J Supercomput (2016) 72:3868–3886
DOI 10.1007/s11227-016-1633-y

GPU-enabled back-propagation artificial neural network for digit recognition in parallel

Ricardo Brito[1] · Simon Fong[1] · Kyungeun Cho[2] · Wei Song[3] · Raymond Wong[4] · Sabah Mohammed[5] · Jinan Fiaidhi[5]

Published online: 10 February 2016
© Springer Science+Business Media New York 2016

Abstract In this paper, we show that the GPU (graphics processing unit) can be used not only for processing graphics, but also for high speed computing. We provide a comparison between the times taken on the CPU and GPU to perform the training and testing of a back-propagation artificial neural network. We implemented two neural networks for recognizing handwritten digits; one consists of serial code executed on the CPU, while the other is a GPU-based version of the same system which executes in parallel. As an experiment for performance evaluation, a system for neural network

✉ Simon Fong
ccfong@umac.mo

Ricardo Brito
mb55405@umac.mo

Kyungeun Cho
cke@dongguk.edu

Wei Song
sw@ncut.edu.cn

Raymond Wong
wong@cse.unsw.edu.au

Sabah Mohammed
sabah.mohammed@lakeheadu.ca

Jinan Fiaidhi
jfiaidhi@lakeheadu.ca

1 Department of Computer and Information Science, University of Macau, Macau SAR, China

2 Department of Computer and Multimedia Engineering, Dongguk University, Seoul, Korea

3 College of Information Engineering, North China University of Technology, Beijing, China

4 School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

5 Department of Computer Science, Lakehead University, Thunder Bay, Canada

training on the GPU is developed to reduce training time. The programming environment that the system is based on is CUDA which stands for compute unified device architecture, which allows a programmer to write code that will run on an NVIDIA GPU card. Our results over an experiment of digital image recognition using neural network confirm the speed-up advantages by tapping on the resources of GPU. Our proposed model has an advantage of simplicity, while it shows on par performance with the state-of-the-arts algorithms.

Keywords Artificial neural networks · Parallel execution · NVIDIA · CUDA

1 Introduction

It is well known that the GPU hardware contains many execution cores which can be used for high demanding graphics applications processing, image rendering and many other tasks concerning images. One of the most popular GPU brands is the NVIDIA graphics card. The NVIDIA released a framework that allows programmers to take advantages of the GPU for applications other than graphics processing or image rendering [1]. They provided the CUDA framework [2] which allows any programmer with a programming foundation in C/C++ to port general-purpose applications to the GPU. It allows programmers to write parallel programming code. In the last decade, there is an emerging trend in tapping the power of GPU for building more efficient machine learning algorithms [3]. Some success cases including the popular machine learning models such as Extreme Learning Machine (ELM) [4] and Support Vector Machine (SVM) [5] have been implemented on GPU parallel processing. A GPU-enabled version of ELM is implemented for time series prediction too [6]. Other than SVM and ELM, artificial neural network (ANN) is a good candidate which is able to execute in parallel and ride on the advantages of parallelism on GPU programming. In the research communities, several toolkits [7–9] are available for building GPU-based ANN. A most popular topology of ANN that was implemented on GPU is feedforward neural network [10]. Some are programmed in Java [11]. Latest Restricted Boltzmann Machines which is a deep learning type of ANN was made to run on GPU [12]. To the best of the authors' knowledge, there are not many back-propagation type of neural network source code available for free that runs on GPU. Neural networks are inherently parallel algorithms, and there are many variations. The closest implementation is by [13]; however, the application domain is on face recognition, whereas our application domain is digit recognition.

The focus of this paper is to describe in detail how a back-propagation artificial neural network runs in parallel in the GPU. The source code is available at: http://www.simonjamesfong.com. In particular, through an experiment of digit recognition, we will show how we accelerated the training process and implemented the training algorithm in the GPU. The execution model of the GPU as well is explained in this paper. The reminder of the paper is organized as follows. Section 2 covers the backgrounds of GPU programming and the basis of ANN. Our proposed model of back-propagation neural network (BPNN) to be running on GPU is presented in Sect. 3. The speed-up experiment is described in Sect. 4. Section 5 concludes the paper.

2 Background

2.1 GPU programming

The GPU's hardware structure consists of an array of streaming multiprocessors which contain cores; these cores are responsible for processing information. When writing code to be executed on the GPU, the programmer organizes everything in threads and the idea is to have each thread to execute an instruction in parallel. The GPU also contains a shared memory, which allows threads to share resources. Threads are organized inside blocks, which is just a group of threads. Threads inside the same block share registers and also shared memory which is a fast access memory. The GPU's global memory is where data will be located for processing. When the programmer wants to process some data on the GPU, he loads the data in the GPU's global memory, processes it and copies the result back to the CPU. In Fig. 1, a representation of the GPU's architecture is shown.

Each block declares an array of shared memory. Each thread inside the block will multiply one input (IN) with one weight (WN) and store the result in the shared memory array, and then when all the threads are done, they will work together to reduce the shared memory array in a single sum, which will then go through an activation function. This in our case is a sigmoid function, and this result will be stored in the hidden node that this block calculates. Every block does the same thing, and in this way each block calculates each hidden node in parallel. So we have a lot of parallelization going on

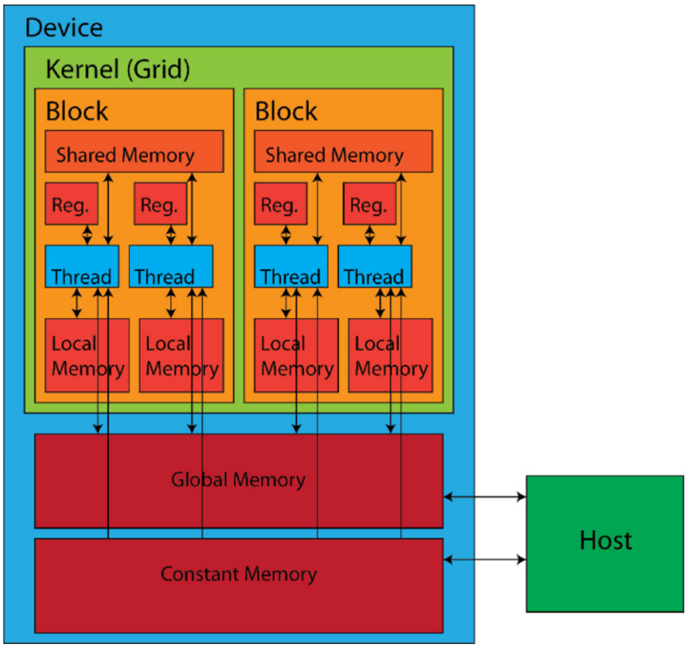


Fig. 1 A block diagram of the GPU architecture

here. The blocks calculate the hidden nodes in parallel and the operations inside each block to calculate each hidden node are also done in parallel, taking advantage of shared memory.

A grid in Fig. 1 is simply a naming convention for a group of blocks. For example, if we launch a kernel with a configuration of $\lll 10, 100 \ggg$, then we have one grid of 10 blocks where each block executes 100 threads. The arrows in Fig. 1 GPU architecture are links connecting global memory and constant memory to the threads, meaning that constant and global memory can be accessed by any thread in any block. The arrows from registers to threads mean that registers are only accessed by the thread that was assigned to it. Arrows connecting shared memory and local memory to the threads inside a block mean that the shared memory and local memory are only accessible by threads inside that block. It basically represents memory hierarchy in GPU programming.

A function that runs in the GPU is called a kernel and a kernel launches blocks on the GPU; blocks contain threads and blocks are scheduled to run on one of the GPU's streaming multiprocessors. There is a limit to the number of blocks in a streaming multiprocessor. The threads inside the block are processed by the CUDA cores, which are responsible for execution processing. Ideally, threads are expected to run at the same time in parallel, but it depends on the programmer's skill to take advantage of the hardware in hand, which is the GPU card.

A kernel is a function that runs in parallel on the GPU, executing many threads at the same time. But these threads are organized into blocks, which are simply groups of threads. For example, if we launch a kernel with the following configuration: Kernel $\lll 10, 100 \ggg$, it means we are launching a kernel in which the threads are organized into ten blocks containing 100 threads per block resulting in a total of 1000 threads being launched on the GPU. One of the reasons for organizing threads into blocks is that threads inside the same block share resources. A resource that threads inside the same block share is shared memory. For example, we can declare an array which can be shared between all threads of a block, such that each block contains a separate array which is only visible to threads inside the same block. This is the approach we used in the feedforward phase of our program. Our feedforward phase consists of two functions: *FeedForwardIH*(…) and *FeedForwardHO*(…), where *IH* in the stands for input to hidden (input layer to hidden layer) and *HO* stands for hidden to output (hidden layer to output layer).

At the beginning of our program, we allocate the input and the weights on the GPU's global memory, which is the memory that is visible to all the blocks in the GPU. Then in the *feedforwardIH* phase, we have to obtain a dot product between the weights and the inputs. The result of this dot product is the nodes of the hidden layer. The dot product is done in the following way, as narrated in Fig. 2.

Please note that for illustration purposes, less input nodes are drawn. There are in the real program 784 input nodes. Hence, the diagram is only for illustration purpose an example. There are Hidden nodes ranging from 64 to 512, typically.

Each input (I1 to I7) has a connection to a hidden node (Hidden1 to Hidden3). This connection is a weight which is a real number. Each connection has a unique weight. To calculate the value of a hidden node, we multiply each input by its weight that connects it to that hidden node we want to calculate and sum the result and apply a sigmoidal

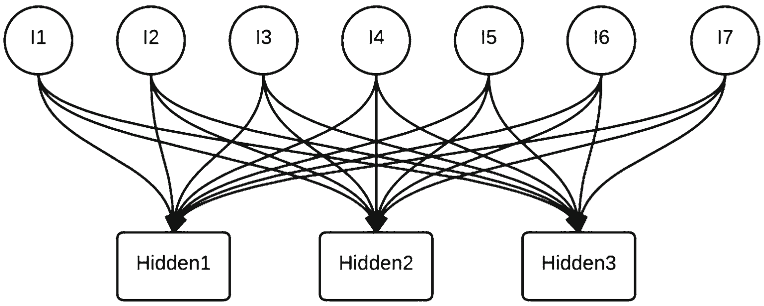


Fig. 2 Example of connections between inputs and hidden nodes

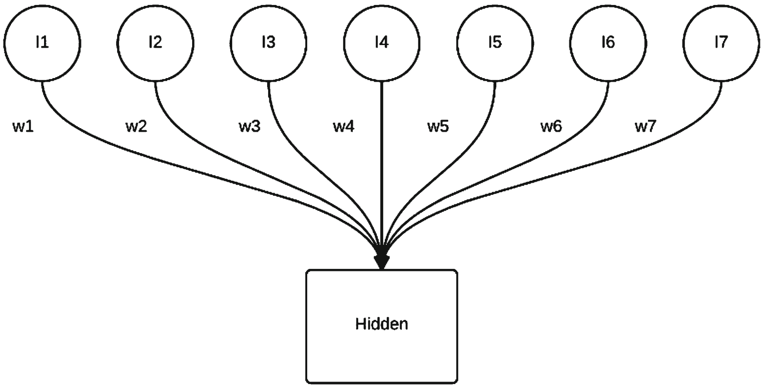


Fig. 3 Example of weights as connections between several inputs and one hidden node

function* to this sum. In the GPU, we take advantage of shared memory to do this operation. Because each block will compute a hidden node, each block has an array of shared memory, such that the threads inside each block can calculate the product of the input with the weights and store the result in a shared memory array. Then this result is summed into one single result which will be applied a sigmoidal function. This result represents a hidden node. All the hidden nodes are calculated in this way, in parallel. In the diagram above, the weights are represented by the arrow connections from the input to the hidden (the weights are all unique to each connection). So each block deals with a hidden node, as shown in a simplified example of very basic structure in Fig. 3.

The weight updates can be done in parallel, because each weight is calculated based on deltas that were calculated in a previous step and the input to that layer is used in the update process. For example, to update the weights that connect the input to the hidden layer, we add to each weight a product of the *teachingStep* (which is a constant), and *deltas* for the hidden layer and the input nodes. All these operations can be done in parallel, by launching *N* threads for each weight. For example, for the input layer to hidden layer weights, each block will update the weights that connect a hidden node to the input layers. This is done in the same way that we calculated the *feedforwardIH*, but in this case using the shared memory may not be necessary, because we just perform a direct update for each weight in parallel. The weight updates are all done in parallel for each layer.

When we run NNs on GPU, we basically copy the input nodes and the weights into the GPU and all operations are performed there in parallel. First, we initialize everything on the CPU: we read the input from a file, initialize the weights with random numbers and allocate memory in the GPU to store calculation results (hidden layer nodes, output nodes, hidden layer deltas, output layer deltas). After this is done, we copy the weights and the input to the GPU and perform all the calculations there: feedforward, back-propagation and weights updates.

When we execute the NN on the GPU, basically to avoid back and forth copies between CPU and GPU (these copies are expensive), we move everything into the GPU and only when we are done we copy the results back to the CPU. The activation function of our GPU NN is performed on the GPU. It is a sigmoid function. After performing the dot products of the feedforward phase, the result is passed through a sigmoid function, which is defined as the following, Eq. (1).

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}}. \tag{1}$$

This is done with the purpose of normalizing the result in the range from 0 to 1. The neural network operations are highly parallel. Inside each layer, the nodes are independent of each other, so this makes it very suitable for executing on the GPU.

2.2 Supervised learning in artificial neural network

An artificial neural network (ANN) is a brain-inspired model that is used to teach specific tasks to the computer. Usually, the way a computer is programmed is by giving a set of instructions on exactly what it needs to do. But the idea behind neural networks is to teach the computer how to perform a task without having to write specific instructions on the computer on how to do it. The idea is to teach the computer how to perform a task by giving it a set of examples in a continuous process called training. The training consists of a repetition where the computer tries to learn something but repeatedly makes mistakes. It learns through this mistakes, so that in the next steps it can perform better. When it has made enough mistakes and the number of mistakes becomes smaller, it means that the training has completed successfully and the computer is ready to use its newly acquired knowledge to perform the task it was taught. In Fig. 4, it shows a generalized structure of an artificial neural network (on the left) and the corresponding computation on the CUDA execution map (on the right). In particular, the intensive and parallel neuron update computation between all layers of neurons is moved to the GPU, while the overall ANN operation is controlled by the CPU part of the program.

An artificial neural network is organized in layers and each layer contains neurons. Usually, there are three or more layers; some type of ANN such as Belief Network can extend the layers into dozens or more. To generalize, the three-layer network has an input layer, output layer and multiple middle layers called hidden layers. Usually,

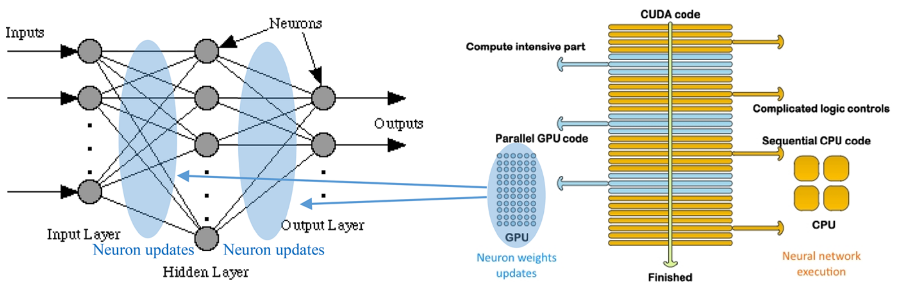


Fig. 4 The parts of ANN execution in CUDA code

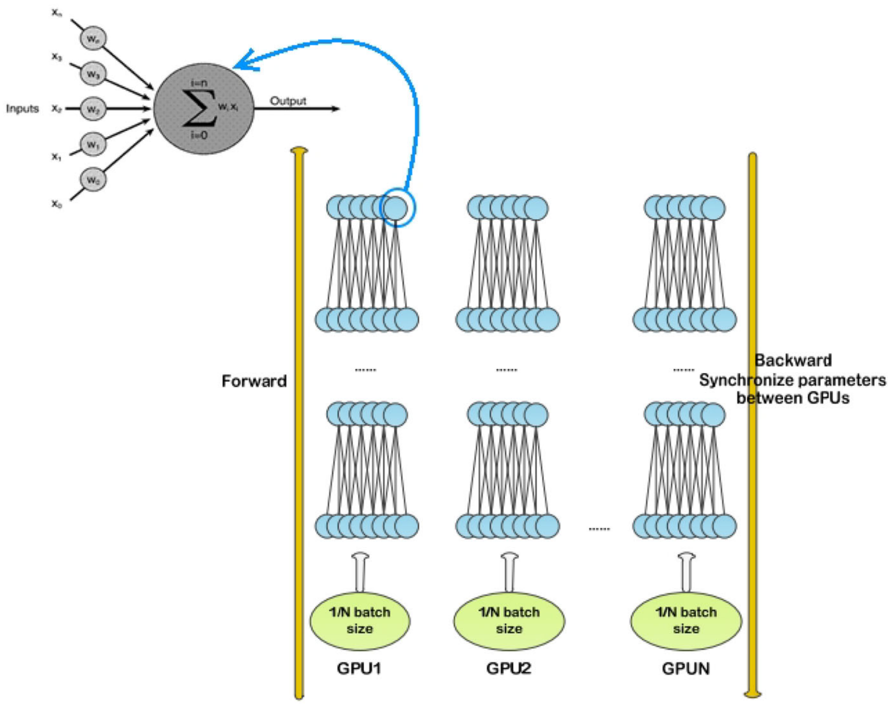


Fig. 5 The CUDA parallelism and the basic structure of neuron—activation function and neuron weight updates

the middle layers are where the neuron computation is most intensive, coded to run on GPU for speed-up gains by parallelism. The first layer is called the input layer which takes the input to produce a certain output in the output layer, and the layer between the input and the output layer is the hidden layer which does the processing of the inputs before they can produce an output. The input layer neurons are just for the purpose of holding an input. What connects the layers to each other are the weights, which are just numbers that are changed in every iteration according to the error that is committed when doing a prediction. Each neuron in the hidden and output layer takes the input that is connected to them, multiplies it by its respective weight and sum it with the rest of their inputs multiplied by their weights and then this sum is stored in the neuron. The next step is to pass the sum through an activation function. This happens in the feedforward phase where the goal is to produce an output. In the error calculation phase, the output produced is compared to the desired output and then this error is back-propagated from the output layers to the hidden layers to correct the weights that connect these layers, such that in the next iteration the error is smaller. The basic structure of a neuron is shown. As shown in Fig. 5, each of these neuron and its associated computation is made to operate in parallel on its respective GPU. Since the execution is independent, each of these computation tasks including the activation function and weight updates can be run concurrently. However, the updates with respective to layers must be synchronized, so the overall ANN updates by iteration is done in serial at the CPU.

Basically, every neuron has inputs and weights connected to them which are multiplied and summed together to produce an output which goes through an activation function. Usually, a sigmoidal function or hyperbolic tangent function is used. These computation codes are moved to GPU in our model.

3 Our proposed GPU BPNN model

We propose a method which takes advantage of the architectural characteristics of the GPU to process each phase of the BP-ANN training, so that the training time can be decreased. To show the speed-up, a popular benchmarking test dataset for training a neural network to recognition digit images is used. The handwritten digits set that we use is the MNIST dataset and we train the neural network using a collection of images for each digit as an input.

3.1 Training a BP-ANN for recognizing handwritten digits

The MNIST handwritten digits dataset [14] consists of a training image set of 60,000 images and a testing set of 10,000 images. The MNIST training set is composed of 30,000 patterns from SD-3 and 30,000 patterns from SD-1. The test set is composed of 5000 patterns from SD-3 and 5000 patterns from SD-1. The 60,000 pattern training set contains examples from approximately 250 writers. The writers from the training set and the test set were disjoint. A snapshot of the data is shown in Fig. 6 as perceived by human eyes. The digitized counterpart in multi-dimensional matrices of zeros and positive integers as shown in Fig. 7. These matrices of numbers would be used to train and test the ANNs.

The operation flow of the proposed GPU-based BPNN is depicted in Fig. 8. The steps are elaborated as follows.

1. The first step is to read the images and their labels from the input files.
2. Next, the digests of the images (as in Fig. 7) and the information of the labels are allocated to the GPU's global memory, so they can be processed by the kernels that will access them.
3. Next, the training is done in parallel in the GPU.

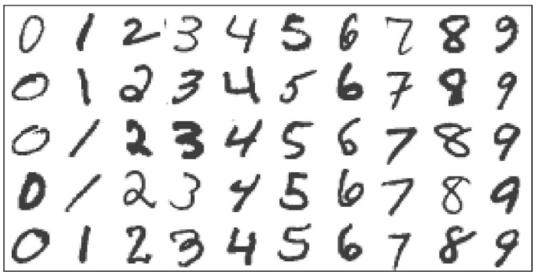


Fig. 6 A snapshot of handwritten digits in image formats

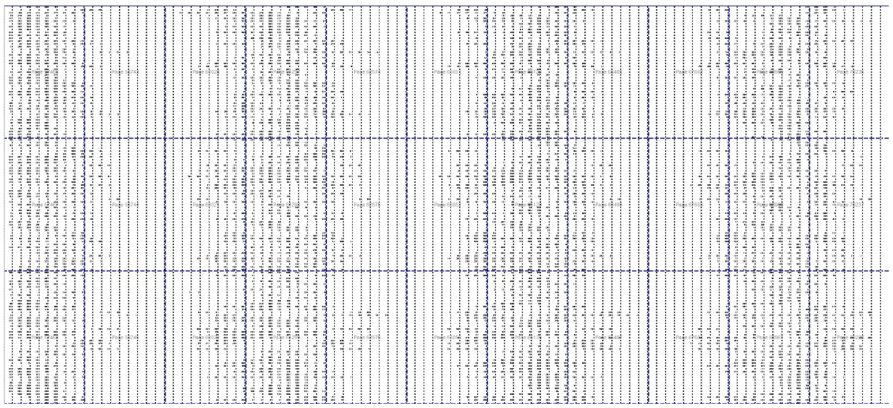


Fig. 7 The digitized representation of the handwritten digits

4. After the training has completed the results, the system is ready for testing.
5. Then the result of the test is displayed to the user.

3.2 The neural network algorithms on the GPU

The neural network training process starts with loading the images into the first layer of the network. The MNIST dataset consists of images of sizes 28 $\times$ 28 greyscale pixels, which amount to a total of 784 pixels in each round of data input. These pixels are preprocessed and normalized into binary zeros and ones, so that they can be used as input data to the neural network. In this experiment, the configuration of the neural network comprised 784 input nodes in our model in the input layer. For the hidden layer, there are 533 nodes and for the weights that connect the adjacent nodes there are a matrix of 533 $\times$ 784 elements per layer, where each row is a set of numeric weights that link the hidden nodes to the input nodes. In the feedforward phase, the image pixel value at each input is multiplied by the weight that connects it to a hidden node. Then the sum of all these multiplications are passed through an activation function, and the computed results are stored in the hidden layer nodes. This same process repeats for every node in the hidden layer.

In the CPU version where GPU is not in use at all, the execution sequence is a serial process in which every multiplication is done one at a time. So if the number of nodes

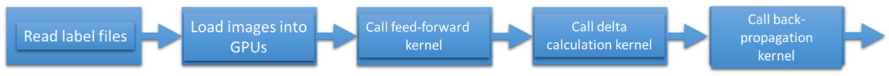


Fig. 8 The steps of BP-ANN training process in GPUs

in the hidden layer is N and the number of inputs is M, this operation would take an execution time of $O(N \times M)$. Such time consumption is due to the use of CPU in which the code runs sequentially. Under this condition, for each hidden node, we would have an FOR loop that loops through all the N elements and also another nested for loop to loop through all the input nodes, M. In the CPU version, the execution looks as follows, typically, which would be very time-consuming in execution.

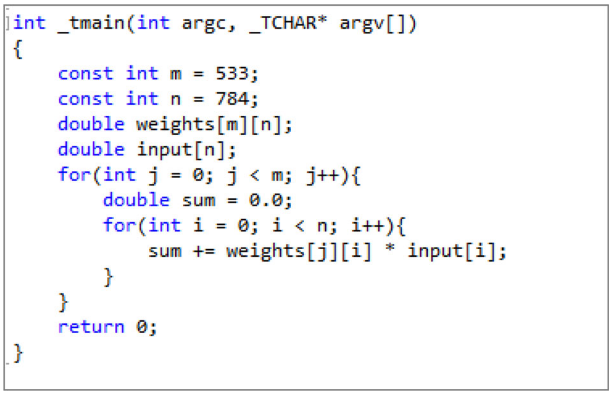
```
FOR (int$j = 0$; $j < hiddenLayerSize$; $j++$){
        FOR (int$i = 0$; $i <$ inputNodesSize; $i++$){
                // code                                    .
        }
}
```

However for the GPU version, we have a kernel that performs parallel computation such that everything is calculated in parallel using blocks of threads.

Nevertheless, the idea in using GPU acceleration is to have each thread perform a simple multiplication in the feedforward phase, such that all these multiplications are done in parallel, individually and independently at the same time, by every thread. The intensive multiplication over matrix of values is depicted in Eq. 2.

$$\sum_{j}^{m}\sum_{i}^{n} xi \times wji \begin{bmatrix} w01 & \dots & w0n \\ \dots & wji & \dots \\ wjn & \dots & wmn \end{bmatrix}. \tag{2}$$

The hidden node calculation is done according to the above equation, where we have n inputs of x, such that $[x_1, x_2, x_3, \dots, x_n]$. We have a matrix of weights that looks like the matrix in Eq. 2. This means that in serial code, we would have two nested For-loops that iterate through every element in the inputs and the weights matrix:

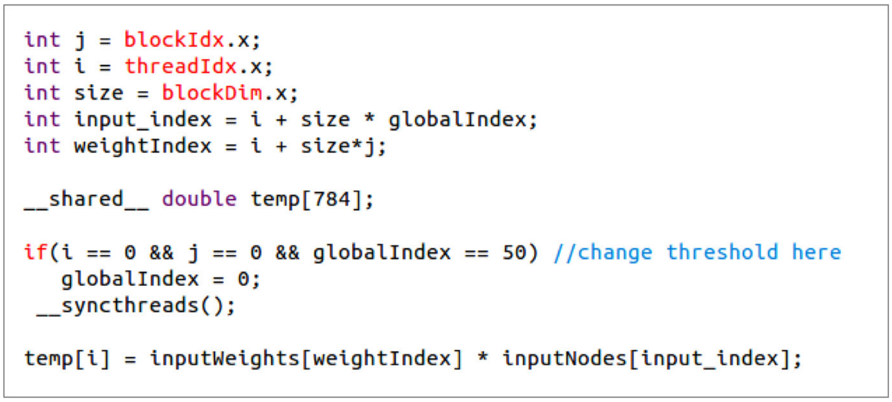


```
int _tmain(int argc, _TCHAR* argv[])
{
    const int m = 533;
    const int n = 784;
    double weights[m][n];
    double input[n];
    for(int j = 0; j < m; j++){
        double sum = 0.0;
        for(int i = 0; i < n; i++){
            sum += weights[j][i] * input[i];
        }
    }
    return 0;
}
```

This approach would take $O(n \times m)$ trend of time in CPU. But in our approach, each thread in the GPU takes care of each matrix operation at the same time in parallel.

In our implementation, to perform computation in the feedforward phase, we launch 533 blocks, each containing 784 threads. In each block, we perform the multiplication between the weights and the inputs, store the sum and pass this sum through an activation function. Finally, the result is stored in the hidden node. This approach beats other approaches in which each thread was responsible for processing one row of weights. In our approach, each thread processes a weight multiplication individually, reducing the processing time in this fashion.

Here is an example of program code from our implementation:



```
int j = blockIdx.x;
int i = threadIdx.x;
int size = blockDim.x;
int input_index = i + size * globalIndex;
int weightIndex = i + size*j;

__shared__ double temp[784];

if(i == 0 && j == 0 && globalIndex == 50) //change threshold here
   globalIndex = 0;
 __syncthreads();

temp[i] = inputWeights[weightIndex] * inputNodes[input_index];
```

By parallelizing all the weights updates, when the kernel is launched we will have controls over blocks of threads running in parallel. Each thread is responsible for a single operation between input x_i and weight w_{ji}.

The next phase is the delta error calculation in which for each node in the hidden and output layer, we try to determine the gradient such that we can use it in the weight update phase. The normal sequential approach without any aid of GPU would be looping through every node and calculating the error sequentially. But in the GPU mode, this is done in a different way. We have ten output nodes in our system for recognizing digits from 0 to 10. We could launch ten threads and calculate each one separately. Theoretically, this sounds good, but in practice it is not so straightforward, because to initialize a GPU kernel there is an overhead cost. We only want to pay for that cost if launching the same function on the CPU is much more expensive, but in this case it is not so cost-effective. Therefore, the piggyback approach we used was to directly launch a kernel that calculates both the output gradients and the hidden layer gradients. This kernel would contain 533 threads, where in the first phase some if not all these threads would be used to calculate the output gradients. In a later phase, all these threads would be used to separately calculate each of the 533 hidden nodes gradients in parallel, giving enough occupancy to the GPU so that the utilization rate would remain full and efficient. The delta calculation is done according to the following Eq. 3:

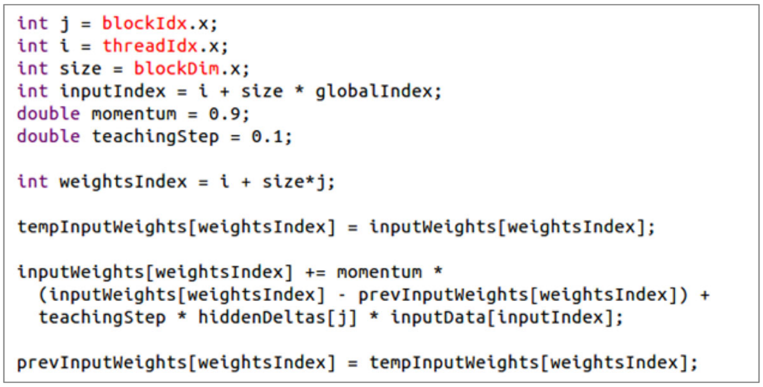
$$\begin{gathered} E(w_{ji}) = \frac{1}{2}(y_{tarj} - y_j)^2 \qquad y_j = f(a_j) = \sum_i w_{ji} x_i \\ \frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ji}} = -(y_{tarj} - y_j)x_i = -\delta x_i. \end{gathered} \tag{3}$$

So we calculate the derivatives and then change the weights by a small increment in the opposite direction to the direction of the gradient.

The last phase is the weights update phase where we would need to update each weight both in the connections between the input and hidden layer and also in the connections between the hidden and the output layer. To update the weights between the input and hidden layers, we would have to update 533 × 784 weights. To update the weights between the output and the hidden layer, we would need to update 533 × 10 weights. If we execute this serially, it certainly takes a long time. To overcome this bottleneck, we utilize two kernels that perform these operations in parallel. The kernel that updates the input weights launches 533 × 784 threads, such that each weight is updated in parallel by each thread. The kernel that updates the weights between the hidden and the output layer launches 10 × 533 threads and each weight is updated in parallel by each thread, reducing the execution time of the back-propagation algorithm. The computation for the update is done according to the following Eq. 4:

$$\Delta w = w - w_{\text{old}} = -\eta \frac{\partial E}{\partial w} = +\eta \delta x \quad \text{or} \quad w = w_{\text{old}} + \eta \delta x. \tag{4}$$

Here is the code for our weight change implementation:



```
int j = blockIdx.x;
int i = threadIdx.x;
int size = blockDim.x;
int inputIndex = i + size * globalIndex;
double momentum = 0.9;
double teachingStep = 0.1;

int weightsIndex = i + size*j;

tempInputWeights[weightsIndex] = inputWeights[weightsIndex];

inputWeights[weightsIndex] += momentum *
  (inputWeights[weightsIndex] - prevInputWeights[weightsIndex]) +
  teachingStep * hiddenDeltas[j] * inputData[inputIndex];

prevInputWeights[weightsIndex] = tempInputWeights[weightsIndex];
```

By the logics of this implementation, each thread performs an update for weight w_{ji} in parallel, reducing execution speed from a double For-loop to a single instruction per thread.

The workflow of a GPU-enabled BP-ANN is summarized into a list of the following steps:

1. Read the image data and label data.
2. Initialize the weights randomly.
3. Copy the weights to the GPU.
4. Copy the input to the GPU.
5. Initialize the neural network.
6. Call the feedforward kernel (input to hidden layer)—533 $\times$ 784 threads.
7. Call the feedforward kernel (hidden to output layer)—10 $\times$ 533 threads.
8. Call the kernel to calculate the deltas—533 threads.
9. Call the kernel to update the weights (input to hidden)—533 $\times$ 784 threads.
10. Call the kernel to update the weights (hidden to output)—533 $\times$ 10 threads.

The back-propagation part is implemented on the GPU. The following diagram shows what is implemented in the GPU NN; it shows essentially the workflow as above and in Fig. 8 in relation to the transfers between CPU and GPU (Fig. 9).

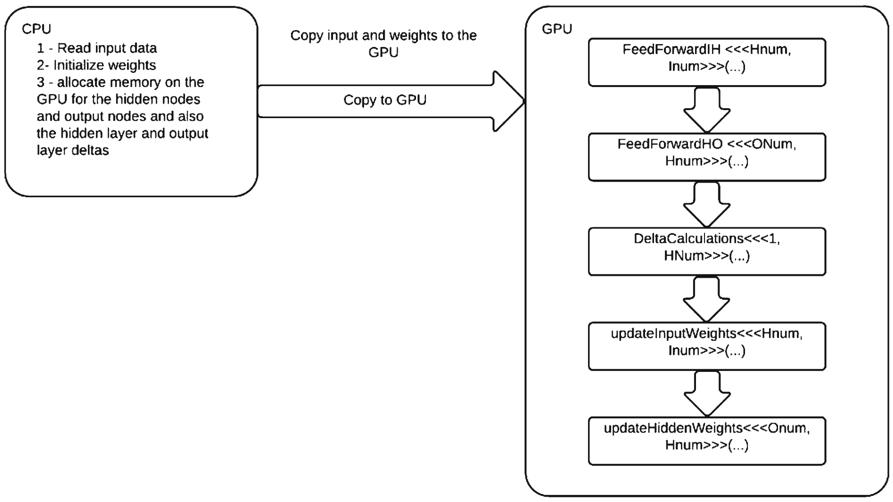

CPU
1 - Read input data
2- Initialize weights
3 - allocate memory on the GPU for the hidden nodes and output nodes and also the hidden layer and output layer deltas
Copy input and weights to the GPU
Copy to GPU
GPU
FeedForwardIH <<<Hnum, Inum>>>(...)
FeedForwardHO <<<ONum, Hnum>>>(...)
DeltaCalculations<<<1, HNum>>>(...)
updateInputWeights<<<Hnum, Inum>>>(...)
updateHiddenWeights<<<Onum, Hnum>>>(...)

Fig. 9 Porting the flow of execution from CPU to GPU

The above diagram shows the whole program execution flow. The feedforward process has two kernels, one for calculating the hidden nodes (*FeedForwardIH*) and another for calculating the output nodes (*FeedForwardHO*). The back-propagation phase consists of the following three functions:

1. DeltaCalculation—This function calculates the errors in the output layer and hidden layers, so we call it with 1 block containing threads equal to the number of nodes in the hidden layer, and we use these threads to calculate the errors (called *deltas*) in the hidden and output layers. These deltas will be used to update the weights in the layers of the GPU-enabled BP-ANN.
2. UpdateInputWeights—This function updates the weights that connect the input layer to the hidden layer. So we launch a number of blocks equal to the number of hidden nodes, and each block has a number of threads equal to the number of input nodes. Each block is executed in parallel and inside each block, the threads update the weights in parallel.
3. UpdateHiddenWeighs—This function updates the weights that connect the hidden layer to the output layer. So we launch a number of blocks equal to the number of output nodes, and each block has a number of threads equal to the number of hidden nodes. Each block is executed in parallel, and inside each block the threads update the weights in parallel.

For the topology of the NN, several configurations are implemented. There are 784 input nodes, 10 output nodes, and four configurations of hidden nodes amounting to 64, 128, 256 and 512. We tested the neural network for different topologies as reported in the next section of the paper. The ten output nodes correspond to the specific testing case where digits from 0 to 9 are used in the recognition test.

4 Experiment results

For both the CPU and GPU versions of the handwritten recognition software programs implemented using BP-ANN, we conduct a fair and equal comparison. That means the same topology, same configuration and same random seeds were used for both versions of BP-ANN, according to the following table.

We experimented different population sizes and different number of epochs to compare the speeds of the two versions of ANN implemented for CPU and GPU. An epoch is basically defined as the runtime in which the neural network goes through all the population for one round. In the context of training a neural network for pattern recognition, an epoch means the number of times that the program loops through the training data. Our training data has 60,000 elements and an epoch means looping through all of the 60,000 elements, e.g., ten epochs mean that we have looped through the training data (60,000 elements) ten times repeatedly. Our test data contain 10,000 elements. Our hardware is basically a computer running Ubuntu (Linux OS), with a NVDIA GEFORCE GTX 750 GPU card. The weights are initialized at random values ranging from −0.25 to 0.25, the teaching step is 0.01 and the momentum is 0.9 and are shown in Table 1.

For comparing the computing powers between NN that is implemented purely on CPU and NN that runs on GPU, we run epochs from 100 to 10,000. The following runtimes are recorded as in Table 2, as a speed test.

A graph is given to show the difference between the CPU and GPU execution speeds in Fig. 10.

In the *X*-axis of the chart in Fig. 10, we have the number of epochs and in the *Y*-axis we have the runtime in seconds. As seen from the graph in Fig. 10 and Table 2, as we increase the number of epochs and the population size, the GPU's performance

Table 1 Neural network configuration

	CPU	GPU
Weights interval	[−1, 1]	[−1, 1]
Learning rate	0.1	0.1
Momentum	0.9	0.9
Topology	784, 533, 10	784, 533, 10

Table 2 Computation times

Population	Epochs	CPU time in seconds	GPU time in seconds
10	100	5.8687	0.283682
20	1000	117.362351	2.674261
30	1000	177.518307	2.646332
40	1000	236.354287	2.677458
50	1000	295.3512	2.675260
60	10000	3552.425203	26.610284

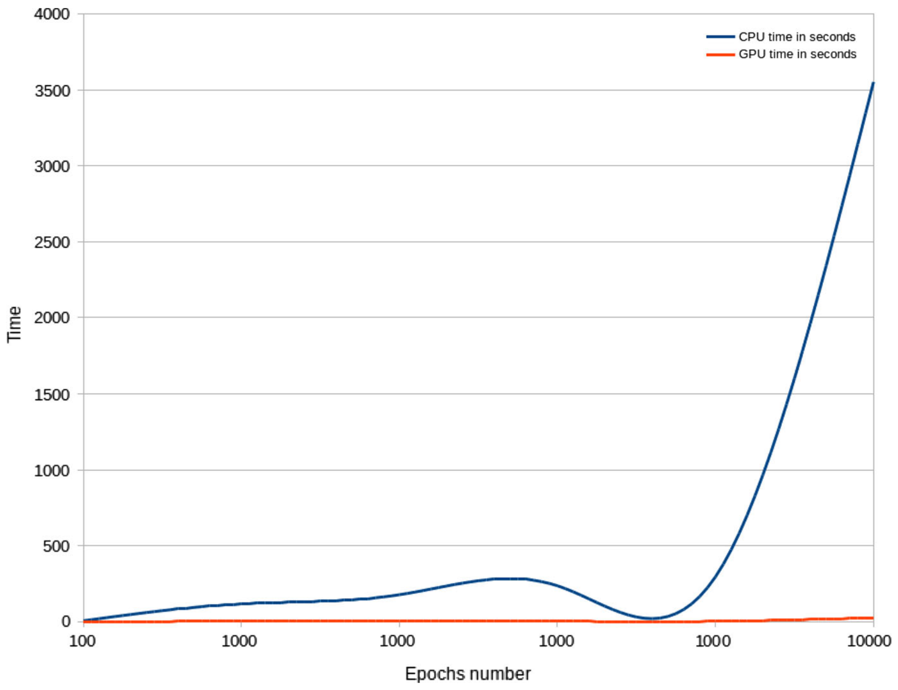


Fig. 10 Computation times between two versions of BP-ANN

beats the CPU's performance by multitudes. The steep increase for the CPU version of BP-ANN implies that when the epoch reaches 10,000, the increasing rate of time becomes exponential. This is significant because when the computation time of the CPU version of BP-ANN turns prohibitively large, the GPU version is still efficient.

Performance wise, in terms of prediction accuracy, both CPU versions and GPU versions of BP-ANN attain very high accuracy. (Accuracy is defined here by simply the number of correctly recognized digits over the total number of testing digits.) Using the training dataset of MNIST that contains 60,000 instances and the testing dataset of the same with 10,000, our neural network achieves quite a high accuracy rate at a relatively short training time by our GPU-enabled version of BP-ANN. Our speed test results are compared with the best ever achieved results by the other state of arts as reported recently in [15], which attempted to solve the same problem. Table 3 is shown as a performance reference for the purpose of comparing our accuracy and training time to others.

Our GPU-enabled BP-ANN in 30 epoches can achieve as high as 98.06 % accuracy, at a training time of 1192.3 s. By ranking the speed gain which is accuracy over time, our GPUenabled BP-ANN outperforms the deep learning networks in speed gain; it is beaten only by the category of extreme learning machines.

In the experiment, we run the GPU-enabled BP-ANN, while the same sizes of input and output nodes (784 and 10, respectively) are kept constant. Different performance results for different number of hidden layer nodes are obtained. The results are sum-

Table 3 Performance comparison of GPU-enabled BP-ANN with state-of-the-art deep networks [15]

State-of-arts NN algorithms	Training time (s)	Testing accuracy (%)	Speed gain
Multi-layer extreme learning machine	444.655	99.03	0.22271199
Extreme learning machine	545.95	97.39	0.178386299
ELM Gaussian kernel	790.96	98.75	0.124848286
GPU enabled BP-ANN (with 30 epoches)	1192.3	98.06	0.082246948
Deep belief network	20580	98.87	0.004804179
Deep Boltzmann machine	68246	99.05	0.001451367
Stacked auto-encoder	N/A	98.6	N/A
Stacked eenoising auto-encoder	N/A	98.72	N/A

marized in Table 4. Some interesting results are 97.26 % accuracy in just 306.507 s, which is fast. It can reach 95.69 % accuracy in just 39.798050 s, in a relatively short time, suitable for time-critical applications.

A major advantage of our GPU-enabled BP-ANN is simplicity. There is no need to build complex structures like ELM or deep learning networks with many layers to implement a highly accurate NN and their implementations take much more time than ours.

There is another interesting observation. If we have a small neural network such that the number of inputs in the input layer and the number of hidden and output nodes are small, then it will not be efficient to run it on the GPU. This is because initializing data on the GPU, allocating space and performing memory copies incur certain overhead. If this overhead plus the GPU execution time is higher than the CPU execution time, then it is not recommended to use the GPU for that particular problem, because it would perform slower than the CPU.

5 Conclusion

From the experiments reported in this paper, we showed how we can train a simple neural network using the GPU, as we can improve the speed by a really high factor in comparison to the CPU version of neural network. We contribute to the design of BP-ANN using GPU acceleration at the programming code level. It is observed that the GPU should only be used if the data to be processed has enough attributes to benefit from a lot of parallelism from the GPU. Otherwise, the cost of initializing the GPU would be too expensive and using the GPU would not justify the overhead of the costs of data transfer. So, if we need to do training on a dataset with a small number of attributes, the CPU version is better. However for training with data that carry many attributes, GPU-enabled BP-ANN is a recommended choice. By comparing our GPU-enabled BP-ANN side by side with other advanced neural networks, our model is relatively simple, capable to achieves similar accuracy rate at the cost of moderate training time.

Table 4 Performance of our GPU-enabled BP-ANN in different configurations of hidden nodes

Complexity	1-epoches			15-epoches			30-epoches		
#Hidden nodes	Training time (s)	Testing accuracy (%)	Speed gain	Training time (s)	Testing accuracy (%)	Speed gain	Training time (s)	Testing accuracy (%)	Speed gain
64	5.517597	92.65	16.7917302	82.483551	96.04	1.16435336	164.839182	95.78	0.58105117
128	10.531344	94.64	8.98650733	157.655788	96.82	0.61412271	316.412874	97.1	0.30687753
256	20.395331	94.33	4.62507816	306.506776	97.26	0.31731762	615.546697	97.81	0.1588994
521	39.79805	95.69	2.40438916	596.06984	97.93	0.16429283	1192.263082	98.06	0.08224695

Acknowledgments The authors are thankful for the financial support from the research grant "Peer-production approaches to e-Learning (PPAeL)," Grant No. FDCT 019/2011/A1, offered by Macau Fundo para o Desenvolvimento das Ciências e da Tecnologia.

References

1. Park SI, Ponce SP, Huang J, Cao Y, Quek F (2008) Low-cost, high-speed computer vision using NVIDIA's CUDA architecture. In: 37th IEEE applied imagery pattern recognition workshop, pp 1–7
2. NVidia CUDA Zone. http://www.nvidia.com/object/cuda_home.html
3. Steinkrau D, Simard PY, Buck I (2013) Using GPUs for machine learning algorithms. In: 12th International conference on document analysis and recognition, pp 1115–1119
4. Lopez-Fandino J, Heras DB, Arguello F (2014) Efficient classification of hyperspectral images on commodity GPUs using ELM-based techniques. In: Conference PDPTA'14, CSREA Press, July 21–24, pp 1–13
5. Catanzaro B, Sundaram N, Keutzer K (2008) Fast support vector machine training and classification on graphics processors. In: Proceedings of the 25th international conference on machine learning (ICML 2008), Helsinki, Finland, pp 104–111
6. van Heeswijk M, Miche Y, Lindh-Knuutila T, Hilbers P, Honkela T, Oja E, Lendasse A (2009) Adaptive ensemble models of extreme learning machines for time series prediction. In: 19th International conference on artificial neural networks, Limassol, Cyprus, 9
7. Neural Networks on the GPU. http://leenissen.dk/fann/html_latest/files2/gpu-txt.html
8. Neural Networks with Parallel and GPU Computing. http://www.mathworks.com/help/nnet/ug/neural-networks-with-parallel-and-gpu-computing.html
9. A Neural Network on GPU. http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU
10. Huang G-B, Chen L, Siew C-K (2006) Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE TNN 17(4):879–892
11. Hayashi A, Ishizaki K, Koblents G, Sarkar V (2015) Machine-learning-based performance heuristics for runtime CPU/GPU selection. In: Proceedings of the principles and practices of programming on the Java platform, pp 27–36
12. Ribeiro B, Goncalves J (2012) Restricted Boltzmann machines and deep belief networks on multi-core processors. In: The 2012 international joint conference on neural networks (IJCNN), 10–15 June 2012, pp 1–7
13. Huqqani AA, Schikuta E, Ye S, Chen P (2013) Multicore and GPU parallelization of neural networks for face recognition. In: International conference on computational science, ICCS 2013, Procedia Computer Science, vol 18, pp 349–358
14. LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324
15. Cambria et al (2013) Extreme learning machines. IEEE Trans Cybern 28(6):30–59