Parallel computing final term: mean shift

Angelo D'Amante, Fabian Greavu

Università degli studi di Firenze
angelo.damante@stud.unifi.it fabian.greavu@stud.unifi.it

June 16, 2021

Abstract

In this project we've developed mean-shift algorithm. First we are going to introduce the Mean Shift algorithm and definition of density function. Then we show the C++ sequential implementation and two parallel versions, CUDA in C++ and Team Threads in OpenMp. Finally, we will show the speedups obtained with CUDA with respect to various tile-width, with OMP compared to various team threads. All the code can be found on github at (1).

I. Introduction

The Mean shift is a non-parametric feature-space analysis technique for locating the maxima of a density function, a so-called mode-seeking algorithm. It is a procedure for locating the maxima the modes of a density function given discrete data sampled from that function. (2).

A. Definitions

Given n data points x_i, $i = 1, \cdots, n$ on a *d-dimensional* space $\mathbb{R}^d$, the multivariate kernel density estimate obtained with kernel $K(\cdot)$ and window radius h is

$$f(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\Big(\frac{x - x_i}{h}\Big) \tag{1}$$

for radially symmetric kernels, it suffices to define the profile of the kernel $k(x)$ satisfying

$$K(x) = c_{k,d} k(||x||^2) \tag{2}$$

where $c_{k,d}$ is a normalization constant which assures $K(x)$ integrates to 1. The modes of the densisty function are located at the zeros of the gradient function $\nabla f(x) = 0$. (3)

Typically a Gaussian kernel on the distance to the current estimate is used,

$$K(x) = \exp(-\frac{x^2}{2\sigma^2}) \tag{3}$$

B. Mean Shift

Starting from the initial estimate x and the choice of the kernel $K(x)$ (as Gaussian kernel), the weighted mean of the density, determined by K is

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)} \tag{4}$$

where $N(x)$ is a neighborhood of x, a set of points for which $K(x_i) \neq 0$.

The difference, $m(x) - x$ is called *mean shift* and the algorthm, at this point, sets x with $m(x)$ and repeats the estimation untill $m(x)$ converges.

II. Sequential Version

The general mean shift procedure for a given point x at step t is as follows,

1. Compute the mean shift vector $m(x^t)$

2. Translate the kernel window by $m(x^t)$
$$x^{t+1} = x^t + m(x^t)$$
3. Iterate previous step until convergence.

Let be I the number of iterations and n the size of dataset, show the pseudo-code,

Algorithm 1 Mean Shift algorithm

```
procedure MeanShift(original_points)
    new_data = original_points
    while iteration < I do
        for p in new_data do
            num = 0
            den = 0
            for op in original_points do
                d = Distance(p, op)
                gaussian = K(dist)
                num = num + op * gaussian
                den = den + gaussian
            end for
            p = num/den
        end for
    end while
    data = reduce_centroids(data, ϵ)
    return data
```

With:

- the `Distance` method, computes euclidean distance between two points,
- $K(\text{dist})$ computes a gaussian kernel defined in (3),
- the `reduce_centroids` method, reduces the converged data points to a subset that ideally, should be the cluster centers.
  This is done with the help of a variable, $\epsilon \in \mathbb{R}$, that has to be set by the user. This quantity represents the minimum distance for which two points can be considered belonging to the same cluster.

The sequential version has been implemented in C++ and the time complexity is $O(In^2)$.

III. CUDA Implementation

CUDA (Compute Unified Device Architecture) is a hardware architecture for parallel computing. Staring from the Sequential mean-shift we can optrimizie its runtime by computing the points location over each iteration in parallel using CUDA cores.
The big advantage of CUDA is being able to use lots of cores for simple computation (from hundreds to thousands). Each core will be called from the CPU and executing a procedure (called **kernel**).
The MeanShift algorithm will change into following algorithm:

Algorithm 2 Mean Shift CUDA main

```
procedure MeanShift(original_pts)
    new_data = original_pts
    for i = 0; i < NUM_ITER; i++ do
        cuda_kernel«BLOCKS, THREADS»(original_pts, new_data)
        swap(original_pts, new_data)
    end for
    data = reduce_centroids(original_pts, ϵ)
    return data
```

First we are going to introduce a simple **naive** version and a second version using **Shared Memory** and **Tiling** technique.

A. Naive CUDA mean-shift

Starting from algorithm 1 a straightforward (naive) idea is to compute each point update in main for loop on a single CUDA core in parallel.
With that in mind we can simply move the body of the main for cycle inside a CUDA **Kernel** that will be called from algorithm 2.

B. Shared Memory CUDA mean-shift

The naive version uses direct GPU global memory in order to access all datas to calculate weights for current point. This access can take time and make warps smaller.
The idea is to copy the right amount of datas from one BLOCK of datas into a shared memory array with fixed size and access it in order

Algorithm 3 CUDA naive

```
procedure MS_NAIVE_KERNEL(pts_a, pts_b)
    tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    x = tid*2
    y = x+1
    if tid < POINTS_NUM then
        new_p = [0,0]
        for i in POINTS_NUM do
            d = Distance(pts_a[x], pts_b[y])
            if tid < POINTS_NUM then
                new_p = calc_weights(...)
            end if
        end for
    end if
    pts_a[xy] = new_p
end procedure=0
```

to speedup computation.
Having access on a small portion of the datas, a tiling methodology is needed (split datas into tiles and load them accordingly).
Naive algorithm becomes:

Algorithm 4 CUDA Sahred Memory

```
procedure MS_NAIVE_KERNEL(a, b)
    __shared__ float local_data[TILE_W*2];
    __shared__ float flag_data[TILE_W];
    for t in BLOCKS do
        tid = (blockIdx.x * blockDim.x) + threadIdx.x;
        x = tid*2
        y = x+1
        if tid < POINTS_NUMBER then
            local_data = copy_data(tid, t)
        end if
        __syncthreads();
        for i in TILE_W do
            new_p = [0,0]
            for i in POINTS_NUM do
                d = Distance(a[x], b[y])
                if tid < POINTS_NUM then
                    new_p = calc_weights(...)
                end if
            end for
        end for
        __syncthreads();
    end for
    if tid < POINTS_NUM then
        a[xy] = new_p
    end if
end procedure
```

IV. OpenMP Implementation

OpenMP (Open Multiprocessing) is an API for shared-memory parallel programming. It follows the fork-join programming model. The idea is speeding up parts of the application by exploiting CPU parallelism (4).

The advantage of OpenMP is that it does not require sequential program restructuring. In fact, the sequential part is elaborated by thread master and parallel region is elaborated by threads of CPU follows a static or dynamic schedulation. We only need to add compiler directives to transform the sequential program into a parallel program.

We can notice that with a single line code (`pragma command`) it is possible switch from a sequential to parallel version. We also note that the problem deals with independent data, so the absence of critical sections made this implementation particularly simple.

A. Schedulations

There are two schedulations for `#pragma omp parallel for` directive,

- `schedule(static)`: Each thread has a predefined set of iterations to perform. It is possible set a number of threads with the clause `num_threads(int)`.
- `schedule(dynamic)`: Each thread performs an iteration as soon as possible. When it finishes, it takes another one.

In this project, both versions have been implemented and we will see in the results section that goes dynamic version is more performant.

Algorithm 5 Mean Shift algorithm OMP

```
procedure MeanShift(original_points)
    new_data = original_points
    while iteration < I do
        #pragma omp parallel for
        for p in new_data do
            num = 0
            den = 0
            for op in original_points do
                d = Distance(p, op)
                gaussian = K(dist)
                num = num + op * gaussian
                den = den + gaussian
            end for
            p = num/den
        end for
    end while
    data = reduce_centroids(data, ε)
    return data
```

V. Results

In this section we evaluate the speedups obtained with the two parallel implementations, CUDA and OpenMP. The datasets evaluated are $\{500, 1000, 2000, 5000, 10000\}$. Several centroids were considered.

The timings of the sequential version was taken with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.

Dataset	Timing [ms]
500	872
1000	3481
2000	13692
5000	85434
10000	339862

Table 1: *Timings for Sequential Version.*

We need these results to calculate the speedup for four parallel versions, CUDA-naive, CUDA-shared, OMP-static and OMP-dynamic.

$$S = \frac{t_s}{t_p} \tag{5}$$

where t_s is sequential timing and t_p is parallel timing.

A. CUDA Speedupds

We followed two approaches for CUDA version: Naive and Shared Memory. Both results were calculated on Nvidia GTX 1050ti (mobile version).

For **naive version**, the GPU timings obtained are shown in 6.

Dataset	Number of threads			
	8	16	32	64
500	5.05	5.28	5.23	6.16
1000	11.35	10.01	9.58	9.51
2000	41.67	22.46	19.92	19.95
5000	199.86	108.22	55.51	55.11
10000	685.08	405.86	217.88	159.28
Dataset	Number of threads			
	128	256	512	1024
500	4.96	6.3626	5.04	5.05
1000	10.05	9.9049	10.13	11.97
2000	19.93	19.739	20.22	22.95
5000	54.95	57.6777	48.10	47.431
10000	185.8	187.389	216.41	219.31

Table 2: *GPU Timings [sec] for CUDA naive version.*

With respect to table 2 we can divide by sequential time and obtain following speedups (Tab 3

Dataset	Number of threads			
	8	16	32	64
500	177.00	169.45	171.10	145.19
1000	318.71	361.28	377.31	380.26
2000	350.10	649.60	732.34	731.36
5000	449.83	830.70	1619.39	1631.25
10000	523.69	883.97	1646.59	2252.36
Dataset	Number of threads			
	128	256	512	1024
500	180.28	140.66	177.25	177.12
1000	359.67	365.27	356.86	302.04
2000	732.01	739.24	721.52	635.65
5000	1636.09	1558.74	1868.81	1895.47
10000	1930.37	1914.57	1657.78	1635.87

Table 3: *GPU Speedups for CUDA naive version.*

For **shared memory versin**, the GPU timings obtained are shown in Tab. 4.

Dataset	Number of threads			
	8	16	32	64
500	4.61	3.73	3.29	3.24
1000	11.99	8.18	7.05	6.96
2000	40.37	19.15	14.55	13.48
5000	195.69	91.79	43.16	47.80
10000	655.66	308.41	168.60	187.15
Dataset	Number of threads			
	128	256	512	1024
500	3.35	3.93	3.14	10.15
1000	7.13	7.89	7.95	11.43
2000	14.07	14.68	14.89	23.24
5000	49.01	54.46	54.33	54.33
10000	164.30	177.30	175.15	201.79

Table 4: *GPU Timings [sec] for CUDA shared memory version.*

With respect to table 4 we can divide by sequential time and obtain following speedups (Tab 5

Dataset	Number of threads			
	8	16	32	64
500	193.85	239.47	271.67	275.52
1000	301.59	442.04	512.96	519.48
2000	361.42	761.65	1002.47	1082.38
5000	459.41	979.40	2082.73	1880.72
10000	547.18	1163.28	2127.90	1916.99
Dataset	Number of threads			
	128	256	512	1024
500	267.15	227.54	284.25	88.13
1000	506.97	458.47	454.89	316.41
2000	1036.92	993.43	979.49	627.79
500	1834.35	1650.60	1654.73	1654.59
10000	2183.63	2023.47	2048.35	1777.87

Table 5: *GPU Speedups for CUDA shared memory version.*

After checking all the speedups in both versions we can see that better results were achieved using TILE_WIDTH between 64 and 128.

Shared memory also has better performances with bigger datasets (at least 2000 points) and major applications uses a huge amunt of datas. Still naive implementation works really well.

B. OpenMP Speedupds

As mentioned above, 2 scheduling techniques have been implemented. Static and Dynamic, and in both cases the results were taken with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.

For **dynamic schedulation**, the CPU timings obtained are shown in 6.

Dataset	Timing [ms]
500	129
1000	476
2000	2017
5000	13470
10000	58436

Table 6: *CPU Timings for OMP Static Schedulation.*

comparing the two tables 1 and 6, the speedup (calculated with 5) obtained is shown in table 7.

Dataset	Speedup
500	6.760
1000	7.313
2000	6.788
5000	6.343
10000	5.816

Table 7: *Speedups for OMP Dynamic Schedulation.*

For **static schedulation**, the results obtained were taken considering the possibility of launching from 1 to 8 threads.

Comparing table 8 with sequential timings in table 1, the speedup obtained is shown in table 9.

We can see that in general the performance of the dynamic version is better. In fact, due to the structure of the problem, given by the density estimation, this is one of the rare cases in which the dynamic scheduling is more performing.

Dataset	Number of threads				
	1	2	4	6	8
500	0.88	0.86	0.25	0.18	0.17
1000	3.40	1.72	0.91	0.64	0.72
2000	13.93	6.90	3.67	2.56	3.03
5000	85.59	43.90	24.26	16.97	18.85
10000	345.5	182.7	101.4	80.61	79.54

Table 8: *CPU Timings [sec] for OMP Static Schedulation.*

Dataset	Number of threads				
	1	2	4	6	8
500	0.946	1.856	3.523	4.543	4.068
1000	0.951	1.891	3.509	4.836	4.317
2000	0.979	1.950	3.670	4.444	4.022
5000	0.969	1.939	3.261	4.777	4.353
10000	1.008	1.930	3.371	4.773	4.598

Table 9: *Speedups for OMP Dynamic Schedulation.*

In particular, the worst case is due to the static version with only one thread, as expected.

VI. Conclusions

In this article we approached the mean-shift algorythm aiming at creating multiple paralle versions using CPU cores and GPU CUDA cores. Being able to implement the points shifting cycle in parallel, we optimized it to get a better speedup in some cases.

When using CPU cores with OpenMP we gained maximum speedup of 4700 with 6 threads using dynamic schedulation. Higher scores were made on bigger datasets. Static version starts slow but with bigger datasets it struggles to speedup.

When using GPU cores with CUDA we gained maximum speedup of 2183.63 with 128 number of threads using shared memory version. Compared with the naive version it brings up the speedup on bigger datasets as predicted.

References

[1] Fabian Greavu Angelo D'Amante. mean-shift-in-parallel. `https://github.com/fabian57fabian/mean-shift-in-parallel`, 2021.

[2] Wikipedia contributors. Mean shift — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Mean_shift&oldid=992487504`, 2020. [Online; accessed 15-May-2021].

[3] Mean shift clustering. `https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/TUZEL1/MeanShift.pdf`.

[4] Openmp documentation. `https://openmp.org/wp-content/uploads/spec30.pdf`.