IJCSE International Journal of Computer Sciences and Engineering Open Access

Research Paper Volume-5, Issue-10 E-ISSN: 2347-2693

Parallel Implementation of Gradient Descent Algorithm for Backpropagation Networks

K. Devkota[1*], P. Bhattarai[2]

[1*]Department of Electronics and Computer Engineering, Institute of Engineering, Tribhuvan University, Kathmandu, Nepal
[2]Department of Electronics and Computer Engineering, Institute of Engineering, Tribhuvan University, Kathmandu, Nepal

[*]*Corresponding Author: kdkapildevkota@gmail.com, Tel.: +977-9843002224*

Available online at: www.ijcseonline.org

Received: 28/Sep/2017, Revised: 07/Oct/2017, Accepted: 23/Oct/2017, Published: 30/Oct/2017

***Abstract*—** The problem of computational efficiency in adaptive algorithms, which is current and pressing, can be solved through their implementation in parallel frameworks, like CUDA, OpenCL, etc. The approach taken to parallelize any complex operation requires its separation into several distinct and independent sub-operations. We employed the same procedure to parallelize the BP (or Backpropagation) network algorithm. The function breakdown of the BP network involved breaking its overall operation into Feed-forward and Back-Propagate sub-operations, which was further divided into smaller independent execution groups. We applied parallel constructs on those independent execution groups and used the MNIST dataset to compare the algorithm's performance with respect to the sequential algorithm. Comparing their performances, we found that the efficiency of the algorithm depended on the size of the BP network. In the large network with massive number of weight connections, we saw a significant improvement in the convergence time. This makes our algorithm preferable in feedforward networks having large number of hidden layers, neurons and weight connections.

***Keywords*—**Backpropagation, Supervised Learning, CUDA, parallel

I. INTRODUCTION

As the volume and dimensionality of data increases, more computational power is needed to extract relevant information from it. There are primarily two ways to adapt with the inescapable increment in data dimensions over time; one is to make the algorithm more optimized and scalable to larger datasets, other is to exploit the hardware innovations to improve the algorithms performance. By parallelizing an algorithm, we are striving for the latter goal. In parallel computation of algorithms, many calculations or the execution of processes are carried out simultaneously [1]. This can be exploited by BP networks, which are highly flexible to parallel optimizations and can result in a significant performance improvement.

The idea behind Backpropagation algorithm was to find a way to train a multi-layered neural network such that it can learn the appropriate internal representations to allow it to learn any mapping of input to output [2].So, the Backpropagation network, like other supervised learning methods, requires additional training datasets to construct the approximation of the system under consideration. So, the size of datasets is imperative to the performance of the network, as large size datasets may require bigger, multilevel BPNs to converge. That is why the parallelization of the network is an appealing option for performance enhancement through the reduction of convergence time. This method however doesn't reduce the convergence rate of the BP network. The convergence rate of the BP network is the inverse of the average number of times, the network parameters have to be updated for the error to be reduced to a tolerable amount. In order to improve performance by increasing the convergence rate, methods like self-adaptive learning rate technique [4], heavy ball method, etc. have to be pursued.

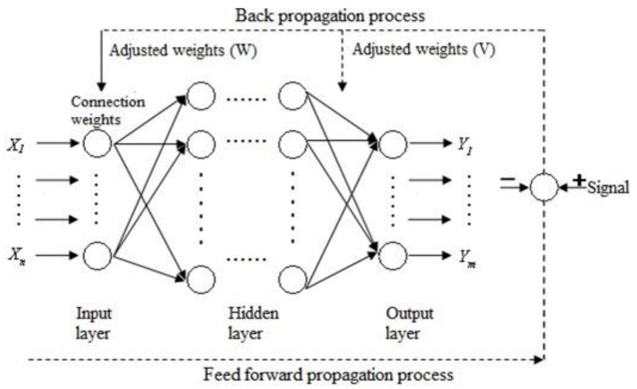

Figure 1: BP artificial neural network framework [3]

The goal of parallel implementation of Backpropagation Network can be accomplished through many available parallel computational methods. These methods may require additional hardware components to do the parallel

computations. Using dedicated hardware to do the machine learning typically ends up in disaster because of cost, obsolescence, and poor soft-ware. As real-time processing is also an added requirement for many neural network applications, fully parallel specially designed hardware implementations, such as an FPGA-based realization of a neural network may also sound appealing. But, it is somewhat expensive and involves extra design overheads. [5] So, GPU based parallelization are more prefer-able because of their availability and ease of use. Graphic Processing Units (GPUs), now available on every PC, provides an attractive alternative which yield speedup in both training and testing in Machine Learning algorithms. [6]
The simplest approach to train a BP network using gradient information, in order to update network parameters is the Gradient descent optimization method [7]. Gradient descent optimization requires a sequential flow of weight and bias values from one level to another, so it cannot be fully parallelized across levels. But, by implementing parallelization within levels, we can get a significant performance improvement. In this paper, we try to explore a generic understanding of BPNs and adjust it, so as to make it more suitable for parallelization.

II. BACKPROPAGATION NETWORK

Backpropagation network can be visualized as a collection of levels, each having a finite number of nodes and sequenced in such a way that every adjacent levels have their nodes fully connected to each other. Figure 1 shows the diagram of a conventional BPN, with a detailing representation of levels, nodes, weights and biases.

The complete definition of a BP network requires the unique identification of all network parameters, including weights, biases and neuronal outputs. The neurons (or nodes) in the network can be uniquely addressed by the layer in which the neuron is positioned and the position of the neuron within that layer. For example, we can take the outermost layer (i.e. output) as layer 0, and increment the address as we progress towards the input layer. Similarly, the weights can be uniquely addressed between the two nodes of the adjacent layer by addressing the weight by the address of layer (towards the output), the position of node in the outer layer and the position of the node in the inner layer. This gives a three dimensional identification of weights in the network.

The Backpropagation algorithm begins by assigning random non-zero values to every weights and biases. The feedforward part of the algorithm takes input values from the input layer and forwards it to hidden layers. The result propagates from layer to layer by computing the threshold of the weighted addition of intermediate values form the previous layers. Each layer may have a distinct threshold function. The threshold functions can have an effect on the speed and the convergence of the network. The continual application of the weighted summation followed by threshold computation of the known intermediate values form one layer to obtain the unknown intermediate values of the other layer, the output values can be obtained in the output layer.
Since feedforward network is a supervised learning network, the input vector is accompanied by a target vector. To optimize the value of weights so that the output vector obtained from the output layer converges to the target vectors, the error values are transferred inwards, towards the input layer. As the error gradient propagates inwards, the corresponding weights between layers get updated.

Suppose in an *n*-layered neural network, for *p*-input vectors, the output should converge to *q*-output vectors *T*. During feedforward computation, we can assume the output vector as. Let the weights between the nodes in the network be rep-resented as w_{ijk}, the *i*-index specifying the outward layer of the weight connection (facing the output layer), *j*-index representing the position of the outward node at the outward layer, and *k*-index identifying the position of the inward node at the inward layer. For the sake of convenience, the output layer is indexed as 0^{th} layer and the index increments by 1 as we progress towards the input layer. Similarly, output of every neurons can be represented as z_{ij}, where the *i*-index identifies the layer of the neuron, and the *j*-index represents the position inside the layer in which the neuron is present. The same procedure can be followed to index the biases of neurons in the network b_{ij}.

Each connection input to a neuron has an associated weight. In a BPN network, a neuron has input connections only with neurons of the previous layer. The neuronal output is the weighted summation of outputs from the adjacent previous layer, followed by a functional transformation. The transforming function is often called a threshold function, as it often limits the summed value between two extreme values. So, the neuronal output can be computed as:

$$a_{ij} = \sum_k w_{ijk}\, z_{i+1,k} + b_{ij} \qquad (1)$$

$$z_{ij} = F(a_{ij}) \qquad (2)$$

Where, a_{ij} is the weighted summation of outputs from nodes from previous layers, and *F* is the threshold function. We use gradient descent algorithm to converge the network to the global minima. The global minima correspond to the point of minimum error at the output. The error value (*E*) of the network can thus be mathematically described as:

$$E = \frac{1}{2}\sum_{i=0}^{n}(t_i - z_{0i})^2 \qquad (3)$$

Where, t_i is the target value at the i^{th} neuron of the output layer, and z_{ij} is the computed output value at the same neuron. In gradient descent optimization, we compute the gradient of the global error of the system with the parameter to be optimized. If the system has a global error E, and the optimization parameter is η, then the gradient of E by η is actually its partial derivative with respect to η. To update the value of η, so that the error minimizes to a maxima, we subtract to η, some part of its gradient. So the updated parameter will be:

$$\eta_{new} = \eta_{old} - \alpha\frac{\partial E}{\partial \eta} \qquad (4)$$

The updated η becomes closer to the nearest minima, regardless of whether the minima is local or global. This variation of gradient descent optimization is called the steepest descent method, as we can show that the steepest direction of η is $\nabla\eta$ [8]. The steepest descent method is surprisingly a less effective method for global minima search, and the implementation of heuristics like momentum and delta-bar-delta procedures can significantly improves the search performance. [9] But nevertheless, in a system with single minima, this optimization algorithm always converges to the global minima. So, implementing the gradient descent method of optimization and the parameters to optimize are weight connections between the nodes and the bias values of the nodes of the entire network. Then, the updated weights and biases become:

$$w_{ijk}(updated) = w_{ijk} - \alpha\frac{\partial E}{\partial w_{ijk}} \qquad (5)$$

$$b_{ij}(updated) = b_{ij} - \alpha\frac{\partial E}{\partial w_{ijk}} \qquad (6)$$

So, the prime focus of the weight and bias update process in gradient descent algorithm for BPN network is the computation of the partial derivative of the network error with weights and biases at different layers. We can reduce the complexity of the computation by introducing a new parameter δ_{ij}. This parameter is actually the partial derivative of the global error with respect to a_{ij}.

$$\delta_{ij} = \frac{\partial E}{\partial a_{ij}} \qquad (7)$$

So,

$$w_{ijk}(updated) = w_{ijk} - \alpha.\delta_{ij}.\frac{\partial a_{ij}}{\partial w_{ijk}} \qquad (8)$$

$$b_{ij}(updated) = b_{ij} - \alpha.\delta_{ij}.\frac{\partial a_{ij}}{\partial b_{ij}} \qquad (8)$$

$$\frac{\partial a_{ij}}{\partial w_{ijk}} = z_{i+1,k}, \frac{\partial a_{ij}}{\partial b_{ij}} = 1 \qquad (10)$$

So the equation becomes:

$$w_{ijk}(updated) = w_{ijk} - \alpha.\delta_{ij}.z_{i+1,k} \qquad (11)$$

$$b_{ij}(updated) = b_{ij} - \alpha.\delta_{ij} \qquad (12)$$

The value of δ_{ij} should be propagated backwards, from output layer to the input layer. This backwards propagation of δ values is why the network is called Backpropagation Network.

$$\delta_{ij} = \frac{\partial E}{\partial a_{ij}} = \sum_k \frac{\partial E}{\partial a_{i-1,k}}\frac{\partial a_{i-1,k}}{\partial a_{ij}} \qquad (13)$$

$$\delta_{ij} = \sum_k \delta_{i-1,k}\frac{\partial a_{i-1,k}}{\partial a_{ij}} \qquad (14)$$

From (10),

$$\frac{\partial a_{i-1,k}}{\partial z_{ij}} = w_{i-1,kj}.F(a_{ij}) \qquad (15)$$

So,

$$\delta_{ij} = F(a_{ij}).\sum_k(\delta_{i-1,k}w_{i-1,kj}) \qquad (16)$$

At the output layer:

$$\delta_{0i} = a_{0i}(z_{0i} - t_i) \qquad (17)$$

Because we already know the values of δ for the output units, it follows that by recursively applying equation (16), we can evaluate the δ for all the hidden units in a feed-forward network, regardless of its topology.

The BPN training algorithm so constructed is:

1) Given input vector, forward propagate the neuronal output from layer to layer, until the output becomes available at the output layer.

2) Compute δ_{0i} at the output neurons, and propagate the δ-values backwards, until we reach the input layer.

3) Provided the training rate α, update w_{ijk} and b_{ij} as:

$$w_{ijk}(updated) = w_{ijk} - \alpha.\delta_{ij}.z_{i+1,k}$$

$$b_{ij}(updated) = b_{ij} - \alpha.\delta_{ij}$$

4) After all the weights and biases are updated, return to 1, until the global error is inside the tolerance zone.

III. PARALLEL IMPLEMENTATION

In Single Instruction Multiple Data (SIMD) parallelism, which is a class of parallel architecture in Flynn's taxonomy [10], a common set of instructions acts on multiple data, and modern day GPUs employ this form of parallel architecture. The basic implementation of SIMD involves the programming of an instruction set, which are then copied to multiple SIMD processors for independent and simultaneous execution. CUDA calls this programming construct, a kernel function. So, we need to implement the BP network as a set of so-called kernel functions, to be executed in the parallel cores of a multi-core SIMD processor. We can conveniently break down the complete BPN operations into three independent parts;
Forward Propagation of Data, Reverse Propagation of Delta Values, and Weight and Bias Update operations. Parallelism in the first two operations can be realized within layers, but the weight and bias update of the complete network can be performed simultaneously, which results in a significant improvement in computation time. Further descriptions of parallel implementation of the abovementioned BPN operations are described as:

A. *Forward Propagation of Data*

The forward propagation process computes the neuronal outputs from layer to layer. So the process is parallel within a layer, but we cannot simultaneously compute the neuronal outputs of nodes belonging to separate layers, since the output of a neuron from a layer depends on the outputs of the nodes from the preceding layers. So the kernel size should be the size of the layer under consideration. The kernel algorithm for the forward propagation is shown below:

```
Input: $l, n, i, prev, z_{n+1,j}, b_{ni}, w_{nij}, ip_i$
                                        $0 \leq j < prev$
Output: $z_{n,i}, a_{n,i}$
 1: if $(i = l)$ then                   # input layer
 2:     $a_{n,i} \leftarrow ip_i$
 3:     $z_{n,i} \leftarrow ip_i$
 4: else                                # hidden or output layer
 5:     $a_{n,i} \leftarrow 0$
 6:     for $j = 0$ to $prev - 1$ do
 7:         $a_{ni} \leftarrow a_{ni} + w_{nij} \cdot z_{n+1,j}$
 8:     end for
 9:     $a_{ni} \leftarrow a_{ni} + b_{ni}$
10:     $z_{ni} \leftarrow thres(a_{ni})$
11: end if
12: return $z_{ni}, a_{ni}$
```

Algorithm 1 Algorithm for computation of forward values at node *i* for layer *n*, for the network with *l* layers

The pseudo code gets the *a*-value by performing the weighted summation of *z*-values from the preceding layer, and apply threshold function, represented by *thres* in the pseudocode, to obtain the *z*-value.

B. *Reverse Propagation of Delta Values*

This process, similar to forward propagation process, is parallel within a layer, since the delta values of a layer is dependent on the delta values of the layer immediately succeeding it. So, like forward propagation, the size of the kernel should be equal to the size of the layer under

Algorithm 4 Algorithm for the update of bias values of node *i* of layer *n*.

consideration. The kernel algorithm for reverse propagation is:

```
Input: $n, i, next, a_{ni}, z_{ni}, d_{n-1,j}, t_i, w_{n-1,ij}$
                                        $0 \leq j < prev$
Output: $d_{ni}$
 1: if $(i = 0)$ then                   # output layer
 2:     $d_{ni} \leftarrow (z_{ni} - t_i) \cdot d\_thres(a_{ni})$
 3: else                                # hidden or input layer
 4:     $d_{ni} \leftarrow 0$
 5:     for $j = 0$ to $next - 1$ do
 6:         $d_{ni} \leftarrow d_{ni} + w_{n+1,ji} \cdot d_{n-1,j}$
 7:     end for
 8:     $d_{ni} \leftarrow d_{ni} \cdot d\_thres(a_{ni})$
 9: end if
10: return $d_{ni}$
```

Algorithm 2 Algorithm for computation of reverse values at node *i* for layer *n*, for the network with *l* layers

In the pseudo code, the computation of the δ-value of a node starts by performing the weighted summation of δ-values from the succeeding nodes. The *d_thres* function represents the derivative of the threshold function of the layer.

C. *Weight and Bias Update*

After we obtain the neuronal output and delta value at every node of the network, we can begin the weight update process. Since, weight update of a weight connection is independent of the weight update at any other weight connection; the number of the kernel threads is equal to the total number of weight connections of the network. The weight update algorithm is shown below:

```
Input: $w_{nij}, d_{ni}, z_{n+1,j}, rate$
Output: $w_{nij}$
 1: $w_{nij} \leftarrow w_{nij} + rate \cdot d_{ni} \cdot z_{n+1,j}$
 2: return $w_{nij}$
```

Algorithm 3 Algorithm for the update of weight connection between node *i* of layer *n* and node *j* of layer *n+1*.

Similarly, for updating bias parameters, the algorithm is:

IV. PERFORMANCE ANALYSIS

The testing of our algorithm needed an effective measure of comparison between the parallel and non-parallel codes and a suitable hardware configuration to test the codes on. To measure the effectiveness of the parallel code, we decided to run the code in the worst-case scenario, which was to run the

```
**Input:** $b_{ni}, d_{ni}, rate$
**Output:** $b_{ni}$
  1: $b_{ni} \leftarrow b_{ni} + rate \cdot d_{ni}$
  2: **return** $b_{ni}$
```

tests in a rig with relatively powerful CPU and a middle to low-tier GPU. A good performance measure at that scenario would certainly imply an even better performance when high-end, state-of-the-arc GPUs are available. So, we used a computer with mobile NVIDIA GeForce 84OM GPU with 384 shader cores, clocked at 129 MHz as our GPU to run the parallelized code, and an Intel Core i7-5600U CPU @ 260GHz, 2594 MHz as our CPU to run the sequential version of the code. There was also the issue of developing the internal architecture of the network, which involves deciding the number of nodes within each layer of the network. Since, a large of independent parameters while building the network may cause difficulty in performance analysis, we need to reduce the number of independent network parameters to smallest possible value. A reasonable way to represent a network can be to use four parameters; the size of Input and Output layers, the number of nodes in the network and the total number of layers the network has. We can populate hidden layers in the network by assigning each layer certain amount of nodes that would maximize the total weight connections in the network. By maximizing the number of weight connections in a network, we are trying to increase the number of computations during forward and reverse propagation, as the number of weight connections often correlates with the complexity of computation. This certainly aids in our analysis, as we want to know the behaviour of the GPU-code compared to the CPU-code when the size of computation is large. To determine the node count of each hidden layers for maximum possible number of weight connections, the following strategy was used. Let the number of layers in the network be l and the number of nodes at layer *i* ben_i^2, which is squared to ensure positiveness. Then,

$$N = \sum_{i=1}^{l} {n_i}^2 \qquad (18)$$

If *I* is the required number of nodes at the output layer, and *O* at the output layer,

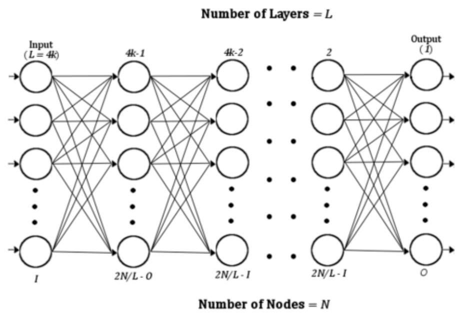

Figure 2. BP network configuration having maximum number of Weight connections

$$N = \sum_{i=2}^{l-1} {n_i}^2 + I + O \qquad (19)$$

The total number of network weights can be computed as:

$$W = \sum_{i=0}^{l-1} {n_i}^2 . {n_{i+1}}^2 \qquad (20)$$

Then, to maximize the weight-count function, given in (20) while being constrained to (19), we create the Langrangian function, given in (21).

$$L = W + \lambda\left(\sum_{i=2}^{n-1} {n_i}^2 + I + O - N\right) \qquad (21)$$

As we partially differentiate with respect to L, we get:

$$\begin{gathered} {n_1}^2 = I \\ {n_1}^2 + {n_3}^2 = \lambda \\ \ldots.. \\ \ldots.. \\ \ldots.. \\ {n_{l-2}}^2 + {n_l}^2 = \lambda \\ {n_l}^2 = O \end{gathered} \qquad (22)$$

From (22), we get:

$$\begin{gathered} {n_3}^2 = \lambda - I \\ {n_5}^2 = I \\ \ldots.. \\ \ldots.. \\ \ldots.. \\ {n_{l-4}}^2 = O \\ {n_{l-2}}^2 = \lambda - O \end{gathered} \qquad (23)$$

If the number of layers (*l*) is a multiple of 4, λ becomes:

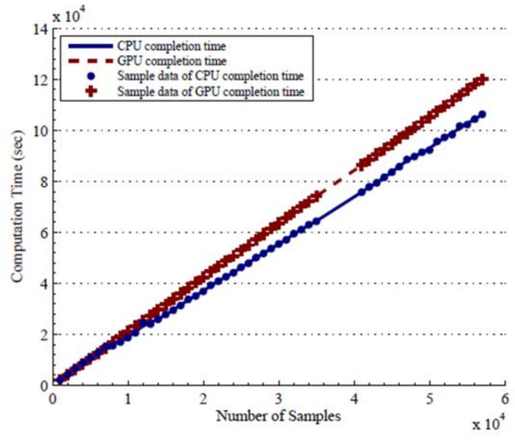

Figure 3: Plot between the network's average completion time and the total number of MNIST samples for Number of Nodes = 4000, Number of Layers = 8 and Number of Weights = 1999286

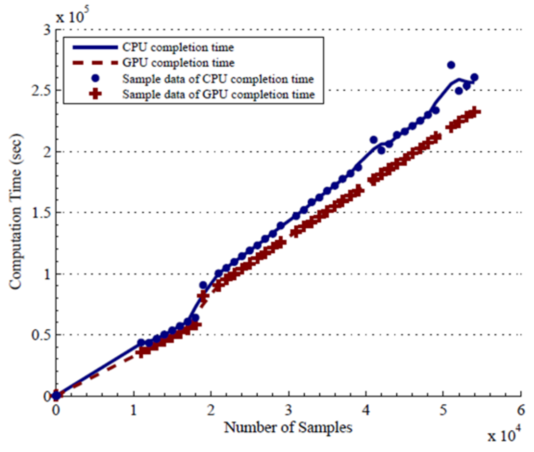

Figure 4: Plot between the network's average completion time and the total number of MNIST samples for Number of Nodes= 4000, Number of Layers = 8 and Number of Weights = 1999286

$$\lambda = \frac{2.N}{l} \tag{24}$$

By replacing the value of λ from (24) on equations in (23), we can create the BP network having largest number of weight connections for a required number of nodes and layer-size. We used this technique to create our BP network for performance analysis, taking the total node-count and layer-count as input. Figure (2) shows the internal configuration of a BP network with maximum weight distribution. Similarly, we used the MNIST dataset to analyse the performance of the parallelized code. The MNIST database (Modified National Institute of Standards and Technology database) is a large database comprised of handwritten digits which is commonly used for training and testing various image processing systems. The MNIST database of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger set available from NIST. [11]. The original bi-level images from NIST were size normalized to fit in a 20x20 pixel box, while preserving their aspect ratio. After applying anti-aliasing techniques, the resulting MNIST images centres around a 28x28 box. So, if we use a flattened MNIST image vector without any interpolation as our input vector, the size of input layer becomes 784. Since the MNIST dataset is simply too large for our testing apparatus, tests should be performed on a section of the dataset. This gives us a way to measure the performance of our parallel code; we could simply compare the time it takes to complete a series of BPN operations on both sequential and parallel codes, with increasing number of MNIST image samples. We can define the BPN operation to be the total time a BP network takes to perform a forward-reverse-update operation for 100 iterations. It can be conceded that it may require more than 100 iterations for such a large dataset to converge. But, we are more interested in comparing the change in computation speed than the change in convergence factor. The graphs in figures. 3 and 4 show the relationship between the number of MNIST image samples and the time it took for the network to complete forward-backward-update operations, for both sequential and parallel codes. When the number of samples became exceedingly large and approaches the maximum sample size, the computer took far too long time to converge its output. To overcome this, we adjusted the number of BP network's iterations in accordance with the sample size and extrapolated the result for 100 iterations.

We tried to see the differences between the CPU and GPU computation time for node size, 2000 and 4000, and layer count, 4 and 8 respectively. As we plotted the graphs for the two specifications mentioned above, we got the expected linear relationship between the MNIST sample size and the iteration time. But what we found different was the approximate slope of the plot for CPU and GPU codes. In figure 4, where the node count was 4000 and number of layers, 8, we found the CPU code to be slower than the GPU code at every part of the curve. That was not the case when the node count was 2000 and the number of layers, 4, as in figure 3. This indicates that size plays an important factor in deciding performance of the parallelized network. This may be due to the fact that the host memory-device memory interactions take a significant amount of time, and there is not enough arithmetic intensity or number of numeric computations performed per memory transaction. [12] But, when the number of nodes increases, the parallel-optimization begins to show its effects and the disparity in computation time reduces. At larger node size however, we see that the average CPU completion time is significantly larger than the average GPU completion time. This kind of performance speedup using graphics processors has been noted in many other computational problems, like GPU-based matrix multiplication using Strassen's algorithm. [13]

V. Conclusion

The Parallelization of Backpropagation network presents itself as an appealing option, if the number of nodes in the network is exceedingly large. This way, we can perform minimal change in the programming design, while ensuring significant improvement in performance.

VI. References

[1] G. S. Almasi, A. Gottlieb, "*Highly Parallel Computing*", Benjamin-Cummings Publishing Co., Inc., USA, 1989.

[2] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "*Learning representations by back-propagating errors*," Nature, vol. 323, no. 6088, pp. 533–536, 1986.

[3] C. Li, C. Yu, "*Performance evaluation of public non-profit hospitals using a BP artificial neural network: The case of Hubei province in china*," International Journal of Environmental Research and Public Health, Aug 2013.

[4] Y. Li, Y. Fu, H. Li, and S. W. Zhang, "*The improved training algorithm of back propagation neural network with self-adaptive learning rate*," in 2009 International Conference on Computational Intelligence and Natural Computing, pp. 73–76, 2009.

[5] J. Zhu, P. Sutton, "*FPGA implementations of neural networks–a survey of a decade of progress*," Field Programmable Logic and Application, pp. 1062-1066, 2003.

[6] I. B. D. Steinkraus, P.Y. Simard, "*Using GPUs for machine learning algorithms*," Document Analysis and Recognition, pp. 1115-1120, 2009.

[7] C. M. Bishop, "*Pattern Recognition and Machine Learning*", Springer-Verlag New York, USA, 2006.

[8] Y. Yuan, "*Step-sizes for the gradient method*," AMS/IP Studies in Advanced Mathematics, 1999.

[9] R. A. Jacobs, "*Increased rate of convergence through learning rate adaptation*," Neural Networks, vol. 1, 1988.

[10] M. J. Flynn, "*Some computer organizations and their effectiveness*," IEEE Transactions on Computers, vol. C-21, no. 9, pp. 948–960, 1972.

[11] E. Kussul, T. Baidyk, "*Improved method of handwritten digit recognition tested on MNIST database*," Image and Vision Computing, vol. 22, no. 12, pp. 971 – 981, 2004.

[12] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C.Phillips, "*GPU computing*," Proceedings of the IEEE, vol. 96, no. 5, pp. 879–899, May 2008.

[13] U. Ray, T.K. Hazra, U.K. Ray, "*Matrix Multiplication using Strassen's Algorithm on CPU & GPU*", International Journal of Computer Sciences and Engineering, Vol.4, Issue.10, pp.98-105, 2016.

Authors Profile

Mr. K. Devkota pursed Bachelor of Electronics and Communications Engineering from the Institute of Engineering (IOE), Nepal in the year 2016. He is currently working at the Department of Electronics and Communications Engineering (DOECE) of IOE. His main research work focuses on Artificial Intelligence, High Performance Computing and Machine Learning. He has almost a year of teaching experience and a year of research experience.

Mr. P. Bhattarai pursed Bachelor of Computer Engineering from the Institute of Engineering, Nepal in the year 2016. He is currently working as a freelance Java developer at his hometown, Biratnagar. His main research work focuses on the applicative aspects of Machine Learning, primarily Computer Vision and Natural Language Processing. He has 1 year of Research Experience.