit (ALU) and a small amount of memory called cache. This is connected to a large amount of dynamic random access memory, called *DRAM*. The cache is static memory, which makes it much faster than DRAM, but it also makes it much more expensive, which means that the amount that can be mounted on a CPU is limited. In order to balance the speed of memory and its cost, multiple cache levels are used, which are increasing in size and getting more distant from the core and slower:

- L1 cache, is the fastest memory, second only to the CPU registers, is divided in *L1-d* (data) and *L1-i* (instruction) and usually is 64 kB in size for each core.
- L2 cache, which is a bit slower than L1 and is usually between 256 kB to 32 MB in size for each core.
- L3 cache, which unlike L1 and L2 is shared between all the cores in the CPU, and is the largest of the three, being between 32 MB to 96 MB.

Caches are useful for speeding up the access of elements during the execution of a program.
When a piece of code is executed and it tries to read some data, for example an element of an array, the CPU first looks for it in the L1 cache. If it does find it, that is called a *cache hit*, and the data is thus accessed immediately, with a delay of roughly 1 ns. If on the other hand the data is not found in L1, the processor looks for it first in the L2 cache, then in L3 and finally in the DRAM (access time of about 100 ns). Once the data is found in the DRAM, a buffer of data is copied to the cache in blocks of 64 B (for modern CPUs), which are called *cache lines*. This is an optimization which is based on two common assumptions which tend to be true:

1. if an element in memory has been accessed, it is likely that is will be accessed again soon
2. if an element in memory has been accessed, it is likely that the element adjacent to it will be accessed next

These are called the principles of *time locality* and *spatial locality*.

GPUs, *Graphical Processing Units*, have an architecture (shown in Figure 2.4) that shares many elements with that of a CPU, but presents several key differences which make it particularly fit for heavy parallelization tasks. As shown in Figure 2.5, a GPU is made up by arrays of *streaming processor cores*, organized in *streaming multiprocessors* [9]. Each streaming processor core is highly multithreaded, managing dozens of concurrent threads. Each streaming multiprocessor is composed of several SP cores, *special function units (SFU)*, instruction and constant L1 caches, a multithreaded instruction unit and a block of memory called *shared memory*. This general processor array architecture is scalable in order to get smaller or larger GPU configurations by simply scaling the number of multiprocessors and the number of memory partitions.

Whereas CPUs can assume high cache hit rates, higher than 99.9%, the hit rates for GPUs are closer to 90%, and that high number of losses must be dealt with. Furthermore, while a CPU can easily halt in the rare occurence of a cache miss, a GPU needs to work

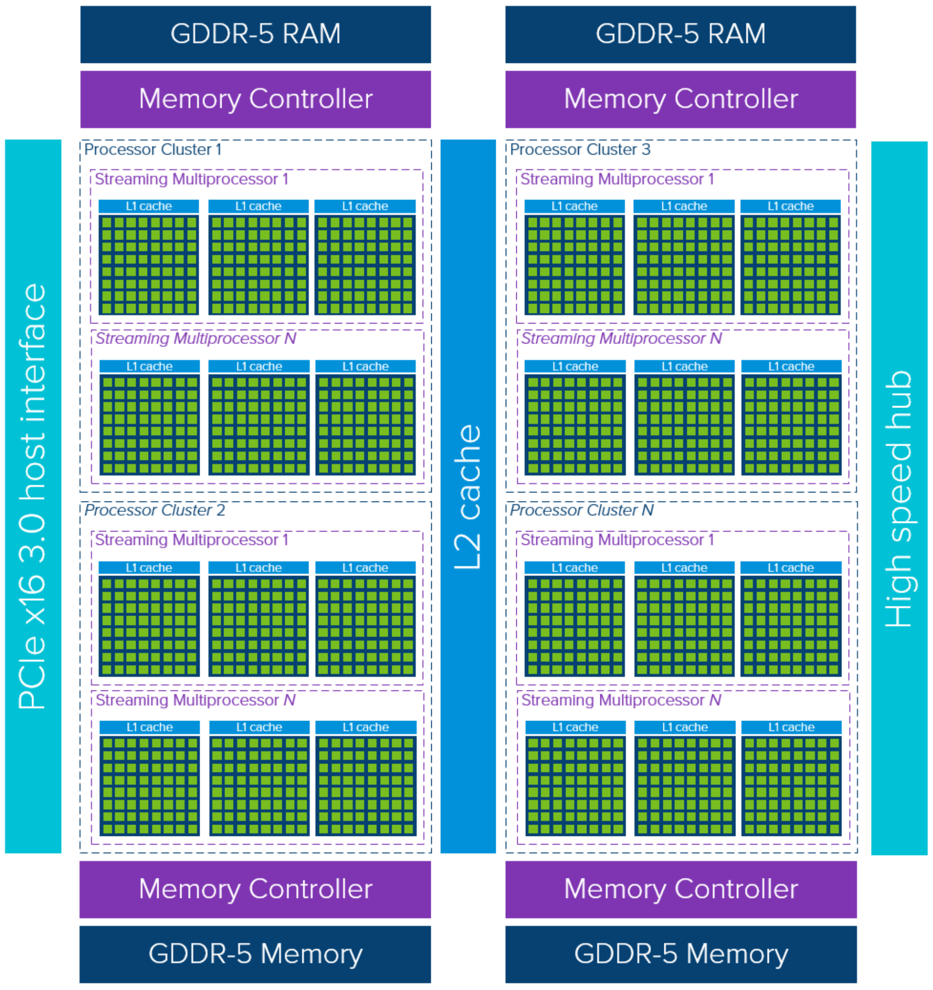


Figure 2.4: Illustration of a GPU architecture. Notice how the cores are grouped inside streaming multiprocessor, and how each core has a much higher number of processing units, which is the reason behind their better performance for executing parallel algorithms.

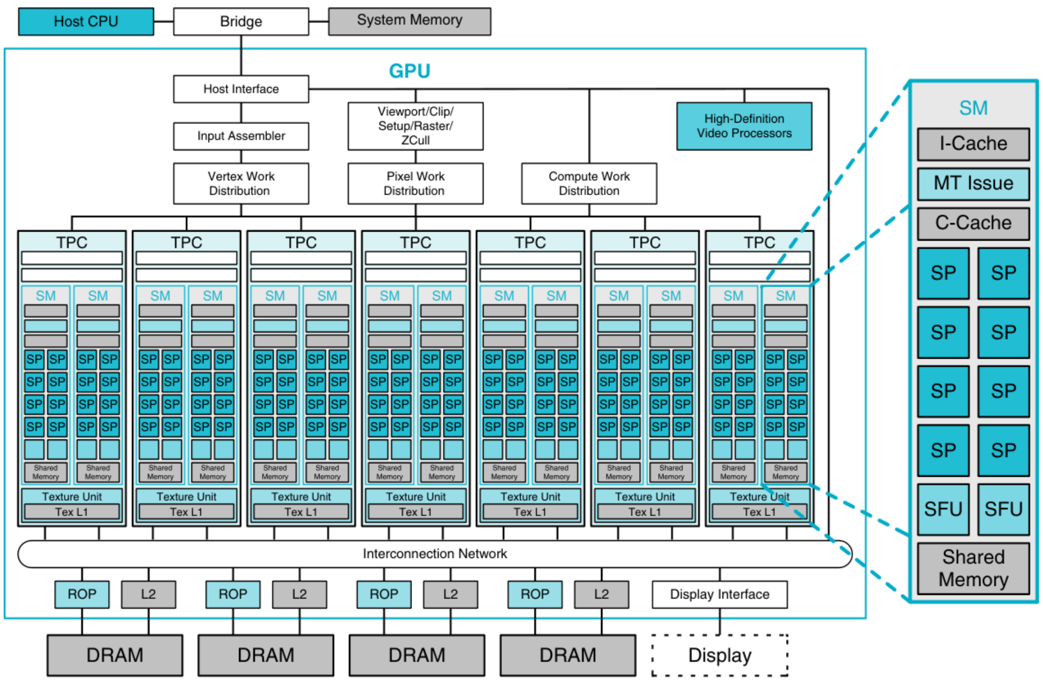


Figure 2.5: A more in depth representation of a modern GPU architecture [9].

with a mixture of hits and misses, requiring what is called a *streaming cache architecture*. The global memory is stored in external DRAM, meaning that it is not local to any one of the streaming multiprocessors because it is meant for communication among different thread blocks in different grids.

The whole device is made up by processor clusters, each containing streaming multiprocessors, SM. Each SM may contain up to 8 thread blocks, which then can contain up to 1024 cores. Each core has an L1 cache, the L2 cache is shared by all the streaming multiprocessors, there is an external random access memory called global memory and each tread block has another small amount of memory called *shared memory*. Shared memory is an extremely important optimization because it allows the threads in a block to communicate and share data, and is much faster than global memory.

2.2 Introduction to parallel computing

We live in a world where the amount of data produced in any field of study is increasing at an astounding rate. As the size of data to be processed increases, the computing power of the machines is also expected to increase, at a similar or possibly higher rate.

However, reality is the opposite. The increase in computing power of modern CPUs is increasing much slower than would be desirable. If the demand for higher performance cannot be answered with faster processors, then the solution is to employ high-performance computing techniques, like re-thinking algorithm in a parallel paradigm. For a generic algorithm, if t_s is the serial execution time, f is the fraction that can be parallelized and p the number of cuncurrent threads, the parallel execution time t_p is given by Amdahl's law (Figure 2.6):

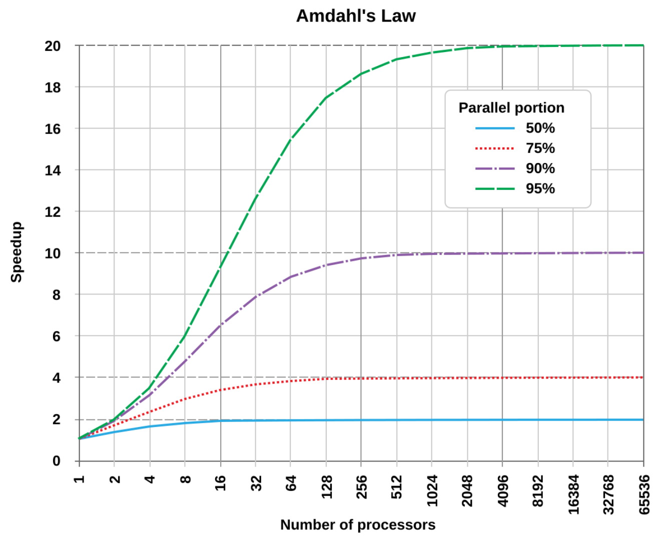


Figure 2.6: Ahmdal's law [10].

$$t_p = ft_s + (1-f)\frac{t_s}{p} \tag{2.1}$$

From this it follows that the speed-up of a parallel algorithm with respect to the serial version is:

$$S(p) = \frac{t_s}{t_p} = \frac{t_s}{ft_s + (1-f)\dfrac{t_s}{p}} \tag{2.2}$$

where the upper limit is:

$$S_{max} = \lim_{p\to\infty} S(p) = \frac{1}{f} \tag{2.3}$$

It is clear that in order to obtain the highest possible performance boost from parallel computing the serial part of the whole execution should be as small as possible. This is not trivial though, and is the true difficulty behind parallel computing, because it is not enough to re-write algorithms in a parallel way, but often it is necessary to completely re-think them and write them from scratch in parallel.

2.2.1 Parallel computing

A computer program is composed of a series of *tasks*, and a task is defined as a sequence of instructions. *Parallel computing* is a computation paradigm where many operations are carried out simultenously in order to reduce execution times and improve performance. Although parallel computation is similar to concurrency, the two are actually very different: when multiple tasks are running concurrently it means that they are all active and scheduled for execution; they are logically parallel in the sense that, they can be executed at the same time without changing the final outcome. On the other hand, parallelism is a condition where a multitude of task are actually being executed at the same time.

The most fundamental concept in parallel computing is that of a *thread*. A thread is an execution context, in particular a stack and a set of registers. In other words, a thread is a small set of instructions to be executed as part of a larger process. In parallel computing many threads are defined and to each is assigned a different portion of data, so that the entire execution can be divided into many small pieces that can run and progress in parallel (an illustration of the sheduling of tasks to a multitude of processing units is shown in Figure 2.7). The most basic example from this is the sum of two vectors: when running code serially (in other words, with a single thread), each pair of elements is summed sequentially, whereas in parallel execution there is a multitude of threads, where each one sums a portion of the whole length of the vectors.

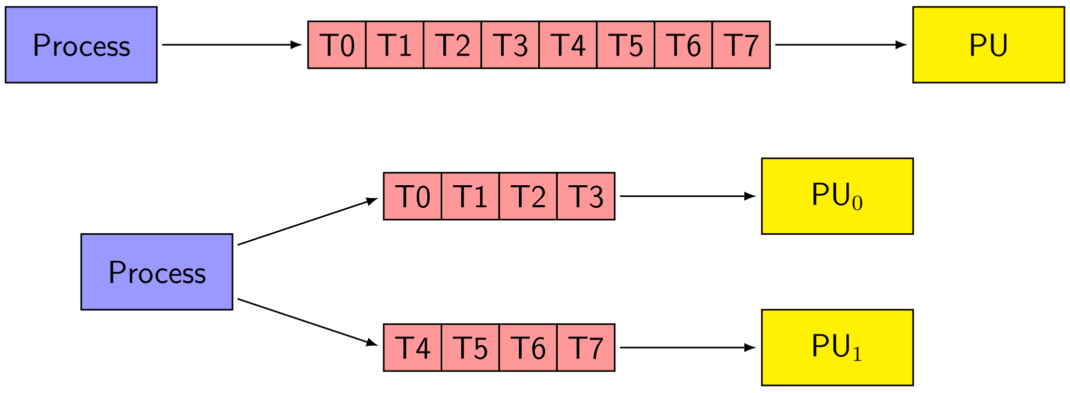


Figure 2.7: Comparison of the serial and parallel execution of a process.

When multiple tasks of the same program are running at the same time, the risk of errors naturally increases. In particular, two of the most typical errors in parallel programming are *data races* and *false sharing*.

Data race is a condition that occurs when more than one thread is trying to access to the same memory location at the same time without any type of synchronization. This type of error generally results in undefined behaviour, because there is no way of knowing a-priori which one of the threads is going to access the resource first, so the final result is impossible to predict. What makes this type of error even more dangerous is that often the program would not crash and would terminate with no apparent error, which makes finding the error much more difficult. Race conditions are usually handled using *atomic operations*. These types of operations don't have an intermediate state (that's why they are called atomic) and ensure that only one thread at a time can modify a valiable by syncrhonizing the threads the memory accesses of the different threads.

False sharing is a conditions that occurs when the cache lines of two threads are partially overlapping. When one of the two threads modifies one of the elements in its cache line it invalidates the cache line of the other thread, because now it contains an old data value, so the processor needs to update the data in global memory and reload the cache line of the second thread. This type of error can be prevented by designing data structures so that the data buffers allocate memory in multiples sizes of cache lines, which is known as *padding*. For this to be effective, software needs to be designed in a cache-friendly way: data should be aligned in such a way that the elements accessed by

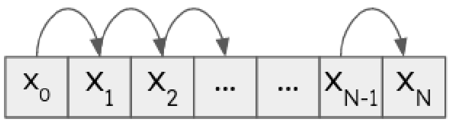


Figure 2.9: Sequential memory access pattern.

the same thread are adjacent in memory (Figure. 2.8). This is known as a *sequential memory access pattern* (Figure. 2.9), and is the best access pattern when programming in parallel on a CPU.

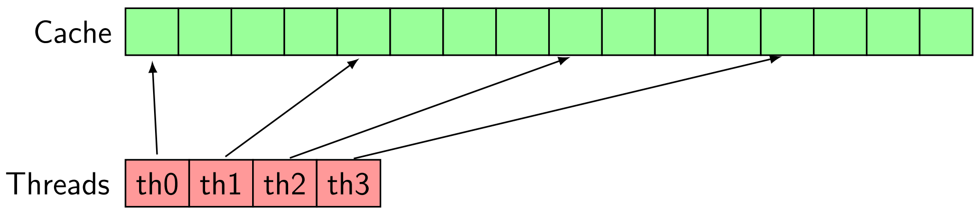


Figure 2.8: Alignment of data inside a cache line for efficient CPU parallel execution.

2.2.2 GPU computing

Parallel computing is a fundamental tool in modern software development, and as the number of cores in modern processors increases its effect gets increasingly significat. However, as explained in section 2.1, GPU's architecture makes them much more indicated for parallel programming, because the number of threads that can be handled by their streaming multiprocessors is much higher than that of even the best modern CPUs. For this reason investing in GPUs and in developing software that runs on them is crucial for keeping up with today's world increasing need for computational power.

Developing software on GPUs is similar to working with two different machines, one local and one remote. The reason for this is that the CPU and the GPU are two completely different devices, each with its own processor, global memory and caches.

2.3 The need for Software and Performance portability

As said in the previous sections, parallel computing is a very active field of research today, with many applications in a wide range of branches of science, and taking advantage of the very efficient massively-parallel accelerators available nowadays is crucial for every scientific application. However, the problem is that today there are many different types of accelerators available, and each of them comes with its specific tool for developing software on it: Nvidia GPUs have the CUDA C++ extension, AMD GPUs have the HIP/ROCm platform, Intel GPUs have the OneAPI programming model and so on. Writing general software that can run on all of the types of accelerators available would require to maintain a large number of codebases, and this is simply not possible when the scale of the software framework gets very large.

Performance portability of software represents the ability of a program to operate efficienctly accross different platforms, processors types and architectures.

In order to mitigate this problem, there are a number of performance portability libraries currently under development, like SYCL [11], Kokkos [12] and Alpaka [13]. The goal of these libraries is to only write the code one time and be able to compile it for every possible accelerator while maintaining performance as close to the native ones as possible.

2.3.1 The Alpaka library

The Alpaka library is a header-only abstraction library for accelerator development. Its aim is to provide performance portability across accelerators through the abstraction of the underlying levels of parallelism.

The goal of Alpaka is to write a single source code which can then be compiled and run on single or multiple different backends, without having to re-write any part of the code and, in particular, without having to develop, test and maintaing many different versions of the same software. Alpaka achieves this level of parallelism using preprocessor symbols to enable different backends, as well as general host-side and device-side APIs. It also relies heavily on the use of C++ templates and type traits, in particular their use for defining constraints on the types of the parameters taken by the API functions and thus choosing the correct function specialization for the specific backend.

One of the strengths of Alpaka's model is that it separates the algorithms from their parallelization strategies. The algorithms are defined as function objects called *kernels*, whereas the parallelization strategy is defined by the *accelerator* type and the mapping of threads and blocks, called *work division*.

The syntax of Alpaka is very similar to that of native CUDA, which is a great advantage because it makes it much easier to integrate it into already parallel software workflows implemented in CUDA.

In Alpaka, host and device memory is managed with *shared-pointers-like* objects, which makes memory handling safer and with almost no risk of memory leaks. There are two types of memory wrappers in Alpaka: *buffers*, which own the memory that they point to, and *views*, which are non-owning and serve as a cheap way of accessing the memory owned by another container. In addition to making the use of memory easier and safer, buffers have the further advantage of handling the use of pinned memory over pageable memory [1] when possible, thus speeding up data transfers between the host and the accelerator.

```cpp
#include <alpaka/alpaka.hpp>

template <typename TAccTag>
void main(const TAccTag&)
    using Dim = alpaka::DimInt<1u>;
    using Idx = std::size_t;
    using Acc = alpaka::TagToAcc<TAccTag, Dim, Idx>;
```

[1]Data allocated in a *pageable* memory region can be *swapped* to the storage in the case of limited available space in memory. When data contained in such a memory region needs to be copied to the device (GPU), it first needs to be copied to a temporary *pinned* data region, which can't be swapped.

```

    auto const platform = alpaka::Platform<Acc>{};
    auto const devAcc = alpaka::getDevByIdx(platform, 0);

    QueueAcc queue(devAcc);

    Idx const numElements(1 << 10);
    Idx const elementsPerThread(8u);
    alpaka::Vec<Dim, Idx> const extent(numElements);

    auto const platformHost = alpaka::PlatformCpu{};
    auto const devHost = alpaka::getDevByIdx(platformHost, 0);

    using BufHost = alpaka::Buf<DevHost, Data, Dim, Idx>;
    BufHost bufHost(alpaka::allocBuf<Data, Idx>(devHost, extent));

    /* initialize data inside buffer */

    using BufAcc = alpaka::Buf<DevAcc, Data, Dim, Idx>;
    BufAcc bufAcc(alpaka::allocBuf<Data, Idx>(devAcc, extent));

    alpaka::memcpy(queue, bufAcc, bufHost);

    alpaka::KernelCfg<Acc> const kernelCfg = {extent, elementsPerThread};

    auto const workDiv = alpaka::getValidWorkDiv(
        kernelCfg,
        devAcc,
        kernel,
        /* kernel arguments */);

    auto const taskKernel = alpaka::createTaskKernel<Acc>(
        workDiv,
        kernel,
        /* kernel arguments */);

    alpaka::enqueue(queue, taskKernel);
    alpaka::wait(queue);

    alpaka::memcpy(queue, bufHost, bufAcc);
    alpaka::wait(queue);
}
```

Listing 2.1: Generic Alpaka application

In Alpaka, kernels are defined as function objects [2] and have several requirements for the definition of the `operator()`:

- it must be templated on the accelerator type
- the accelerator must be passed as the first parameter by const-reference
- it must be const, so no parameters of the `struct` can be changed
- it must the marked with the `ALPAKA_FN_ACC` macro, to indicate that it will be executed on the accelerator

2.3.1.1 One more dimension: elements

When writing generic code that can be executed both in serial and in parallel, there is one factor that must be taken into account: the number of threads in each block and thus

[2] A function object is an instance of a class or structure which can be called as a function. In C++ function objects can be implemented through a `struct` and by providing an overload for the call operator, `operator()`.

the number of operations done by each thread highly depends on the type of accelerator.

For a CPU, the most optimal thing is that each block contains just one thread and that thread will have to do more operations[3]. On the other hand, in a GPU the most optimal thing is to have many threads in each block, with each thread doing just one operation. This difference makes it very hard to write generic code. Take for example the addition of two vectors in serial C++ and parallel CUDA: Writing such a function

```cpp
void addition(const float* a,
              const float* b,
              float* c,
              size_t n) {
    for (size_t i{}; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

```cpp
__global__ void addition(const float* a,
                         const float* b,
                         float* c,
                         size_t n) {
    auto index =
        threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n) {
        c[index] = a[index] + b[index];
    }
}
```

Figure 2.10: Comparison of serial and parallel implementation of vector addition. Note how in the serial version a single thread is doing all the work sequentially. However in the parallel version, the threads in a block are being executed in parallel [4], and each one is responsible for just one addition.

in a generic way would certainly lead to incredibly verbose code [5].

In order to address this problem, Alpaka introduced a new level of task abstraction: *elements*.

Elements represent operations which need to be performed in order to completely execute an algorithm. As an example, Listing 2.2 shows the alpaka implementation of vector addition. First the index of each thread is defined, and then in line 11 the number of elements for each thread is obtained for the specific backend from the defined work division: if the backend is a CPU and the code is being run in serial, the code is going to be run by a single thread executing the operations sequentially, whereas in multithreading each thread is going to operate on adjacent elements, and finally on a GPU adjacent threads are operating on adjacent elements, and eacth thread will operate on elements separated by a stride equal to the number of threads (the *block size*).

```cpp
 1 struct KernelVecAdd {
 2     template <typename TAcc, typename TIdx>
 3     ALPAKA_FN_ACC void operator()(const TAcc& acc,
 4                                   const float* a,
 5                                   const float* b,
 6                                   float* c,
 7                                   size_t n) const {
 8         // define the thread index for each thread
 9         const TIdx gridThreadIdx(alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u]);
10         // calculate the number of elements for each thread
11         const TIdx threadElemExtent(alpaka::getWorkDiv<alpaka::Thread, alpaka::Elems>(acc)[0u]);
12         // calculate the index of the first element for each thread
13         const TIdx threadFirstElemIdx(gridThreadIdx * threadElemExtent);
```

[3]A good choice would be to take 16 sequential elements, so that they can all be loaded in the same cache line. This is another reason why sequential memory access patterns are favoured for CPU computing.

[5]It could be done using C/C++ preprocessor, using macros defined during compilation to identify the compiler and select the correct implementation through a series of `#ifdef`. This would get very verbose very quickly, and would be unfeasible for large codebases.

```

        if(threadFirstElemIdx < numElements) {
            // Calculate the number of elements to compute in this thread.
            // The result is uniform for all but the last thread.
            const TIdx threadLastElemIdx{threadFirstElemIdx + threadElemExtent};
            const TIdx
                threadLastElemIdxClipped{(numElements > threadLastElemIdx) ? threadLastElemIdx :
     numElements};

            for(TIdx i{threadFirstElemIdx}; i < threadLastElemIdxClipped; ++i) {
                c[i] = a[i] + b[i];
            }
        }
    }
};
```

Listing 2.2: Alpaka kernel for vector addition.

2.3.2 Performance portability at the CMS experiment

The High-Luminosity upgrade of the LHC (HL-LHC) aims at increasing the luminosity [6] of the accelerator by a factor 10 with respect to the original design value, reaching an istantaneous luminosity of $7 \times 10^{34} cm^{-2} s^{-1}$ with an average pileup [7] of 200 proton-proton collisions. This increase in data will significantly increase the need for computational power both in the online as well as the offline software used in the high-energy physics experiments at CERN. In the last years all the main data centers around the world have been using heterogeneous computing platforms for improving the throughput and the energy efficiency of their workload. For this reason, the experiments at CERN have been improving their software frameworks for taking advantage of the diffusion of these new efficient computing platforms.

In particular, starting from the end of the 2010s, the CMS experiment [14] started investing considerable effort in the research and development of new techniques for reimplementing the algorithms and data structures of their software workflow. In a paper [15] published in 2020 they described the heterogeneous implementations of new algorithms for the reconstruction of particle tracks and vertices, that leveraging parallel accelerators allowed for more complex algorithms to be used, which resulted in better physics result, as well as an increase in throughput.

For the software of an experiment such as CMS, all the available computing platforms are needed, which makes efficient software performance portability a fundamental requirement. Several options were tested on a standalone version of the heterogeneous version of the pixel reconstruction [16]: `std::par`, OpenMP, Kokkos, SYCL and Alpaka. The throughput of the event processing was compared on NVIDIA and AMD GPUs as well as multithreaded CPUs, comparing the performance obtained through the performance portability option with the ones obtained with a native implementation for each

[6] *Luminosity* is the ratio of the number of events detected in a certain period of time divided by the cross-section σ of the process

$$L = \frac{1}{\sigma} \frac{\mathrm{d}N}{\mathrm{d}t}$$

The luminosity is one of the values that quantify the performance of a particle accelerator, because the higher its value is, the more data is available to analyze.

[7] When the luminosity is high, multiple collisions per beam-crossing occur, a phenomenon known as *pileup*.

platform. Among all the options considered, Alpaka proved itself to be the most stable, general and efficient, providing time and space performance close and in some cases better than the native implementations. This motivated the gradual porting of the CMS reconstruction software to Alpaka [17], to be used both for the offline reconstruction but also, most importantly, for the High Level Trigger to be used for the data acquisition in Run 3.

Chapter 3

The CLUEstering library

3.1 The CLUE algorithm

The CLUE algorithm [1] is a fast and fully parallelizable density-based clustering algorithm.
The algorithm was developed in the CMS experiment at CERN to be applied on high granularity calorimeters. Its main strength is that, unlike other clustering algorithms which have some serial steps which can't be accelerated with the use of parallel processors, CLUE is a fully parallel algorithm. This allows to take advantage of parallel accelerators like GPUs and greatly increment performance.

Another characteristic of CLUE is that it natively supports the use of different weights for all the points (which originally were the energies picked up by the calorimeters in the CMS detector), whereas other density-based algorithms require to manually define a metric that uses the weights when calculating the distance between the points. Instead, CLUE calculates the "energy-density" of each point, and uses it to construct clusters that originate from the points with the highest energy density and iteratively link the points in their surroundings.

The algorithm requires three parameters in input: a geometrical parameter δ_c, that represents the radius inside which the local density of each point is calculated, ρ_c, which represents the density threshold for a point to be a seed, which is the origin of each cluster, and finally δ_o determines the radius in which the points can be linked to form the same cluster.

3.1.1 Query over neighbors and calculation of local density

The clustering space is divided in a fixed grid $\mathcal{T}$ of rectangular bins, called tiles τ.

Query of points neighborhoods is the fundamental step in density-based clustering algorithms. In order to avoid looping over the entire dataset for each point, which would be extremely computationally expensive, the neighborhood of each point i, also called search box $S_{\delta_c}(i)$, is defined as the set of tiles τ touched by the square window of side δ_c around i:

$$S_{\delta_c}(i) = \{\tau \in \mathcal{T} \mid \tau \cap \mathcal{U}_{\delta_c}(i) \neq \varnothing\} \tag{3.1}$$

Then the set of points over which the query for point i takes place, $\Omega_{\delta_c}(i)$, is:

$$\Omega_{\delta_c}(i) = \{j \mid j \in S_b(j)\} \tag{3.2}$$

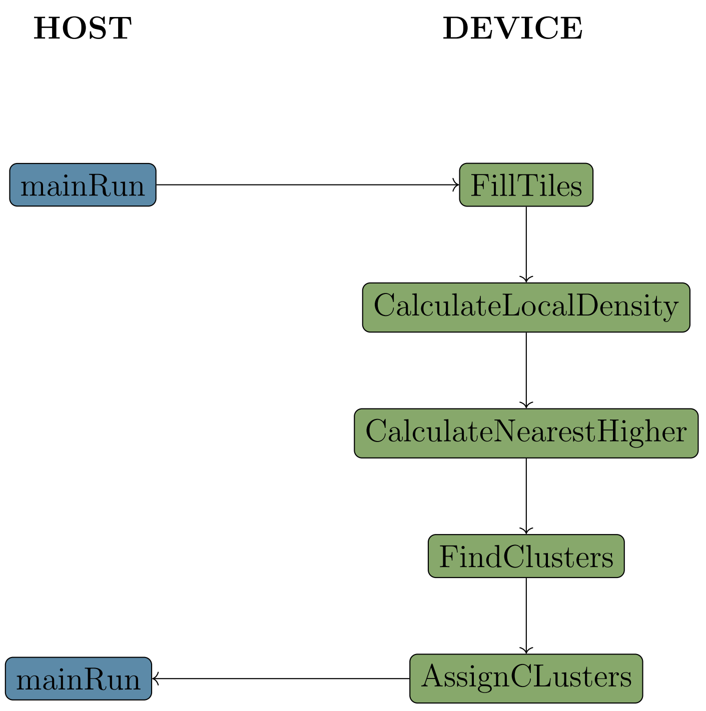


Figure 3.1: Illustration of the workflow of the CLUE algorithm. Notice how the entire algorithm is executed on the device, so that the only memory copies with the host are at the beginning and at the end of the execution, which results in much better performance.

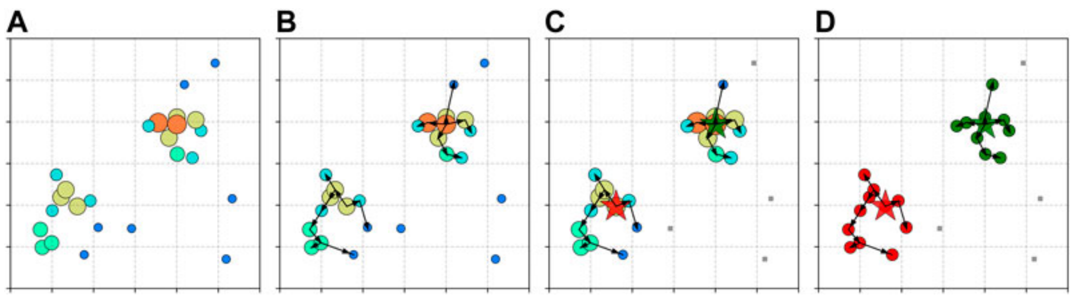


Figure 3.2: Steps of the CLUE algorithm. First, the density of each point is calculated and used to find the most energetic neighbor for each one. Then, the most energetic neighbors are marked as seeds and thus the clusters are constructed.

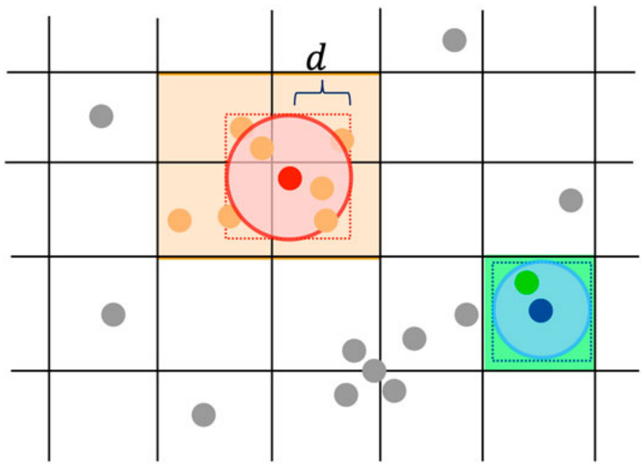


Figure 3.3: Illustration of the search-box used to speed-up the query of neighbors for each point.

This set is guaranteed to contain all the neighbors of the point i within a radius δ_c, $N_{\delta_c}(i)$:

$$N_{\delta_c}(i) = \{j \in \Omega_{\delta_c}(i) \mid d_{ij} < \delta_c\} \tag{3.3}$$

where it is clear that $N_{\delta_c}(i) \subseteq \Omega_{\delta_c}(i)$. In this way it's not necessary to loop over all points, and in particular if δ_c is small enough querying over the set $N_{\delta_c}(i)$ will have complexity $O(1)$.

The tiles serve the same purpose as the KD-trees and R-trees in other density-based algorithms. For the goal of this algorithm a fixed-grid data points indexing is the most optimal choice because both the setup and the query can be easily parallelized and the layout of the data is cache-friendly, which is crucial for the efficient parallel execution of the algorithm.

The scan over N_{δ_c} is used to calculate the local density of each point. This is done using a convolutional kernel $\chi(d_{ij})$ and taking into account the weight of each neighbouring point, w_j. The local density of a point i is then:

$$\rho_i = \sum_{j \in N_{\delta_c}(i)} \chi(d_{ij}) w_j \tag{3.4}$$

The default convolutional kernel is a flat kernel (Eq. 3.5) with parameter 0.5. The other default kernels implemented are Gaussian and exponential (Eq. 3.6).

$$\chi_{flat}(d_{ij}; m) = \begin{cases} m, & \text{if } d_{ij} <= \delta_c \\ 0, & \text{if } d_{ij} > \delta_c \end{cases} \tag{3.5}$$

$$\begin{aligned} \chi_{gaus}(d_{ij}; \mu, \sigma, A) &= A \exp\left(-\frac{(d_{ij} - \mu)^2}{2\sigma^2}\right) \\ \chi_{exp}(d_{ij}; \tau, A) &= A \exp\left(-\frac{d_{ij}}{\tau}\right) \end{aligned} \tag{3.6}$$

3.1.2 Calculation of the nearest highers

After calculating the density of each point, the algorithm needs to find the *nearest-higher*s, the most energetic neighbour of each point.

Let nh_i be the nearest-higher of point i and δ_i the distance between i and nh_i.

$$\text{nh}_i = \begin{cases} \min\limits_{j \in N_{\delta_o}(i)} d_{ij}, & \text{if } |N_{\delta_o}(i)| \neq 0 \\ -1, & \text{otherwise} \end{cases} \tag{3.7}$$

$$\delta_i = \begin{cases} d_{i,\text{nh}_i}, & \text{if } |N_{\delta_o}(i)| \neq 0 \\ +\infty, & \text{otherwise} \end{cases} \tag{3.8}$$

The algorithm loops over all the points in the search box $S_{\delta_o}(i)$, where the size of this search box will usually be equal or slightly larger than that of $S_{\delta_c}(i)$, and finds the nearest point j with $\rho_j > \rho_i$.

3.1.3 Finding and assigning clusters

The last two steps of the algorithm consist of, respectively, defining the clusters by finding their seeds and assigning each point to a cluster or marking it as an outlier.

It is important to underline the difference between seeds and outliers, since both are points with no nearest higher, meaning $\text{nh}_i = -1$ and $\delta_i = +\infty$. The difference between the two is that for seeds $\rho > \rho_c$ and for outliers $\rho < \rho_c$.

For each point the list of followers is defined as:

$$F_i = \{j \mid \text{nh}_j = i\} \tag{3.9}$$

and the cluster indices are passed from the seed to all the points in the cluster by iteratively going through their chains of followers iteratively.

3.2 Generalizing the algorithm

The original algorithm was developed for being used exclusively in the calorimeters of the CMS experiment, and was inapplicable in any other context, without modifying by hand a considerable amount of code and geometrical constants.
The specificity of the original algorithm laid in:

- the number of dimensions. The algorithm was implemented to reconstruct exclusively bidimensional clusters of each layer of the detector.
- the geometrical constants. All throughout the code were hardcoded variables representing the geometrical shape of the CMS detector. In particular these constants where used for constructing the tiles that the clustering space was subdivided into. Since the range of the coordinates was fixed, there was no need to use more complicated and flexible criteria.

```cpp
class Points2D {
    Vector<float> m_x;
    Vector<float> m_y;
};
```

```cpp
class PointsND {
    // inner vector = single point
    Vector<Vector<float>> m_coords;
};
```

Figure 3.4: AoS and SoA implementation of Points.

Both of these assumptions are limiting factors for the general use of the algorithm, because they prevent the clustering of points in higher dimensional datasets (even just in tridimensional) and the construction of the tiles through constants would require them to be hard-coded for each single application.

In the following is described how these two limits have been overcome, in order to produce a more general purpose algorithm.

3.2.1 Increasing the dimensionality

For supporting a generic number of dimensions, the first step is to rework the coordinate containers of the *point* class. The most simple way to do so is to migrate from a series of containers to a single *array of structures* which packs all the coordinates for all the points. At this point a decision has to be made, whether to make the inner array representing the coordinates of a single point statically or dinamically sized.
To make the array statically sized would require to make the size known at compile time and this can be done with C++ *non-type template parameters*. This choice has the advantages of improving runtime performance in several occasions and is particularly optimal for GPUs[1], where dynamical allocation is not optimal [18].

```cpp
template <uint8_t NDim>
class Points {
    Vector<Array<float, NDim>> m_coords;
};
```

Listing 3.1: General implementation of Points

Now that the data structures have been generalized, there is a second, more complicated problem.
As discussed above, in the original algorithm the query of the points inside the search box was implemented with two nested loops ranging in the two spatial dimensions. However, generalizing this would require to implement a number N, not known a priori, of nested loops.
The most simple way to achieve this result is to use a recursive function.

This solution is not the most efficient one though, and furthermore it presents problem for the GPU implementations, because recursive functions are highly inefficient on GPUs.
Both of these problematics can be solved easily by using templated recursion to implement what is called *loop unrolling*. This solution is more efficient than the previous one because the recursive calls are dealt with at compile time, and it also solves the problem for the GPU execution, because each time that the template parameter is decreased and

[1]Since the dimensionality is known at compile time, when executing kernels on the accelerator memory can be allocated statically. This is done during compilation, so there is no risk of clash between threads.

the next term of the recursion is called, the callee is in fact a different function from the previous.

3.2.2 The construction of the tiles

As said previously, the tiles are used to decrease the compulational complexity of the query of the neighbors for each point, so it's of paramount importance that the tile size is chosen in such a way that the number of iterations for this process is as small as possible. If the size of the tiles cannot be known a priori, it has to be deduced from the dataset.
For this reason the construction of the tiles starts from a parameter which the user can choose, $N_{perTile}$ represents the desired average number of points in each tile. From this the required number of tiles is calculated as:

$$N_{tiles} = \frac{|X|}{N_{perTile}} \tag{3.10}$$

where X is the dataset.

The tiles are constructued as N-dimensional rectangles and the number of tiles in each dimension is the same for all the dimensions, which is equal to:

$$N_{tilesPerDim} = N_{tiles}^{\frac{1}{NDim}} \tag{3.11}$$

Finally, the size of the tiles is calculated using the extremes of the coordinates X:

$$T_i = \frac{\max_j X_{ji} - \min_j X_{ji}}{N_{tilesPerDim}} \tag{3.12}$$

where T_i is the width of tiles in the dimension i and X_{ji} is the coordinate i of the j-th point in the dataset..

3.3 Developlment of a Python interface

The backend of the library is written in C++. This is the best option, since C++ is one of the fastest programming languages in the world, and many of the libraries for performance portability currently under development are written with it. However, publishing the library with only a C++ interface would not be the best choice, because clustering is particularly used in machine learning workflows, and these are mostly written in Python [2], as are most of the libraries used in the field, like PyTorch, Keras, Scikit-learn, and many more. The extensive use of Python in machine learning, as well as many more other fields in software development, is due to its simplicity of use, low verbosity and abundance of libraries providing a wide range of functionalities working out-of-the-box. The main drawback of Python however is that, being an interpreted language, its time and spatial performance are worse than that of other languages like C, C++ [19] and Rust, which are compiled and also allow a lower-level access to the computer resources.

[2] Also the R language is used for these types of application, so the implementation of an R interface for CLUEstering would make sense in the future.

In order to address this problem, many of the most popular libraries available in Python are actually written in a lower-level language like C++, and subsequently *binded* to Python, providing an interface to the backend. This provides both a handy interface to the library functionalities as well as a faster execution time that would not have been possible by writing the library directly in Python.

For these reasons, a Python interface for the library was developed[3], allowing it to be used in a much simpler way, with a more automated installation process (the next section covers this in detail) and to be more appealing for a wider group of people in the scientific community.

3.4 Installation

In order to make the library as easy to use for the future users as possible, the first step is that the installation is as straightforward as possible. In particular, with the use of Alpaka, the library supports many different backends, and the goal is to install all of them automatically, if the user's system supports them.

The installation script first looks for the libraries needed to install each backend (TBB for the CPU-multithreaded backend, CUDA for the Nvidia GPU backend and HIP-ROCm for the AMD GPU backend), and it compiles all the backend for which the depencency was found. This allows the users to have, out-of-the-box, all the backends that they can use without specifying them or without having to install separate versions of the library.

To make the library more easily available it was uploaded to the Python Package Index (PyPi) [21] [22], which is the largest software repository for Python, containing hundreds of thousands of libraries. In this way the users can read the description of the project, the dependencies, browse the past releases and, most importantly, install the library with a single command line instruction.

3.5 Outline of the Python library

The Python library is based on the *clusterer* class, which represents a collection of the clustering data and its output. The clusterer provides methods for loading data, plotting the input and the clustering output, changing the default convolutional kernel and saving the output to csv file [4]. Below is shown a basic usage of the library:

```python
import CLUEstering as clue

clusterer = clue.clusterer(3., 10., 5.)
clusterer.read_data('dataset.csv')
clusterer.run_clue()
clusterer.cluster_plotter()
```

Listing 3.2: Basic usage of the CLUEstering library.

[3]The Python interface was developed using the *Pybind11* library [20], as it's one of the most popular libraries for binding C++ and Python codes, while granting a small overhead in execution time and very little boilerplate.

[4]Comma separated value.

The clusterer is initialized passing the three main parameters of the algorithm: δ_c, ρ_c and δ_o. The average number of points per tile has a default value of 128, so it can be omitted. The number of coordinates of the dataset is not needed either because it's deduced automatically.

The input data must provide the coordinates and the weight for each point. The dataset can be provided in a multitude of formats:

- Python lists or *numpy arrays*. The data must contain two different arrays, one for the coordinates and one for the weights. The coordinates can be provided either as arrays of structures or as structures of arrays.
- Python dictionaries, pandas dataframes or csv files. The only requirement for these labelled data types is that the coordinate columns must be named `"x*"`, where the absterisc is the index of each coordinate, and the column for the point weights must be named `"weight"`. This naming allows the library to deduce the dimensionality of the dataset without needing it to be specified as a parameter.

3.5.1 Running the algorithm

The algorithm is run with the `run_clue` method of the `clusterer` class. The parameters taken by the method are shown below in Listing 3.3. The backend to be used for running the algorithm is chosen by passing a string, where the allowed values are `"cpu serial"`, `"cpu tbb"`, `"gpu cuda"` and `"gpu hip"`. It's also possible to specify the id of the device to be used and the number of threads. The `dimensions` parameter is used for clustering using only a subset of all the coodinates of each point, and it requires a list of the dimensions indexes to be used.

```python
def run_clue(self,
             backend: str = "cpu serial",
             block_size: int = 1024,
             device_id: int = 0,
             verbose: bool = False,
             dimensions.: Union[list, None] = None)
    -> None:
```

Listing 3.3: Parameters taken by the run_clue method.

Chapter 4

Computational analysis of the library

4.1 Assessing the cluster quality

As mentioned in Section 1.4, the quality of the clusters reconstructed by an algorithm can be assessed for datasets with no known truth labels using the silhouette score. In this section the output of CLUEstering will be analyzed for varying values of the input parameters by comparing the silhouette scores. For datasets where the truth labels are known, the homogeneity, completeness and normalized mutual information scores will also be used. Another useful visual tool is the confusion matrix, that allows to see how many points were correctly or incorrectly assigned to the different clusters.

Clustering datasets with known truth labels can easily be generated using the tools provided by the scikit-learn Python library. This library provides the `make_blobs` function, which generates round clusters with arbitrary width, and the `make_moons` function, which generates two partially intersecting moon-shaped clusters.

The first dataset used for testing the quality of the clusters produced by CLUEstering is a simple set of four blobs with partially overlapping edges, as shown in Figure 4.1. The clusters are reconstructed using $\delta_c = 1$, $\rho_c = 10$ and $\delta_o = 1$. In Figure 4.2 is shown the normalized confusion matrix of the clustering, which shows that approximately all the points in the dataset were assigned to the correct cluster. Furthermore, Figure 4.3 shows the values of the homogeneity, completeness and normalized mutual information scores obtained by keeping fixed $\rho_c = 10$ and $\delta o = \delta_c$, while changing the δ_c values uniformly in the range $[0.05, 1.50]$. The plot shows that for $\delta_c > 0.4$ all the three scores are approximately equal to 1, indicating almost perfect clustering of the data. Furthermore, Figures **??** and **??** show respectively the silhouette scores for each point and the scaling of the silhouette average with the δ_c parameter, and both suggest that with the combination of parameters described above, the clusters are well reconstructed.

The second dataset is similar to the first with the difference that the clusters are distributed anisotropicly, and thus have a more elongated shape. In addition, the clusters are rotated with respect to the axes, and this allows to assess that the algorithm works even when the data is not distributed symmetrically. Figure 4.6 shows the clusters obtained by applting CLUEstering with $\delta_c = 1$, $\rho_c = 10$ and $\delta_o = \delta_c$: the clusters appear well separated and the high silhouette scores in Figure 4.9, with exception of a

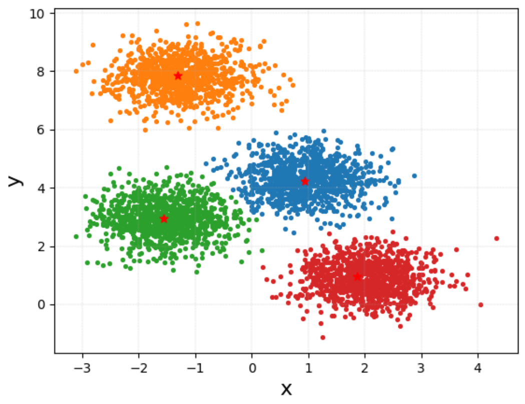


Figure 4.1: “Blob” dataset clustered with CLUEstering.

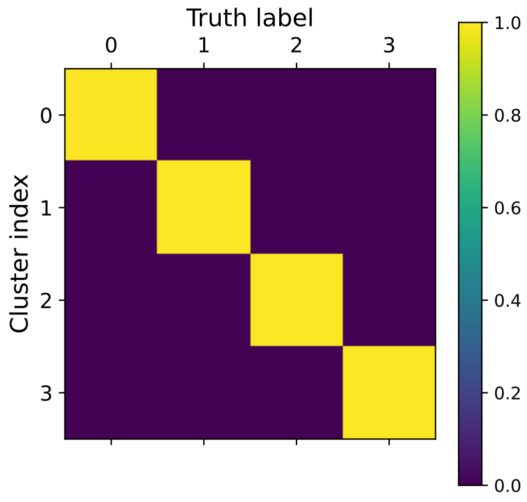


Figure 4.2: Confusion matrix for the “blob” dataset.

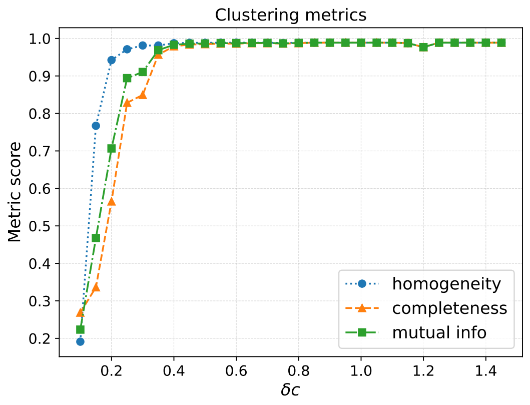


Figure 4.3: Scaling of the clustering metrics with δ_c for the “blob” dataset.

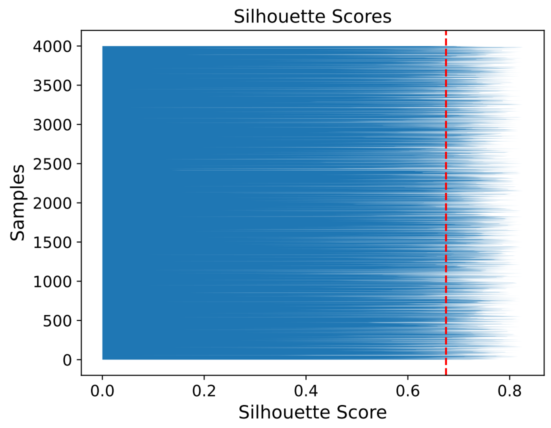


Figure 4.4: Silhouette scores for the “blob” dataset.

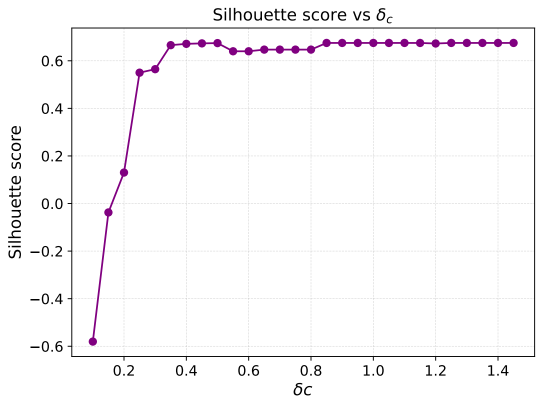


Figure 4.5: Scaling of the silhouette average with δ_c for the “blob” dataset.

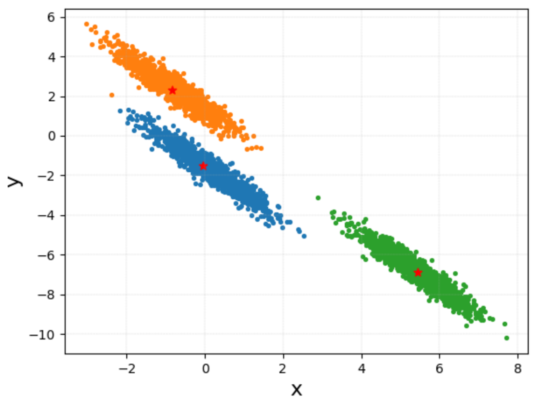


Figure 4.6: “Aniso” dataset clustered with CLUEstering.

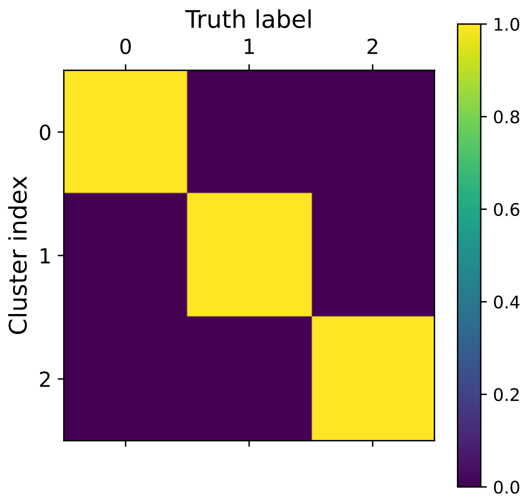


Figure 4.7: Confusion matrix for the “aniso” dataset.

few points, indicate that the clusters were well reconstructed. Figures 4.8 and 4.10 show respectively the scaling of the homogeneity, completeness and mutual information and the average silhouette score with δ_c, from which can be deduced that the best values for the parameter is found near the range $[1, 1.2]$.

The last dataset contains two partially intesecting half-moons. This dataset is particularly useful for checking that the algorithm works for clusters with curved shapes and when the distance between the clusters is not constant. As shown in Figure 4.11, with $\delta_c = 0.4$, $\rho_c = 1$ and $\delta_o = \delta_c$ CLUEstering reconstructs well the two clusters, as confirmed by the confusion matrix in Figure 4.12. With this set of parameters, the value of homogeneity is 99.7%, completeness of 99.7% and a normalized mutual informtion equal to 99.7%. In Figure 4.8 is shown how the values of the three metrics change by fixing ρ_c and changing δ_c: the homogeneity is high for small values, because the two classes are well separated so if the clusters reconstructed are small they can't contain points

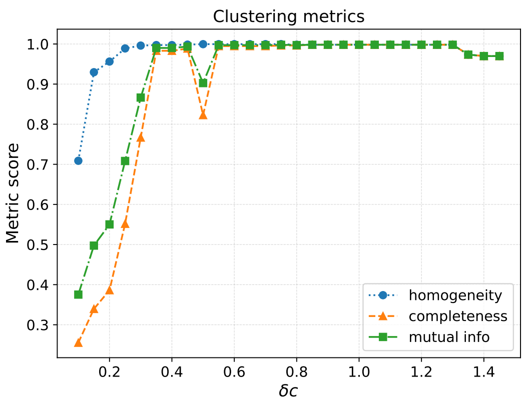


Figure 4.8: Scaling of the clustering metrics with δ_c for the "aniso" dataset.

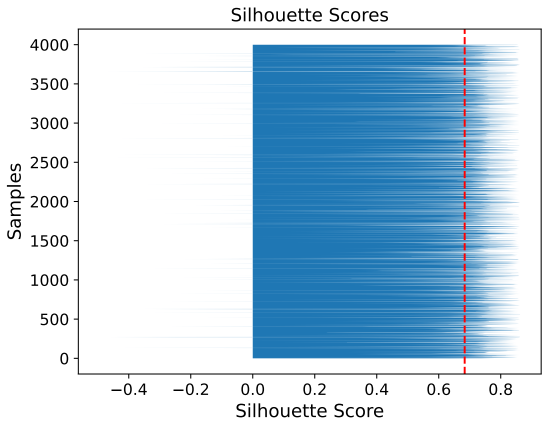


Figure 4.9: Silhouette scores for the "aniso" dataset.

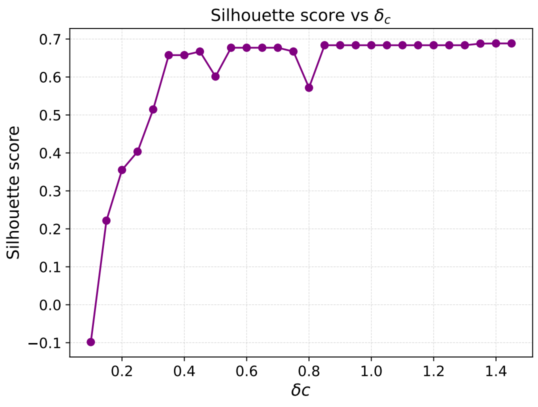


Figure 4.10: Scaling of silhouette average with δ_c for the "aniso" dataset.

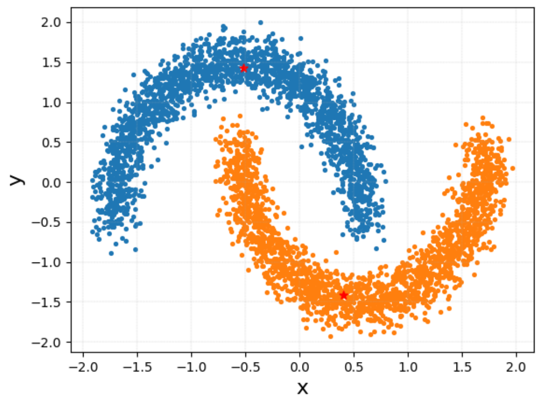


Figure 4.11: "Moons" dataset clustered with CLUEstering.

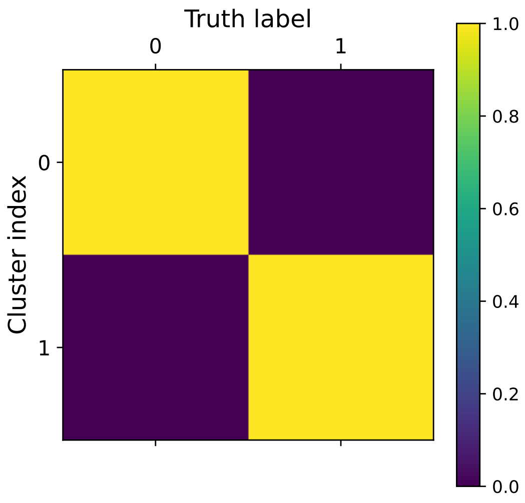


Figure 4.12: Confusion matrix for the "moons" dataset.

from the other class. For this same reason the completeness and mutual information start from small values. The three metrics then reach a peak of approximately 100% for $\delta_c = 0.4$, which confirms the results presented above.

In Figure 4.14 are shown the silhouette scores for the all the points and what can easily be seen is that many points have negative scores and the largest positive values only reach 0.6, which results in an average of 0.4. These values indicate that the quality of these clusters can't be assessed with the silhouette method, and this is probably due to the shape of the clusters: since the clusters are partially overlapping the distance of a point with the points in the same cluster is not smaller than the distance with the points in the other cluster for all the combinations, and this decreases the average.

However, given the good separation of the two clusters the Dunn index is particularly indicated for this dataset: as shown in Figure 4.15 the values of the Dunn index have a clear maximum in correspondence of $\delta_c = 0.4$, which confirms that it's the parameter that produces the best clusters.

Finally, the clustering obtained with CLUEstering has been compared with DBSCAN and HDBSCAN. In Tables 4.1, 4.2 and 4.2 are shown the comparison of the values of the metrics as well as the execution times. The quality of the clusters are generally comparable among the three algorithms, with at most differences smaller than 1%. Regarding the speed of execution, it can be seen from the bar plot in Figure 4.16 that CLUEstering is much faster than HDBSCAN in all datasets and marginally faster than DBSCAN for all datasets except the "blobs" dataset. It should be noted however that the quality of the clusters produced by DBSCAN for this dataset is noticeably worse, which balances the slight better performance.

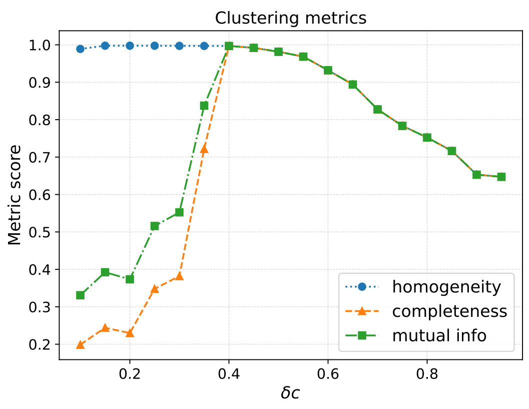


Figure 4.13: Scaling of the clustering metrics with δ_c for the "moons" dataset.

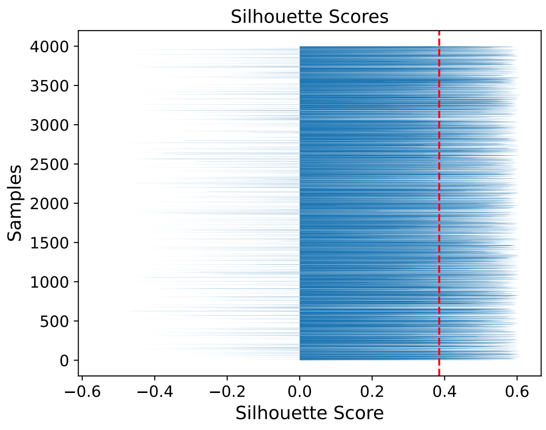


Figure 4.14: Silhouette scores for the "moons" dataset. The abundance of negative values and the low average indicate that the silhouette score is not a good metric for this particular dataset.

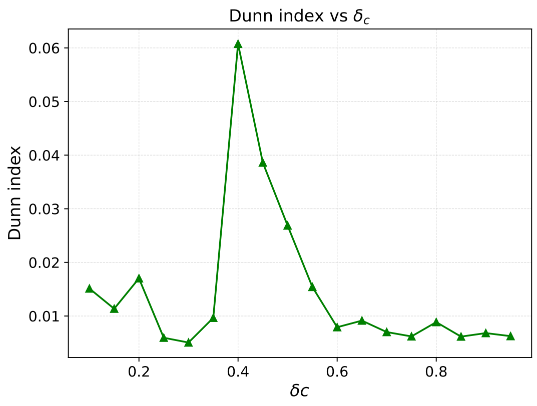


Figure 4.15: Scaling of the Dunn index with δ_c for the "moons" dataset. The index has a clear global maximum in $\delta_c = 0.4$, indicating that the parameter is the one that produces the clusters with the best separation.

	CLUEstering	**DBSCAN**	**HDBSCAN**
Execution time	34 ± 4 ms	42 ± 6 ms	135 ± 2 ms
Homogeneity	99.8%	99.9%	99.9%
Completeness	99.8%	99.5%	99.6%
Mutual info.	99.8%	99.7%	99.8%
Silhouette	68.4%	67.7%	68.1%

Table 4.1: Comparison of the results of CLUEstering, DBSCAN and HDBSCAN on "aniso" dataset.

	CLUEstering	**DBSCAN**	**HDBSCAN**
Execution time	$26 \pm 3\,\mathrm{ms}$	$31 \pm 3\,\mathrm{ms}$	$141 \pm 9\,\mathrm{ms}$
Homogeneity	99.7%	99.7%	100%
Completeness	99.7%	98.7%	99.2%
Mutual info.	99.7%	99.2%	99.6%
Silhouette	38.5%	29.8%	20.2%

Table 4.2: Comparison of the results of CLUEstering, DBSCAN and HDBSCAN on “moon” dataset.

	CLUEstering	**DBSCAN**	**HDBSCAN**
Execution time	$34 \pm 3\,\mathrm{ms}$	$29 \pm 2\,\mathrm{ms}$	$131 \pm 3\,\mathrm{ms}$
Homogeneity	99.9%	74.0%	97.5%
Completeness	99.9%	95.1%	92.3%
Mutual info.	99.9%	22.3%	94.8%
Silhouette	67.5%	58.8%	65.3%

Table 4.3: Comparison of the results of CLUEstering, DBSCAN and HDBSCAN on “blob” dataset.

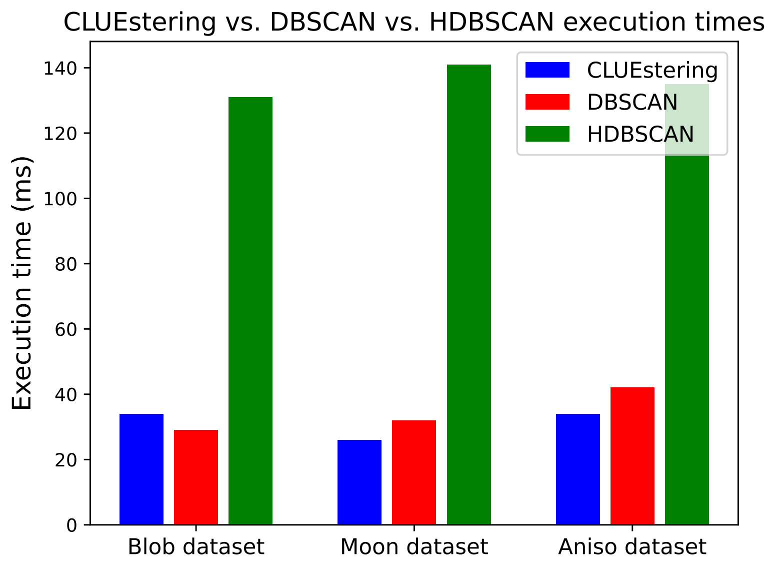


Figure 4.16: Comparison of the execution times of CLUEstering, DBSCAN and HDBSCAN.

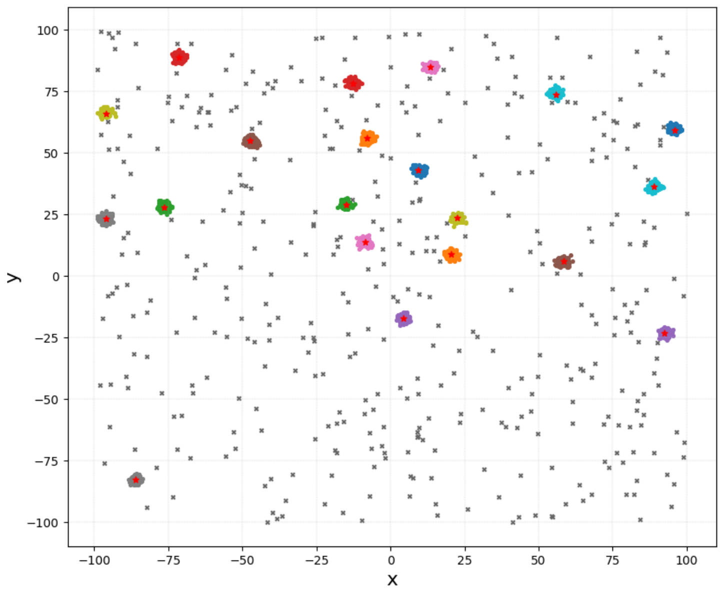


Figure 4.17: Dataset containing gaussian blobs and uniform noise.

4.2 Benchmarks

4.2.1 Scaling of execution time with dataset size

Nowadays the amount of data produced in any kind of application is increasing at an astounding rate. For this reason it's crucial that modern software is able to handle big data.
The library has first been benchmarked by studying the execution times as a function of the number of points in a dataset. In order to remove any possible bias, the data had to be generated in a general way: 90% of the points were used to make up an arbitrary number of clusters, whereas the remaining 10% would be used to produce noise by scattering points over the entire clustering space according to a uniform distribution. In this way the probability density of each point in the dataset is given by:

$$\rho(\mathbf{r}) = \mathcal{U}(\mathbf{r}_{min}, \mathbf{r}_{max}) + \sum_{i \in C} \mathcal{N}(\mathbf{r}_i, \sigma) \tag{4.1}$$

The measures of the execution times with this class of dataset are shown in Figure 4.18.

4.2.2 Execution time in function of the dimensionality

One of the main changes of the CLUEstering algorithm with respect to the original CLUE is the support of higher-dimensional datasets. For this reason it's interesting to study how the execution times vary as the number of dimensions increases. There is an inherent problem in performing this kind of analysis: as the dimensionality increases, it

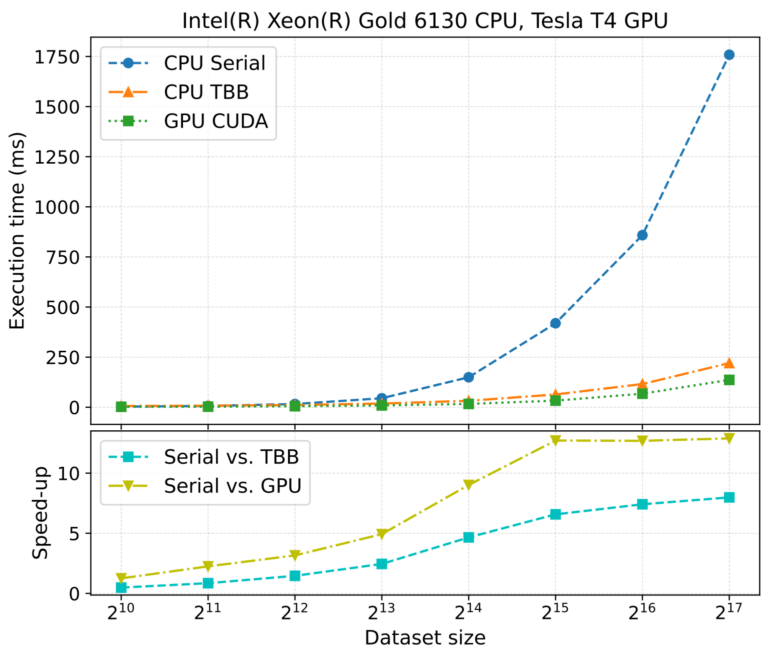


Figure 4.18: Scaling of execution time with dataset size.

must be assured that the computational load of the added dimensions is comparable to that of the previous, so that the increase in the execution time can be attributed purely to the increase in dimensions and is not masked by internal effects which are difficult to predict. To this scope, the same dataset described in section 4.2.1 was used: by keeping the total number of points constant and distributing them randomly in all the directions it is assured that the additional tiles in the new dimensions get filled as much as the others.

The plot in Figure 4.19 shows a steady increase in the execution time with the number of dimensions, as would be expected.

4.2.3 Tracing of the algorithm

Tracing of software means recording information about its execution for debugging or analysis purposes. Tracing is often used to analyze the execution times of the main portions of a piece of software, in order to assess its performance and identify its bottlenecks. There are several tools which can be used for tracing C++/CUDA software: for base serial C++ `gprof` [23] can be used, and its Nvidia counterpart `nvprof` for CUDA software. These two profilers give a very detailed analysis of the execution, the time by each individual function as well as the number of its calls. For CUDA software the Nvidia Nsight Systems [24] is another very common tool, which also provides a handy graphical user interface for easier visualization and analysis of the results and easy connection to remote machines with `ssh`.

Tracing the execution of CLUEstering is interesting for analyzing the time taken by each of its components and comparing the trace of the different backends allows to see which parts are the most expensive for each processor. The main components of the execution of the CLUEstering algorithm are: the setup of the tiles, the data transfer

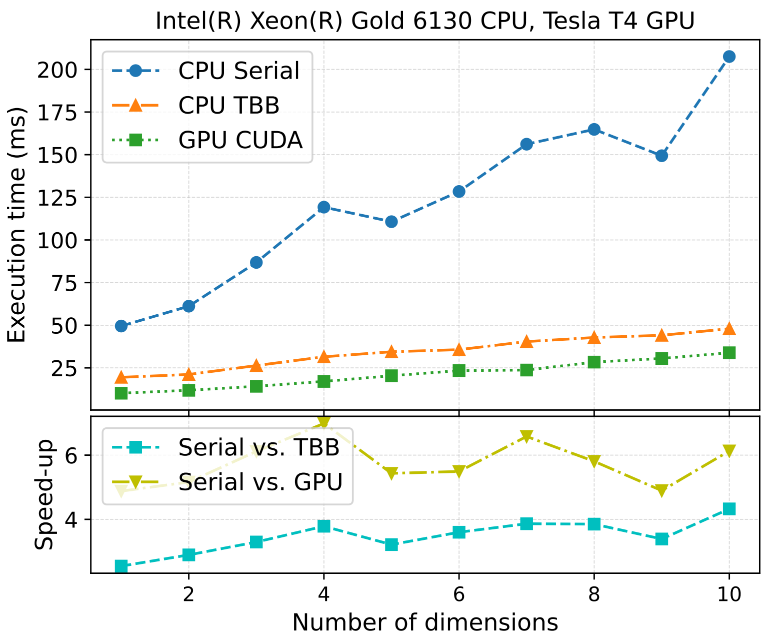


Figure 4.19: Scaling of execution time with the number of dimensions.

from host to device, the calculation of local density and the nearest higher for each point, finding and assigning clusters and the final data transfer from the device back to the host.

In Figure 4.20 can be seen the result of the tracing for the serial, TBB and CUDA backends run on a random dataset like the one in Figure 4.17 generated with 2048 points.

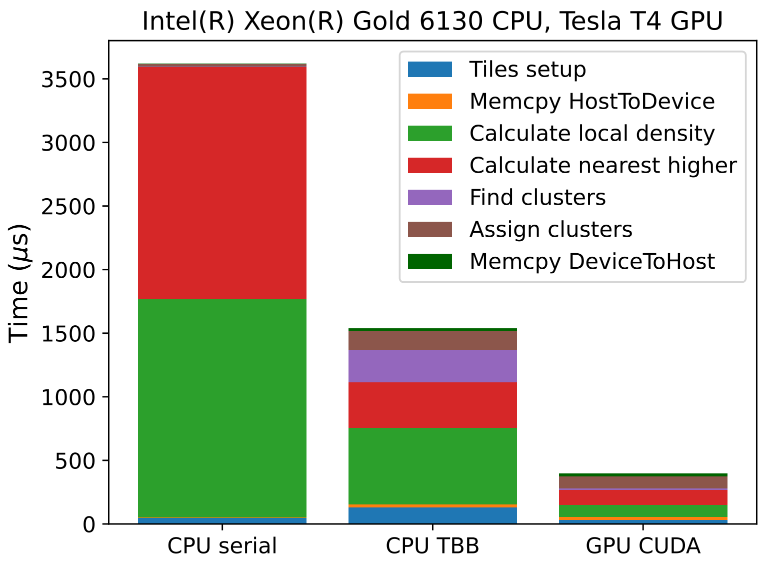


Figure 4.20: Results of the tracing on a dataset containing 2048 points.

Chapter 5

Applications of CLUEstering

The following chapter shows the application of CLUEstering to two problems coming from two different branches of science: vertex reconstruction in high-energy physics and reconstruction of stars from the point spread functions detected by CCD photometers in astronomy.

5.1 Vertex reconstruction in High-energy physics

5.1.1 Introduction to vertex reconstruction at CMS

The Compact Muon Solenoid, CMS (shown in Figure 5.1), is one of the two general-purpose detectors at LHC. While the accelerator is running, at the center of the detector every 25 nanoseconds the two beams of protons cross and the protons collide, and the energy released by these collisions can produce a moltitude of particles. In order to identify the particles produced by each beam crossing, which is called an *event*, the interation points corresponding to each proton-proton collision have to be reconstructed.

An *interaction point*, also called *vertex*, is a point in space where particles collide, interact or decay. In other words, it's the point where the particle trajectories intesect. Vertices are divided into *primary vertices*, which correspond to the interaction of the two particle beams, and *secondary vertices*, that are correlated to the decay of unstable particles.

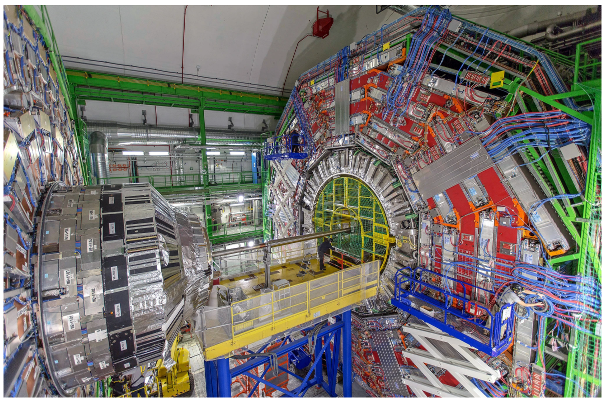


Figure 5.1: The CMS detector.

In the CMS detector, the particle hits are reconstructed through the energy deposits left by the charged particles on the silicon sensors of the detector layers. From the hits the tracks are then reconstructed using pattern recognition algorithms based on the concept of *cellular automata* and *Kalman filter*. Then the reconstruction of the vertices identifies the tracks associated to the same interactions, and this is crucial for understanding the physics of each vertex and identifying the particles produced in each proton-proton collision.

Vertex reconstruction is divided in two stages [25]: *vertex finding*, which groups tracks that have a compatible spatial origin, and *vertex fitting*, which deduces the properties of the vertex from the set of compatible tracks. One of the main quantities that is computed in vertex fitting is the z coordinate of the vertex, which is also one of the main criteria used when comparing the simulated tracks with the reconstructed tracks obtained with an algorithm [1].

The quality of the reconstructed vertices is computed by comparing them to the simulated vertices, for which the truth information is known. Reconstructed vertices are said to be *misidentified*, or *fakes*, if they are not matched to any of the simulated vertices. Reconstructed vertices that are matched to the same simulated vertices are called *duplicates*. If more than one simulated vertices are matched to the same reconstructed vertex, the latter is said to be *multimatched*, or *merged*. The metrics used to quantify the physics performance of the reconstruction method are:

- efficiency, the fraction of simulated vertices matched to at least one reconstructed vertex.
- purity, the fraction of pure reconstructed vertices, i.e. the reconstructed vertices where the number of tracks which don't belong to the matched simulated vertex is below a certain threshold.
- fake rate, the fraction of reconstructed vertices that are not associated to any simulated vertex.
- duplicate rate, the fraction of simulated vertices associated with multiple reconstructed vertices.
- merge rate, the fraction of reconstructed vertices matched to multiple simulated vertices.

Then, for matched vertices two more metrics are defined: *resolution*, which is the standard deviation of the one-dimensional distance between the simulated and reconstructed vertices, and the *vertex coordinate pull*, defined as the ratio between the one-dimensional distance and the error on the reconstructed coordinate.

5.1.2 Results of the clustering

The goal of this section is to use CLUEstering for finding vertices in a simulated dataset of the CMS experiment. The clustering is done in one dimension using the z of the

[1]In the vertex reconstruction of the CMS experiment for example a reconstructed vertex is considered matched to a simulated vertex if $|\Delta z| < 1$ and $|\Delta z|/\sigma_z < 3$, where σ_z is the uncertainty on the reconstructed vertex z position.

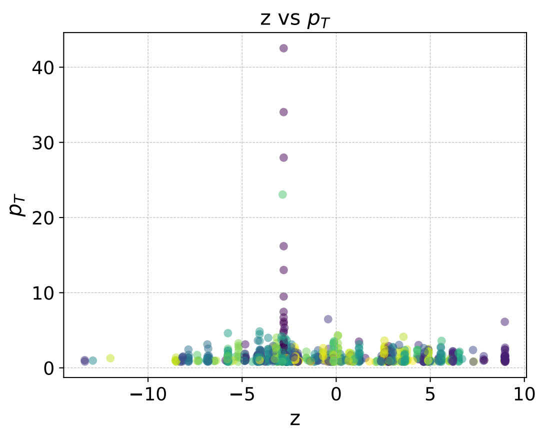


Figure 5.2: Scatted plot of the z coordinate and transverse momenta of the simulated tracks.

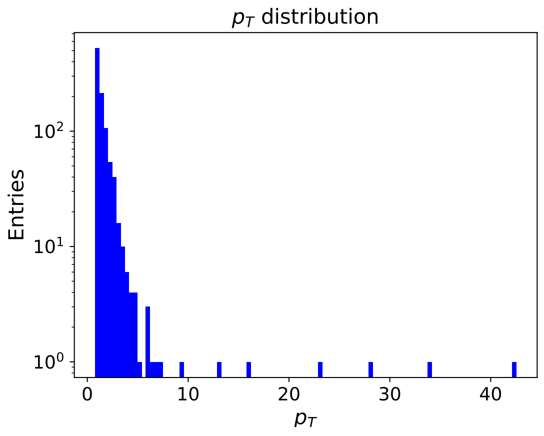


Figure 5.3: Distribution of the p_T values for the simulated tracks.

simulated tracks as the coordinate and their transverse momenta p_T as weight. The choice for clustering in z is sensible because it's the coordinate which is usually used for comparing simulated and reconstructed vertices after vertex fitting.

In order to improve the purity of the reconstructed vertices, cuts are applied to the values of pseudorapidity η [2] and p_T of the simulated tracks. In particular, all the simulated tracks with transverse momentum smaller than $0.8\,GeV$ or pseudorapidity larger than 2.4 are discarded.

In order to use CLUEstering on this dataset, the first step is getting an idea of the correct parameters to use. In Figure 5.2 is a shown a scatter plot of the z coordinate of the simulated tracks and their transverse momenta. The plot shows that the data is divided into many small groups of tracks with similar momentum and very close values of z. This suggest that a correct value for δ_c can be inferred by taking the approximate width of any of those clusters, which by inspecting the image more closely is contained in the range $\delta_c \in [0.1, 0.5]$. Regarding ρ_c, the histogram in Figure 5.3 shows the distribution of the p_T values, which is being used as a weight for the points. The plot shows that the vast majority of points have a weight in the range $p_T \in [0, 5]$, that combined with the expected average number of tracks associated with a vertex, which again can be estimated from Figure 5.2, suggests a value of ρ_c of 10, which can later be increased if the resulting number of clusters is too high. The value of δ_o, on the other hand, largely depends on the quality of the vertices that want to be reconstructed: a large δ_o value would produce few large clusters, decreasing the fake rate, however it would increase the merge rate and decrease the purity of the vertices; on the other hand, while a small value of δ_o would make the clusters smaller and thus probably improve the purity of the vertices, it would also increase a lot the number of clusters, which would result in a much higher fake rate. For this reason the reconstruction has been repeated many times by

[2] *Pseudorapidity* η is a spatial coordinate that describes the angle of a particle trajectory with respect to the beam axis. It's defined as:

$$\eta = -\log\left[\tan\left(\frac{\theta}{2}\right)\right]$$

where θ is the angle between the positive beam direction and the particle 3-momentum vector.

fixing the values of δ_c and ρ_c and using many values of δ_o values taken uniformly in the range $\delta_o \in [0.01, 0.9]$, which steps of 0.01.

Figure 5.4 shows the results of the vertex reconstruction with CLUEstering: in the majority of the range chosen for δ_o the efficiency is high, near and above 90%, while the purity is lower, below 40%. This indicates that the vertices reconstruted are not very pure, which is also shown by the values of fake rate and merge rate, which suggests that in order to obtain good physical results the one-dimensional clustering is probably too simple, and other coordinates would have to be included. Regarding the values of fake, merge and duplicate rate there are some considerations: the fake rate is higher for smaller values of δ_o, and this is easily explained by the fact that the resulting clusters are more in number but also much smaller, so they can't possibly be matched to any simulated vertex; the merge rate is low for small values of δ_o but quickly grows as the parameter increases, and this is predictable because as the clusters grow larger the number of clusters containing more than one simulated vertex is bound to increase. Finally, the duplicate rate is very small for almost all the combinations of parameters, which is explained by two things: first, since the clustering is one-dimensional, the probability of clusters overlapping is smaller; second, for small values of δ_c and δ_o the clusters are small, which makes it harder for them to be matched to a simulated vertex, whereas for higher values of the parameters the clusters are larger, which makes it difficult for two adjacent parameters to share enough tracks with a simulated vertex for both to be matched.

Analyzing the plots in Figure 5.4 it can be seen that the combination of δ_c and δ_o that maximizes efficiency and purity while compromising the fake and merge rates are $\delta_c = 0.2$, $\delta_o = 0.052$ and $\delta_c = 0.2$, $\delta_o = 0.074$. Then, Figure 5.5 shows for these combinations of parameters the scaling of the metrics with ρ_c, sampled in uniform intervals in the range $\rho_c \in [1, 30]$. It can be seen that both for $\delta_o = 0.052$ and $\delta_o = 0.074$ the best value of efficiency and purity, while maintaining low fake and merge rates, is obtained for $\rho_c = 5$. Furthermore, Figure 5.7 shows that for $\delta_c = 0.2$, $\rho_c = 5$ and $\delta_o = 0.074$ the algorithm reconstructs 43, where the number of reconstructible simulated vertices is 62, which means that CLUEstering correctly reconstructs more than two thirds of the reconstructible vertices.

5.2 Clustering of stars from a telescope image

5.2.1 Introduction to stellar photometry

Photometry is a technique used in astronomy which consists in measuring the intensity of the light radiated by celestial objects, also called *flux*. Light is measured by telescopes containing devices called *photometers*, which make use of *charge-coupled devices* (CCDs) [3].

Stars, due to ther distance from Earth can be typically treated as point light sources [4].

[3]A *charge-coupled device* (CCD) is an integrated circuit made up by MOS capacitors, that convert the incoming photons into electrons at the semiconductor-oxide interface, whose energy is then interpreted and stored as an image.

[4]While this is true for "smaller" celestial objects, like stars, this is not true for objects as large as galaxies of clusters of galaxies.

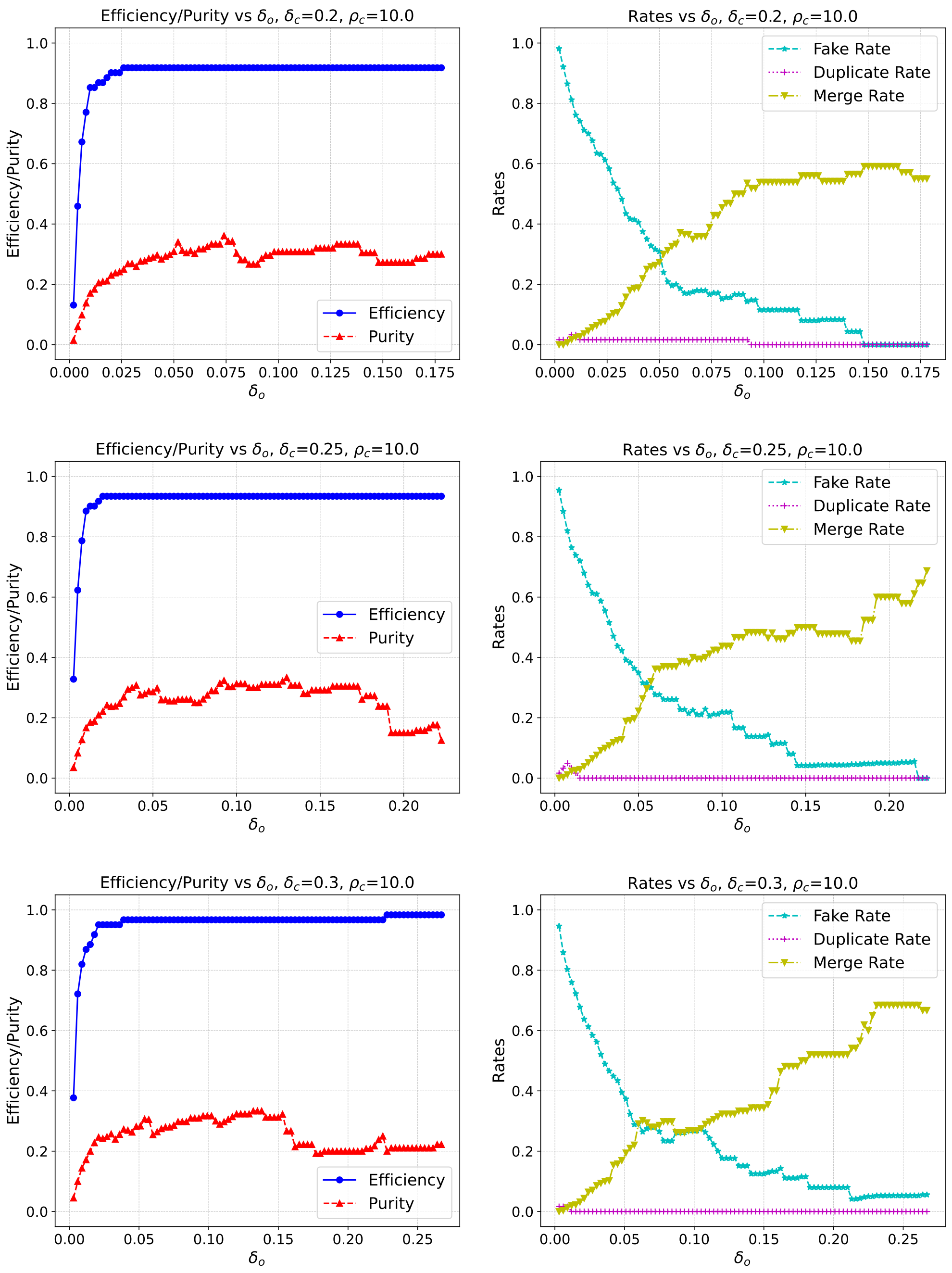


Figure 5.4: Results of the vertex reconstruction for different values of δ_c and δ_o.

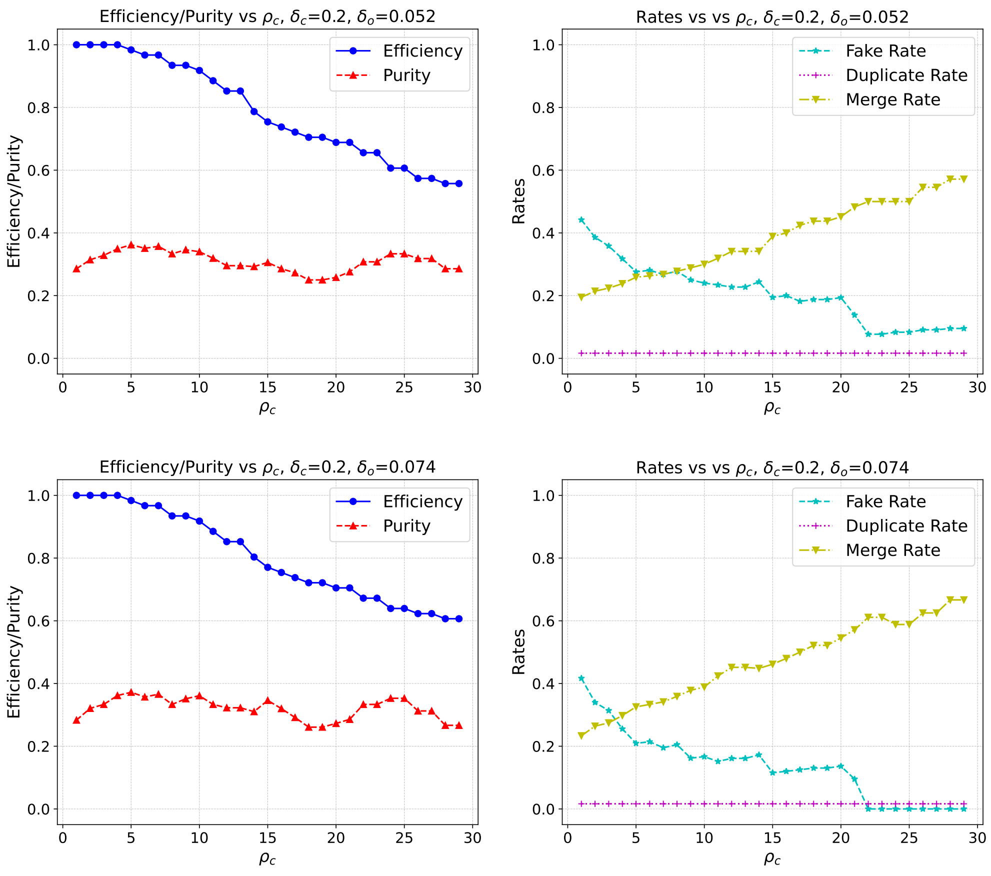


Figure 5.5: Results of the vertex reconstruction as a function of ρ_c fixing δ_c and δ_o.

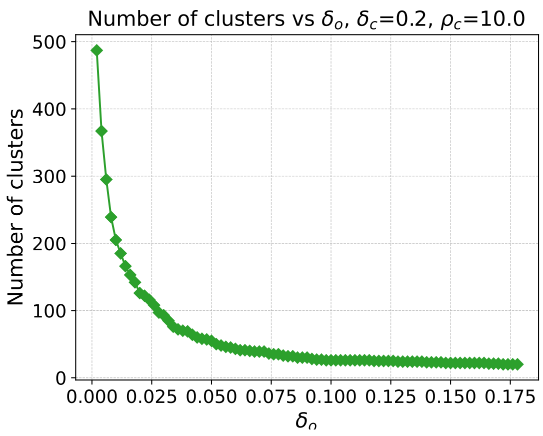


Figure 5.6: Scaling of the number of clusters reconstructed by CLUEstering with δ_o.

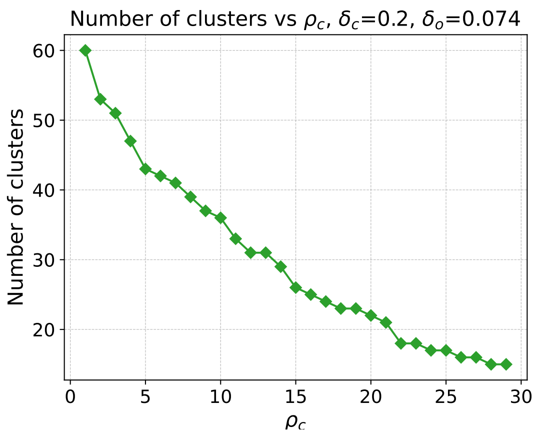


Figure 5.7: Scaling of the number of clusters reconstructed by CLUEstering with ρ_c.

However, their patterns when picked up by photometers are never point-like, insead they are spread out, turning on many neighouring pixels and forming what are called *point spread functions*. The widening of the patterns picked up by the telescope's photometer is mainly due to two effects:

- *seeing*, which is related to the inhomogeneities in the atmosphere's refractive index caused by its turbulent mixing due to the variations in temperature and pressure.

$$\theta_{seeing} \sim \lambda^{-1/5} \tag{5.1}$$

- *diffraction*, which is related to the limit in the spatial resolution of the image.

$$\theta_{diff} \sim \frac{\lambda}{D} \tag{5.2}$$

The image reconstructed by the detector signal contains for each pixel a number of counts that is linearly proportional to the number of photons falling in that pixel. The count in each pixel is given by the combinations of signal and noise: the signal is composed by the astronomical resolved sources and by the sky background, whereas noise is due to poissonian fluctuations[5] and instrumental effects.

In a detector the signal received from a source increases linearly with the exposure time, but each pixel can only store a limited number of electrons. This means that when the exposure time increases the signal of some sources will reach the saturation level (as shown in Figure 5.8), which is dependent on the instrument used. When the saturation level is reached the pixel is full and its count can't increase further, so the extra-electrons will fill the nieghboring pixels. Saturation can be avoided by limiting the exposure, but in that way dimmer sources might not be detectable anymore.

Magnitude is a measure of the brightness of celestial objects. It works in reverse, meaning that smaller values indicate a brighter object, and is designed in a logarithmic scale in such a way that each increase of 1 magnitude means a ≈ 2.512 brighter object. There are two main definitions of magnitude: *apparent magnitude*, *absolute magnitude.* The apparent magnitude of an object is calculated using a reference object for which the flux and magnitude are known, so that the magnitude is:

$$m_{app} = m_{ref} - 2.5 \log_{10} \left(\frac{F}{F_{ref}} \right) \tag{5.3}$$

where F is the flux of the object.

Absolute magnitude is a measure of the magnitude which is independent on the distance of the object from Earth, and is defined as the apparent magnitude that the object would have if it was at a known distance, which is usually chosen to be 10 parsecs. Then the absolute magnitude is calculated as:

$$M = m - 2.5 \log \left(\frac{d}{10} \right)^2 \tag{5.4}$$

[5]The number of photons emitted by a source and eventually detected by the experimental apparatus follows a poisson distribution, which means that the signal will have statistical noise given by the variance of the distribution.

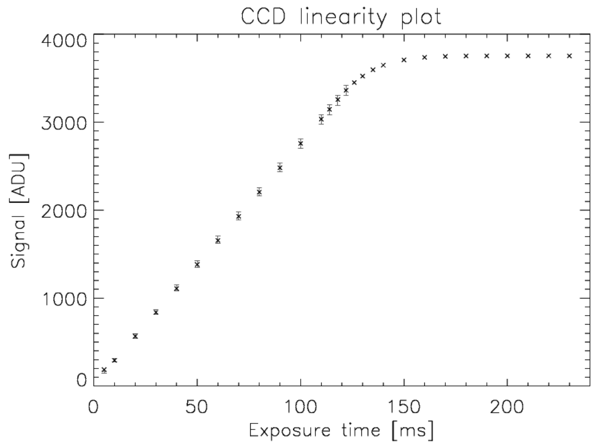


Figure 5.8: Scaling of signal measured in a CCD photometer with exposure time. It can be seen that for long expsure times the pixels get saturated and the number of counter electrons can't get any higher, meaning that the signal stops increasing.

where m is the object's apparent magnitude and d its distance from Earth.

When magnitude is measured using a photometer, what is computed is called *instrumental magnitude*, which is an uncalibrated measure of the apparent magnitude, obtained as:

$$m = -2.5 \log_{10}(f) \tag{5.5}$$

where f is the source's flux. The difference between instrumental magnitude and the other two definitions is that, since it's not scaled to a reference object, it's only significant when compared to the instrumental magnitude of the objects in the same image. So it's meant as a comparison of the brightenss of the objects in the same image, rather than an absolute measure.

After measuring the signal with the photometer and taking the PSF images, the next step is usually to compute the magnitude values for all the objects. This is usually done in two ways: *aperture photometry* and *PSF-fitting*.

In the aperture photometry method, the simplest of the two, the flux of the source is computed by summing the pixel contents inside a fixed radius and subtracting the flux attributed to the sky, calculated as the product of the average sky pixel count and the area of the aperture used. So the formula for the instrumental magnitude in aperture photometry is:

$$m = -2.5 \log\left(\sum_{p \in A} p - \overline{p}_{sky} * \mu(A)\right) \tag{5.6}$$

where p are the values of pixels inside the aperture A, that contain both the signal due to the object as well as the sky, and $\mu(A)$ is the area of the aperture. In aperture photometry the main difficulty is choosing the proper aperture, since a radius too large may include other stars counts, but a radius too small would cut out part of the object.

PSF-fitting calculates the magnitudes by modelling the distribution of the PSFs and then calculating the volume of the best-fit curve for each of the objects. As said above, all the stars can be equally considered as point-like sources, which means that all the

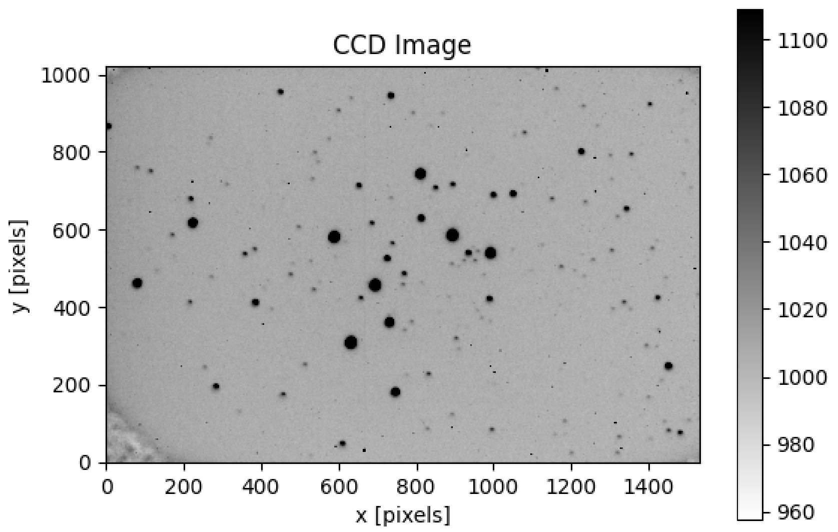


Figure 5.9: Image containing the PSFs detected by the RHUL telescope.

PSFs can be modelled using the same function, which is usually a gaussian. Then, the instrumental magnitude of each source is calculated as

$$m = -2.5 \log_{10}(V) \tag{5.7}$$

where V is the volume under the best-fit curve. PSF-fitting is usually the preferred choice in the case of stellar crowding, when the counts inside a given aperture radius are contaminated by the partial overlap of several sources, which means that aperture photometry is not adequate.

5.2.2 Results of the clustering

The image in Figure 5.9 was taken with the telescope of the Royal Holloway [26] Observatory of the University of London. The telescope is a Meade LX200 model, a Schimdt-Cassegrain with a 10-inch primary mirror.

The first step for using CLUEstering is to estimate the three parameters, δ_c, ρ_c and δ_m. The δ_c parameter can be estimated by considering the size of the smallest cluster to be reconstructed in the dataset. In this particular datasets, which is an image of 1530×1020 pixels, the smallest relevant feature should have a size of at least 10 pixels, so $\delta_c = 10$ is a good starting value which will require at most some small tweaking. The choice of ρ_c usually just requires to estimate the minimum number of points that need to be in a space region in order to have a cluster, and in this way it's equivalent to the `minPts` parameter of DBSCAN. However, if the weights of the points are not all equal the estimation of ρ_c requires some further considerations. Since for this particular dataset the clusters represent physical entities, stars in particular, the parameter can be estimated based on the lowest intensity for a signal to be significant. An initial estimate of $\rho_c = 10000$ was chosen.

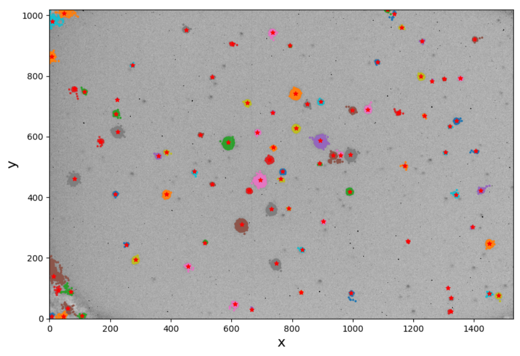


Figure 5.10: Reconstruction of the stars from the PSF image with CLUEstering.

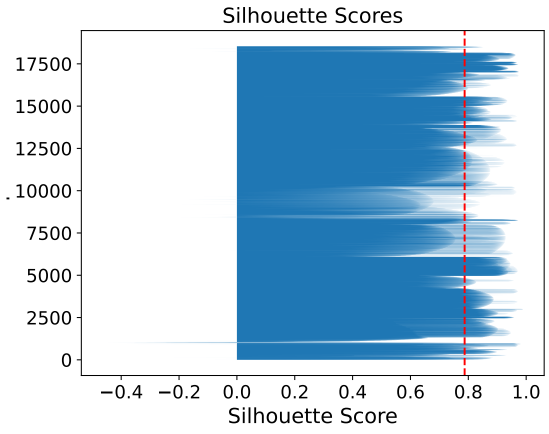


Figure 5.11: Silhouette scores obtained using the CLUEstering library on the dataset.

Finally, the δ_m parameter is initialized by default equal to δ_m, and in many cases this value is correct. It should be adjusted in the case that the clusters appear to be too small, which would mean there are points which are getting excluded because there are no points in their neighborhood with high enough energy.

In Figure 5.10 is shown the output of the clustering using this set of parameters. The plot is overlayed to the original image (Figure 5.9), and this allows to see that most of the stars have been correctly detected and recostructed as clusters.

The quality of the clusters produced can be assessed by computing the silhouette score for all the points, which can be seen in Figure 5.11: most of the points have a high score and the average over the whole dataset is 78.7%, indicating that the points are well matched to the clusters. There are however several points with negative score. These can be attributed to the clusters generated by the noise in the left corners of the image. Thus it's reasonable to remove the points in those corners and repeat the clustering [6].

Figure 5.12 shows the result of the clustering after manually removing the left corners of the image: the clusters due to the artifacts have disappeared and Figure 5.13 shows that the points with negative silhouette score are significantly less and the average score has increased to 80.0%.

As a further test for the quality of the clustering procedure, is shown a comparison of the total fluxes for the stars reconstructed with CLUEstering and with state-of-the-art Python libraries. The reference values are obtained using the `photoutils` library [27], which is a widely used Python library for photometry analysis: the stars are identified using the `DaoStarFinder` class, which detects the stars using two parameters, a threshold for the minimum pixel value for selecting sources and the full-width half-maximum (FWHM) of the gaussian kernel. The star finder computes the centroids of the stars, whose pixel count can be calculated using the `CircularAperture` class, which performs aperture photometry with a given radius.

[6]The artifacts in the image can be dealt with in several ways: they can be removed by hand in pre-processing, if it's easy to confidently determine which points are noise and which aren't, or they can be removed in post-processing, either by hand or using filters or machine learning techniques.

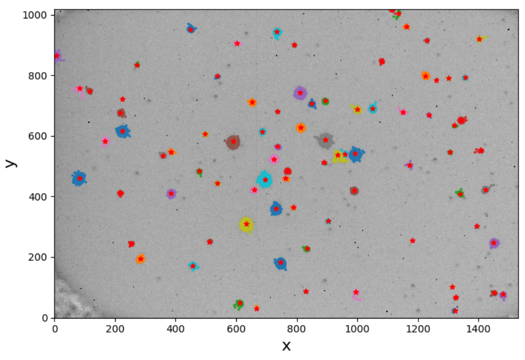


Figure 5.12: Reconstruction of the stars with CLUEstering after cleaning the image.

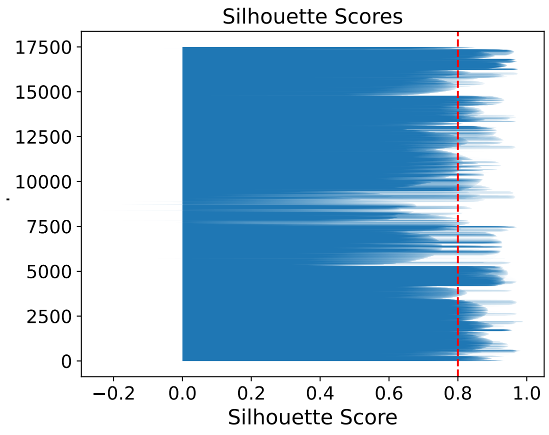


Figure 5.13: Silhouette scores obtained with after trimming the left corners of the image.

The signal of the image is obtained by subtracting the pixel values the median, which represents the background sky level of the image. The star finder is constructes with FWHM = 5 and a signal threshold equal to 6σ, where σ is the standard deviation of the pixel values. Then the aperture photometry is performed with an aperture $r = 9$, obtaining the distribution of total pixel counts for all the reconstructed stars. The same analysis is performed using CLUEstering with $\delta_c = 5$, $\rho_c = 20000$ and $\delta_o = \delta_c$. The data for CLUEstering is the image data with a cut of 1040 on intensity and subtracted by the median of the unfiltered image. The results of the two analysis methods are shown in Figure 5.14: The two histograms have a similar shape, indicating that the results are comparable for the majority of the stars reconstructed. Furthermore the two methods reconstruct approximately the same number of stars (64 for CLUEstering and 63 for DAOStarFinder). CLUEstering's fluxes are slightly shifted towards higher values, which might be due to the cut in the pixel intensity done during pre-processing. As a final remark, the reconstruction of the stars with CLUEstering took $59 \pm 2\,ms$, while DAOStarFinder took $262 \pm 15\,ms$[7], so CLUEstering resulted to be more than 4 times faster, without considering the pre-processing phases. In conclusion, the results obtained with CLUEstering suggests that it could be a useful tool in the reconstruction of stars from PSF images. However it should be noted that its robustness and the dependency of its results with the cut on the input data and the clustering parameters would require further study on a larger number of images.

[7]For both the analysis methods the time measures were taken 10 times and the mean and standard deviations were computed.

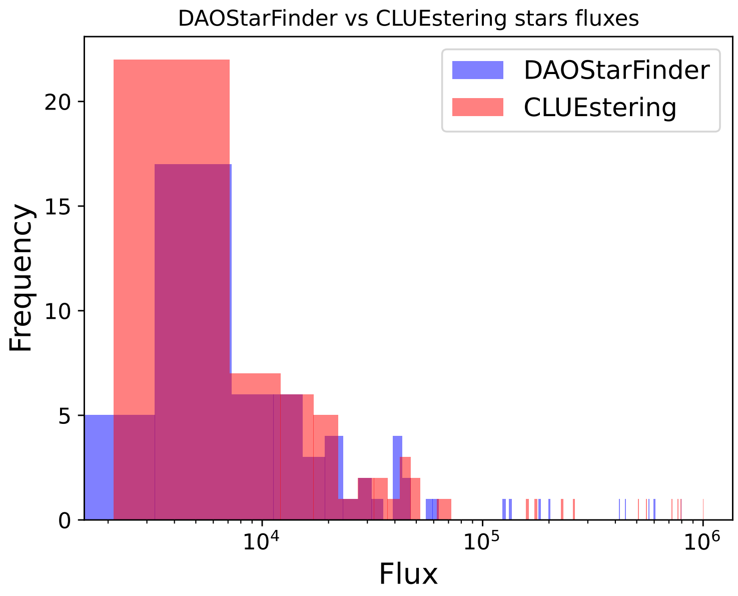


Figure 5.14: Comparison of stars aperture sums with CLUEstering and DAOStarFinder.

Conclusions and future developments

This thesis presented the characteristics and the development of the CLUEstering library. CLUEstering is a Python library which provides a wrapper around a generalization of the CLUE algorithm, the clustering algorithm used for the high-granularity calorimeters of the CMS experiment at CERN.

What makes CLUEstering different is the algorithm that it uses, because it's a very efficient and entirely parallel algorithm, which is developed to take advatage of modern parallel processors, like GPUs, in the most efficient way possible. Furthermore, the backend of the library is entirely developed using the Alpaka library, an highly efficient performance portability library that allows the automatic compilation of the library for many different platforms without requiring any code duplication or the constant maintenance of several codebases.

The benchmarks of the library show that, as expected, the parallel implementations of the algorithm, in particular the GPU backends, improve considerably the performance. In particular the parallel backends scale much better with the size of the dataset as well as the number of dimensions of the data, making the algorithm much more flexible and applicable to a wider range of applications.

The quality of the clusters produced by CLUEstering has been assessed for some common datasets using the most widely used metrics, both for labelled data, the silhouette and Dunn indices, and for unlabelled data, the homogeneity, completeness and mutual information scores. The quality of the clusters has been shown to be comparable if not better to that obtained with the DBSCAN and HDBSCAN algorithm, and the comparison of the execution times showed that CLUEstering achieved comparable results noticeably faster, in particular with respect to HDBSCAN.

Finally, CLUEstering was applied to two problems in different areas of science: vertex reconstruction in high-energy physics and star detection from PSF images in astronomy. For vertex reconstruction, CLUEstering was able to reconstruct good clusters and achieving good results for efficiency and purity, as well as fake/duplicate/merge-rates. Regarding the detection of stars, CLUEstering could reproduce the same number of stars, or more, that were found using standard photometry tools available in Python. The quality of the clusters produced was assessed with the silhouette score (because no truth labels were known), which presented high values for all the points in the dataset and an average score around 80%. Then, the pixel counts of each star were computed and compared to the ones obtained using aperture photometry, which showed similar results and faster execution times. In conclusion, CLUEstering proved to be a valuable tool for both of these applications, and with further research it could be used for real-life analysis.

While there are many clustering libraries already available, many of them already very well known among the scientific community, like DBSCAN, HDBSCAN, OPTICS and many more, this work shows that CLUEstering provides a new great alternative. CLUEstering is a very fast algorithm and unlike the other clustering algorithms available, it is thought and developed to run on heterogeneous computing systems and leverage many different types of accelerators, which gives it a strong advantage when working with very large datasets.

Currently, the library is available both on GitHub and on PyPi and is completely functional. There is still a lot of room for improvement, and in the future we will keep developing the library in order to improve its performance and provide more features to make it easier to use and produce better clustering results.

Appendices

Appendix A

Example of the silhouette method

```python
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_samples, silhouette_score

X, y = make_blobs(
    n_samples=500,
    n_features=2,
    centers=4,
    cluster_std=1,
    center_box=(-10.0, 10.0),
    shuffle=True,
    random_state=1,
)

range_n_clusters = [2, 3, 4, 5, 6]

for n_clusters in range_n_clusters:
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)

    ax1.set_xlim([-0.1, 1])
    ax1.set_ylim([0, len(X) + (n_clusters + 1) * 10])

    clusterer = KMeans(n_clusters=n_clusters, random_state=10)
    cluster_labels = clusterer.fit_predict(X)

    silhouette_avg = silhouette_score(X, cluster_labels)
    print(
        "For n_clusters =",
        n_clusters,
        "The average silhouette_score is :",
        silhouette_avg,
    )

    sample_silhouette_values = silhouette_samples(X, cluster_labels)

    y_lower = 10
    for i in range(n_clusters):
        ith_cluster_silhouette_values = sample_silhouette_values[cluster_labels == i]

        ith_cluster_silhouette_values.sort()

        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i

        color = cm.nipy_spectral(float(i) / n_clusters)
        ax1.fill_betweenx(
            np.arange(y_lower, y_upper),
            0,
            ith_cluster_silhouette_values,
            facecolor=color,
            edgecolor=color,
            alpha=0.7,
        )

        ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

        y_lower = y_upper + 10  # 10 for the 0 samples

    ax1.set_title("The silhouette plot for the various clusters.")
    ax1.set_xlabel("The silhouette coefficient values")
    ax1.set_ylabel("Cluster label")

    ax1.axvline(x=silhouette_avg, color="red", linestyle="--")

    ax1.set_yticks([])  # Clear the yaxis labels / ticks
    ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

    colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
    ax2.scatter(
        X[:, 0], X[:, 1], marker=".", s=30, lw=0, alpha=0.7, c=colors, edgecolor="k"
    )

    centers = clusterer.cluster_centers_
    ax2.scatter(
        centers[:, 0],
        centers[:, 1],
        marker="o",
        c="white",
        alpha=1,
        s=200,
        edgecolor="k",
    )

    for i, c in enumerate(centers):
        ax2.scatter(c[0], c[1], marker="$%d$" % i, alpha=1, s=50, edgecolor="k")

    ax2.set_title("The visualization of the clustered data.")
    ax2.set_xlabel("Feature space for the 1st feature")
    ax2.set_ylabel("Feature space for the 2nd feature")

    plt.suptitle(
        "Silhouette analysis for KMeans clustering on sample data with n_clusters = %d"
        % n_clusters,
        fontsize=14,
        fontweight="bold",
    )
    plt.savefig(f"silhouette{n_clusters}.png")
```

Appendix B

Example of the Davies-Bouldin index with kMeans clustering

```python
from sklearn.cluster import KMeans
from sklearn.metrics import davies_bouldin_score as db_score
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5), sharey=False)
ax1.set_title('Clustering data')
ax1.set_xlabel('X', fontsize=12)
ax1.set_ylabel('Y', fontsize=12)
X, y_true = make_blobs(n_samples=300, centers=4,
                       cluster_std=0.50, random_state=0)

ax1.scatter(X[:, 0], X[:, 1], s=50, c='red', alpha=.5)
max_clusters = 10
scores = []
for ncl in range(2, max_clusters):
    # K-Means
    kmeans = KMeans(n_clusters=ncl, random_state=1).fit(X)

    # store the cluster labels
    labels = kmeans.labels_
    # calculate the score
    scores.append(db_score(X, labels))

ax2.set_title('Davies Bouldin Score')
ax2.plot(range(2, max_clusters), scores, marker='o')
ax2.set_xlabel('Number of clusters', fontsize=12)
ax2.set_ylabel('Davies-Bouldin index', fontsize=12)
ax2.grid(ls='--', lw=0.5)
plt.tight_layout()
plt.savefig('db.png')
```

Appendix C

Pseudocode of the DBSCAN algorithm

Algorithm 2 Overview of the DBSCAN algorithm

```
Require: dataset, eps, minPts
  clusterId ← 0                                    ▷ Initialise the point as noise
  for point p in dataset do
      if p.label is UNCLASSIFIED then
          if ExpandCluster(dataset, p, eps, minPts) then
              clusterId ← clusterId + 1
          end if
      end if
  end for
```

Algorithm 3 ExpandCluster function of DBSCAN algorithm

```
Require: dataset, point, eps, minPts
  seeds $\leftarrow$ RegionQuery(dataset, point, eps, minPts)
  if seeds.size $\leq$ minPts then
      point.label = NOISE
      return FALSE
  end if
  seeds.delete(point)
  while seeds is not empty do
      point q $\leftarrow$ seeds.first()
      result $\leftarrow$ RegionQuery(dataset, q, eps, minPts)
      if result.size $\geq$ minPts then
          for i FROM 1 to result.size do
              point r $\leftarrow$ result.get(i)
              if r.label IN {UNCLASSIFIED, NOISE} then
                  if r.label is UNCLASSIFIED then
                      seeds.append(r)
                  end if
                  r.clusterId = clusterId
              end if
          end for
      end if
      seeds.delete(q)
  end while
  return TRUE
```

Algorithm 4 RegionQuery function of DBSCAN algorithm

```
Require: dataset, point, eps, minPts
  neighborhood N $\leftarrow$ empty set
  for point q in dataset do
      if dist(point, q) $\leq$ eps then
          N $\leftarrow$ N $\cup$ {q}
      end if
  end for
```

Appendix D

Pseudocode of the CLUE algorithm

Algorithm 5 Calculate local density

```
for $i$ in $points$ do
    $\rho_i = 0$
    for $j$ in $\Omega_{\delta_c}(i)$ do
        if $dist(i,j) < \delta_c$ then
            $\rho_i \mathrel{+}= \chi_{ij} w_j$
        end if
    end for
end for
```

Algorithm 6 Calculate nearest highers

```
for $i$ in $points$ do
    $\delta_i = +\infty$
    $\mathrm{nh}_i = -1$
    for $j$ in $\Omega_{\delta_o}(i)$ do
        if $dist(i,j) < \delta_o$ and $\rho_j > \rho_i$ then
            if $dist(i,j) < \delta_i$ then
                $\mathrm{nh}_i = j$
                $\delta_i = dist(i,j)$
            end if
        end if
    end for
end for
```

Algorithm 7 Find and assign clusters

```
$k = 0$
$stack = []$
**for** $i$ **in** $points$ **do**
    $isSeed = \rho_i > \rho_c$ **and** $\delta_i > \delta_c$
    $isOutlier = \rho_i < \rho_c$ **and** $\delta_i > \delta_c$
    **if** $isSeed$ **then**
        $\text{clusterId}_i = k$
        $++k$
        $stack.push(i)$
    **else**
        **if not** $isOutlier$ **then**
            $followers_{\text{nh}_i}.add(i)$
        **end if**
    **end if**
**end for**
**while** $stack.size > 0$ **do**
    $i = stack.back$
    $stack.pop()$
    **for** $j \in followers_i$ **do**
        $clusterId_j = clusterId_i$
        $stack.push(j)$
    **end for**
**end while**
```

Appendix E

Implementation of the tiles as a OneToMany associator

The Tiles were originally designed as nested statically-sized `VecArrays`[1]. This solution is very straightforward and efficient, but hides a problem which is related to the generality desired for this library. The size of the outer array indicates the maximum number of tiles, whereas the size of each of the innermost arrays indicates the maximum depth of each of the tiles, i.e. the maximum number of points that can be saved inside of it. This implementation has a very clear advantage: since template parameters must be specified at compile time, the size of `VecArrays` must be decided a-priori. However, it's impossible to tune them in a way which is efficient, or even functional for every possible dataset. Furthermore, allocating equal size for all the tiles implies that, since many of them are going to me nearly empty, a lot of device memory would be wasted.

A solution for solving this issue while decreasing performance as little as possible is to use a *One-To-Many associator*. This type of associator, shown in Figure E.1 is a data structure composed of two buffers: the content buffer, that contains the data we need to store which in our case is the point ids for all the tiles, and the offset buffer, which indicates the position in the content buffer of the first element of the i-th group, in our case the i-th tile.

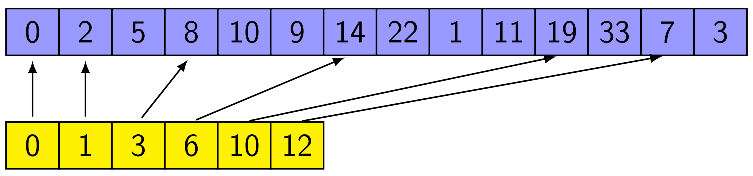


Figure E.1: Illustration of a "One to Many" associator.

[1]A VecArray is a simple static array which provides an integer attributes that keeps track of the number of elements currently used inside the array, i.e. it's dynamic size. This allows the array to be used as a dynamic array despite having a static maximum size.

Bibliography

[1] Marco Rovere et al. “CLUE: A Fast Parallel Clustering Algorithm for High Granularity Calorimeters in High-Energy Physics”. In: *Frontiers in Big Data* 3 (2020). ISSN: 2624-909X. DOI: 10.3389/fdata.2020.591315. URL: https://www.frontiersin.org/articles/10.3389/fdata.2020.591315.

[2] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data.* USA: Prentice-Hall, Inc., 1988. ISBN: 013022278X.

[3] Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise”. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining.* KDD’96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[4] Oded Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook, 2nd ed.* Jan. 2010. ISBN: 9780387098227.

[5] Charu C. Aggarwal and Chandan K. Reddy. *Data Clustering: Algorithms and Applications.* 1st. Chapman & Hall/CRC, 2013. ISBN: 1466558210.

[6] Julia Bell Hirschberg and Andrew Rosenberg. “V-Measure: A conditional entropy-based external cluster evaluation”. In: (2007). DOI: 10.7916/D80V8N84. URL: https://academiccommons.columbia.edu/doi/10.7916/D80V8N84.

[7] Peter J. Rousseeuw. “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257.

[8] Dr. Felice Pantaleo. “Computer architecture evolution and the performance challenge”. In: *Efficient Scientific Computing.* Bertinoro, Italy, 2023.

[9] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface.* ISSN. Elsevier Science, 2020. ISBN: 9780128245583. URL: https://books.google.it/books?id=e8DvDwAAQBAJ.

[10] Wikimedia Commons. *Ahmdal’s law.* 2008. URL: https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg.

[11] Ruyman Reyes et al. “SYCL 2020: More than meets the eye”. In: *Proceedings of the International Workshop on OpenCL.* IWOCL ’20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388649. URL: https://doi.org/10.1145/3388333.3388649.

[12] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: `10.1109/TPDS.2021.3097283`.

[13] Erik Zenker et al. "Alpaka - An Abstraction Library for Parallel Kernel Acceleration". In: IEEE Computer Society, May 2016. arXiv: `1602.08477`. URL: `http://arxiv.org/abs/1602.08477`.

[14] "The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment". In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08004. 361 p. DOI: `10.1088/1748-0221/3/08/S08004`. URL: `https://cds.cern.ch/record/1129810`.

[15] Andrea Bocci et al. *Heterogeneous reconstruction of tracks and primary vertices with the CMS pixel tracker.* 2020. arXiv: `2008.13461 [physics.ins-det]`.

[16] Nikolaos Andriotis et al. "Evaluating Performance Portability with the CMS Heterogeneous Pixel Reconstruction code". In: *EPJ Web of Conferences* 295 (May 2024). DOI: `10.1051/epjconf/202429511008`.

[17] Andrea Bocci et al. "Performance portability for the CMS Reconstruction with Alpaka". In: *Journal of Physics: Conference Series* 2438.1 (Feb. 2023), p. 012058. ISSN: 1742-6596. DOI: `10.1088/1742-6596/2438/1/012058`. URL: `http://dx.doi.org/10.1088/1742-6596/2438/1/012058`.

[18] Martin Winter et al. "Are dynamic memory managers on GPUs slow? a survey and benchmarks". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 219–233. ISBN: 9781450382946. DOI: `10.1145/3437801.3441612`. URL: `https://doi.org/10.1145/3437801.3441612`.

[19] Farzeen Zehra et al. "Comparative Analysis of C++ and Python in Terms of Memory and Time". In: (Dec. 2020). DOI: `10.20944/preprints202012.0516.v1`. URL: `http://dx.doi.org/10.20944/preprints202012.0516.v1`.

[20] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python.* https://github.com/pybind/pybind11. 2017.

[21] *Python Package Index (PyPI).* URL: `https://pypi.org/`.

[22] *CLUEstering, PyPi.* URL: `https://pypi.org/project/CLUEstering/`.

[23] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. "Gprof: A call graph execution profiler". In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction.* SIGPLAN '82. Boston, Massachusetts, USA: Association for Computing Machinery, 1982, pp. 120–126. ISBN: 0897910745. DOI: `10.1145/800230.806987`. URL: `https://doi.org/10.1145/800230.806987`.

[24] *Nvidia Nsight Systems.* URL: `https://developer.nvidia.com/nsight-systems`.

[25] Samuel Van Stroud et al. *Vertex Reconstruction with MaskFormers.* 2023. arXiv: `2312.12272 [hep-ex]`. URL: `https://arxiv.org/abs/2312.12272`.

[26] *Royal Holloway, University of London.* URL: https://www.royalholloway.ac.uk/research-and-teaching/departments-and-schools/physics/research/research-groups/astronomy/.

[27] Larry Bradley et al. *astropy/photutils: 1.12.0.* Version 1.12.0. Apr. 2024. DOI: 10.5281/zenodo.10967176. URL: https://doi.org/10.5281/zenodo.10967176.