I.J. Intelligent Systems and Applications, **2019, 5, 18-26**
Published Online May 2019 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijisa.2019.05.03

Accelerating Training of Deep Neural Networks on GPU using CUDA

D.T.V. Dharmajee Rao
Aditya Institute of Technology and Management, Tekkali-532201, Srikakulam, Andhra Pradesh, India
E-mail: dtvdrao@gmail.com

K.V. Ramana
JNTUK College of Engineering, JNTUK University, Kakinada - 533003, Andhra Pradesh, India
E-mail: vamsivihar@gmail.com

Received: 15 October 2018; Revised: 16 November 2018; Accepted: 13 December 2018; Published: 08 May 2019

Abstract—The development of fast and efficient training algorithms for Deep Neural Networks has been a subject of interest over the past few years because the biggest drawback of Deep Neural Networks is enormous cost in computation and large time is consumed to train the parameters of Deep Neural Networks. This aspect motivated several researchers to focus on recent advancements of hardware architectures and parallel programming models and paradigms for accelerating the training of Deep Neural Networks. We revisited the concepts and mechanisms of typical Deep Neural Network training algorithms such as Backpropagation Algorithm and Boltzmann Machine Algorithm and observed that the matrix multiplication constitutes major portion of the work-load for the Deep Neural Network training process because it is carried out for a huge number of times during the training of Deep Neural Networks. With the advent of many-core GPU technologies, a matrix multiplication can be done very efficiently in parallel and this helps a lot training a Deep Neural Network not consuming time as it used to be a few years ago. CUDA is one of the high performance parallel programming models to exploit the capabilities of modern many-core GPU systems. In this paper, we propose to modify Backpropagation Algorithm and Boltzmann Machine Algorithm with CUDA parallel matrix multiplication and test on many-core GPU system. Finally we discover that the planned strategies achieve very quick training of Deep Neural Networks than classic strategies.

Index Terms—Deep Neural Networks, Matrix multiplication, CUDA, Many-core GPU systems.

I. INTRODUCTION

In the recent years, Deep Neural Networks (DNN) are widely used in many domains such as Handwriting recognition, Image identification, Object identification, Data classification, Pattern recognition, Speech and Natural Language processing etc. In any case, one drawback of DNN is that training stays terribly slow, particularly in compute intensive algorithms involving plenty of complex data sets [1]. Hence DNN needs a huge computational power for significant speedup of the training of DNN which were not available a few years ago. Researchers worked out the way to train DNN practically. The reason DNN has turned out to be extremely well known is training it effectively became possible and researchers used them to dominate the state of the art approaches [2].

As the need to train computationally intensive DNN algorithms is on rise, the convergence time involved in training these algorithms must reduce. This is accomplished with the upcoming many-core GPU technologies and high performance parallel programming models such as NVIDIA CUDA, AMD Brook+, OpenGL, and DirectX [3]. GPU (Graphics Processing Unit) is seen as compute device that is a coprocessor/accelerator to Central Processing Unit or host machine with its own memory and runs several parallel threads. The GPU is particular for large data compute-intensive parallel applications [4]. Significantly more circuitry is dedicated to data processing instead of data storage and flow management. CUDA is a development platform intended for composing and executing general purpose programs on the NIVIDIA GPU. Similar to graphics and animation algorithms and applications, CUDA algorithms and applications can also be quickened by data parallel computation of more number of parallel threads [5]. An instance of kernel is called thread, namely a program executing on the GPU. A collection of threads running physically in parallel is called Warp. A collection of threads that execute together by sharing memory on a GPU is called Thread Block. A collection of thread blocks that execute a single CUDA program in parallel is called a Grid [6].

A computing system comprises of a conventional CPU (Host) and at least one GPU (Device). The GPUs are massively parallel coprocessors/accelerators furnished with an extensive number of arithmetic execution units [7]. A CUDA source code comprises of various stages that are executed either on the CPU (Host) or a GPU (device). The stages that show almost no data parallelism

This work is open access and licensed under the Creative Commons CC BY 4.0 License.

are executed on the CPU. The stages with rich amount of data parallelism are executed on the GPU [8]. CUDA threads are significantly lighter weight than the CPU threads. It implies that the generation cost, resource utilization, and switching cost of GPU threads is much smaller than CPU threads. Due to efficient hardware support, CUDA threads take only a few cycles to create and schedule of threads where as CPU threads need thousands of clock cycles. When kernel function is invoked or launched, all the threads that are created take advantage of data parallelism [9].

In the modern DNN algorithms, matrix multiplication is an essential building block and constitutes 70-80% of the work-load for the DNN training process because it is carried out for more number of times during the training of DNN. As the matrix multiplication supports data parallelism, it can be done very efficiently by implementing it parallel using CUDA and executing on GPU [10]. In our previous work, we have used fast Winograd's matrix multiplication, fast parallel Winogard's matrix multiplication, and parallel blocked matrix multiplication with collapse clause for efficient training of DNN algorithms. This paper aims to prove experimentally that the matrix multiplication on GPU via CUDA is dawn of computing among various Standard (Sequential), CBLAS library subroutine on CPU, and CUBLAS library subroutine on GPU for matrix multiplication. The same paper proposes to modify Backpropagation Algorithm (BPA) and Boltzmann Machine Algorithm (BMA) by using CUDA parallel matrix multiplication to accelerate the learning of DNN on many-core GPU system.

The remainder of the paper is structured as outlined below. The relevant related work is reviewed in section 2. Section 3 illustrates the implementation of matrix multiplication using Standard algorithm, ATLAS CBLAS library subroutine for CPU and NVIDIA CUBLAS library subroutine, CUDA algorithm for GPU. Section 4 explains our implementation of BPA and BMA with parallel matrix multiplication using CUDA for NVIDIA GPU. Experimental observations are presented in section 5. Finally, section 6 summarizes the conclusions and directions for future enhancement of the paper.

II. Related Work

To enhance the performance of compute intensive algorithms and to exploit the capabilities of modern many-core processors, recently many researchers started implementing them using CUDA and executing on GPU. Sivakumar Selevarasu et al. have implemented MMDBM classifier with quick sort and radix sort techniques in both CPU and GPU computing and tested on medical database [11]. In this paper, the authors have compared the results of GPU to the CPU computing and showed that GPU quick sort and radix sort algorithms provide rapid and exact results with minimum execution time than same CPU algorithms for the MMDBM classifier. Shunlu Zhang et al. proposed a parallel Neural Network (NN) training technique using CUDA on GPUs [12]. Their results showed that the proposed technique achieves higher efficiency than traditional CPU implementation of Backpropagation NN training. Teng Li et al. have described an efficient GPU implemented Deep Belief Network (DBN) with Pre-training and Fine-tuning processes [13]. The authors have showed in their results that GPU implemented methods achieves significant speedup in both processes. Moreover they have proved that these results are superior to that of the OpenBLAS on the CPU and CUBLAS on the GPU. A.S. Al-Hamoudi and A.A. Biyalani have parallelized two kinds of data mining algorithms (KNN and Decision tree) on two different platforms (CUDA on GPU & OpenMP on Dual-Core) [14]. Authors have used UCI Machine Learning datasets for testing of the KNN and Decision tree algorithms. They reported average performance with OpenMP and remarkable performance with CUDA.

Nowadays due to exponential growth of data to be sorted, the applications are demanding fast information processing. Some authors modified known sorting algorithms such as Bubble sort, Quick sort, and Shell sort for effective use on CUDA platform [15, 16]. They have developed parallel programs for sorting the data and compared the performance with the sequential implementation and found that the reduced execution time with GPU implementation. T. Kalaiselvi et al. have proposed Per-Pixel Threading (PPT) and Per-Slice Threading (PST) and implemented some of the advanced image pre-processing algorithms for accelerating the computer aided diagnosis (CAD) systems in MRI volume analysis [17]. Authors have collected the image dataset from Whole Brain Atlas (WBA) maintained by Harvard Medical School and used for testing purpose. Their experiments showed that the GPU-based implementation achieved speedup of 3-338 times for PPT model and up to 30 times for PST model compared to conventional CPU process. Teja U. Naik and Nitesh Guinde have implemented the Gauss Seidel algorithm for solving Eigen values of symmetric matrices with CUDA on GPU and noticed that the speedup of GPU is better than CPU [18]. Keh Kok Yong et al. have compared the performance of three compression techniques (Huffman coding, LZSS, and Block-sorting) experimentally by implementing them using CUDA on GPU [19]. Among the above three algorithms, authors have proved that CUDA implemented Huffman coding has given the best performance in terms of compression ratio and compression speed.

III. Matrix Multiplication

The product of two square matrices A and B is a square matrix C whose elements are commonly defined as

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} . b_{k,j} . \quad (1)$$

Where $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$ are the elements in the ith row and j^{th} column, for the matrices A, B, and C, respectively and n is the size of the square matrix.

A. Standard Matrix Multiplication Algorithm

The standard (sequential) matrix multiplication algorithm has the structure of triple nested loops (il - i loop, jl - j loop, and kl - k loop) with cubic complexity. It is well known, well researched, and well understood algorithm. Listing 1 below shows the source code for the direct implementation of (1).

```
void seqMatMult(double* A, double* B, double* C, int width)
    {
        for (int il = 0; il < width; il++)
          for (int jl = 0; jl < width; jl++) {
                double Csum = 0;
                for (int kl = 0; kl < width; kl++) {
                    double x = A[il * width + kl];
                    double y = B[kl * width + jl];
                    Csum += x * y;
                }
                C[il * width + jl] =Csum;
                }
        }
```

Listing 1. Source code for Standard Matrix multiplication

B. CBLAS Matrix Multiplication Algorithm

BLAS is a collection of low level matrix and vector arithmetic operations such as multiply a vector by a scalar, multiply two matrices etc. LAPACK is built on top of the BLAS. LAPACK is a collection of high level linear algebra operations such as matrix factorizations that are used to find Eigen values of a matrix, singular value of a matrix, or to solve a linear system of equations etc. A portable high performance BLAS library with a CBLAS interface for the CPU is being provided by ATLAS Package. At present, it provides FORTRAN 77 and C interfaces to a portably efficient BLAS implementation as well as a few routines from LAPACK. Listing 2 shows how matrix multiplication is implemented using CBLAS.

```
void cblasMatMult(double *A, double *B, double *C,  int width)
{

double alpha = 1.0;
double beta = 0.0;

cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
width, width, width, alpha, A, width, B, width, beta, C, width);

}
```

Listing 2. Source code for Matrix multiplication using CBLAS

C. CUBLAS Matrix Multiplication Algorithm

BLAS library consists of set of implementations for matrix multiplication such as GEMM with single precision and double precision. While the reference BLAS implementation is not especially quick, there are number of third party optimized implementations like MKL from Intel, ACML from AMD, and CUBLAS from NVIDIA for the GPU [20]. The source code for multiplication of two matrices on GPU device with CUBLAS is shown in Listing 3.

```
void cublasMatMult(double *A, double *B, double *C, int width)
    {
    int da = width, db = width, dc = width;
    double alfa= 1;
    double bta = 0;
    double *alpha = &alfa;
    double *beta = &bta;
                                        //create a handle for CUBLAS
            cublasHandle_t handle;
            cublasCreate(&handle);
                                        // Do the actual multiplication
cublasDgemm(handle,CUBLAS_OP_N, CUBLAS_OP_N, width,
width, width, alpha, A, da, B, db, beta, C, dc);
                                        // Destroy the handle
            cublasDestroy(handle);          }
```

Listing 3. Source code for Matrix multiplication using CUBLAS

D. CUDA Matrix Multiplication Algorithm

Data parallelism refers to the program property whereby many arithmetic operations are carried out perfectly on required data structure simultaneously. The concept of data parallelism is applied to typical matrix multiplication. As shown CUDA Matrix multiplication in Listing 4, the output matrix C is produced by performing a dot product between the rows of first matrix A and the columns of second matrix B. These dot product operations for finding different elements of output matrix C can be performed on the GPU in a simultaneous manner without affecting results of each other.

```
__global__ void kernelMatMult(double* A, double* B, double* C,
int width)
    {
    int tx = threadIdx.x + blockDim.x * blockIdx.x;
    int ty = threadIdx.y + blockDim.y * blockIdx.y;
    double Cvalue = 0;
        for (int kl = 0; kl < width; kl++) {
            double Adelement = A[ty * width + kl];
            double Bdelement = B[kl * width + tx];
            Cvalue += Adelement * Bdelement;
            }
        C[ty * width + tx] = Cvalue;
        }
```

Listing 4. Source code for Matrix multiplication using CUDA

IV. CUDA Implementation of DNN Algorithms

Accelerating time consuming DNN training algorithms such as BPA and BMA is still an open problem that is being investigated widely. To achieve significant acceleration of DNN training algorithms, one has to use the remarkable computing power provided by modern many-core GPU devices by implementing algorithms using CUDA [21].

A. CUDA Backpropagation Algorithm (CUBPA)

Backpropagation is an algorithm that has been broadly used for training shallow neural networks with single hidden layer for its simplicity of implementation and

efficiency. However, the efficiency of Backpropagation Algorithm (BPA) decreases greatly when it is used to train DNN with multiple hidden layers [22]. As mentioned in introduction, matrix multiplication task is carried out for a huge number of times and thus it constitutes major portion of the work-load of BPA training process. CUDA programming makes it possible to accelerate the training of BPA by dividing the entire matrix multiplication task into the smaller tasks and running them simultaneously on a large number of cores available on the GPU. So as to decrease the high convergence time, the classic BPA gets adjusted by using CUDA kernel matrix multiplication algorithm as summarized in algorithm 1. A DNN consists of eight layers (Input, Output, and Six hidden) for experimentation is shown in Fig. 1.

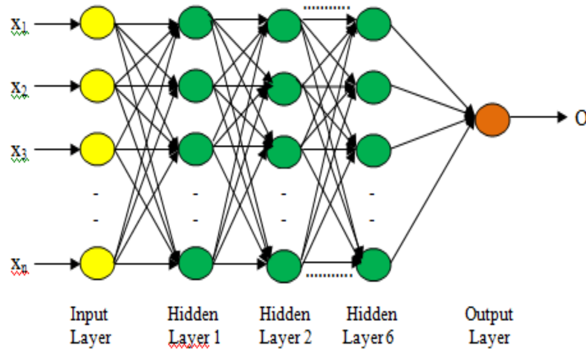

Fig.1. A multilayer Deep Neural Network

We use the following notations in order to describe the CUDA implementation of Backpropagation Algorithm (BPA). Here

D is a set of training patterns with target values.
X is a training pattern.
T_j is target output at neuron j in the output layer.
A_j is the actual output at neuron j.
$w_{i,j}$ is the connecting weight between neuron j of the current layer from neuron i of the previous layer.
E_j is the error of neuron j in the output/hidden layer.
η is the learning rate.

The BPA works as given under. The small random values are assigned to weights and biases to initialize the network. The input pattern is then applied and the net input, I_j, is calculated as sum of weighted input according to (2).

$$I_j = \sum_i w_{ij} A_i + \theta_j \tag{2}$$

The actual output at each neuron, A_j, for the obtained net input, I_j, is calculated by using the following sigmoid function (3).

$$A_j = \frac{1}{1+e^{-I_j}} \tag{3}$$

The error, E_j, of the neuron in the output layer is computed by subtracting the actual output value from the target value and this difference is multiplied by the derivative of sigmoid function as shown in (4).

$$E_j = A_j(1-A_j)(T_j - A_j) \tag{4}$$

Similarly the error of the neuron in the hidden layer is calculated as the product of weighted sum of the propagated errors and derivative of sigmoid function as shown in (5).

$$E_j = A_j(1-A_j)\sum_K E_k w_{jk} \tag{5}$$

Having obtained the error for hidden layer units, the weights are updated according to (6) and (7), where $\Delta w_{i,j}$ is the change in weight $w_{i,j}$.

$$\Delta w_{ij} = (\eta) E_j A_i \tag{6}$$

$$w_{ij} = w_{ij} + \Delta w_{ij} \tag{7}$$

Similarly biases are changed according to (8) and (9), where $\Delta\theta_j$ is the change in bias θ_j.

$$\Delta\theta_j = (\eta) E_j \tag{8}$$

$$\theta_j = \theta_j + \Delta\theta_j \tag{9}$$

The training process stops when the error is below some specified threshold or a pre-specified number of iterations have expired.

Algorithm 1: CUDA implementation of BPA

1. *Allocate CPU memory for weights (w) and biases (θ);*
2. *Allocate GPU memory for weights (d_w) and output (d_A);*
3. *InitializeWeights and InitializeBiases;*
4. ***for** itr=1 to numIterartions **do*** {
5. ***for** ptrn=1 to numPaterns **do*** {
6. ***for** each inputLayerUnit j* {
7. $A_j = I_j$;
8. ***for** each hidden or outputLayerUnit j* {
9. *cudaMemcpy* $(d_w_{i=1...n,j}, w_{i=1...n,j})$;
10. *cudaMemcpy* $(d_A_{i=1...n}, A_{i=1...n})$;

11. $Net_j = \text{ker}\,nelMatMult <<< numThreads >>>$
    $(d_w_{i=1...n,j}, d_A_{i=1...n});$
12. $I_j = Net_j + \theta_j;$
13. $A_j = \dfrac{1}{1+e^{-I_j}}$; }
14. *for each unit j in the outputLayer*
15. $E_j = A_j(1-A_j)(T_j - A_j);$
16. *for each unit j in the hiddenLayers, from the last to first hiddenLayer {*
17. *cudaMemcpy* $(d_E_{k=1...n}, E_{k=1...n});$
18. *cudaMemcpy* $(d_w_{j,k=1...n}, w_{j,k=1...n});$
19. $Net_k = \text{ker}\,nelMatMult <<< numThreads >>>$
    $(d_E_{k=1...n}, d_w_{j,k=1...n});$
20. $E_j = A_j(1-A_j)Net_k;\}$
21. *for each weight w_{ij} in network {*
22. $\Delta w_{ij} = (\eta)E_j A_j;$
23. $w_{ij} = w_{ij} + \Delta w_{ij};$ }
24. *for each bias θ_j in the network. {*
25. $\Delta\theta_j = (\eta)E_j;$
26. $\theta_j = \theta_j + \Delta\theta_j;$ }
27. }
28. }
29. }

B. CUDA Boltzmann Machine Algorithm (CUBMA)

Structurally, a Restricted Boltzmann Machine (RBM) is a shallow neural network that consists of a set of visible units, v, and a set of hidden units, h, and connections between units are symmetrically weighted and bidirectional [23] as depicted in Fig. 2.

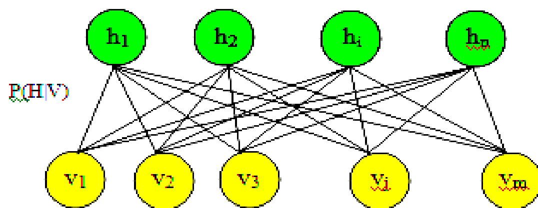

Fig.2. Model of RBM

Training a Deep Boltzmann Machine (DBM) is a computationally time consuming task that involves training a stack of several RBMs as shown in Fig. 3 and requires certain high amount of training time. Moreover, as the dimensionality and quantity of data increases, the computing load of training a DBM increases rapidly [24].

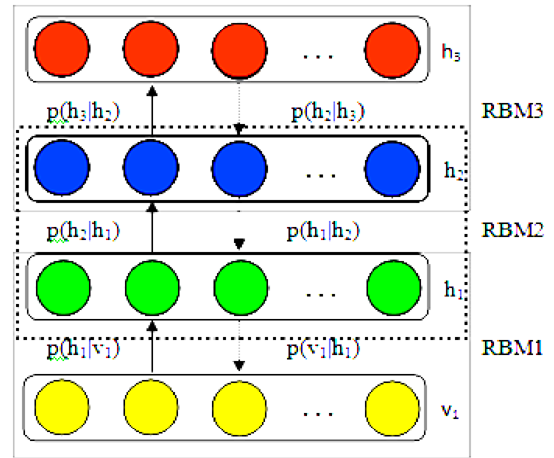

Fig.3. DBM: A stack of RBMs

So as to decrease the long training time, the classic BMA gets adjusted by using CUDA kernel matrix multiplication algorithm as explained in algorithm 2. We use the following notations in order to describe CUDA implementation of BMA. Here

v_i = set of visible units where $i = 1......n.$
h_j = set of hidden units where $j = 1......n.$
vb = visible bias unit.
hb = hidden bias unit.
$w_{i,j}$ = the weight on the connection from v_i to h_j.
w^j_{hidden} = the hidden node h_j weight vector.
$w^j_{visible}$ = the visible node v_j weight vector.

All the visible units are assumed to be conditionally independent of hidden vector H for an RBM network i.e.

$$P(V \mid H) = \prod_{i=1}^{n} P(v_i \mid H). \tag{10}$$

Similarly all the hidden units are assumed to be conditionally independent of visible vector V i.e.

$$P(H \mid V) = \prod_{i=1}^{n} P(h_i \mid V). \tag{11}$$

The conditional distribution of hidden and visible unit j is given by (12) and (13).

$$p(h_j = 1 \mid V) = \sigma(hb_j + (w^j_{hidden} * V)) \tag{12}$$

$$p(v_j = 1 \mid H) = \sigma(vb_j + (w^j_{visible} * H)) \tag{13}$$

Where σ is the logistic or sigmoid function as shown in (14).

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{14}$$

Algorithm 2: CUDA implementation of BMA

1. *Allocate CPU memory for visible vector (V), hidden vector (H), weights (w) and biases (vb, hb);*
2. *Allocate GPU memory for visible vector (d_V), Hidden vector (d_H) and weights (d_w);*
3. *InitializeWeights and initializeBiases;*
4. *Initialize the visible units states to the training data;*
5. *Positive phase: Reconstruction of hidden units using positive statistics (E_j) is given by*

   cudaMemcpy $(d_w^j{}_{hidden}, w^j{}_{hidden})$;

   cudaMemcpy(d_V, V);

   sum=kernelMatmult<<<numThreads>>> $(d_w^j{}_{hidden}, d_v)$

   $$P(h_j = 1/V) = \frac{1}{1+e^{-(hb_j + sum))}}$$

6. *Negative phase: Reconstruction of visible units using negative statistics (E_j) is given by*

   cudaMemcpy $(d_w^j{}_{visible}, w^j{}_{visible,})$;

   cudaMemcpy(d_H, H);

   sum = kernelMatMult<<<numThreads>>> $(d_w^j{}_{visible}, d_H)$

   $$P(v_j = 1/H) = \frac{1}{1+e^{-(vb_j + sum)}}$$

7. *Update phase:*

   $w_{ij} = w_{ij}^{old} + \eta(positive(E_j) - negative(E_j))$;

   Iterate with all training vectors till the error is below some specified threshold

V. Results and Discussions

The performance of proposed methods was compared against classic methods by measuring the convergence time consumption of DNN training process. All the programs were developed using C/C++/CUDA, then tested on many-core GPU system and the execution time was measured. So as to assess the performance of our implementations, testing was conducted on a computer (HP Compaq, Intel Core I7-2600 3.40 GHz Processor, 4 Cores, 8 Threads, 64 Bit, 8 MB Cache, 4GB RAM with a GPU accelerator NVIDIA Quadro K620, 384 CUDA cores, 2GB RAM) running Ubuntu operating system (Linux 4.4.0-57) with software version gcc/g++ 5.4.0., and nvcc V7.5.17**.** A similar domain was used for all examinations that were completed for this paper. All the programs were executed for five times and the mean execution time was computed.

A. *Matrix multiplication on GPU via CUDA: The Dawn of Computing*

For simple understanding, we have used the square matrices with a size that is a power of 2. With cautious implementation, the investigations can also be conducted to non square matrices. The elements of matrices are generated by a random function and type casted into desired type of data.

The execution times of all four algorithms for matrix multiplication are furnished in the Table 1. From the Table, it was observed that the Standard algorithm performs more slowly mainly due to its sequential nature. Among the four Algorithms, CUDA algorithm is showing the tremendous performance than other three algorithms. It is clearly appeared in Fig. 4 that the execution time taken by CUDA algorithm is almost negligible compared to all other algorithms, thus the matrix multiplication on GPU using CUDA is the dawn of computing. Fig. 5 shows the speedup of CUDA over Standard, CBLAS, and CUBLAS implementations for matrix multiplication.

Table 1. Execution time comparison of four matrix multiplication algorithms

Matrix Dimension	Elapsed time (Secs)				Speedup of CUDA over		
	Standard	CBLAS	CUBLAS	CUDA	Standard	CBLAS	CUBLAS
4	0.000001	0.000002	0.000267	0.000013	0.08	0.15	20.54
8	0.000011	0.000002	0.000269	0.000013	0.85	0.15	20.69
16	0.000071	0.000005	0.000437	0.000013	5.46	0.38	33.62
32	0.000560	0.000026	0.000430	0.000014	40.00	1.86	30.71
64	0.002971	0.000210	0.000303	0.000003	990.33	70.00	101.00
128	0.017985	0.000887	0.000648	0.000005	3597.00	177.40	129.60
256	0.123594	0.003768	0.002033	0.000004	30898.50	942.00	508.25
512	0.995183	0.021060	0.011976	0.000008	124397.88	2632.50	1497.00
1024	10.885084	0.160893	0.093565	0.000010	1088508.40	16089.30	9356.50

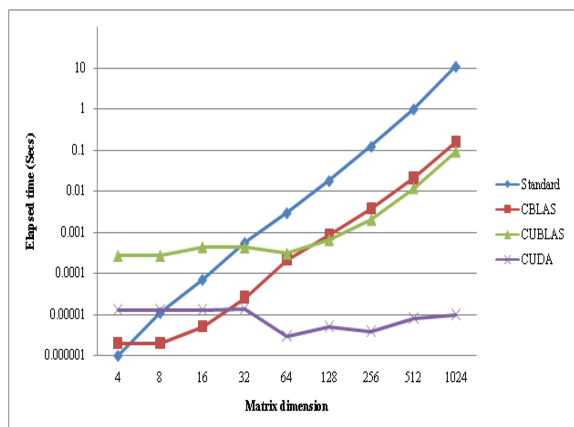

Fig.4. Performance of CUDA matrix multiplication vs Standard, CBLAS ksubroutine on CPU and CUBLAS subroutine on GPU

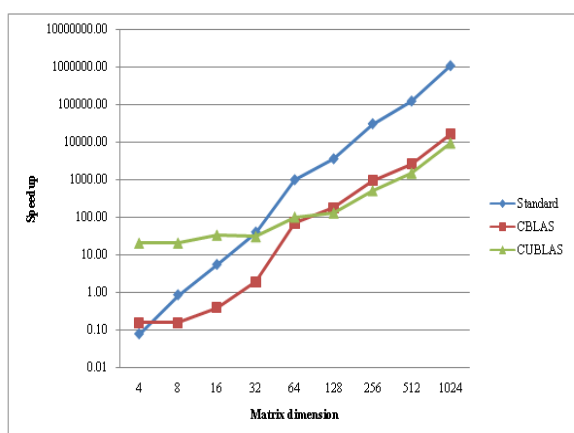

Fig.5. Speedup trends of CUDA matrix multiplication over Standard, CBLAS subroutine on CPU and CUBLAS subroutine on GPU

B. Performance Comparison of BPA and CUBPA

The classification of odd and even numbers was chosen for the performance analysis of Backpropagation training algorithm as it is simple problem to implement. The number of units in the input layer is equal to the size of training pattern (binary form of given number). Number of units is equal in input and all hidden layers except output layer. One output unit is employed to represent two different classes where the output 1 represents odd class, and the output 0 represents even class. The training patterns are produced using random function. The small random values varying from -1.0 to 1.0 are assigned to weights and biases. The observations are recorded after training the Deep Neural Network (DNN) for pre-specified 30,000 epochs.

Table 2 exhibits that proposed CUDA implementation (CUBPA) shows significant improvement in reducing DNN convergence time compared to direct implementation (BPA). Moreover, from the Table, it is evident that our GPU-based CUDA implementation (CUBPA) achieved a significant performance speedup and in addition the speedup achieved increases further with increasing the size of training pattern. From the Fig. 6, it is crystal clear that the training time for CUBPA has reached lowest and almost stable there from the pattern size 40 even though the size of training pattern is increasing where as the training time for BPA is increasing as the size of the training pattern is increasing.

Table 2. Training time comparison of BPA and CUBPA

Pattern Size	Elapsed time (Secs)		Speedup
	BPA	CUBPA	
10	0.87249	1.26188	0.69
20	6.48486	1.59640	4.06
30	20.38150	2.47829	8.22
40	50.48880	0.10349	487.85
50	93.12040	0.10333	901.24
60	160.77000	0.10344	1554.18
70	256.47700	0.10331	2482.57
80	392.08200	0.10334	3793.99
90	544.53800	0.10331	5270.79
100	758.84900	0.10242	7409.44

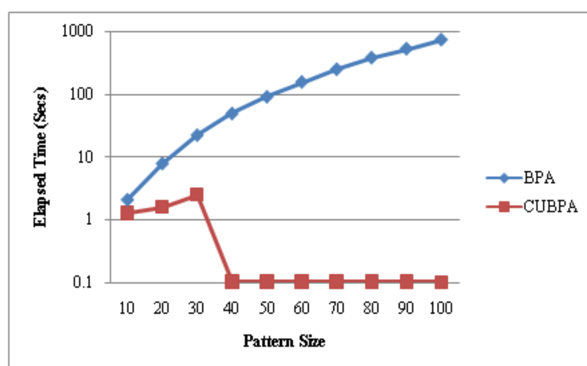

Fig.6. Performance of BPA vs CUBPA

Table 3. Training time comparison of BMA and CUBMA

Pattern Size	Elapsed time (Secs)		Speedup
	BMA	CUBMA	
10	1.3897	2.15814	0.64
20	11.0734	2.73395	4.05
30	34.9284	4.26819	8.18
40	85.4838	0.17219	496.46
50	159.8400	0.17219	928.26
60	275.7080	0.17201	1602.88
70	440.4230	0.17195	2561.32
80	672.0940	0.17201	3907.34
90	934.8770	0.17209	5432.61
100	1302.9900	0.17209	7571.66

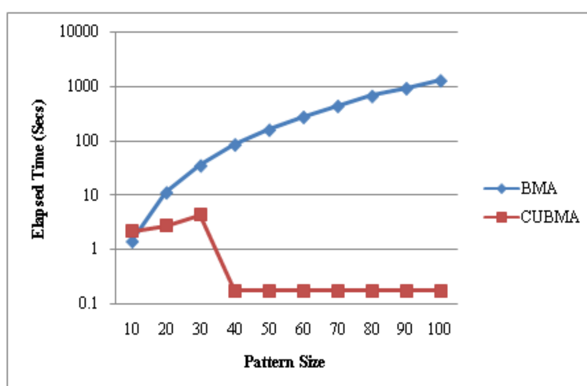

Fig.7. Performance of BMA vs CUBMA

C. Performance Comparison of BMA and CUBMA

A DBM with six RBM layers is constructed to generate the original binary pattern for a given incorrect pattern as input. The observations are recorded after learning the Deep Neural Network (DNN) for pre-specified number of epochs (30,000) in both forward and backward directions for each layer. Table 3 exhibits that proposed CUDA implementation (CUBMA) indicates significant improvement in reducing DNN learning time in comparison with direct BMA implementation. Moreover, from the Table, it is evident that our GPU-based CUDA implementation (CUBMA) achieved a significant performance speedup and in addition the speedup achieved increases further with increasing the size of training pattern. From the Fig. 7, it is totally evident that the training time for CUBMA has reached lowest and almost stable there from the pattern size 40 even though the size of training pattern is increasing where as the training time for BMA is increasing as the size of the training pattern is increasing.

VI. Conclusions and Future Work

In this paper, we examined and implemented Standard subroutine, CPU-based ATLAS-optimized CBLAS library subroutine, GPU-based NVIDIA-optimized CUBLAS library subroutine, and CUDA subroutine for computing matrix multiplication. By the evidence of results, it is crystal clear that CUDA matrix multiplication performs very much faster than other three implementations. The reason for using the state-of-the-art highly optimized CBLAS library and CUBLAS library matrix multiplication subroutines in addition to Standard matrix multiplication is for fair comparison. Then it has been proposed to use CUDA matrix multiplication to modify BPA and BMA. After our experiments, we found that the proposed methods (CUBPA and CUBMA) have achieved exponential speedup in training Deep Neural Networks than existing standard methods.

For the future work, the existing algorithms will be revisited and examined in the areas of Cryptography, Image Processing, video Frames, Bio-Informatics, and Weather Forecasting for possible implementation and thus performance improvement by exploiting the capabilities of modern multi-core CPU-based and many-core GPU-based systems via CUDA and other parallel programming models and paradigms.

References

[1] I-Hsin Chung et al., "Parallel Deep Neural Network Training for Big Data on Blue Gene/Q" *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1703-1714, 2017. DOI: 10.1109/TPDS.2016.2626289.

[2] Canping Su et al., "An Efficient Deep Neural Networks Training Framework For Robust Face Recognition", *IEEE International Conference on Image Processing (ICIP)*, Beijing, China, pp. 3800-3804, 2017.

[3] Anala M.R. et al., "Comparative Study of Computationally Intensive Algorithms on CPU and GPU", *International Journal of Applied Engineering Research ISSN 0973-4562*, vol. 11, no. 5, pp. 2996-2999, 2016.

[4] Mouna Afif, Yahia Said, and Mohamed Atri, "Efficient 2D Convolution Filters Implementations on Graphics Processing Unit Using NVIDIA CUDA", *I.J. Image, Graphics and Signal Processing,* no. 8, pp. 1-8, 2018.

[5] Mohammad Usman Ashraf, Fadi Fouz, and Fathy Alboraei Eassa, "Empirical Analysis of HPC Using Different Programming Models", *I.J. Modern Education and Computer Science,* no. 6, pp. 27-34, 2016.

[6] Sunitha N.V., Raju K., and Niranjan N. Chiplunkar, "Performance Improvement of CUDA Applications by Reducing CPU-GPU Data Transfer Overhead", *International Conference on Inventive Communication and Computational Technologies*, Coimbatore, India, pp. 211-215, 2017.

[7] Arun Kumar Parakh, M.Balakrishnan, and Kolin Paul, "Performance Estimation of GPUs with Cache" *IEEE 26th international Parallel and Distributed Processing Symposium Workshops & Ph D Forum*, Shanghai, China, pp. 2384-2393. 2012. DOI: 10.1109/IPDPSW.2012.328

[8] Sapna Saxena and Neha Kishore, "PRDSA: Effective Parallel Digital Signature Algorithm for GPUs", *I.J. Wireless and Microwave Technologies.* No. 5, pp. 14-21, 2017.

[9] Ke Yan, Junming Shan, and Eryan Yang, "CUDA-based Acceleration of the JPEG Decoder", *Ninth International Conference on Natural Computation (ICNC)*, Shenyang, China, pp. 1319-1323, 2013.

[10] Musab COSKUN et al., "An Overview of Popular Deep Learning Methods", *European Journal of Technic*, vol. 7, no. 2, pp. 164-175. 2017. DOI: 10.23884/ejt.2017.7.2.11

[11] Sivakumar Selvarasu, Ganesan Periyanagounder, and Sundar Subbiah, "A MMDBM Classifier with CPU and CUDA GPU Computing in Various Sorting Procedures", *The International Arab Journal of Information Technology*, vol. 14, no. 6, pp. 897-906, 2017.

[12] Shunlu Zhang, Pavan Gunupudi, and Qi-Jun Zhang, "Parallel Back-Propagation Neural Network Training Technique Using CUDA on Multiple GPUs", *IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*, Ottawa, Canada, pp. 1-3, 2015.

[13] Teng Li et al., "Optimized Deep Belief Networks on CUDA GPUs", *International Joint Conference on Neural Networks (IJCNN)*, Killarney, Ireland, pp. 1-8, 2015.

[14] Adwa S. Al-Hamoudi, and A. Ahmed Biyabani, "Accelerating Data Mining with CUDA and Open MP", *IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA)*, Doha, Qatar, pp. 528-535, 2014.

[15] Bakulev Aleksandr Valerievich et al., "The implementation on CUDA Platform Parallel Algorithms Sort the Data", *6th Mediterranean Conference on Embedded Computing (MECO),* Bar, Montenegro, pp. 1-4, 2017.

[16] Neetu Faujdar, and Satya Prakash Ghrera, "A Practical Approach of GPU Bubble Sort with CUDA Hardware", *7th International Conference on Cloud Computing, Data Science & Engineering - Confluence*, Noida, India, pp. 7-12, 2017.

[17] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundraram, "Performance of Medical Image Processing Algorithms Implemented in CUDA running on GPU based Machine",

I.J. Intelligent Systems and Applications, no. 1, pp. 58-68, 2018.
[18] Teja U. Naik and Nitesh Guinde, "Implementing the Gauss Seidel Algorithm for Solving Eigenvalues of Symmetric Matrices with CUDA", *IEEE International Conference on Computing Methodologies and Communication*, Erode, India, pp. 922-925, 2017.
[19] Keh Kok Yong, Meng Wei Chua, and Wing Kent Ho, "CUDA Lossless Data Compression Algorithms: A Comparative Study", *IEEE Conference on Open Systems (ICOS)*, Langkawi, Malaysia, pp. 7-12, 2016.
[20] Pai-Wei Lai et al., "Accelerating Strassen-Winograd's Matrix Multiplication Algorithm on GPUs", *20th Annual International Conference on High Performance Computing*, Bangalore, India, pp. 139-148, 2013.k
[21] Zhilu Chen et al., "A Fast Deep Learning System Using GPU", *IEEE International Symposium on Circuits and Systems (ISCAS)*, Melbourne VIC, Australia, pp. 1552-1555, 2014.
[22] Javier A. Cruz-lopez, Vincent Boyer, and Didier El-Baz, "Training Many Neural Networks in Parallel via Back-Propagation", *IEEE International Parallel and Distributed Processing Symposium Workshops*, Lake Buena Vista, FL, USA , pp. 501-509, 2017. DOI: 10.1109/IPDPSW.2017.72
[23] Lei Jin et al., "Training Large Scale Deep Neural Networks on the Intel Xeon Phi Many-core Coprocessor," *IEEE 28th International Parallel & Distributed Processing Symposium Workshops*, Phoenix, AZ, USA, pp. 1622-1630, 2014. DOI: 10.1109/IPDPSW.2014.194
[24] Noel Lopes, Bernardete Ribeiro, and Joao Goncalves, "Restricted Boltzmann Machines and Deep Belief Networks on Multi-Core Processors", *WCCI 2012 IEEE World Congress on Computational Intelligence*, Brisbane, QLD, Australia, pp. 1-7, 2012.

Authors' Profiles

D.T.V. Dharamajee Rao is currently working as Professor in the Department of Computer Science and Engineering at Aditya Institute of Technology and Management, Tekkali, Srikakulam, Andhra Pradesh, India. He received B.Tech. degree in Computer Science and Engineering in 1993 and M.Tech. degree in Computer Science and Technology in 2001 from Andhra University, Visakhapatnam, Andhra Pradesh, India. He is pursuing Ph.D. in the Department of Computer Science and Engineering, JNT University, Kakinada, Andhra Pradesh, India. He got published more than 12 papers in International and National, Conferences and Journals. His current research interests include Data Mining, Neural Networks, Parallel Computing and Linear Algebra Techniques.

K.V. Ramana received B.Tech. degree in Electronics and Communication Engineering from JNT University, Hyderabad. Telangana, India in 1986, M.Tech. degree in Computer Science and Engineering from University of Hyderabad, Hyderabad, Telangana, India in 1990, and Ph.D. in Computer Science and Engineering from Rayalaseema University, Kurnool, Andhra Pradesh, India in 2011. He is working as Professor in the Department of Computer Science and Engineering, JNTUK College of Engineering, JNTUK University, Kakinada, Andhra Pradesh, India. He got published more than 20 papers in International and National, Conferences and Journals. His research interests include Data Warehousing and Mining, Neural Networks, Image Processing, and Pattern Recognition.

How to cite this paper: D.T.V. Dharmajee Rao, K.V. Ramana, "Accelerating Training of Deep Neural Networks on GPU using CUDA", International Journal of Intelligent Systems and Applications(IJISA), Vol.11, No.5, pp.18-26, 2019. DOI: 10.5815/ijisa.2019.05.03