PC-2019/20 Course Project: Mean Shift Algorithm, a GPU implementation with CUDA

Lorenzo Pisaneschi
E-mail address
lorenzo.pisaneschi1@stud.unifi.it

Abstract

Mean Shift is a non-parametric clustering algorithm which does not require prior knowledge of the number of clusters. However, it has a very large computational cost and often other clustering techniques are preferred, like k-means algorithm. Despite this, Mean Shift is embarrassingly parallel itself, so it can be easily implemented exploiting GPU architectures and computing power to improve results obtained with CPU implementations.
In this paper, after a briefly introduction to the algorithm, the sequential and the parallel Mean Shift implementations are presented and analyzed: the main goal of these studies is to show and evaluate the performance improvements of parallel code using GPU over the sequential one CPU based.

1. Introduction: the Mean Shift Algorithm

The Mean Shift algorithm[1] is a clustering technique which does not require knowledge about number of clusters to find; it is based on Kernel Density Estimation. KDE is a method to estimate the underlying distribution (the probability density function) for a set of data.
Given a set on *n points*, at each step a kernel function is applied to each point of the set; this application generates a probability surface; at points level, it determines a *shift* to each point toward the direction of the nearest peak of the KDE surface following the gradient direction. The procedure ends when all point reaches the maxima of the underlying distribution estimated by the chosen kernel; the centroids will be the *peaks* and clusters the *regions* around them.

1.1. The kernel function

Core of the Mean Shift procedure is obviously the kernel function [2]. The most popular kernel function (also used in the experiments presented in this paper) is the **Gaussian Kernel**:

$$k(x) = e^{\frac{-x}{2\sigma^2}} \qquad (1)$$

Where σ is the standard deviation: in the follow we will name σ as bandwidth. The bandwidth controls the smoothing of kernel function: it is the only parameter necessary for Mean Shift computation. The more this value is small, the more clusters will be found; the more bandwidth is big, the less clusters will be found. So, the bandwidth setting in kernel functions is extremely important.
At each iteration, new positions are calculated for each point in according to kernel function $K(x - x_i)$, a Gaussian kernel in our case; this function determines the weight of nearby points for re-estimation of the weighted mean of the density determined by K, $m(x)$:

$$m(x) = \frac{\sum_{x_i \epsilon N(x)} K(x - x_i)x_i}{\sum_{x_i \epsilon N(x)} K(x - x_i)} \qquad (2)$$

Where $N(x)$ is the neighboorod of x, a set of point for which $K(x_i) \neq 0$.
The difference $m(x) - x$ is called *mean shift.*
The algorithm sets $x \leftarrow m(x)$ until $m(x)$ converges.
An array contains the points labelled with their centroids coordinates is returned.

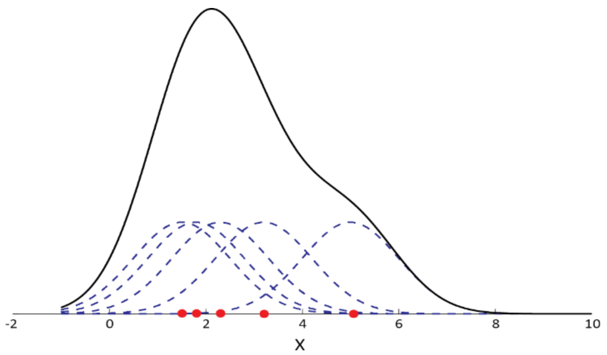

Figure 1 An example of Gaussian Kernel function and the shift computation

1.2. Pseudocode

```
Algorithm 1: Mean shift
procedure    Mean Shift (X, n, bandwidth, Iterations)
    it ← 0
    $Y_0$ ← X
    while it < Iterations do
        $Y_{t+1}$ = Shift($Y_t$, X, bandwidth)
        Swap($Y_t$, $Y_{t+1}$)
        it ← it + 1
    return $Y_{t+1}$
```

```
Algorithm 2: Shift
procedure    Shift ($Y_t$, n, bandwith)
    for all y in $Y_t$ do
        $m_i$ = Compute Shift($y_i$, X, bandwidth)
        $y_i$ = $x_i$+$m_i$
    return $Y_{t+1}$
```

```
Algorithm 3: Compute Shift
procedure    Compute Shift($y$, X, bandwidth)
    weights ← 0
    $m_i$← 0
    for all $x_j, j = 1,.., |X|$ do
        $w_j \leftarrow K(\frac{||y_i - x_j||}{bandwidth})$
        $m_i \leftarrow m_i + x_i w_j$
        weights ← weights + $w_j$
    return $m_i$ / weights - $y_i$
```

1.3. Algorithm Analysis

Algorithm 1 is the *Mean Shift* itself, which uses the *Shift* and the *Compute Shift* procedures to shift each points with respect to the kernel function, a Gaussian one in this case. The algorithm returns an *Y* array which contains all the points labeled with the belonging centroid coordinates (x, y). It's clear that the points are assigned to a cluster with respect to a weighted average between y_j and each x_i calculated exactly in the *Shift procedure* using the *Compute Shift* one, where the kernel is effectively used: nearest points will have an higher weight, so they will converge to their natural local maximum. It's demonstrated that Mean Shift algorithm is computationally an expensive algorithm: $O(Tn^2)$ where T is the number of iterations computed, and n is the dataset X dimension, i.e. the points number.
The Mean Shift algorithm has many applications: it can be used for visual tracking, image processing like image segmentation or smoothing and many real time tasks.
Mean shift is an easy algorithm, which requires as input only the kernel bandwidth; however, it could be very slow on CPUs. In the follow, a GPU Mean Shift implementation using CUDA is presented to avoid this procedure slowness.

2 Implementation

As mentioned above, the Mean Shift algorithm is slow on CPUs by is nature complexity; on the other hand, it is embarrassingly parallel: each point could be processed independently by the kernel function. Made this premise, we can consider an only-read array *X*, the input, and the *Y* which will be returner at the end of the computation: we can assign a point to one thread that will read it from the data structure X without any interference with the others; then, after ***Algorithm 3*** *Compute Shift* is done, each thread will write in Y theirs results, will synchronize with the others and will be ready for the next Mean Shift iteration.

2.1 Parallel Mean Shift: GPU programming with CUDA

```
Algorithm 4: Parallel Mean Shift
procedure    Parallel Mean Shift (X, n, bandwidth,
             Iterations)
    it ← 0
    $Y_0$ ← X
    while t < Iterations do
        $Y_{t+1}$ = ParallelShift($Y_t$, X, bandwidth)
        SynchronizeThreads()
        Swap($Y_t$, $Y_{t+1}$)
        it ← it + 1
    return $Y_{t+1}$
```

In this section a GPU implementation using **CUDA** [3] (Compute Unified Device Architecture) will be discussed with the purpose of sequential and parallel mean shift implementations performances comparison. Then, a further optimization over the CUDA code will be analysed.
Generally, there are two main differences between CPU and GPU hardware which demonstrate that using GPU is better over CPU for Mean Shift implementation: the number of cores available and the relative context switching. A GPU has many more cores than a CPU and an hardware scheduler: therefore as mentioned above, a GPU hardware fits perfectly the Mean Shift algorithm implementation architecture because each point shifting through each main procedure iteration can be assigned to one thread. Consequently, in the GPU Mean Shift parallel version there will be many less memory contexts switching compared to the sequential version of the algorithm: in fact, there will be less memory readings and less memory latency. A GPU is projected just for boost computing and have a large bandwidth in memory access. At this point, is possible to enter in the CUDA scenario: the CUDA **SIMT** paradigm (Single Instruction, Multiple Threads) can be adopted to obtain speedup thanks to GPU hardware programming. The SIMT paradigm differs from the SIMD (Single Instruction Multiple Data) one in the

fact that multiple threads can execute independently and asynchronously.
After this brief introduction, it's clear that threads have to be organized in the best way to improve performance with parallelism. Given an *X* dataset with *n points*, each thread will be assigned to one point. In CUDA, the SIMT paradigm is reached up executing groups of 32 at the same time threads called ***warps;*** when a warp ends, the next one start. Warps belong to ***blocks*** which make up ***grids***. Since this threads organization, it's a best practice to set the block dimension a multiple of 32 (which is a "cuda magic number" for warp size). In the presented implementation, `BLOCK_DIM 32` has been found to be a good choice. Number of blocks, the ***grid dimension***, is [n / `BLOCK_DIM`]. Since the Mean Shift here presented refers to 2D points, the dataset is stored using an *Array of Structure*, an array where each element is a structure of the same type.

$$X = [\,(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\,] \qquad (3)$$

A general point can be accessed according to this formula:

$$index = (BlockDim * BlockIdx + tx) * k \qquad (4)$$

Where k = 2 is the number of point coordinates (x, y), *BlockDim* is exactly the `BLOCK_DIM` explicated before, *BlockIdx* is the block id inside the grid and *tx* is the thread identifier. This formula depends linearly by *tx* for memory access optimization: same block threads will access near global memory locations *coalescing* all these accesses into a consolidated access to consecutive DRAM locations (***bursts***) loading data in GPU registers/ cache reducing I/Os operations and memory latency. This RAM access pattern is called ***coalescing access***.
Another aspect to be analysed is ***divergence***. All threads have to execute same instruction at same time, but if a conditional statement is present, same block threads in a warp could have a different execution flow. To fix this problem, sub warps including threads which have to follow different flows are created, negatively influencing performance, since some threads will be unavoidably idle.

2.2 Shared Memory

There is a trade-off between *global memory* and *shared memory* in CUDA [4]: global memory is large but slow, while shared memory is small but fast. Then a further optimization could be made correctly programming the shared memory and organizing data. The idea is to decrease the number of readings from global memory ($O(n)$, one access for each point) so that threads belongs the same blocks cooperate reading from their shared memory. The pattern which use shared memory is called ***tiling.*** In tiling pattern, there is first a *loading phase*, where data form a *tile* of global memory are loaded into block shared memory between interested threads; then there is a *synchronization phase*, which ensure that all points has been correctly loaded and threads are ready to start their computation; finally, after each thread has ended its calculation phase, then the algorithm pass to the next tile. Let `BLOCK_DIM = TILE_WIDTH` the number of points being stored in the block shared memory: the global memory accesses with tiling pattern will be theoretically $O(\frac{n}{\text{TILE\_WIDTH}})$. Obviously we have to deal with physical shared memory narrowness.

Algorithm 5*: Parallel Mean Shift with Tiling*

```
procedure Shared Memory Mean Shift (X, n, bandwith, Iterations)
    i = (BlockDim * BlockIdx + tx) * k
    $m_i$, w ← 0
    $Y_i \leftarrow Y_t[i]$
    while currentTile < allTilesNumber do
        SMIndex ← Index(currentTile, tx)
        SharedMem[tx] ← X[SMIndex]
        SynchronizeThreads()
        mi,w ← SMShift(SharedMem $y_i$, $m_i$,w)
        SynchronizeThreads()
        currentTile ← currentTile + 1
    return $Y_{t+1}$
```

3 Experiments and Results

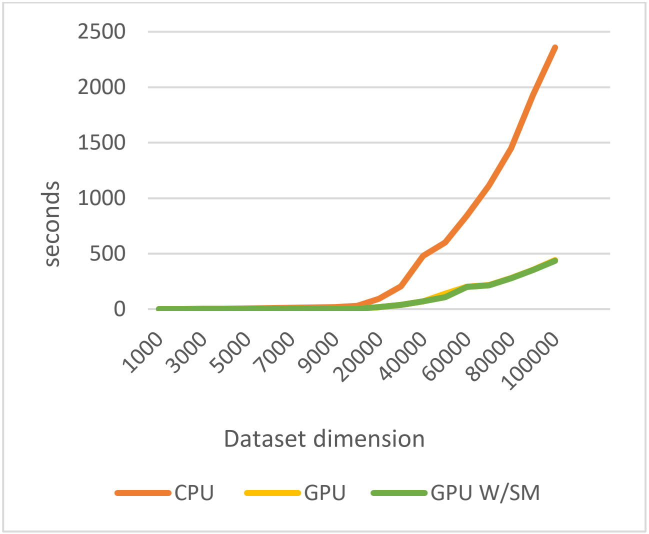

Chart 1 CPU and GPU times varying dataset dimension

In this section the experiments over different Mean Shift algorithm implementations are presented. The aim is to compare different temporal performances obtained with the sequential and the parallel versions of the algorithm in order to measure the SpeedUp obtained using GPU instead of CPU. Then, the parallel version with Shared Memory is

compared with the without one.
Mean Shift with all its versions has been executed with different dataset of different sizes; the averaged time obtained from 5 different algorithm execution has been chosen as reference time for each algorithm execution for each dataset. As block dimension 32 is the value chosen, like the tile width parameter.
All the experiments have been conducted 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

The SpeedUp is the ratio:

$$Speedup = \frac{t_s}{t_p} \qquad (5)$$

where t_s and t_p are sequential time and parallel time respectively.

Table 1 Times measured for Mean Shift Algorithm in its sequential and w/ SM parallel version

Dataset dimension	CPU Time	GPU Time w/ Tiling	Speedup
1000	0.28784s	0.27812s	**1,086x**
2000	1.01536s	0.47945s	**1,898x**
3000	2.16842s	0.66521s	**2,499x**
4000	3.89613s	0.90817s	**4,238x**
5000	5.84816s	0.14012s	**5,073x**
6000	8.30332s	2.53403s	**3,224x**
7000	11.3616s	3.00423s	**3,725x**
8000	14.7856s	3.43435s	**4,225x**
9000	18.6382s	3.61504s	**4,769x**
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**

How can we observe from the *Table 1* results, speedups is almost constant varying the dataset dimension, except for the first datasets presented (1000 to 3000 points). This can be explained by the fact that parallelism isn't a good choice with little dimension datasets: a CPU could performs better than a GPU with a little dimensions points dataset for the Mean Shift.

Table 2 Times measured for Mean Shift Algorithm in its parallel version w/o Tiling and w/ Tiling parallel version

Dataset dimension	GPU w/o Tiling Time	GPU Time w/ Tiling
1000	0.26421s	0.27812s
2000	0.53534s	0.47945s
3000	0.86756s	0.66521s
4000	0.91959s	0.90817s
5000	1.15268s	0.14012s
6000	2.57123s	2.53403s
7000	3.05346s	3.00423s
8000	3.58076s	3.43435s
9000	3.93837s	3.61504s
10000	4.37535s	3.67366s
20000	17.5043s	16.3537s
30000	39.3619s	37.7643s
40000	70.6665s	68.6471s
50000	138.031s	108.553s
60000	203.483s	199.829s
70000	217.531s	214.159s
80000	279.998s	278.723s
90000	354.513s	353.804s
100000	442.235s	434.847s

In *Table 2*, it's possible to consider a further performance improvement thanks to the Shared memory paradigm. The `TILE_WIDTH = 32` is the setting which ensures the fastest Mean Shift execution for the experiments here presented.

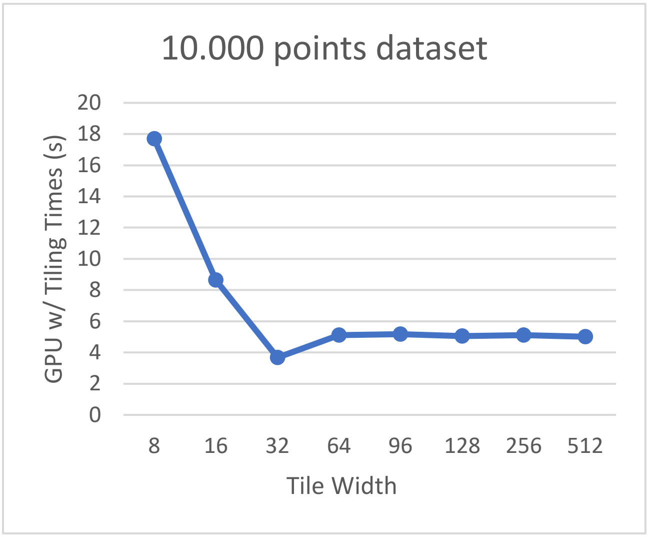

Chart 2 GPU times varying tile width

Chart 2 shows GPU times varying *Tile Width* dimension in CUDA Mean Shift algorithm implemented using Shared Memory. Performances depends on *Tile Width* dimension and dataset dimension given a *Tile Width*. A small tile width dimension could undersize active wraps, while an high tile width dimension value could limits the overall shared memory available. Obviously, hardware specifications are crucial.

4 Conclusion

Mean Shift algorithm [1] could be a good choice for clustering problems where we do not want to decide how many clusters we want to find; however, it's a computational expensive algorithm due to his subroutines over each point of a given dataset and the KDE calculation. However, it has been demonstrated that it's possible to boost this algorithm using GPU programming with CUDA thanks to the fact that each point can be independently processed; furthermore, it has been shown that it's possible to obtain a further optimization reducing memory access thanks to *Tiling Pattern* which exploits GPU Shared Memory, a small but faster memory than the global one.

Resources

Code is available on Github:

- Sequential and OMP version: https://github.com/pisalore/MeanShift_sequentialCPP
- Parallel version with 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.