SOMGPU: An Unsupervised Pattern Classifier on Graphical Processing Unit

Raghavendra D Prabhu

Abstract— Graphical Processing Units (GPUs) have been, lately used for general purpose tasks owing to their implicit parallel nature. One such task is that of pattern classification. Highly parallel tasks like these suffer from performance loss owing to the sequential nature of Central Processing Unit (CPU). To match the image processing power of human brain even slightly, this problem beckons the utilization of enormous computational power and parallel environs of GPUs. Unless there is a task which can be parallelized to the required extent the gain obtained is lost owing to the overhead involved. Thus, it is equally important to understand some limitations of GPU before venturing in this direction and deal with it appropriately to obtain satisfactory results. Artificial Neural Networks (ANN) are found to be appropriate while dealing with pattern recognition problems. Kohonen's Self Organizing Map (SOM) has been used for classification out of other approaches for its implicit parallel nature, albeit with minor modifications to make it suit the parallel environment. nVIDIA GeForce 6150 Go with Microsoft Research Accelerator as the high level library has been chosen as the platform to provide this environment.

I. Introduction

The field of pattern classification has always been intriguing. There are several reasons for this, out of which prominent ones have been its computationally intensive nature, vastness of the domain of patterns, degree of supervision required. To solve the latter problem, Self Organizing Maps (SOMs) [1], [2] based on unsupervised learning strategy have been used.

Self Organizing Maps have had a profound presence in the field of machine learning. Several Artificial Neural Network (ANN) literatures testify this. Unsupervised learning differs from its supervised counterpart in that the latter modifies its strategies (feedback) based on difference between actual output and required output (which is known during training phase).On the other hand, unsupervised learning exploits the implicit structure of input domain to assign them to their respective classes. A direct consequence of this is that demarcation is sharper in case of supervised. But, it can be seen that using supervised learning for patterns which are not known beforehand and which have complex mapping is clearly not feasible. Kohonen's algorithm based on competitive unsupervised learning plays significant role here. A 2-dimensional quadratic layer of weights have been used which are randomized on first run but future runs use the weights which are stored to stable storage in the first run. Input vectors are arrays of floating-point values which best approximate the pattern obtained after sampling the original monochromatic image. These one-dimensional input vectors are fully/partially connected to output layer of neurons. Winner Takes All (WTA) [2] strategy has been employed where output neuron with maximum activation is assigned output of 1.0 and all others is assigned 0.0. The criterion for winner selection can also be based on the factor of minimal Euclidean distance. Maximum activation method is used when the weights are normalized beforehand. Designated numbers of neurons in the vicinity of winner neuron have their weights updated along with the winner neuron. A neuron winning persistently can also be suppressed temporarily to absorb others into learning process. This neighborhood is decided by a rough approximation of the Mexican Hat function which is normalized second derivative of a Gaussian function. Learning is controlled by the parameter alpha which is initialized to a value in {0, 1}.This training process is repeated for several patterns for a certain number of iterations until there is convergence. The neighborhood size is decreased steadily over time along with alpha in order to attain stability. The output of this phase is a set of weights which map the input domain to a great extent. SOMs are also used in problems of NP-Complete nature like Travelling Salesman Problem (TSP) [3], [4] where general algorithmic approach turns to be expensive.

Clearly the process is time devouring even for a modern computer considering an input vector of length 1000 and weights of size 1000 × 2000 and several hundred patterns and several iterations. The human brain with its vast intricate network of neurons has excelled in this field to a degree beyond comparison. This is where the role played by Graphical Processing Unit (GPU) is prominent, in effectively reducing gargantuan nature of the problem without crumbling under the load. The concept of using GPUs for intensive problems is not new. They have been in use for several years in rendering high quality images in real time (up to a billion pixels per second), virtual reality simulations and real-to-life games. The vast potential had remained largely untapped. Efforts are being made since 2003 to map real life problems to fit the structure/topology and nature of GPUs. This has been termed as GPGPU (General Purpose Computation on Graphical Processing Units) [5] and clearly is a beacon of hope when Moore's law is saturating for Central Processing Units (CPUs). Modern GPUs comprise of nearly 128 cores which can perform highly intensive parallel tasks independently compared to its

Raghavendra D Prabhu is a student of B.Tech, Computer Engineering at National Institute of Technology, Karnataka (NITK), India. (Phone: +91 9900405904; email: raghu.prabhu13@gmail.com).

978-1-4244-1823-7/08/$25.00©2008 IEEE

CPU counterpart where 'true' parallelism is still uncertain. Tasks need to be of Single Program Multiple Data (SPMD) type to be efficiently implementable on a GPU. Evidently, there should be a scope for compartmentalizing the tasks with little or no dependencies among them i.e. a matrix representation. The problem at hand closely resembles them due to the quadratic arrangement of neurons considered here. SPMD model has been implemented using Microsoft Research Accelerator library.

Rest of this document is organized as follows. Section II deals with related work in this field. In Section III the design of pattern recognition problem using SOM is considered. Modifications to the above algorithm to fit the GPU and its implementation are discussed in detail in Section IV. Finally, Section V deals with experiments conducted and the results obtained, observations made and Section VI concludes the document.

II. Related work

Several approaches have been followed to improve the performance of SOM and hence of pattern classification. Luo Zhongwen et al. also implement SOMs on GPU [6] but the fundamental difference lies is the way in which weights are updated. It explicitly finds the location of winner using a multi-pass method based on Kohonen's algorithm and then updating is done based on the position, which is not the case here, as will be detailed in following sections. The multi-pass method can be expensive for larger maps. Its reliance on low-level textures also constrains it to use vector of length 4. But, the fact that it uses low-level textures gives it the advantage of fine-tuning the execution pipeline in solving problems of space and time complexity. The work presented in this paper obviates the need for finding the explicit position of winner neuron and hence avoids the overhead incurred.

Schabauer et al [4] propose cluster architecture to solve the classical travelling salesman problem. But cluster architectures have their own problems/overhead in terms of communication costs, task management, task distribution and fault-recovery which can offset the gain derived. However, such architectures have assumed importance when factors like locality of reference are weighted more. Neagoe et al. proposes Concurrent Self Organizing Maps (CSOM) [7] for biometrics where a collection of small SOM's are trained individually based on WTA strategy to provide results for one class only. This has led to increased recognition rate and reduced training time.

Vectorisation and partitioning of parameter-less SOM on GPU has been proposed by Campbell et al. in [8]. It is more suitable for interactive data exploration. Exploitation of GPU for matrix multiplication operations can be seen in [9] in the implementation of faster neural networks for text detection. It also proposes converting several inner-product operations into a single matrix operation.

III. Design of problem

In this section a generic approach to the pattern recognition problem is considered. Only monochromatic patterns drawn with mouse/stylus have been considered. A Vector Space model is adopted where the primary task is to construct a vector representing the image at-hand in most appropriate way with minimal loss of information and minimum entropy. It can also be seen that any such representation is sparse due to the nature of pattern being considered. Therefore, reduction and sampling is necessary since the length of input vector of SOM depends on this and this length is constrained by the memory of the Video RAM. Reduction also reduces unnecessary load on GPU.

Following method is adopted to reduce the image vector:

1) Image is scanned across its length and breadth and pixel value is checked and elements of a 2-D matrix are assigned 1.0 or 0.0 accordingly.

2) Now some sampling is done on this matrix to reduce it to length of input vector. Bounding box algorithm is used. The outputs of the algorithm are the co-ordinates of the smallest rectangle box which bounds the figure completely.

3) In the algorithm, the matrix is scanned from left to right till first 1(minimum) is found and similarly last 1(maximum) is found and hence are assigned to minX and maxX. Similarly minY and maxY are calculated.

4) Now a new matrix is created with values between those bounds of the old matrix.

5) At this stage sampling is performed. A new matrix is created with each cell equal to the average of sum of values in 8 neighboring cells along with this cell.

$$\text{output}\ (m,n) = \left(\sum_{i=1}^{9} \sum_{j=1}^{9} \text{input}\ (i,\ j) \right) \Big/ 9 \qquad (1)$$

This is repeated twice.

It is interesting to note that the same can be implemented on a GPU. It is essentially an image convolution with filter of value '1'.

$$B(i,\ j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i+k-1,\ j+l-1) K(k,l) \qquad (2)$$

where K is the convolution kernel/filter and I(i,j) is the i,j^{th} pixel of the image to be processed and B is the processed image.

Since the image can be represented as a matrix of pixels, convolution on GPU is done as follows.

$$M = \sum_{i=-r}^{r} \delta \left(\sum_{j=-r}^{r} \delta(\text{imatrix}\ ,0,\ j),i,0 \right) \Big/ (r+2)^2 \qquad (3)$$

where δ(imatrix,i,j) shifts a matrix 'i' units to left and 'j' units to bottom with '0' in the voids created, r is the number of neighborhood pixels(on each side) taken into account.In (1) it is taken as 1 (r = 1).M is a matrix obtained by finding

the average. The required matrix is obtained by scanning M with a gap of 'r' cells ignoring the borders.

5) This new matrix is scanned in row-major order to create a one-dimensional vector. If this vector is less than the input vector length, then this vector is padded with zeroes since SOM input vectors should be of same length.

6) Above procedure is repeated for all the images and all the images are hence vectored.

At this stage a new Kohonen network is initialized with following values.

TABLE I
PARAMETERS WITH VALUES

Parameter	Value
Size of input layer	1500
Size of output layer	30
alpha	0.4
Initial Neighborhood size	6
Total Iterations	200
Number of patterns	= Number of images

Initialization: Weights are initialized to pseudo-random values and are normalized in the first run. However, in consequent runs, we can use the weights obtained in the first run (and saved to secondary memory) if the problem domain is unchanged. An immediate consequence of this is that rich information obtained in first run can be reused and further optimized consequently leading to a faster convergence [1].

Without any GPU optimization following procedure is repeated for certain number of iterations during training phase:

1) For each pattern in the set

a) The winner neuron among the available neurons is found based on the criteria of activation calculated using the following formula:

$$\text{winner} = \max_j \left(\sum_i w_{ij} x_i\right) \tag{4}$$

b) Certain numbers of neurons in the neighborhood of the winner neuron get their weights updated along with the winner neuron according to the following formula.

$$w_{ij} = w_{ij} + h_{c(x),i}(x_i - w_{ij})$$

where h is a neighborhood function defined as

$$h_{c(x),i} = \alpha(t)\exp\left(-\| r_i - r_c \|^2 \Big/ 2\sigma^2(t)\right)$$

However for simplicity it can be safely assumed to be as

$$w_{ij} = w_{ij} + \alpha(t)(x_i - w_{ij}) \tag{5}$$

where α (t) is function of alpha decreasing monotonically over the time with neighborhood size(σ(t)), r_i and r_c are positions on the grid.

2) The neighborhood size is decreased linearly (again for simplicity) along with alpha by certain amounts based on cycle rate (which is the number of cycles after which reduction takes place every time).

Note: Only training phase is considered since it is the one which is time consuming. Once the weights are trained, testing is straight-forward and inexpensive and is implemented on CPU.

IV. GPU MAPPING

The procedure detailed in previous section needs to be tweaked to ensure its parallelization on a GPU to appreciable extent. This is because the algorithm is not explicitly data parallel. But before that, we need to consider the fragment types which can be parallelized.

Broadly, there are two kinds of control dependency.

1) Spatial Dependency: Fragments are sometimes executed sequentially due to lack of additional hardware available. An instance of this can be applying an operator to the independent elements of a set. This is dealt on sequential systems with iteration. This sort of dependency is dealt here with a GPU. Parallelization of this may or may not call for modification of the algorithm.

2) Temporal Dependency: This kind of dependency arises when result of a calculation on a set of element is required in the next iteration on the same set of elements. As is obvious from the nature of the problem, parallelizing this dependency is not possible unless a complete redesign of the algorithm is done to eliminate this dependency.

GPGPU primitives do not permit the index of an element in a parallel array to be revealed. Therefore, the important aspect to be noticed is the role played by the winner neuron. It can be clearly seen that its role in training phase is just to indicate the neurons whose weights need to be updated and hence we don't need the absolute position of the winner neuron. The algorithm to be proposed obtains the position implicitly in order for the updating to occur. A mask-based approach is used which marks the neurons to be updated.

Revised version of the algorithm is as follows.

1) The vectors representing the image are obtained as detailed in the previous section.

2) FPA (Floating Point Array) representation is created for the input(containing the patterns(collection of vectors),will be referred to as pinput) and 2-D weights(which are randomized in first run, will be referred to as pweight). The size of the input matrix will be number_of_patterns × length of an image vector and of weight vector will be number_of_inputs × number_of_outputs.

(Note: All operations detailed here forth are GPU operations and act on parallel vectors/arrays (prefixed with 'p') as a whole and not on individual cells and hence knowing index of an element is not possible. Complex operations like multiplication are carried out in a single instruction.)

Let numpat = number of patterns.

ni = number of neurons in input layer.
no = number of neurons in output layer.
neisize = neighborhood size.
numimage = number of images.

3) pacc, a matrix product of pinput and pweight is obtained.

$$pacc = product\ (\ pinput\ ,\ pweight\ ) \quad (6)$$

4) Now, maximum element is found for each row in above product and put into column vector pmxval. Clearly this vector contains activations of winner neuron for each input pattern and is of length equal to numpat.

5) Clearly, we cannot obtain the index of the winner neuron for reasons detailed above. So a workaround is needed.

A new binary matrix is created which acts as a mask to indicate as to which neurons are to be updated. The mask will have '1' corresponding to neurons to be updated and zero otherwise.

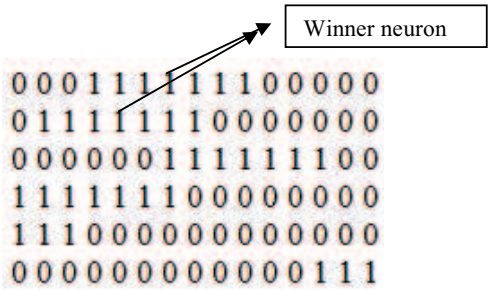

Fig. 1. showing the mask required for all 6 patterns with neighborhood size 3 in either direction.

It is done as follows.

a) pmxval is replicated along the x –direction to make it of size numpat × no.

b) A new matrix called pwinner is created by subtracting pmxval from pacc.

$$pwinner = subtract\ (\ pacc\ ,\ pmxval\ ) \quad (7)$$

c) pwinner contains zero for winner neuron and non-zero otherwise. To make neurons in its neighborhood defined by neisize equal to zero as well, pwinner is AND (Boolean, shown by $\sum$) by the matrix obtained by rotating(shown by ∇) pwinner in the range [-neisize , neisize]. Rotating a matrix involves changing the order of columns circularly.

$$pneighbor = pwinner$$
$$pneighbor = \sum \left(pneighbor\ , \nabla_{i=-neisize}^{neisize}\ (\ pwinner\ )\right) \quad (8)$$

d) Now pneighbor contains zero for neurons to be updated and non-zero otherwise. It is converted to a matrix with '1' in place of '0' and '0' otherwise to make the binary mask pmask.

6) To apply this mask, (5) is slightly modified as follows

$$w_{ij} = (1-\alpha)w_{ij} + \alpha\delta x_i \quad (9)$$

where x_i is the slice of pinput which has been replicated to be of size ni × no and δ is the slice of pmask which has been similarly replicated. To see why slicing is required we need to understand that the above mask is for all patterns, so while updating weights it has to be done for each pattern individually. Therefore, matrix is sliced row-by-row and each such slice is replicated vertically to make it conformable to be added to pweight. Both slicing and replication can be implemented on GPU as atomic instructions.

Steps 3-5 are implemented purely on a GPU.

The above operation is iterated in a loop numpat times, slicing each row successively.

After analyzing the above algorithm it is imperative to note that if Kohonen's algorithm is implemented as it is on a GPU without any modifications, then severe performance degradation can occur while finding the position of the winner neuron and while updating the weights in its vicinity. This is discussed in next section with empirical results. This is to easy to justify when we observe that the above operations increase the traffic between GPU and CPU, since in finding index in a normal way GPU cannot help us, we need to use a CPU. In that case only matrix multiplication operations will be implemented on GPU efficiently.

V. Experimental Results

A. Environment

The following experiments are conducted on a computer running Dual-Core AMD Turion 64 X2 1.66 GHz with 512 MB RAM and nVIDIA GeForce 6150 Go GPU with 256 MB Video RAM, supporting Pixel Shader Version 3.0.To implement the algorithm several GPGPU libraries/languages are available. They differ in the level of abstraction they offer to the programmer. High level shading languages like Cg [10], HLSL and OpenGL Shading Language (GLSL) [11] require knowledge of graphics API like OpenGL and DirectX. High level GPGPU languages/libraries like Brook [12], Sh, and Accelerator [13] focus more on algorithm rather than implementation and obviate the need to have deep knowledge of graphics extensions. Vendor specific extensions like nVIDIA Compute Unified Device Architecture (CUDA) [14] and AMD Close to Metal (CTM) [15] are also available which while providing high level accesses also expose some low level details to the programmer. Microsoft Research Accelerator has been chosen for implementing the algorithm. The Operating System chosen is Microsoft Windows XP SP2 with DirectX 9.0c and runtime environment is .NET 2.0 with C# 2.0 as the language.

Operations detailed in (6)-(7) to obtain winner matrix is implemented with the following statements in C# with Accelerator.

```
fmaxval =PA.MaxVal(PA.InnerProduct(dinput,dweight),1);
fmaxval= PA.Replicate(fmaxval, numpat, no);
winnerMatrix = PA.Subtract(facc, fmaxval)
```

PA is prefix indicating that it is a Parallel Array operation.

The library used supports data parallelism. Explicit partition of data (as has been shown in previous section since the algorithm by default is not data parallel) eliminates any data dependencies. This obviates the need for any synchronization primitives, one of the hardware limitations of a GPU.

B. Implementation Consideration

Since the maximum video memory available is 256 MB, all the operations detailed will operate on matrix of maximum size 4096 × 4096. High-end GPUs available have up to 768 MB video memory and hence considerably higher array sizes. Hence, even better results can be expected on these high-end GPUs. Research is on to make array size independent of memory available by using virtual graphics memory. The available runtime does not support 'scatter' operation for GPU, hence initialization, randomization are all done on CPU itself. Also unrolling the loop can be done only to a limited extent since the shader length cannot exceed the instruction limit. Only two-dimensional arrays are possible at present. Even though it is possible to implement higher dimensional arrays using lower dimension arrays but due to lack of GPU native support and primitives such an implementation is not efficient. This restriction is also due to the GPGPU library used.

An important consequence of these is that in following places in the algorithm, sequential looping is inevitable.

1) Slicing and replication operation while updating the weights.
2) Iterating the network for a fixed number of iterations.
3) Iteration of (8) over all rotations.

Even though slicing and replication mentioned above are GPU primitives they need to be repeated for all the rows corresponding to different patterns, hence the sequential looping. The effect of this is not as severe on the speedup obtained as anticipated.

Certain speedup obtained is lost in the process of obtaining the 'normal' arrays from parallel arrays since we need these in testing phase which is sequential and for storing them. Only primitives with 32-bit precision are considered, again constrained by GPU and GPGPU library.

Final and the most important issue is that GPU queues all its operations until we need explicit array representation or when the result is actually needed. Hence, to obtain speedup GPU is explicitly forced by using 'Evaluate' statements provided by library. The count and position of these statements determines the net speedup obtained. In the above algorithm, 'Evaluate' is used in (6) to obtain an additional speedup of 10% approximately. Usage without proper care can disturb the optimizations implemented internally by the GPU, hence, should be used carefully after analyzing the result. Empirical results obtained during the trials conducted have confirmed this.

C. Algorithmic Complexity

Before delving into the details of the results obtained analyzing the algorithm in a theoretical manner is necessary. Theta (Θ) notation has been adopted for asymptotic analysis. The assumption underlying the analysis is that all the GPU operations are Θ (1) treating it as a black-box. So in analyzing the running time we concentrate mainly on its sequential parts [16].

The analysis broadly concentrates on two major areas.

1) Building the update mask. The main operation is the iteration detailed in (8). This has complexity of Θ (neisize). But as we know neisize depends on number of output neurons (no). Hence, it becomes Θ (no).

2) Updating the weights. Here also we find a very conspicuous iteration which is of repeatedly slicing the rows and replicating them. This leads us to estimate it be Θ (numpat) which again is equal to Θ (numimage).

Putting them all together, for a single iteration they become Θ (no) + Θ (numimage).

Extending them over iterations n, they become

$$\theta(n * no) + \theta(n * m) + k \tag{10}$$

where k is a constant signifying cost of disposing the video memory in each iteration and m(=numimage) is the number of images.

In contrast while analyzing the CPU version, following procedure is adopted.

To find the position of winner neuron for each pattern and for each iteration the complexity is Θ(ni * no + no).

To update the weights similarly we can find that it is equal to O(no * ni) which reduces to Θ(neisize * ni) since neisize number of neurons are actually updated.

Therefore, for a single pattern and single iteration we find that it is Θ(ni*no + no) + Θ(neisize * ni).

Extending to 'm' patterns (= numimage) and 'n' iterations it becomes

$$\theta(m * n * ni * no + no * m * n) + \theta(neisize * ni * m * n) \tag{11}$$

Comparing (10) and (11) it is clear that the CPU version has the added complexity of 'ni', the length of input vector and 'neisize', the neighborhood distance and 'm', the number of images. This is not surprising since the parallelization is prominent in the sections where these parameters are involved.

D. Results

All the experiments conducted vary CPU and GPU versions on following parameters with respect to time:

1) Number of patterns.
2) Number of iterations.
3) Network size, defined as number of input neurons × number of output neurons.

Nearly 10 to 20 trials were conducted for each test case; hence the execution time considered here is average case. Nature of results produced is identical in both cases, hence only running time is considered for evaluation.

The measurement of time is accurately done with negligible skew, with minimum resolution of 0.838μs using Win32 API QueryPerformanceCounter and DirectX timer provided by the runtime. The measurement overhead is about 0.1927μs which is used to further adjust the readings obtained. It can also be noted that due to the difference in the order of time complexity between the two, these discrepancies can be ignored.

In Fig. 2. number of patterns available is varied against time. Here input layer size is 1000 and output layer size is 2000 with alpha being 0.4. Fig. 3. sketches the loci of the network size curve with respect to time. Here the number of patterns is 20 and alpha is 0.4. The dip in the GPU curve can be explained by its caching (a similar computation is performed faster) and pipeline optimizing capabilities. And lastly variation of running time of the algorithm with number of iterations is shown in Fig. 4..The difference between this and other two is conspicuous because of the presence of loop overhead here and hence even GPU curve is linear. The speedup derived also depends on arithmetic intensity. Arithmetic intensity is defined the ratio of the computation performed to the bandwidth ratio. In the trials of Fig.2. and Fig. 3. arithmetic intensity is considerably higher than that of trials in Fig. 4., hence the observed speedup.

Some significant observations which can be made are,

1) At initial stages running time in case of CPU curve is significantly less compared to that of GPU. This can be explained by the fact that at initial stages overhead is considerably high and at later stages gain factor dominates if following relation is considered.

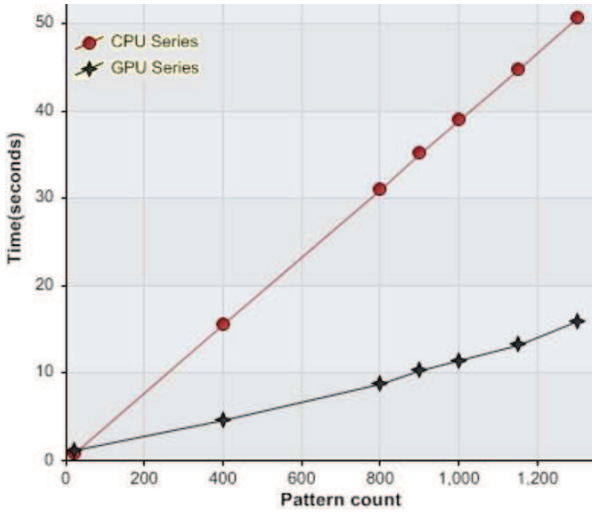

Fig.2. showing variation of pattern count with respect to time.

Net gain = gain due to parallelism – overhead (12)

The overhead is manifests mainly in the form of communication cost between the main memory and the video memory.

2) As the size (pattern or network size) factor doubles or increases by greater amounts, CPU curve is nearly linear (Fig.2.) or exhibits exponential characteristics (Fig.3.).On the other hand, the growth rate of GPU curve is comparatively low and does not grow with problem size in same way. This reassures the fact that the gain obtained from GPU is high if the problem is of considerable size as from (12).

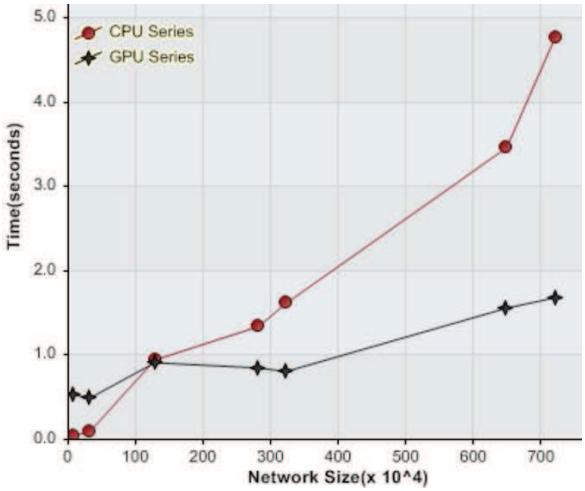

Fig.3. Curve describing variation of time taken with respect to network size which is in the order of 10000.

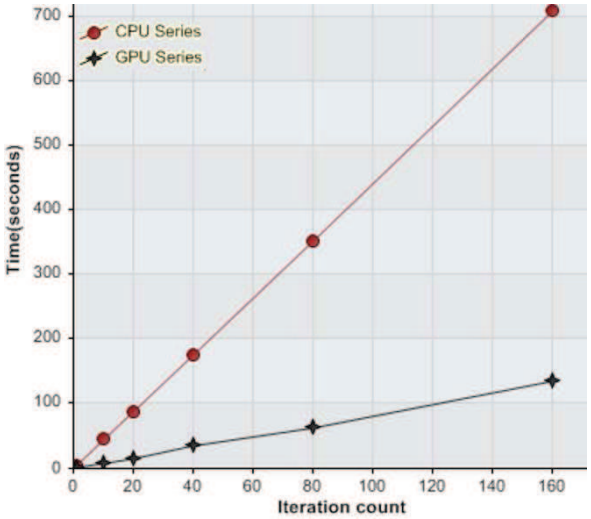

Fig.4. showing the similarity in behavior exhibited when iteration count is varied against time.

The final experiment conducted is to compare the performance of SOM on a GPU with and without any modifications. The performance of the algorithm without any modification is as worse as being implemented on a CPU. The reason being that, when implemented as it is, several CPU instructions are interleaved between GPU ones, thus incurring a great amount of overhead. A great deal of overhead is also incurred in transferring results between CPU and GPU regularly. One such example is the operation used to find the position of winner neuron. After obtaining the necessary product of weight and input matrices on GPU, it needs to be converted from a GPU array to a normal primitive array and then maximum needs to found and position determined through a series of iterative steps. And in steps following it, again it needs to be converted to a GPU array. This drives up the complexity making it as worse (or sometimes even more) as running it on a CPU as shown in Fig.5.

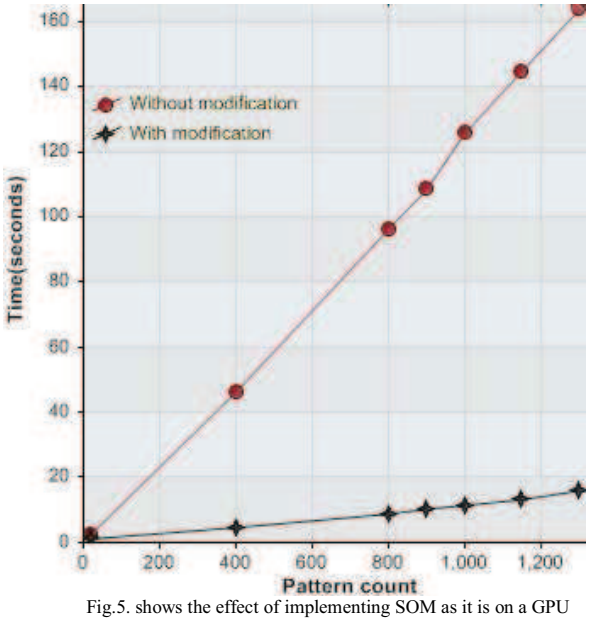

Fig.5. shows the effect of implementing SOM as it is on a GPU and with modifications.

After observing the results of the experiments conducted and analyzing them, it is necessary to compare it with the theoretical bounds derived in the previous section. The experimental results match the theoretical bounds when the parameters considered are large in number. However, when the number of parameters is small, actual performance fails to match the theoretical performance. As discussed in (12), this is mainly due to the overhead factor which dominates. Runtime and API overhead also contribute to this. At the same time, performance observed at greater values has surpassed the theoretical limits. This is mainly due to the optimizations implemented by the GPU internally. More deviation from the theoretical bounds can be observed if the number of sequential components in the algorithm increases as evident from Fig. 5.

VI. Conclusion

The implications of designing an algorithm for a GPU and using that algorithm in pattern classification has been presented in this paper supported by the results of a series of tests conducted. These tests have clearly shown that the algorithm design for a GPU is still in its growing phase and GPU can complement a CPU, if not replace it for some time to come. Nevertheless, this work paves way for future attempts to implement general tasks on a GPU overcoming many of its shortcomings.

Future Work can be in the following areas:

1) Increasing the degree of parallelism of the algorithm by reducing sequential iterations in the algorithm.

2) Enhancing the arithmetic intensity of the algorithm to increase the speedup.

3) Transformation of existing iterative phases into GPU primitives.

4) Handling image processing on the GPU in a more 'native' way to broaden the scope for achieving parallelism.
5) Achieving initialization, randomization on GPU itself i.e. efficient implementation of 'scatter' operation.
6) Hand-optimizing the algorithm further to increase its efficiency.
7) Overcoming the restriction on the size of the images imposed by the video memory of GPU.

REFERENCES

[1] Teuvo Kohonen, Self-organizing maps Springer Verlag, New York, 1997.
[2] Samuel Kaski, Teuvo Kohonen, "Winner-take-all networks for physiological models of competitive learning," Neural Networks, vol. 7,(6-7), pp. 973-984, 1994.
[3] Vieira, F. Neto, A. Costa, J., "An efficient approach of the SOM algorithm to the traveling salesman problem," in VII Brazilian Symposium on Neural Networks, 2002, pp. 152-.
[4] Hannes Schabauer, Erich Schikuta, Thomas Weishäupl, "Solving Very Large Traveling Salesman Problems by SOM Parallelization on Cluster Architectures," in PDCAT 2005, pp. 954-958.
[5] Mark Harris, "Mapping computational concepts to GPUs," in SIGGRAPH '05: ACM SIGGRAPH 2005 Courses, ACM, 2005, pp. 50.
[6] Zhongwen Luo, Hongzhi Liu, Zhengping Yang, Xincai Wu, "Self-Organizing Maps computing on Graphic Process Unit," in 13th European Symposium on Artificial Neural Networks, Belgium, 2005, pp. 557-562.
[7] Victor-Emil Neagoe, Armand-Dragos Ropot, "Concurrent Self-Organizing Maps for Pattern Classification," in 1st IEEE International Conference on Cognitive Informatics, 2002, pp. 304-312.
[8] Alexander Campbell, Erik Berglund and Alexander Streit, "Graphics Hardware Implementation of the Parameter-Less Self-organizing Map," in Intelligent Data Engineering and Automated Learning - IDEAL, 2005, pp. 343-350.
[9] Kyoung-Su Oh, Keechul Jung, "GPU implementation of neural networks," Pattern Recognition, vol. 37, (6), pp. 1311-1314, 2004.
[10] Randima Fernando, Mark J. Kilgard, The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Addison-Wesley, 2003.
[11] Randi J. Rost, OpenGL Shading Language Pearson Education, Inc, 2004.
[12] Ian Buck, Tim Foley et al., "Brook for GPUs: Stream Computing on Graphics Hardware," in SIG-GRAPH '04, ACM, 2004, pp. 777-786.
[13] David Tarditi, Sidd Puri, and Jose Oglesby, "Accelerator: using data-parallelism to program GPUs for general-purpose uses," Microsoft Corporation, Tech. Rep. MSR-TR-2004-184, December, 2005.
[14] Hubert Nguyen. GPU Gems 3 Addison-Wesley Professional, 2007.
[15] Justin Hensley. (2007,Aug.). Close to the Metal. Presented at SIGGRAPH'07.[Online].Available: http://ati.amd.com/developer/gdc/2007/Hensley-Close_to_the_Metal(Siggraph07_GPGPUCourse).pdf.
[16] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.