# GPU Implementation of Self Organizing Maps

Computação de Alto Desempenho - $1^{st}$ project

2018/2019

## 1 Goal

The goal of this project to develop a GPU implementation of Self Organizing Map (SOM) algorithm, and compare its performance (speedup, efficiency and cost) against a given sequential implementation of the algorithm. You may implement your solution in either CUDA or OpenCL. No higher level frameworks, such as Thrust, SkePU, TensorFlow, Marrow, or others, are allowed. The project must be carried out by a group of, at most, 2 students.

## 2 Self Organizing Map

SOM [1] is a very popular artificial neural network model that is trained via unsupervised learning, meaning that the learning process does not require human intervention and that not much needs to be known about the data itself. The algorithm is used for clustering (feature detection) and visualization in exploratory data analysis. Application fields include pattern recognition, data mining and process optimization. If you are curious about the fundamentals and the application of the algorithm, site `https://www.superdatascience.com/blogs/the-ultimate-guide-to-self-organizing-maps-soms` is a good starting point. Note however, that you do not need this information to accomplish the task asked in this project.

The SOM algorithm is presented in Algorithm 1. The fitting of the model to the input dataset is represented by a map (represented by variable *map*), which is an array of $nrows * ncols$ vectors of $nfeatures$ features, i.e. a tensor of size $nrows \times ncols \times nfeatures$. The algorithm begins by initialing *map* with vectors of random values, and then, for each input $i$ performs the following steps:

1. Compute the distance from $i$ to all vectors of *map*. The *distance* function may be any. You will be asked to implement 2 functions.

2. Compute the Best Matching Unit ($bmu$), which is the vector closest (with minimum distance) to $i$. Note that the $argmin$ function returns the coordinate of the map where the $bmu$ is.

3. Update the map, given the input $i$ and the $bmu$.

With regard to the *update_map* procedure, several learning rates may be considered. In this project you will consider only the one given by formula

$$learning\_rate(t) = \frac{1}{t}$$

**Algorithm 1** The SOM algorithm

$nrows$ – number of rows of $map$
$ncols$ – number of columns of $map$
$nfeatures$ – number of features in each input vector
$map$ – tensor of loats of size $nrows \times ncols \times nfeatures$.
$max\_distance$ – maximum distance between two vectors in the initial configuration of the map.

**procedure** SOM($inputs$)
    $map \leftarrow$ initialize with $n$ random vectors with values $\in [0, 1]$
    $max\_distance \leftarrow \sqrt{nrows^2 + ncols^2}$
    $t \leftarrow 0$ current iteration of the algorithm
    **for all** $i \in inputs$ **do**
        $t \leftarrow t + 1$
        $distances \leftarrow \forall_{j,k}\ (distance(map[j][k], i))$
        $bmu \leftarrow argmin(distances)$
        $map \leftarrow update\_map(t, i, bmu, \#inputs)$
    **end for**
**end procedure**

**procedure** UPDATE_MAP($t, i, bmu, input\_count$)
    $learn\_rate \leftarrow learning\_rate(t)$
    $neighborhood \leftarrow \forall_{j,k}\ (neighborhood\_function(t, bmu, (j, k), input\_count))$
    $map \leftarrow \forall_{j,k}\ (map[j][k] + (learn\_rate * neighborhood[j][k] * (i - map[j][k])))$
**end procedure**

**function** NEIGHBORHOOD_FUNCTION($t, bmu, current\_point, inputs\_count$)
    $theta \leftarrow (max\_distance/2) - ((max\_distance/2) * (t/inputs\_count))$
    $sqrDist \leftarrow |bmu - current\_point|^2$
    $n \leftarrow e^{-(sqrDist/theta^2)}$
    **return** $n > 0.01\ ?\ n : 0$
**end function**

# 3 GPU Implementation

You must implement a C/C++ program that receives the following command line:

```
gpu_som number_rows number_columns datafile outputfile [distance]
```

where `number_rows` and `number_cols` denote, respectively, the number of rows and columns of the map, `datafile` is the name of the file holding the input data set, and `outputfile` is the name of the file to where the final state of the map must be written. `distance` is the parameter that allows the user to choose between the two distance functions to implement. It is a optional parameter defaulted to the Euclidean distance.

Given this configuration, your program must execute in the GPU as much of the presented SOM algorithm as possible. In particular, the SOM map must reside is GPU's memory and be modified there, as it receives inputs read from the input file and transferred to the GPU. A close analysis to Algorithm 1 will unveil several massively parallel computations, such as the ones that are performed for all the vectors of the matrix.

The map must only be transferred to the host explicitly via a function implemented for the purpose, with a name such as `get_map`. You must naturally do this at the end of the computation, to store the map's final state to the output file, but you can use it for debugging purposes (not in the version to evaluate for performance).

In order for you to test and evaluate your solution, several input data files will be provided in the next few days.

**Distance Functions:** As mentioned in the previous section, you must implement two distance functions:

**Euclidean distance** $distance(v,u) = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \cdots + (u_n - v_n)^2}$

**Cosine distance** $distance(v,u) = 1 - \frac{u_1 \times v_1 + u_2 \times v_2 + \cdots + u_n \times v_n}{\sqrt{u_1^2 + u_2^2 + \cdots + u_n^2} \times \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}}$

**Performance Measurements:** You must calculate the execution time of the versions[1] your algorithm from the moment the SOM map is initialized (do not include this initialization) up until the map is written to the output file. These execution times must be compared against a sequential version that will be given in the next few days.

**Report:** Along with the code of your solution, you must deliver a report of, at most, 5 pages presenting your solution, your experiment results and your conclusions. Concerning the solution, focus on explaining which parts of the algorithm are executed on the GPU, and describing the algorithms you devised to accomplish such execution.

[1] One version for each distance function.

## References

[1] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, Sep 1990.