ISSN:0975-9646

GPU Parallelization of Neural Network

Snehal M. Wagh[1], Prof. Dipti Pawar[2]

Dept. of Computer Eng., Sinhgad College of Engg. Pune. India[1,2]

Abstract: Graphics Processing Unit (GPU) can provide remarkable performance gains when compared to Central Processing Unit (CPU) for computational intensive application. GPU has acquired programmability to perform general purpose computation fast by running ten thousands of threads concurrently. The GPU can be used not only for processing graphics but also for high speed computing. GPU uses the SIMD, that same portion of code will be executed in parallel and applied to various elements of a data set. Thus they are more attractive to be used as dedicated hardware in many fields such as machine learning. Training in a multilayer neural network with back-propagation is usually time consuming process. This paper describes the GPU parallelization of back-propagation neural network. The GPU can be used not only for processing but also for high speed computing.

***Keywords*: GPU, Back-Propagation Algorithm, Neural Network, Parallel Computing.**

I. INTRODUCTION

The GPU hardware contains many execution cores which can be used for high demanding graphics application [1]. One of the most popular GPU brand is the NVIDIA graphics card. The GPU is graphics processing unit which perform the operation in parallel. The GPU is work in neural network to perform the operation. GPU have different memory location. The popular machine learning model such as Extreme Learning Machine (ELM) and support vector Machine have been implemented on GPU parallel processing. The global memory is basic communication area the CPU and GPU which is accessible from all the thread but has a long latency time. Each streaming processors (sp) memory is very fast and the entire variable defined in this memory can be accessed by all block. Apart from these two memories, there are also constant memory and texture memory which are specifically designed for different purpose. A kernel the program executed by device, is run in parallel using large number of thread.

Training an artificial neural network is time consuming due the large number of epochs and weight updates required to reach an optimal performance. So then it made to increase the convergence speed or to reduce the computational cost [2]. Artificial neural network are widely used for data mining and pattern recognition [3]. Pattern classification deals with the problem of identifying underlying structure of data. Nowadays fuzzy logic based techniques interest in neurofuzzy pattern recognition systems [4]. Back-propagation neural network uses the supervised learning for training. Parallelization of ANN execution the problem of slow ANN execution can be somehow mitigated by using the modern microprocessor architectures. Instruction set of the modern microprocessor contains instructions for used multiply-and-add operations. In this case, multiplication and addition within a synapse are performed at the same time. Sometimes several instructions can be executed simultaneously. In addition, the multi-core architectures of modern microprocessor can be utilized. The processing effort can be divided among different cores. However, the number of cores is relatively small and the speed-up would also be small.

A group of neurons is used to process a part of data from the input data layer. The neurons are grouped together in such a way that minimizes the amount data that must be

read. In this way, the same data can be used with several neurons at the same time. This minimizes the number of memory access cycles where the data can only be read serially. When the neurons have finished their job, another patch of input area is processed [5][6]. If necessary, another set of weights is used. Several groups of neurons can be executed in parallel by different processing cores. In the hardware implementation as well as with the CUDA solution, each neuron from a group is processed simultaneously with the others. An attempt was also made to parallelize the training of a multi-layer ANN using back-propagation learning algorithm. It was found that much less parallelization of the code is possible. First of all the training is done in repetitive cycles where different training patterns are exposed to the net. Because of the nature of the back propagation, the layers had to be evaluated one-by-one the results from just one of the layers influenced the evaluation of others.

The training and execution of ANN is performed in three steps as preparation of the initial data, transfer of the data to the CUDA device, evocation of the kernel routine and transfer of the result to the host. The same data may be evaluated using several kernels sequentially and data transfer operations may overlap with the kernel execution [7]. It is also feasible to distribute the workload between several CUDA devices. The upper-end graphical cards incorporate two GPUs, which can be used as two independent CUDA devices.

II. GPU ARCHITECTURE

When the programmer wants to process some data on the GPU, it loads the data in the GPU's global memory, processes it and copies the result back to the CPU [1]. GPU architecture is as shown in Figure 1. 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. The factors in GPU structure contains.

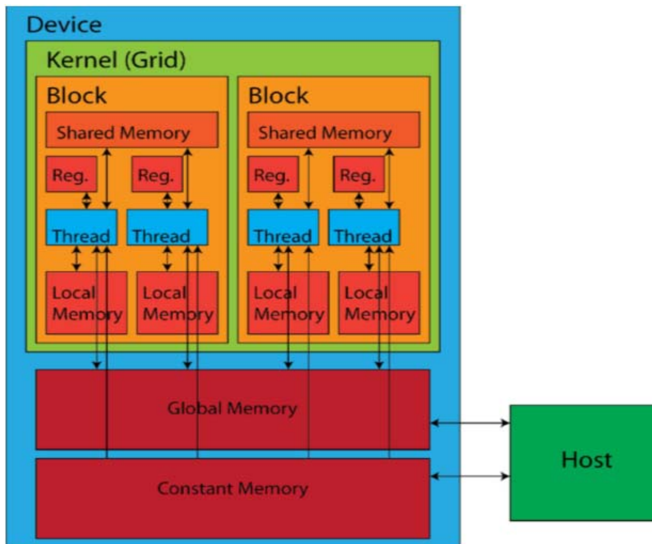

Figure1. A block diagram of the GPU architecture

The Grid

A grid is a group of threads all running the same kernel. These threads are not synchronized. Every call to CUDA from CPU is made through one grid. Starting a grid on CPU is a synchronous operation but multiple grids can run at once. On multi-GPU systems, grids cannot be shared between GPUs because they use several grids for maximum efficiency.

The Block

Grids are composed of blocks. Each block is a logical unit containing a number of coordinating threads, a certain amount of shared memory. Just as grids are not shared between GPUs, blocks are not shared between multiprocessors. All blocks in a grid use t 6the same program. A built in variable "blockIdx" can be used to identify the current block. Block IDs can be 1D or 2D (based on grid dimension). Usually there are 65,535 blocks in a GPU.

The Thread

Blocks are composed of threads. Threads are run on the individual cores of the multiprocessors, but unlike grids and blocks, they are not restricted to a single core. Like blocks, each thread has an ID (threadIdx). Thread IDs can be 1D, 2D or 3D (based on block dimension). The thread id is relative to the block it is in. Threads have a certain amount of register memory. Usually there can be 512 threads per block.

Global memory

It is a read and writes memory. It is slow and uncached and requires sequential & aligned 16 byte reads and writes to be fast (coalesced read/write).

Texture memory

It is a read only memory. Its cache optimized for 2D spatial access pattern.

Constant memory

This is where constants and kernel arguments are stored. It is slow, but with cache.

Shared memory

All threads in a block can use shared memory for read or write operations. It is common for all threads in a block and its size is smaller than global memory. The number of threads that can be executed simultaneously in a block is determined the shared memory that is specified and it denotes the occupancy of that block.

Local memory

It is generally used for whatever does not fit into registers. It is slow and uncached, but allows automatic coalesced reads and writes.

Registers

This is likely the fastest memory available. One set of register memory I given to each thread and it uses them for fast storage and retrieval of data like counters, which are frequently used by a thread.

III. BACK-PROPAGATION ALGORITHM

An artificial neural network is an information processing system with certain performance characteristics in common with biological neural network. In the back-propagation neural network neurons are interconnected with each other as shown in Figure 2. The back-propagation algorithm consists of two phase which is testing and training. In the feed forward pass an input vector is presented to a network and propagated forward to the output. In back-propagation phase the network output is compared to the desired output, network weights are then adjusted in accordance with an error correction rule [8].

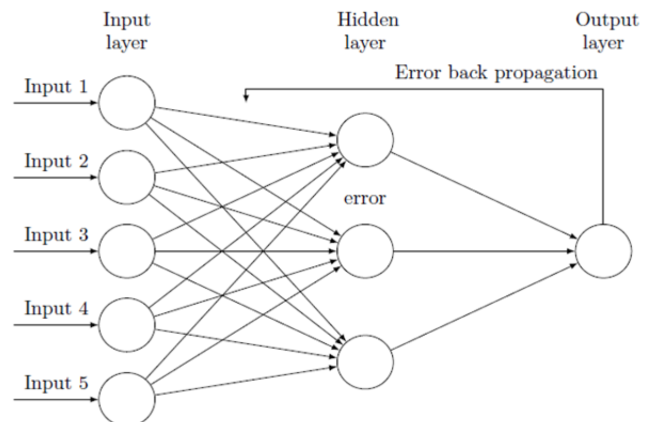

Figure2. Back-propagation neural network

To adjust the weights and biases of the neural network, a standard back-propagation algorithm is used [8][9]. The first step is to apply the input image into the network and calculate its output. This will give you an error that will be used for back-propagation. To calculate the error, the following equation is used.

E = O(1 - 0) (Target - 0) (1)

Where E is the error, O is the output of the network, and target is the desired output of the network i.e. if the image contains a quad rotor, 0 otherwise. After getting the error, the next step is to correct the weights of the output neuron. To do this, we use equation 2.

$$W_{new}=W_{old} + LR * (E * O_{hidden}) \quad (2)$$

Where O*hidden* is the output of the hidden layer neuron which the weight being calculated is associated with LR is called the learning rate, a parameter between 0 and 1 that is used to adjust the rate at which the neural network is trained.

A higher learning rate means that the neural network will learn faster, however, a learning rate too high may lead to instability of the network. After changing the weights of the output layer, the next step is to change the weights associated with the hidden layer. First, it get the error of the hidden layer. However, this is not as straightforward as getting the error for the output layer because no target for the hidden layer i.e. we do not know what its correct value must be. Therefore, we use the equation

$$Error = O_{hidden}(1 - O_{hidden})(E * W_{new}) \quad (3)$$

Where E is the error of the output neuron calculated in equation 1 and W new is the output layer weight associated with that hidden neuron. After getting the errors for the hidden layer neurons, use equation 2 to get the new weights. The input vector is composed of Hu & Zernike moments of each level of gray and geodesic descriptors on binary image to form input of the Artificial neural network which is used for object recognition [10]. The extracted vectors are put together to form a unique input data to the neural network for object recognition.

IV. GPU PARALLELIZATION OF BACK-PROPAGATION NEURAL NETWORK

GPU operates as a highly multi-threaded co-processor (device) to CPU (host). While CPU is responsible for sequential computing and logical transaction, GPU is specialized for compute-intensive, highly parallel computation. The original BP algorithm is describes in detail [3]. BPN is constructed of one input layer, one output layer and one or more hidden layers with connection between adjacent layers. A vector of neurons and a matrix of weights together with an activation function are called a layer. This procedure is replaced with many 2-dimensional matrixes assignment naturally.

At last, we summarize following solutions or principles [1]:

1. Read the 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 feed-forward kernel (input to hidden layer)—533 × 784 threads.

7. Call the feed-forward kernel (hidden to output layer)—10 × 533 threads.

8. Call the kernel to calculate the deltas—533 threads.

9. Call the kernel to update the weights (input to hidden)—533 × 784 threads.

10. Call the kernel to update the weights (hidden to output)—533 × 10 threads.

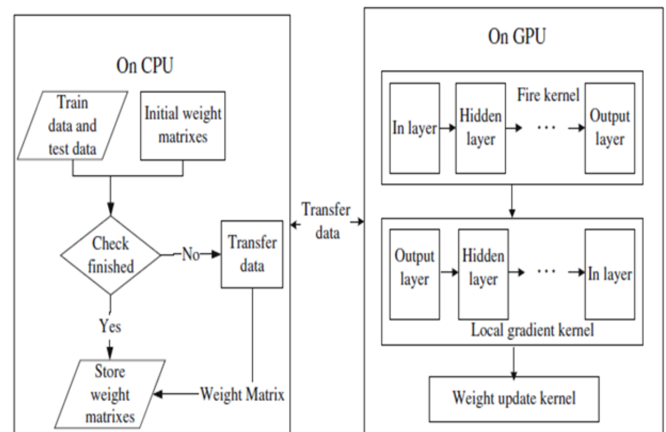

Figure 3. Overall flow of Back-Propagation Network on GPU

In Figure 3 shows the program execution flow of back-propagation neural network on GPU [3]. The feed-forward process has two kernels, one for calculating the hidden nodes (*Feed-ForwardIH*) and another for calculating the output nodes (*Feed-ForwardHO*).

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.

V. CONCLUSION

The training phase of neural network is very time consuming due to the large number of epochs and weight updates required to reach an optimal performance. GPU parallelization of neural network reduces the training time. GPU can improve the speed in comparison to the CPU version of neural network. If we need to do training on a dataset with small number of attributes the CPU version is better. GPU should only be used the data to be processed has large number of attributes to benefit a parallelism from GPU.

REFERENCES

[1] Ricardo Brito, Simon Fong, Kyungeun Cho, Wei Song, Raymond Wong, Sabah Mohammed, Jinan Fiaidhi, "GPU-enabled back-propagation artificial neural network for digit recognition in parallel", Springer Science and Business Media New York 10 feb 2016.

[2] Xavier Sierra-Canto, Francisco Madera-Ramirez, Victor Uc-Cetina, "Parallel Training of a Back-Propagation Neural Network using CUDA", IEEE Ninth International Conference on Machine Learning and Applications 2010.

[3] Yaobin Wang, Pingping Tang, Hong An, Zhiqin Liu, Kun Wang and Yong Zhou, "Optimization and Analysis of Parallel Back Propagation Neural Network on GPU Using CUDA", Springer International Publishing Switzerland S. Arik et al. (Eds.): ICONIP, Part III, pp. 156-163, 2015.

[4] Dipti Pawar, "Fuzzy Min-Max Neural Network with Compensatory Neuron Architecture for Invariant Object Recognition", IEEE International Conference on Computer, Communication and Control, 2015.

[5] Jinfeng Liu and Lei Guo, "Implementation of Neural Network Back-propagation in CUDA". Springer-Verlag Berlin Heidelberg of Intelligence Computation and Evolutionary Computation, AISC 180, pp. 1021-1027, 2013.

[6] Altaf Ahmad Huqqani, Erich Schikuta, Sicen Yea, Peng Chen. "Multicore and GPU Parallelization of Neural Networks for Face Recognition", Computer Science International Conference on Computational Science, pp: 349 – 358, 2013.

[7] Domen Verber "Implementation of Massive Artificial Neural Networks with CUDA". University of Maribor Slovenia.

[8] Reiichiro Christian S. Nakano, Argel Bandala, Gerard Ely Faelden, Jose Martin Maningo, Elmer P. Dadios."Implementation of an Artificial Neural Network in Recognizing In-flight Quadrotor Images" IEEE 2015.

[9] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar, "GPGPU Processing In Cuda Architecture" Advanced Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012.

[10] Shilpa Bane, D. R. Pawar, "Survey on Feature Extraction methods in Object Recognition", International Journal of Computer Science and Information Technologies, vol, 5, pp. 3224–3226, 2014.