# PC-2018/19 Course Project: Clustering with Mean Shift, a GPU implementation

Federico Vaccaro
federico.vaccaro@stud.unifi.it

## Abstract

*The Mean Shift algorithm is a simple yet elegant clustering technique; however due its computational complexity, often other algorithms are preferred, specially on big datasets. Despite this, Mean Shift is embarassingly parallel, reason why it fits perfeclty to GPU architectures, in a such way that the execution times are definitely less onerous, and the Mean Shift become a feasible way to perform clustering for large datasets.*

## 1. Introduction to the Mean Shift algorithm

### 1.1. Comparison between probability and clustering

Mean Shift is an iterative algorithm to perform clustering, introduced by Fukunaga and Hostetler [3], which idea is based on the *Kernel Density Estimation*. The KDE is a non parametric technique to estimate the underlying PDF to the probability distribution which generated the dataset. The KDE idea is exploited in this way: for each iteration, for each point belonging to the dataset, we apply a *shift* to the point toward the direction of the nearest *peak* of the KDE surface (*i.e.* the mode) following the gradient direction. This function, will show some peaks in correspondence of the regions with the major density of points. We identify these regions with the clusters, and their peaks with the centroids. The unique parameter of the Mean Shift is the *bandwidth*, which controls the smoothing of the theorical PDF: if a lower bandwidth models a noiser function, this will have more local maxima, *i.e.* more clusters. So the main advantage of the Mean Shift is that it isn't needed to specify the cluster numbers, or to be initialized, like the more popular *K-Means*: the only knob is the bandwidth value. Below, is reported the pseudocode about how the single iteration of the Mean Shift works.

It's clear that the **Algorithm 1** has $O(n^2)$ complexity cost, where $n$ is the dataset $X$ dimension, because for each $\mathbf{x_i}$, a shift $\mathbf{m_i}$ is computed, computing that cost $O(n)$ (**Algorithm 2**). $Y$ is the shifted point set. **Algorithm 1** is simply the pseudo-code about a single iteration, so the cost

**Algorithm 1** single Mean Shift iteration

```
procedure ShiftPoints($Y_t, X, bandwidth$)
    for all $\mathbf{y_i}, i = 1, ..., |Y_t|$ do
        $\mathbf{m_i} \leftarrow$ ComputeShift($\mathbf{y_i}, X, bandwidth$)
        $\mathbf{y_i} = \mathbf{x_i} + \mathbf{m_i}$
    return the set $Y_{t+1} \leftarrow \{\mathbf{y_i}, i = 1, ..., |Y_t|\}$
```

**Algorithm 2** Shift computing algorithm

```
procedure ComputeShift($\mathbf{y_i}, X, bandwidth$)
    $weights \leftarrow 0$
    $\mathbf{m_i} \leftarrow \mathbf{0}$
    for all $\mathbf{x_j}, j = 1, ..., |X|$ do
        $w_j \leftarrow K(\frac{||\mathbf{y_i} - \mathbf{x_j}||}{bandwidth})$
        $\mathbf{m_i} \leftarrow \mathbf{m_i} + w_j \mathbf{x_j}$
        $weights \leftarrow weights + w_j$
    return $\mathbf{m_i}/weights - \mathbf{y_i}$
```

**Algorithm 3** Mean Shift algorithm

```
procedure MeanShift($X, n, bandwidth, MaxIterations$)
    $t \leftarrow 0$
    $Y_0 \leftarrow X$
    while $t < MaxIterations$ do
        $Y_{t+1} =$ ShiftPoints($Y_t, X, bandwidth$)
        Swap($Y_t, Y_{t+1}$)
        $t \leftarrow t + 1$
    return $Y_{t+1}$
```

of $T$ iterations is $O(Tn^2)$. In the **Algorithm 1**, $X$ represents the original dataset, and $Y_{t+1}$ the shifted point set.

### 1.2. The Kernel Function

**Algorithm 1** is reported in a manner such that highlights the computational cost, however some mathematical details misses about the *kernel function* $K(r)$ and the shift $\mathbf{m_i}$. The *mean shift* formula is:

$$\mathbf{m_i} = m(\mathbf{y_i}) = \frac{\sum_{j=1}^{n} K(\frac{||\mathbf{y_i} - \mathbf{x_j}||}{bandwidth})\mathbf{x_j}}{\sum_{j=1}^{n} K(\frac{||\mathbf{y_i} - \mathbf{x_j}||}{bandwidth})} - \mathbf{y_i} \tag{1}$$

$$= \frac{\sum_{j=1}^{n} w_{ij}\mathbf{x_j}}{\sum_{j=1}^{n} w_{ij}} - \mathbf{y_i} = \widetilde{\mathbf{m}}_\mathbf{i} - \mathbf{y_i} \tag{2}$$

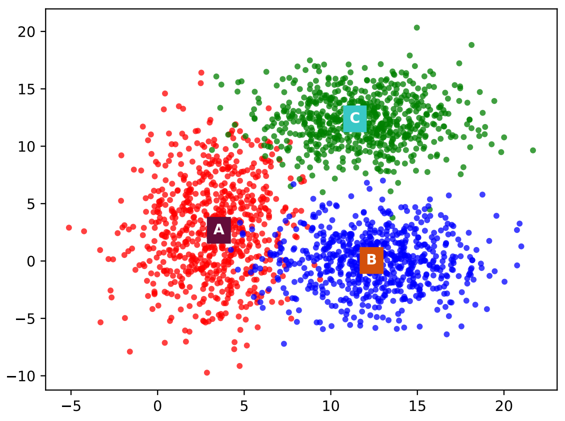

Figure 1. Visualization of the dataset. The points A, B e C are the centroids found by parallel Mean Shift. The centroids correspond with the gaussians centers $\mu$, which generated the dataset.

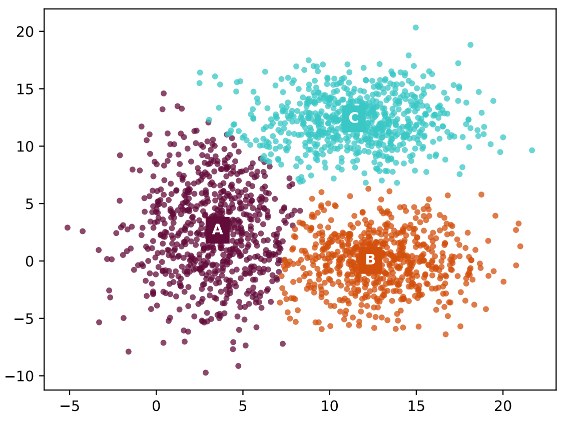

Figure 2. How points are actually clustered by Mean Shift

It's easy to see that the shift is a vector computed by a weighted average between $\mathbf{y_i}$ and each other $\mathbf{x_j}$: the nearest points will have an higher weight, so the points belonging to a cluster (*i.e.* densest regions) converges to their centroid/local maximum. The most common *kernel* having this property and able to guarantee the convergence is the **gaussian kernel**:

$$K(r) = \frac{1}{(2\pi bandwidth)^{k/2}} e^{-\frac{1}{2}\frac{r^2}{bandwidth^k}} \tag{3}$$

Watching to the (2), it's sufficient in the **Algorithm 1** assign $\mathbf{y_i} \leftarrow \widetilde{\mathbf{m}}_\mathbf{i}$. The discussed implementation is specific for $k = 2$ or $k = 3$, so we won't be worry about large $k$ but we'll focus on the dataset dimension. Also, $k = 3$ is a common case, because in Computer Vision, the Mean Shift is used to perform *Image Segmentation* [1] or *Object Tracking* [9].

## 2. The implementation

The procedure of the **Algorithm 1** is *embarassingly parallel*: each point $y_i \in Y_t$ can be distinclty processed. Let's consider two arrays $Y_t$ and $Y_{t+1}$, allocated in memory, plus $X$ the (read-only) array of the original dataset: the first is where the input is read, the second where the output is written. If we assign to each thread an index $i : 1, .., n$, during the execution of *ComputeShift*, each element of $Y_t$ isn't modified, but only read; each thread have its own local variable $\mathbf{m_i}$ (computed through a read-only procedure) and at the end of the execution each thread write in main memory the variable $\mathbf{y_i}$ in $Y_{t+1}$. This approach respects the *Bernstein's conditions*, because the thread have not to share any data and the points can be processed in any order. Of course, there must be a *thread barrier* at the end of each iteration, because of the iterative nature of the algorithm. Below, is reported the pseudo code of the entire algorithm in the **Algorithm 4**.

**Algorithm 4** Mean Shift algorithm, parallel version

```
procedure PARALLELMEANSHIFT($X, n, bandwidth, MaxIterations$)
    $t \leftarrow 0$
    $Y_0 \leftarrow X$
    while $t < MaxIterations$ do
        $Y_{t+1} = $ PARALLELSHIFTPOINTS($Y_t, X, bandwidth$)
        WAITTHREADSFORFINISH()
        SWAP($Y_t, Y_{t+1}$)
        $t \leftarrow t + 1$
    return $Y_{t+1}$
```

### 2.1. GPU, memory coalescing and divergence

In this section will be discussed some details about the GPU implementation of the algorithm, how it fits these architectures efficiently and how the naive implementation can be improved.

One of the disadvantages of the CPUs over the GPUs, is that they suffer from the context switching cost when they are exposed to an high number of active threads, number too much superior to the cores count. Conversely, the GPUs, having a larger number of *register files* and hardware scheduler, they're not just able to manage an high quantity of threads, but this practice is actually encouraged to flavour the *memory latency hiding* [1]. So, given a dataset $X$, it's a

[1] *i.e.* while a thread group - *warp* in the CUDA glossary - is idling waiting for the I/O, another warp goes running, keeping employed the Cuda cores

natural choice start a number $|X| = n$ of threads and assign the processing of each element to them.
Since now, the used terms will be from the CUDA terminology [8]. In the GPU, threads are organized in *blocks*; the threads composing each block, are executed in parallel (by the **SIMT** paradigm) in group of 32 named *warps*. It's a good rule keep the block size multiple of this number. Then, in this implemantation, `BLOCK_DIM = 64` was experimentally a good choice. The number of blocks is $\lceil n/$ `BLOCK_DIM` $\rceil$. Note that the points are stored in memory in the *Array Of Structure* shape:

$$X = [x_1, y_1, x_2, y_2, ..., x_n, y_n] \tag{4}$$

In the examinated case $k = 2$, so isn't necessary use a *Structure of Array*. Note that the formula to access to the $i$-th point in the Array is:

$$index = (BlockDim * BlockIdx + tx) * k \tag{5}$$

Where $BlockDim$ is equal `BLOCK_DIM`, and $BlockIdx$ is the id within the *grid* which the thread $tx$ belongs, and $tx$ its identificatore inside the block. The formula depends linearly from $tx$, for that reason thread belonging to the same block will access close memory addresses. The access to the VRAM is *burst*, meaning that a sequence of datas adjacent in memory will be loaded from main memory to the GPU caches/registers: for this reason, with a single access will be satisfyied at the same time more requests to the VRAM by multiple thread, because they will try to access to adjacent addresses. This access pattern to the ram is said *coalesced access* [4]. This approach diminish the I/Os, and the time spent idling by the warps. The last aspect to consider is the *divergence* [2]: GPUs don't have at their disposal *speculative hardware* and the threads inside a warp are all at the same point of execution; for this reason if the execution flow *diverges*, *i.e.* a conditional is encounterend so that thread within a warp have to follow a different flow, these threads are splitted from their original warp [2], so they will be put in a undersized warp with the divergent threads: some thread will be unavoidably in idle. The Mean Shift algorithm, fits well even here the GPU architectures because it doesn't have critical conditional instructions: these are there just to avoid that the last block's threads write/read from *Out of Range Memory Locations*.

## 2.2. Shared Memory and Tiling

A further optimization to limit the memory accesses is the usage of the *Shared Memory* [5]. The Shared Memory can be understood as a *programmable cache*: it's a memory inside the GPU, characterized by a little dimension and very-low latency. The key idea is to write on it those datas being read several times from main memory, and, once copyed in the Shared Memory, make the threads read from it. Getting back to the algorithm, the shift is computed in function of every other points in the dataset, so each thread needs to access $O(n)$ times in central memory, one for each point, resulting in a lot of total access. The pattern through the Shared Memory is exploited is called *Tiling* [6]. Each thread of the block, load first into the Shared Memory the point to which it has been assigned. At the end of the loading phase[3], it shows up a kind of collaboration between threads: these make a partial computing of the mean shift based upon what is in the shared memory. Once that each thread has computed its partial mean shift, they load a point from the next tile to the shared memory until the entire pointset has been "seen". Let `TILE_WIDTH = BLOCK_DIM` being the number of points contained for a block in Shared Memory; each thread reduce its number of reading from the main memory from $O(n)$ to $O(n/$ `TILE_WIDTH` $)$. This will results in more time spent busy from the threads. Clearly `TILE_WIDTH` has an upper bound given the narrowness of the Shared Memory quantity *(48KB per block, 96KB per SM on the Pascal architecture [7])*. Is reported below the code of the Tiling-variant of the Mean Shift.

**Algorithm 5** Parallel Mean Shift Iteration w/ Tiling

```
procedure PARALLELMS($X, Y_t, Y_{t+1}, bandwidth$)
    $i = (BlockDim * BlockIdx + tx) * k$
    $\mathbf{m_i}, w \leftarrow 0$
    $\mathbf{y_i} \leftarrow Y_t[i]$
    $CurrentTile \leftarrow 0$
    while $CurrentTile < TotalTiles$ do
        $SMIndex \leftarrow Index(CurrentTile, tx)$
        $SharedMem[tx] \leftarrow X[SMIndex]$
        WAITTHREADSFORLOADING()
        $\mathbf{m_i}, w \leftarrow$ PARTIALSHIFT($SharedMem, y_i, \mathbf{m_i}, w$)
        WAITTHREADSFORCOMPUTING()
        $CurrentTile \leftarrow CurrentTile + 1$
    $\mathbf{y_i} \leftarrow \mathbf{m_i}/w$
```

It can be noticed that the code gets more complex; now isn't enoughg a single **for** cycle to compute the shift $\mathbf{m_i}$, but it's necessary to insert an outer loop, cycling over each Tile. Also it must be mantained in memory two indices: the first is $index$ and it refers to the element that the thread is shifting; the other is $SMIndex$ and it refers to the element that the threads load in Shared Memory from the $CurrentTile$. CurrentTile is computed by this formula:

[2]Actually, the CUDA compiler is quite aggressive towards the branching: it often replace the conditional with a tool called *predication*. As Robert Crovella says: *it's impossible to say whether either realization (*i.e. *divergence or predication) would lead to truly divergent code or just predicated code (or both).* [2]

[3]n.b., it's necessary to put here a barrier at block level, to avoid that some threads attempt to access to shared memory location not assigned yet

**Algorithm 6** Shift computing algorithm w/ shared memory

```
procedure PARTIALSHIFT($SharedMem, \mathbf{y_i}, \mathbf{m_i}, w, bandwid$
    for $j = 1, ...,$ TILE_WIDTH do
        $\mathbf{x_j} \leftarrow SharedMem[j]$
        $w_j \leftarrow K(\frac{||\mathbf{y_i} - \mathbf{x_j}||}{bandwidth})$
        $\mathbf{m_i} \leftarrow \mathbf{m_i} + w_j \mathbf{x_j}$
        $w \leftarrow weights + w_j$
    return $\mathbf{m_i}, w$
```

| Dim | CPU Time | GPU Time | SpeedUp |
|---|---|---|---|
| 500 | 0.095s | 0.00031s | **303x** |
| 5000 | 7.43s | 0.0032s | **2275x** |
| 50000 | 945.73s | 0.085s | **11042x** |
| 500000 | $\sim$ 96813s | 8.75507s | **11057x** |

Table 1. Times measured and SpeedUp obtained by the GPU implementation, varying the dataset dimension.

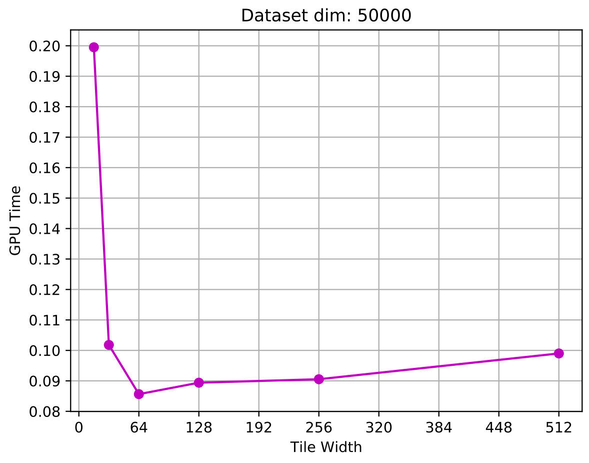

Figure 3. GPU Time varying the tile width.

$$SMIndex \leftarrow \tag{6}$$

$$(CurrentTile * \texttt{TILE\_WIDTH} + tx) * k \tag{7}$$

## 3. Experimental results

Various experiments have been made to compare first of all the SpeedUp of the parallel implementation respect to the sequential version, written in C++. Each time has been measured running each test 5 times for the sequential implementation and 20 for the gpu one and averaging them. Also, has been analyzed the scaling of the algorithm respect to the dataset dimension and the performance gain using different tile dimension. The SpeedUp $S_P$ is calculated as:

$$S_P = \frac{t_s}{t_p} \tag{8}$$

The experiments have been conducted on a machine equipped with: The experiments have been conducted on Ubuntu 16.04 LTS, on a machine equipped with

- Intel Core i7 8700k @ 4ghz 6 core/12 threads processor
- RAM 16 GB DDR4 3200Mhz C14
- NVidia GTX 1080 Ti 11GB (running on CUDA 10)

The dataset has been artificially generated sampling by three gaussian distributions. The number of iteration to the convergence of the algorithm depends by $bandwidth$ value, however has been run a fixed number $T = 15$ of iteration.

By the **Table 1**, can be noticed how at the increase of the dataset dimension, it increases the *speed up* factor until it converges to a factor 11000x. This can be easily explained because by increasing the dataset dimension, it increase number of the threads spawned on the GPU. The employing of the CPU become practically impossible *(almost 27h of execution)* when the dataset dimension reach 500,000 elements. The CPU Time hasn't been measured, but it has been estimated by doing a quadratic regression (because of the $O(n^2)$ computational cost) on the previous experiment results.

| Dim | non SM Time | SM Time | SpeedUp |
|---|---|---|---|
| 500 | 0.00061s | 0.00031s | **1,96x** |
| 5000 | 0.0057s | 0.0032s | **1,75x** |
| 50000 | 0.15s | 0.085s | **1,78x** |
| 500000 | 15.66s | 8.75s | **1,78x** |

Table 2. Times and SpeedUp obtained with the Shared Memory implementation respect to the naive one.

The **Table 2** reports the SpeedUp obtained by using the SharedMemory, with `TILE_WIDTH = 64`. How can be observed, the speed up is almost constant respect to the dataset dimension, except for the case when $Dim = 500$. It can be explained because the shared memory of each block capable of containing a relevant portion of the dataset.

The plot in **Figure 3** shows the spent time at the varying of the tile width. The minimum time is obtained with `TILE_WIDTH = 64`. This value has been used for the rest of the experiments. From a certain value, increasing the tile width don't yield any improvement because the overall Shared Memory is available in limited quantity. Must be noticed that when the `TILE_WIDTH = 16`, the time is doubled, because there is only one undersized warp per block.

## 4. Conclusions

The Mean Shift isn't usually the first choice about clustering algorithms, because of its computational cost $O(n^2)$. However, it's been discussed how the problem fits well to the GPU architectures and has a very natural parallel implementation, and how it can be improved. Thanks to this devices, now it's possible to process huge dataset in very short times without the need of a supercomputer. For the Mean Shift, the GPGPU computing appears heavily faster and more appealing than the traditional CPU processing.

### 4.1. Resources

Link to the discussed CUDA implementation on GitHub `https://github.com/fede-vaccaro/CUDA_MeanShift`
Link to the sequential version `https://github.com/fede-vaccaro/MeanShift_CPP`

## References

[1] D. Comaniciu and P. Meer. Mean shift analysis and applications. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1197–1203 vol.2, Sep. 1999.

[2] R. Crovella. Avoiding thread divergence. `https://devtalk.nvidia.com/default/topic/795405/avoiding-thread-divergence/`, 2014. [Online; accessed 16-01-2019].

[3] K. Fukunaga and L. D. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Trans. Information Theory*, 21:32–40, 1975.

[4] M. Harris. How to access global memory efficiently in cuda c/c++ kernels. 2013. [Online; accessed 16-01-2019].

[5] M. Harris. Using shared memory in cuda c/c++. 2013. [Online; accessed 16-01-2019].

[6] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[7] NVIDIA. Pascal tuning guide. 2018. [Online; accessed 16-01-2019].

[8] NVIDIA. Cuda documentation. `https://docs.nvidia.com/cuda/`, 2019. [Online; accessed 16-01-2019].

[9] Z. qiang Wen and Z. Cai. Mean shift algorithm and its application in tracking of objects. *2006 International Conference on Machine Learning and Cybernetics*, pages 4024–4028, 2006.