PARALLELIZE NEURAL NETWORK TRAINING WITH GPU COMPUTING

A Project

Presented to the faculty of the Department of Computer Science
California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Snehal Sakhare

FALL
2015
PARALLELIZE NEURAL NETWORK TRAINING WITH GPU COMPUTING

A Project

by

Snehal Sakhare

Approved by:

__________________________, Committee Chair
Dr. Pinar Muyan-Ozcelik

__________________________, Second Reader
Dr. V. Scott Gordon

__________________________
Date
Student: Snehal Sakhare

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

______________________________, Graduate Coordinator
Dr. Jinsong Ouyang

______________________________
Date

Department of Computer Science
Abstract

of

PARALLELIZE NEURAL NETWORK TRAINING WITH GPU COMPUTING

by

Snehal Sakhare

Neural Networks (NNs) have been broadly used for many applications like Speech Recognition, Image Recognition, and character Recognition. However, the investment of larger training time and curse of dimensionality limits the usability of NN. There are alternative training algorithms for NNs that are parallel in nature. GPU computing is suitable to the problem domains, which are parallel in nature. Thus, this project is aimed to reduce training time of NNs by exploiting the parallelism in the training algorithms using GPU. Particle Swarm Optimization (PSO) is a NN training algorithm that involves minimal dependencies, which is suitable for parallelization. In this project, I have investigated the parallelization of NN training using PSO technique on the GPU utilizing CUDA 7.0 toolkit [1]. The NN we use in the study classifies the Fisher’s Iris dataset [2]. This is a 3-class classification problem based on four features (sepal length, sepal width, petal length, petal width) of the flowers as input. I could successfully implemented part of the PSO algorithm on the GPU. Then I compared the GPU implementation of PSO written in CUDA to the CPU implementation written in C++. In the current version of GPU implementation, a very
small computation, where we update the all particle’s velocity and position, happens on the GPU and most of the execution takes place on the CPU. As a result, even after increasing the number of particles the GPU implementation is not faster than the CPU implementation. However in future this code can be completely parallelized to leverage the full power of the GPU. In addition, we are looking forward to use a different NN structure with larger particles and model a complete parallel version of it.

_______________________, Committee Chair
Dr. Pinar Muyan-Ozcelik

_______________________
Date
ACKNOWLEDGEMENTS

It is with immense gratitude that I acknowledge the support and help of my professor, Dr. Pinar Muyan-Ozcelik, who has always encouraged me into this research. Without her continuous guidance and persistent help, this project would not have been a success to me. I want to thank Dr. V. Scott Gordon as my second reader for his contribution in completion of this project. I am grateful to the California State University, Sacramento and department of Computer Science without whom this project would not have been an achievement.

I also thank my family and friends, for their endless love and support throughout my life.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Equations</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xiii</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>Chapter 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2. BACKGROUND OF THE STUDY</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Artificial Neural Networks</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1. Multilayer Perceptron</td>
<td>4</td>
</tr>
<tr>
<td>2.2. Backpropagation Algorithm</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Particle Swarm Optimization</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 Particle Swarm Optimization Algorithm</td>
<td>9</td>
</tr>
<tr>
<td>2.4 GPU Computing</td>
<td>10</td>
</tr>
<tr>
<td>2.4.1 CUDA Programming Model</td>
<td>11</td>
</tr>
<tr>
<td>Tables</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Table 1: Accuracy CPU vs GPU</td>
<td>25</td>
</tr>
<tr>
<td>Table: 2 Completion Time CPU vs GPU</td>
<td>25</td>
</tr>
</tbody>
</table>
## LIST OF EQUATIONS

<table>
<thead>
<tr>
<th>Equations</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equation 1: Compute Particle Velocity Vector</td>
<td>8</td>
</tr>
<tr>
<td>Equation 2: Compute Particle Position Vector</td>
<td>9</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1: Natural neurons</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2: MLP Artificial Neural Network</td>
<td>4</td>
</tr>
<tr>
<td>Figure 3: Backpropagation Activation Function</td>
<td>7</td>
</tr>
<tr>
<td>Figure 4: CPU versus GPU source [7]</td>
<td>11</td>
</tr>
<tr>
<td>Figure 5: Grid of Thread Blocks source [9]</td>
<td>12</td>
</tr>
<tr>
<td>Listings</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>Listing 1 : CPU PSO Train method</td>
<td>18</td>
</tr>
<tr>
<td>Listing 2 : GPU PSO Train method</td>
<td>19</td>
</tr>
<tr>
<td>Listing 3: Input Parameters</td>
<td>19</td>
</tr>
<tr>
<td>Listing 4: CUDA Memory Allocation</td>
<td>21</td>
</tr>
<tr>
<td>Listing 5: CUDA Memory copy</td>
<td>21</td>
</tr>
<tr>
<td>Listing 6: Kernel to Update Particle’s velocity and position</td>
<td>22</td>
</tr>
<tr>
<td>Listing 7: Invoke kernelUpdateParticle Kernel</td>
<td>22</td>
</tr>
<tr>
<td>Listing 8: Overall Structure of PSO CUDA method</td>
<td>23</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Neural networks are complex functions, which process numeric data and output weights, and bias values. These set of weights and bias values can then determine the output on unseen data. [3] There are many Artificial Neural Network (ANN) libraries like Fast Artificial Neural Network Library (FANN), Open ANN or ANNs can be implemented programmatically using C# or C++.

Backpropagation (BP) is a very popular approach of training a neural network, however it tends to get slower and sometimes requires thousands of epochs. [4] Backpropagation neural networks could be parallelized [5] but it does not exhibit complete parallelism. PSO is an evolutionary learning approach somewhat similar to genetic algorithms. In PSO, particles update their values at once unlike BP technique. Each particle obtains a best solution computed so far and eventually obtains global values, which is considered best values. PSO over comes the curse of dimensionality problems, especially for NN training.

In this project, I implemented PSO algorithm on the CPU and the GPU and compare their results. I demonstrated neural network predictor to predict the species (setosa, versicolor or virginica) based on flowers’ characteristics using Fisher’s iris data set [2].
This data set includes measurements of type, petal width (PW), petal length (PL), sepal width (SW), and sepal length (SL) for a sample of 150 irises with their species, where species- setosa, versicolor, virginica are correspondingly encoded as 001, 010 and 100.

For the CPU implementation, I referred to the sample code for Training NN using PSO written in C#, provided by Visual Studio Magazine [3]. I converted this piece of code written C# to C++ to be able to integrate it with CUDA implementation of PSO. In addition, to make this code work on GPU I needed to flatten the matrix structure into single array. For GPU implementation, I referred to the study on PSO-GPU implementation for Image segmentation co-authored by Kristiadi, et. al. using CUDA [6]. In their experiment, they used GPU implementation to do image segmentation, whereas I applied PSO-GPU implementation to do NN training. I have also implemented GPU compatible program using CUDA programming language [7] and ran it on NVidia K620 GPU.
Chapter 2

BACKGROUND OF THE STUDY

2.1 Artificial Neural Networks

Artificial neural networks process is inspired from the human brain’s Central Nervous System, and thus brings a certain level of intelligence on silicon. Natural neurons receive electric signals from synapses, which is a small gap between dendrites of the neuron. When signals received are strong enough to activate neurons, those neurons emit signal through axon. This signal then travels in the same fashion to the next neuron. The complexity of a real neuron is highly abstracted in modelling artificial neurons. Neurons work in unison to solve a specific problem. It is an adaptive system, which changes its structure according to the information. This information then flows through the system. A neural network is trained with a set of training data so that a desired set of inputs produces a desired set of outputs.

![Figure 1: Natural neurons](image)
2.1.1. Multilayer Perceptron

Neural networks typically consist of nodes, which are connected to each other by links forming layers. Each link has some weight associated with it. An input layer receives a trigger, which is shifted to one or more hidden layer, and finally to output layer as output is calculated. Multilayer Perceptron (MLP) is the most popular kind of feed-forward neural network, meaning that signals are propagated in forward direction. Thus, the computation takes place layer-by-layer basis. The weights of the nodes are calculated using ‘activation function’. NNs need some 'learning rule’, which modifies the weights of the connections according to the input patterns they are presented with.

Figure 2: MLP Artificial Neural Network
Figure 2 shows diagram of 4-6-3 MLP artificial neural network topology. It consists of three main layers:

I. *Input Layer:* This example has 4 nodes which accepts numerical input to the system. Input is multiplied by interconnection weight and is passed to the next layer.

II. *One or more hidden layers:* Computation of the system is performed in hidden layers. Weights are adjusted and are passed to the next layer for computation.

III. *Output Layer:* Output of the system is calculated in the output layer.

IV. *Weights:* Strength of the link between the nodes.

Neural networks learn by the set of numeric data samples. These data samples contain input points as well as corresponding output points. If the data is not numeric then it can be converted into appropriate number representation. A part of the data set is fed into the network and a smaller part of the data set is kept aside for testing purpose. The network adjusts weights within neurons with the help of training algorithms to get the desired output. Multiple iterations of training algorithms are required to achieve the desired output. A network can be considered as trained once we achieve reasonable accuracy. We then know the most appropriate set of weights of the network. Once the network is trained, it is verified against the test data. Then accuracy is determined against the real output, thus we know the network was successful or not. During testing, the network uses values of weights already calculated in training. So having good training and testing accuracy we can use the network to solve a general problem.
Neural network training is a very skillful process, which involves adjusting the weights of neurons to produce the desired set of outputs. There are various training algorithms available to train artificial neural networks. One of the most popular training algorithms available is Backpropagation algorithm.

2.2. Backpropagation Algorithm

Backpropagation is one of the most common and popular methods of training feed forward artificial neural networks. MLPs are organized in layers and they send their signals in the forward direction, and then the errors are propagated backwards. The backpropagation training method uses supervised training, which means that we provide each input with a target vector signifying the desired output. Then the error is calculated, which is the difference between actual output and desired output. The intention behind iterating through this procedure multiple times is to reduce this error, and hence this error is fed through the network. The training begins with random weights and the goal is to bring the outputs closer to the expected output.

In the forward pass of the algorithm, neurons from the hidden layer propagates values of its weights to the output layer. Values for delta are propagated backward from the output layer to the hidden layer during the reverse pass of the algorithm. The sigmoid activation function of the MLPs implementing backpropagation algorithm, is to keep output within 0 and 1 threshold, is shown in Figure 3.
The algorithm for a three-layered network can be summarized as following:

I. Initialize all weights to small random values.
II. Repeat
III. For each training example do-
IV. Forward propagate the input features of the example to determine the MLP’s outputs.
V. Back propagate the error to generate $\Delta w_{ij}$ for all weights $w_{ij}$.
VI. Update the weights using $\Delta w_{ij}$.
VII. End for
VIII. Until stopping criteria reached.

The main advantage of using the Backpropagation algorithm is its ability to solve complex problems when trained properly. The limitation of the approach is that learning is not guaranteed and convergence obtained can be very slow.
2.3 Particle Swarm Optimization

Particle swarm optimization (PSO) was developed in 1995 by Kennedy and Eberhart as a population based stochastic optimization technique. It utilizes a population of particles also known as “swarm”, which will be flown through a huge space of possible solutions with given velocities towards more optimum solutions. The PSO search method is often linked to the movement in organisms of bird flock or fish school [8]. At each iteration, the velocities of the individual particles are stochastically adjusted according to the previous best position for each particle itself and the swarm’s best position. Both the particle best and the swarm’s best are derived according to a given fitness function.

In PSO, we deal with two vectors for each particle: position vector and velocity vector. In the case of a neural network, a particle's position vector represents the values for the network's weights and biases. The goal is to determine the position/weights vector so that the network generates computed outputs that match the outputs of the training data. Each particle’s movement is determined by its current velocity, personal best-known position, and best-known position in the entire swarm. Hence, for particle i in dimension d, its movement is calculated using the equation shown in Equation 1.

\[
V_{id} = w * v_{id} + c_1 * r_1 * (p_{id} - x_{id}) + c_2 * r_2 * (p_{gd} - x_{id})
\]

Equation 1: Compute particle velocity vector
Velocity from Equation 1 is used to determine the new position of a particle. The particle’s new position can be calculated using the equation shown in Equation 2.

\[ x_{id} = x_{id} + v_{id} \]

Equation 2: Compute particle position vector

Where,

- \( W \) is constant called the inertia,
- \( C_1 \) and \( C_2 \) are nonnegative constant,
- \( r_1 \) and \( r_2 \) are random number between 0 and 1,
- \( p_{id} \) is personal best known position,
- \( p_{gd} \) is best known position in the entire swarm, and
- \( x_{id} \) is current position of particle.

2.3.1 Particle Swarm Optimization Algorithm

1: procedure PSO
2: Initialize the population
3: repeat
4: for \( i = 1 \) to number of individuals do
5: \( G(x_i) \). \( G() \) evaluates goodness
6: end for
7: for i = 1 to number of individuals do
8: P(\sim x_i , \theta) . Modify each individual using parameters \theta
9: end for
10: until stopping criteria

2.4 GPU Computing

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVidia. CUDA enables to run programs on graphic processing units (GPUs) [1]. GPU NVidia series G80 was first to bring together CUDA architecture in the year of 2007. Introduction of GPU Computing now enables GPUs used for applications beyond graphics and provide speedup for parallel algorithms like PSO.

GPU Computing is parallel computing with multiple processor architectures, which is also called heterogeneous computing [6]. Explaining more, GPU-accelerated applications use high-level languages to run the sequential part of their workload on the CPU, which is optimized for single-threaded performance and accelerating parallel processing on the GPU. In GPU computing, many-core GPUs are combined with multicore CPUs to achieve higher performance. The following diagram shows CPU multiple cores and GPU many cores, and this is taken from ‘NVidia Cuda C Programming guide’ [9].
2.4.1 CUDA Programming Model

As mentioned earlier, there are thousands of cores present inside GPU, and CUDA programming provides access to each of the Execution Units of GPU. To manage a bigger task, CUDA splits work into many smaller tasks that will be controlled by the smallest processor unit called thread [9]. Each thread uses its index to access elements in array so that the collection of all threads cooperatively processes the entire data set. Threads are grouped into blocks, and blocks are grouped into a grid [9]. Each block has a unique index in the grid. We can specify the number of threads per block and the number of blocks per grid in kernel call. Its syntax is specified as - \texttt{<<<...>>>}. Figure 5 is taken from ‘NVidia Cuda C Programming guide’ [9] and it shows an example of the two-dimensional blocks within a grid.

Figure 4: CPU versus GPU source [7]
2.5 Supervised Learning using Iris Data

We are implementing the Supervised Learning approach in order to train NN to predict species of flower using Fishers Iris Data [2]. Supervised Learning is one of the machine learning techniques, where the training dataset is labelled or classified with correct results. The system calculates the set of neural network weights and bias values from the input, and uses them to predict the output of the unseen cases. I am applying the Supervised Learning technique to the training of the Iris Data [2].
Chapter 3

RELATED WORK

The most important factor when training an NN is the size or structure of the ANN [5]. It takes experience to come up with a suitable ANN structure in order to solve a particular problem. Thus, I decided to use one of the use cases along with its NN structure that has been proposed in related studies [3, 6, 10, 11].

Secondly, I explored the NN library to implement CPU code preferably written in C or C++. We considered open source libraries, such as FANN and Open NN, to solve the two-spiral problem as our pilot problem. FANN library supports several different training algorithms, and the default algorithm might not always be the best suited for a specific problem [12]. However, we could not determine the best structure using FANN to solve the two-spiral problem. Previous research shows that the two-spiral problem can become significantly challenging to solve using NN [10, 13]. Open NN on the other hand, has issues to compile on the .Net environment that we are using.

Visual Studio magazine article states that, PSO can be used to train a neural network and presents C# source code for the demo program [3]. In addition, it is very easy to translate this code into any of high-level language like C++. I converted the code to C++ to integrate it with the CUDA code, which has the GPU implementation of PSO. This article solved the problem of providing a use case with its suitable NN structure.
Kristiadi et al. used GPU implementation of PSO to do image segmentation. I am using it in NN training, which is a different domain. Their paper [6] showed confidence that PSO runs 170% faster when it used Graphic Processing Unit (GPU) in parallel mode versus when used CPU alone. This speed is growing as the number of particles gets higher. [6] This performance boost will get very promising when we use recently introduced powerful hardware (i.e. GPUs). Therefore, we decided to evaluate the performance boost of predicting species from Fisher’s iris data, which is simpler than Image segmentation.
Chapter 4

PARTICLE SWARM OPTIMIZATION IMPLEMENTATION

To implement PSO we are using 30 item, which is a small subset of the 150 items of famous Fisher’ Iris data [2]. As mentioned earlier, we are using 80% of this subset for training, and remaining 20% of the subset is used for testing. The Appendix-A shows the encoded training and testing dataset.

4.1 Neural Network Structure

The neural network structure is shown in Figure 2 of chapter -2. There are 4 nodes in the input layer, 6 nodes in the hidden layer, and 3 nodes in the output layer of the neural network. All of these nodes are instantiated. The following are values for the common parameters of NN.

- **Input Layer:** 1 layer with 4 nodes. Each node represents flower’s sepal length, sepal width, petal length, petal width.
- **Hidden Layer:** 1 layer with 6 nodes.
- **Output Layer:** 1 layer with 3 nodes representing species of flower- setosa, versicolor, virginica correspondingly encoded as (001), (010) and (100).
- **Activation Function:** TANH and SOFTMAX
- **Bias used:** Yes.
- **Learning method:** Particle Swarm Optimization
Error Rate: 0.060. This is the acceptable amount of error measured as Mean Squared Error after training.

Probability death: 0.005

Maximum Iterations/epochs: 700

A fully connected 4-6-3 neural network will have weights and bias values-

\[(4 \times 6) + (6 \times 3) + (6 + 3) = 51.\]

This program uses 12 particles in the set of particles, which also called ‘swarm’. The swarm is iterated 700 times and every time, it tries to find the set of weights and bias values with the minimum NN training error. Every iteration Swarm remembers the global best, which is the best set of values known so far within the swarm and particle best. After the PSO training has finished, the vector containing 51 weight and bias values of the ultimate best are displayed. NN uses this weight and bias vector to evaluate accuracy on six test items (20% of 30 item set). The network can correctly classify 4 items out of 6 items which means the network have accuracy= 0.6777 on test data.

4.2 Particle Swarm Optimization

PSO is an iterative process where the collection of particles change their position with some velocity to represent a minimum error solution. Here NN training is happening using PSO, where the particle’s movement is updated using Equation 2 and is dependent upon three factors- particle’s current velocity, its best-known position and, global best-known position. A particle is combined of three vectors- positions vector, velocity vector
and pBest vector; which stores the particle’s best position. General algorithm implementation is outlined below:

1. Initialize N particles in the swarm randomly, set pBest (personal best position of particle P) and gBest (swarm’s overall best position) with particles position.

2. For iteration = 1 to NUM_OF_PARTICLES do
   a. For each particle p do:
      i. Update velocity vector of p, having size = num_of_particels * num_weights
         Update position vector of p, having size = num_of_particels * num_weights
      ii. return position vector of p
   b. For each particle p do:
      i. If MeanSquaredError(p.position) < MeanSquaredError(p.pBest)
         copy (p.pBest, p.position)
      ii. If MeanSquaredError(p.position) < MeanSquaredError(p.pBest)
         copy (gBest, p.pBest)

Therefore, gBest contains the optimal solution of the NN training weight and bias values after some iterations.
4.2.1 Parallel PSO implementation

Here, we discuss more on leveraging parallelism present in PSO implementation. There is no dependency existing between particles while computing individual velocity and position vector. Therefore, we can indeed process all the particles in parallel, as one particle does not have to wait for another particle’s computation to finish. In this project, velocity and position computations are happening in parallel on GPU/device, whereas pBest and gBest are determined serially using CPU/host. However, as I will further explain in the future work section, these computations can also be done on GPU to gain more speedup.

4.3 Code Implementation

This section will list programming details specific to the NN training, its helping functions, and PSO-GPU code. Many of these methods are commonly used to implement any feed-forward neural network. The usual methods required to operate NN are SetWeights, GetWeights, and ComputeOutputs. The pso_train function is used to train the NN on a CPU using the PSO technique with its helper function -MeanSquaredError. The signature of the pso_train method is shown in Listing 1.

```
void pso_train (float *positions, float *velocities,
            float *pBests, float *gBest);
```

Listing 1 : CPU PSO Train method
Similarly, the cuda_pso function uses MeanSquaredError as a helper function to train the NN using the PSO. Additionally, the cuda_pso function performs extra tasks of memory allocation, data copying and data freeing on a GPU. The signature of the cuda_train method is shown in Listing 2.

```c
extern "C" void cuda_pso(float *positions, float *velocities, float *pBests, float *gBest)
```

**Listing 2: GPU PSO Train method**

```c
#define Num_Of_Particles 12
#define Num_Of_Dimensions 51
#define Start_Range_Min -5.12f
#define Start_Range_Max 5.12f
#define Inf 9999.0
#define Max_Iter 700
#define Epsilon 0.0005
#define Phi 3.1415
#define Omega 0.5
#define c1 1.49
#define c2 1.49
#define numInput 4
#define numHidden 6
#define numOutput 3
#define exitError 0.060
#define probDeath 0.005
```

**Listing 3: Input Parameters**

Listing 3 shows the parameters- maxEpochs and exitError, to control the iterative PSO process and its termination. Input parameter probDeath controls particle’s random
death. The following parameters are defined and listed in the my_kernel.h file so that the CPU and GPU code can access them at single place.

Initialization is the first step before calling CPU PSO and GPU PSO training methods. During initialization, pBest is set to a random position and velocity is set to zero. Each iteration CPU code and GPU code computes pBest and gBest with the help of MeanSquaredError method to calculate error associated with the position. Mean squared error (MSE) is calculated as one target output is (0, 1, 0) for versicolor- one of the category of flower we are trying to predict, and the computed output is (0.2, 0.9, 0.1), then 

\[\text{MSE} = (0.2 - 0)^2 + (0.9 - 1)^2 + (0.1 - 0)^2 = 0.04 + 0.01 + 0.01 = 0.06.\]

MSE is the average of all errors that occurs in the training data.

4.3.1 GPU Kernels Implementation

As memory allocation takes place on host, GPU needs its own memory to store data and work on the stored data. Just like malloc function in C, CUDA runtime provides \texttt{cudaMalloc} for dynamic allocation of memory on device. Listing 4 shows that we have allocated memory for position, velocity, pBest and gBest vectors on device.
To copy data between host and device CUDA has provided `cudaMemcpy`, which is very similar to `memcpy` in C. We copy data from host to device before invoking kernel and after kernel finishes computation copying back from device to host memory. Flag value (e.g. `cudaMemcpyHostToDevice` copies data from host memory to device memory) determines the direction of flow. Listing 5 shows particle data is initialized by copying data from host to device variables created and allocated memory earlier.

```c
// Memory allocation
    cudaMemcpy((void**)&devPos, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS);
    cudaMemcpy((void**)&devVel, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS);
    cudaMemcpy((void**)&devPBest, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS);
    cudaMemcpy((void**)&devGBest, sizeof(float) * NUM_OF_DIMENSIONS);

// Copy particle data from host to device
    cudaMemcpy(devPos, positions, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);
    cudaMemcpy(devVel, velocities, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);
    cudaMemcpy(devPBest, pBests, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);
    cudaMemcpy(devGBest, gBest, sizeof(float) * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);
```

Listing 4: CUDA Memory Allocation

Listing 5: CUDA Memory copy
The heart of the CPU code is the actual kernel implementation. Each thread is represented by thread index ‘i’ and is executed in parallel. Listing 6 provides details about kernel implementation to update position and velocity of particles.

Listing 6: Kernel to Update Particle’s velocity and position

As explained earlier, position and velocity are computed in parallel fashion after that pBest and gBest are determined in serial fashion like on CPU.

Listing 7: Invoke kernelUpdateParticle Kernel

Listing 7 shows how the kernelUpdateParticle kernel is invoked. Method getRandomClamped() is used in the parameters list to get random number.
ranging from 0.0f to 1.0f inclusive. Listing 8 shows the overall arrangement of GPU PSO method and describes when exactly kernel is invoked.

```c
// Initialize PSO particle on CPU
// Allocate GPU Memory
// Copy particle’s data from CPU to GPU
// PSO main function begin
    for (int iter = 0; iter < MAX_ITER; iter++)
    {
        // 1. Update position and velocity on GPU
        kernelUpdateParticle << <blocksNum, threadsNum >> >(devPos,
        devVel, devPBest, devGBest, 0.5f, 0.5f);

        // Copy pBest, position, velocity from GPU
        // 2. Find pBest and gBest serially
        // For calculating error construct NN using each particle
        // position as set of weights

        // Construct NN with new random values of particle
        // position as set of weight and bias values

        cudaMemcpy(devPBest, pBests, sizeof(float) * size,
                    cudaMemcpyHostToDevice);
        cudaMemcpy(devGBest, gBest, sizeof(float) * 
                    NUM_OF_DIMENSIONS,
                    cudaMemcpyHostToDevice);

        // cleanup memory
    } // PSO main function end
```

Listing 8: Overall Structure of PSO CUDA method
Chapter 5

RESULTS

I have implemented the small portion of the PSO to be able to run on the GPU. NN construction, data initialization, and rest of PSO algorithm are still executed serially in the CUDA environment. This project has shown that there exists a faster training approach for the neural network using Particle Swarm Optimization. In addition, PSO training algorithm gives satisfactory accuracy after 700 epochs.

Table 1 explains about accuracy seen on training data set and testing data set. The numbers are generated on both CPU and GPU. We see that the CPU accuracy is better than the GPU accuracy because the randomness introduced in CPU version is different from GPU version. The more the randomness better the accuracy. In the CPU version every iteration, each particle uses a fresh pair of random value for velocity computation whereas, in the GPU version a pair of random values is passed and all particle use same pair of random value for velocity computation. Thus, the accuracy on GPU is hampered. We can use NVIDIA CUDA Random Number Generation library (cuRAND) to generate random values for each thread running on GPU however, I could not explore more because of the time constrain.
Table 1: Accuracy CPU vs GPU

<table>
<thead>
<tr>
<th>Number of Particles</th>
<th>Training Accuracy</th>
<th>Testing Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>12</td>
<td>0.8333</td>
<td>0.6667</td>
</tr>
<tr>
<td>20</td>
<td>0.6667</td>
<td>0.6667</td>
</tr>
<tr>
<td>50</td>
<td>0.6667</td>
<td>0.6667</td>
</tr>
<tr>
<td>100</td>
<td>0.9167</td>
<td>0.6667</td>
</tr>
</tbody>
</table>

Table 2 explains about completion timing seen on CPU and GPU. We expect that the speedup provided by the GPU increases as we increase the number of particles. However, we have parallelized a very small portion of the PSO and most of the code is executed serially. GPU PSO needs to perform addition tasks of memory allocation and data copying, so the completion time taken by the CPU is less than the GPU completion time.

Table 2: Completion Time CPU vs GPU

<table>
<thead>
<tr>
<th>Number of Particles</th>
<th>Completion Time (ms) with randomness</th>
<th>Completion Time (ms) without randomness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>12</td>
<td>1.263</td>
<td>1.690</td>
</tr>
</tbody>
</table>

I removed the randomness involved in velocity calculation and substituted with constant factor in order to verify computations values on CPU and GPU. I observed that the computation all the values were matching and showed exact accuracy = 0.6667 (for 700 epochs and 12 particles) on the CPU and the GPU. Table 3 describes the GPU takes more completion time than the CPU.
Chapter 6

FUTURE WORK

This project only considers small dataset and limited computation are performed on GPU while training the neural network. There can be efforts put to come up with more optimized and parallel version of PSO. Because of time restrictions, I could not fully parallelize the PSO implementation. There could be a one more kernel for pBest and gBest calculations.

Similarly, the PSO implementation can be tried by using different training use case for larger dataset then GPU can really outperform CPU. If we were to train the network, which requires more than 100 PSO particles would make a good test case [6]. It is observed that, if we use less number of the particles, GPU implementation shows poor speedup than CPU implementation. To increase accuracy of PSO on the GPU, we can use NVIDIA CUDA Random Number Generation library (cuRAND) to generate random values for each thread running on GPU.

Explaining further GPU copies data from CPU into local memory to process locally, this time is constant but additional overhead than CPU and it is counted towards GPUs total time. For the small number of particles, the overtime is more on GPU that the CPU implementation of complete processing time on CPU / GPU. As particles gets larger
in number CPU time times gets worse because CPU process each particle sequentially. On the contrary, GPU takes constants time as the overhead remains constant.

Explaining further, the overhead time is likely not affected by the number of the particles in parallel mode. It gives us clear insight that the bigger the particles number, higher the speed up that the parallel processing (GPU implementation) gain over the sequential processing (CPU only implementation). [6]
APPENDIX A

Fisher’s Iris Data Set

The training data is:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.3</td>
<td>2.9</td>
<td>5.6</td>
<td>1.8</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>6.9</td>
<td>3.1</td>
<td>4.9</td>
<td>1.5</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>4.6</td>
<td>3.4</td>
<td>1.4</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>7.2</td>
<td>3.6</td>
<td>6.1</td>
<td>2.5</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>7.6</td>
<td>3.0</td>
<td>6.6</td>
<td>2.1</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>4.9</td>
<td>2.4</td>
<td>3.3</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>9</td>
<td>4.9</td>
<td>3.1</td>
<td>1.5</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>11</td>
<td>6.4</td>
<td>3.2</td>
<td>4.5</td>
<td>1.5</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>12</td>
<td>4.4</td>
<td>2.9</td>
<td>1.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>13</td>
<td>5.8</td>
<td>2.7</td>
<td>5.1</td>
<td>1.9</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>14</td>
<td>6.3</td>
<td>3.3</td>
<td>6.0</td>
<td>2.5</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>15</td>
<td>5.2</td>
<td>2.7</td>
<td>3.9</td>
<td>1.4</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>16</td>
<td>7.0</td>
<td>3.2</td>
<td>4.7</td>
<td>1.4</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>17:</td>
<td>6.5</td>
<td>2.8</td>
<td>4.6</td>
<td>1.5</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>18:</td>
<td>4.9</td>
<td>2.5</td>
<td>4.5</td>
<td>1.7</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>19:</td>
<td>5.7</td>
<td>2.8</td>
<td>4.5</td>
<td>1.3</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>20:</td>
<td>5.0</td>
<td>3.4</td>
<td>1.5</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>21:</td>
<td>6.5</td>
<td>3.0</td>
<td>5.8</td>
<td>2.2</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>22:</td>
<td>5.5</td>
<td>2.3</td>
<td>4.0</td>
<td>1.3</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>23:</td>
<td>6.7</td>
<td>2.5</td>
<td>5.8</td>
<td>1.8</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

The test data is:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1:</td>
<td>7.1</td>
<td>3.0</td>
<td>5.9</td>
<td>2.1</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2:</td>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3:</td>
<td>6.3</td>
<td>3.3</td>
<td>4.7</td>
<td>1.6</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>4:</td>
<td>6.6</td>
<td>2.9</td>
<td>4.6</td>
<td>1.3</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>5:</td>
<td>7.3</td>
<td>2.9</td>
<td>6.3</td>
<td>1.8</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
APPENDIX B

Source code

Kernel.cu

#include <cuda_runtime.h>
#include <cuda.h>
#include <math_functions.h>
#include "my_kernel.h"

float tempParticle1[NUM_OF_DIMENSIONS];
float tempParticle2[NUM_OF_DIMENSIONS];

__global__ void kernelUpdateParticle(float *positions, float *velocities, float *pBests, float *gBest, float r1, float r2)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i >= NUM_OF_PARTICLES * NUM_OF_DIMENSIONS)
        return;

    float rp = r1;
    float rg = r2;

    velocities[i] = OMEGA * velocities[i] + c1 * rp * (pBests[i] - positions[i]) + c2 * rg * (gBest[i % NUM_OF_DIMENSIONS] - positions[i]);

    // Update posisi particle
    positions[i] += velocities[i];
}

extern "C" void cuda_pso(float *positions, float *velocities, float *pBests, float *gBest)
{
    int size = NUM_OF_PARTICLES * NUM_OF_DIMENSIONS;

    float *devPos;
    float *devVel;
    float *devPBest;
    float *devGBest;

    // Memory allocation
    cudaMalloc((void**)&devPos, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS);
    cudaMalloc((void**)&devVel, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS);
    cudaMalloc((void**)&devPBest, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS);
cudaMalloc((void**)&devGBest, sizeof(float) * NUM_OF_DIMENSIONS);

// Thread & Block number
int threadsNum = 32;
int blocksNum = (NUM_OF_PARTICLES* NUM_OF_DIMENSIONS / threadsNum) + 1;

// Copy particle datas from host to device
 cudaMemcpy(devPos, positions, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);
 cudaMemcpy(devVel, velocities, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);
 cudaMemcpy(devPBest, pBests, sizeof(float) * NUM_OF_PARTICLES * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);
 cudaMemcpy(devGBest, gBest, sizeof(float) * NUM_OF_DIMENSIONS, cudaMemcpyHostToDevice);

// PSO main function
for (int iter = 0; iter < MAX_ITER; iter++)
{

    // Update position and velocity
    kernelUpdateParticle << <blocksNum, threadsNum >> >(devPos, devVel, devPBest, devGBest, getRandomClamped(), getRandomClamped());

    // Update pBest
    cudaMemcpy(positions, devPos, sizeof(float) * size, cudaMemcpyDeviceToHost);
    cudaMemcpy(velocities, devVel, sizeof(float) * size, cudaMemcpyDeviceToHost);
    cudaMemcpy(pBests, devPBest, sizeof(float) * size, cudaMemcpyDeviceToHost);

    for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i +=

        NUM_OF_DIMENSIONS)
        {
            for (int j = 0; j < NUM_OF_DIMENSIONS; j++)
            {
                tempParticle1[j] = positions[i + j];
                tempParticle2[j] = pBests[i + j];
            }

            if (MeanSquaredError(tempParticle1, NUM_OF_DIMENSIONS) < MeanSquaredError(tempParticle2, NUM_OF_DIMENSIONS))
            {
                for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
                    pBests[k] = positions[i + k];

                if (MeanSquaredError(tempParticle2, NUM_OF_DIMENSIONS) < MeanSquaredError(gBest, NUM_OF_DIMENSIONS))
                {
                    for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
                        gBest[k] = pBests[i + k];
                }
            }
        }
}
// 4. optional: does curr particle die?
double die = getRandomClamped();
if (die < probDeath)
{
    // new position, leave velocity, update error
    for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i++)
    {
        positions[i] = getRandom(START_RANGE_MIN,
        START_RANGE_MAX);
    }

    // global best by chance?
    for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i += NUM_OF_DIMENSIONS)
    {
        for (int j = 0; j < NUM_OF_DIMENSIONS; j++)
        {
            tempParticle1[j] = positions[i + j];
            tempParticle2[j] = pBests[i + j];
        }
        if (MeanSquaredError(tempParticle2,
        NUM_OF_DIMENSIONS) < MeanSquaredError(gBest,
        NUM_OF_DIMENSIONS))
        {
            for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
            gBest[k] = pBests[i + k];
        }
    }
}

} //end of prob death step

cudaMemcpy(devPBest, pBests, sizeof(float) * size,
cudaMemcpyHostToDevice);
cudaMemcpy(devGBest, gBest, sizeof(float) * NUM_OF_DIMENSIONS,
cudaMemcpyHostToDevice);

} // cleanup
cudaFree(devPos);
cudaFree(devVel);
cudaFree(devPBest);
cudaFree(devGBest);
Kernel .cpp

#include <string.h> // memcpy
#include "my_kernel.h"

float* inputs;  // input-hidden
float* ihWeights;
float* hBiases;
float* hOutputs;
float** hoWeights; // hidden-output
float* oBiases;
float* outputs;

void ShowVector(float* vector, int size, int valsPerRow, int decimals, bool newLine)
{
    for (int i = 0; i < size; ++i)
    {
        if (i % valsPerRow == 0) printf("\n");
        printf("%0.3f  ", vector[i]);
    }
    if (newLine == true) printf("\n");
}

static float HyperTanFunction(float x)
{
    if (x < -20.0) return -1.0; // approximation is correct to 30 decimals
    else if (x > 20.0) return 1.0;
    else return tanh(x);
}

static float* Softmax(float* oSums, int size)
{
    // does all output nodes at once so scale doesn't have to be re-computed each time
    // determine max output sum
    float max = oSums[0];
    for (int i = 0; i < size; ++i)
    {
        if (oSums[i] > max)
            max = oSums[i];
    }
    // determine scaling factor -- sum of exp(each val - max)
    double scale = 0.0;
    for (int i = 0; i < size; ++i)
    {
        scale += exp(oSums[i] - max);
    }
    float* result = new float[size];
    for (int i = 0; i < size; ++i)
    {
        result[i] = exp(oSums[i] - max) / scale;
    }
    return result; // now scaled so that xi sum to 1.0
}
void NeuralNetwork()
{
    inputs = new float[numInput];
hBiases = new float[numHidden];
hOutputs = new float[numHidden];
oBiases = new float[numOutput];
outputs = new float[numOutput];

    ihWeights = new float*[numInput];
    for (int r = 0; r < numInput; ++r)
        ihWeights[r] = new float[numHidden];

    hoWeights = new float*[numHidden];
    for (int r = 0; r < numHidden; ++r)
        hoWeights[r] = new float[numOutput];
} // ctor

void SetWeights(float* weights, int size)
{
    NeuralNetwork();
    // copy weights and biases in weights[] array to i-h weights, i-h biases,
    // copy weights and biases in weights[] array to o-h weights, o-h biases
    int numWeights = (numInput * numHidden) + (numHidden * numOutput) +
        numHidden + numOutput;
    //printf("size of weights in setweights = %d", size);
    if (size != numWeights)
        throw ("Bad weights array length: ");

    int k = 0; // points into weights param
    for (int i = 0; i < numInput; ++i)
        for (int j = 0; j < numHidden; ++j)
            ihWeights[i][j] = weights[k++];
    for (int i = 0; i < numHidden; ++i)
        hBiases[i] = weights[k++];
    for (int i = 0; i < numHidden; ++i)
        for (int j = 0; j < numOutput; ++j)
            hoWeights[i][j] = weights[k++];
    for (int i = 0; i < numOutput; ++i)
        oBiases[i] = weights[k++];
}

float* ComputeOutputs(float* xValues, int size)
{
    if (size != numInput)
        throw ("Bad xValues array length");

    float* hSums = new float[numHidden]; // hidden nodes sums scratch array
    float* oSums = new float[numOutput]; // output nodes sums
    for (int i = 0; i < numHidden; ++i) //initialize
        hSums[i] = 0;
    for (int i = 0; i < numOutput; ++i) //initialize
        oSums[i] = 0;
for (int i = 0; i < size; ++i) // copy x-values to inputs
    inputs[i] = xValues[i];

for (int j = 0; j < numHidden; ++j) // compute i-h sum of weights * inputs
    for (int i = 0; i < numInput; ++i)
        hSums[j] += inputs[i] * ihWeights[i][j]; // note +=

for (int i = 0; i < numHidden; ++i) // add biases to input-to-hidden sums
    hSums[i] += hBiases[i];

for (int i = 0; i < numHidden; ++i) // apply activation
    hOutputs[i] = HyperTanFunction(hSums[i]); // hard-coded

for (int j = 0; j < numOutput; ++j) // compute h-o sum of weights * hOutputs
    for (int i = 0; i < numHidden; ++i)
        oSums[j] += hOutputs[i] * hoWeights[i][j];

for (int i = 0; i < numOutput; ++i) // add biases to input-to-hidden sums
    oSums[i] += oBiases[i];

float* softOut = Softmax(oSums, numOutput); // softmax activation does all outputs at once for efficiency
memcpy(outputs, softOut, sizeof(float) * numOutput);

float* retResult = new float[numOutput]; // could define a GetOutputs method instead
// Array.Copy(this->outputs, retResult, sizeof(retResult));
memcpy(retResult, outputs, sizeof(float) * numOutput);
delete hSums;
delete oSums;
return retResult;
} // ComputeOutputs

float MeanSquaredError(float* weights, int size)
{
    SetWeights(weights, size); // copy the weights to evaluate in
    float* xValues = new float[numInput]; // inputs
    float* tValues = new float[numOutput]; // targets
    float sumSquaredError = 0.0;
    int train_size = 24;
    for (int i = 0; i < train_size; ++i) // walk through each training data item
    
        memcpy(xValues, testData[i], sizeof(float)* numInput);
memcpy(tValues, trainData[i] + numInput, sizeof(float)*numOutput);

float* yValues = ComputeOutputs(xValues, numInput); // compute the outputs using centroids, widths, weights, bias values
for (int j = 0; j < numOutput; ++j)
    sumSquaredError += ((yValues[j] - tValues[j]) * (yValues[j] - tValues[j]));
delete xValues;
delete tValues;
return sumSquaredError / train_size;
}

int MaxIndex(float* vector, int size) // helper for Accuracy()
{
    // index of largest value
    int bigIndex = 0;
    float biggestVal = vector[0];
    for (int i = 0; i < size; ++i)
    {
        if (vector[i] > biggestVal)
        {
            biggestVal = vector[i]; bigIndex = i;
        }
    }
    return bigIndex;
}

double Accuracy(const float Data[24][7], int rows, float* gBest)
{
    SetWeights(gBest, NUM_OF_DIMENSIONS);
    // percentage correct using winner-takes all
    int numCorrect = 0;
    int numWrong = 0;
    float* xValues = new float[numInput]; // inputs
    float* tValues = new float[numOutput]; // targets
    float* yValues; // computed Y

    for (int i = 0; i < rows; ++i)
    {
        memcpy(xValues, Data[i], sizeof(float)*numInput);

        memcpy(tValues, Data[i] + numInput, sizeof(float)*numOutput);
        yValues = ComputeOutputs(xValues, numInput);
        int maxIndex = MaxIndex(yValues, numOutput); // which cell in yValues has largest value?

        if (tValues[maxIndex] == 1.0) // ugly. consider AreEqual(double x, double y)
            ++numCorrect;
        else
            ++numWrong;
    }
delete xValues;
delete yValues;
return (numCorrect * 1.0) / (numCorrect + numWrong); // ugly 2 - check for divide by zero
float getRandom(float low, float high) {
    return low + float(((high - low) + 1) * rand() / (RAND_MAX + 1.0));
}

float getRandomClamped() {
    return (float)rand() / (float)RAND_MAX;
}

void pso(float *positions, float *velocities, float *pBests, float *gBest) {
    float tempParticle1[NUM_OF_DIMENSIONS];
    float tempParticle2[NUM_OF_DIMENSIONS];

    // PSO main function
    for (int iter = 0; iter < MAX_ITER; iter++) {

        float rp = getRandomClamped();
        float rg = getRandomClamped();

        // 1. Update velocity and position
        for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i++) {
            velocities[i] = OMEGA * velocities[i] + c1 * rp * (pBests[i] - positions[i]) + c2 * rg * (gBest[i % NUM_OF_DIMENSIONS] - positions[i]);

            // Update posisi particle
            positions[i] += velocities[i];
        }

        // 2. Find pBest and gBest
        for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i += NUM_OF_DIMENSIONS) {
            for (int j = 0; j < NUM_OF_DIMENSIONS; j++) {
                tempParticle1[j] = positions[i + j];
                tempParticle2[j] = pBests[i + j];
            }

            if (MeanSquaredError(tempParticle1, NUM_OF_DIMENSIONS) < MeanSquaredError(tempParticle2, NUM_OF_DIMENSIONS)) {
                for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
                    pBests[k] = positions[i + k];
            }
        }
    }
}
if (MeanSquaredError(tempParticle2, NUM_OF_DIMENSIONS) < MeanSquaredError(gBest, NUM_OF_DIMENSIONS))
{
    for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
        gBest[k] = pBests[i + k];
}

// 4. optional: does curr particle die?
double die = getRandomClamped();
if (die < probDeath)
{
    // new position, leave velocity, update error
    for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i++)
    {
        positions[i] = getRandom(START_RANGE_MIN, START_RANGE_MAX);
    }

    // global best by chance?
    for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i += NUM_OF_DIMENSIONS)
    {
        for (int j = 0; j < NUM_OF_DIMENSIONS; j++)
        {
            tempParticle1[j] = positions[i + j];
            tempParticle2[j] = pBests[i + j];
        }
        if (MeanSquaredError(tempParticle2, NUM_OF_DIMENSIONS) < MeanSquaredError(gBest, NUM_OF_DIMENSIONS))
        {
            for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
                gBest[k] = pBests[i + k];
        }
    }
} //end of prob death step 4
} // maximum iteration or epochs loop
Main.cpp

#include "my_kernel.h"

static void ShowVector(float* vector, int size, int valsPerRow, int decimals, bool newline)
{
    for (int i = 0; i < size; ++i)
    {
        if (i % valsPerRow == 0) printf("\n");
        printf("%0.3f ", vector[i]);
    }
    if (newline == true) printf("\n");
}

static void ShowMatrix(double matrix[][7], int numRows, int decimals, bool newline)
{
    for (int i = 0; i < numRows; ++i)
    {
        printf("\n%d: ", i);
        for (int j = 0; j < 7; ++j)
        {
            if (matrix[i][j] >= 0.0) printf("" );
            else printf("\- ");
        }
        printf("%.3f ", abs((matrix[i][j])));
        printf("\n");
    }
    if (newline == true) printf("\n");
}

int main(int argc, char** argv)
{
    // Particle
    float positions[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    float velocities[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    float pBests[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    float dumy_positions[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    float dumy_velocities[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    float dumy_pBests[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    // gBest
    float gBest[NUM_OF_DIMENSIONS];
    srand((unsigned)time(NULL));

    // Initialize particles
    for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i++)
    {
        positions[i] = getRandom(START_RANGE_MIN, START_RANGE_MAX);
        dumy_pBests[i] = pBests[i] = dumy_positions[i] = positions[i];
        velocities[i] = dumy_velocities[i] = 0;
    }
    printf("\nSetting numParticles = %d ", NUM_OF_PARTICLES);
printf("\nSetting maxEpochs = %d", MAX_ITER);
printf("\nSetting early exit MSE error = %.3f", exitError);
printf("\nSetting probDeath = %.3f", probDeath);

for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
gBest[k] = pBests[k];

clock_t begin = clock();

// GPU PSO main function
cuda_pso(positions, velocities, pBests, gBest);

clock_t end = clock();

printf("\n==================== GPU Training =======================\n");
printf("Time elapsed : %10.3lf ms\n", (double)(end - begin) / CLOCKS_PER_SEC);
printf("\n Best weights >>");
ShowVector(gBest, NUM_OF_DIMENSIONS, 10, 3, true);

printf("Accuracy on training data  = %0.4lf\n", Accuracy(trainData, 24, gBest));

printf(" GPU Testing \n");
printf("Accuracy on Testing data  = %0.4lf\n", Accuracy(testData, 6, gBest));

// ================== END OF GPU =================== //

for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
gBest[k] = dumy_pBests[k];

begin = clock();

// CPU PSO main function
pso(dumy_positions, dumy_velocities, dumy_pBests, gBest);

end = clock();

printf("\n==================== CPU =======================\n");
printf("Time elapsed : %10.3lf ms\n", (double)(end - begin) / CLOCKS_PER_SEC);
printf("\n Best weights >>");
ShowVector(gBest, NUM_OF_DIMENSIONS, 10, 3, true);
printf("Accuracy on training data  = %0.4lf\n", Accuracy(trainData, 24, gBest));

printf(" CPU Testing \n");
printf("Accuracy on Testing data  = %0.4lf\n", Accuracy(testData, 6, gBest));

// ================== END OF CPU =================== //

return 0;
My_kernel.h

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define NUM_OF_PARTICLES 100
#define NUM_OF_DIMENSIONS 51
#define START_RANGE_MIN -5.12f
#define START_RANGE_MAX 5.12f
#define INF 9999.0
#define MAX_ITER 700
#define EPSILON 0.0005
#define phi 3.1415
#define OMEGA 0.5
#define c1 1.49
#define c2 1.49

#define numInput 4
#define numHidden 6
#define numOutput 3
#define exitError 0.060
#define probDeath 0.005

const float trainData[24][7] = { { 6.3, 2.9, 5.6, 1.8, 1, 0, 0 }, { 6.9, 3.1, 4.9, 1.5, 0, 1, 0 }, { 4.6, 3.4, 1.4, 0.3, 0, 0, 1 },
    { 7.2, 3.6, 2.5, 1, 0, 0 }, { 4.7, 3.2, 1.3, 0.2, 0, 0, 1 }, { 4.9, 3, 1.4, 0.2, 0, 0, 1 },
    { 4.9, 2.4, 3.3, 1, 0, 0, 0 }, { 5.4, 3.9, 1.7, 0.4, 0, 0, 1 }, { 4.9, 3.1, 1.5, 0.1, 0, 0, 1 },
    { 6.4, 3.2, 4.5, 1.5, 0, 1, 0 }, { 4.4, 2.9, 1.4, 0.2, 0, 0, 1 }, { 5.8, 2.7, 5.1, 1.9, 1, 0, 0 },
    { 6.3, 3.3, 6, 2.5, 1, 0, 0 }, { 5.2, 2.7, 3.9, 1.4, 0, 1, 0 }, { 7, 3.2, 4.7, 1.4, 0, 1, 0 },
    { 6.5, 2.8, 4.6, 1.5, 0, 1, 0 }, { 4.9, 2.5, 4.5, 1.7, 1, 0, 0 },
    { 5.7, 2.8, 4.5, 1.3, 0, 1, 0 }, { 5, 3.4, 1.5, 0.2, 0, 0, 1 }, { 6.5, 3, 5.8, 2.2, 1, 0, 0 },
    { 5.5, 2.3, 4, 1.3, 0, 1, 0 }, { 6.7, 2.5, 5.8, 1.8, 0, 0 } );

const float testData[6][7] = { { 4.6, 3.1, 1.5, 0.2, 0, 0, 1 }, { 7.1, 3, 5.9, 2.1, 1, 0, 0 }, { 9.1, 3.5, 1.4, 0.2, 0, 0, 1 },
    { 6.3, 3.3, 4.7, 1.6, 0, 1, 0 }, { 6.6, 2.9, 4.6, 1.3, 0, 1, 0 }, { 7.3, 2.9, 6.3, 1.8, 0, 0 } );

float getRandom(float low, float high);
float getRandomClamped();
float MeanSquaredError(float* weights, int size);
float* ComputeOutputs(float* xValues, int size);
double Accuracy(const float Data[24][7], int rows, float* gBest);
void SetWeights(float* weights, int size);
void ShowVector(float* vector, int size, int valsPerRow, int decimals, bool newLine);
int MaxIndex(float* vector, int size);
void NeuralNetwork();
extern "C" void cuda_pso(float* positions, float* velocities, float* pBests, float*gBest);
void pso(float* positions, float* velocities, float* pBests, float*gBest);
Sample Output

C:\Users\SCSAKHAR> "C:\Users\SCSAKHAR\Documents\Visual Studio 2013\Projects\cuda_wiseodd\x64\Debug\my_gpu_pso.exe"

Setting numParticles = 12
Setting maxEpochs = 700
Setting early exit MSE error = 0.060
Setting probDeath = 0.005

== GPU Training ==

Time elapsed : 1.306 ms

Best weights >>
-0.503  5.476  2.812  5.039  2.263  -0.273  -1.969  4.968  -5.016  5.160
1.464  1.445  0.880  -0.509  -0.359  0.458  -1.543  -4.025  -0.280  2.764
4.730  4.156  0.812  -3.925  4.007  -0.941  -3.356  -0.385  -2.537  -3.786
2.214  0.696  4.097  4.220  2.854  4.290  -3.184  0.392  1.979  -2.605
6.045
Accuracy on training data  = 0.6667
GPU Testing
Accuracy on Testing data  = 0.6667

== CPU ==

Time elapsed : 0.743 ms

Best weights >>
-0.503  5.476  2.812  5.039  2.263  -0.273  -1.969  4.968  -5.016  5.160
1.464  1.445  0.880  -0.509  -0.359  0.458  -1.543  -4.025  -0.280  2.764
4.730  4.156  0.812  -3.925  4.007  -0.941  -3.356  -0.385  -2.537  -3.786
2.214  0.696  4.097  4.220  2.854  4.290  -3.184  0.392  1.979  -2.605
6.045
Accuracy on training data  = 0.6667
CPU Testing
Accuracy on Testing data  = 0.6667
Bibliography


