GPU Acceleration of Sparse Neural Networks

Aavaas Gajurel[†], Sushil J Louis[†],
Rui Wu[§], Lee Barford[†‡], Frederick C Harris, Jr.[†]

[†]*Dept. of Computer Science and Engineering*
University of Nevada, Reno
Reno, NV

[§]*Dept. of Computer Science*
East Carolina University
Greenville, NC

[‡]*Keysight Laboratories*
Keysight Technologies
Reno, NV

avs@nevada.unr.edu, sushil@cse.unr.edu,
wur18@ecu.edu, lee.barford@ieee.org, fred.harris@cse.unr.edu

***Abstract*—In this paper, we use graphics processing units(GPU) to accelerate sparse and arbitrary structured neural networks. Sparse networks have nodes in the network that are not fully connected with nodes in preceding and following layers, and arbitrary structure neural networks have different number of nodes in each layers. Sparse Neural networks with arbitrary structures are generally created in the processes like neural network pruning and evolutionary machine learning strategies. We show that we can gain significant speedup for full activation of such neural networks using graphical processing units. We do a prepossessing step to determine dependency groups for all the nodes in a network, and use that information to guide the progression of activation in the neural network. Then we compute activation for each nodes in its own separate thread in the GPU, which allows for massive parallelization. We use CUDA framework to implement our approach and compare the results of sequential and GPU implementations. Our results show that the activation of sparse neural networks lends very well to GPU acceleration and can help speed up machine learning strategies which generate such networks or other processes that have similar structure.**

***Keywords*—GPU, Neural Networks, CUDA, Graph processing**

I. Introduction

Artificial neural networks, first proposed by McCuloch and Pitts in 1943 [1], are universal function approximators loosely based on biological neural networks. Neural networks with back propagation [2] is a robust method in machine learning. Artificial neural networks(ANN) have interconnected nodes that are separated into three types - inputs, outputs and hidden nodes. Inputs nodes are sensor nodes that take in values from outside the system, output nodes are the nodes that produce the answers from the network and hidden nodes are the nodes which lie in the information propagation path of the neural network. Each node is activated based on the nodes from which it has incoming connections, and the activation is calculated by weighting all the incoming node values with corresponding connection weight and summing all the values. The sum is then thresholded for the final activation. Generally the sum is passed through sigmoid function to constrain it within -1 and $+1$ values.

$$sigmoid(x) = (1/(1 + e^{-4.97*x}))$$

Conventionally, neural networks have structure which consists of nodes cleanly segmented into layers as shown in Figure 1 with incoming nodes shown in red and outgoing nodes shown in green. Nodes in each layer are not connected with each other and are fully connected with the nodes in preceding layer and the following layer. This structure lends very well to vectorization where each layer is represented by matrices of weights and the propagation and activation can be calculated with the multiplication of the layer matrices. This property is utilized for GPU acceleration of such neural network. As GPUs are well suited for large in matrix multiplications, such neural networks have seen large speedups, even for huge networks [3].

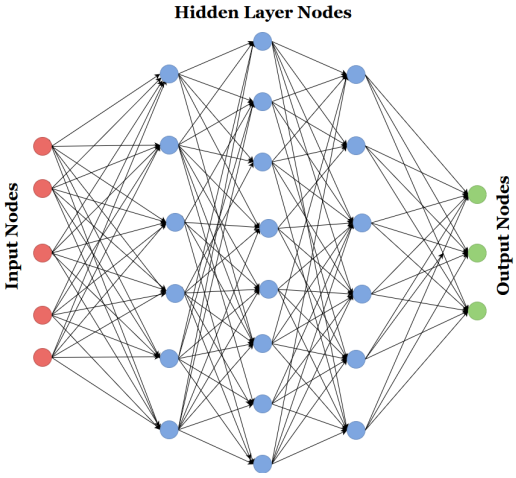

Fig. 1. Representation of conventional feed forward neural network with input, hidden and output layers

This paper is concerned with the activation of sparse and arbitrary structured neural networks. Neurons in sparse neural networks do not have full connection with nodes from preceding layer and following layer. Neural networks have arbitrary structure when nodes from sparse networks are also pruned. In such case, such networks cannot be cleanly separated into layers, *i.e.* they are not fully connected and can have incoming and outgoing links to any node in the graph as shown in Figure 2. Sparse networks are a subset of arbitrary structured neural networks(ASNN)s and are generated by neural network pruning algorithms [4], [5]. ASNNs can be generated by pruning both connections and nodes from fully trained dense networks. They are also created by rule based network structure generators, and some of which are applied in machine learning to generate networks best fit for a given problem. Neural Evolution of Augmenting Topologies (NEAT) [6] with direct encoding, HyperNEAT [7] which uses generative

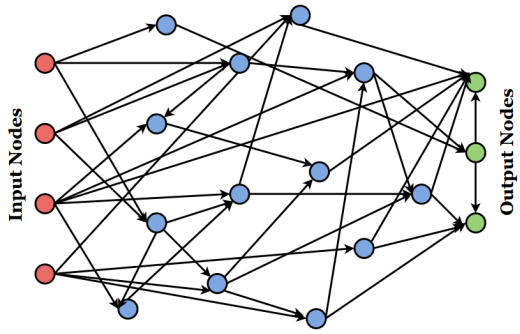

Fig. 2. Representation of neural network with arbitrary structure

encoding, and grammar based substitution and bi-directional growth encoding [8] are some of the few processes which can generate ASNNs. With the start of application of neuro evolution to deep neural networks [9], full propagation of these network will take significant portion of compute time of total program running time. GPU acceleration methodologies used for conventional NN will not work with ASNNs, thus, if we can use GPU to accelerate arbitrary neural networks, we can have considerable gains in speed and both memory and power efficiency.

Graphics processing units were originally developed as a co processor alongside the main CPU, to offload the processing of graphics related tasks which are are massively parallelizable. With co-accelerated advancements in games and GPU hardware, the GPU architecture reached a point where they could be used for general purpose computation. Initially, problems for general purpose computation in GPU(GPGPU) had to be converted into OPENGL shader programs utilizing non standard methods [10] but with eager support from GPU manufactures, platforms for general purpose computation on GPU were created. Compute Unified Device Architecture (CUDA) [11] and OpenCL [12] being two prominent examples. With vendor support and capable frameworks, GPUs have become a useful tool for massive data parallel problems.

This new found power of general purpose computation on GPUs have been extensively utilized in neural network development and research. One of the reason for deep learning growth has been attributed to advancement in GPU capability [13]. There are numerous GPU libraries supporting neural network acceleration [14] and frameworks around them [15]. But GPU acceleration on arbitrary structure neural networks is still lacking; which we explore in this paper. For GPU acceleration of ASNN, we first perform a pre processing step that segregates all the nodes into dependency hierarchy which we call layers. Then we use thread parallalization available with CUDA interface to compute all the nodes in that layer at the same time. With our methodology, we have shown that we can get significant speedup with the use of GPUs and the speedup gets better with respect to increase in neural network connections and depth of the network.

Remainder of this paper is organized as follows. Section II describes previous approaches related to our current work, section III describes the neural network representation, sequential and GPU activation methodology and experimental setup. In Section IV, we describe our results and compare the timing and speedup between two strategies. lastly in Section V, we draw our conclusions and explain possible future directions.

II. Related Work

GPUs have been used for general purpose computation from before the time they provided formal support for it. Before APIs like CUDA and OPENCL were offered by the GPU vendors, researchers utilised OPENGL shader languages to coerce the GPU into performing non graphical processing [10], [16]. The application of GPUs for accelerating data parallel tasks has only increased after the introduction of the supported APIs. NVIDIA maintains a host of different libraries targeted to various application domains like deepLearning, signal processing, linear algebra and others [17].

The power of GPU have been eagerly utilized in the machine learning field as machine learning requires crunching through big numerical calculations and large number of iterations for making sense out of big datasets [18]. GPUs have been applied to Neural networks, and have been especially useful with the advent of deep learning, which uses neural networks with large number of neurons and hidden layers [19]. Previous work on implementation of neural networks in GPUs have started before the introduction of CUDA, where the authors utilized texture processing pipeline of the GPU to accelerate Multi layer perceptron and self organizing maps with significant speedup [20]. Scherer et. al. have shown in [21] that GPU can have gains of up to two orders of magnitude for convolution neural networks. Cheltur et. al. have also shown that convolutional neural networks can be efficiently computed in GPUs with data framing in a form of matrix and performing matrix multiplication to compute the network [22]. Coates et. al. have shown that many consumer grade GPUS in separate machines can be used for acceleration of convolutional neural networks by using CUDA, and using openMPI for multi GPU coordination [23]. Zhang et. al. have also looked at accelerating sparse neural networks with custom hardware accelerators [24].

Other types of neural networks have also seen good results from GPU implementation. Nageswaran et. al. have implemented a configurable simulation environment for the efficient simulation of large scale spiking neural networks on GPU [25]. Juang et. al. have also shown significant speedup on fuzzy neural networks with high dimensional inputs by using parallel processing on GPUS [26], and GPUS have also been able to reduce recurrent neural networks training time by a factor of 32 [27].

Neural network in general form are also graph structures, and there have also been numerous research on graph processing on GPUs. Luo et. al. showed that the speedup of upto 10x could be achieved with GPU implementation for breath first search [28]. Harish and Narayan give implementation of various graph processing algorithms on GPU in [29] and note that in some cases sequential approach does not transfer well to the GPU approach. In this paper, we are looking into networks with non uniform structure and have to perform a pre processing step on that structure to segregate the nodes where we have to apply graph processing approaches.

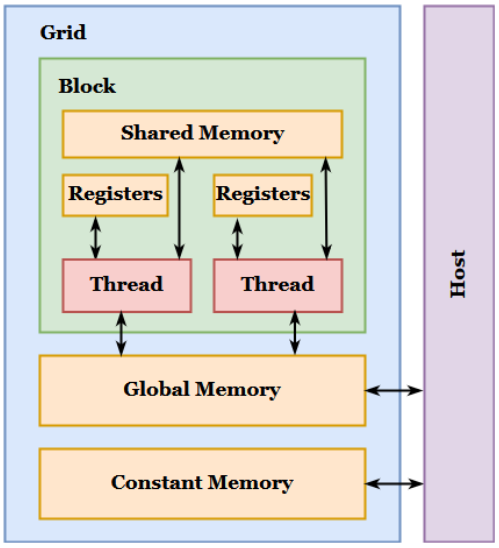

Fig. 3. Cuda memory architecture showing the relation between grid, block and threads and the corresponding proximity and connections of different kinds of memory elements

Algorithm 1 Network segmentation algorithm

```
function SEGMENT_NETWORK(R, IN, OP, CON)                    ▷
R=required, IN=inputs, OP=outputs, CON=connections
    L ← []
    s ← IN
    while True do
                                  ▷ Candidate nodes for the next layer
        c = { b for (a,b) in CON if a in s and b not in s}
                        ▷ Used nodes whose entire input set is in s
        t ← {}
        for n in c do
            all ← (for a in s for (a,b) in CON if b = n)
            if n in R and all then
                t ← t U n
            end if
        end for
        if t = φ then
            break
        end if
        L ← L + t
        s ← s U t
    end while
    return L
end function
```

III. Methodology

A. CUDA

The CUDA application programming interface provides a way to structure our operations to run on Nvidia GPUs. The memory model in CUDA is divided into grids, blocks and threads which have access to specific kinds of memory and are all interfaced with the CPU, called a host, via a PCI bus as shown in Figure 3. Code execution can be segmented to run in grids and blocks, both of which can be molded to have one to three dimensions depending on the problem. Each block runs the kernel, a block of CUDA procedure, in individual threads. Threads within a block can share a portion of memory called shared memory, which has very low latency as it resides on the chip. Current GPUs can run 32 threads in a block, which is also called a warp, at a single time where same instructions of a kernel executes on all the threads but runs on different data. Optimizing for efficient allocation of warps could lead to better performance. Threads in a block can be synchronized with __syncthreads() API call which syncs all the threads in a block at the code location where all of them execute __syncthreads(). Concept of unified memory was introduced in CUDA 4.0 which does away with the manual process of memory copying from device to host and back. Now, we allocate a portion of memory that is shared between both device and host and GPU driver takes care of transfer of data when data is accessed from either device. We use cudaMallocManaged() to allocate shared memory and use cudaDeviceSynchronize() before accessing any device data from the host.

B. Sequential activation

For sequential activation of arbitrary neural network, we first perform pre processing on network structure to segment the network into sequential hierarchy of nodes, which we call layers of the ASNN. The algorithm to segment the network is given in Algorithm 1. The function takes all nodes, input nodes, output nodes and a structure containing all the network connections as input for processing. First, we find candidate nodes from the connections pool based on the nodes that have already been assigned to some layer. The candidates are those nodes for which incoming nodes all lie on the nodes that have already been assigned to a layer and all its outgoing nodes are not in the assigned set. From the candidate set, we only add those nodes to the next layer if their entire input set is contained in the nodes which are already assigned a layer value.

For sequential propagation of the neural network, All the nodes starting of the input layer is sequentially activated till the output layer from which the answer is obtained. The activation is calculated by going through all the incoming nodes and multiplying the connection weights with the node values and then summing them and then squashing them with the sigmoid function.

C. Parallel GPU activation

For single GPU activation of the neural network, we use the fact that all the nodes belonging to the same layer can be activated at once without compromising the output of the network in any way. For representation of the structure of network in the GPU, we use a custom data structure called CudaNode as depicted in Algorithm 2. Each CudaNode structure represents a single node of the neural network where each node contains a unique id, the number of incoming nodes, the integer array containing the node ids for the incoming connections and another float array for the corresponding weights for the incoming nodes. Then, we also have a Boolean that specifies if the node is a sensor i.e. takes external input for the network. The layer variable is set by the prepossessing of the neural network sequentially. The array of CudaNode structs are sorted in ascending order based on their layer number,

Algorithm 2 Data structure to represent a single node in a network

```
1: struct CudaNode
2:     Integer: id                          ▷ Unique id for the node
3:     Integer: layer                       ▷ Layer the node is in
4:     Integer: numInNodes                  ▷ No. of incoming nodes
5:     Boolean: isSensor                    ▷ True if input node
6:     Integer[]: inNodes                   ▷ Array of incoming Node ids
7:     Float[]: inWeights                   ▷ Array of incoming Node
   weights
8: end struct
```

where the input layer starts with value 0 and then climbs up to the last layer in the network. This is done to allow for better cache performance for unified memory in the GPU as input and output nodes will be close to each other in the array after being sorted.

CUDA kernel for GPU activation is described in Algorithm 3. The kernel for GPU activation takes the value for total number of layers in the network, another integer array containing number of nodes in each layer, the main CudaNodes array containing sorted node entities, and an input array of floats which contains values for input layer of the neural network. Size of the output float array is equal to the size of number of nodes in the network as each node writes its activation result to this array, which all other nodes will also be able to observe and write to. We have a variable cl which determines the current layer that the kernel is processing and sid, the start id which holds the id of the first node of the layer being computed. We will already have spawned many threads which will correspond to one node of the layer being activated. In case the current node is a sensor, we just perform sigmoid activation on the input variable from input array corresponding to the current node id. If the current node is not a sensor, we sum the values from all the incoming nodes after weighting them with the connection weight then do the sigmoid activation on the resulting sum.

After one activation of one node is completed, we call __syncthreads() in the kernel to wait for all other nodes on the layer to finish computing. After synchronization, we increase the current layer variable to denote that we have progressed one layer of the neural network and increase start id by the number of nodes which were present in the completed layer. We compare current layer variable against total number of layers of the neural network and exit out of the loop if current layer is greater then the total layers, which signifies that the network has completed activation. After completing activation, we make sure to call cudaDeviceSynchronize() function before we read in the answers from the host to give the host enough time to copy the results from the device memory to host memory. Then, we read in the values from output array which has the final activations of output nodes in the network.

D. Experimental setup

We used the CUBIX machine in our department for running all our experiments, which had the following configuration:

- Two 6 core Intel Xeon CPUs (E5-2620 0 @ 2.00GHz)
- CPU caches of L1: 32K, L2: 256K, L3: 15360K
- 64 GB of RAM
- 8 GTX 1080 with 8GB DRAM each

For all the results that follow, experiments were run 10 times and averaged for GPU activation timings and run 5 times and averaged for sequential activation timings.

Algorithm 3 CUDA kernel for calculating activation for ASNNs

```
1: ▷ Integer: TL = Total layers in a network
2: ▷ Integer[]: NNL = Number of nodes in layers
3: ▷ CudaNode[]: n = Array of all the nodes
4: ▷ Float[]: in = Input values for network
5: ▷ Float[]: op = Array for output values
6: function CUDA_ACTIVATION(TL, NNL, n, in, op)
7:     Integer: cl ← 0                      ▷ Current Layer
8:     Integer: sid ← 0                     ▷ Start id
9:     Integer: id ← threadIdx.x
10:    while cl < TL and id < NNL_cl do
11:        CudaNode: cn ← n_{sid+id}
12:        if cn is a sensor then
13:            op_{cn.id} ← call sigmoid(in_{cn.id})
14:        else
15:            Float: sum ← 0
16:            for i from 0 → cn.numInNodes - 1 do
17:                sum += cn.inWeights_i * op_{cn.inNodes[i]}
18:            end for
19:            op_{cn.id} ← call sigmoid(sum)
20:        end if
21:        call __syncthreads()
22:        sid ← sid + NNL_cl
23:        cl ← cl + 1
24:    end while
25: end function
```

IV. Results and discussion

A. Sequential results

From Figure 4 we can see that the execution of networks take linear amount of time with respect to the number of connections in the network. Increase in number of layers also correspond with the increase in execution time. Thus, with large number of connections and many layers, the execution time drastically increases. It is also possible for a network with same number of connection to have different execution time based on number of layers, as even with same number of connections, network with deeper layers take longer to execute.

B. Single GPU result

From Figure 5 we see that there is a general upward trend for the execution time with respect to the number of connections. But it should be noted that the slope is very flat compared to the sequential execution graph. The minimum is at 1ms and the maximum at 2.5ms.

C. Comparison and speedup

From Figure 4 we notice that compared to sequential execution, the GPU execution time lies flat and skims the x-axis. We can also compare the log of execution time from

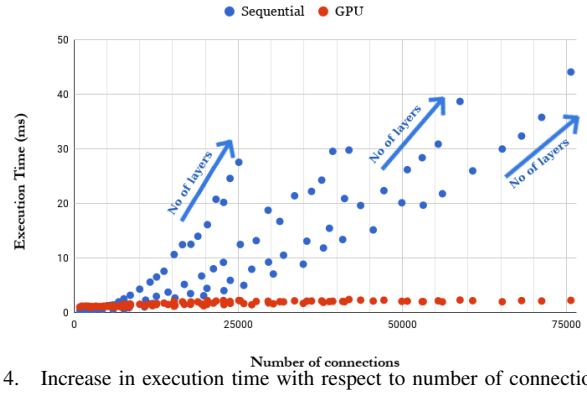

Fig. 4. Increase in execution time with respect to number of connections in a network for Sequential(blue) and GPU(red) approaches

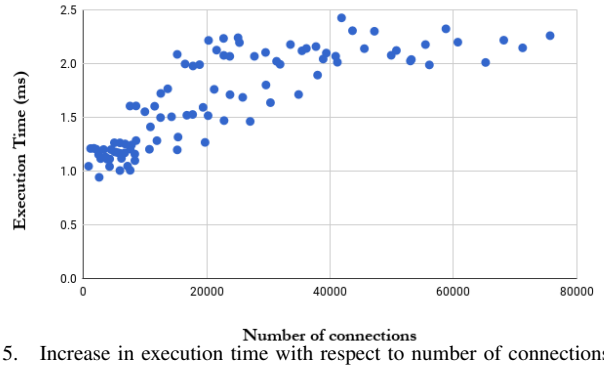

Fig. 5. Increase in execution time with respect to number of connections in a network using GPU implementation

Figure 6 where we see that the sequential execution time is very low for smaller networks but grows log linear with the number of connections. The log graph of execution time for sequential method has a steep slope at start, and still has positive slope after 30000 connections, while the GPU execution curve is flat with the x axis after 30000 connections. Thus, we can be certain that the GPU approach will scale well with increase in number of connections.

For speedup, we can see from the Figure 7 that initially, the speedup factor of sequential timings compared with GPU timings are lower than one which means that the GPU approach is slower than the sequential approach for very small

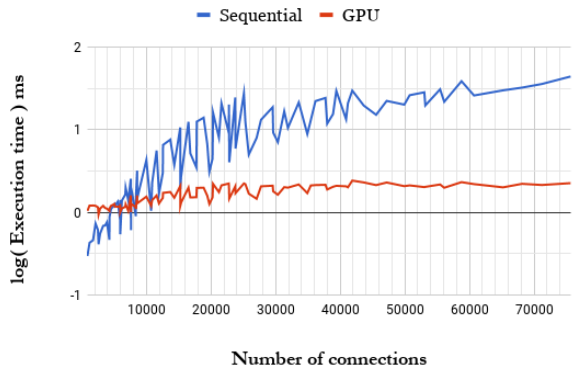

Fig. 6. Log of execution time with respect to number of connections in network, showing the comparison of execution times for Sequential and GPU approaches

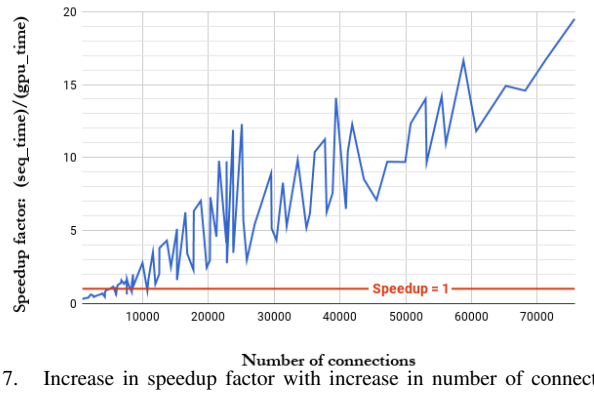

Fig. 7. Increase in speedup factor with increase in number of connections in a network showing the linear increase in speedup. Red line shows where speedup crosses the factor of 1

networks. This is predictable as overhead of copying to and from the device negates any speedup from the computation happening at the device. Till 8000 connections, speedup from GPU is similar to the the sequential implementation, but after 8000 connections, speedup shows linear increase with number of connections. The jagged structure of the line is due to the variation in number of layers possible for a same connection count i.e. low depth networks can be computed quickly compared to deeper networks. From the graph, we can see that the speedup factor is fairly high for low depth networks too and is linearly increasing with increase in number of connections. This results signify that we will have more gains the larger our network gets. If we take a network with 70,000 connections, we can get up to 15 times speedup, which will have huge gains given that the networks are evaluated for thousands of iterations for any given problem.

V. Conclusion and Future Work

Our research focused on finding effective ways of accelerating arbitrary structured neural networks. We were able to show that: by pre-processing the network to segment it into dependent layers and then using CUDA threads to execute all the nodes in the same layer at the same time, we can get speedup that increases with the size of the connections in the neural network. From our experimentation, we have shown linear speedup increase for our GPU implementation compared to our sequential implementation.

We can further improve on this work by extending the approach to incorporate grid wide thread locking to synchronize threads in a grid group which will significantly increase the number of nodes which can be processed simultaneously. The natural extension of this work is, to find ways to perform network segmentation in GPU itself; which will also have significant impact on the overall efficiency of current approach. Our approach of using GPUs to accelerate arbitrary neural networks can also be used for other domains of research, as networks found in nature generally have non uniform structure, so our research can be incorporated for their study and simulation. One prominent example is of biological brains which have arbitrary structure with huge number of nodes and connections. We could simulate and study such large networks if we can extend the processing capacity by coordinating multiple GPUs to compute a single network.

VI. Acknowledgments

This material is based in part upon work supported by the National Science Foundation under grant number IIA-1301726. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[2] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*, Elsevier, 1992, pp. 65–93.

[3] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, 2004, pp. 133–137.

[4] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, 1990, pp. 598–605.

[5] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in neural information processing systems*, 1993, pp. 164–171.

[6] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[7] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial life*, vol. 15, no. 2, pp. 185–212, 2009.

[8] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: From architectures to learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, 2008.

[9] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, *et al.*, "Evolving deep neural networks," *arXiv preprint arXiv:1703.00548*, 2017.

[10] F. Ino, J. Gomita, Y. Kawasaki, and K. Hagihara, "A gpgpu approach for accelerating 2-d/3-d rigid registration of medical images," in *International Symposium on Parallel and Distributed Processing and Applications*, Springer, 2006, pp. 939–950.

[11] D. Kirk *et al.*, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[12] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[13] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, IEEE, 2010, pp. 317–324.

[14] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning.," in *OSDI*, vol. 16, 2016, pp. 265–283.

[16] D. Göddeke, *Gpgpu-basic math tutorial*. Univ. Dortmund, Fachbereich Mathematik, 2005.

[17] NVIDIA. (2018). "Gpu-accelerated libraries for computing," [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries.

[18] D. Steinkraus, I. Buck, and P. Simard, "Using gpus for machine learning algorithms," in *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*, IEEE, 2005, pp. 1115–1120.

[19] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[20] Z. Luo, H. Liu, and X. Wu, "Artificial neural network computation on graphic process unit," in *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, IEEE, vol. 1, 2005, pp. 622–626.

[21] D. Scherer, H. Schulz, and S. Behnke, "Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors," in *International conference on Artificial neural networks*, Springer, 2010, pp. 82–91.

[22] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[23] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[24] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[25] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Neural networks*, vol. 22, no. 5-6, pp. 791–800, 2009.

[26] C.-F. Juang, T.-C. Chen, and W.-Y. Cheng, "Speedup of implementing fuzzy neural networks with high-dimensional inputs through parallel processing on graphic processing units," *IEEE Transactions on Fuzzy Systems*, vol. 19, no. 4, pp. 717–728, 2011.

[27] X. Chen, Y. Wang, X. Liu, M. J. Gales, and P. C. Woodland, "Efficient gpu-based training of recurrent neural network language models using spliced sentence bunch," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[28] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th design automation conference*, ACM, 2010, pp. 52–55.

[29] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International conference on high-performance computing*, Springer, 2007, pp. 197–208.