Mean shift clustering: OpenMP and CUDA implementations

Fernando Chirici
E-mail address
fernando.chirici@stud.unifi.it

Abstract

In this paper will be presented and analyzed the Mean shift clustering algorithm. Specifically, following a small introduction about it's characteristics there will be reported an OpenMP implementation, naturally based on C++, and a CUDA one.
For each of those will be shown a set of experiments built in order to find the ideal configuration parameters to use these clustering tools on a growing set of input data and understand limitation ad capacity of the two parallel versions of Mean shift.

Future Distribution Permission

1. Introduction to Mean shift

Mean shift is an unsupervised learning algorithm that is mostly used for **clustering**. It is widely used in real-world data analysis (e.g., image segmentation) because it's non-parametric and doesn't require any predefined shape of the clusters in the feature space. In fact more generally Mean shift is a model for detecting modes and their regions of influence in the feature space based on the Kernel Density Estimation, a technique to estimate the underlying density function of a distribution of probability over a certain dataset. Unlike many other clustering algorithm, like K-means, this algorithm also does not require to specify the number of cluster to be output, instead the value will depend on the distribution.
The algorithm is structured so that at each step a kernel function is applied to each point in order to shift it in the direction of the local maxima of the kernel function itself. This procedure repeats until every point haven't reached such maxima finally forming a cluster. The kernel function used in this implementation, as it usually happens, is a Gaussian kernel:

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

where d is the distance between the center point to any other point and the standard deviation σ is defined as bandwidth so the parameter to adjust how fast the weight decreases with the increasing of the distance. The algorithm is not highly scalable, as it requires multiple nearest neighbor searches during its execution. It will be showed that it's complexity of

$$O(n^2) \tag{2}$$

will highly impact on the sequential execution and that the parameter configuration is critical.

The experiments on the Mean shift clustering algorithm have been done working in a 3-dimensional space. This choice has been take in order to have fewer trivial parameters to confront between the OpenMP and the CUDA implementations.

2. C++ and OpenMP implementation

The OpenMP version of the Mean shift clustering algorithm has been naturally developed in C++ using, as already said, the Guassian kernel as kernel function. The Algorithm 1 shows how the implementation has been structured. Given the input set P of 3-dimensional points, the function iterates until maximum iterations isn't reached. Inside the outer loop, the OpenMP directive ***pragma*** is called in order to define parallel region in which work is done by threads in parallel. In this case, the directory is statically bind to the enclosing parallel region. The static binding has been preferred to the dynamic one because of the structure and memory access of the parallelized loop pragma works on. In fact, static schedule means that iterations blocks are mapped statically to the execution threads in a round-robin way. The static flag guarantees that if multiple loops characterized by the same amount of iterations are running using the same number of threads using static scheduling, then each thread will receive exactly the same iteration ranges in every in every parallel region. This is particularly important in NUMA[1] systems because from the first memory access at the first loop, that memory will then reside in the NUMA node where the executing thread is. From the second loop on, the same thread is be able to access the same memory location much faster since it will reside on the NUMA node. Another reason to choose the static scheduling instead of the dynamic one is that the workload on the parallel section in evenly divided by the deterministic evolution of the algorithm.

Algorithm 1: OpenMP Mean shift implementation

```
Data: P
ShiftedP_set = P;
while is not reached maximum iterations do
    #pragma parallel for schedule(static);
    for each point p in ShiftedPset do
        p = shift_single_point(p, P);
    end
end
End;
```

In the Algorithm 2 is reported the implementation of the shifting point function. In order to return the shifted value (new value) of a point p of the feature space, the function needs the current value of the point p, the input set P and the bandwidth, a parameter that controls the smoothing of

[1]Non-uniform memory access

Algorithm 2: shift_single_point implementation

```
Data: P, p, BW
shiftedP, weight = 0;
for each point x in P do
    d = Squared distance(p, x);
    w = GaussKernel(d,BW) ;
    shiftedP = shiftedP + w * x ;
    weight = weight + w;
end
return shiftedP/weight;
```

the resulting probability density function (so increasing the bandwidth would generate fewer but bigger clusters). Inside the function, for every point of the featured space, the euclidean distance from the current point p is calculated in order to evaluate it in the kernel function. The updated point is returned weighed according to its new weight.

2.0.1 OpenMP experiments

The experiments were run on a Macbook Pro 2017 with a 3,1 GHz Intel Core i7 quad-core hyperthreading to 8 logical cores. In order to achieve interesting results in acceptable times, every test runs the Mean Shift clustering algorithm on different 3-dimensional input sets of sizes [1000; 10000, 35000; 50000] at first sequentially, removing the pragma directive, then using 2, 4, 6 and 8 threads and moving the bandwidth in order to find the best performance.

Table 1: Speedup bandwidth 2

Mode	Time	Speedup	Points
Sequential	7.312	-	1000
2 cores	3.81295	1.918	1000
4 cores	2.27233	3.218	1000
6 cores	2.02205	3.616	1000
8 cores	**1.83168**	**3.992**	**1000**
Sequential	854.922	-	10000
2 cores	402.092	2.126	10000
4 cores	170.191	5.0233	10000
6 cores	119.799	7.136	10000
8 cores	**115.395**	**7.409**	**10000**
Sequential	7829.59	-	350000
2 cores	3276.77	2.389	350000
4 cores	1895.40	4.131	350000
6 cores	1559.57	5.020	350000
8 cores	**1321.15**	**5.926**	**350000**
Sequential	15221.1	-	500000
2 cores	6004.2	2.535	500000
4 cores	3399.5	4.477	500000
6 cores	2786.98	5.461	500000
8 cores	**2466.52**	**6.171**	**500000**

The first experiments shows how the algorithm works with a bandwidth equal to 2. As visible from Table 1 above and Figure 1, the speedup defined by parallelizing on an increasing number of threads is definitely visible. The 8 core version in fact is always the best one, outperform the 2 core one of 1.98s and the sequential one of 5.478s working on the 1000 point set, and keeps the same trend for the bigger sets. In Figure 2 and 3 is visible that the same trend also apply to the experiments using bigger bandwidth, respectively equal to 6 and 14. It's interesting to find the best performance achievable by Mean shift on those input sets. After what's already been showed, it's reasonable to look for the best configuration in 8 core mode of execution.

Table 2: Best bandwidth for input sets

Bandwidth	Time	Points
2	1.83168	1000
6	1.1326	1000
14	**1.05149**	**1000**
2	115.395	10000
6	**110.901**	**10000**
14	115.46	10000
2	**1321.15**	**35000**
6	1343.03	35000
14	1402.42	35000
2	**2766.52**	**50000**
6	3197.641	50000
14	3221.142	50000

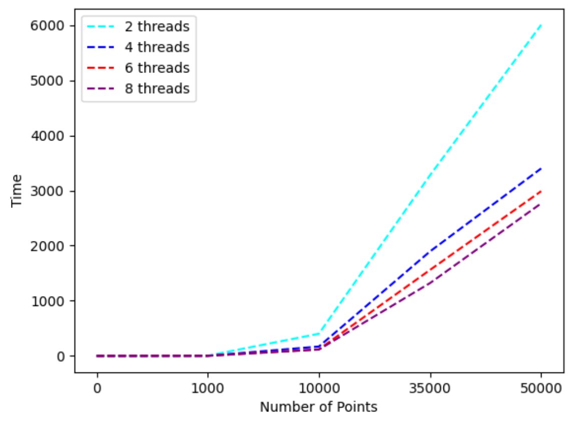

Figure 1. OpenMP Mean shift speedup - BDW=2

It's also interest to notice from Table 2 that somewhat counterintuitively the best performance in small dataset is achieved by big bandwidth (14) where in big one is achieved by small bandwidth (2). the reason is probably attributable to the mere starting position of the points.

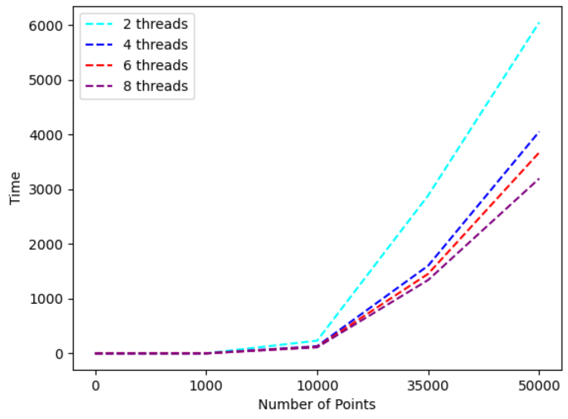

Figure 2. OpenMP Mean shift speedup - BDW=6

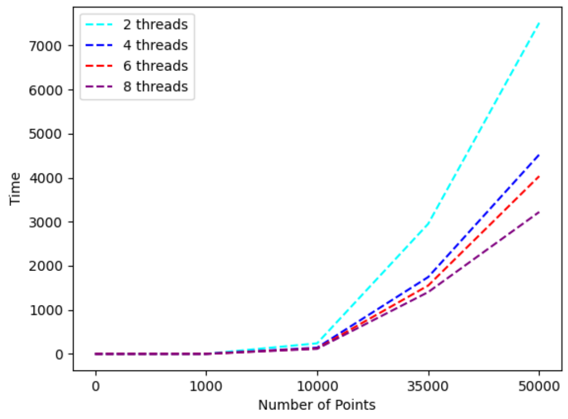

Figure 3. OpenMP Mean shift speedup - BDW=14

2.1. CUDA implementation

The CUDA implementation of Mean shift is naturally similar to the OpenMP one but built in order to exploit as much as possible from the GPU. For the working environment has been chosen *Google Colab* that, in free mode, allow you to run on an NVIDIA Tesla K80 GPU and 12GB of ram.

A CUDA implementation of Mean shift allow to take advantage of a great number of GPU cores. Following what's already been said for the OpenMP implementation, in this case is also possible to assign individual shifting to different cores. The implementation is build in order to take advantage from the fact that modern DRAM systems are designed to always be accessed in burst mode. In fact the points are stored in memory as a *structure of arrays* in order to work with coalesced memory and to be more suitable to work on L2 cache memory.
In order to only need one coordinate to define the grid and the blocks, the array of points is shape as a 1-dimension array arranged by concatenating the sets x y and z coordinates in fixed blocks of BLOCK_DIM size. This means that the access to the array takes the form:

$$blockDim.x * blockIdx.x + threadIdx.x \quad (3)$$

where $blockDim.x = BLOCK\_DIM$ This shape allows threads to work through consecutive global memory locations and enable the hardware to merge and map every memory access into a consolidated access to consecutive DRAM locations, drastically increasing performances.

In the CUDA implementation, the Mean shift algorithm (Algorithm 3) is implemented as a CUDA kernel that is invoked from the main function (Algorithm 4) in order to accomplish the maximum amount of iterations set during the configuration.
The GRID is defined as a function of the block size as follow:

$$GridDim = [dim3](nPoints/BLOCK\_DIM); \quad (4)$$

Algorithm 3: CUDA Mean shift core implementation

```
Data: P
ShiftedP_set = P;
while is not reached maximum iterations do
    CUDA_mean_shift(ShiftedP_set,P);
end
End;
```

The CUDA mean_shift kernel take in input the set of shifted points, the input set and the bandwidth. At first *idx* is accessed following the instruction reported in (3). Then, before computing the shift of its corresponding point each thread has

Algorithm 4: CUDA_mean_shift implementation

```
Data: ShiftedP_set, P, BDW
idx = blockIdx.x $*$ blockDim.x $+$ threadIdx.x;
if $idx < |P|$ then
    shiftedP = 0;
    weight = 0;
    p = shiftedP_set[idx];
    for each point x in originalP_set do
        d = Squared dist(p, x);
        w = GaussKernel(dist,BW);
        shiftedP = shiftedP $+$ w $*$ x weight = weight $+$ w
    end
    shiftedP_set[idx] = shiftedP/weight
end
```

to check if its index is not greater that the number of points. In this case the thread simply does nothing. This may happen when the number of points is not multiple of BLOCK_DIM, therefore more threads have been instantiated than necessary. Instead, if the condition is match, the algorithm proceed to calculate the shifted point just like showed in the OpenMP implementation.

2.1.1 CUDA experiments

The CUDA implementation of Mean shift algorithm, as expected, gives much better results than the C++ and OpenMP ones therefore in order to test the performances of such algorithm the tests run over increasing input sets of sizes [100000, 250000, 525000] points, quite bigger then those used to test OpenMP. Apart from the CUDA implementation itself, another contributing factor is that Google Colab offer an ideal development environment for such applications.

In Figure 4 are visible the results of the test executed in order to find the best block size, hence the ideal grid, to run Mean shift clustering on the 100k input set. The ideal block size, marked in red, is 64 allowing the algorithm to run in an average time of 0.465359. Following (4), this block size corresponds to a grid of size 1562.

The same test has been run for the other input sets and the results are visible in Table 3.

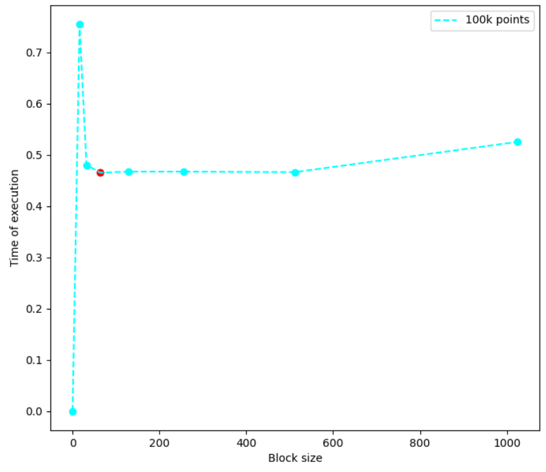

Figure 4. Best block size for 100k input set.

Table 3: Best block size for input sets

Points	Time	Block size	Grid size
100000	0.465359	64	1562
250000	3.064378	128	1953
525000	12.991542	256	2050

Is visible the proportional trend that the dimensions of block and grid keeps following the size of the input set.

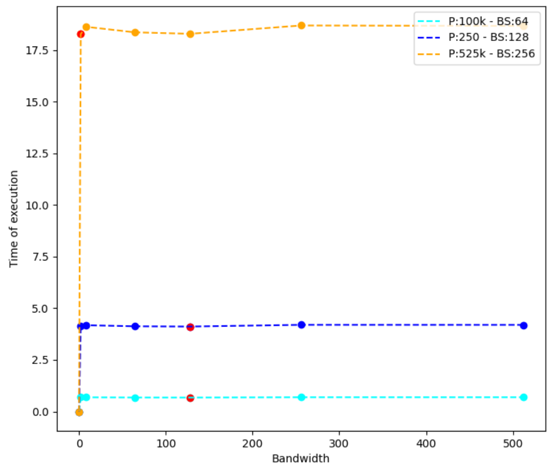

Figure 5. Best bandwidth for every optimal grid.

Established the best setup of the grid to run the algorithm on the chosen input sets, it's interesting to find out how the bandwidth influence the performances.

In Figure 5 is reported a test that shows for every input sets different executions set up with the best block size found as the bandwidth variates from 2 to 1024. As visible the changes are not quite relevant and always variate on a range of maximum 1%.

2.2. OpenMP and CUDA comparison

It can be seen in Figure 6 the impressive difference between running the algorithm in OpenMP and CUDA set on their best configuration to run over the 100k input set. Specifically the first one, as discovered in section 2.1.1, is set to run on 8 cores and a bandwidth at 2 while the second one, following the results of the experiments, is set to run with a block size equal to 32 and bandwidth 2. The speedup is obvious and reported in the summary on Table 4.

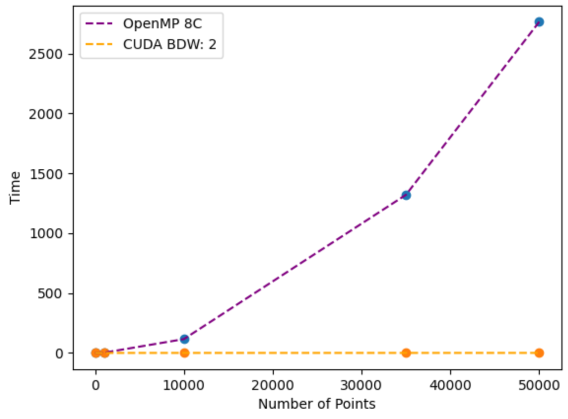

Figure 6. Speedup CUDA over OpenMP*

Table 4: Speedup CUDA over OpenMP*

Points	OpenMP	CUDA	Speedup
1000	1.83168	0.002386	767.68
10000	115.395	0.015821	7293.78
35000	1321.15	0.103233	12797.75
50000	2766.52	0.200798	13777.63

*: Results obtain on different hardware.

3. Conclusions

Through this experiments has been possible to observe the incredibly positive effects of parallelization on both the OpenMP and the CUDA implementation of Mean shift, a non parametric clustering algorithm. The OpenMP implementation is not only very easy to obtain from a C++ program, just by adding a pragma directive, but also show speedup on the execution up to a factor 7.409. The CUDA implementation, unfortunately forced to be tested on a different environment, show an incredible speedup over the OpenMP version up to a factor 13777.63. This last implementation in fact, running on the Google Colab GPU doesn't start to run at lower rates until reaching input sets over 500k 3-dimensional points.

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. IEEE, 1999.

[2] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 10.2.89, 2020.

[3] G. n. Tanner. Mean shift. machine learning explained., 2022.

[4] wikipedia$_2$022.$Meanshift, Dec$2022.