AGH	Module: **Introduction to CUDA and OpenCL**	Report title: **Kernel Density Estimation**		
	Instructor: **dr hab. inż. Szumlak Tomasz**			
	Name and surname: **Marcin Filipek** **Anca Dărăbuț**	Group: **11**	Date: **24.01.2020**	Grade:

1. short introduction

Kernel density estimation represents a method of estimating the probability density of a random variable using a sample of observations.

The function used for this purpose is a non negative, real-valued and integrable one, called kernel. It has the following properties:

- it is a probability density function
- the distribution is symmetric around the chosen point

In order to perform the kernel estimation, the steps mentioned below were followed:

1. The number of samples was chosen
2. The kernels were drawn around each sample (centered around it)
3. The abovementioned kernels were combined point by point
4. Normalizing the distribution

The kernel used in the main part of this project is the Gaussian probability distribution function, having the following formula:

$$K(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

There are many other kernels, but this one has been chosen due to its smooth character.

The estimation of each point is determined with the formula:

$$\hat{f}(x) \frac{1}{mh} \sum_{i=1}^{m} K\left(\frac{x - x_i}{h}\right),$$

2. functionality description

The starting point was the CPU base-line implementation, in order to check the functionality of the program, then we moved forward to the multi-threaded version.

The CPU version had 2 nested loops that could be easily unrolled using the multi-threaded one, thus accelerating the kernel density estimation code (the proof can be found in the next section).

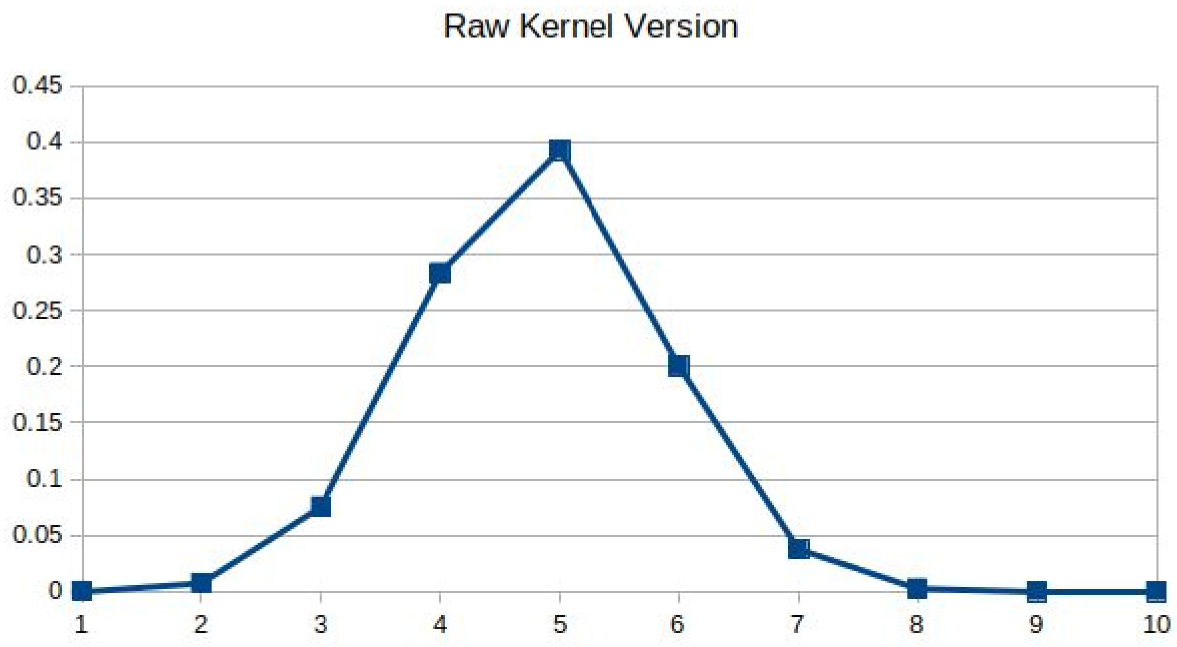
For the multi-threaded kernel density estimation program, we tried to implement different versions and improving the estimation curve.

1. Raw version

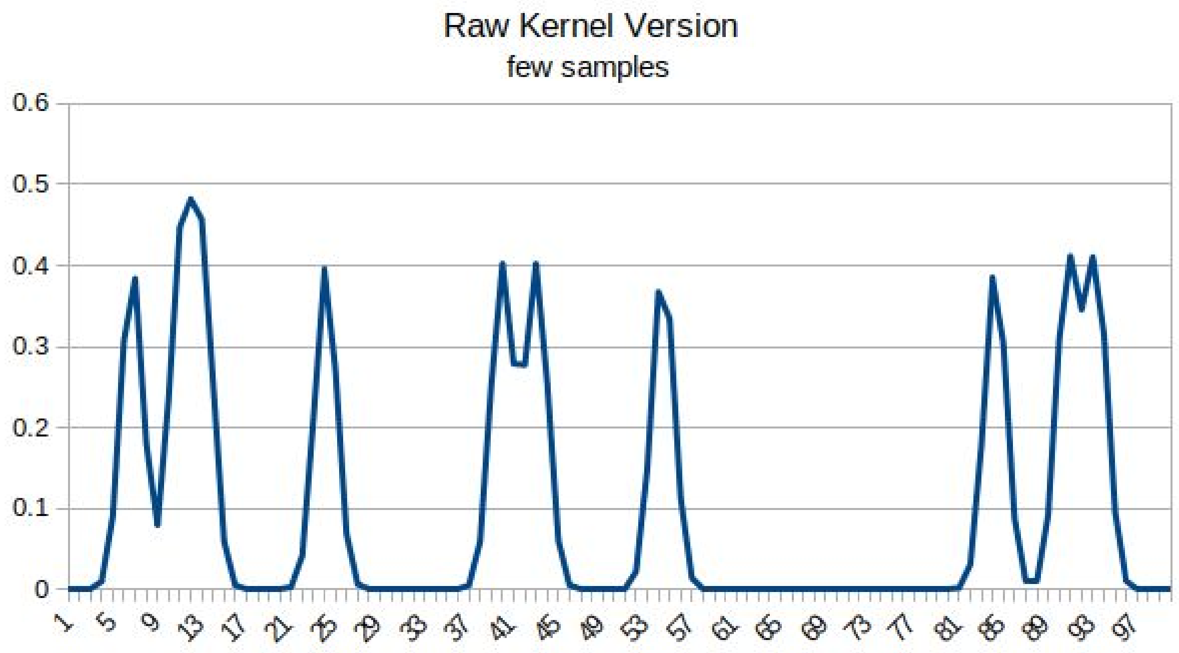
This is the basic version of kernel density estimation, without any enhancements, used for checking whether multi-threaded version is correctly implemented. For comparison, two different cases were analyzed.

1.1. small data size

The first case shows how the program behaves in case of small data size, hence there are 10 points and 1 sample.

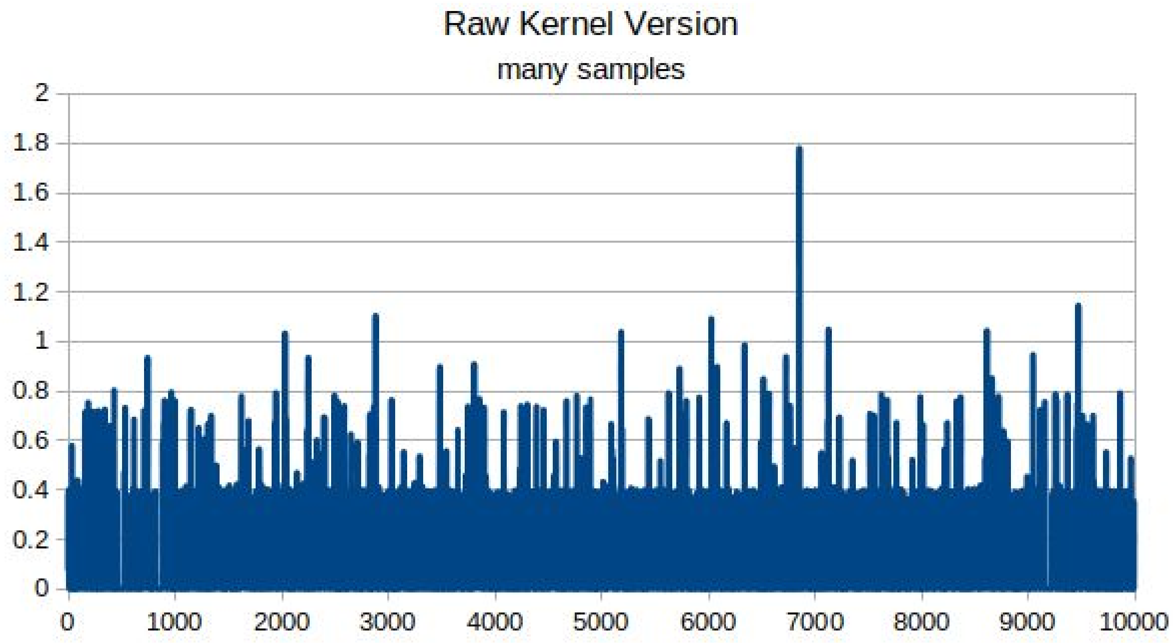


In order to have pattern for this case in further comparisons, the subsequent data sizes were used: 100 points, 10 samples.



1.2. big data size

For the second case, when data is involved (10000 points and 1000 samples), the plot looks like this:

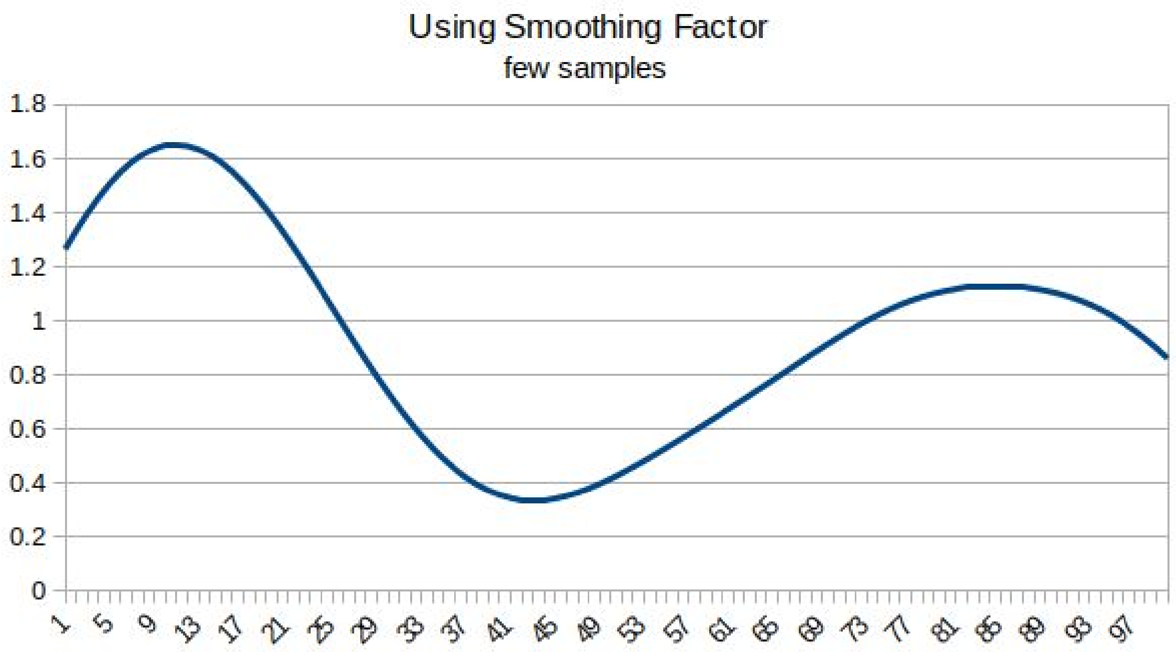


It can be noticed that the tendency in case of big data structures is to go to the uniform distribution.

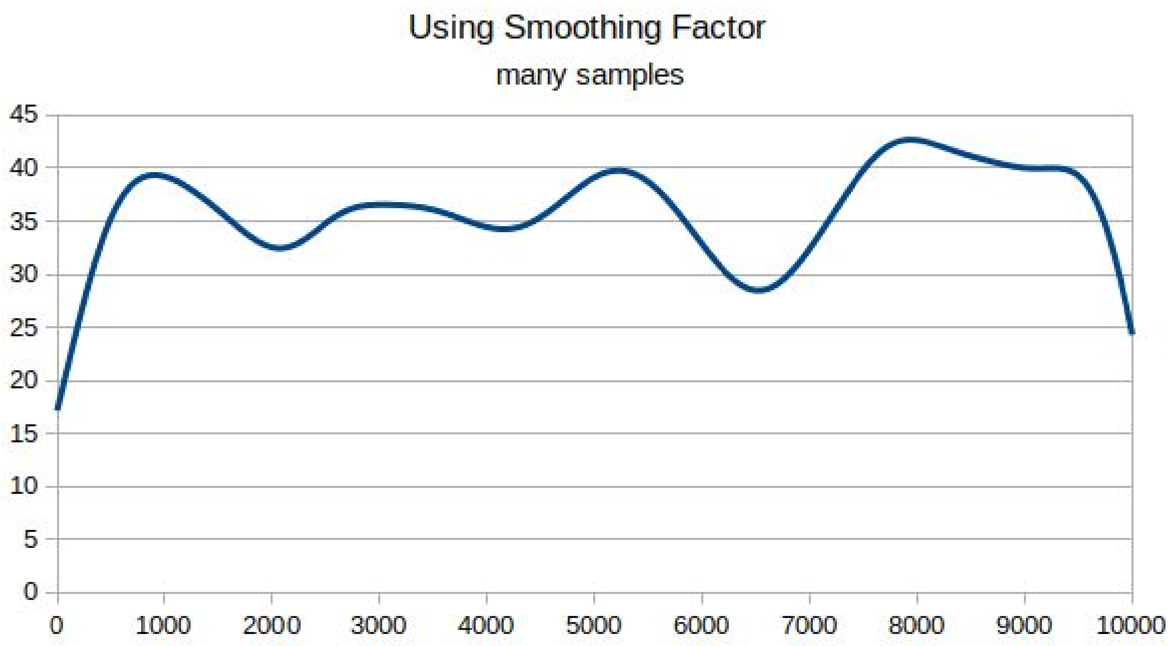
2. Adding smoothing

Smoothing factor allows us to adjust the number of local extrema. The argument of the kernel is divided by the smoothing factor.

2.1. small data size



2.2. big data size

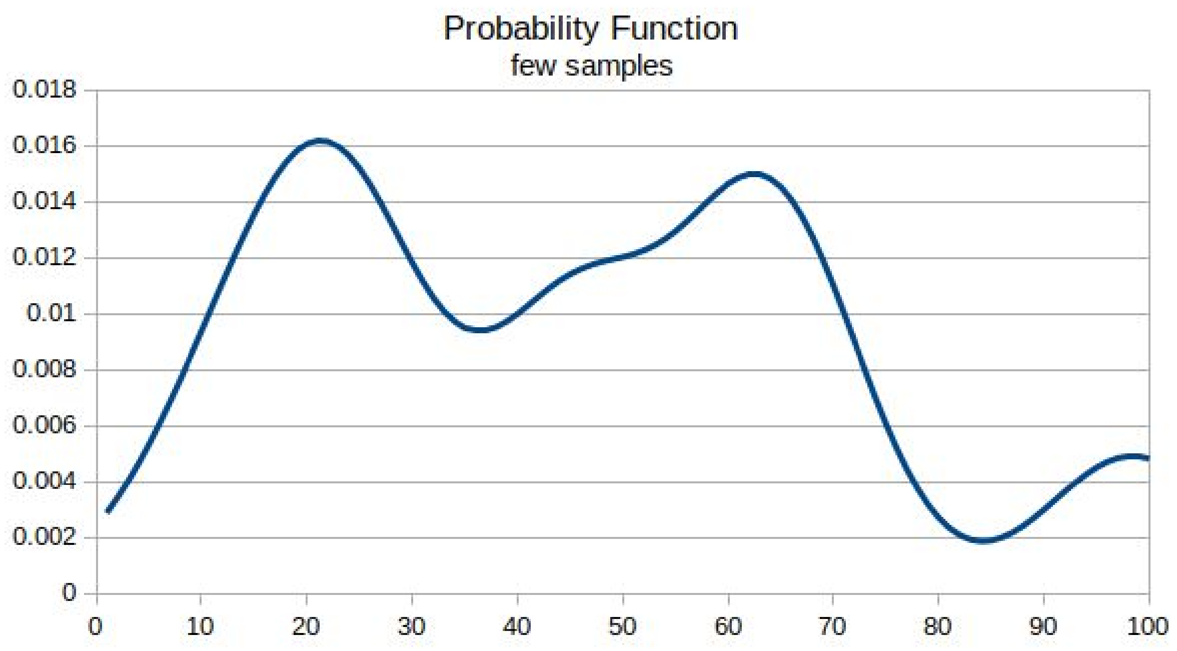


After performing smoothing, the curve is rounded and depicts the real shape of the probability function, without being normalized.

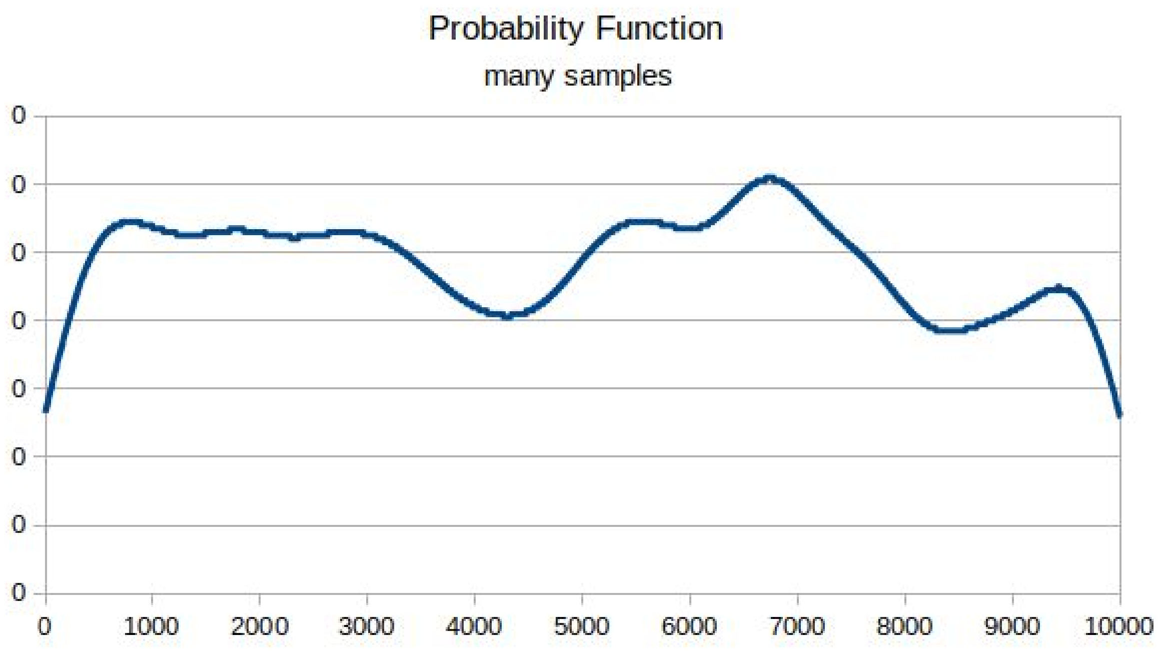
3. Normalizing

Normalization implies dividing the value in each point by the number of samples used for computation. This is done in order to have the area under the graph equal to 1, hence obtaining the probability function.

3.1. small data



3.2. big data



As it can be noticed, a great number of samples can be problematic when it comes to normalization, because the memory of the computer is limited and, when we divide the values by great numbers, the error is significantly increased. Consequently, the appearance of the graph is pixeled. Moreover, we were unable to show the real values from the Oy axis.

3. time profiling

As it was mentioned in the previous section, the times for both CPU and GPU implementations were measured and included in the table below:

points/samples	CPU function	GPU kernel
1000/100000	32092.321 ms	2767.907 ms
1000000/1000	31926.446 ms	1448.671 ms

The multi-threaded version is significantly faster, 10 times in the former case and more than 20 times for the latter one.

For the multi-threaded kernel density estimation program, some more detailed time profiling was done, in order to study its behaviour for different number of points and samples.

points/samples	time [ms]	points/samples	time [ms]
1000/100	0.562	100/1000	3.579
1000/1000	3.788	1000/1000	3.796
1000/10000	35.988	10000/1000	18.710
1000/100000	336.344	100000/1000	183.138
1000/1000000	2724.917	1000000/1000	1398.802

When the number of samples is increased linearly without changing the number of points, the time grows linearly.

Each thread calculates one point in parallel, thus the increase of time is different, as it depends on the time needed to initialize such a great grid and to synchronize a growing number of threads.