Implementing   
the Kohonen Self-Organizing Map algorithm on Graphics Processing Unit

## Nicolas Said University of Technology of Belfort-Montbeliard 90010 Belfort cedex - +33 (0) 3 84 58 30 00

**Abstract:** *Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus adipiscing nulla ac enim. Aliquam sodales condimentum sapien. Donec pellentesque, mi quis dapibus egestas, dolor tortor tempor eros, vitae egestas lectus nunc vel risus. Suspendisse lacinia nibh at purus. Pellentesque porttitor libero eu felis. Maecenas iaculis erat. Nam euismod rutrum tortor. Nunc non dolor. Pellentesque laoreet posuere velit. Vivamus cursus tellus vitae enim imperdiet tristique.*

## Introduction

## Kohonen Self-Organizing Maps

### Brief presentation

Kohonen Self-Organizing Maps belong to a class of neural networks based on unsupervised learning. Also called SOM (for Self Organizing Maps), they were introduced by Teuvo Kohonen in 1975. Kohonen presents them as a “new, effective software tool for the visualization of high-dimensional data. It converts complex, nonlinear statistical relationships between high-dimensional data items into simple geometric relationships on a low-dimensional display. As it thereby compresses information while preserving the most important topological and metric relationships of the primary data items on the display, it may also be thought to produce some kind of abstractions”. Like most of the type of neural networks, the SOMs are directly inspired by the behavior of living beings. In that precise case, the algorithm is inspired by the cerebral cortex. It has been observed that the cortical structures are either linear or planar, however, the sensorial data that they must process are highly dimensional. The cortex seems to apply a sort of projection inside of its neural structures, from the high-dimension data space to the linear or planar space of cortical neural structures space. One of interesting property of this internal operation is that the so-called projection seems to preserve the topology of the original space. This property is very interesting and can be used to build powerful visualization or classification tools.

The SOM tries to imitate this behavior and use the properties of such non-linear projection.  
Well, the projection process appears to actually be vector quantization, in which the codebook is built automatically through the learning process.

### Principle

The SOM is composed of two neural network layers:  
-The output layer, to which we give a topology (most of the time, a rectangular mesh). Every vector of the output layer is associated to a referring vector, of the same dimension as the input data.   
-The input layer, composed of as much neuron as we have input data to classify, also associated to referring vectors (which are in fact the actual input data for classification)

Every input neuron is connected to each neuron of the output layer.

We then use a competitive learning algorithm which will modify the value of the referring vectors of the output layer. Those vectors will then constitute the codebook of a vector quantization (if we use a VQ analogy).

It means that every output neuron referring vector will be representative of a portion of the input space, thus we can use the so-called map as a projection of the input space into less-dimensional space.

### Mathematical Formalization

Given A the grid of output neurons. Such a neuron map associate to each input vector , a neuron identified by its position in the grid , such as its referring vector is the closest to .

This association is given by this function:

Given this function, we can define the applications of the map :

*Vector quantization:*  
We approximate every point in the input space to the closest prototype in the map with :

*Classification:*  
Using the function , we can assign to each neuron in the map a label corresponding to a class, all the points of the input space which project to the same neuron in the map, belong to the same class.

To modify the output neurons referring vector and to make them converging toward the input space vectors, we use an iterative learning algorithm.

Using this mapping function, each point of the input space will be associated to a output neuron in the map. The input point which are close in the input space will be associated to the same or close neuron in output-layer space, thus we can use this as a classification tool.

Learning Algorithm:  
**Initialization** of the output layer referring vectors to random values.

**(Loop) Until** the convergence conditions are not met:

**(Loop) For each** input neuron, chosen randomly,   
and given v its referring vector:

We search the output neuron having the referring vector closest to , called :

The winning neuron and its neighbors (defined by a neighboring function) move their referring vectors toward the input vector according to :

With

where is the learning ratio and is the neighboring function.

La fonction de voisinage décrit comment les neurones dans la proximité du vainqueur *s* sont entraînés dans le mouvement de correction. On utilise en général :

où σ s'appelle *coefficient de voisinage*. Son rôle est de déterminer un rayon de voisinage autour du neurone vainqueur.

Sigma est fonction de t, et est défini par :

avec N le nombre d'itérations prévues.

**(End of loop)**

**(End of loop)**

## General Processing on graphic processor

With the coming of programmable graphics hardware and their increase in processing power, memory and parallel processing capability, the use of graphics hardware for general purpose computation became a reality and practical solution for solving certain problem using relatively cheap hardware as supercomputer.

GPGPU stands for General Processing on the GPU, and this is a technique that consists in using the GPU chip on the video card as a coprocessor that accelerates operations that are normally executed on the CPU.   
  
Since GPUs are inherently parallel, there are certain type of calculations can be executed much faster than in conventional CPUs. In fact, GPU programming is part of a wide trend towards multi-core and parallel programming. A modern CPU can have 2 or 4 cores, while a medium range GPU has usually 32 processing cores as minimum. However, the nature of the cores in a GPU is very different from the ones found on a CPU. They are optimized for parallel vector operations, and are not very efficient in algorithms that have complex flow structures or random memory access. In any case, the latest advances are making GPUs much more flexible and closer to CPUs. On the other hand, CPUs companies are currently working in hybrid designs that will combine CPU and GPU in a single chip.  
  
The purpose of this paper is not to provide a full documentation on GPGPU techniques. So I won’t describe in details the techniques and concept inherent to GPGPU.

## Algorithm modification

Before starting porting the SOM algorithm to the GPU we have to identify which part of this algorithm can or cannot be parallelized. In order to do that, we have to identify the dependencies which may exist between every fragment of data we have to deal with.

As the SOM algorithm relies on WTA strategy, we cannot parallelize anything without modifying the algorithm, as a strong time-dependency exists between epochs of different neurons.

On the other hand, we can modify the algorithm to take advantage of SIMD processing capabilities of GPU and still obtain good classification results.

For example, we can lower the BMU research frequency, and instead of searching for BMU sequentially (before every neuron map update), we can search for them after every epoch, at once.

Thus, we can parallelize the BMU research algorithm along the number of input neuron (unrolling one loop inside the BMU research algorithm, and lowering the complexity).

However, due to strong time and space dependency, we cannot parallelize entirely the output neuron layer update, and have to use an iterative procedure on the CPU to perform this operation.

## Implementation

The experimental prototype uses the C# language for ease of prototyping, and the Microsoft Research Accelerator library which provides an access to GPU processing capabilities by the use of high-level abstraction model of GPU memory and basic SIMD operations.

The Accelerator library is an experimental GPGPU library developed by Microsoft Research. It exposes the GPU memory from the .Net framework language, and allow to use SIMD GPU operations directly from C#.

Accelerator provides abstract model for GPU memory called Parallel Arrays. A parallel array is similar to a classic CPU array, but a very specific set of operations are allowed on them. Basically, there is two kind of PA operations :

Arithmetic operation, such as addition, substraction, product, division…

Basic linear algebra : inner product, outer product.

Transformations : Expand, Shift, Rotate, Pad…

Reductions over dimensions : Sum, Product, etc…

The GPU SOM Algorithm is composed of two routines :

**FindBMU**, which finds the BMU for each input neuron in parallel.As you can see in those two listings, the nested loop in the CPU version disappeared and the GPU version is only composed of simple sequence of Accelerator instructions

The loops have been “unrolled”. Using SIMD architecture and parallelism capabilities of GPU, we reduced the complexity of this part of the algorithm.

public override void FindBMU()

{

//Useful locals

int alen =

this.m\_Parent.NeuronMapShape.GetFlatLength();

int m\_PatternLength = m\_CurrentPattern.Length;

float[] Pattern = m\_CurrentPattern;

//Create distances array

float[] distances = new float[alen];

for (int i = 0; i < alen; ++i)

{

double sum = 0;

for (int j = 0; j < m\_PatternLength; ++j)

sum += Math.Pow(m\_Parent.NeuronMap[i, j] - Pattern[j], 2.0f);

distances[i] = (float)(Math.Sqrt(sum));

}

//Find the minimal distance

int min\_ind = 0;

for (int i = 0; i < alen; ++i)

{

if (distances[i] < distances[min\_ind])

min\_ind = i;

}

//Return the BMU coords and value

m\_BMUCoord = Parent.NeuronMapShape.GetSpatialPosition(min\_ind);

for (int i = 0; i < m\_PatternLength; ++i)

m\_BMUWeight[i] = m\_Parent.NeuronMap[min\_ind, i];

}

Let’s see how the GPU version of this routine works in practice

public override void FindBMU()

{

//Normalize the weight vector

FPA transpose = PA.Transpose(m\_GPUWeight, 1, 0);

FPA weightsq = PA.InnerProduct(m\_GPUWeight,

PA.Transpose(m\_GPUWeight, 1, 0));

FPA weightsum = PA.Sum(weightsq, 0);

FPA weightlength = PA.Sqrt(weightsum);

weightlength =

PA.Stretch(PA.AddDimension(weightlength, 1), 1,

m\_Parent.DataSource.PatternLength);

FPA weightnorm = PA.Divide(m\_GPUWeight, weightlength);

weightnorm = PA.Transpose(weightnorm, 1, 0);

//Normalize the input vector

FPA inputsq = PA.InnerProduct(m\_GPUInput,

PA.Transpose(m\_GPUInput,1,0));

FPA inputsum = PA.Sum(inputsq, 0);

FPA inputlength = PA.Sqrt(inputsum);

inputlength = PA.Stretch(PA.AddDimension(inputlength,

1), 1, m\_Parent.DataSource.PatternLength);

FPA inputnorm = PA.Divide(m\_GPUInput, inputlength);

FPA pacc = PA.InnerProduct(inputnorm, weightnorm);

//Replication bug here...

FPA bmxval = PA.MaxVal(pacc, 1);

//MSR Vivian Swelson workaround

DFPA bmxvalEvaluated = PA.Evaluate(bmxval);

bmxval = PA.AddDimension(bmxvalEvaluated, 1);

bmxval = PA.Stretch(bmxval, 1,

m\_Parent.NeuronMap.GetLength(0));

//Winner matrix (0 = winner)

FPA pwinner = PA.Subtract(pacc, bmxval);

//Convert to 1 = winner, 0 otherwise

FPA zero = new FPA(0.0f, pwinner.Shape);

FPA one = new FPA(1.0f, pwinner.Shape);

BoolParallelArray bmask = PA.CompareEqual(pwinner, zero);

m\_PWinner = PA.Cond(bmask, one, zero);

}

And DoEpoch, which updates the whole neuron map for each input neuron.

## Complexity

First, we will consider that each GPU operation runs in constant time.

One iteration of the learning algorithm is composed of the two following steps :

Finding BMU:

Constant time (refer to experimental data)

This operation runs in O( k ).

Update the output layer for each input value:

## Results

The following results have been obtained, using relatively archaic graphic hardware on an average laptop.

Figure 1 shows the execution time for the FindBMU routine against the number of input neurons. As the theoretical analysis of the modified algorithm stated, the running time of the GPU version is constant. While the execution time of the CPU version seems to be linear. As the time complexity of the FindBMU algorithm is no x plen, as we keep plen constant in the

experiment, thus the algorithm shows execution time linearly proportional to number of input neurons.

And as the loop is unrolled in the GPU version, thus it’s exhibit a constant execution time.

Figure 2 shows the execution time against the dimensionality of input pattern in a time-serie forecasting prototype using SOM algorithm. While the dimension of input is increased, the number of numerical values to be treated byt the algorithm remains constant. We just change the chunk size.

We can clearly see that the GPU version runs way faster than the CPU version, as the dimensionality increases.