Fuzzy ART Neural Network Parallel Computing on the GPU

Mario Martínez-Zarzuela, Francisco Javier Díaz Pernas,
José Fernando Díez Higuera, and Míriam Antón Rodríguez

Higher School of Telecommunications Engineering
University of Valladolid (Spain)
mario.martinez@tel.uva.es
http://gti.tel.uva.es

Abstract. Graphics Processing Units (GPUs) have evolved into powerful programmable processors, faster than Central Processing Units (CPUs) regarding the execution of parallel algorithms. In this paper, an implementation of a Fuzzy ART Neural Network on the GPU is presented. Experimental results show training process is slower on the GPU than on a dual-core Pentium 4 at 3.2 GHz. Once the Neural Network has been trained, the proposed design manages to accelerate Fuzzy ART testing process up to 33 times on a GeForce 7800GT graphics card.

1 Introduction

Graphics Processing Units (GPUs) have been used for many years as CPU co-processors, helping them in the task of rendering complex images onto the computer screen. Inside this special purpose processors, data downloaded from the CPU is transformed along the *graphics pipeline*. In some stages within this pipeline, several computation units can work in a parallel fashion speeding up computations.

Some years ago the whole functionality of the pipeline within the GPU was hard-wired, because every graphics application needed more or less the same computation stages. As the demand on ever increasing special effects on games and other virtual reality applications grew, GPUs became more and more powerful, offering an incredible computational horsepower at low cost.

Gradually, high level programming of some processing unit in the GPU became posible, making this powerful processor rather flexible. Today, a community of researchers from many scientific research field, works together helping each other to accelerate their own algorithms using the GPU as a high performance processor [1]. Mapping algorithms to fit the GPU is tricky, and programmers must study the architecture and working inside of the GPUs, before being able to take advantage of it as a computational resource [2].

Fuzzy ART is an unsupervised neural network capable of incremental learning, widely used in an universe of applications as medical sciences, economics and finance, engineering and computer science, or pattern recognition and classification

F. Sandoval et al. (Eds.): IWANN 2007, LNCS 4507, pp. 463–470, 2007.

[3]. Fuzzy ART is an extension of the original ART 1 system designed by Grossberg et al [4], capable to learn and categorize only binary patterns. Fuzzy ART incorporates computations from fuzzy set theory into the ART network, replacing the intersection operator ($\cap$) by the MIN operator ($\wedge$), and thus making it possible to learn and recognize both analog and binary input patterns.

Although graphics processing units are being considered in many fields of computation, programming of different kinds of ANNs (Artificial Neural Networks) on the GPU has not been deeply explored. Thomas Rolfes [5] showed how a multi-layer perceptron (MLP) network could be implemented on the GPU using matrix-matrix products to compute the activation levels for the neurons. Kyoung-Su et al. [6] implemented a GPU-based MLP for text detection and Steinkraus et al. [7] achieved a 3x speedup in their GPU implementation of a generic 2-layer fully connected neural network. Other groups of researchers, as Zhongwen [8] and Campbell [9], have used the GPU for implementing self-organizing maps (SOM) with great results. Finally, Bernhard et al. achieved a speed increase of between 5 and 20 times faster simulating large networks of spiking neurons on the GPU [10].

In this paper, we propose a Fuzzy ART Neural Network implementation on the GPU, both for training and testing processes. The rest of this paper is organized as follows: for completeness, Section 2 summarizes Fuzzy ART architecture and training algorithm. Section 3 briefly explains how GPUs work and how to speed up computations taking advantage of parallel execution. Section 4 explains our implementation of a Fuzzy ART Network on the GPU. Section 5 summarizes the experimental results that were obtained for measuring our implementation performance. Finally, Section 6 draws the main conclusions and outlines future research tasks.

2 Fuzzy ART Networks

ART systems are comprised of three layers or fields of nodes. The first field F_0 represents the input pattern; the upper field F_2 represents the active code or category of the input pattern being selected; the middle layer F_1 receives both bottom-up inputs from F_0 and top-down inputs from F_2. The F_0 activity vector is denoted by $\boldsymbol{I} = (I_1, \cdots, I_M)$ where each component I_i is within the $[0, 1]$ interval. The F_1 activity vector is denoted by $\boldsymbol{x} = (x_1, \cdots, x_M)$ and the F_2 activity vector (output) is denoted by $\boldsymbol{y} = (y_1, \cdots, y_N)$. Each component in the $\boldsymbol{y}$ vector represents a specific category from those emerged during the self-organizing learning of the network. Thus the system is able to categorize N different input patterns, each one defined by M components. Each node of the F_2 field has an associated weight vector or long-term memory (LTM) traces $\boldsymbol{w_j} = (w_{j1}, \cdots, w_{jM})$ wich subsumes information both from bottom-up and top-down weight vectors. Initially, all weights are set to one, so each category is said to be *uncommitted*. When a category is first selected it becomes *committed* and

the corresponing node in F_2 re-adapts its associated weights $\boldsymbol{w_j}$. For each input $\boldsymbol{I}$ and F_2 node j, the *choice function*, T_j, is defined by:

$$T_j(\boldsymbol{I}) = \frac{|\boldsymbol{I} \wedge \boldsymbol{w_j}|}{\alpha + |\boldsymbol{w_j}|} \tag{1}$$

where the fuzzy MIN operator $\wedge$ is defined by $(\boldsymbol{p} \wedge \boldsymbol{q}) \equiv \min(p_i, q_i)$ and the norm $|\cdot|$ is defined by $|\boldsymbol{p}| \equiv \sum_{i=1}^{M} |p_i|$. The category choice is indexed by J, where $T_J = \max(T_j : j = 1 \cdots N)$. Then, $\boldsymbol{w_j}$ is said to be a *fuzzy subset* of $\boldsymbol{I}$ and it is fed down from F_2 in order to measure its resemblance to the input pattern $\boldsymbol{I}$. The system enters in *resonance* if the *match function* meet the *vigilance criterion* ρ:

$$\frac{|\boldsymbol{I} \wedge \boldsymbol{w_j}|}{|\boldsymbol{I}|} \geq \rho \tag{2}$$

When this occurs and learning is enabled, vector $\boldsymbol{w_j}$ is updated (3). Otherwise, node J is inhibited making $T_j = 0$ and the node in F_2 with the biggest activity is selected.

$$\boldsymbol{w_J^{new}} = \beta(\boldsymbol{I} \wedge \boldsymbol{w_J^{old}}) + (1 - \beta)\boldsymbol{w_J^{old}} \tag{3}$$

3 Programmable Graphics Hardware

Commodity graphics cards provide a tremendous computational horsepower. NVIDIA's GeForce 7800 GTX GPU is able to sustain 165 GFLOPS against the 25.6 GFLOPS theoretical peak for the SSE units of a dual-core 3.7 GHz Intel Pentium Extreme [1]. The great difference in performance between these two platforms is due to fundamental architectural designs. CPUs are designed for executing general purpose programs comprised of sequential instructions operating on single data. CPU designers try to optimize complex control requirements with minimum latency, so many transistors in the chip are devoted to branch prediction, out of order execution and caching. On the other hand, GPUs are designed for exploiting data parallelism following a *stream programming model* [2]. In this programming model a series of computations (*kernel*) are made over an ordered set of data (*stream*). The main restriction of the model is also one of the reasons the GPU has such a great performance: operations on each stream element are independent, allowing the execution of the kernel on different hardware units simultaneously, and avoiding waits that could occur because of inter-unit data sharing.

GPUs have two types of programmable processors, namely *vertex* and *fragment* processors. The latter is preferred for GPGPU applications, as much more fragment units are present in the *graphics pipeline* (24 fragment processors and 8 vertex processors on a GeForce 7800 GTX). Both kind of processors are devised to operate on four component vectors, as the basic primitives of 3D computer graphics are 3D vertices in projected space (x, y, z, w) and four component colors $(red, green, blue, alpha)$. Floating point operations can be done in fixed point

(16 bits) or single float (32bits) precision. Vertex processors operate in *multiple-instruction-multiple-data* (MIMD) fashion, and fragment processors are only able to operate in a *single-instruction-multiple-data* (SIMD) way [2].

Vertex and fragment units are programmed using *shaders* that can be written using different high level languages as Cg, GLSL or HLSL. Vertex shaders are used to process streams of vertices (positions, colors, normal vectors, etc) that define the elements composing polygonal geometric models. Along the *graphics pipeline* vertices are grouped into primitives, that are sampled into a *stream of fragments* before going down to the fragment processor or *pixel shader*. A *fragment* can be thought as a proto-pixel, that would be drawn on the screen (actually the *frame buffer)* if it is not discarded away after a series of tests. A further explanation on the different parts of the pipeline can be found in [2].

Both *vertex* and *fragment shaders* are able to fetch data from textures, a 2D or 3D array of data that can be randomly accessed. For GPGPU computations LUMINANCE and RGBA textures are normally used, making it possible to store 1 or 4 floating point data per texture element (*texel*). When an image is generated, it can be written to the *frame-buffer* memory or rendered to a texture, allowing a direct feedback of the output to the pipeline's entry. In modern GPUs *Multiple Render Targets* can be used for rendering up to 4 RGBA textures at a time [2].

4 GPU Implementation of a Fuzzy ART Network

In this paper we propose two different implementations of a Fuzzy ART Neural Network on the GPU, one being for training and the other for testing processes. Each one of these implementations involve different arranges for data, with the aim of maximizing the task parallelism and thus the performance of the GPU in each process.

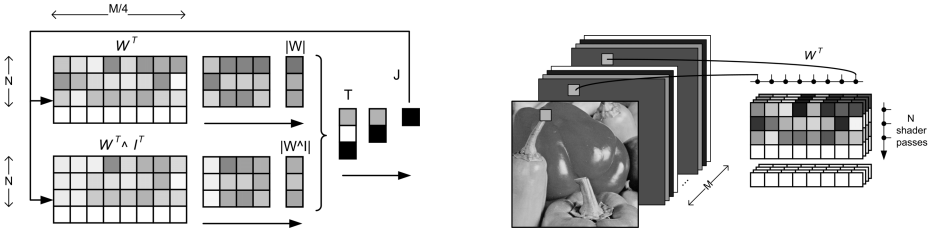
4.1 Training Mode

Learning is not a parallel but a sequential process. For this reason, trying to parallelize the way a Fuzzy ART network learns seems not a good idea. Different input patterns can not be learned at the same time, because they all would generate different categories. Optimizing the training process for parallel execution must be done at the time of searching for the category that most resemblances to the input pattern. Fuzzy ART implementations on the CPU sequentially compute the activity for every F_2 field neuron. Then, a sort operation is made in order to know which neuron is most fired by the input pattern (1). If the category stored in this neuron resembles enough to the input pattern, it's associated weights are updated with the new information; if not, the next most fired neuron must be analyzed. In our GPU implementation, we can compute the activity of every output neuron in a parallel fashion. Moreover, we can obtain the match function (2) for every neuron simultaneously.

Weights of every committed neuron $\boldsymbol{w_j}$ are stored as rows in a texture W^T (see Fig. 1(a)). Input pattern $\boldsymbol{I}$ is rendered to every row of a texture I^T with

same dimensions as W^T, so that during category choice, it can be compared to every F_2 neuron at once. Generalized size reduction operations on the GPU using a *ping-pong* technique [2], are used to obtain $|\boldsymbol{I} \wedge \boldsymbol{w_j}|$ and *Multiple Render Targets* support allows calculating the module of newer committed categories $|\boldsymbol{w_j}|$ at the same time. The use of RGBA textures allows to run MIN and SUM operations on 4 vector components in one clock cycle on every fragment shader unit, making the process faster. If dimensions of input patterns are not multiple of four, unused channels of the RGBA textures must be padded with zeros. Reduced textures are then used to store the activity of each neuron, satisfying the match criteria, on the R channel of a texture T; G channel is used to store the category index; *alpha* channel takes the value of 1 in case the match criteria is satisfied and 0 in other case; finally, channel B can be used for printing the matching rate, which we found very useful for debugging purposes.

The J_{th} neuron is found using a row reduction operation over texture T, in which those fragments not satisfying the match criteria are discarded. If the system enters in resonance, the weights of the selected category are updated by rendering into the corresponding sub-region of the texture W^T using *scissoring* [2]. If not, the new pattern is learned by rendering to an unused row of weights in W^T using equation (3).



(a) Training Mode (b) Testing Mode

Fig. 1. Arrange for data on the GPU for Training and Testing modes

4.2 Testing Mode

Fuzzy ART testing algorithm is easier and much more profitable to implement on the GPU. In this testing process, several input patterns can be categorized in a parallel fashion when learning mode is switched off. We propose to take advantage of every fragment processor available on the GPU for categorizing each pattern in several *shader* passes. Figure 1(b) schemes the organization for data in the GPU. Bidimensional textures are more amenable to be used on the GPU than one dimensional textures, so input patterns should be distributed to fit this shape. This kind of organization maps specially well for pattern recognition in image processing applications, where it is useful to categorize each pixel of an image from a computed vector of features $\boldsymbol{I}$.

In the proposed system, for every (x, y) coordinate on the input data, a pattern is stored along the z direction. A single RGBA texture I^T can store 4

component input vectors, and many RGBA textures can be used next to other for storing greater patterns $I_k^T : k = 1 \cdots M/4$. Again, if the dimension of $\boldsymbol{I}$ is not a multiple of 4, unused channels must be padded with zeros. The output of the categorization *shader* is another RGBA texture with the same dimensions as the input textures.

A single RGBA texture W^T is used for storing F_2 field neuron weights on the GPU. Each row stores a long-term-memory trace $\boldsymbol{w_j}$, just as in the arrange proposed for the testing algorithm. The input vector components stored on the k_{th} input texture I_k^T are compared to the corresponding column of weights $M/4$ on W^T. Category choice occurs through the execution of a shader for N times, being N the number of categories in the neural network. In each pass, the activation of the j_{th} output neuron and the match function are computed for every input pattern. This values are rendered into the RGBA output texture, which is used as input for the next iteration, again using a *ping-pong* technique. The other two channels in the output texture are used for indexing the selected category of the pattern and counting the number of categories the input pattern has been tested against.

If the activation in pass j is bigger than the computed activation in pass $j-1$ and the match criteria is satisfied, then the index category is updated on the output texture. Rendering both the index of the selected category and the match function to the output texture allows the expert to visually analyze the result. In our implementation, different gray levels on channel R belong to different categories and *alpha* channel represents the level of resemblance to the input pattern of the selected category.

5 Experimental Results

In order to measure the performance of our implementation, several tests were done both on the CPU and the GPU. The former with a Fuzzy ART C++ implementation, and the latter with our C++/OpenGL/Cg implementation. Timings were taken on a dual-core 3.2 GHz Pentium 4 with 1GB RAM and a GeForce 7800GT 256 MB GPU.

Performance of the Fuzzy ART Network relays on several factors: i.e. length of the input pattern $\boldsymbol{I}$, number of input patterns P presented to the network and number of created categories N. During the learning process, the number of committed categories varies depending both on the grade of similarity between patterns and the *vigilance criterion* ρ. For the training tests, a synthetic *benchmark* comprised of several sets of patterns was generated. In each set, the number of components for input vectors and the number of expected categories varies (see Table 1). In order to guarantee the number of categories that could be committed was not too much influenced by the length and number of input patterns, a *Multivariate Normal Distribution* was used for pattern generation. Being N the number of categories in a set of P *complement coded* patterns

$\boldsymbol{I_p} = (\boldsymbol{a_p}, \boldsymbol{a_p}^c) \equiv (a_{p_1} \cdots a_{p_M}, a^c_{p_1} \cdots a^c_{p_M})$, the n patterns belonging to category N_i within the set were generated using a normal distribution for each vector $\boldsymbol{a} \sim N_N(\mu, \sum)$, and then obtaining its complement coding $\boldsymbol{a}^c$. In vector μ, the mean for every component is selected to be in the $(0, 1)$ range and covariances were set to null in covariance matrix $\sum$. Finally, parameters in the network were choosen to be constant and with the following values: $\rho = 0.9$, $\alpha = 0.05$, $\beta = 1$. Before each set of patterns P is presented to the network, patterns are randomly mixed up. Randomly generated sets of patterns were configured to include 10 categories in 10×10^3 patterns, 50 categories in 50×10^3 patterns and 100 categories in 100×10^3 patterns.

Table 1 reveals that the training process takes more time to execute on the GPU than on the CPU. As stated before, learning is a sequential process, thus we can not re-write Fuzzy ART learning algorithm for an optimal parallel execution. Although our design achieves to keep busy the best part of fragment shaders, the *arithmetic intensity* for this algorithm is too low. Another bottleneck can be found in that our implementation downloads patterns from the CPU to the GPU one by one, forcing the GPU to wait for new data. However, our design demonstrated to be faster than a Matlab implementation of Fuzzy ART, where even a collection of 50×10^3 patterns with dimension 4 takes 380 s to be trained, 15 times slower than our implementation.

For measuring the time taken by the testing process, a different collection of benchmarks was generated. The network was tested using previously learned weights. In this case, the GPU demonstrated to be dozens of times more efficient than the CPU (see Table 1). Our implementation deeply exploits the streaming programming model, so that several input patterns can be categorized in parallel. The GPU used in the tests has 20 fragment shaders and a greater reduction of time is expected when using a more modern GPU with many more fragment processors.

Table 1. Times for training and testing on the CPU and the GPU

			TRAINING		**TESTING**		
$dim(\boldsymbol{I})$	**PATTERNS**	**CATS**	**CPU (s)**	**GPU (s)**	**CPU (s)**	**GPU (s)**	**SPEEDUP**
4	10×10^3	15	0,0582	4,2128	0,0535	0,0014	**38,5**
	50×10^3	59	0,4606	25,2468	0,4704	0,0145	**32,4**
	100×10^3	119	1,4212	53,8550	1,4954	0,0563	**26,6**
8	10×10^3	8	0,0595	4,7471	0,0545	0,0012	**46,2**
	50×10^3	50	0,5706	30,8801	0,5919	0,0157	**37,8**
	100×10^3	100	1,8734	65,3028	1,9809	0,0605	**32,7**
16	10×10^3	10	0,0743	6,0570	0,0702	0,0018	**38,9**
	50×10^3	55	0,9131	35,1509	0,9075	0,0300	**30,3**
	100×10^3	111	3,3745	70,2651	3,3425	0,1181	**28,3**
32	10×10^3	10	0,0961	6,2251	0,0932	0,0029	**32,7**
	50×10^3	50	1,4913	35,3596	1,4449	0,0523	**27,6**
	100×10^3	100	5,3758	74,8078	5,3725	0,2135	**25,2**
MEAN							**33,1**

6 Conclusions and Future Work

An implementation of a Fuzzy ART Neural Network on a GPU was introduced in this paper. Our design successfully faces the problem of integrating both training and testing processes on a commodity graphics card. GPUs are quickly evolving and every few months a new generation of improved processors is made publicly available. Forward compatibility of our design is guaranteed and we can expect an incredible performance with newer cards, as the recently appeared NVIDIA GeForce 8800, which incorporates 128 unified shaders. Giving the good results achieved, potentially our design has application on a wide range of fields, such as pattern recognition and decision making.

Experimental results show how the proposed design exploits the inherent parallelism of graphics cards. Fuzzy ART testing process is performed on the GPU up to 33 times faster than on a dual-core CPU, due to being adapted for its execution following the stream programming model. Training process, though, is still slower on the GPU. Our future work includes using Pixel Buffer Objects (PBOs), an OpenGL extension, to achieve faster transfer rates from CPU to and from GPU memory.

References

1. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, vol. 26 (2007)
2. Pharr, M. (ed.): GPU Gems 2 (Programming Techniques for High-Performance Graphics and General-Purpose Computation). Addison-Wesley, London (2005)
3. Rao, V.B., Rao, H.: C++, neural networks and fuzzy logic, 2nd edn. MIS:Press, New York, NY, USA (1995)
4. Carpenter, G.A., Grossberg, S., Rosen, D.B.: Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system. Neural Networks 4(6), 759–771 (1991)
5. Rolfes, T.: Artificial neural networks on programmable graphics hardware. In: Game Programming Gems 4 (Game Programming Gems Series). Charles River Media, Inc., Rockland, MA, USA (2004)
6. Oh, K., Jung, K.: Gpu implementation of neural networks. Pattern Recognition 37(6), 1311–1314 (2004)
7. Steinkrau, D., Simard, P.Y., Buck, I.: Using gpus for machine learning algorithms. In: ICDAR '05: Proceedings of the Eighth International Conference on Document Analysis and Recognition, Washington, DC, USA, pp. 1115–1119. IEEE Computer Society, Washington (2005)
8. Luo, Z., Liu, H., Wu, X.: Artificial neural network computation on graphic process unit. In: IJCNN '05: Proceedings of the 2005 IEEE International Joint Conference on Neural Networks, Montreal,Canada, pp. 622–626 (August 2005)
9. Campbell, A., Berglund, E., Streit, A.: Graphics hardware implementation of the parameter-less self-organising map. In: IDEAL. pp. 343–350 (2005)
10. Bernhard, F., Keriven, R.: Spiking neurons on gpus. In: Alexandrov, V.N., van Albada, P.M.S.G.D., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3994, pp. 236–243. Springer, Heidelberg (2006)