Mean-Shift Algorithm

Federico Nocentini
E-mail address
federico.nocentini@stud.unifi.it

Corso Vignoli
E-mail address
corso.vignoli@stud.unifi.it

Abstract

Mean shift is a non-parametric feature-space mathematical analysis technique for locating the maxima of a density function. It has a $O(n^2)$ computational cost with a great parallel structure and so it's suitable to be parallelized. In this work two version of the algorithm are implemented, with Flat Kernel and Gaussian Kernel. Therefore an OpenMP and a CUDA implementation, with both kernel, will be presented and the execution times of each version will be compared. A particular focus will be given to the speedup obtained with the parallel versions for datasets of increasing dimension.

1. Introduction

Mean Shift is an unsupervised clustering algorithm originally presented by Fukunaga and Hostetler. Its aim is to discover blobs in a smooth density of samples. It is a centroid-based algorithm that works by updating candidates for centroids to be the mean of the points within a given region (also called bandwidth). These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids. Therefore contrary to KMeans, we don't need to choose the number of clusters nor the shape of the distribution.
Mean shift builds upon the concept of kernel density estimation (KDE), that is a method used to reconstruct the probability density function (PDF) given a set of data points. At each step a kernel function is applied to each point belonging to the dataset to shift it in the direction of the local maxima specified by the kernel. The algorithm ends when all points have reached the maxima of the underlying distribution estimated by the chosen kernel. The set of points shifted to a certain local maximum is identified as a cluster.
There are a lot of different types of kernel functions. Two of them are shown below, with σ and λ constants chosen due to the space in which the points are distributed :

- *Gaussian kernel* : $K(x) = e^{-\frac{x^2}{2\sigma^2}}$
- *Flat kernel* : $K(x) = \begin{cases} 1 & \text{if } x \leq \lambda \\ 0 & \text{if } x > \lambda \end{cases}$

The standard deviation σ is used in the *Gaussian kernel* and it represents the smoothing parameter (called **bandwidth**). It is critical because it determines the amount of smoothing: a very small value may cause the estimator to show insignificant details (many small clusters), while a very large value causes oversmoothing of the information contained in the sample (fewer but larger clusters).
The λ constant is used in the *Flat kernel* and it represents the radius of the region of interest in which the points are used for the shifting. If the value is too small, the number of iterations necessary for convergence becomes very large.

1.1. Formal Definition

The position in which each point is shifted at each step of the algorithm is computed as a weighted

average between the point and the others, where the weights are calculated with the Kernels shown before. Suppose x is a point to be shifted and $N(x)$ is the neighborhood of x, a set of points for which $K(x_i) \neq 0$. Let $dist(x; x_i)$ be the distance from the point x to the point x_i. The new position x' where x has to be shifted is computed as follows :

$$x' = \frac{\sum_{x_i \in N(x)} K(dist(x, x_i))x_i}{\sum_{x_i \in N(x)} K(dist(x, x_i))}$$

We than apply this formula to every point until convergeance: all the points have reached their corresponding local maximum of the underlying distribution. The algorithm ends when all the points have stopped shifting. The idea behind the algorithm shows that Mean Shift is embarassingly parallel, because each point can be processed distinctly from the others. This suggests that it is convenient to build a parallel version: this work presents two different parallel implementations, one with OpenMP and the other with CUDA.

2. Proposed Approach

In this paper we propose 3 different implementations of the algorithm:

1. A sequential version with C++
2. A parallel version with OpenMP
3. A parallel version with CUDA

We choose to build implementations of Mean Shift that work in 2-dimensional spaces and the Euclidean distance has been used to measure the distance between two points. The main data structures used in the algorithm are two vectors: *original_points* contains all the points in their original positions (it remains unchanged throughout the several implementations) and *shifted_points* is where new positions are stored after the shifting step.
In each implementation the function *kernel(dist, constant)* can be called with *constant*=λ (*Flat kernel*) or *constant*=σ (*Gaussian kernel*).

2.1. Sequential with C++

The proposed sequential implementation is a simple translation of the principle behind the Mean Shift in C++. The *Algorithm 1* shows the pseudocode of the algorithm and *Algorithm 2* the subroutine used to shift each single point .

Algorithm 1 Mean Shift

```
function MEANSHIFT(original_points)
    shifted_points ← original_points
    while iteration < MAX_ITERATIONS do
        for p in shifted_points do
            p ← shiftpoint(point, original_points)
```

Algorithm 2 Shift Point

```
function SHIFTPOINT(point, original_points)
    shifted ← 0
    weight ← 0
    for op in original_points do
        dist ← L2DISTANCE(point, op)
        w ← kernel(dist, constant)
        shifted ← shifted + op * w
        weight ← weight + w
    return shifted/weight
```

2.2. Parallel with OpenMP

The OpenMP library lets us transform the sequential version into a parallel one with a single *pragma* directive. The core of the implementation is the same as the sequential one, the only thing that changes is the *#pragma omp parallel*. *Algorithm 3* which allows us to compute our code in parallel on the CPU.

Algorithm 3 OpenMP Mean Shift Core

```
function MEANSHIFT(original_points)
    shifted_points ← original_points
    while iteration < MAX_ITERATIONS do
        #pragma omp for schedule(static)
        for p in shifted_points do
            p ← shiftpoint(point, original_points)
```

In the sequential and the OpenMP parallel implementation, in order to obtain the centroids, once all points shifted we use the function *getCentroids(shifted_points)* defined in *Algotithm 4* .

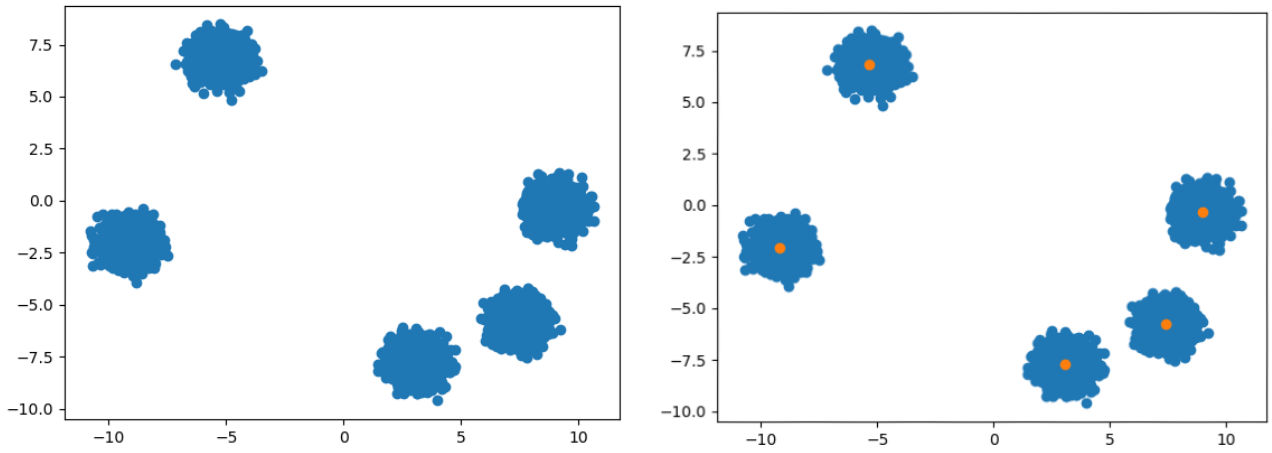

Figure 1. Example of a dataset with 10000 points respectively not clustered and clustered

Algorithm 4 Get Centroids

function GETCENTROIDS($shifted\_points$)
  while $i < N$ **do**
    $sp \leftarrow shifted\_points[i]$
    if $sp \neq 0$ **then**
      $j \leftarrow i + 1$
      while $j < N$ **do**
        $sp\_2 \leftarrow shifted\_points[j]$
        if $L_2DISTANCE(sp, sp\_2) < \epsilon$ **then**
          $shifted\_points[j] \leftarrow 0$

In order to obtain the centroids once run the algorithm, we select only the shifted points that are non-zero.

2.3. CUDA

Implementing the algorithm with CUDA lets us take advantage of the great number of cores of GPUs. Indeed, the processing of each point can be assigned to a different thread. In our implementation, the N *original_points* to be clustered are stored in 2 arrays of one dimension, which contain the features of each point, respectively:

$$p_x = [x_1, x_2, \ldots, x_N] \, , \, p_y = [y_1, y_2, \ldots, y_N]$$

The same data organization is applied to *shifted_points*:

$$s_x = [x_1, x_2, \ldots, x_N] \, , \, s_y = [y_1, y_2, \ldots, y_N]$$

The *Algorithm 5* shows the pseudocode of the core of the algorithm.

Algorithm 5 Mean Shift Core CUDA

function MEANSHIFT CORE(p_x, p_y, s_x, s_y)
  while iteration $< MAX\_ITERATIONS$ **do**
    MEANSHIFT(p_x, p_y, s_x, s_y)
    $iteration \leftarrow iteration + 1$

The *Algorithm 6* shows the pseudocode of the kernel of the algorithm.

Algorithm 6 Mean Shift CUDA

function MEANSHIFT CORE(p_x, p_y, s_x, s_y)
  $idx \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  if $idx > N$ **then**
    return
  $weight \leftarrow 0$
  $new_x \leftarrow 0$
  $new_y \leftarrow 0$
  while $i < N$ **do**
    $dist \leftarrow L_2DIST(p_x[i], p_y[i], s_x[idx], s_y[idx])$
    $w \leftarrow kernel(dist, constant)$
    $new_x \leftarrow p_x[i] * w$
    $new_y \leftarrow p_y[i] * w$
    $weight \leftarrow weight + w$
    $i \leftarrow i + 1$
  $new_x \leftarrow new_x / w$
  $new_y \leftarrow new_y / w$
  $s_x[idx] \leftarrow new_x$
  $s_y[idx] \leftarrow new_y$

In CUDA, we implemented a parallel version of the function *getCentroids(s_x, s_y)* shown in *Algorithm 7*.

Algorithm 7 Get Centroids CUDA

```
function GETCENTROIDS(s_x, s_y)
    idx ← blockIdx.x * blockDim.x + threadIdx.x
    if idx > N then
        return
    if s_x[idx] ≠ 0 and s_y[idx] ≠ 0 then
        j ← idx + 1
        while j < N do
            if L_2(s_x[idx], s_y[idx], s_x[j], s_y[j]) < ε then
                s_x[j] ← 0
                s_y[j] ← 0
```

3. Experimental Results

We have produced results in order to compare the implementations with both kernels. The metric used to compare the performances of the sequential algorithm with the OpenMP and the CUDA versions is the speedup, computed as:

$$S = \frac{t_S}{t_P}$$

where t_S and t_P are respectively the execution time of the sequential and the parallel implementation. The datasets used to evaluate the different implementations have been generated with the $make\_blob$ function of *sklearn.datasets* [1]. They are gaussian distributions with 5 centers and standard deviation equal to 0.5 and are composed by respectively 100, 1000, 10000 and 100000 2D points.
The $MAX\_ITERATIONS$ constant has been set to 10 because it has been estimated empirically that 10 iterations are enough to make all the centroids converge.
The parameters for each experiment are set to:

- $\sigma = 2$;
- $\lambda = 1$;
- $\epsilon = 0.001$;

The tests have been executed on a machine with:

- OS: Ubuntu 20.04 LTS
- CPU: Intel® Core™ i7-10710U, 6 Cores/12 Threads
- RAM: 16 GB DDR4
- GPU: GeForce GTX 1650 Mobile / Max-Q 4GB with CUDA 10.1

To make the results more reliable and representative each execution time has been obtained as the average of the times measured running each test 5 times for the sequential and the OpenMP versions and 15 times for each CUDA implementation.

3.1. OpenMP

To evaluate the performances of the OpenMP implementation, it has been executed on each dataset with 12 number of threads. We evaluated both versions with both kernels and compared the speedups with the correspondent sequential implementations.

The results for the 100, 1000, 10000 and 100000 dataset are shown respectively in *Table 1*, *Table 2*, *Table 5* and *Table 6*.

Dim	100	
Kernel	**Flat**	**Gaussian**
Sequential	0.00792 s	0.01153 s
OpenMp	0.00562 s	0.00777 s
Speedup	1.4	1.48

Table 1. Speedup for 100 points

Dim	1000	
Kernel	**Flat**	**Gaussian**
Sequential	0.7274 s	1.1186 s
OpenMp	0.1167 s	0.1745 s
Speedup	6.23	6.41

Table 2. Speedup for 1000 points

Dim	10000	
Kernel	**Flat**	**Gaussian**
Sequential	71.92 s	112.25 s
OpenMp	11.87 s	17.34 s
Speedup	6.05	6.47

Table 5. Speedup for 10000 points

Flat Kernel					
Dim	**Sequential**	**OpenMP**	**CUDA**	**OpenMP Speedup**	**CUDA Speedup**
100	0.00792 s	0.00562 s	0.00262 s	1.4	3.0
1000	0.7274 s	0.1167 s	0.0661 s	6.23	11.00
10000	71.92 s	11.87 s	3.61 s	6.05	19.87
100000	6754.34 s	1087.91 s	356.56 s	6.21	18.94

Table 3. Global comparison between sequential, OpenMP and CUDA (With *Flat kernel*) varying dataset dimension

Gaussian Kernel					
Dim	**Sequential**	**OpenMP**	**CUDA**	**OpenMP Speedup**	**CUDA Speedup**
100	0.01153 s	0.00777 s	0.00255 s	1.48	4.5
1000	1.1186 s	0.1745 s	0.0805 s	6.41	13.89
10000	112.25 s	17.34 s	3.79 s	6.47	29.56
100000	10892.23 s	1791.14 s	381.81 s	6.08	28.58

Table 4. Global comparison between sequential, OpenMP and CUDA (With *Gaussian kernel*) varying dataset dimension

Dim	100000	
Kernel	**Flat**	**Gaussian**
Sequential	6754.34 s	10892.23 s
OpenMp	1087.91 s	1791.14 s
Speedup	6.21	6.08

Table 6. Speedup for 100000 points

For the 100 points datasets (*table 1* and *table 8*), it's curious to see that the execution time is better for the *Flat kernel* but the clustering is worse with that Kernel. This is due the number of points, with only 100 points the *Flat kernel* doesn't work very well. This beheavior of bad clustering disappear with bigger datasets, indeed, the execution time is always better with the *Flat kernel*.

3.2. CUDA

To evaluate our CUDA implementation (with *Gaussian kernel*) we test the block dimension with fixed dataset dimension, 10000 observations.

Block Dim	**CUDA**
32	3.893204 s
64	**3.79634 s**
128	3.945014 s
256	4.696157 s
512	6.201023 s
1024	6.333493 s

Table 7. Execution time for 10000 observations dataset varying the block dimension (best result in bold)

To evaluate the performances of the CUDA implementation, it has been executed on each dataset like the OpenMP implementation with block dimension fixed at **64**. The results for the 100, 1000, 10000 and 100000 dataset are shown respectively in *Table 8*, *Table 9*, *Table 10* and *Table 11*.

Dim	100	
Kernel	**Flat**	**Gaussian**
Sequential	0.00792 s	0.01153 s
CUDA	0.00262 s	0.00255 s
Speedup	3.0	4.5

Table 8. Speedup for 100 points

Dim	1000	
Kernel	**Flat**	**Gaussian**
Sequential	0.7274 s	1.1186 s
CUDA	0.0661 s	0.0805 s
Speedup	11.00	13.89

Table 9. Speedup for 1000 points

Dim	10000	
Kernel	**Flat**	**Gaussian**
Sequential	71.92 s	112.25s
CUDA	3.61 s	3.79 s
Speedup	19.87	29.56

Table 10. Speedup for 10000 points

Dim	100000	
Kernel	**Flat**	**Gaussian**
Sequential	6754.34 s	10892.23 s
CUDA	356.56 s	381.81 s
Speedup	18.94	28.58

Table 11. Speedup for 100000 points

3.3. Comparison

As a final result, a global comparison has been conducted for each dataset dimension and each implementation have been considered. In table *Table 3* and in table *Table 4* we can see their comparison.
Looking at the results, we can state that the use of the *Flat kernel* makes the algorithm run faster. This was predictable since the complexity of the *Gaussian kernel* is higher than the flat one. The speedups obtained with OpenMP tend to stabilize at **6** regardless of the kernel. That is not true for CUDA speedups, because with the *Gaussian kernel* we obtain speedups around **30**, with the flat one they stabilize around **20**. From both tables we can see that the CUDA implementantion of both the Kernels abundantly outperforms both the sequential and the OpenMP ones, at the expense of a more complicated implementation. However OpenMP lets to achieve a noticeable speedup with just a single directive.

4. Conclusions

In this work, two kernel version of the Mean Shift clustering algorithm was presented in a sequential version and two parallel version: the first uses CPU parallelism and the other GPU parallelism. It was shown how its embarassingly parallel structure makes it suitable for parallel computing. A parallel implementation with OpenMP was developed by adding just a directive and it lets to obtain a speedup equal to more than **6**. Then a CUDA implementation was presented and, due to its speedup, we showed how the use of GPUs makes Mean Shift work extremely well.

References

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "" scikit-learn: Machine learning in python," journal of machine learning research, vol. 12, p," 2011.

[2] D. Demirović, "An implementation of the mean shift algorithm," *Image Processing On Line*, vol. 9, pp. 251–268, 2019.

[3] S. Węglarczyk, "Kernel density estimation and its application," in *ITM Web of Conferences*, vol. 23, p. 00037, EDP Sciences, 2018.

[4] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] C. Zeller, "Cuda c/c++ basics," *NVIDIA Coporation*, 2011.