



# MeanShift algorithm: a GPU implementation with CUDA

Parallel computing Course  
Mid-term project

Lorenzo Pisaneschi

Firenze, 30 gennaio 2020



# The Mean Shift algorithm

- Mean Shift is a non parametric clustering algorithm
- It can **classify** dataset points, clustering them into groups without knowing clusters number
- It is based on KDE (Kernel Density Estimation) which calculates **peaks** and **clusters** consequently
- At each step points will be shifted to their nearest peak





# Kernel Density Estimation

- KDE is a method to estimate the underlying distribution (the probability density function) for a set of data.
- Given an  $N$  points dataset, at each iteration a point will be processed by a **Kernel function**
- This operation will generate a probability surface according to which each point will be clustered





# The Gaussian Kernel

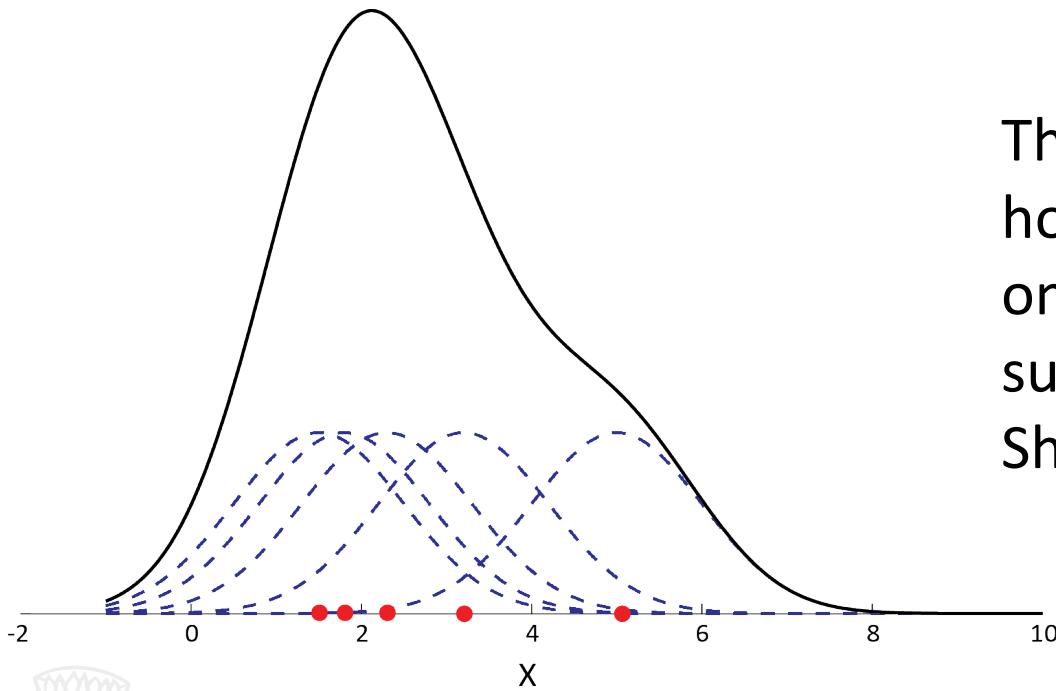
The most popular kernel function is the **Gaussian Kernel**:

$$k(x) = e^{\frac{-x}{2\sigma^2}}$$

Where:

- $x$  is the d – dimensional point to be processed by the kernel
- $\sigma$  is the variance. We also call  $\sigma$  **band-width**
- The bandwidth is the only one input parameter for the Mean Shift algorithm

# The Gaussian Kernel - example



This is an 1-d example on how each point contributes on generate the probability surface during the Mean Shift computation



# MeanShift – Further details [1]

- At each iteration, each point is processed by a Shift function
- The *Shift* operation works using the kernel function  $k(x)$  and the **bandwith** parameter
- This operation computes ***means*** and ***weights*** shifting points gradually closer to their clusters





## MeanShift – Further details [2]

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x - x_i)x_i}{\sum_{x_i \in N(x)} K(x - x_i)}$$

Is the calculated estimation of the weighted mean of the density determined by  $K$ ,  $m(x)$

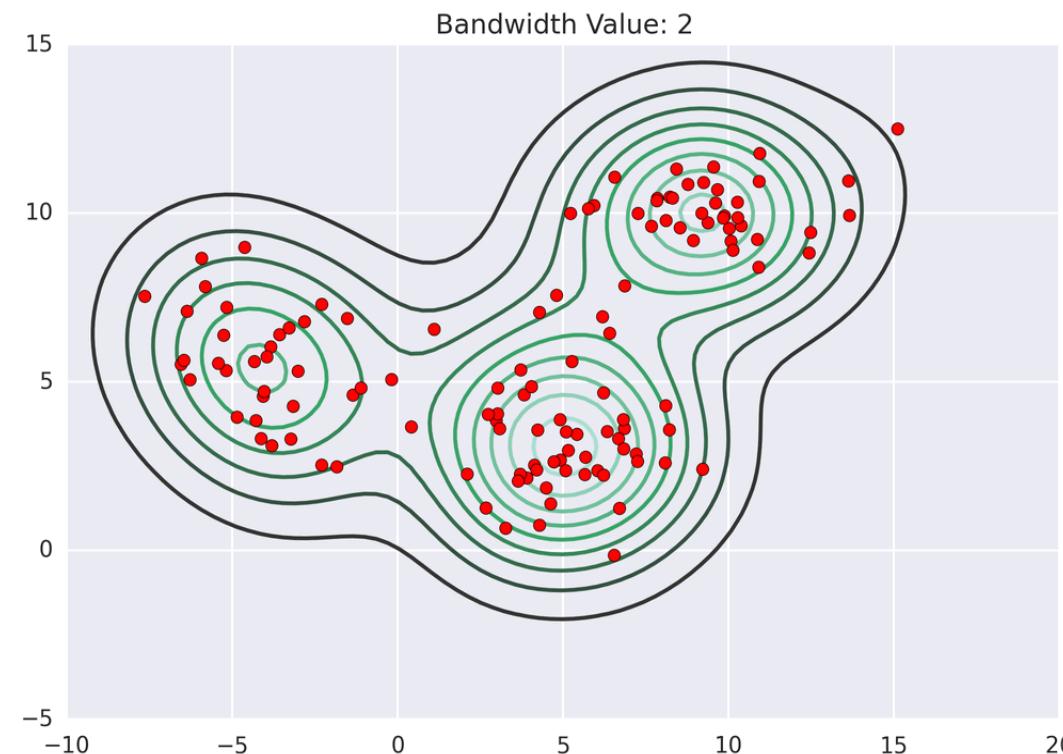
$$m(x) - x$$

Is the mean shift

$$x \leftarrow m(x)$$

Is the provided algorithm updated until there is no convergence or max iteration limit is reached

# MeanShift execution



- N d-dimensional point
- Bandwidth value
- Centroids updating using the kernel function
- Keep going until procedure did not converges



# MeanShift – Analysis

- Mean shift is an expensive algorithm:  $O(Tn^2)$
- Where:
  - $n$ : dataset dimension ( $n$  d-dimensional points)
  - $T$ : max iterations number
- In a traditional CPU implementation, this algorithm could be very slow
- On the other hand, each point could be processed independently by the others



# Parallel MeanShift with CUDA

```
procedure ParallelMeanShift( $X$ ,  $n$ , bandwidth, Iterations)
 $it \leftarrow 0$ 
 $Y_0 \leftarrow X$ 
while  $t < Iterations$  do
     $Y_{t+1} = \text{ParallelShift}(Y_t, X, bandwidth)$ 
    SynchronizeThreads()
    Swap( $Y_t, Y_{t+1}$ )
     $it \leftarrow it + 1$ 
return  $Y_{t+1}$ 
```





# Parallel MeanShift with CUDA

***procedure ParallelMeanShift((X, n, bandwidth, Iterations)***

*it*  $\leftarrow 0$

*Y*<sub>0</sub>  $\leftarrow X$

***while t < Iterations do***

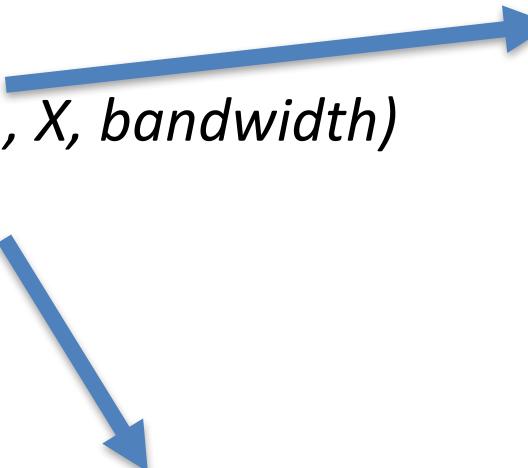
*Y* <sub>$t+1$</sub>  = ***ParallelShift(Y<sub>t</sub>, X, bandwidth)***

***SynchronizeThreads()***

***Swap(Y<sub>t</sub>, Y <sub>$t+1$</sub> )***

*it*  $\leftarrow it + 1$

***return Y <sub>$t+1$</sub>***



Each mean will  
be computed  
thanks to point  
independent  
processing

All threads work  
must to be  
synchronized!!!





# Parallel MeanShift with CUDA

- Since each point could be processed individually, MeanShift can be implemented using **parallel programming**
- Furthermore, it could be implemented exploiting **GPU power**
- In a traditional CPU implementation, this algorithm could be very slow
- CUDA (Compute Unified Device Architecture) is perfect in order to reach this goal
- Memory management: **thrust library**

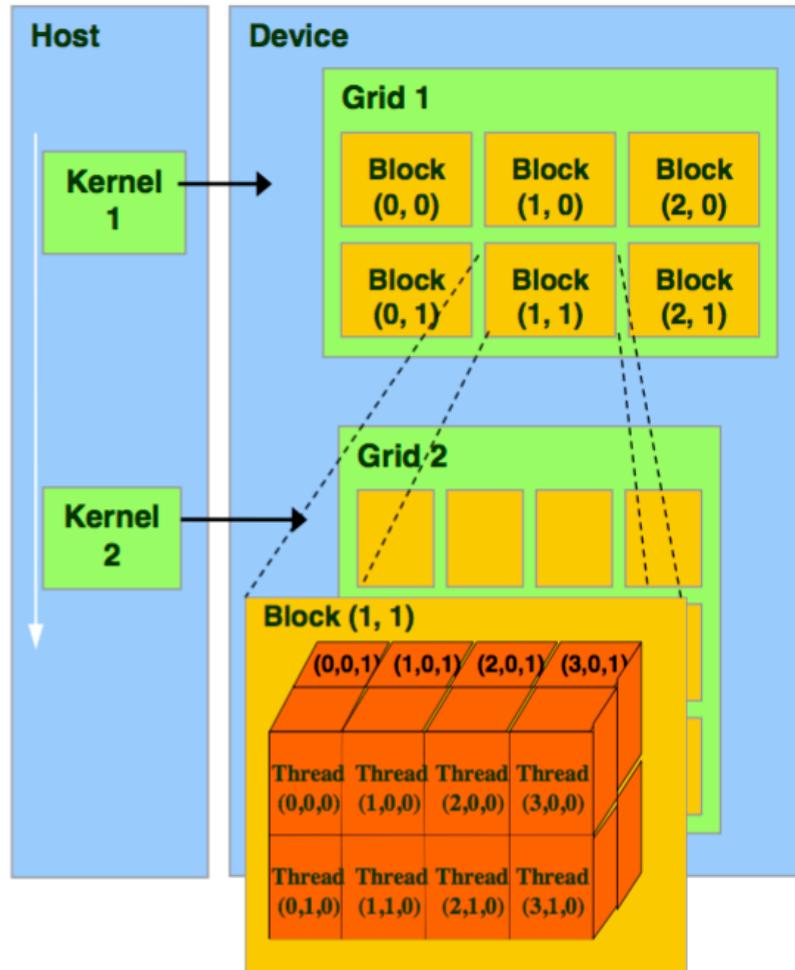


# CUDA SIMT paradigm

- **SIMT: Single Instruction Multiple Threads**
- It differs from SIMD (Single instruction Multiple Data) because in SIMT, threads can be executed asynchronously and independently
- In CUDA SIMD paradigm is ensured by the fact that threads are concurrently scheduled entering the same program



# CUDA: GPU organization



CUDA organizes **threads** in **blocks** which belongs to **grids** that are executed when a **kernel function** is called. Groups of 32 threads with consecutive indices are executed at same time concurrently into **warps** using SM (**Stream Multiprocessors**) where the **CUDA cores** are.



# MeanShift GPU organization

- The dataset is stored using an Array of Structure, an array where each element is a structure of the same type.

$$X = [ (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n) ]$$

- BLOCK\_DIM 32
- Grid dimension: [n / BLOCK\_DIM ].
- A general thread is accessed with this formula:  
$$\text{index} = (\text{BlockDim} * \text{BlockIdx} + \text{tx}) * k$$
- Where k = 2



# Shared Memory

- A further optimization is reached using CUDA Shared Memory
- `__global__ void cuda_MeanShift_2D`
- Shared Memory is smaller than global memory but faster
- Threads which belong to same block can cooperate thanks to shared memory: **TILING PATTERN** can help so that if `BLOCK_DIM = TILE_WIDTH` there will be  $O(n / \text{TILE\_WIDTH})$  memory accesses.
- `__syncthreads()` must be called



# Experiments and Results

- A comparison between sequential and parallel MeanShift is presented
- The experiments have been executed on a machine equipped with:
  - Intel(R) Core i7-8550U CPU @ 1.80GHz 4 Cores, 8 Threads
  - RAM 16 GB DDR4 2400Mhz
  - GPU "GeForce GTX 1050" 4GB with CUDA 10.2
  - (5 SM 640 CUDA cores)





# Dataset generation

- Experiments were conducted using datasets of different dimensions (1000 up to 100k points)
- Points are 2-dimensional (x, y)
- Points have been generated using:
  - ***std::default\_random\_engine*** (which generates pseudo-random numbers)
  - ***std::normal\_distribution*** (which generates random numbers according to normal distribution)

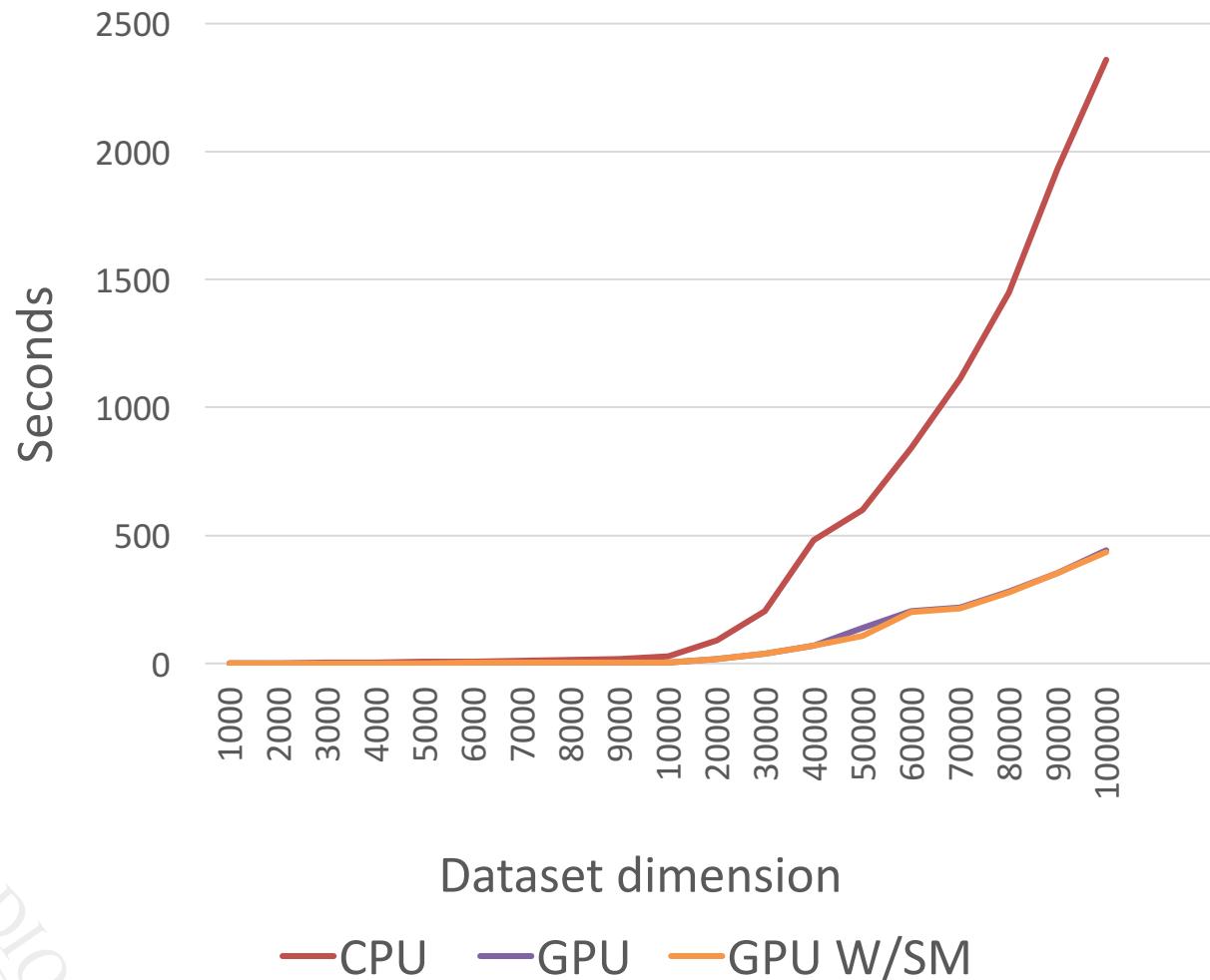




# Parallel MeanShift SpeedUps

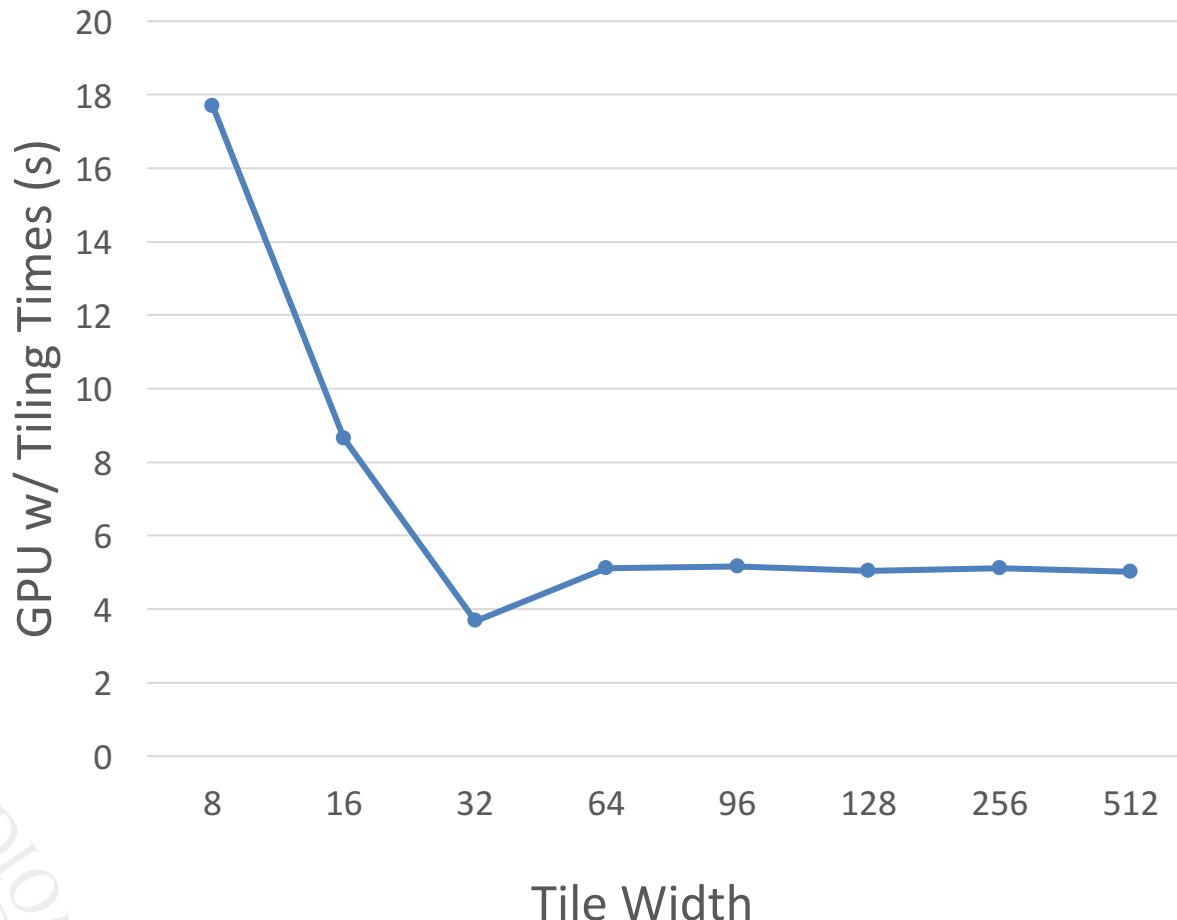
Dataset dimension	CPU Time	GPU Time w/ Tiling	Speedup
1000	0.28784s	0.27812s	1,086x
5000	5.84816s	0.14012s	5,073x
10000	28.4357s	3.67366s	6,507x
20000	91.4664s	16.3537s	5,22x
30000	204.659s	37.7643s	5,199x
40000	480.582s	68.6471s	6,801x
50000	600.642s	108.553s	4,35x
60000	841.745s	199.829s	4,1366x
70000	1113.37s	214.159s	5,1182x
80000	1448.34s	278.723s	5,173x
90000	1932.42s	353.804s	5,450x
100000	2358.56s	434.847s	5,333x

# Performances comparison



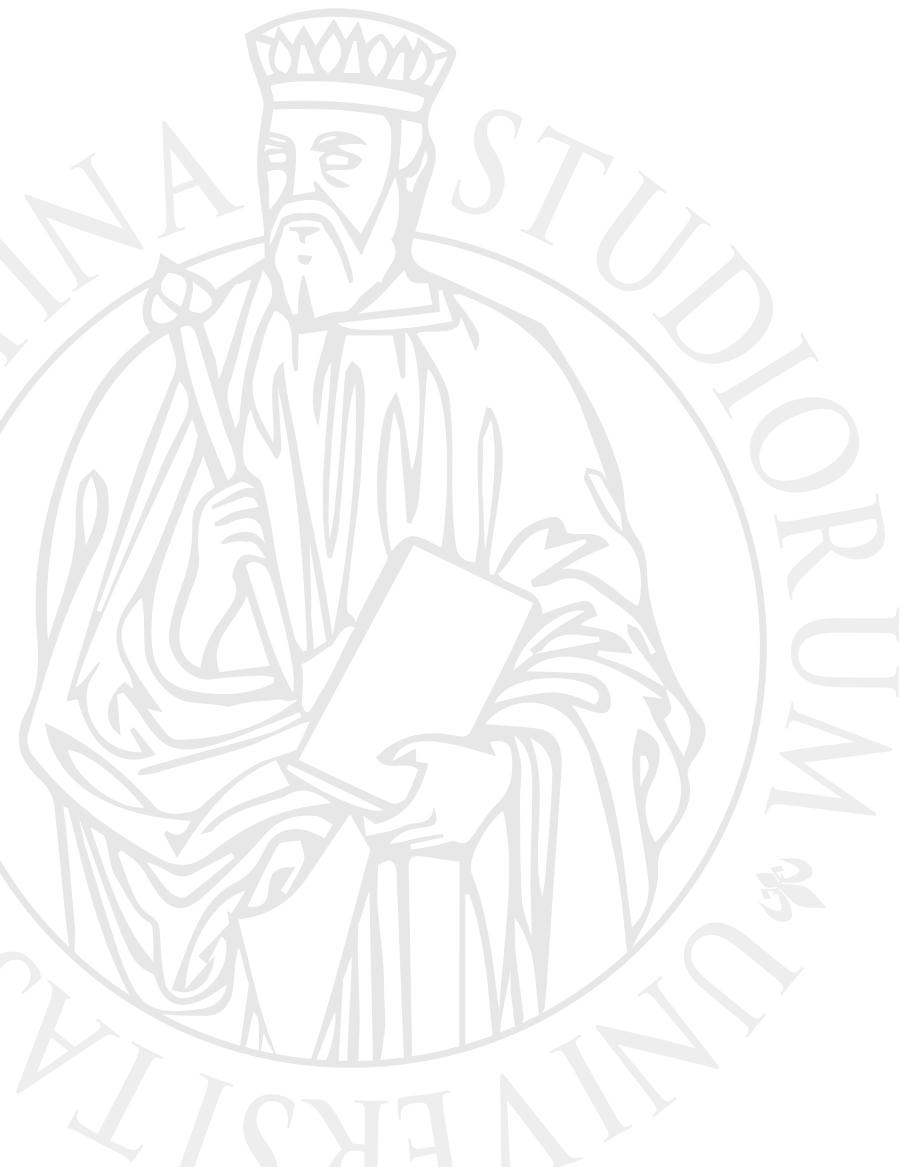
# Tiling with different TILE WIDTH

10.000 points dataset





UNIVERSITÀ  
DEGLI STUDI  
FIRENZE



# K Means Algorithm: comparison between CPU and GPU implementation using OMP and CUDA

Parallel computing Course  
Mid-term project

Lorenzo Pisaneschi

Firenze, 30 gennaio 2020

# The k-Means algorithm

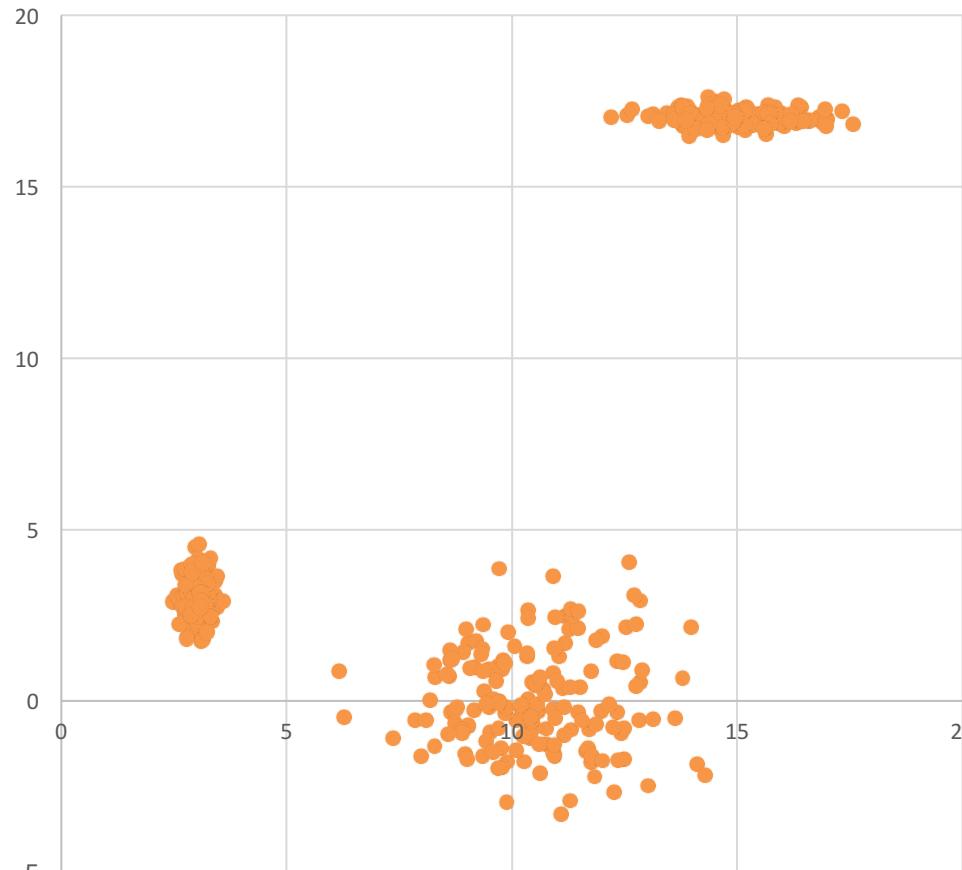
- K-Means is one of the most used *unsupervised* machine learning algorithms in order to classify a set of data
- The goal is to assign each point to one of the  **$k$  cluster**, where  $k$  is an input parameter (clusters number)
- A **cluster** is a group of similar points
- **Centroids** define clusters
- **Similarity** between points is calculated according to a chosen **metric**

# The k-Means: steps

1. Initializing phase: k centroids are chosen according to some initialization criteria
2. At each iteration distances are calculated between each **points** and **centroids** accumulating coordinates and counting the number of points belonging to each cluster
3. **Means** are calculated, averaging the accumulated coordinates values with the corresponding center
4. Then centroids coordinates are updating



# The k-Means: stop criteria



- K-Means stops when max iteration is reached or the procedure converges (there are not further centroids updating)
- **k parameter** is crucial as the initialization phase: different results could be reached



# Sequential k-Means

**k-Means** (*inputPoints*, *clusters*, *maxIterations*)

*it*  $\leftarrow$  0

*isNotConvergence*  $\leftarrow$  false

**while** *t* < *Iterations* **do**

*maxIterations* & *isNotConvergence* **do**

**CalculateAllDistances**(*inputPoints*, *clusters*)

*isNotConvergence* = **FindNewCentroids**(*clusters*)

*it*  $\leftarrow$  *it* + 1



# K-Means – Consideration

$$\operatorname{argmin}_S \sum_{i=1}^k \sum_{x \in S_i} \|x - \mu_i\|^2$$

- Set of d-dimensional observations  $(x_1, x_2, \dots, x_n)$
- Metric: here Euclidean distance
- K-Means aims to **partition** n-observations into  $k << n$  clusters so as to **minimize** the whitin-cluster sum of squares (**WCSS**, i.e. variance)



# K-Means – Analysis

- K-Means is  $O(knd)$
- Where:
  - $k$ : clusters number
  - $n$ : dataset dimension ( $n$  d-dimensional points)
  - $d$ : point dimension
- K-Means algorithm is widely used
- However, finding the optimal solution is NP-hard in general Euclidean-space; local minima solution could produces wrong results



# Parallel K-Means

- K-Means is an expensive algorithm: for each point at each iteration  $k$  distances are calculated
- It's possible to implement k-Means using parallel and GPU programming since distances can be calculated separately like accumulators and counters can be updated one independently by the other
- In the follow OMP and CUDA k-Means implementations are discussed and presented



# OMP

- OMP is an API which supports multi-platform shared memory multiprocessing programming in C++
- With OMP is possible to use multithreading by using some simple **directives** that indicates parallel region and how variables must be shared among threads
- Engineering in a parallel region, a main thread (master) is created; **fork-join model** is followed.
- The parallel execution mode is **SPMD**





# OMP k-Means

- Implicit barrier is provided by main thread (for synchronization)
- Each thread have to know ***dataset dimension*** and ***clusters number***
- Each thread has its own ***clusterindex*** and ***minDistance*** variables





# OMP k-Means implementation

- `#pragma omp parallel`  Parallel region definition
- `#pragma omp parallel default(shared)  
private(minDistance, clusterIndex)  
firstprivate(pointsSize, clustersSize)`  Variables sharing directives
- `#pragma omp for schedule(static)`  Assignment of work loads to threads is controlled
- `#pragma omp critical  
if (distance < minDistance) {  
 minDistance = distance;  
 clusterIndex = j;  
}`  Critical section for minDistance update



# OMP k-Means considerations

- Sequential consistency is respected
- Bernstein's condition is respected
- Operations are correctly executed and data are shared in the right way improving execution performances
- ***Critical section*** is fundamental for the correct distance updated; however, it influences needed execution time
- Threads number can be chosen varying setting OMP\_THREADS\_NUM



# CUDA

- Another k-Means parallel version has been implemented using CUDA
- GPU architectures, as seen for MeanShift, fits perfectly also k-Means algorithm
- **GPGPU** programming using the **SIMT** paradigm could help in boosting application purpose
- In the follow **CUDA occupancy** is analyzed deeper
- Memory management: **thrust library**



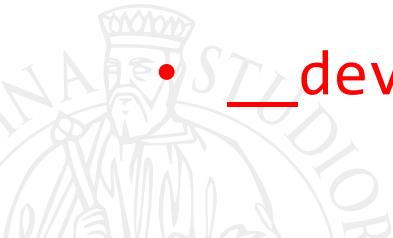
# CUDA k-Means implementation

The following CUDA **kernel**s are provided:

- **`__global__ void cuda_kMeans_CalculateDistances`**
- **`__global__ void cuda_kMeans_updateCentroids`**
- **`__global__ void cuda_kMeans_clearAll`**

The Euclidean Distance is the chosen metric:

- **`__device__ float euclideanDistance`**





# Experiments and Results

- The aim is to compare the k-Means sequential version with the OMP and CUDA ones
- *OMP k-Means*: the goal is to show performance variation varying **OMP\_THREADS\_NUM**
- *CUDA k-Means*: the goal is to show performance variation varying **BLOCK\_DIM**
- The experiments have been executed on a machine equipped with:
  - Intel(R) Core i7-8550U CPU @ 1.80GHz 4 Cores, 8 Threads
  - RAM 16 GB DDR4 2400Mhz
  - GPU "GeForce GTX 1050" 4GB with CUDA 10.2
  - (5 SM 640 CUDA cores)

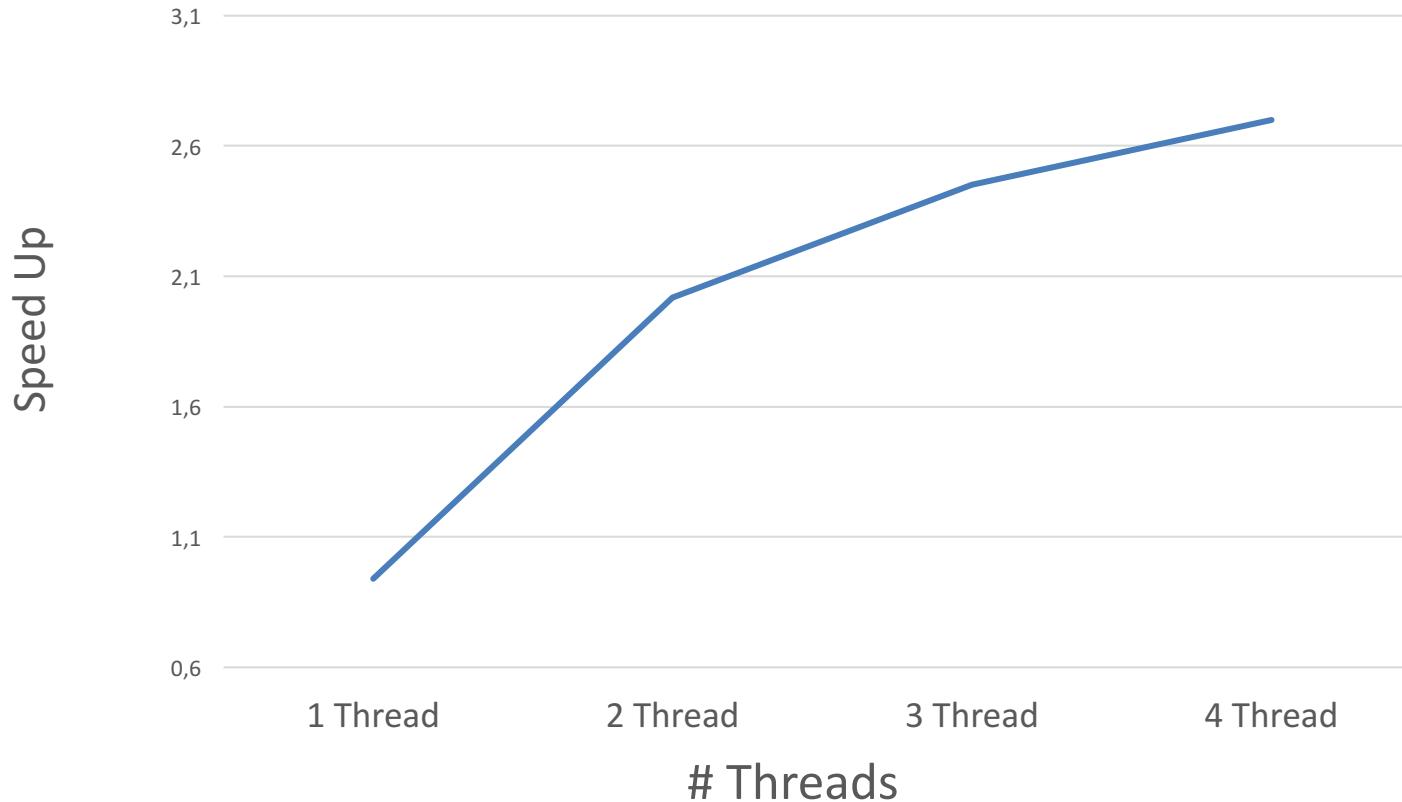


# Parallel OMP k-Means SpeedUps

Dataset size	CPU Time	OMP Time	Speedup
1000	0,0087537s	0,0309955s	-
10000	0,0714324s	0,0618166s	1,898x
100000	0,7779713s	0,2858521s	1,155x
500000	3,5961935s	1,3595931s	2,645x
1MLN	6,7640652s	2,1600539s	3,131x
10MLN	13,387533s	3,5475999s	3,773x

# #Threads variation

Dataset size: 100K

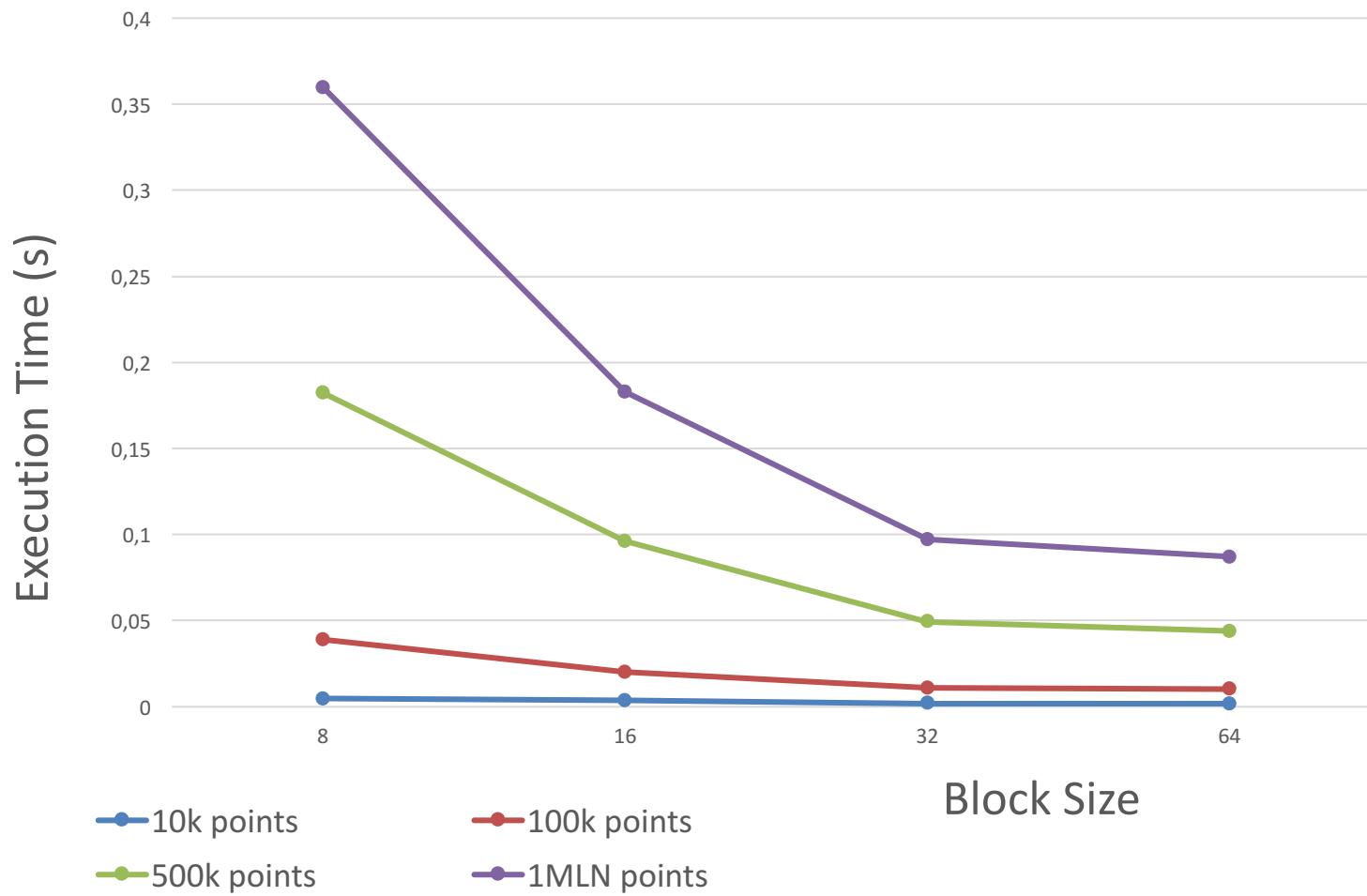




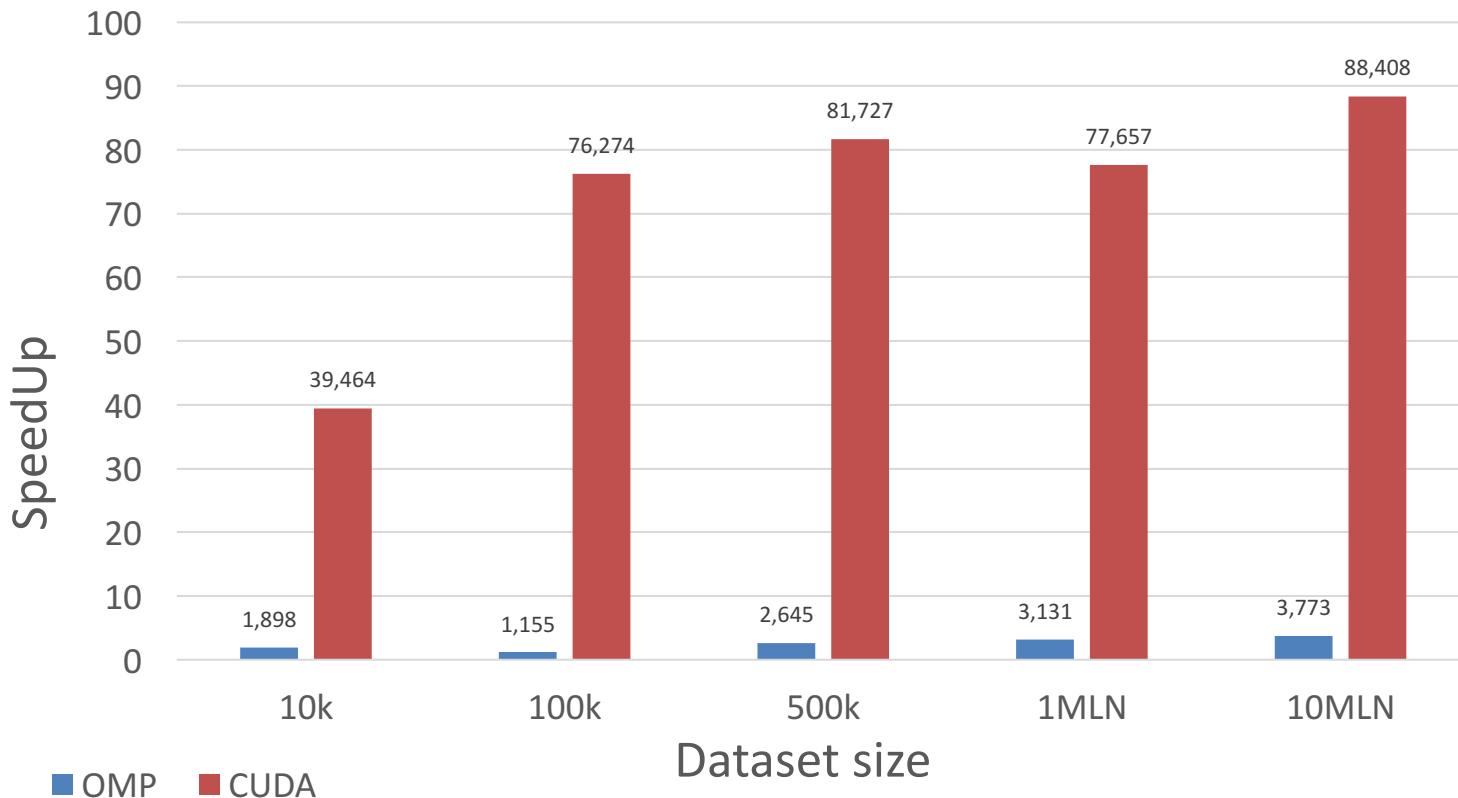
# Parallel CUDA k-Means SpeedUps

Dataset size	CPU Time	CUDA Time	Speedup
1000	0,0087537s	0,0013126s	66,721x
10000	0,0714324s	0,0018166s	39,464
100000	0,7779713s	0,0102352s	76,274x
500000	3,5961935s	0,0440193s	81,727x
1MLN	6,7640652s	0,0871926s	77,657x
10MLN	13,387533s	0,1686117s	81,408x

# Block size variation



# OMP-CUDA k-Means performances comparison





# Considerations and conclusion – [1]

- Clustering is a widely used technique in various fields
- Medical research, custom service, e-commerce; clustering algorithm can be used also in image processing tasks like image segmentation
- In general, clustering algorithm can retrieve information from dataset





# Considerations and conclusion – [2]

- K-Means and Mean Shift algorithms exploit the same task
- However, they differ for input parameters, computational costs, implementation strategies
- K-Means is faster than MeanShift, while Meanshift is more adactable in real world applications
- An important suggestion could be the follow: combine MeanShift and k-Means clustering in order to obtain a powerful clustering algorithm



**THANKS FOR YOUR ATTENTION!**



# Resources

Code is available on Github

**Sequential and OMP version (k-Means):**

[https://github.com/pisalore/kMeans\\_OMP](https://github.com/pisalore/kMeans_OMP)

**Parallel version with CUDA (k-Means):**

[https://github.com/pisalore/kMeans\\_CUDA](https://github.com/pisalore/kMeans_CUDA)

**Sequential and OMP version (MeanShift):**

[https://github.com/pisalore/MeanShift\\_sequentialCPP](https://github.com/pisalore/MeanShift_sequentialCPP)

**Parallel version with CUDA (MeanShift):**

[https://github.com/pisalore/MeanShift\\_CUDA](https://github.com/pisalore/MeanShift_CUDA)



# 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. Authors. The frobnicatable foo filter, 2006. ECCV06 submission ID 324. Supplied as additional material eccv06.pdf.
2. 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.
3. NVIDIA. Cuda documentation. <https://docs.nvidia.com/cuda/>, 2019. [Online; accessed 16-12-2019].
4. D. B. Kirk and W-M. W. Hwu, Morgan Kaufman. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.
5. Lloyd, Stuart P. "Least squares quantization in PCM." *Information Theory, IEEE Transactions on* 28.2 (1982): 129-137.
6. MacQueen, James. "Some methods for classification and analysis of multivariate observations." *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 14.* 1967.
7. Wikipedia k-means clustering. [https://en.wikipedia.org/wiki/K-means\\_clustering](https://en.wikipedia.org/wiki/K-means_clustering). [Online; accessed 6-01-p2020].
8. Thomas Dauber and Gudula Rünger, Springer. "Parallel Programming for Multicore and Cluster Systems", - Chapt. 6
9. NVIDIA. Cuda documentation. <https://docs.nvidia.com/cuda/>, 2019. [Online; accessed 16-012019].
10. D. B. Kirk and W-M. W. Hwu, Morgan Kaufman. "Programming Massively Parallel Processors: A Hands-on Approach"- 2nd edition - Chapt. 1-2-5
11. D.T. Marr, F. Binus, D.L. Hill, G. Hinton, D.A. Konfaty, J.A. Miller, and M. Upton. Hyperthreading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1): 4–15, 2002.