SOMGPU :

An Unsupervised Pattern Classifier on Graphical Processing Unit

Raghavendra D Prabhu

EC0253

Introduction

- Self Organizing maps(SOM) – competitive unsupervised learning
- Kohonen’s algorithm and application to pattern classification
- Input vectors from image and random 2-D quadratic weights
- Winner Takes All (WTA) strategy
- Parameters of the algorithm - alpha, neighborhood size and Mexican Hat function
- Applications of SOM - NP-Complete - approximate

Introduction(contd.)

- Implementing SOM on sequential or pseudo-parallel machines for real life problems
- Comparison to a human brain
- Prominent role played by GPU and the analogy - size of the problem
- “Embarrassingly parallel”
- SPMD tasks and SOM – processing cost
- GPGPU libraries
- Automatic Parallelization – burden on compiler
- Other Neural Network and AI environments

Related Work

- Explicit location of winner - multi-pass method - update of weights - OpenGL (PBuffer) - limitations
- Fundamental difference in the approaches
- Concurrent Self Organizing Maps – accuracy
- Use of Cluster Architecture – SDP
- Vectorisation, Partitioning of parameter-less SOM
- Only for matrix multiplication operations – converting several inner product operations to a single matrix operation

Design of the problem

- Construct a vector representing the image – reduction and sampling
- Length of the input vector and size of VRAM
- Method adopted:
  - Binary matrix from image
  - Bounding box algorithm
  - Sampling with padding
    - Same as image convolution with filter of value 1
    - Implementation of sampling on GPU

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

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

Design(contd.)

Algorithm without GPU:

1. 2-D Weights are randomized and normalized
2. For each pattern in the set
   1. The winner neuron is selected among others based on maximum activation
   2. Neurons in the neighborhood of the winner neuron have their weights updated

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

   3. Neighborhood size and learning rate α are decreased accordingly

- Output of training phase is a set of weights which map

the input domain preserving topological ordering

Mapping to GPU

- Algorithm is by itself not data parallel - types
- Fragments which can be parallelized - spatial and temporal dependency
- Primitives do not permit index of array element to be extracted
- Role played by the winner neuron - To indicate the neurons whose weights need to be updated
- Obtain the position implicitly to update weights using a mask based approach

Mapping to GPU(contd.)

Revised algorithm

1. Vectors representing the image are obtained as before
2. Floating Point Array representation for array – Disposable Arrays
3. Size of input matrix and weight matrix – patterns, input and output neurons
4. *pacc* - matrix product of input and weight matrix
5. Maximum element is found for each row into *pmxval*
6. Index of the winner neuron cannot be obtained – coarse grained

Mapping to GPU(contd.)

- A new binary matrix to act as a mask

Winner neuron

0	0	0	1	1	1	1	1	1	1	0	0	0	0	0
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Binary matrix

1. pmxval, the column vector with maximum values is replicated along x-direction
2. New matrix, *pwinner* obtained by subtracting pmxval from *pacc*
3. *pwinner* is AND with matrices obtained by rotating *pwinner* in the range *neisize* to obtain *pneighbor* – necessity
4. *pmask* is obtained by transforming *pneighbor*
5. Weight update equation is slightly modified

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

Binary matrix(contd.)

- Matrices are *sliced* row-by-row and each slice is *replicated* vertically to make it conformable – Need for slicing
- Operations implemented using GPU primitives – slicing, rotating, subtracting, matrix multiplication, replication, inner product.
- Steps detailed above repeated till there is convergence or max iterations reached
- Performance degradation occurs if original algorithm implemented as it is - increased traffic – previous work

Environment

- Dual-Core AMD Turion with 512 MB RAM and GeForce 6150 Go GPU with 256 MB
- Accelerator – GPGPU library .NET 2.0 runtime with C# 2.0 as the language and DirectX 9.0c
- GPGPU libraries available with different level of abstractions – Cg,Sh,Brook,CUDA,CTM

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

Implementation Considerations

- Limitations on the size of video memory and the operations which can be implemented
- Limitations on the shader length – unrolling the loop
- Only two dimensional arrays possible - higher dimensions from lower arrays
- Inevitable sequential looping – network iteration, successive slicing and replication, successive rotations
- Data parallel library – explicit partition of data – synchronization primitives not needed
- Queuing of operations by GPU – Evaluate statement

Algorithmic Complexity

- Concentrate mainly on sequential areas in theta asymptotic analysis
- Two major areas - Building the update mask and updating the weights
- Over ‘n’ iterations, complexity in case of GPU

$$\theta(n * no) + \theta(n * m) + k$$

- In case of CPU – finding winner neuron and update

$$\theta(m * n * ni * no + no * m * n) + \theta(neisize * ni * m * n)$$

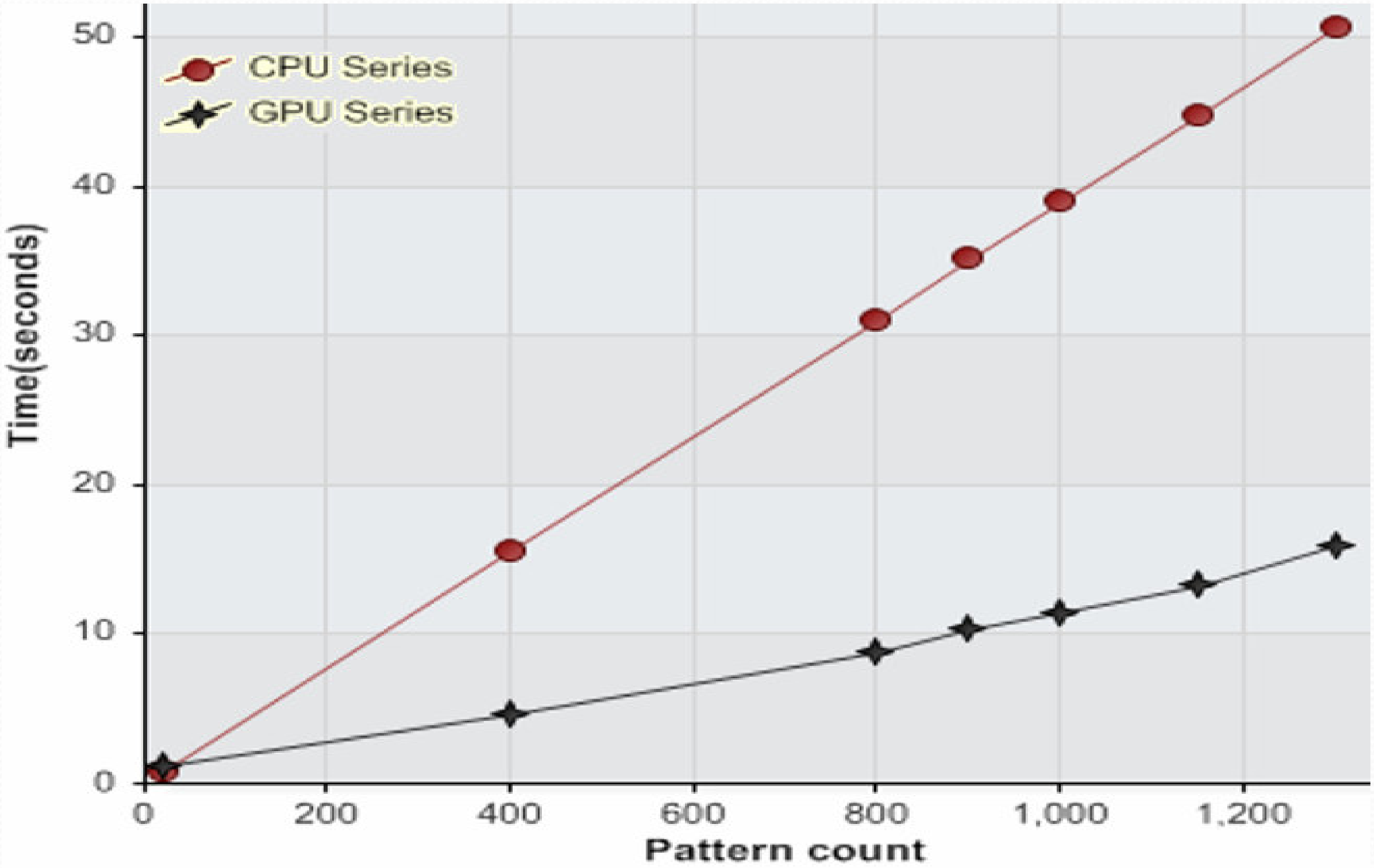
- Theoretical comparison between the two and assumptions

Results

- Comparing the time required by CPU and GPU while varying number of patterns, iterations and network size
- Counters used - QueryPerformanceCounter and DirectX timer and associated discrepancies – necessary assumption
- Nature of results produced is identical in both cases, hence only running time is considered for evaluation
- Time taken by GPU - compilation, loading and execution

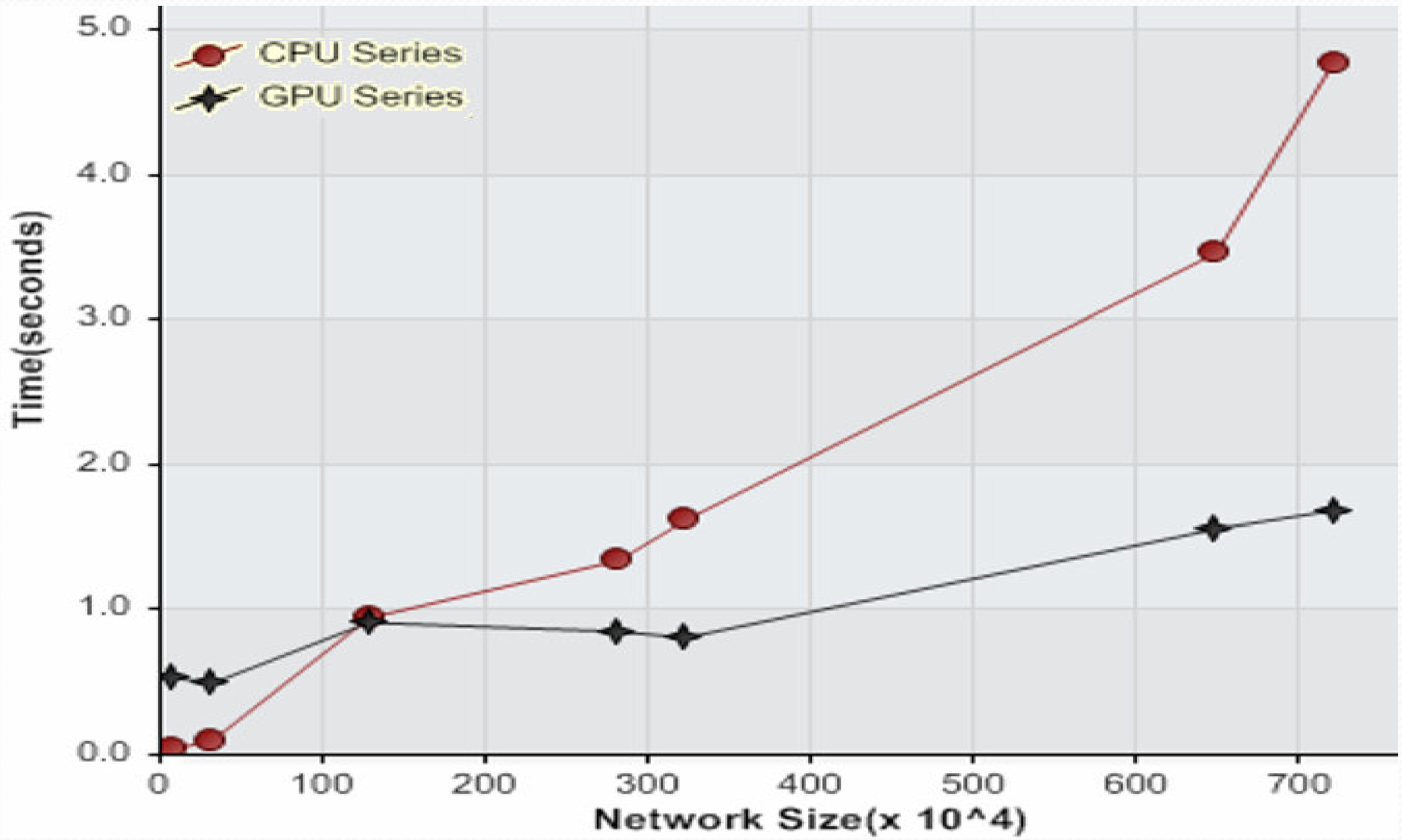
Result – I: Pattern

- Input layer = 1000 Output layer = 2000 alpha = 0.4



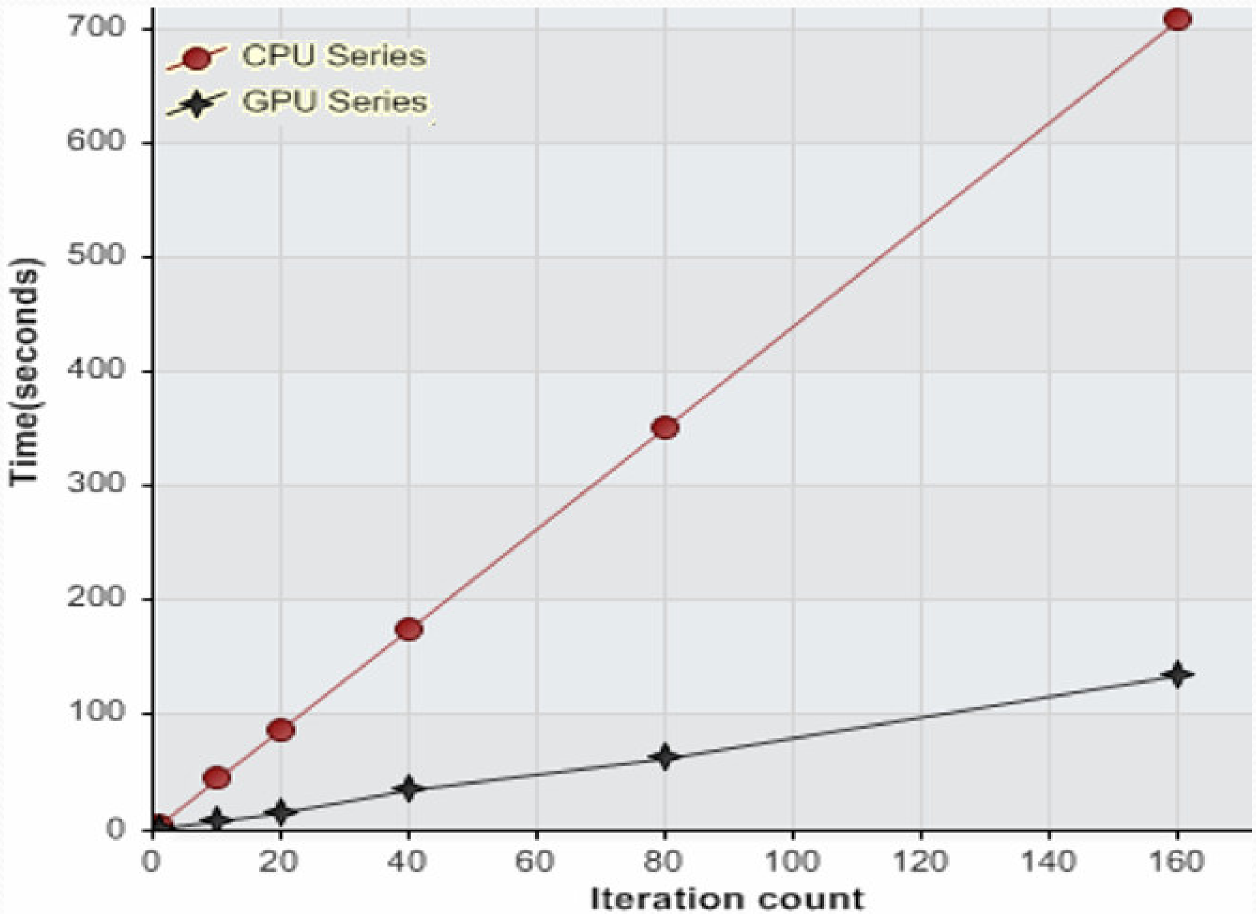
Result – II: Network Size

- Number of patterns = 20 alpha = 0.4
- Dip in the curve



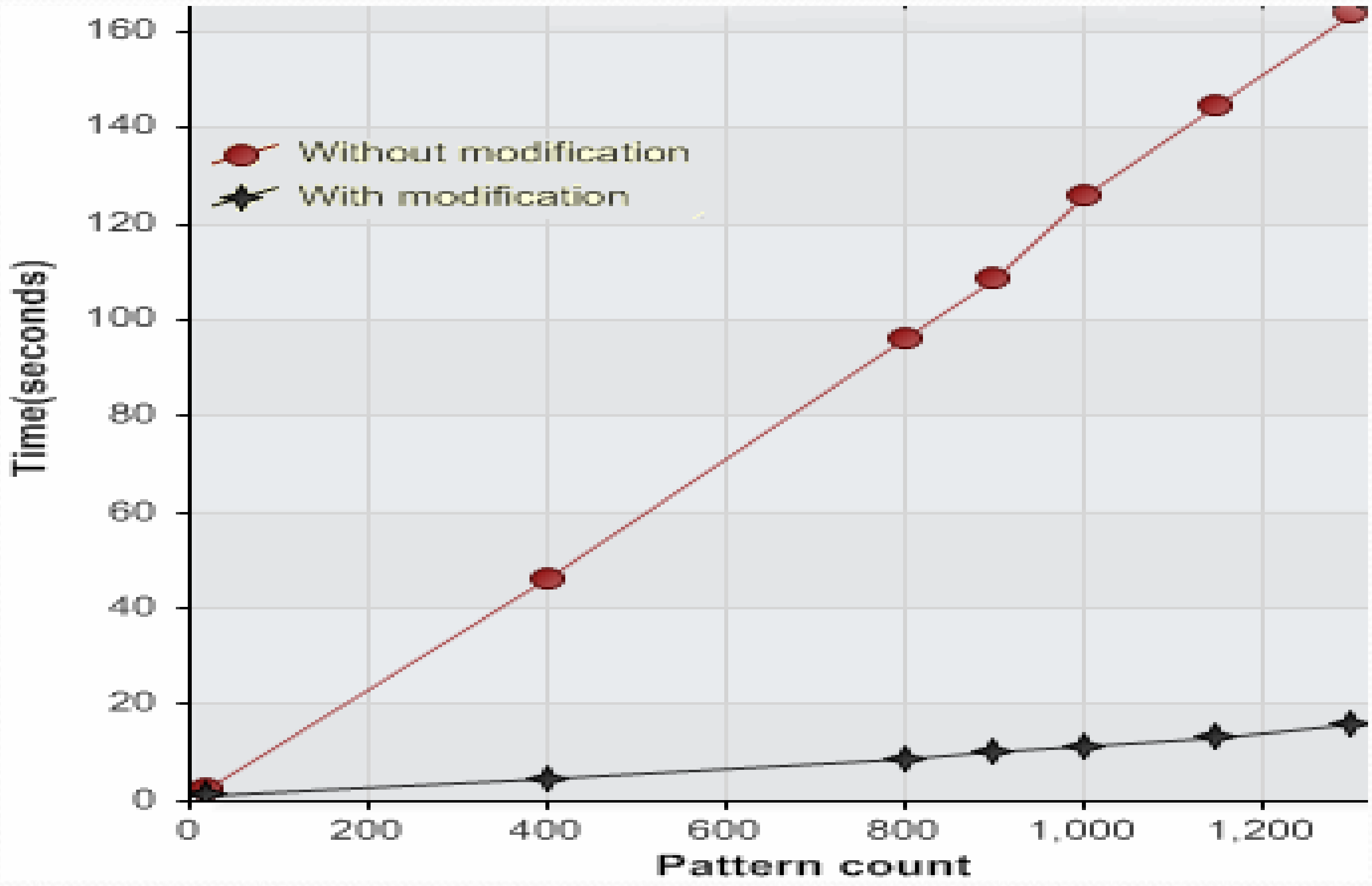
Result – III: Iterations

- Iteration overhead



Result – IV: Modification

- Position of winner neuron is explicitly obtained on CPU and result transferred to GPU – only matrix multiplication



Observations

- Arithmetic intensity and its effects
- Difference between 3rd and 1st,2nd - GPU curve
- Domination of CPU in earlier stages – overhead
- Growth rate as problem size dominates
- Performance loss caused by interleaving CPU instructions as in Result - IV -- importance of the algorithm - previous work
- Compare theoretical bounds with results - number of sequential components - basic assumptions - internal optimizations

Conclusion

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.

Algorithm design for a GPU is still in its growing phase

GPU can complement a CPU, if not replace it for some time to come.

Future Work

- Increasing the degree of parallelism
- Enhancing the arithmetic intensity
- Transformation of existing iterative phases into GPGPU primitives
- Overcoming the restriction on the size of the images imposed by the video memory of GPU
- Achieving initialization, randomization on GPU itself i.e. efficient implementation of ‘scatter’ operation