Mean Shift clustering algorithm

Lorenzo Agnolucci

Università degli Studi di Firenze
Dipartimento di Ingegneria dell'Informazione

Firenze, 27 Aprile 2020

Outline

- Mean Shift is a non-parametric clustering algorithm
- It is based on *Kernel Density estimation*
- The only parameter is the *bandwidth*
- $O(n^2)$ computational cost
- Common application in computer vision: image segmentation
- It is embarassingly parallel

Algorithm

- At each step a kernel function is applied to each point to make it shift towards the local maxima
- Most used kernel: Gaussian kernel

$$K(x) = e^{-\frac{x^2}{2\sigma^2}} \tag{1}$$

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

$$x' = \frac{\sum_{x_i \in N(x)} K(dist(x, x_i))x_i}{\sum_{x_i \in N(x)} K(dist(x, x_i))} \tag{2}$$

$N(x)$ is the neighborhood of x, a set of points for which K(x_i) $\neq 0$

- Algorithm stops when all points have stopped shifting, that is they have reached the local maxima

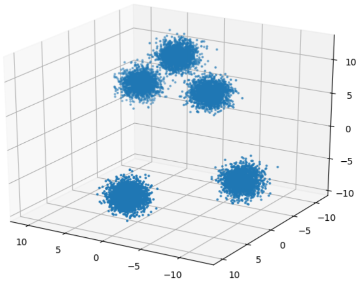
UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO
Dipartimento di Ingegneria dell'Informazione

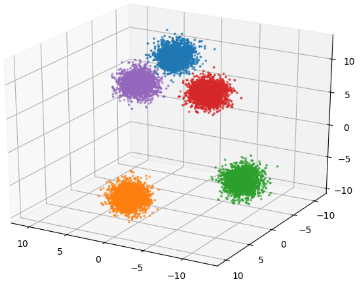
Algorithm 1 Mean shift core

function MEANSHIFT(*originalPoints*)
  shiftedPoints $\leftarrow$ *originalPoints*
  while iterationIndex $<$ *MAX_ITERATIONS* **do**
    for each point *p* in *shiftedPoints* **do**
      $p \leftarrow$ SHIFTPOINT(p, *originalPoints*)

Algorithm 2 Shift a single point

function SHIFTPOINT(p, *originalPoints*)
  shiftedP $\leftarrow 0$
  weight $\leftarrow 0$
  for each point x in *originalPoints* **do**
    $dist \leftarrow dist(p, x)$
    $w \leftarrow GKernel(dist, BW)$
    $shiftedP \leftarrow shiftedP + w * x$
    $weight \leftarrow weight + w$
  return *shiftedP* / *weight*





Algorithm 3 OpenMP Mean shift core

function OPENMPMEANSHIFT(*originalPoints*)
  shiftedPoints ← *originalPoints*
  while iterationIndex < *MAX_ITERATIONS* **do**
    #pragma parallel for schedule(static)
    for each point *p* in *shiftedPoints* **do**
      p ← SHIFTPOINT(*p*, *originalPoints*)

Algorithm 4 Shift a single point

function SHIFTPOINT(*p*, *originalPoints*)
  shiftedP ← 0
  weight ← 0
  for each point *x* in *originalPoints* **do**
    dist ← *dist*(*p*, *x*)
    w ← *GKernel*(*dist*, *BW*)
    shiftedP ← *shiftedP* + *w* ∗ *x*
    weight ← *weight* + *w*
  return *shiftedP* / *weight*

- just a pragma directive
- static scheduling: loop divided statically in chunks of equal size

Taking advantage of GPUs:

- One thread for each point to shift
- Coalescing of accesses to memory:
  - Points stored as a *Structure of Arrays*

$$[x_1, \ldots, x_n, y_1, \ldots, y_n, z_1, \ldots, z_n] \tag{3}$$

  - Access to the array in the form of:

$$blockDim.x * blockIdx.x + threadIdx.x \tag{4}$$

Algorithm 5 CUDA Naive version Mean Shift core

function NAIVECUDAMS(*originalPoints*)
  shiftedPoints $\leftarrow$ *originalPoints*
  while iterationIndex $<$ *MAX_ITERATIONS* **do**
    NAIVEKERNEL(*shiftedPoints*, *originalPoints*)

Algorithm 6 CUDA Naive version Kernel

function NAIVEKERNEL(*shiftedPts*, *originalPts*)
  $tx \leftarrow threadIdx.x$
  $bx \leftarrow blockIdx.x$
  $idx \leftarrow bx * blockDim.x + tx$
  if $idx < |originalPts|$ **then**
    $shiftedP \leftarrow 0$
    $weight \leftarrow 0$
    $p \leftarrow shiftedPts[idx]$
    for each point x in *originalPts* **do**
      $dist \leftarrow dist(p, x)$
      $w \leftarrow GKernel(dist, BW)$
      $shiftedP \leftarrow shiftedP + w * x$
      $weight \leftarrow weight + w$
    $shiftedPts[idx] \leftarrow shiftedP / weight$

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO Dipartimento di Ingegneria dell'Informazione

A further optimization is possible:

- Each thread reads *O(n)* points from global memory to compute the shift
- Shared Memory can be exploited with the Tiling pattern
- *TILE_WIDTH = BLOCK_DIM*
- *O(n)* accesses reduced to *O(n/TILE_WIDTH)*

Algorithm 7 CUDA Tiling version Mean Shift core

```
function TilingCUDAMS(originalPoints)
    shiftedPoints ← originalPoints
    while iterationIndex < MAX_ITERATIONS do
        TilingKernel(shiftedPoints, originalPoints)
```

Algorithm 8 CUDA Tiling version Kernel

```
function TilingKernel(shiftedPts, originalPts)
    tx ← threadIdx.x
    bx ← blockIdx.x
    idx ← bx * blockDim.x + tx
    tile ← SharedMemArray[TILE_WIDTH]
    shiftedP ← 0
    weight ← 0
    for tileIter < numTiles do
        tileIdx ← tileIter * TILE_WIDTH + tx
        if tileIdx < |originalPts| then
            tile[tx] ← originalPts[tileIdx]
        else
            tile[tx] ← nullPoint
        __synchthreads()                                ▷ End of loading
        if idx < |originalPts| then
            p ← shiftedPts[idx]
            for i with i < TILE_WIDTH do
                x ← tile[i]
                if x! = nullPoint then
                    dist ← dist(p, x)
                    w ← GKernel(dist, BW)
                    shiftedP ← shiftedP + w * x
                    weight ← weight + w
        __synchthreads()                                ▷ End of computing
    if idx < |originalPts| then
        shiftedPts[idx] ← shiftedP / weight
```

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO Dipartimento di Ingegneria dell'Informazione

- Performances compared with the speedup metric, computed as:

$$S = \frac{t_S}{t_P} \tag{5}$$

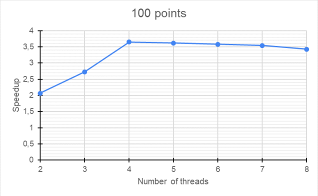
- Tests executed on a machine with:
  - OS: Ubuntu 18.04 LTS
  - CPU: Intel Core i7-8565U 1.8GHz up to 4.6GHz with Turbo Boost, 4 cores/8 threads
  - RAM: 16 GB DDR4
  - GPU: NVidia GeForce MX250 2GB with CUDA 10.1

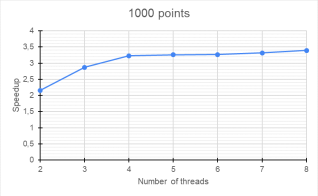
- Each time is the average of 5 experiments for the sequential and OpenMP versions and of 15 experiments for both CUDA implementations

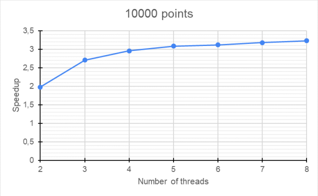
Experimental results (cont.)

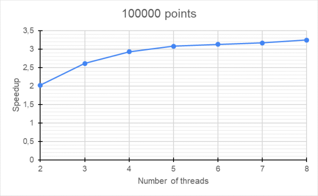
- The implementations have been evaluated on gaussian distributions composed by respectively 100, 1000, 10000, 100000 and 250000 3D points
- bandwidth set to 2
- *MAX_ITERATIONS* constant set to 10, which has been empirically estimated to be enough to make all points converge to the local maxima

Number of threads gradually increased for each dataset



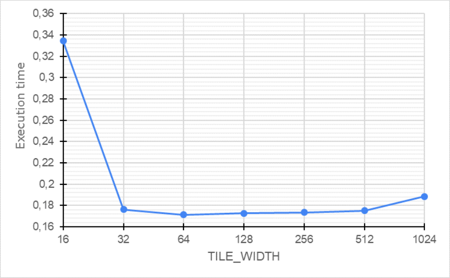






Not tested on 250k points dataset: too long execution time

Execution time for 10000 points for the CUDA tiling implementation varying *TILE_WIDTH*



TILE_WIDTH	CUDA Tiling
16	0.334405 s
32	0.176171 s
64	**0.171354 s**
128	0.172666 s
256	0.173306 s
512	0.175091 s
1024	0.18845 s

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO Dipartimento di Ingegneria dell'Informazione

Dim	CUDA Naive	CUDA Tiling	Speedup
100	0.000401 s	0.000437 s	1.09
1000	0.003044 s	0.003359 s	1.11
10000	0.171354 s	0.191216 s	1.12
100000	15.79 s	18.29 s	1.16
250000	100.38 s	121.10 s	1.20

Tiling CUDA speedup

Dim	Sequential	CUDA Tiling	Speedup
100	0.005384 s	0.000401 s	13.43
1000	0.475485 s	0.003044 s	156.21
10000	49.29 s	0.171354 s	287.64
100000	4560.37 s	15.79 s	288.80
250000	† 27768 s	100.38 s	† 276.61

- † time has been estimated with a quadratic regression (due to the $O(n^2)$ computational cost)

Global comparison between sequential, OpenMP and Tiling CUDA best results:

- Greatest speedups with CUDA, at the expense of a more complicated implementation
- OpenMP lets to reach noticeable speedups with a simple implementation (just a directive)

Dim	Sequential	OpenMP	OpenMP Speedup	CUDA Tiling	CUDA Speedup
100	0.005384 s	0.001473 s	3.66	0.000401 s	13.43
1000	0.475485 s	0.140161 s	3.39	0.003044 s	156.21
10000	49.29 s	15.25 s	3.24	0.171354 s	287.64
100000	4560.37 s	1403.78 s	3.25	15.79 s	288.80
250000	† 27768 s	† 8720 s	† 3.18	100.38 s	† 276.61

Table: † times have been estimated with a quadratic regression

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO Dipartimento di Ingegneria dell'Informazione

- The embarassingly 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