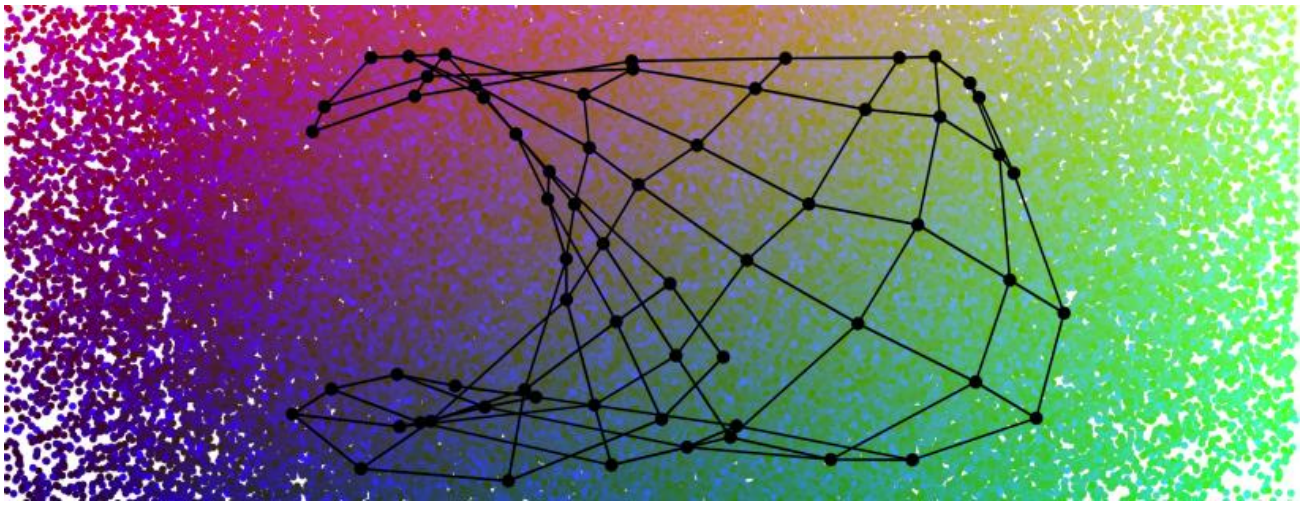
GPU's Implementation of Self-Organising Maps
(S.O.M. Algorithm)



HIGH PERFORMANCE COMPUTING

Integrated Master (BSc. + MSc.) of Computer Science and Engineering

Faculty of Sciences and Technology of New University of Lisbon

(FCT NOVA | FCT/UNL)

2018/2019 - 2nd Semester

PREPARED BY

Rúben André Barreiro (Student no. 42648)

r.barreiro@campus.fct.unl.pt

Self-Organising Maps

What is a Self-Organising Map?

"A Self-Organising Map (S.O.M.) or Self-Organising Feature Map (S.O.F.M.) is a type of Artificial Neural Network (A.N.N.) that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map, and is therefore a method to do dimensionality reduction. Self-Organising Maps differ from other Artificial Neural Networks as they apply competitive learning as opposed to error-correction learning (such as backpropagation with gradient descent), and in the sense that they use a neighborhood function to preserve the topological properties of the input space."

[***Wikipedia*** - *https://en.wikipedia.org/wiki/Self-organizing_map*]

The algorithm is used for clustering (feature detection) and visualization in exploratory data analysis. Application fields include pattern recognition, data mining and process optimisation.

Problem

So, what's the problem? And how to mitigate it or solve it?

Sometimes, in Computer Science, to compute and execute tasks that require large computational resources and too much run-time, such as very complicated numerical simulations, it's better to use some kind of parallelisation and high performance computing mechanisms.

The CPU (if it's not specified any mechanism for parallelisation), usually execute tasks in a sequential way, even if their work processes, that are being processed it's not dependable of each other.

One recent way to mitigate this problem with the large quantity of computational resources and run-times, was introduced with the GPU computing.

"The GPU computing is defining a new, supercharged law to replace Moore's law. It starts with a highly specialized GPU parallel processor and continues through system design, software, algorithms, and optimized applications. Each GPU-accelerated server replaces dozens of commodity CPU servers, delivering a dramatic boost in application throughput and cost savings."

[***NVIDIA*** - *https://www.nvidia.com/en-us/high-performance-computing/*]

CUDA

CUDA API (Compute Unified Device Architecture)

"CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing — an approach termed GPGPU (General-Purpose Computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels."

"The API includes a set of CUDA ISA's (Instruction Set Architecture) instructions and the parallel computing engine on the GPU. It exposes the different memory types of the board and forces the developer to configure the global memory accesses, the cache, the amount and the layout of the threads. The developer will also be responsible for staggering activities between the GPU and the CPU."

[***Wikipedia*** - *https://en.wikipedia.org/wiki/CUDA*]

Project's Goal

Implementing a Self-Organising Map, using GPU's programming and parallelisation to improve the performance of the overall computing

The goal of this project it's to develop a GPU's implementation of the Self-Organising Map (S.O.M.) Algorithm, using parallelism and tunings for the computing of kernels' executions. And then, compare the overall performance (analysing some aspects as velocity, efficiency and cost) of that GPU's implementation with a sequential implementation, for the same algorithm.

It was asked to be developed a solution, using the CUDA or OpenCL APIs.

In this report will be explained how the algorithm and respectively functions was developed, using the previously mentioned aspects and mechanisms.

In this report, will be also shown some experimental results and some comparisons against a given sequential implementation of this same algorithm.

At last, at the end of this report, will be taken and shown some conclusions about the collected experimental results and some comparisons made between the multiple implementations.

GPU's Implementation

Explanation of programming of the kernels of the GPU's Implementation, using the CUDA API

The Algorithm to be Implemented

Algorithm 1 The SOM algorithm

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

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

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

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

Where *learning_rate(t)* is given by the following formula:

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

It was implemented two CUDA-based versions of the algorithm (one version with the four core kernels asked and, a version based on the first one but with another some parallelism optimisations in the distances' kernels and using CUDA's Streams). The implementation of the four kernels will be explained next, as also, some other optimisations performed in another version.

Each CUDA's kernel launches a grid (or an array/vector) of Threads, divided by Blocks of the same dimension, where every Thread and Block are identified by an ID.

To access the ID of the current executing Thread in the overall grid of Threads, by the following formula:

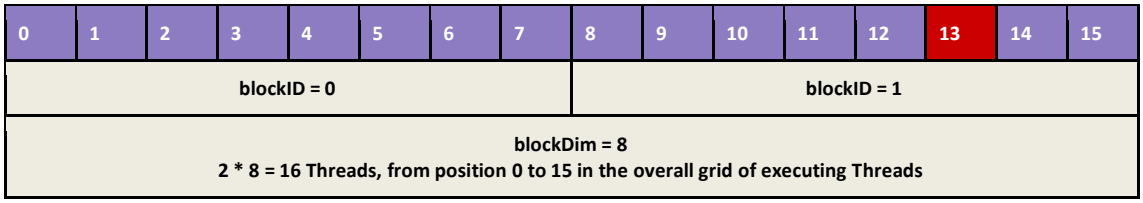
- *executingThreadIndex* = (*blockIndex* x *blockDim*) + *threadIndex*

Where:

I. The *blockIndex* it's the ID of the block on which the current executing Thread it's computing;
II. The *blockDim* it's the dimension of the blocks launched inside the overall grid;
III. The *threadIndex* it's the ID of the the current executing Thread which it's place inside the array/vector of the currently executing block;

Example:

1) To access to the executing Thread in the overall grid with the ID no. 13, in a grid with blocks of dimension of 8 Threads per each one, like the following one:



2) It's the necessary to calculate the first position of the block where it's placed the executing Thread that's pretended to access. Since the Thread with the ID no. 13 it's placed in the block with the ID no. 1, we have:

   - blockIndex x blockDim = 1 x 8 = 8

3) And finally it's applied some kind of offset given by the adinitionaly sum of the ID of the current executing Thread which it's placed inside the array/vector of the block that's being accessed. Since the Thread with the ID no. 13 it's the Thread with the ID no. 5 placed in the block with the ID no. 1, we have:

   - (blockIndex x blockDim) + threadIndex = (1 x 8) + 5 = 8 + 5 = 13

The computed results are placed in this arrays/vectors or grids, in the GPU's memory, and it's almost always necessary to the user to transfer the related data to the CPU, and sometimes, finishes the computing process locally in the CPU memory. Basically, all the computing processes that need parallelism must be transfer from the CPU's memory (Host) to the GPU's memory (Device) and, at the end, it's necessary make the opposite transfer flow back, from the GPU's memory (Device) to the CPU's memory (Host).

1. Distance Function

For the *Distance* function, was implemented five distances' functions (which two of them, was mandatory to implement and are the default ones in the application, the others are optional to use). The distances' functions implemented are the following:

1) ***Euclidean 2D*** **(mandatory)**
2) ***Cosine*** **(mandatory)**
3) ***Manhattan*** **(bonus)**
4) ***Minkowski*** **(bonus)**
5) ***Chebyshev*** **(bonus)**

Since the accesses in memory made for all the distances' functions are similar, it will be only explained the accesses to the indexes.

All the distances' functions are implemented by a CUDA's kernel, that will receive as arguments the general map of observations, the current observation that's being considered, the number of features (or components) for each observation, the size of the general map of observations, and also, the matrix of distances, where the final results will be placed. This functions use the following specifications:

a) **It's calculated the ID of the current executing Thread in the overall grid of Threads to be considered as the index of the general map of observations;**
b) **If the previously mentioned index or ID it's inside of the dimensions of the matrix of distances, it will be summed to that index, some kind of offset, given for each feature (or component) of the observation that's being currently computed. This access will be used for one of the vectors to compute the distance;**
c) **For each observation accessed, will be considered every feature (or component) that's being analysed, for the previously mentioned current vector being used for computing the distance, according to the used distances' function at the moment;**
d) **The result will be placed in a matrix of distances with the same dimensions of the general map of observations, but instead of have an observation with multiple features, will have only the calculated distance for every observation processed;**

The accesses to the indexes on the general map of observations in the distances' functions, in the simplest version, are only made by the previously mentioned operations on the indexes of the overall grid of the executing threads. But additionally was implemented a optimised version, that takes advantage of the computed index to access also the index of the features of an observation and improves the performance, by using parallelism properties of the threads and parallel reduction operations. This it's made with the support of *shared memory* and uses the following modified specifications:

a) **For access the indexes of each feature of an observation, it's reused the ID of the executing Thread in the current executing block, the threadIndex, but it's only considered if that *threadIndex* it's lower than the number of features of an observation;**
b) **Instead of, making a global summation considering each feature analysed incrementally, it's only considered unit vectors (with only one observation) and its result it's placed in a shared memory;**
c) **The final global sum it's made by summing the values placed in the shared memory, doing a sum reduction parallel operation, in a logarithmic complexity;**
d) **The final result will be placed in the first index of the shared memory and will be used to fulfill all the indexes of the matrix of distances;**

2. ArgMin Function

For the *ArgMin* function to find the *BMU (Best Matching Unit)*, was implemented a CUDA's kernel, that will receive as arguments the previously computed matrix of distances, the dimensions of that matrix of distances, and two arrays/vectors to keep the minimum value of all the distances for the observation that's being currently analysed and its respectively index, where the final results will be placed, since the *ArgMin* function it's supposed to return the index of the minimum value and not that value itself. This function uses the following specifications:

For access the indexes of each feature of an observation, it's reused the ID of the executing Thread in the current executing block, the threadIndex, but it's only considered if that *threadIndex* it's lower than the number of features of an observation;

a) **It's performed a parallel reduction with the support of two shared memories, to be kept the temporary minimum value of all the distances for the observation that's being currently analysed and its respectively index:**
b) **This parallel reduction it's performed by doing parallel comparisons between the assessed values instead of having a global variable to be keeping the minimum value until the moment. These comparisons and the parallel process to find the minimum value of each block will be made considering consecutive divisions in half of the blocks' size, in a logarithmic way, considering always the blocks' size and the currently executing Thread's ID inside each block.**
c) **At the end of the process, both minimum value of all the distances for the observation that's being currently analysed and its respectively index, will be placed in the first position of the respectively shared memories;**
d) **Since, this process only computes the minimum value of all the distances for the observation that's being currently analysed and its respectively index, for each block, the both values at shared memories will be transferred to the corresponding arrays/vectors to keep that values, and posteriorly, transferred to the CPU's memory, to be find the global minimum value and the respectively index, between all the blocks;**

3. Neighborhood Function

For the *Neighborhood* function to compute the matrix of Neighborhood to a specific observation, was implemented a CUDA's kernel, that will receive as arguments the current iteration, the number of features (or components) for all the observations, the previously computed *BMU (Best Matching Unit), as the index of that minimum value, as explained before,* the size of the matrix of Neighborhood, which will have the same dimensions of the matrix of distances, the maximum distance that these matrices may have, and the matrix of Neighborhood itself, where the final results will be placed. This function uses the following specifications:

a) **It's calculated the ID of the current executing Thread in the overall grid of Threads to be considered as the index of the matrix of *Neighborhood*;**
b) **If the previously mentioned Thread's ID it's inside of the dimensions of the matrix of *Neighborhood*, it will be compute to that index, the result of the neighborhood function specified in the algorithm of the project's description;**
c) **The computing and calculations for the Neighborhood function will be made, taking in consideration, the value of *theta* based in the received arguments of the maximum distance that these matrices may have, the current iteration and the number of inputs (number of observations) of the file that's being processed, the *square root* of the *two power* of the *absolute* value of the difference between the BMU (Best Matching Unit), previously calculated. And finally, it will be calculated the exponential value of the negative value resulting of the previously calculated square root divided by two power of the theta value;**
d) **The result will be placed in a matrix of Neighborhood, in the previously specified and calculated index. If the final result of the all the previous operations is lower than 0.01, otherwise the result place in that index, will be 0.0;**

4. UpdateMap Function

For the *UpdateMap* function to update the indexes of the general map of observations, accordingly with the computing steps made by the previously three functions, was implemented a CUDA's kernel, that will receive as arguments the current iteration, the number of features (or components) of an observation, the observation that's being currently processed, the dimensions of the general map of observations, the matrix of Neighborhood and also, the general map of observations, where will be kept the updated values in the corresponding indexes. This function use the following specifications:

a) **It's calculated the ID of the current executing Thread in the overall grid of Threads to be considered as the index of the general map of observations;**
b) **If the previously mentioned Thread's ID it's inside of the dimensions of the general map of observations, it will be compute to that index, the update value for the corresponding observation, accordingly to the specified in the algorithm of the project's description;**
c) **The computing and calculations for the update of observations contained in the general map of observations will be made by the summation of all the products between the learning rate (which is 1 divided by the current iteration), the value of the previously mentioned index in the matrix of Neighborhood and the difference between all the features of the observation that's being currently processed and all the features of the observation of the corresponding to the previously calculated index;**

CUDA's Parallel Streams

In the CUDA's optimised version it's implemented also, with the support of CUDA's Parallel Streams to overlap computation with communication.

This parallelism that was pretended to obtain, in order to improve the overall performance and runtime of the global process, it's based in process more than observation at the same moment.

If it's four or more remaining observations to process, it will be launched four parallel CUDA's kernels in different CUDA's parallel streams.

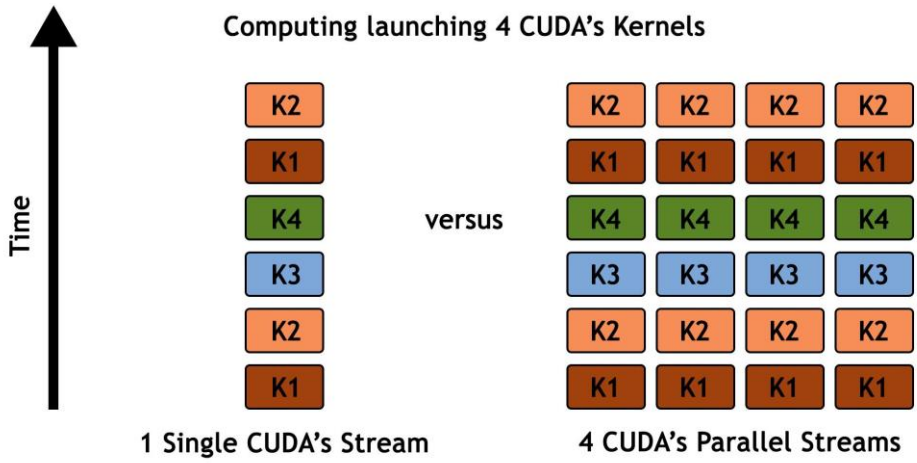
If it's only three remaining observations to process, it will be launched three parallel CUDA's kernels in the different CUDA's parallel streams.

If it's only two remaining observations to process, it will be launched two parallel CUDA's kernels in the different CUDA's parallel streams.

If, at last, it's only one remaining observation to process, it will be launched a single CUDA's kernel that will not be in any CUDA's parallel stream.

The CUDA's kernels in different CUDA's streams runs in parallel and in an independent way from each other. This it's a good mechanism to improve the overall computing performance and runtime of the execution of the algorithm for all observations that have to be processed.

The following simple example, shows the differences between the supposed behaviours of CUDA's kernels executing in different parallel streams:



Evaluations & Comparisons

It was performed some experimental tests (most specifically, a total of 4 tests), considering the input file *iris.data*. The table of the best obtained results are shown in the following table:

Performed Test no.	Dimensions of the matrices	Runtimes of executions (in milliseconds) for each version of the *SOM* algorithm		
		CPU's Sequential Version	GPU's Parallel Version	GPU's Optimised Parallel Version
#1	[20x20]	*486 ms*	*73 ms*	*66 ms*
#2	[200x200]	*49 525 ms*	*72 ms*	*69 ms*
#3	[500x500]	*305 281 ms*	*69 ms*	*71 ms*
#4	[1000x1000]	*1 207 795 ms*	*72 ms*	*68 ms*

Table 1: The table of evaluation and comparison of the experimental tests, considering the three versions of the Self-Organising Maps algorithm for the input file of *iris.data*

It was performed some experimental tests (most specifically, a total of 4 tests), considering the input file *correlated-stream.data*. The table of the best obtained results are shown in the following table:

Performed Test no.	Dimensions of the matrices	Runtimes of executions (in milliseconds) for each version of the *SOM* algorithm		
		CPU's Sequential Version	GPU's Parallel Version	GPU's Optimised Parallel Version
#1	[20x20]	*2 280 ms*	*336 ms*	*334 ms*
#2	[200x200]	*305 186 ms*	*438 ms*	*377 ms*
#3	[500x500]	*1 956 461 ms*	*442 ms*	*381 ms*
#4	[1000x1000]	*6 736 328 ms*	*679 ms*	*391 ms*

Table 2: The table of evaluation and comparison of the experimental tests, considering the three versions of the Self-Organising Maps algorithm for the input file of *correlated-stream.data*

Computer's Characteristics and Specifications

OS Name	Microsoft Windows 10 Home
Version	10.0.17763 Build 17763
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	RUBEN-LAPTOP
System Manufacturer	ASUSTeK COMPUTER INC.
System Model	FX503VD
System Type	x64-based PC
System SKU	
Processor	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2801 Mhz, 4 Core(s), 8 Logical ...
BIOS Version/Date	American Megatrends Inc. FX503VD.305, 15/03/2018
SMBIOS Version	3.0
Embedded Controller Version	1.13
BIOS Mode	UEFI
BaseBoard Manufacturer	ASUSTeK COMPUTER INC.
BaseBoard Product	FX503VD
BaseBoard Version	1.0
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\Windows
System Directory	C:\Windows\system32
Boot Device	\Device\HarddiskVolume5
Locale	United States
Hardware Abstraction Layer	Version = "10.0.17763.404"
User Name	RUBEN-LAPTOP\rubenandrebarreiro
Time Zone	GMT Daylight Time
Installed Physical Memory (RAM)	16.0 GB
Total Physical Memory	15.9 GB
Available Physical Memory	7.68 GB
Total Virtual Memory	22.1 GB
Available Virtual Memory	9.98 GB
Page File Space	6.25 GB
Page File	C:\pagefile.sys
Kernel DMA Protection	Off
Virtualization-based security	Not enabled
Device Encryption Support	Elevation Required to View
Hyper-V - VM Monitor Mode E...	Yes
Hyper-V - Second Level Addres...	Yes
Hyper-V - Virtualization Enable...	Yes
Hyper-V - Data Execution Prote...	Yes

Conclusions

The implementation of this project using parallelism of the GPU with the support of CUDA's kernels properties give huge improvements in the computing performance, specially when we are dealing with computing processes on huge and massive amounts of data. The runtime of the computing and the processing decrease in exponential scale.

Sometimes, using the CUDA's Parallel Streams to overlap computation with communication, can leverage even more the global computing performance, decreasing even more the runtimes and the processing costs, since we can make operations that are not dependable of each others, simultaneously, using a kind of processing channels, most specifically parallel computing streams.

Bibliography

- ***https://en.wikipedia.org/wiki/CUDA***
- ***https://docs.nvidia.com/cuda/index.html***
- ***https://devtalk.nvidia.com/***
- ***http://horacio9573.no-ip.org/cuda/modules.html***

Project's Repository

If you are interested in check the *CUDA* and *C++ (C Plus Plus)* versions of the several implementations of the algorithm, as also, the generated outputs from some experimental tests performed, you can check it all in the following *GitHub*'s repository:

- ***https://github.com/rubenandrebarreiro/gpu-cuda-self-organising-maps***