High Performance Pattern Recognition on GPU

Sheetal Lahabar   Pinky Agrawal   P. J. Narayanan
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, 500032 INDIA
{sheetal@students., pinky@students., pjn@}iiit.ac.in

***Abstract*—The pattern recognition (PR) process uses a large number of labelled patterns and compute intensive algorithms. Several components of a PR process are compute and data intensive. Some algorithms compute the parameters required for classification directly for each test pattern using a large training set. Most algorithms have a training step, the results of which are used by a computationally cheap classification step. In this paper, we present high-performance pattern recognition algorithms using a commodity Graphics Processing Unit (GPU). Our algorithms exploit the high-performance SIMD architecture of GPU. We specifically study the Parzen windows scheme for density estimation and the Artificial Neural Network (ANN) scheme for training and classification in this paper. We present fast implementations of these on a NVIDIA 8800 GTX GPU. Our implementation of Parzen windows can simultaneously estimate probability values for 1K test patterns in about $14ms$ based on an input data set of 16K patterns. Our ANN can run an epoch of batch-training on the NIST data set with 56K 484-dimensional patterns and 10 output categories in less than 200 milliseconds. The speedup is more than 300 times for Parzen windows and 100 times for ANN over the CPU implementations using a commodity GPU that costs about $\$400$.**

I. Introduction

Pattern recognition is concerned with the design of systems that detect trends and classify patterns. Important application areas are optical character recognition, speech recognition, fingerprint identification, DNA sequence identification, and many more.

Most pattern recognition systems have two components: training and classification [5]. The system is trained using a large number of labelled patterns using relevant features. The training could be iterative as with ANN, Support Vector Machines (SVM), etc., and proceeds till the error on a small set of testing patterns is sufficiently low. The trained system can be used for classification of unknown patterns using a computationally low process. The training process is compute intensive and time consuming especially when large number of training patterns are involved. The classification accuracy often depends on the effort spent on training and most applications settle for a suitable tradeoff. Using new training data to improve the performance is uncommon due to the large computational effort. There are other types of pattern recognition techniques that use the whole training set to directly evaluate parameters for classification of each unknown pattern. These incur heavy computational cost for classifying each pattern as computation involves all the input patterns. Parzen windows, k-Nearest Neighbour, etc., are examples of the same.

The rapid increase in the performance of graphics hardware have made GPU a strong candidate for performing many compute intensive tasks. GPUs now include fully programmable processing units that follow a stream programming model and support vectorized floating-point operations. High level languages have emerged to support the new programmability. NVIDIA's 8-series GPU with CUDA Computing environment provides the standard C like language interface to the programmable processors, which eliminates the overhead of learning an inadequate API [13]. GPUs provide tremendous memory bandwidth and computational horsepower. For example, the NVIDIA GeForce 8800 GTX can achieve a sustained memory bandwidth of 86.4 GB/s and a theoretical maximum of 346 GFLOPS [13].

Several GPU algorithms have been developed for sorting [7], geometric computations, matrix multiplication, FFT [11] and graph algorithms. Larsen and McAllister [10] initially proposed an approach for computing matrix products using simple blending and texture mapping functionalities on GPUs. Hall *et al.* [8] and Moravanszky [12] described improved algorithms that performs implicit blocking. Fatahalian *et al.* [6] proposed another approach based on blocking for computing matrix products using fragment shaders. NVIDIA's CUBLAS library [14] which comes with the CUDA software pack is an implementation of simple BLAS (Basic Linear Algebra Subprograms) on GPU, that allows optimized matrix and vector operations. Davis [4] presented simulation of ANN on GPUs using Brook [2]. Reiter *et al.* [16] described HMM search implementation to compute the Viterbi probability for biological protein sequences. Cao *et al.* [3] presented algorithm for scalable clustering on GPUs. However, little work has been done to exploit the computational capability of GPUs for highly compute intensive aspects of pattern recognition, such as ANN training and Parzen-windows. GPU applications on these can be useful, for instance, retraining the network with new training patterns added on the fly.

The Parzen-window approach is a method of estimating non-parametric density from observed patterns. In the classifiers based on Parzen-windows, the densities are estimated for each category and the test pattern is classified by the category corresponding to the maximum posterior. We describe a parallel implementation of Parzen-windows using CUDA API that can be used to classify large number of test patterns in parallel. We can estimate the probability densities for a test pattern in $14\mu s$ using 16K input patterns. This makes Parzen-window based classifiers practical. The training of the ANN changes its parameters based on the signals that flow through it. It is an iterative process, where each iteration has high computational complexity. We describe the batch learning of network as a set of matrix operations, which are well suited to the GPUs, because of highly parallel computational requirements and regular data access pattern. We implemented the backpropagation ANN training algorithm using fragment shaders and CUBLAS library. Our ANN batch-training can run an epoch on a data set with 56K 484-dimensional training patterns in less than $200ms$.

II. Preliminaries

We review the concepts related to the GPU and the PR algorithms we address in this section.

A. GPU Architecture

GPUs have a parallel architecture with massively parallel processors. The graphics pipeline is well suited to the rendering process because it allows the GPU to function as a stream processor. Recent GPUs with Shader Model 4 [1] allow users to write vertex, fragment and geometry shader programs as shown in Figure 1. The programmable parts of the graphics pipeline operates on a large number of vertices and fragments spawning a thread for each, to keep the parallel processors occupied. The General-Purpose computation on Graphics Processing Units (GPGPU) uses GPU for non-graphics computations by posing it as a graphics rendering problem. Most GPGPU algorithms use programmable fragment processors, as it the most parallelizable component of the pipeline, which maps each input pixel to an output pixel of the framebuffer. Since, GPU memory layout is optimized for graphics rendering, an optimal data structure may not be available for GPGPU solutions. Creating efficient data structures using the GPU memory model is a challenging problem in itself. Memory size and operations (*gather* and *scatter)* of the GPU are other restricting factors.

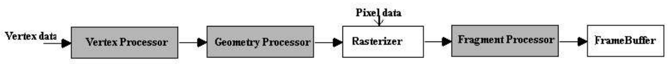

Fig. 1. The graphics pipeline with the programmable stages shown shaded

NVIDIA's GeForce 8-series GPUs with the CUDA programming model provides an adequate API for non-graphics applications. The CPU sees a CUDA device as a multi-core co-processor. CUDA design does not have memory restrictions of GPGPU. It increases the programming flexibility by providing both *scatter* and *gather* memory operations i.e. ability to read and write at any location in memory.

At the hardware level, NVIDIA's 8-series GPU is a set of SIMD multiprocessors with eight processors each. Each multiprocessor contains a parallel data cache or *shared memory*, which is shared by all its processors as shown in Figure 2.

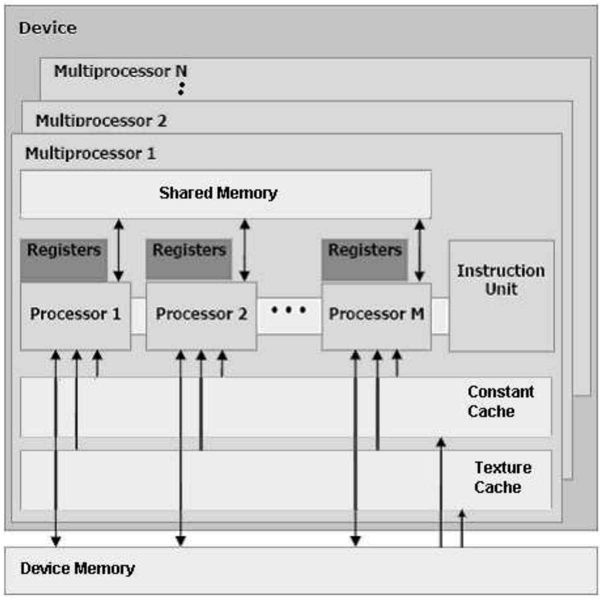

Fig. 2. A set of SIMD multiprocessors with on-chip shared memory

It also has a read-only *constant cache* and *texture cache* that is shared by all the processors. A set of local 32-bit *registers* is available per processor. The multiprocessors communicate through the *global* or device memory. At the software level, the CUDA model is a collection of *threads* running in parallel. A thread block is a batch of SIMD-parallel threads that runs on a multiprocessor at a given time and can communicate through shared memory and can be synchronized. The computations are organized as a *grid* of *thread blocks* as shown in Figure 3. Each thread executes a single instruction set called the kernel. Thus, the CUDA model allows programmers to better exploit the parallel power of the GPU for general-purpose computing.

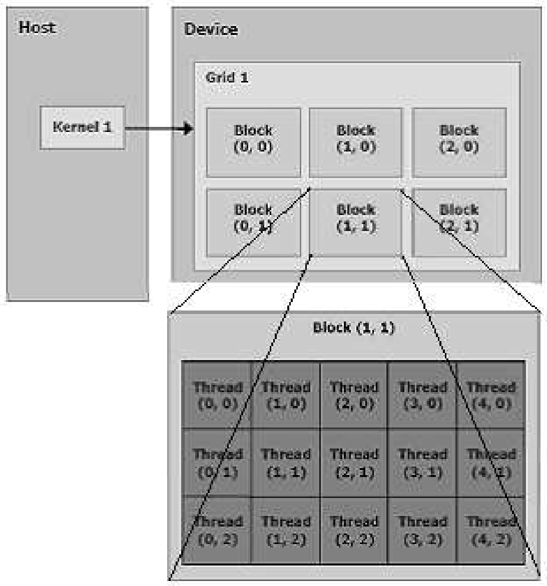

Fig. 3. CUDA programming model

B. Parzen Windows

Parzen-window method [15] approximates the unknown density function $p(\mathbf{x})$ from the N patterns $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ from a category. The N^{th} density estimate for $p(\mathbf{x})$is given as:

$$p_N(\mathbf{x}) = \frac{1}{Nh_N}\sum_{i=1}^{N}\varphi\left(\frac{\mathbf{x}-\mathbf{x}_i}{h_N}\right), \quad (1)$$

where $\varphi(\mathbf{x})$ is a kernel density function and h_N is the window width. $p_N(\mathbf{x})$ converges to original $p(\mathbf{x})$, if $h_N \rightarrow 0$ when $N \rightarrow \infty$.

Parzen-window method can be implemented in a parallel fashion for computing the probability estimate based on N normalized d-dimensional patterns, randomly sampled from c classes, since the probability estimate for each class is independent as given by Equation 1. For this two tables are required. The first table has N d-dimensional entries of the input patterns. This table is used to compute the N kernel density functions for a given test pattern. Gaussian kernel density function is given by $e^{(\mathbf{x}_i^t\mathbf{x}-1)/\sigma^2}$, where σ determines the width of the effective Gaussian window. Another table has N c-dimensional entries which give the information about the category of the input patterns. The probability that x belongs to a category is computed by summing the kernel density function for all the input patterns belonging to that category.

The parallel implementation of Parzen-window can be achieved by a series of matrix operations. Given the input patterns, we form a $N \times d$ matrix T_1 for the first table, where each row contains an input pattern. A $N \times c$ matrix T_2 is formed for the second table, where each row has 1 in the column corresponding to its category while others are 0. For classifying m unknown patterns we form a $m \times d$ matrix K and the classification process is expressed as:

$$\begin{aligned} I &= g(K * T_1^T), && (2)\\ C &= I * T_2, && (3) \end{aligned}$$

where, $g(A)$ denotes that $e^{(a_{ij}-1)/\sigma^2}$ is computed for every element of the matrix A. Each element C_{ij} of C denotes the probability estimate of i^{th} unknown pattern belonging to the j^{th} category.

C. ANN: Training and Classification

Neural Networks have two primary modes of operation: feedforward and training. Figure 4 shows a simple three-layer ANN. During the feedforward operation, a d-dimensional input pattern $\mathbf{x}$ is presented to the input layer; each input unit then emits its corresponding component x_i. Each of the n_H hidden units computes its net activation, net_j, as the inner product of the input layers signals with weights w_{ji} at the hidden unit. The hidden unit then emits intermediate vector y_j as in Equation 4. Each of the c output units functions in the same manner as the hidden units do, computing net_k as the inner product of the hidden unit signals and weights at the output unit, as in Equation 5. The final signals z_k emitted by the network, are used as discriminant functions for classification.

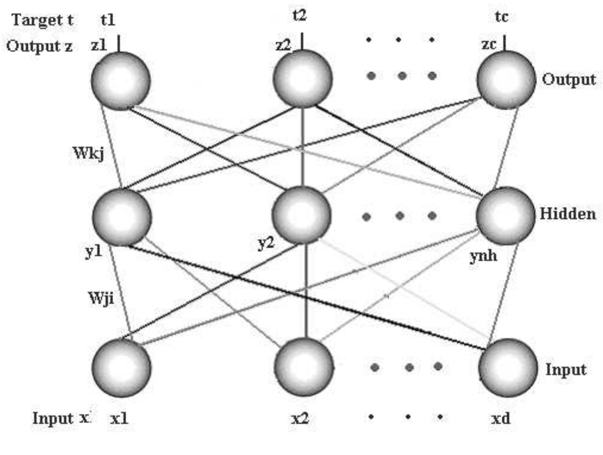

Fig. 4. d-n_H-c three-layer ANN

$$y_j = f(net_j) = f(\sum_{i=1}^{d} x_i w_{ji}) = f(\mathbf{w}_j^t\mathbf{x}) \quad 1 \leq j \leq n_H, \quad (4)$$

$$z_k = f(net_k) = f(\sum_{j=1}^{n_H} y_j w_{kj}) = f(\mathbf{w}_k^t\mathbf{y}) \quad 1 \leq k \leq c, \quad (5)$$

where $f(.)$ is the non-linear activation function like sigmoid.

Backpropagation is a general method for supervised training of multilayer neural networks based on a gradient descent procedure. During network training, the output signals are compared with a target vector **t**, and any difference (training error) is used in training the weights throughout the network. This process is repeated until the error falls below a threshold. The weights are changed in the direction that will reduce the error. The hidden-to-output and the input-to-hidden weights are updated using a learning rate η as:

$$\begin{aligned} \Delta w_{kj} &= \eta(t_k - z_k)f'(net_k)y_j && (6)\\ \Delta w_{ji} &= \eta\left[\sum_{k=1}^{c} w_{kj}(t_k - z_k)f'(net_k)\right]f'(net_j)x_i. && (7) \end{aligned}$$

In the on-line version of backpropagation, the weights are updated after presenting each input pattern while in the batch training, all the training patterns are presented first and their corresponding weight updates summed; only then are the actual weight vectors are updated. Batch training has better convergence properties.

The batch training of ANN can be reformulated as a series of matrix operations which are inherently parallel. For N d-dimensional input patterns, each belonging to one of the c categories, we form a $N \times d$ input matrix X by stacking the input vectors rowwise and similarly a $N \times c$ target matrix T is formed by stacking the c-dimensional target vectors. A $n_H \times d$ input-to-hidden weight matrix W_{JI} and a $c \times n_H$ hidden-to-output weight matrix W_{KJ} are initialized with random values, where n_H are the number of hidden units. Hence, Equations 4-7 as matrix operations become:

$$Y = f(net_J) = f(X * W_{JI}^T), \tag{8}$$

$$Z = f(net_K) = f(Y * W_{KJ}^T), \tag{9}$$

$$\Delta W_{JI} = \eta\left[\{((T-Z)\ .\ f'(net_K)) * W_{KJ}\}\ .\ f'(net_J)\right]^T * X, \tag{10}$$

$$\Delta W_{KJ} = \eta\left[(T-Z)\ .\ f'(net_K)\right]^T * Y, \tag{11}$$

where (.) means element-element multiplication and $f(M)$ means sigmoid of every element of the matrix M.

III. Implementation on the GPU

In this section, we present the implementations details of the above algorithms.

A. Parzen Windows on GPU

The density estimation algorithm for m unknown patterns using Parzen-windows is given in Algorithm 1. We use the same notations as described in Section II-B.

Algorithm 1 Parzen-windows using matrix-math

```
$M \leftarrow K * T_1^T$     } CUMAT
$I \leftarrow g(M)$          } Kernel
$C \leftarrow I * T_2$       } CUMAT
```

CUMAT refers to the *cublasSgemm()* function for matrix multiplication, provided by the CUBLAS library [14]. Similarly, Kernel refers to the *kernel function*, which is executed on the GPU. We divide the $m \times N$ matrix I into $\frac{m}{16} \times \frac{N}{16}$ thread blocks, with 16×16 threads per block. The hardware map thread blocks to parallel multiprocessors on the GPU. The Kernel function is executed for every thread.

Kernel:
$C[i,j] = e^{(C[i,j]-1)/\sigma^2}$

B. Backpropagation on GPU

The backpropagation training algorithm using matrix operations is given in Algorithm 2. We will follow the same notation for matrices as discussed in Section II-C. The network parameters, initial synaptic weights, number of hidden units, sigmoid function and learning rate are set according to the techniques for improving backpropagation by Haykin [9].

Algorithm 2 ANN training — Using CUDA — Using Shaders

```
for $i = 0$ to $epochs$ do
    $net_J \leftarrow X * W_{JI}^T$                          } CUMAT     } Shader 1
    $Y \leftarrow f(net_J)$                                  } Kernel 1  }
    $net_K \leftarrow Y * W_{KJ}^T$                          } CUMAT     } Shader 1
    $Z \leftarrow f(net_K)$                                  } Kernel 1  }
    $\delta_K \leftarrow \eta(T-Z)\ .\ f'(net_K)$            } Kernel 2  } Shader 2
    $\Delta W_{KJ} \leftarrow \delta_K^T * Y$                } CUMAT     } Shader 3
    $I \leftarrow \delta_K * W_{KJ}$                         } CUMAT     } Shader 4
    $\delta_J \leftarrow \eta I\ .\ f'(net_J)$               } Kernel 3  } Shader 5
    $\Delta W_{JI} \leftarrow \delta_J^T * X$                } CUMAT     } Shader 3
    $W_{JI} \leftarrow W_{JI} + \Delta W_{JI}$               } CUADD     } Shader 6
    $W_{KJ} \leftarrow W_{KJ} + \Delta W_{KJ}$               } CUADD     } Shader 6
end for
```

1) Using fragment shaders: This algorithm implements each epoch as a multipass method ($\frac{N}{4\times s}$ passes) requiring multiple renderings to the framebuffer. The input is stored into $\frac{N}{4\times s}$ 4-channel textures, s being the maximum allowable texture size. Our implementation packs 4 consecutive elements from a matrix column into a texel for input, target and other intermediate matrices. The 4 elements of a texel p can be accessed simultaneously in order as $p.xyzw$. The elements can also be accessed in arbitrary order (e.g. $p.yxwz$) and an element can be referenced multiple times (e.g. $p.xxxx$). For example: $r.xyzw = p.xxyy$, assigns the x element of p to the x and y element of texel r and similarly for other elements. Single channel textures are used for the weight matrices. This packing of data allows efficient shaders.

The matrix multiplication of any $P \times Q$ and $Q \times R$ matrix is a multipass algorithm, requiring $\frac{Q}{b}$ passes, where b is a scalar representing the block size that gives optimal performance [4]. Each pass accepts i, j, and k arguments via interpolated texture coordinates, multiplies b elements and accumulates the partial result of $C[i,j]$ using GL_BLEND, till all the Q elements are multiplied. After each pass, k is incremented by b.

Shader 1:

```
for $m = 0 \ldots b-1$ do
    $C[i,j].xyzw = A[i,m+k].xyzw * B[j,m+k].xxxx +$
                   $C[i,j].xyzw$
if last pass
    $C[i,j].xyzw = f(C[i,j].xyzw)$
```

Shader 2:

```
$C[i,j].xyzw = \eta(A[i,j].xyzw - B[i,j].xyzw) *$
               $f'(D[i,j].xyzw)$
```

Shader 3:

```
for $m = 0 \ldots b-1$ do
    $C[i,j].r = dot(A[m+k,i].xyzw * B[m+k,j].xyzw) +$
                $C[i,j].xyzw$
```

Shader 4:

```
for $m = 0 \ldots b-1$ do
    $C[i,j].xyzw = A[i,m+k].xyzw * B[m+k,j].xxxx +$
                   $C[i,j].xyzw$
```

Shader 5:
$C[i,j].xyzw = \eta A[i,j].xyzw \; * \; f'(B[i,j].xyzw)$

Shader 6:
$C[i,j].x = A[i,j].x + B[i,j].x$

2) Using CUDA API: In the Algorithm 2, matrix-matrix multiplications are done by using *cublasSgemm()* function referred to as CUMAT and the weight matrix is updated using *cublasSaxpy()* function referred to as CUADD. We write kernel functions for element-element multiplication and for computing the sigmoid, $f(.)$ of the net_j and net_k. We used 256 threads per block.

Kernel 1:
$C[i,j] = f(A[i,j])$

Kernel 2:
$C[i,j] = \eta(A[i,j] - B[i,j]) \; * \; f'(D[i,j])$

Kernel 3:
$C[i,j] = \eta(A[i,j]) \; * \; f'(B[i,j])$

IV. Results

We tested our algorithms on 3 GHz Pentium IV CPU and a NVIDIA GeForce 8800 GTX graphics processor. To generate fragment programs, we use NVIDIA's Cg compiler. We benchmarked our GPU algorithms for ANN training and the optimized CPU based ANN implementation provided by FANN (Fast Artificial Neural Network) and MATLAB.

N	t1 (ms) CUDA	t1/N (ms) CUDA	t2 (ms) MATLAB	t2/N (ms) MATLAB
1	0.319	0.319	10.0	10.00
16	0.434	0.027	70.0	4.37
128	1.870	0.014	520.7	4.06
256	3.647	0.014	1039.4	4.06
512	7.207	0.014	2083.0	4.06
1K	14.736	0.014	4166.0	4.06
2K	28.980	0.014	8360.1	4.08
4K	57.810	0.014	19450.0	4.74

TABLE I
PARZEN-WINDOW RESULTS FOR N PATTERNS

We used two datasets for our experiments. A dataset from http://archive.ics.uci.edu/beta/datasets/Letter+Recognition, consisting of 16K 16-dimensional feature vectors, where each input belongs to one of the 26 capital letters in the English alphabet is used to estimate the probability density for each of the 26 categories for the test patterns. We compute the density estimate for m test patterns in parallel. Table I shows the performance results of estimating densities for different number of test patterns. We observe that the time taken for estimating the probability densities for a test pattern decreases if the probabilities are estimated for a large number of test patterns in parallel. On an average our implementation achieves 325 times speedup over an implementation on MATLAB. For 64 or more test patterns the time taken for a pattern on CUDA is 14.1 μs.

We used another dataset from http://www.nist.gov, consisting of 22×22 binary images of handwritten numbers from 0-9. For this data, we trained a 3-layer ANN with 484 (all the image coordinates), 128 and 10 units at the input, hidden and output layers respectively. The ANN is trained with 8K to 56K input images. As shown in Table II and Figure 5, the CUDA implementation achieves 90-110 times speedup over MATLAB and 120-140 times speedup over FANN implementations. ANN training with 56K patterns for an epoch on CUDA requires $190ms$ as compared to $28.27s$ for FANN. The shader implementation achieves 40-50 times speedup over MATLAB and 55-70 times speedup over FANN implementations. For this large dataset, MATLAB gives memory problems when trained for more than 32K patterns.

N	t1 (s) CUDA	t2 (s) Shader	t3 (s) MATLAB	t4 (s) FANN
8K	1.41	3.23	128.78	141.83
16K	2.84	6.24	278.74	344.71
24K	4.30	8.80	423.02	524.16
32K	5.66	11.59	600.61	789.19
40K	6.96	14.37	-	961.67
48K	9.03	17.14	-	1179.26
56K	9.77	20.00	-	1413.66

TABLE II
ANN RESULTS FOR N PATTERNS FOR 50 EPOCHS

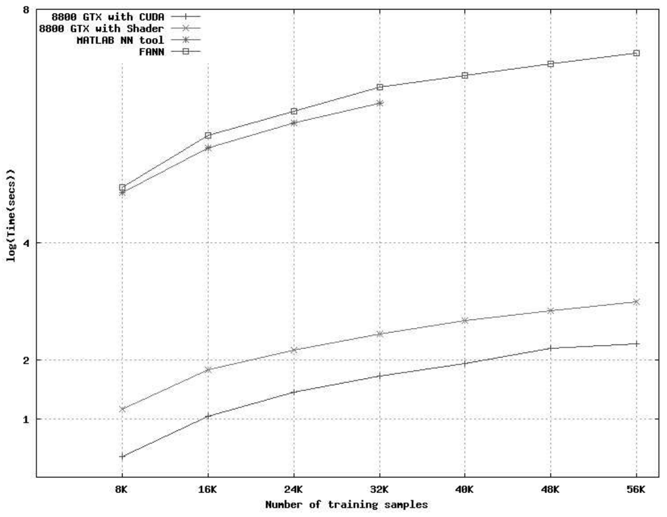

Fig. 5. Results for training ANN for 50 epochs

We tested the trained network with 2K unknown patterns. Training with 32K input patterns for 200 epochs takes $3472s$ and gives 84% accuracy on FANN, while GPU takes $23s$ and gives 82% accuracy. When trained for 1000 epochs GPU takes $113s$ and gives 89% accuracy. Table III shows the classification time for different number of test patterns. We observe that for more than 1K test patterns, the average classification time for a pattern is $1.45\mu s$.

N	t1 (ms) CUDA	t1/N (ms) CUDA	t2 (ms) FANN	t2/N (ms) FANN
128	0.83	6.48e-03	16.59	0.129
512	0.89	1.71e-03	68.27	0.133
1K	1.57	1.54e-03	134.37	0.131
2K	3.06	1.49e-03	266.60	0.129
8K	11.86	1.44e-03	1076.51	0.131
16K	24.04	1.46e-03	2139.71	0.130
32K	48.08	1.46e-03	4283.02	0.130
56K	83.49	1.45e-03	7574.14	0.132

TABLE III
CLASSIFICATION TIME FOR N PATTERNS ON ANN

V. CONCLUSIONS

In this paper, we presented the implementation of two data intensive pattern recognition operations on commodity GPUs. The algorithms exploit the high computing power of the GPUs and provide fast performance of the algorithms. Our implementation can compute the conditional or posterior probabilities in about 14 microseconds when done in parallel using the Parzen window method. This brings the Parzen window algorithms into the realm of real-time pattern recognition techniques. Similarly, the GPU ANN can perform an epoch of batch training with 56K samples in about 200 milliseconds. This makes frequent retraining possible during an application. The computational ease of such algorithms on an inexpensive hardware makes it possible to acquire labelled samples live and use them to adjust the behaviour of the system.

REFERENCES

[1] Blythe D. The Direct3D 10 System. Microsoft Corporation, 2006.
[2] BrookGPU http://graphics.stanford.edu/projects/brookgpu/
[3] Cao F., Tung A. K. H. and Zhou A. *Scalable Clustering Using Graphics Processors*. Lecture Notes in Computer Science, vol. 4016/2006, pp. 372-384, 2006.
[4] Davis C. E. *Graphics Processing Unit Computation of Neural Networks*. University of New Mexico, 2001.
[5] Duda R. O., Hart P. E. and Stork D. G. *Pattern Classification*. 2nd edition, John Wiley and Sons, 2001.
[6] Fatahlian K., Sugerman J., and Hanrahan P. *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*. Proc of ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, 2004.
[7] Govindaraju N. K., Manocha D., Raghuvanshi N., Tuft D. *Gpusort: High performance sorting using graphics processors*. http://gamma.cs.unc.edu/GPUSORT
[8] Hall J. D., Carr N., and Hart J. *Cache and bandwidth aware matrix multiplication on the GPU*. Technical Report UIUCDCS-R-2003-2328, University of Illinois at Urbana-Champaign.
[9] Haykin S. Neural Networks: A Comprehensive Foundation, 2nd edition, 1998.
[10] Larsen E. S., and McAllister D. *Fast matrix multiplies using graphics hardware*. Proc. of ACM-IEEE conference on Supercomputing, 55-55. 2001.
[11] Moreland K. and Angel E. *The FFT on a GPU*. Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 112119, July 2003.
[12] Moravanszky A. *Dense matrix algebra on the GPU*. 2003.
[13] NVIDIA: NVIDIA CUDA compute unified device architecture programming guide, 2007. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.
[14] NVIDIA: NVIDIA CUBLAS Library, 2007. http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf.
[15] Parzen E. *On Estimation of a Probability Density Function and Mode*. Annal of Mathematical Statistics. Vol. 33, pp. 1065-1076, 1962.
[16] Reiter D. H., Houston M. and Hanrahan P. *ClawHMMER: A Streaming HMMer-Search Implementation*. Proc. of ACM/IEEE conference on Supercomputing, 2005.