Mean Shift Clustering Algorithm

Lorenzo Agnolucci
E-mail address
lorenzo.agnolucci@stud.unifi.it

Abstract

Mean Shift is a non-parametric clustering technique with a $O(n^2)$ computational cost, but its embarassingly parallel structure makes it suitable for parallel computing. In this work an OpenMP and a CUDA implementation will be presented and the execution times of each version will be compared. A particular focus will be given to the speedup obtained with the parallel versions for datasets of increasing dimension.

1. Introduction

Mean Shift is a non-parametric clustering algorithm originally presented by Fukunaga and Hostetler [2]. It is based on the *Kernel Density Estimation*, a technique to estimate the underlying probability density function (PDF) of a probability distribution that generated a certain dataset.

The only parameter of the Mean Shift is the *bandwidth*, that controls the smoothing of the resulting PDF (*e.g.* a higher bandwidth will result in fewer but larger clusters). So the main advantage of the Mean Shift is that, unlike K-Means, it does not need to specify the number of clusters.

At each step a *kernel function* is applied to each point belonging to the dataset to shift it in the direction of the local maxima specified by the kernel. The algorithm ends when all points have reached the maxima of the underlying distribution estimated by the chosen kernel. The set of points shifted to a certain local maximum is identified as a cluster. There are many different types of kernel functions, but the most used is the *Gaussian kernel*:

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

The standard deviation σ is the *bandwidth* parameter.

The position in which each point is shifted at each step of the algorithm is computed as a weighted average between the point and all the others, where the weights are calculated with a given kernel function. Suppose x is a point to be shifted and $N(x)$ is the neighborhood of x, a set of points for which K(x_i) $\neq 0$. Let $dist(x, x_i)$ be the distance from the point x to the point x_i. The new position x' where x has to be shifted is computed as follows [6]:

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

The mean shift algorithm applies that formula to each point iteratively until they converge, that is until the position does not change. This means that all the points have reached their corresponding local maximum of the underlying distribution. The algorithm ends when all the points have stopped shifting.

The idea behind the algorithm shows that Mean Shift is embarassingly parallel, because each point can be processed distinctly from the others. This suggests that it is convenient to build a parallel version: this work presents two different parallel implementations, one with OpenMP

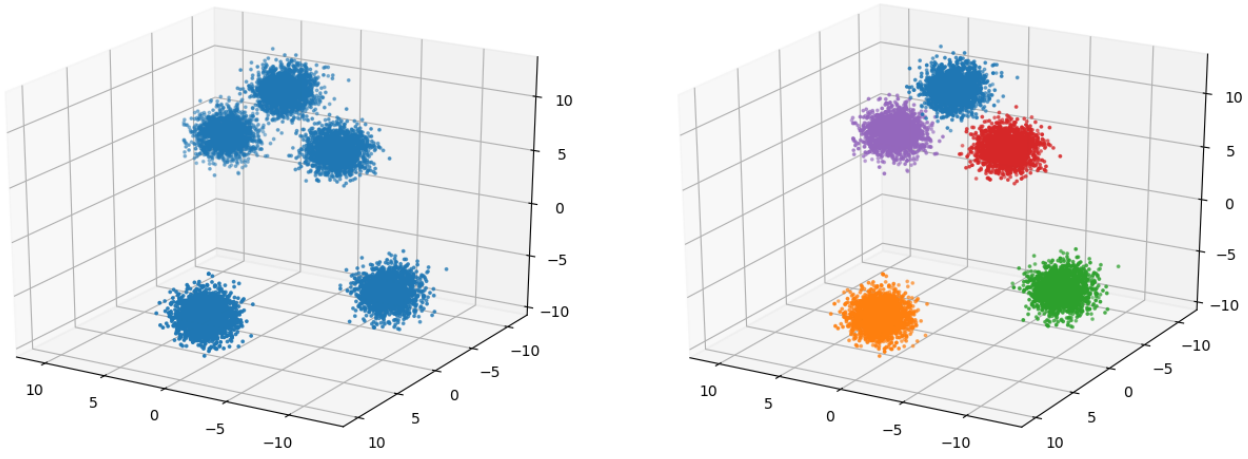

Figure 1. Example of a dataset with 10000 points respectively not clustered and clustered

and the other with CUDA. Also, Mean Shift has a computational cost of $O(n^2)$, so making it parallel can compensate for the increasing execution time and make it suitable for problems with a lot of data.

2. Proposed approach

The proposed sequential implementation is a simple translation of the principle behind the Mean Shift in C++. The **Gaussian kernel** has been chosen as the kernel function and the Euclidean distance has been used to measure the distance between two points. Since the Guassian kernel can't be equal to zero the neighbors of each point are all the other points. The *Algorithm 1* shows the pseudocode of the core of the algorithm:

Algorithm 1 Mean shift core

```
function MEANSHIFT(originalPoints)
    shiftedPoints ← originalPoints
    while iterationIndex < MAX_ITERATIONS do
        for each point p in shiftedPoints do
            p ← SHIFTPOINT(p, originalPoints)
```

The $MAX\_ITERATIONS$ constant represents the number of times each point is shifted, while the BW constant stands for the bandwidth. In this implementation the algorithm is built to work specifically with 3D points (even if modifying it to make it work with other dimensions would be trivial) because a common application of Mean Shift in *Computer Vision* is *Image Segmentation* on RGB images [1].

Algorithm 2 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
```

Figure 1 shows a dataset with 10000 points respectively not clustered and clustered with the algorithm.

2.1. OpenMP version

Using the OpenMP library lets to transform the sequential version into a parallel one with a single **pragma** directive. Indeed, as can be seen in *Algorithm 3*, the OpenMP implementation differs from the sequential one for just a statement.

The pragma directive is used just before the *for* loop: in this way there is no need of any criti-

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)
```

cal section because in each iteration each point is shifted indipendently from the others. It is important to note that a **static scheduling** has been used because in this way the *for* loop is divided statically in chunks of equal size, and this assures that each thread receives the same workload because the number of iterations is fixed from the start.

2.2. CUDA version

Implementing the algorithm with CUDA lets to take advantage of the great number of cores of GPUs. Indeed the processing of each point can be assigned to a different thread. In this work two different versions will be presented: a naive one and a more optimized one that uses tiling and shared memory. From now on it will be used the CUDA terminology [3].

The array of the points has only one dimension, so both the blocks and the grid require only the *x* coordinate to be defined. In particular, *blockDim.x* will be equal to a constant $BLOCK\_DIM$, while the number of blocks will be $\lceil numPoints/BLOCK\_DIM \rceil$.

Both the proposed implementations are built to exploit 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* to allow **coalescing** and to be more L2 cache friendly (in contrast to *Arrays of Structures* and the corresponding strided access):

$$[x_1, \ldots, x_n, y_1, \ldots, y_n, z_1, \ldots, z_n] \quad (3)$$

Moreover, the access to the array is in the form of:

$$blockDim.x * blockIdx.x + threadIdx.x \quad (4)$$

For these reasons all threads in a warp access to consecutive global memory locations. This lets the hardware to coalesce all the accesses into a consolidated access to consecutive DRAM locations increasing performances.

2.2.1 Naive version

In the naive version the algorithm is simply implemented as a CUDA kernel. A wrapper function is also necessary to call the kernel the number of times specified by $MAX\_ITERATIONS$. The wrapper and the kernel can be seen respectively in *Algorithm 4* and *5*.

Algorithm 4 CUDA Naive version Mean Shift core

```
function NAIVECUDAMS(originalPoints)
    shiftedPoints ← originalPoints
    while iterationIndex < MAX_ITERATIONS do
        NAIVEKERNEL(shiftedPoints, originalPoints)
```

Algorithm 5 CUDA Naive version Kernel

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

Before computing the shift of its corresponding point each thread has to check if its index is not greater that the number of points: in this case the thread simply does nothing. This happens when the number of points is not multiple of $BLOCK\_DIM$, therefore more threads have been instantiated than necessary.

2.2.2 Tiling version

Starting from the naive version a more optimized implementation can be developed. Indeed in the *Algorithm 5* can be seen that each thread to compute the shift needs to access ***O(n)*** times in global memory, one for each point. This results in a loss of performance caused by the relatively slow access to global memory.

A way to overcome this problem is to exploit the **Shared Memory** with the *Tiling* pattern, relying on the fact that the original points are read by all the threads but not modified. At first, during the loading phase, each thread loads into the Shared Memory the point which was assigned to, and then it computes a partial shift based on only the points contained in the current tile. Due to the limited amout of Shared Memory (48KB per block, 96KB per *Streaming Multiprocessors* for the *Pascal Architecture* [4]) this process must be repeated for several iterations, until all the points are loaded and used to compute the shift. Let $TILE\_WIDTH = BLOCK\_DIM$ be the number of points contained at once in the Shared Memory, then each thread reduces the number of accesses to the global memory from $O(n)$ to $O(n/TILE\_WIDTH)$.

As explained previously in section 2.2.1, both a wrapper function and a kernel exploiting Shared Memory have been developed (respectively *Algorithm 6* and *Algorithm 7*).

Algorithm 6 CUDA Tiling version Mean Shift core

```
function TILINGCUDAMS(originalPoints)
    shiftedPoints ← originalPoints
    while iterationIndex < MAX_ITERATIONS do
        TILINGKERNEL(shiftedPoints, originalPoints)
```

Algorithm 7 presents some points of particular interest:

- two indexes are required: *idx* refers to the shifting point which the thread was assigned to, *tileIdx* represents the point that the thread has to load into the Shared Memory
- the two $\_\_syncthreads()$ represent necessary thread barriers: the first one assures that each thread has loaded the corresponding point to the Shared Memory before the computing, the second one guarantees that all the threads have computed the partial shift before the next iteration
- during the loading phase a boundary check is performed: if a thread is to load a point that is not in the valid index range it loads a dummy point with a flag value. Then, during the computing of the partial shift, each thread checks if the current point is a dummy one and in that case it simply ignores it: in this way the dummy points will not affect the final value of the shifting point.

Algorithm 7 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
```

3. Experimental results

The metric used to compare the performances of the sequential algorithm with the OpenMP and

the CUDA versions is the **speedup**, computed as:

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

where t_S and t_P are respectively the execution time of the sequential and the parallel implementation.

The datasets used to evaluate the different implementations have been generated with the $make\_blob$ function of *sklearn.datasets* [5]. They are gaussian distributions with 5 centers and standard deviation equal to 1 and are composed by respectively 100, 1000, 10000, 100000 and 250000 3D points.

The $MAX\_ITERATIONS$ constant has been set to 10 because it has been estimated empirically that 10 iterations are enough to make all the points converge to the local maxima, while the value associated to the BW (i.e. bandwidth) constant was 2.

The tests have been 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

To make the results more reliable and representative each execution time has been obtained as the average of the times measured running each test 5 times for the sequential and the OpenMP versions and 15 times for each CUDA implementation.

3.1. OpenMP

To evaluate the performances of the OpenMP implementation it has been executed on each dataset with an increasing number of threads. The results for the 100, 1000, 10000 and 10000 dataset are shown respectively in *figure 2* and *table 1*, *figure 3* and *table 2*, *figure 4* and *table 3*, *figure 5* and *table 4*. Unfortunately both the sequential and the OpenMP implementations could not be executed on the 250000 points dataset because it would have taken too much time, so they have been estimated (as explained later).

For the 100 points dataset (*figure 2* and *table 1*) interestingly the speedup decreases for more than 4 threads. This shows how for such a low

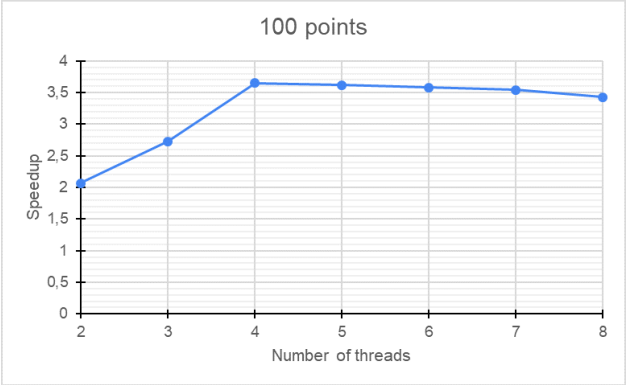

Figure 2. Speedup for 100 points varying the number of threads with OpenMP

Dim	100	
Sequential	0.005384 s	
Thread	**Time**	**Speedup**
2	0.002607 s	2.07
3	0.001973 s	2.73
4	**0.001473 s**	**3.66**
5	0.001487 s	3.62
6	0.001503 s	3.58
7	0.001519	3.55
8	0.001570	3.43

Table 1. Speedup for 100 points varying the number of threads with OpenMP (best result in bold)

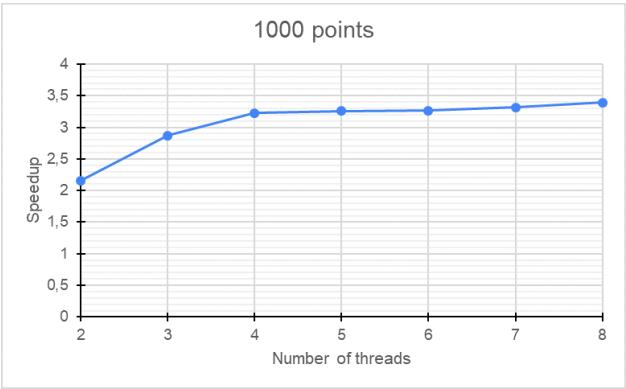

Figure 3. Speedup for 1000 points varying the number of threads with OpenMP

Dim	1000	
Sequential	0.475485 s	
Thread	**Time**	**Speedup**
2	0.220345 s	2.16
3	0.165760 s	2.87
4	0.147428 s	3.23
5	0.145838 s	3.26
6	0.145555 s	3.27
7	0.143497 s	3.32
8	**0.140161 s**	**3.39**

Table 2. Speedup for 1000 points varying the number of threads with OpenMP (best result in bold)

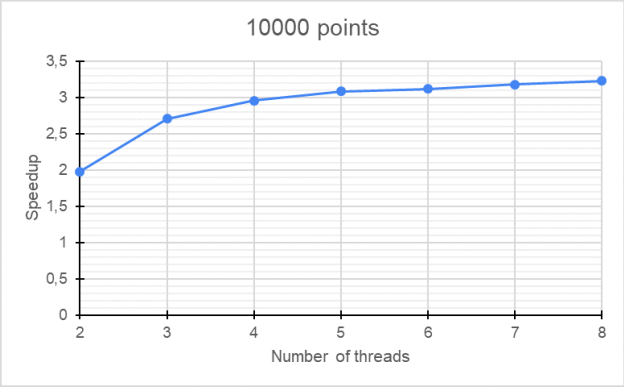

Figure 4. Speedup for 10000 points varying the number of threads with OpenMP

Dim	10000	
Sequential	49.29 s	
Thread	**Time**	**Speedup**
2	24.91 s	1.98
3	18.21 s	2.71
4	16.64 s	2.97
5	15.96 s	3.09
6	15.78 s	3.13
7	15.48 s	3.19
8	**15.25 s**	**3.24**

Table 3. Speedup for 10000 points varying the number of threads with OpenMP (best result in bold)

number of points the overhead of the threads outweighs the gain from using them. As expected this phenomenon disappears for the datasets with more points and the use of more threads lowers the execution time (and consequently increases the speedup). The results show how OpenMP lets to reach a speedup equal to atleast **3** at the expense of a single pragma directive, so OpenMP proves to have an excellent speedup and development cost ratio.

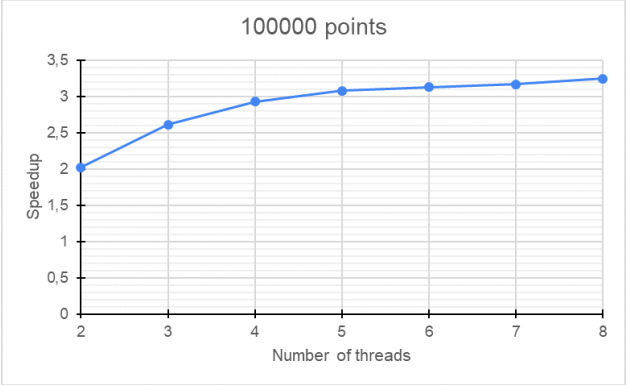

Figure 5. Speedup for 100000 points varying the number of threads with OpenMP

Dim	100000	
Sequential	4560.37 s	
Thread	**Time**	**Speedup**
2	2252.48 s	2.03
3	1744.97 s	2.62
4	1554.59 s	2.94
5	1479.61 s	3.09
6	1456.03 s	3.14
7	1437.89 s	3.18
8	**1403.78 s**	**3.25**

Table 4. Speedup for 100000 points varying the number of threads with OpenMP (best result in bold)

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

Table 5. Execution time for 10000 points for the CUDA tiling implementation varying $TILE\_WIDTH$ (best result in bold)

3.2. CUDA

The first test was aimed at finding the optimal $TILE\_WIDTH$ (and also $BLOCK\_DIM$ because $TILE\_WIDTH = BLOCK\_DIM$), so different execution times were measured for a dataset with 100000 points increasing $TILE\_WIDTH$. As *figure 6* and *table 5* show, the best time was obtained for $TILE\_WIDTH = 64$ so it was used for the other tests.

Then both the naive and the tiling version was

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 6. Global comparison between sequential, OpenMP and Tiling CUDA best results varying dataset dimension († times and speedups are estimated)

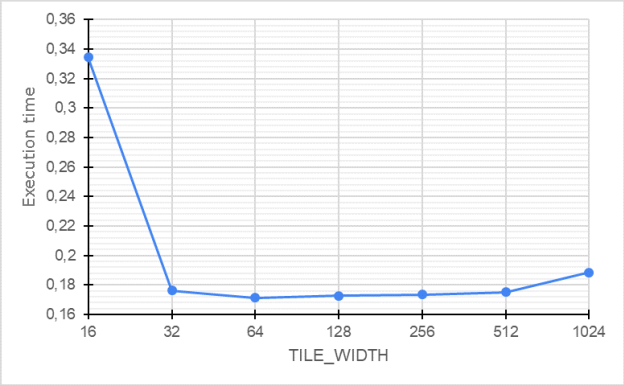

Figure 6. Execution time for 10000 points for the CUDA tiling implementation varying $TILE\_WIDTH$

Dim	CUDA Tiling	CUDA Naive	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

Table 7. Execution time and speedup obtained with naive and tiling CUDA implementations for increasing dataset dimension

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

Table 8. Execution time and speedup obtained with sequential and tiling CUDA implementations for increasing dataset dimension († times and speedups are estimated)

executed on all the datasets. In *table 7* can be seen that exploiting the Shared Memory with the tiling pattern lets to increase the performances of a respectable amount.

The final test was focused on the actual speedup of the tiling CUDA implementation in relation to the sequential algorithm. *Table 8* clearly shows how the use of GPUs lets to considerably increase the performances and to apply the algorithm to otherwise intractable datasets. Indeed it is important to note that the execution time for the 250000 points dataset was not measured but estimated with a quadratic regression (due to the $O(n^2)$) because repeating the test for 5 times would have taken too much time. As expected for datasets with a low number of points (*i.e.* 100 and 1000) the maximum potential of the GPU is not expressed, but then for bigger datasets the speedup increases and stabilize at about **288** when the technical characteristics of the GPU become limiting.

3.3. Comparison

As a final result a global comparison has been conducted and therefore only the best results for each dataset dimension and each implementation have been considered. It is important to note that since the execution time for the 250000 points dataset of the OpenMP implementation was not measured it has been estimated using a quadratic regression and then the corresponding speedup has been computed. In *table 6* can be seen that the CUDA algorithm abundantly **outperforms** both the sequential and the OpenMP ones, at the expense of a more complicated implementation. However OpenMP lets to achieve a noticeable speedup with just a single directive.

4. Conclusions

In this work the Mean Shift clustering algorithm was presented and it was shown how its embarassingly parallel structure makes it suitable for parallel computing. A parallel implementation with OpenMP was developed by adding just a directive and it let to obtain a speedup equal

to more than 3. Then a CUDA implementation (in a naive version and in a more optimized one that uses tiling) was presented and with its about 288 speedup showed how the use of GPUs makes Mean Shift applicable to datasets intractable with a CPU.

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] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on information theory*, 21(1):32–40, 1975.

[3] Nvidia. Cuda documentation. `https://docs.nvidia.com/cuda/`.

[4] Nvidia. Pascal tuning guide. `https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html`.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html`.

[6] Wikipedia. Mean shift. `https://en.wikipedia.org/wiki/Mean_shift`.