Parallel Implementation of Feedforward Neural Networks on GPUs

Sáskya T. A. Gurgel
Andrei de A. Formiga
Centro de Informática
Universidade Federal da Paraíba
João Pessoa, PB, Brazil
Email: saskyagurgel@gmail.com, andrei@ci.ufpb.br

***Abstract*—Neural networks are often seen as a natural model of parallel computation, especially when contrasted with more traditional sequential models like the Turing Machine. The parallelism of neural networks has become more important in recent years through the confluence of two tendencies in the evolution of computer and information technologies: first, parallel computing devices are now ubiquitous, instead of being relegated to a niche market; and second, the amount of data available to analyze and learn from in machine learning applications has increased at a rapid pace. Graphical Processing Units (GPUs) provide great computational power in standard desktop computers, being composed of many simple execution units. In this paper a technique is presented for the parallel implementation of neural networks in GPUs. The technique is explained in relation to the difficulties imposed by the execution model of GPUs. Experimental results indicate that the proposed implementation techniques can easily attain a performance gain of more than one order of magnitude, and are scalable with the processing power of the GPU used.**

***Keywords*-neural networks; parallel; GPUs;**

I. Introduction

Neural networks achieve their goals by the composition of many simple individual computing units – the artificial neurons. As such, these networks have been commonly associated with parallel distributed computing [1] and were thought to provide a more natural model for parallel computation than the mostly sequential computing paradigm derived from the Turing Machine and the Von Neumann architecture.

Graphical Processing Units (GPUs) are processors evolved from the graphics cards created to accelerate 3D rendering in personal computers [2]. GPUs are composed of many simple computing cores, instead of the few complex cores available in current multicore processors. It is thus natural to think of GPUs as a good fit for the neural network model of computation. GPUs also provide computational power rivaling supercomputers of a few years past [3], being desirable for applications that need to analyze great amounts of data.

Among all neural network types and variations, the most well known – and one of the most widely used – are feedforward networks with multiple layers of nodes, trained with backpropagation [4]. One frequently cited drawback of this kind of neural network are the long training times required to achieve good performance. An efficient parallel implementation of training using backpropagation may make it possible for the neural network designer to iterate and experiment more quickly with the network architecture, achieving better results. In production, a parallel implementation of the network is also able to process many problem instances in parallel in a shorter time.

However, the architecture and execution model of GPUs present challenges to the implementation of feedforward neural networks that can nullify the potential gains of the parallel execution. In this paper are presented a small set of techniques for the implementation of neural networks on GPUs that avoid most of the possible problems and achieve good time performance in relation to execution on a CPU. A concrete implementation of the proposed techniques is also described[1], along with experiments made to assess its performance. The results indicate that performance gains over CPU execution depend on the size of the network and the amount of data used for training the network, with larger sizes benefiting the GPU more.

The paper is organized as follows: in Section II feedforward neural networks are briefly described, while a similarly brief treatment of the architecture and execution model of current GPUs are the subject of Section III. Section IV presents the implementation techniques that allow neural networks to be realized efficiently in GPUs. Section V then presents a group of experiments conducted to assess the time performance characteristics of the neural network implementation, and the analysis of the obtained results. Finally, Section VII concludes the paper with last considerations and suggestions for further work in this line of research.

II. Neural Networks

Neural networks are learning machines traditionally associated with supervised learning techniques for classification and regression. A network is composed of a collection of nodes called *(artificial) neurons*. A neuron is a computing element that a number of inputs and a single output; the

[1]The complete implementation and the data used for the experiments are publicly available at the address https://github.com/NeuroGPU/neurogpu/.

978-0-7695-5092-3/13 $26.00 © 2013 IEEE
DOI 10.1109/BRACIS.2013.32

output is based on a linear combination of the inputs which is transformed by an activation function. For a neuron with N inputs $x_1, x_2, \ldots, x_N$ and output y, its output is given by

$$y = a\left(\sum_{i=1}^{N} w_i x_i\right), \tag{1}$$

where $a(x)$ is the activation function. Common activation functions are the sign function, a thresholding function or the sigmoid function $s(x) = 1/(1+e^{-x})$. The inputs x_i can come from the inputs of the network or the outputs of other neurons.

A feedforward neural network is organized in *layers*, with the nodes in a layer receiving their input from the preceding layer, and with the outputs of a layer going as input to the next layer. The outputs of the last layer (usually called the output layer) are the outputs of the network. In feedfoward networks, computation flows only forward in the network, from one layer to the next; in contrast, in recurrent neural networks loops can be formed.

The neural network is trained in a way such that the weights for each node are adjusted to make the outputs of the network match a desired function, either a discrete function (in the case of classification) or a continuous function (in the case of regression). The training process is usually performed as an optimization of the weights subject to a loss or error function. A common method of training feedforward neural networks is the *backpropagation* algorithm [4]. This method consists of flowing the information of the derivatives of the error in relation to the weights of the network backwards, from the output layer to each preceding layer in turn.

III. General Computing on Graphics Hardware

The current Graphical Processing Units (GPUs) evolved from graphics hardware intended for acceleration of 3D rendering in personal computers, mostly used for real-time 3D rendering in games [2]. With time, more parts the rendering pipeline were implemented in the graphics processors, to the point where such processors became quite capable as computing devices. Eventually, the manufacturers of graphics hardware developed programmable graphics processors that could be programmed by users, via software (using the so-called *shaders* [5]). This led some researchers to the idea of using GPUs as general computing devices, by adapting the graphics shaders to non-intended uses [6]. Manufacturers then established standards and tools created to make it easier for programmers to use their hardware as general computing devices: NVIDIA created CUDA [2], while AMD and other companies developed OpenCL [7] as an open standard.

The evolution of GPU hardware from graphics accelerators resulted in an architecture that is quite distinct from the processors used as CPUs in current computers. GPUs have

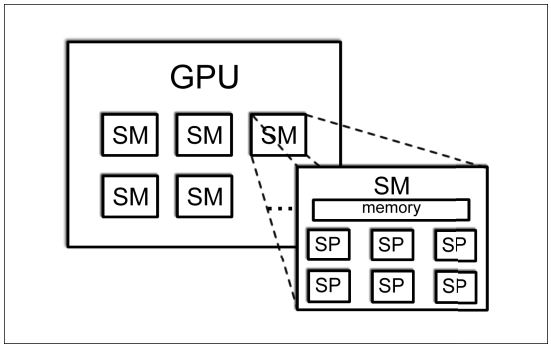

Figure 1. Overall architecture of a typical GPU. The GPU contains multiple Streaming Multiprocessors (SMs), each of which is composed of multiple Streaming Processors (SPs) or cores.

many simple processing cores – totalling hundreds of cores in current hardware – instead of the few complex cores in current CPUs. GPUs are massively parallel processors that are able to execute large parallel tasks much more efficiently than CPUs. However, each group of cores execute the same instructions at the same time, making GPUs a poor fit for applications that do not present data parallelism [3].

Even considering the differing standards from manufacturers, the architectures of GPUs have converged to similar lines. GPUs are composed of a number of Streaming Multiprocessors (SMs). Each SM encases many processing cores or Streaming Processors (SPs), typically 32 in current hardware [3]. SMs are independent units of execution, dispatching threads of execution to their SPs, one thread per core. The general architectural organization of GPUs is displayed in Figure 1. In most cases, the SMs can access only memory that is located in the GPU. This means that any data that must be processed in the GPU must be transferred from the system main memory to the GPU memory.

Each process that is to be executed on the GPU is called a *computational kernel*, or simply a kernel. In CUDA C, the version of the C language for programming the CUDA standard from NVIDIA, each kernel is called as a function. All cores in a Streaming Multiprocessor will execute the same instructions, in lockstep, although the instructions can access different memory locations. Each SM is thus a SIMD (Single Instruction, Multiple Data) computing machine, although as the GPU contains multiple SMs, the execution model is more appropriately called Single Instruction, Multiple Threads (SIMT) [3]. Conditional instructions (for example in *if-then-else* constructs) which force threads in the same SM to different paths of execution cause *thread divergence*. In practice, thread divergence is implemented by making some cores in a SM idle while others execute the conditional instruction, reducing overall throughput.

The architectural organization of GPUs makes it clear that efficient execution on graphical hardware requires programs

which are organized very distinctly from programs designed for CPU execution. Two general principles for efficient execution on GPUs are to keep the processors busy and to avoid thread divergence. The implementation techniques for feedforward neural networks on GPUs described in the following section were designed to adhere to these two general principles.

IV. Implementation Techniques

Both neural networks and GPUs are composed of many simple computing units. This similarity may suggest, as a first idea for a parallel implementation of neural networks in GPUs, to map each node in the neural network to a GPU thread, thus executing the whole network simultaneously in parallel. Analysis of this idea reveals its inadequacy for implementation in GPUs: the non-input layers in a feedforward neural network depend on a neighboring layer for its processing, both during training and during production use. For the forward propagation process, which is used to perform classification or regression on a trained network, the second hidden layer needs the output of the first hidden layer to calculate its own outputs, the third layer depends on the second layer, and so on, up to the output layer. For training the network using backpropagation the dependencies are reversed, with the last hidden layer depending on the output layer and so on, up to the first hidden layer which depends on the second hidden layer. Thus, data in a feedforward network always flows in a single direction, one layer at a time.

This means that if the whole network is mapped to the GPU by executing each node in a thread (and thus in a GPU core), all the network nodes that are not in the layer currently being computed must be inactive, waiting for some data to become available. Depending on the architecture of the network, this implementation strategy would mean that most of the GPU would be inactive at any given time, thus severely impairing the performance of such a parallel implementation.

Most often, a neural network is used (for regression or classification) and trained on multiple input cases comprising a dataset. For training it is usually desirable to have a good number of representative cases to make the network learn a general function mapping inputs to outputs. Although the network can be used, in practical contexts, for classification or regression of a single input case, in many situations it is more desirable to collect many input cases and operate on them in batch.

For this reason, the proposed technique for parallel implementation of neural networks in GPUs is to compute each layer separately, in the required order, but operating on all available input cases at the same time, in parallel. The paralellism occurs between different input cases and on the same network layer, instead of between nodes and layers of the whole network as in the first idea. This idea is illustrated in Figure 2, where it is shown a network with three layers (not counting the input layer), with two neurons in the first hidden layer, three neurons in the second hidden layer and one neuron in the output layer. After each layer is processed for all available input cases, a new parallel kernel is invoked to process the next layer in the required order (the next layer for forward propagation, the preceding layer for backpropagation). With realistic datasets of considerable size, this makes maximal use of the GPU without data dependencies or thread divergence, making the resulting implementation efficient for GPU hardware. The next few subsections give more details about the parallel implementation of neural networks using this organization, both for network use with forward propagation and training with backpropagation. Each part of the process is implemented as a parallel kernel that must be launched into many threads on the GPU device. The following subsections present implementation details of each kernel, including memory organization aspects.

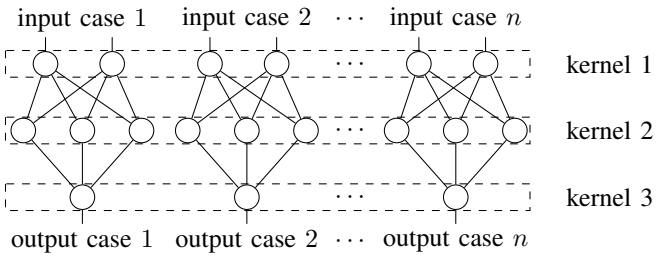

Figure 2. Organization of GPU kernels for parallel neural network execution of a network with 3 layers. Each GPU kernel is responsible for the calculation of a single layer of the network, but with all input cases in a dataset processed in parallel.

A. Forward Propagation

For the use of a neural network as a classification or regression device it must be already trained previously, in a way such that the connection weights between nodes make the network compute the desired function. In this case, using the network entails presenting one or many input cases to it and observing the network outputs. The input case is presented as input to the first hidden layer, and this is used to compute the outputs of the layer; these outputs of the first hidden layer are presented as inputs to the second hidden layer, making it possible to calculate the outputs of the second hidden layer. This process continues until the outputs of the last layer are computed, which are the network outputs. Forward propagation is also used during supervised training, to compute the network outputs for known cases and comparing to the known outputs.

For parallel implementation in a GPU, as discussed above, each layer will be computed in turn, processing all the input cases in parallel. There is a single kernel for forward propagation in a layer, and this kernel is called in turn for

each layer in order: first for the first hidden layer, then for the second hidden layer and so on, up to the output layer.

The forward propagation kernel for a single layer uses as inputs the network weights and the inputs for each neuron, and produces as output the output levels of each neuron. The inputs and outputs arrays must be organized in a way that keeps together all the input cases in the same array. The inputs for the first hidden layer are the attributes from the input case, while the inputs for the subsequent layers are the outputs of the preceding layer. With the input and output arrays, and the network weights, it is simple to implement the computation of node outputs, as shown in Equation (1).

The algorithm for the forward propagation kernel for a single layer is shown as Algorithm 1. This kernel must be launched in a grid with a number of blocks equal to the number of input cases, and the number of threads per block equal to the number of nodes in the layer. The GPU will execute a copy of this kernel for each thread, identifying the copies only by the thread index tix. The kernel inputs are: ws is the matrix with network weights; off is the offset into the weight matrix for the layer weights; wpn is the number of weights per node for the layer; ins is the input array for the layer; nin is the number of inputs; and $outs$ is the output array for the layer. Note that the output array for a layer will become the input array for the next layer. The forward propagation kernels will be launched for each layer in turn, until the outputs of the last layer are computed.

Algorithm 1 Algorithm for forward propagation kernel of a single layer.

Launch grid: (nc, nn) *for* nc *cases and* nn *nodes in layer*
procedure FORWARDLAYER($ws, off, wpn, ins, nin, outs$)
  $oix := tix$ ▷ tix is the thread index
  $iix := case \times nin$ ▷ $case$ is the block index
  $wix := off + tix * wpn$
  $a := ws[wix]$ ▷ bias input
  for $i = 1$ to wpn **do**
    $a := a + ws[wix + i] * ins[iix + i]$
  end for
  $os[oix] := sigmoid(a)$
end procedure

B. Training with Backpropagation

The backpropagation algorithm for training a neural network is an optimization procedure to change the network weights to minimize the value of a loss or error function. The error function is usually the sum of squared differences between the desired output and the computed output.

The traditional backpropagation algorithm is composed of two steps: first the derivatives of the error function in relation to the weights are calculated, and then these derivatives are used to update the network weights in a way similar to gradient descent. The calculation of derivatives is broken into two processes: calculation of a *delta* value for each node in the network, and finally the computation of the derivatives of the error function in relation to the weights, using the delta values obtained previously. The whole training process is repeated over many iterations, called *epochs* in the neural network literature. Equations (2), (3) and (4) show the formulas for calculation of delta values for nodes in the output layer, delta values for nodes in hidden layers and the derivative of the error function in relation to the weights for each input case, respectively.

$$\delta_i = -(d_i - y_i)a'(y_i) \tag{2}$$

$$\delta_i = \sum_k w_{ki}\delta_k a'(y_i) \tag{3}$$

$$\frac{\partial E_p}{\partial w_{ik}} = \delta_i y_k \tag{4}$$

1) Calculating the Deltas: A delta value must be calculated for each neuron and each input case. The delta for output layer nodes is proportional to the difference between the desired value and the computed network outputs, while for hidden layer nodes the delta depends on the deltas of the nodes in the next layer.

Two kernels implement delta calculation: one kernel for computing deltas for the output layer, and other kernel responsible for the deltas of the hidden layers. Data flows backwards in delta calculation, from the output layer to the preceding layer and so on, up to the first hidden layer.

The kernel for the output layer needs access to an array to store the deltas, the array of layer outputs and an array containing the expected outputs for each neuron. In a supervised learning scenario, the expected outputs are provided with the training dataset. For the hidden layers, the kernel needs acces to an array to store the deltas for the current layer, the array of calculated deltas for the next layer, and the network weights together with the associated offset and number of weights per neuron of the next layer.

With access to the data, the kernels for delta calculation implement equations (2) and (3), in a way similar to Algorithm 1 for forward propagation.

2) Error Derivatives: The derivatives of the error function in relation to the weights are used to update the weights in order to reduce the value of the error function. As seen on Equation (4), this is first calculated for each weight and each input case. The kernel that calculates $\frac{\partial E_p}{\partial w_{ik}}$ will compute Eq. (4) for each weight and each input case, and it needs access only to the correct delta value for the node corresponding to weight w_{ik}, and the correct input value for this weight. In contrast to the other kernels, the kernel to calculate error derivatives must be launched with a grid composed of a number of blocks equal to the number of

input cases, and a number of threads per block equal to the number of weights in the whole network.

3) Weight Update: Given the derivatives of the error in relation to the weights, it is possible to update the network weights in the direction which causes the maximal reduction in the value of the error function. This approximates a gradient descent procedure for minimization.

After the kernels for computing error derivatives for each weight and each input case are finished, the overall derivatives for all input cases must be calculated as the sum of the derivatives of each case. In terms of parallel computation this is a set of reduction procedures, one for each weight. This reduction implements Equation (5) for each weight.

$$\frac{\partial E}{\partial w_{ik}} = \sum_{p} \frac{\partial E_p}{\partial w_{ik}} \tag{5}$$

The network weights must be updated according to the overall derivatives calculated as in Eq. (5). As in gradient descent procedures, the new value of the weight will be equal to the old value minus the derivative of the error multiplied by a constant *learning rate*. The rate regulates how much the weights will be changed in each iteration. The weight update follows Equation (6), where η is the learning rate.

$$w'_{ik} = w_{ik} - \eta \frac{\partial E}{\partial w_{ik}} \tag{6}$$

The kernel for weight update needs access to the array of overall derivatives resulting from the reduction, and to the network weights. Given the value of the learning rate, η, the kernel implements Equation (6) directly.

V. Experiments and Results

The implementation techniques described in Section IV were realized in a CUDA C implementation for testing and performance assessment[2]. A sequential implementation of neural networks in C was also created, to serve as basis for comparison.

Both implementations were tested on a computer equipped with a Intel Core i5-750 processor (2.66GHz, 8Mb cache), 4Gb of DDR3 1333MHz RAM and a GeForce GTX 550 Ti graphics card (192 cores) with 1Gb of video memory. The test machine is not a high-performance computer, using parts that are widely available at the time of writing for relatively low prices. Even then, the performance comparison between the CPU and GPU versions point to performance characteristics that are valid for high-performance hardware, and would probably favor the proposed GPU implementation of neural networks even more.

Both implementations were trained to perform classification based on three datasets of varying size, both in number of input cases and number of attributes per case. The first dataset was the extremely simple case of a network to perform the logical exclusive-or (XOR) operation, while the other two datasets were taken from the UCI Machine Learning Repository[3]: the classic *iris* dataset from R. A. Fisher and the *adult* dataset of census data. The results for training networks for the three datasets using both versions are shown in Table I.

Table I
Time performance for training of three datasets using CPU and GPU versions of neural networks.

Dataset	Instances	Attributes	CPU	GPU	Speedup
xor	4	2	0.03s	0.7s	0.04286
iris	105	4	1.72s	0.92s	1.86956
adult	32561	14	49m12s	1m16s	38.842

Looking at the results, it is clear that the performance gains obtained by using the GPU increase with dataset size. The GPU implementation of neural networks performs better than the CPU version except on the extreme case of the XOR network, with only four cases and two attributes. For a small dataset (iris), the GPU version is about twice as fast as the CPU version, while for a moderately-sized dataset (adult) the GPU version is about 38 times faster. In all cases, the predictive performance of the trained networks (measured by a test dataset with data unrelated to the training set) was nearly identical for the GPU and CPU versions.

To further investigate the relative performance characteristics of the GPU and CPU versions, a second experiment was performed using the adult dataset. In each case, a sub-dataset was formed by randomly selecting a number of input cases from the adult dataset, and then the network was trained using the sub-dataset. The results of this experiments are shown in Table II. GPU times and speedup from Table II are illustrated in Figure 3.

Table II
Time performance of training neural networks with increasing number of input cases randomly selected from the adult dataset.

Instances	CPU	GPU	Speedup
10	970ms	466ms	2.08154
100	9.24s	472ms	19.57627
1000	1m31.7s	2.67s	34.34457
10000	15m19.2s	24.55s	37.44196

The results shown in Table II and Figure 3 indicate that, as expected, the speedups obtained by using the proposed GPU implementation of neural networks grow with dataset size; however, it is possible to see that the speedup saturates, for this test machine, at approximately 38 times. Speedup for 10 thousand instances from the adult dataset is about

[2]CUDA C is a language specific to NVIDIA graphics hardware, but the implementation can be easily ported to OpenCL, which is available for a wide range of graphics hardware from many suppliers.

[3]http://archive.ics.uci.edu/ml/

37.4 (as seen in Table II), while for the complete training set of 32 thousand instances the speedup is about 38.8. This indicates that the processing cores of the GPU are being completely utilized, and further parallelism is not improving performance significantly. The line for GPU times in Figure 3 clearly show a slow growth up until a thousand instances, and then a sharper growth after the saturation point. It is expected that changing the GPU to one with more cores would scale the speedups to even larger datasets, that is, the implementation techniques proposed in this paper scale with the available hardware, taking advantage of the processing power made available by the GPU.

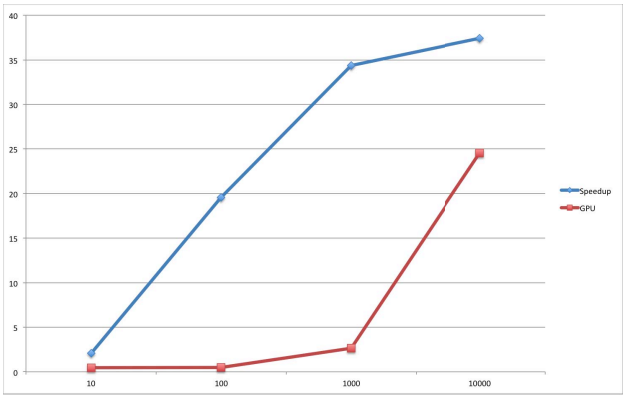

Figure 3. GPU times (red) and speedup (blue) over CPU version for training a neural networks with increasing number of input cases randomly selected from the adult dataset.

VI. Related Work

Parallelization of neural networks has been a topic of research for a long time. Farber [8] describes a parallel implementation of feed-forward neural networks on the Connection Machine, a massively-parallel supercomputer. The Connection Machine was also composed of many simple processing units organized in a SIMD architecture, similar to current GPUs, so there are similarities between Farber's solution and the one presented in this paper. However, many implementation strategies must be adapted for the GPU architecture.

More recently, there has been a specific interest in implementing neural networks on GPUs. Many previous works (e.g., [9], [10], [11]) present implementations of neural networks on GPUs using shader programming, an old form of GPU programming that was designed for graphics programming. As a result, these implementations tend to be less capable and efficient, while also being more difficult to work with, than the one described in this paper. Shader programming also dictates a completely different implementation structure in relation to the use of the more recent GPU programming systems like CUDA. The implementation described in this paper is thus better as a basis for further improvement and experimentation with other neural network architectures.

It's also possible to find software implementations of neural networks on GPUs (e.g. [12]) but they do not include a description of the implementation techniques and thus are more difficult to use as a basis for further research work.

VII. Conclusion

In this paper were presented a set of techniques that allow for efficient implementation of feedforward neural networks on graphics hardware (GPUs). The architecture of GPUs is very distinct from standard CPUs, and thus efficient implementation on graphics hardware require programs which are structured in a different way. The naive idea of mapping each neuron in a network to a single computing core does not work because of data dependencies between network layers, which would leave most of the GPU cores idle at any given moment.

The techniques proposed in this paper are based on the idea of executing each layer in parallel for many input cases at once, both for training and for network use. An implementation using the proposed techniques was created using CUDA C, and experimental results show that the GPU implementation is orders of magnitude faster than the corresponding CPU code. Furthermore, the implementation is scalable with the graphics hardware used, meaning that GPUs with greater computing power (in terms of more cores) will improve performance for larger datasets. For smaller datasets, experiments show that the advantage of using GPUs are less pronounced, due to the overheads associated with kernel launching and data transfer to and from the GPU memory. For many large datasets of interest, the drastically reduced training times are useful to network designers, as they are able to see the results of tuning the network parameters and architecture much faster, and thus perform many more experiments.

The work reported herein can be expanded in a number of ways. One idea is to employ other weight optimization procedures instead of backpropagation, and see how they perform in a massively parallel implementation. Another possibility is to implement other types of neural networks on graphics hardware; recurrent neural networks, Restricted Boltzmann Machines and Convolutional Neural Networks have received a great deal of attention from researchers recently, due to their connection with Deep Learning techniques. Implementing them on GPUs would enable a faster turnaround for experimentation, and also an assessment of which techniques fit better on massively parallel hardware.

A larger objective for further investigation is implementing other machine learning techniques and assessing how well they fit into the architecture of GPUs. When time performance is considered in the selection of machine learning techniques for a given task, the usual understandings of

relative time performance between known techniques are based on sequential execution. Executing machine learning algorithms on a massively parallel environment like GPUs has the potential to change the time performance assessment between them, depending on which algorithms can be mapped well into the SIMT architecture of GPUs.

REFERENCES

[1] J. L. McClelland and D. E. Rumelhart, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (Volume 1)*. Bradford Books, 1987.

[2] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[3] R. Farber, *CUDA Application Design and Development*. Morgan Kaufmann, 2011.

[4] R. D. Reed and R. J. Marks-II, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, 1999.

[5] W. Engel, *Programming Vertex & Pixel Shaders*. Charles River Media, 2004.

[6] J. Fung and S. Mann, "Using multiple graphics cards as a general purpose parallel computer: Applications to computer vision," in *Proceedings of the 17th International Conference on Pattern Recognition (ICPR2004)*. IEEE Press, 2004.

[7] T. G. M. Aaftab Munshi, Benedict Gaster and J. Fung, *OpenCL Programming Guide*. Addison-Wesley, 2011.

[8] R. Farber, "Efficient modeling of neural networks on massively parallel computers," in *Proceedings of the Third International Workshop on Neural Networks and Fuzzy Logic*. NASA, 1992.

[9] O. Kyoung-su and K. Jung, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, pp. 1311–1314, 2004.

[10] Z. Luo, H. Liu, and X. Wu, "Artificial neural network computation on graphic process unit," in *International Symposium on Neural Networks*, 2005.

[11] T. yui Ho, P. man Lam, and C. sing Leung, "Parallelization of cellular neural networks on GPU," *Pattern Recognition*, vol. 41, pp. 2684–2692, 2008.

[12] A. Krizhevsky, "cuda-convnet," https://code.google.com/p/cuda-convnet/.