Mean Shift Implementation

Parallel Computing Course Project

Federico Nocentini, Corso Vignoli

Supervisor: Prof. Marco Bertini

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze

Index

Introduction

Mean Shift is an unsupervised clustering algorithm originally presented by **Fukunaga** and **Hostetler** in 1975.

The computational cost is $O(n^2)$.

Unlike other clustering algorithms (e.g. KMeans) it doesn't need any kind of parameter.

It is based on **Kernel Density Estimation**.

Formal Definition

At each iteration a kernel function is applied to each point to make it shift towards the local maxima.

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

New position x' where x has to be shifted is computed as:

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

$N(x)$ is the neighborhood of x.

$dist(x, x_i)$ computes the distance between two points.

The algorithm stops when all points have stopped shifting, as they have reached the local maxima.

Proposed Approach

In this work 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.

The main data structures used in the algorithm are two vectors:

- *original_points*: contains all the points in their original positions;
- *shifted_points*: where new positions are stored after the iteration.

Sequential Implementation with C++

In each implementation the function *kernel(dist, constant)* can be called with *constant*=λ (**Flat kernel**) or *constant*=σ (**Gaussian kernel**).

The proposed sequential implementation is a simple translation of the principle behind the MeanShift in C++.

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
```

Parallel Implementation with OpenMP

The OpenMP library lets us transform the sequential version into a parallel one with a single **pragma** directive.

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)
```

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
```

Get Centroids after shifting

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

Algorithm 4 Get Centroids

```
function GETCENTROIDS(shifted_points)
    while i < N do
        sp ← shifted_points[i]
        if sp ≠ 0 then
            j ← i + 1
            while j < N do
                sp_2 ← shifted_points[j]
                if L2DISTANCE(sp, sp_2) < ε then
                    shifted_points[j] ← 0
```

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

Exploiting GPUs: CUDA

Implementing the algorithm with CUDA lets us take advantage of the great number of cores of GPUs.

The shifting 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*:

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

Parallel Implementation with CUDA

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)
    $iteration \leftarrow iteration + 1$

Algorithm 7 Get Centroids CUDA

function GETCENTROIDS(s_x, s_y)
  $idx \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  if $idx > N$ **then**
    return
  if $s_x[idx] \neq 0$ **and** $s_y[idx] \neq 0$ **then**
    $j \leftarrow idx + 1$
    while $j < N$ **do**
      if $L_2(s_x[idx], s_y[idx], s_x[j], s_y[j]) < \epsilon$ **then**
        $s_x[j] \leftarrow 0$
        $s_y[j] \leftarrow 0$

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**
    $dist \leftarrow L_2DIST(p_x[i], p_y[i], s_x[idx], s_y[idx])$
    $w \leftarrow kernel(dist, constant)$
    $new_x \leftarrow p_x[i] * w$
    $new_y \leftarrow p_y[i] * w$
    $weight \leftarrow weight + w$
    $i \leftarrow i + 1$
  $new_x \leftarrow new_x / w$
  $new_y \leftarrow new_y / w$
  $s_x[idx] \leftarrow new_x$
  $s_y[idx] \leftarrow new_y$

Experimental Results I

The performances are compared with the speedup metric, computed as:

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

The tests have been executed on a machine with:

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

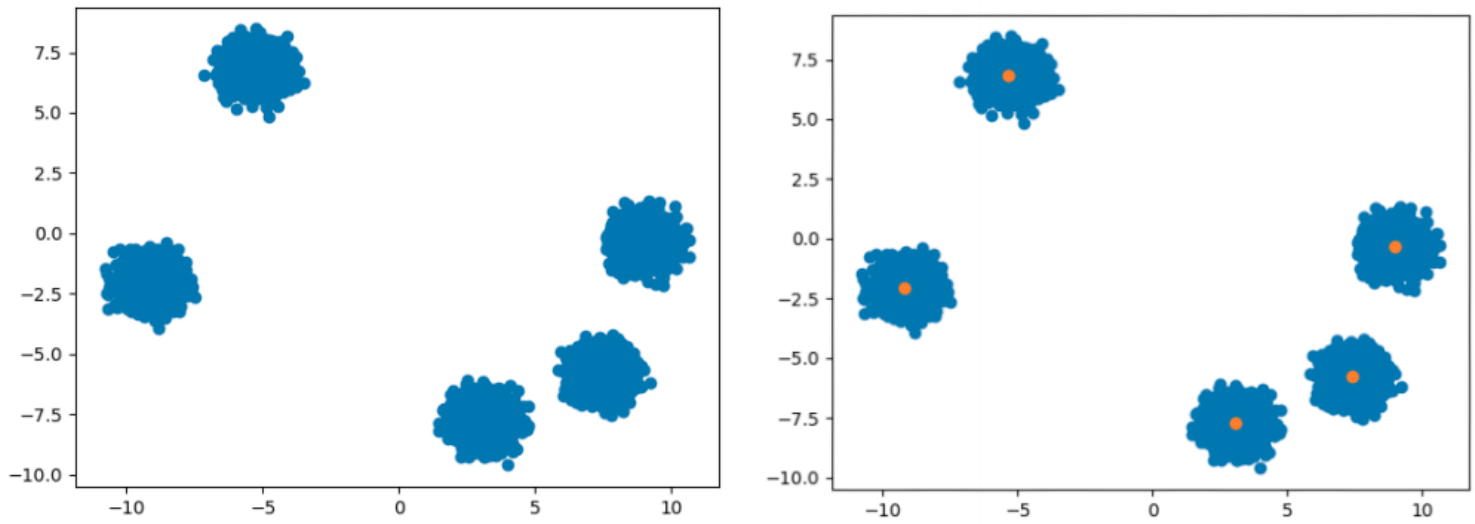

Figure 1. Example of a dataset with 10000 points respectively not clustered and clustered

Experimental Results II

The datasets used to evaluate the different implementations have been generated with the *make_blob()* function of **sklearn.datasets**.

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:

- $\sigma = 2$(**Gaussian Kernel**);
- $\lambda = 1$(**Flat Kernel**);
- $\epsilon = 0.001$.

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.

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

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

CUDA I

To evaluate our CUDA implementations 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)

CUDA II

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**.

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

Global Comparison

The implementation with Flat kernel makes the algorithm run faster.

The greatest speedups are with CUDA, at the expense of a more complicated implementation.

OpenMP lets to reach noticeable speedups with a simple implementation.

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

Conclusions

The embarrassingly parallel structure of Mean Shift makes it suitable for parallel implementations.

OpenMp has an excellent speedup and development cost ratio.

CUDA makes Mean Shift applicable to datasets intractable with a CPU.