USING PSO ALGORITHM FOR TRAINING NEURAL NETWORKS

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

Divya Banur Shashidhara

SPRING
2016
USING PSO ALGORITHM FOR TRAINING NEURAL NETWORKS

A Project

by

Divya Banur Shashidhara

Approved by:

______________________________, Committee Chair
Dr. Pinar Muyan-Ozcelik

______________________________, Second Reader
Dr. V. Scott Gordon

______________________________
Date
Student: Divya Banur Shashidhara

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

Department of Computer Science
Abstract

of

USING PSO ALGORITHM FOR TRAINING NEURAL NETWORKS

by

Divya Banur Shashidhara

Artificial Neural Networks (ANN) have a wide range of applications in the field of science & technology. An attempt was made previously in a research study to train neural networks using the PSO algorithm. The main objective of this previous study was to predict the species based on flowers characteristics by using GPU Computing and CUDA programming language. However, some limitations were observed in the implementation of PSO on CPU and the GPU to obtain necessary accuracy on the data. This previous study provided lower accuracy of 0.337% (for both CPU and GPU) by using GPU kernels. Besides, all parallel parts of the PSO algorithm were not mapped to the GPU architecture and implemented as GPU kernels. The main objective of my project is to improve the CPU accuracy and to add GPU kernels. In the current study, greater accuracy was achieved in the C# implementation by improving the CPU random number generation that uses large prime numbers as seeds in the C++ implementation. Further I have also improved GPU implementation by incorporating an additional kernel and integrating GPU randomization to the existing kernel by using CuRAND library.
The results from the study have achieved 0.87% CPU accuracy for training and testing data. Increasing the number of epochs resulted in reaching an optimum of 1.0% accuracy. GPU accuracy obtained for the same is about 0.67%. But, the GPU implementation does not provide a speedup compared to its CPU implementation for the neural network used in this study. To leverage the power of GPU, we require a network which needs more than 200 particles for its training. Hence, future studies may explore research on investigating such large neural networks by comparing CPU and GPU runtimes.

______________________________, Committee Chair
Dr. Pinar Muyan-Ozcelik

______________________________
Date
ACKNOWLEDGEMENTS

I would like to express my gratitude to my Professor, Dr. Pinar Muyan-Ozcelik, whose expertise, understanding, and patience, added considerably to my project work. I appreciate her vast knowledge and skill in many, and her assistance in writing my project report. I would like to thank the other members of my committee in the Computer Science department for the assistance they provided at all levels of the research project.

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 would also like to thank my family for the support they provided me through my entire life and in particular, I must acknowledge my parents without whose love, encouragement, I would not have finished this project work.
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
<tr>
<td></td>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td></td>
<td>List Of Listings</td>
<td>xii</td>
</tr>
<tr>
<td></td>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.1 Artificial Neural Networks</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.2 How Artificial Neurons work</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.3 Training method for Artificial Neural Networks:</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2.3.1 Back Propagation</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.3.2 Introduction to particle swarm optimization (PSO)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2.4 CPU and the need for a better performance system</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.4.1 The GPU</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.4.2 GPU Computing</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2.5 NVIDIA CUDA</td>
<td>14</td>
</tr>
<tr>
<td>3.</td>
<td>IMPLEMENTATION</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>3.1 CPU Implementation</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>3.2 GPU Implementation</td>
<td>19</td>
</tr>
</tbody>
</table>
4. RESULTS ..................................................................................................................23
5. FUTURE WORK .......................................................................................................25
6. SAMPLE OUTPUT ....................................................................................................26
Appendix A ..................................................................................................................29
Appendix B ..................................................................................................................30
References ...................................................................................................................53
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Epochs time elapsed: CPU Implementation vs. GPU Implementation</td>
<td>24</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A neuron</td>
<td>4</td>
</tr>
<tr>
<td>2. Artificial neural network representation</td>
<td>5</td>
</tr>
<tr>
<td>3. Back Propagation algorithm view</td>
<td>7</td>
</tr>
<tr>
<td>4. PSO algorithm flow diagram</td>
<td>9</td>
</tr>
<tr>
<td>Listings</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1. The main concept of the PSO algorithm in a function</td>
<td>11</td>
</tr>
<tr>
<td>2. Code snippet from NVIDIA.com</td>
<td>13</td>
</tr>
<tr>
<td>3. Code snippet of the main function from main.cpp</td>
<td>16</td>
</tr>
<tr>
<td>4. Code snippet of the main PSO function from kernel_new.cpp</td>
<td>18</td>
</tr>
<tr>
<td>5. getRandomClamped() function from kernel_new.cpp</td>
<td>19</td>
</tr>
<tr>
<td>6. Code snippet for GPU parallelization from kernel.cu</td>
<td>20</td>
</tr>
<tr>
<td>7. Code snippet for CUDA random number generation from kernel.cu</td>
<td>21</td>
</tr>
<tr>
<td>8. Code snippet for pBest implementation from kernel.cu</td>
<td>22</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Artificial Neural Networks (ANN’s) are inspired by the highly complex yet efficient Biological Neural Networks. The successful working of biological systems mainly influences the study of artificial neural networks.

An attempt was made previously in a research work to train neural networks using the PSO Algorithm. The main objective of this study was to predict the species based on flowers characteristics by using GPU Computing and CUDA programming language. In this project an attempt was made to predict the species based on a flowers’ characteristics by using Fisher’s iris data set [1 - 2]. However the results of the study has not provided good accuracy for both, CPU and on the GPU.

The training data set chosen for this experiment is Fisher’s iris data set [1] which 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 [2]. The Visual studio article [3] describes this data set to provide 0.95 and 0.83 accuracy on the training and test data respectively. The reason behind using the GPU for this algorithm is because we want to speed up its runtime. The GPU improves the execution time of the PSO algorithm by running the independent computations involved in this algorithm in parallel using the massively parallel architecture of the GPU that has hundreds of cores. Hence, by running the algorithm on a parallel GPU instead of a serial CPU, we achieve significant speedup.
Firstly, to improve the CPU accuracy, I have made use of “good random numbers” which is determined by the seed value. By making use of a large prime number, I aim to increase the accuracy of the CPU to about 0.87% which is a large improvement than what was made on the previous paper [2]. For the GPU, I have included an additional kernel called “PBest” which is used to calculate the value calculated for the position of the particle when it comes closest to the desired position. The other implementation to improve randomizations in the CUDA code was by making use of the CuRAND library which is the CUDA equivalent for generating random numbers in order to improve the accuracy on the GPU. Adding this to the current CUDA program code helps in achieving an improved accuracy such as 0.67% and stopping variations in accuracy which occur in each attempt of execution.

Hence, this project makes an attempt to increase the accuracy of the trained and test data set for the above mentioned data set on the GPU, to match the accuracy that the CPU implementation has achieved so far by making use of the same data set. Further the project also aim to add the rest of the kernels additionally for the GPU version of the code with the aim of achieving better accuracy by adding more randomizations as per the suggestions made in the previous project [2].
2. BACKGROUND

2.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are being considered as the frontier of technology of the future in computing. These are to a certain degree rudimentary electronic models similar to the neural structure of the human brain. During recent years the ANNs are being designed mainly on the basis of biological neural networks which are inspired mainly by the recent advances in biological research. Hence the advances in biological research are in a way helping the preliminary understanding of the natural thinking mechanism.

The background of Artificial Neural Network is similar to how the human brain processes the information, which is to analyze and recognize objects. Neurons in the ANN are constructed physically or used in a computer for simulation. Every neuron takes a number of input signals on the basis of an internal weighing system logic which generates a single output, which subsequently goes into another neuron as input.

2.2 How Artificial Neurons work

Neuron which is basically main processing element of neural network. It is a key building block of human awareness embraces wide-ranging capabilities. A biological neuron receives inputs from other sources, combines them by some means and carries out a nonlinear operation on the outcome, and brings out the final result [4]. This allows the humans to react to his environment by transporting stimuli. A neuron anatomically consists
of a cell body (soma), dendrites and the axon with its terminals. Dendrites are the neural branches that bring signals to the body of the cell, whereas the main axon simply transmits the signal to the next neural body. The key to neural function is synaptic signaling process, which is partly electrical and partly chemical [5].

Next to this process is the neurotransmitter, which releases it to the next neuron triggering electrical impulses.

![Diagram of a neuron](image)

**Figure 1. A neuron [6]**

The Neural Network has three layers which are interconnected. The first layer consists of input neurons. These neurons send data on to the second layer, which in turn sends the output neurons to the third layer [7].
In order to understand artificial neural computing in a better way, first it is important to know how a conventional 'serial' computer and its software process information works. A serial computer has a central processor that can address an array of memory locations where data and instructions are stored. The processor makes the computations by reading any instruction as well as any data from its memory addresses. The instruction is then executed and the results are saved in a specified memory location as required. In a serial system a standard or parallel computational steps are in the form of deterministic, sequential and logical. Further the variable can be tracked from one operation to another.

In comparison to conventional 'serial' computer, ANNs are not sequential and they are not even deterministic. Besides in ANNs, there are no complex central processors, rather there are many simple ones which generally do nothing more than to take the
weighted sum of their inputs from other processors. ANNs do not execute programmed instructions; they respond in parallel (either simulated or actual) to the pattern of inputs presented to it. There are also no separate memory addresses for storing data. Instead, information is contained in the overall activation 'state' of the network. 'Knowledge' is thus represented by the network itself, which is quite literally more than the sum of its individual components.

2.3 Training method for Artificial Neural Networks:

When a network has been structured for a particular application, that network is ready to be trained. To start this process the initial weights are chosen randomly. Afterwards, the training or learning begins. The process of finding the set of weights and bias values that best match the given existing data is called training the neural network [8].

There are 2 stages to how a neural networks learns things. The first is the “being trained” stage and the second “after being trained” stage where the NN is learning or is learning to operate normally. The initial steps begins with the information (data) being fed into the network in the input stage by making use of the units which leads to the triggering of the consecutive hidden layer units which ultimately arrive at the output units. This design refers to a Feed Forward Network. The inputs are received at the first stage, which are multiplied by the weights of the connections and make use of when they travel along.

The main idea of the neural network is to train use the concept of “Feedback”. This term is derived from normal, human experiences like learning the nuances of a new
instrument and get familiar with the force to use or musical note to keep something in mind while playing. Hence Feedback is referred as something which can be learnt over a period of time. Similarly the ANN learns things by using a “Feedback” process, which is also called as Back Propagation (BP).

2.3.1 Back Propagation

One of the most commonly used Neural Networks training algorithm, is the Back Propagation algorithm (BP). Basically this algorithm is a multi-layer network which uses a weight adjustment built on the sigmoid function, like the delta rule. The back propagation method is an example of supervised learning, when the target of the function is known.

![Figure 3. Back Propagation algorithm view [9]](image)

There are some limitations in the use of back propagation networks. Mainly, they tend to be slow in being able to train the neural networks and do not exhibit much parallelism. This would not be beneficial in case we want to train the neural networks. Instead, by using
PSO it provides a faster alternative since it has parallel computation and hence we can improve its runtime by implementing it on a parallel architecture like the GPU.

2.3.2 Introduction to particle swarm optimization (PSO)

The PSO was introduced by Kennedy and Eberhart in 1995 as a parallel evolutionary computation technique [10] based on the social behavior and movement dynamics of insects, birds and fish. The communication among the birds in the flock is utilized as basic concept for this method. It has been used as effective method in optimizing complex multidimensional problems [11].

2.3.2.1 How PSO algorithms work

The PSO algorithms work by using 2 main terminologies:

- The swarm: the population $P=\{p_1,\ldots,p_n\}$ of the feasible solutions
- The particles: feasible solutions $p_1,\ldots,p_n$ are called particles [12].

Every particle in the swarm keeps a track of its coordinates in the problem space which are associated with the best solution or the fitness it has achieved so far. By using this fitness, its value is calculated and stored which is called as the pbest. This pbest value is used to indicate the closest the data has ever come to the target since the algorithm has started [12]. Another “best” value that is tracked by the global version of the article swarm optimizer is the overall best value, and its location, obtained so far by any particle in the population. This location is called gbest [13]. This value changes only when any particles pBest value comes closer to the target than the gBest value will. With every iteration of the
algorithm, the value of gBest comes closer and closer to the target until one of the particle reaches its target [12].

![PSO Algorithm Flow diagram](image)

**Figure 4. PSO Algorithm Flow diagram [14]**

2.3.2.2 Applications of PSO

Particle swarm optimization has been used across a wide range of applications. The important areas where PSOs have shown promising results includes multimodal problems and problems for which there is no specialized method available or where all specialized
methods give unsatisfactory results. The Researchers in many countries are experimenting with particle swarms and applying them in real-world applications such as:

- The PSO algorithm is suited to study of continuous variable problems and also Performance comparable to Genetic algorithms.

- The PSO algorithm has been extensively used in Training of neural networks. It is also being used in various other fields like identification of Parkinson’s disease, extraction of rules from fuzzy networks, Image recognition, Optimization of electric and in power distribution networks.

2.3.2.3 PSO code implementation with regard to the Iris Fisher’s Dataset

The training data set chosen for this experiment is Fisher’s iris data set [1] which 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 [2].

Out of the 150 samples, the study has used a small subset of the sample, i.e. 30 to test and train accuracy on the CPU. The sample of 30 was further divided as:

- 80 per cent= training data: consisting of 24 samples
- 20 per cent = testing data: consisting of 6 samples to examine and to achieve the desired accuracy. The Appendix-A shows this encoded training and testing dataset.
initialize each particle to random state
(position, velocity, error, best-position, best-error)
save best position of any particle (global -best)

loop until done
  for each particle in swarm
    compute new particle velocity
    use new velocity to compute new position
    compute error of new position
    if new error better than best-error
      best-position = new position
    if new error better than global -best
      global -best = new position
  end for
end loop
return global - best position

Listing 1. The main concept of the PSO Algorithm in a function

2.4 CPU and the need for a better performance system

The basic functions of a CPU include fetching the instruction, decoding the
instruction and executing it. In modern world, the function of the Central Processing Unit
is limited to processing the instructions by decoding the programs and other files. Which
introduces the concept of GPU’s.

2.4.1 The GPU

The GPU is basically a single chip processor capable of processing millions of
instructions per second. Just as the CPU cores are designed to execute a thread of
sequential instruction with high speed, GPUs are created to running multiple threads parallel for rapid execution.

One of the important reasons for shifting to Graphics Processing Unit from the regular Central Processing Unit is to process the graphics much faster.

GPUs do not require large cache not like regular processors. They provide faster memory bandwidth because of parallel processing. This way, we can use the GPU for non-graphics tasks that has parallel computation as well.

2.4.2 GPU Computing

At present, GPU computing has become a major trend in the field of parallel computing by making use of multiple threads to run programs. GPU Computing makes use of the high processing power of the GPU to be used together with the CPU to improve the performance on certain applications, which require the power of GPU computing. Explaining more, the GPU accelerated applications use high-level languages to run the sequential part of their workload onto the CPU, which is only optimized for single-threaded performance and accelerating parallel processing on the GPU [2]. The process of GPU Computation is to effectively offload intense computations and calculations onto the GPU unit, while still making use of the CPU for running one part of the code. All that matters on the users end is that the applications run faster.

An effective example of the GPU computing obtained from NVIDIA developer’s website [14] below explains the accomplishment of vector addition on the GPU.
int main( void )
{
    int a[N], b[N], c[N]; int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++)
    {
        a[i] = -i; b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
    add<<<>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

    for(int i=0; i<N; i++)// display the results
    {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
}

Listing 2. Code snippet from NVIDIA.com [14]

Here, there are three arrays on the device using calls to cudaMalloc(): two arrays, dev_a
and dev_b, to hold inputs, and one array, dev_c, to hold the result [14]. Memory cleanup
is done cudaFree(). By using cudaMemcpy(), the input data is copied to the device with
the parameter called cudaMemcpyHostToDevice and then copied the result data back to the host with cudaMemcpyDeviceToHost. Execution is done using the device code in add() from the host code in main() using the triple angle bracket syntax [14].

2.5 NVIDIA CUDA

Compute Unified Device Architecture (CUDA), is a free programming platform given by NVIDIA. As mentioned earlier, there are hundreds and thousands of cores present inside the GPU and this CUDA programming language provides access to each of these execution units of the GPU. The main reason for introducing this language was to allow the NVIDIA GPUs to be easily used for compute applications by the users.

The CUDA language is basically an extension of C language with its own libraries and compilers.
3. IMPLEMENTATION

This project work implements PSO algorithm on the CPU and the GPU in order to distinguish between the results of CPU and GPU. The aim of this project is to achieve greater accuracy than what has been achieved on the CPU. Besides an attempt was made to show greater accuracy on the GPU by using NVIDIA’s CUDA programming [15]. The training data set chosen for this experiment is Fisher’s iris data set [1]. Out of the total of 150 irises, a small subset of the sample i.e. 30 is used to test and train the accuracy on the CPU.

In our department, we have a dedicated CUDA server equipped with two NVIDIA GPUs (Quadro K1200 and Quadro K620) and an Intel Xeon CPU (E5-2620). This server has the CentOS Linux (release 6.7) operating system, a C/C++ compiler (gcc version 4.4.7) and CUDA programming environment (version 7.5). I am using this server to implement the CPU and GPU implementation of the PSO algorithm. In order to use the Linux enabled CUDA, I have created a Linux Makefile [Appendix B] which is to create the build to execute the code and display the output files.

Firstly, in these algorithms, the approach is to start at a random position in the swarm and calculate the velocity to arrive at the next position. At every position, calculate the error to find the global best position. At the end, calculate the 'accuracy' using the input data set and 'best' positions calculated. Hence it is clear from the above that to get better results, there is a need to improve the 'position' calculation by using good 'random' numbers.
3.1 CPU Implementation

I begin my implementation by calculating “good random numbers” by using the existing program that has used the system library call 'srand' to initialize the random number generation with the 'system time' as the 'seed'. The system time is a large integer that has the value of elapsed time in seconds since the Unix Epoch (1-Jan-1970 00:00:00). Hence the main purpose of this research is to improve the quality of the seed by using a large primary number. By experimenting with different prime numbers, I found out that the number ‘24989’ gave good results of accuracy.

```c
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 dummy_positions[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    float dummy_velocities[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    float dummy_pBests[NUM_OF_PARTICLES * NUM_OF_DIMENSIONS];
    // initialize gBest
    float gBest[NUM_OF_DIMENSIONS];
    // Improve quality of the seed by using a large prime value
    srand((unsigned)24989);
}
```

Listing 3. Code snippet of the main function from main.cpp

The main function from main.cpp has been changed to assume a new Prime value in the srand function. Using a large prime number in place of “System time (NULL)” which it was initially called gives us two advantages:
1. Adding a seed value as a prime number ensures mathematically that it will provide “good” random numbers.

2. It will give the stable results (unlike variations in the accuracies every time the program runs).

Looking at the above method, the analysis of the GPU function in main.cpp within the “cuda_pso” function ensures that we are able to get accurate results for the CPU accuracy, which is very encouraging i.e. 0.83%. So it is clear from this study that the accuracy level has increased from 0.337% from previous study to 0.83% from my study [2].

Secondly, the study [2] looked at the main PSO algorithm, which is written in C++ coding language. It is iteratively calculating the 'next best' position over the number of particles and number of dimensions. However, each iteration is using the same pair of random numbers for the calculation. These values are being calculated by calling 'main' program after generating them afresh at every iteration. In my study I have modified the algorithm to use a pair of freshly generated random numbers in each iteration within the iterating 'for' loop of the PSO function. This change improved the accuracy significantly.
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++)
    {
        // 1. Update velocity and position

        for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i++)
        {
            float rp = getRandomClamped();
            float rg = getRandomClamped();

            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];
        }
    }
}

Listing 4. Code snippet of the main PSO function from kernel_new.cpp

The method in C++ for calling a fresh set of random numbers at every iteration is by using getRandomClamped() function inside the PSO code. The declaration for the same can be made as follows:
// Get random function declaration

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

Listing 5. getRandomClamped() function from kernel_new.cpp

3.2 GPU Implementation

The next contribution to this project has been to add a similar random number generation in the GPU version of the PSO function which has been improved by using the random number generation functions of the 'CUDA' library. This method is used instead of the 'host' library.

By using a CuRAND library, a suggestion made in the previous study to help improve Random number generation in the CUDA code. I have added a function called as “curand”. To implement the functions of this library, the appropriate headers must be included to import the entire class of library. That is <curand.h> and <curand_kernel.h>. This performs the same function as the getRandomClamped functioned mentioned in [Appendix B].
/* this GPU kernel function calculates a random number and stores it in the parameter */

__global__ void random(int* result) {

    /* CUDA’s random number library uses curandState_t to keep track of the seed value we will store a random state for every thread */

    curandState_t state; /* initialize state */

    curand_init(0, /* the seed controls the sequence of random values that are produced */
                0, /* the sequence number is only important with multiple cores */
                0, /* the offset is how much extra we advance in the sequence for each call, can be 0 */
                &state);

    /* curand works like rand - except that it takes a state as a parameter */

    *result = curand(&state) % RAND_MAX;
}

Listing 6. Code snippet for GPU parallelization from kernel.cu

The random number generation function using CuRAND is demonstrated in the code snippet from Appendix B below: 
__global__ void kernelUpdateParticle(float *positions, float *velocities, float *pBests, float *gBest)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= NUM_OF_PARTICLES * NUM_OF_DIMENSIONS)
        return;
    float rp;
    float rg;
    random(&rp);
    random(&rg);
}

Listing 7. Code snippet for CUDA random number generation from kernel.cu

Finally, as a part of the GPU implementation I have added a new kernel for the GPU version of the code with the hopes of improving the particles speedup time.

The kernel “PBest” is the value indicating the closest the particle's data has ever come to the target value indicated. PBest is implemented using the function called “KernelUpdatePBest” which is shown below. This function utilizes the current pbest and gbest value to calculate the value of the pbest in terms of the position it is at currently.
__global__ void kernelUpdatePBest(float *positions, float *pBests, float* gBest)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if(i >= NUM_OF_PARTICLES * NUM_OF_DIMENSIONS || i % NUM_OF_DIMENSIONS != 0)
    return;
    for (int j = 0; j < NUM_OF_DIMENSIONS; j++)
    {
        tempParticle1[j] = positions[i + j];
        tempParticle2[j] = pBests[i + j];
    }
    if (fitness_function(tempParticle1) <
 fitness_function(tempParticle2))
    {
        for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
            pBests[i + k] = positions[i + k];
    }
}

Listing 8. Code snippet for pBest implementation from kernel.cu

The results obtained by adding an additional kernel “PBest” doesn’t specially impact the conditions of accuracy achieved, but we can ensure that all of the kernels are correctly implemented into the GPU code to achieve stability and cut down on the variations with the accuracy achieved with every run.
4. RESULTS

As previously stated, the results addressed by Sakhare’s paper [2] provided lower accuracy of 0.337% (for both CPU and GPU) by using GPU kernels.

The results from my study clearly show that the CPU achieved 0.87% accuracy which was more than the projected for training & testing data by slowly and steadily increasing the number of epochs which resulted in reaching an optimum of = 1.0%. These results of stability in the accuracy generated and increase of accuracy in comparison matches perfectly to the previously used C# Code of the Visual Studio magazine [3].

At the moment, the GPU accuracy remains constant at 0.67% and does not improve as its CPU counterpart when we increase the number of epochs. Further investigation is needed to explain the reason that causes this result. The results obtained can be demonstrated in a tabular form below to get a clear understanding of the accuracy generated by using different “Epochs time” for the same number of 12 particles each time. While using the PSO for the training of the neural network that is used for Iris data set, we cannot achieve a speedup on its GPU implementation compared to its CPU implementation. It is because we cannot take full advantage of the parallel architecture of the GPU with the small number of particles that is required by the Fishers iris dataset [1]. Thus, to take full advantage of the GPU we must use a neural network which requires more particles for its PSO training. The results obtained below are the measure of improved accuracy based on variations in the Epochs time. The elapsed time refers to Training data runtime.
<table>
<thead>
<tr>
<th>Epochs</th>
<th>CPU Accuracy</th>
<th>GPU Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Training Data</td>
<td>Testing Data</td>
</tr>
<tr>
<td>2000</td>
<td>0.87</td>
<td>0.67</td>
</tr>
<tr>
<td>3000</td>
<td>1.0</td>
<td>0.83</td>
</tr>
<tr>
<td>4500</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
5. FUTURE WORK

The real power of the GPU can be demonstrated with higher speedup and this is achieved when there is a larger number of particles involved in the data set which can also significantly improve the runtime of training the neural network.

As a part of the future works, it is ideal to select a dataset with 200+ particles, which can run parallel by making use of the multi core processor power on the GPU. At the moment, as mentioned in the results the GPU accuracy does not matchup to its CPU counterpart as we increase the number of epochs and more investigation is needed to find out why. Due to time limitation, I wasn’t able to test the GPU power by using a larger number of particles, which I predict could have significantly improved the runtime of training the neural network.

For future work, a technique called as Reduction technique can be made use of on the GPU. This way, the gbest can also be implemented on the GPU, which would in turn help us achieve better-run time.
6. SAMPLE OUTPUT

1. CPU Accuracy results only

```bash
[banurshd@helios release]$ ./pso

Setting numParticles = 12
Setting maxEpochs = 700
Setting early exit MSE error = 0.060
Setting probDeath = 0.005
================================== CPU =====================================
Time elapsed : 0.940 ms

Best weights >>
-5.773  3.295  -13.949  7.743  1.061  -1.432  -1.933  -1.081  -0.770  -0.545
-10.055  4.918  -6.471  5.211  2.671  6.378  2.434  -0.976  -3.514  -4.440
  -0.119  0.741  9.054  -3.832  7.312  -1.857  -2.444  -1.602  -0.263  -0.006
-3.873  1.499  1.830  -11.412  -4.441  -0.711  4.045  0.304  2.600  2.689
  4.000  1.744  2.796  -5.170  0.841  -0.851  -0.004  3.997  1.827  -0.535
  2.983
Accuracy on training data = 0.8333
CPU Testing
Accuracy on Testing data = 0.8333
```
2. GPU Accuracy results with new Kernel added

```
[banursh@helios release]$ ./pso

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

====================== GPU Training =======================
Time elapsed :       2.620 ms

  Best weights >>
  1.295 -1.163 -0.876 1.365 2.153 2.543 -2.075 -0.103 0.562 2.005
  0.834 1.561 1.503 -0.244 0.448 0.039 0.593 -0.421 3.956 1.291
  -2.163 -3.776 -0.692 0.783 -3.174 -0.370 3.513 -2.332 -0.082 0.413
  -1.369 2.956 -1.590 -0.607 -1.963 0.616 -1.592 -1.419 2.150 1.400
  -1.385 1.281 -3.157 1.943 -1.750 -0.760 1.053 4.061 -1.440 -1.236
  1.760
Accuarcy on training data = 0.6667
GPU Testing
Accuracy on Testing data = 0.6667

====================== CPU =======================
Time elapsed :       0.660 ms

  Best weights >>
  -3.131 -12.368 -7.032 2.468 0.758 4.525 -0.898 -7.608 -0.933 -3.585
  -11.439 3.790 1.828 -1.384 1.598 0.942 6.778 0.148 5.199 0.540
  -0.270 1.220 2.800 -3.244 2.299 -1.087 -3.268 1.208 -0.027 0.060
  9.007 2.913 5.005 -0.798 -1.323 -2.352 -2.756 -0.642 4.732 6.743
  6.180 1.711 1.841 3.240 -44.617 -0.734 2.056 0.662 4.562 1.879
  -24.851
Accuracy on training data = 0.8750
CPU Testing
Accuracy on Testing data = 0.6667
```
3. CPU and GPU results with CuRAND implementation

[bannurshd@helios release]$ ./pso

Setting numParticles = 12
Setting maxEpochs = 700
Setting earlyExit MSE error = 0.050
Setting probDeath = 0.005
==================== GPU Training =====================
Time elapsed: 3.020 ms

Best weights >>
-4.319  6.040  -0.130  5.541  -1.907  -2.406  -4.555  0.468  -1.479  -2.526
-3.156  3.676  2.179  4.344  1.935  -1.337  2.050  1.450  2.818  -0.145
-0.660  0.189  -4.741  -4.182  5.239  1.380  -1.217  -2.269  0.349  -0.324
 3.595  1.150  -0.404  -0.654  0.571  2.804  2.059  1.135  -2.848  5.095
 3.725  -0.884  3.256  -0.211  -2.659  -0.925  3.572  4.506  5.641  0.270
-1.754
Accuracy on training data = 0.6667
GPU Testing
Accuracy on Testing data = 0.6667

==================== CPU =====================
Time elapsed: 0.680 ms

Best weights >>
-5.337  1.757  -1.365  -1.687  -5.926  -13.133  -4.041  0.581  -1.403  -2.912
-1.461  2.767  -1.729  4.118  4.321  0.085  7.306  2.139  0.427  6.711
-4.243  -1.894  -2.430  -4.221  0.758  1.029  -0.855  1.114  1.644  0.273
-0.718  2.426  1.677  -3.285  -3.981  0.136  5.905  0.565  -2.234  3.572
-6.278  -1.289  1.012  -1.652  -2.926  -0.735  0.994  7.986  5.153  0.665
 3.525
Accuracy on training data = 0.5833
CPU Testing
Accuracy on Testing data = 0.6667
APPENDIX A

As listed by Sakhare[2015][3], I provide the training and testing data set for the selected 30 sample Fisher’s iris data set.

Training data:

<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>6.3</td>
<td>2.9</td>
<td>5.6</td>
<td>1.8</td>
<td>1.0</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

Test data:

<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

Makefile

#############################################################################
# Copyright 1993-2015 NVIDIA Corporation. All rights reserved.
# NOTICE TO USER:
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE. IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND. NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR
# PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT,
# INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM
# USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE
# USE
# OR PERFORMANCE OF THIS SOURCE CODE.
# U.S. Government End Users. This source code is a "commercial item"
as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
#############################################################################
# Makefile project only supported on Mac OS X and Linux Platforms)
#
#############################################################################
# Location of the CUDA Toolkit
CUDA_PATH  := /usr/local/cuda-7.5

###############################################
# start deprecated interface #
###############################################
ifeq ($(x86_64),1)
  $(info WARNING - x86_64 variable has been deprecated)
  $(info WARNING - please use TARGET_ARCH=x86_64 instead)
  TARGET_ARCH  := x86_64
endif
ifeq ($(ARMv7),1)
  $(info WARNING - ARMv7 variable has been deprecated)
  $(info WARNING - please use TARGET_ARCH=armv7l instead)
  TARGET_ARCH  := armv7l
endif
ifeq ($(aarch64),1)
  $(info WARNING - aarch64 variable has been deprecated)
  $(info WARNING - please use TARGET_ARCH=aarch64 instead)
  TARGET_ARCH  := aarch64
endif
ifeq ($(ppc64le),1)
  $(info WARNING - ppc64le variable has been deprecated)
  $(info WARNING - please use TARGET_ARCH=ppc64le instead)
  TARGET_ARCH  := ppc64le
endif
ifneq ($(GCC),)
  $(info WARNING - GCC variable has been deprecated)
  $(info WARNING - please use HOST_COMPILER=$(GCC) instead)
  HOST_COMPILER  := $(GCC)
endif
ifneq ($(abi),)
  $(error ERROR - abi variable has been removed)
endif
###############################################
# end deprecated interface #
###############################################

# architecture
HOST_ARCH   := $(shell uname -m)
TARGET_ARCH := $(HOST_ARCH)
ifeq (,$(filter $(TARGET_ARCH),x86_64 aarch64 ppc64le))
  TARGET_SIZE := 64
else ifeq ($(TARGET_ARCH),armv7l)
  TARGET_SIZE := 32
else
  $(error ERROR - unsupported value $(TARGET_ARCH) for TARGET_ARCH!)
endif
ifeq ($(TARGET_ARCH),$(HOST_ARCH))
  ifeq (,$(filter $(HOST_ARCH)-$(TARGET_ARCH),aarch64-armv7l x86_64-armv7l x86_64-aarch64 x86_64-ppc64le))
    $(error ERROR - cross compiling from $(HOST_ARCH) to
    $(TARGET_ARCH) is not supported!)
  endif
endif
# operating system
HOST_OS := $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
TARGET_OS := $(HOST_OS)
ifeq (,$(filter $(TARGET_OS),linux darwin qnx android))
    $(error ERROR - unsupported value $(TARGET_OS) for TARGET_OS!)
endif

# host compiler
ifeq ($(TARGET_OS),darwin)
    ifeq ($(shell expr `xcodebuild -version | grep -i xcode | awk '{print $2}' | cut -d'.' -f1` \(>= 5),1)
        HOST_COMPILER ?= clang++
    endif
else ifneq ($(TARGET_ARCH),$(HOST_ARCH))
    ifeq ($(TARGET_ARCH),$(HOST_ARCH),x86_64-armv7l)
        ifeq ($(TARGET_OS),linux)
            HOST_COMPILER ?= arm-linux-gnueabihf-g++
        else ifeq ($(TARGET_OS),qnx)
            ifeq ($(QNX_HOST),)
                $(error ERROR - QNX_HOST must be passed to the QNX host toolchain)
            endif
            ifeq ($(QNX_TARGET),)
                $(error ERROR - QNX_TARGET must be passed to the QNX target toolchain)
            endif
            export QNX_HOST
            export QNX_TARGET
            HOST_COMPILER ?= $(QNX_HOST)/usr/bin/arm-unknown-nto-qnx6.6.0eabi-g++
        else ifeq ($(TARGET_OS),android)
            HOST_COMPILER ?= arm-linux-androideabi-g++
        endif
    else ifeq ($(TARGET_ARCH),aarch64)
        ifeq ($(TARGET_OS),linux)
            HOST_COMPILER ?= aarch64-linux-gnu-g++
        else ifeq ($(TARGET_OS), android)
            HOST_COMPILER ?= aarch64-linux-android-g++
        endif
    else ifeq ($(TARGET_ARCH),ppc64le)
        HOST_COMPILER ?= powerpc64le-linux-gnu-g++
    endif
endif
HOST_COMPILER ?= g++
NVCC          := $(CUDA_PATH)/bin/nvcc -ccbin $(HOST_COMPILER)

# internal flags
NVCCFLAGS   := -m$(TARGET_SIZE)
CCFLAGS     :=
LDFLAGS     :=

# build flags
ifeq ($(TARGET_OS),darwin)
LDFLAGS += -rpath $(CUDA_PATH)/lib
CCFLAGS += -arch $(HOST_ARCH)
else ifeq ($(HOST_ARCH)-$(TARGET_ARCH)-$(TARGET_OS),x86_64-armv7l-linux)
    LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
    CCFLAGS += -mfloat-abi=hard
else ifeq ($(TARGET_OS),android)
    LDFLAGS += -pie
    CCFLAGS += -fpie -fpic -fexceptions
endif
ifneq ($(TARGET_ARCH),$(HOST_ARCH))
    ifeq ($(TARGET_ARCH)-$(TARGET_OS),armv7l-linux)
        ifneq ($(TARGET_FS),)
            GCCVERSIONLTEQ46 := $(shell expr `$(HOST_COMPILER) -dumpversion` \< 4.6)
            ifeq ($(GCCVERSIONLTEQ46),1)
                CCFLAGS += --sysroot=$(TARGET_FS)
            endif
            LDFLAGS += --sysroot=$(TARGET_FS)
            LDFLAGS += -rpath-link=$(TARGET_FS)/lib
            LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
            LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-gnueabihf
        endif
    endif
endif
#if Debug build flags
ifeq ($(dbg),1)
    NVCCFLAGS += -g -G
    BUILD_TYPE := debug
else
    BUILD_TYPE := release
endif
ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))
#endif
# Common includes and paths for CUDA
INCLUDES := -I../../common/inc
LIBRARIES :=
SAMPLE_ENABLED := 1

# Gencode arguments
SMS ?= 20 30 35 37 50 52

ifeq ($(SMS),)
$(info >>> WARNING - no SM architectures have been specified - waiving sample <<<)
SAMPLE_ENABLED := 0
endif

ifeq ($(GENCODE_FLAGS),)
# Generate SASS code for each SM architecture listed in $(SMS)
$(foreach sm,$(SMS),$(eval GENCODE_FLAGS += -gencode arch=compute_$sm,code=sm_$sm))

# Generate PTX code from the highest SM architecture in $(SMS) to guarantee forward-compatibility
HIGHEST_SM := $(lastword $(sort $(SMS)))
ifneq ($(HIGHEST_SM),)
GENCODE_FLAGS += -gencode arch=compute_$HIGHEST_SM,code=compute_$HIGHEST_SM
endif
endif

ifeq ($(SAMPLE_ENABLED),0)
EXEC ?= @echo "[@]"
endif

#######################################################################
####################
# Target rules
all: build
build: pso
check.deps:
ifeq ($(SAMPLE_ENABLED),0)
 @echo "Sample will be waived due to the above missing dependencies"
else
 @echo "Sample is ready - all dependencies have been met"
endif
kernel.o:kernel.cu
 $(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<
kernell_new.o:kernel_new.cpp
 $(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<
main.o:main.cpp
 $(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -c $<
-pso: kernel.o kernel_new.o main.o
   $(EXEC) $(NVCC) $(ALL_LDFLAGS) $(GENCODE_FLAGS) -o $@ $+
$(LIBRARIES)
   $(EXEC) mkdir -p
   $(EXEC) cp $@ ..../bin/$(TARGET_ARCH)/$(TARGET_OS)/$(BUILD_TYPE)
run: build
   $(EXEC) ./pso
   $(EXEC) cp $@ ..../bin/$(TARGET_ARCH)/$(TARGET_OS)/$(BUILD_TYPE)/pso
   $(EXEC) rm -f pso kernel.o kernel_new.o main.o
   $(EXEC) rm -rf ..../bin/$(TARGET_ARCH)/$(TARGET_OS)/$(BUILD_TYPE)/pso
   $(EXEC) clobber: clean
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#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

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 }, { 4.7, 3.2, 1.3, 0.2, 0, 0, 1 }, { 4.9, 1.4, 0.2, 0, 0, 1 }, { 7.6, 3, 6.6, 2.1, 1, 0, 0 }, { 4.9, 2.4, 3.3, 1, 0, 1 }, { 5.4, 3.9, 1.7, 0.4, 0, 0, 1 }, { 4.9, 3.1, 1.5, 0.1, 0, 0, 1 }, { 5, 3.6, 1.4, 0.2, 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, 1, 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 }, { 5.1, 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, 1, 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);
static 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);
#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)24989);

    // 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,end;
// GPU PSO main function
begin = clock();
cuda_pso(positions, velocities, pBests, gBest);
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 ============= //

// Use same copy of Initialized particles
for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
gBest[k] = dumy_pBests[k];

begin = clock();
// CPU PSO main function
```c
psd(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("Minimum = %f\n", MeanSquaredError(gBest, 
NUM_OF_DIMENSIONS));
printf("Accuracy on training data = %0.4f\n", 
Accuracy(trainData, 24, gBest));
printf("CPU Testing \n");

printf("Accuracy on Testing data = %0.4f\n", Accuracy(testData, 
6, gBest));
// ================== END OF CPU ================== //

return 0;
}
```cpp
#include <string.h> // memcpy
#include "my_kernel.h"

float* inputs;
float** ihWeights; // input-hidden
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 recomputed 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];
hoOutputs = 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, h-o weights, h-o 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, trainData[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
}

// Get random antara low dan high
float getRandom(float low, float high)
{
    return low + float((high - low) + 1) * rand() / (RAND_MAX + 1.0));
}

// Get random antara 0.0f dan 1.0f inclusive
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++)
    {
        //1. Update velocity and position

        for (int i = 0; i < NUM_OF_PARTICLES * NUM_OF_DIMENSIONS; i++)
        {
            float rp = getRandomClamped();
            float rg = getRandomClamped();
            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
# Kernel.cu

```c
#include <cuda_runtime.h>
#include <cuda.h>
#include <math_functions.h>
#include "my_kernel.h"
#include <unistd.h>
#include <stdio.h>

/* we need these includes for CUDA's random number stuff */
#include <curand.h>
#include <curand_kernel.h>

#define MAX 100

/* this GPU kernel function calculates a random number and stores it in
the parameter */
__device__ void random(float* result) {
    /* CUDA's random number library uses curandState_t to keep track of
the seed value
    we will store a random state for every thread */
    curandState_t state;
    curand_init(0, /* the seed controls the sequence of random values
that are produced */
        0, /* the sequence number is only important with multiple
cores */
        0, /* the offset is how much extra we advance in the
sequence for each call, can be 0 */
        &state);

    /* curand works like rand - except that it takes a state as a
parameter */
    *result = curand(&state) % RAND_MAX;
}

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

__device__ float fitness_function(float x[]) {
    float res = 0;
    float y1 = 1 + (x[0] - 1) / 4;
    float yn = 1 + (x[NUM_OF_DIMENSIONS - 1] - 1) / 4;
    res += pow(sin(phi * y1), 2);
    for (int i = 0; i < NUM_OF_DIMENSIONS - 1; i++) {
        float y = 1 + (x[i] - 1) / 4;
        float yp = 1 + (x[i + 1] - 1) / 4;
```

res += pow(y - 1, 2) * (1 + 10 * pow(sin(phi * yp), 2))
+ pow(yn - 1, 2);
}
return res;

__global__ void kernelUpdateParticle(float *positions, float
*velocities, float *pBests, float *gBest)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= NUM_OF_PARTICLES * NUM_OF_DIMENSIONS)
        return;
    float rp;
    float rg;
    random(&rp);
    random(&rg);

    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];
}

__global__ void kernelUpdatePBest(float *positions, float *pBests,
float* gBest)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= NUM_OF_PARTICLES * NUM_OF_DIMENSIONS || i %
NUM_OF_DIMENSIONS != 0)
        return;

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

    if (fitness_function(tempParticle1) <
    fitness_function(tempParticle2))
    {
        for (int k = 0; k < NUM_OF_DIMENSIONS; k++)
            pBests[i + k] = positions[i + k];
    }
extern "C" void cuda_pso(float *positions, float *velocities, float *pBests, float *gBest)
{
    float tempParticle1[NUM_OF_DIMENSIONS];
    float tempParticle2[NUM_OF_DIMENSIONS];
    int size = NUM_OF_PARTICLES * NUM_OF_DIMENSIONS;

    float *devPos;
    float *devVel;
    float *devPBest;
    float *devGBest;
    //float fx,fy;
    //fx = (float) x/(float)RAND_MAX;
    //fy = (float) y/(float)RAND_MAX;

    // 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);
        // Update pBest
        kernelUpdatePBest << <blocksNum, threadsNum>> >(devPos, devPBest, devGBest);

        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. Particle death function
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 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);
REFERENCES


[Accessed: 9 November 2015]


[Accessed: 2 February 2016]


