Artificial Neural Network Simulation on CUDA

John Pendlebury, Huanhuan Xiong, Ray Walshe
CloudCore Research Group
School of Computing, Dublin City University
Dublin, Ireland
jpendlebury@computing.dcu.ie

***Abstract*—The advent of low cost GPU hardware and user friendly parallel programming APIs, such as NVIDIA CUDA means that affordable, programmable, high-performance computing environments for simulation are now attainable for development of scientific simulations. In this paper the authors present the MineHunter program, a parallel simulation of neural networks on NVIDIA CUDA. The simulation consists of 128 mine hunters in a mine field of 8192 mines, running on an Intel QuadCore i5-2500 3.3GHz 2 x Nvidia GeForce GTX 480. The results presented demonstrate that CUDA improves performance by up to 80% compared with the equivalent CPU implementation.**

***Keywords*-GPU;CUDA;Simulation;Neural-Networks;**

I. Introduction

It was not at all long ago that developers wishing to take advantage of high-performance computing required access hours on dedicated super computer architectures for high performance computing. With the advent of easy to use, low learning curve GPU APIs such as CUDA (Compute Unified Device Architecture) from NVIDIA the age of high performance computing for the masses has arrived.

Neural Networks are a good example of where parallel computing can be utilised very effectively, as they are composed of many independent interconnected processing units and are inherently parallel in nature.

Moreover CUDA has some characertistics that make it very suitable for implementing neural network simulations. CUDA is capable of running thousands of inexpensive threads concurrently. CUDA capable devices typically have thousands of dedicated specialised processing units such as ALUs (Arithmetic Logic Units) capable of rapid mathematical calculation. This allows the parallel simulation of potentially hundreds of thousands of artificial neurons within light weight threads on a GPU.

While there are many examples of complex problem solving with neural networks using CUDA [1] [2] [3] et al., there are few, if any, examples of simple neural network implementations suitable for entry-level discussion in this field. The goal of this paper is to present a GPU based solution for a relatively large scale simulation of simplistic neural network based agents, as a first step to creating large-scale, parallel neural network based multi agents systems.

The outline of this paper is as follows. Section II describes several examples of related work about neural network simulation in hardware platforms, including CPU and GPU environments. Section III presents several simulation programs and parallel algorithm implementations for neural network simulation on CUDA and CPU. The comparitive results of these algorithms are presented in Section IV and the conclusions detailed in Section V.

II. Related Work

Many previous researches focus on accelerating ANN (Artificial Neural Network) simulations by mapping them into paralleling computing, or on dedicated hardware architecture, including clusters, supercomputers or high-performance processors. Some of the earliest work used hyper-cubic parallel computers to model ANNs [4].

Niebur and Brettle (1993) [5] use a hypercube architecture to simulate biological neural networks on massively parallel supercomputers. They develop a simulator that is used to simulate the neural networks of 16,384 neurons coupled by about 1000 synapses per neuron, and estimate the performance for the simulator.

Pleaser et al. (2007) [6] develop an efficient parallel simulation of large-scale neuronal networks on the clusters of multiprocessor computers. The result demonstrates that hybrid approach to neural network simulation, which combines multi-threading and distributed computing, achieves an even better performance than a purely distributed simulation.

Jin et al. (2008) [7] build an efficient model of spiking neural networks on a scalable multiprocessor. The result shows that the system is capable of simulation large-scale neural networks at 1ms resolution efficiently, which provides some generic ideas for design of hardware platforms for computational neural networks.

Even though the performance and power efficiency of these dedicated hardware approaches (mainly CPU-clusters) is much superior to other techniques, the dedicated hardware approach suffers from limited programmability and high-cost [8]. GPUs provides a more powerful and cheaper computational platform for the acceleration of parallel computing for diverse applications.

Oh and Jung (2004) [9] utilize the parallelism of GPU by accumulating numerous input and weight values, and

converting multiple inner-product operations to one matrix operation. And in this way, they utilize the vertex and pixel shaders in a GPU to implement matrix operations. Finally, they get a 20-fold performance enhancement using an ATI RADEON 9700 PRO board compared to CPU-only processing.

Daniel et al. (2008) [1] believe that CUDA implementations can be well suited for neural networks applications. They build the implementation based on three computational kernels: matrix operations, random number generators and sigmoid functions. And they find the GPU implementation has a speed-up of 66-fold over an optimized C++ program running on a 2.83GHz Intel processor.

Guzhva et al. (2009) [2] implement standard back-propagation algorithm for training multiple perceptrons simultaneously on NVIDIA CUDA. They compare the GPU-based implementation to a highly optimized CPU-based computer program, and find that the former has up to 50x speed increase than the latter.

Nageswaran et al. (2009) [3] represent strategies for mapping of large-scale spiking neural networks simulation models on GPU. They find that the CUDA GPU implementation is up to 24 times faster than a CPU version.

III. Simulation

In order to generate the results for this research paper a simulation was created called MineHunters. The purpose of the MineHunters simulation is to emulate the activity of landmine hunting robots detecting landmines in a minefield. In the first part of our research we prototyped the simulation in Java.

A. JMineHunter

The first step in the creation of the simulation was the development of a simple Java application called JMineHunters. Fig. 1 shows a screen shot of the JMineHunter application. The application consisted of a small 2-dimensional grid ranging from -100 to +100 along the x-axis and the y-axis. A number of mines were randomly placed on this grid. The mines are represented in red in Fig. 1. Mine hunters were also randomly placed on the grid, represented in green in Fig. 1.

Each MineHunter was attributed with a simple artificial neural network consisting of only two neurons. Neural networks are very versatile and have been utilised in areas as diverse as face detection [10] and bankruptcy prediction [11]. As noted in [12] the workhorse of neural networks is the standard feedforward three-layer model. It is certainly true to say that the more complex the problem, the more complex the neural network required to solve it will be. That does not detract from the fact that simple neural networks can have useful applications.

Fig. 2 illustrates the configuration of the neural network encapsulated in each mine hunter. Each neural network has four inputs. As shown in Table I the inputs take the coordinates of the mine hunter and the mine.

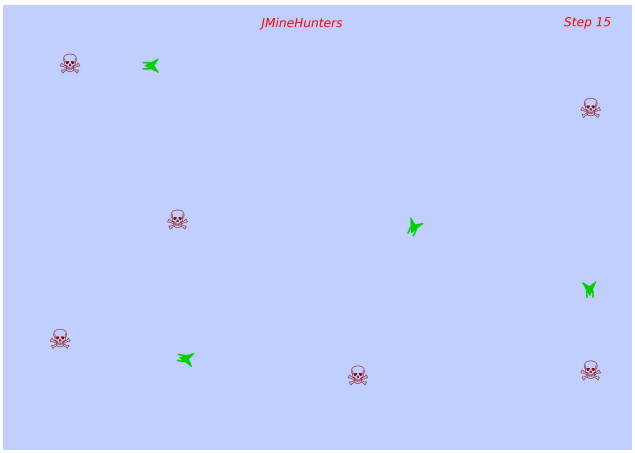

Figure 1. Screenshot of the prototype JMineHunter application.

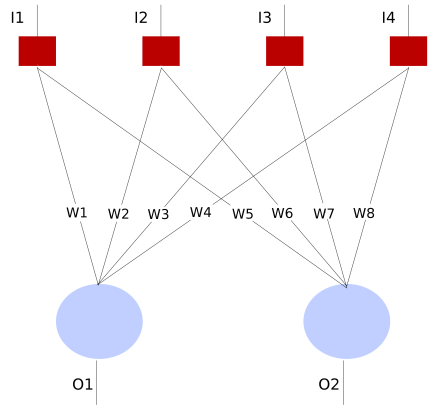

Figure 2. Configuration of neural network for each MineHunter.

Input	Value
I1	The x-coordinate of the mine hunter
I2	The y-coordinate of the mine hunter
I3	The x-coordinate of the mine
I4	The y-coordinate of the mine

Table I
Input values for each neural network.

Table II displays the weights applied to each input by each neuron. The input to the first neuron is:

$$X1 = W1 * I1 + W2 * I2 + W3 * I3 + W4 * I4 \quad (1)$$

The input to the second neuron is:

$$X2 = W5 * I1 + W6 * I2 + W7 * I3 + W8 * I4 \quad (2)$$

Obviously multiplications involving zero are redundant and are not performed.

Weight	Value
W1	-1
W2	0
W3	1
W4	0
W5	0
W6	-1
W7	0
W8	1

Table II
WEIGHT VALUES FOR EACH NEURAL NETWORK.

The output from each neuron is calculated as $O1 = f(X1)$ and $O2 = f(X2)$ respectively, where $f(x)$ is the activation function used. The activation function for each neuron is a modified sigmoid function of the form:

$$f(x) = 2 * (1/(1 + e^{-0.1*x}) - 0.5) \quad (3)$$

Fig. 3 shows the resulting output from this function. The limits for this 2-dimensional world are +/- 100.

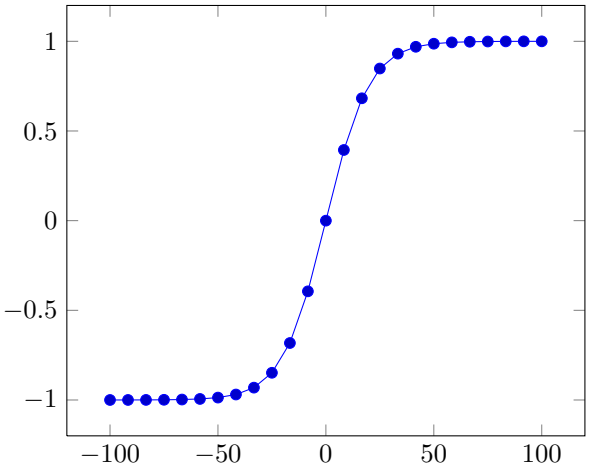

Figure 3. Output from the activation function used.

The consequence of this is that O1 and O2 can be used as the x and y components of a direction vector for the mine hunter. When continually added to a mine hunter's position this will result in the mine hunters's position intercepting that of the mine to which it s targeted.

Algorithm 1 outlines the basic operation of the JMineHunter application.

B. Porting JMineHunter to C

The next step in this project was to port the logic presented in Algorithm 1 to the C programming language. As the new C program did not require a graphical output this step was trivial. The only logical changes made were the number of mines, which were increased from 6 to 8192, and the number of mine hunters, which were increased from 4 to 128. The functions implemented in the C program would later be used as a base line comparison for the CUDA implementation.

```
Data: Uninitialised Map
Result: Solved Initialised Map
initialise random mine positions;
initialise random mine hunter positions;
for each mine hunter do
    get the closest mine to this mine hunter;
    assign the closest mine as the mine hunter's target;
end
while map not solved do
    for each mine hunter do
        update mine hunter's sensors;
        read output from mine hunters NN;
        update mine hunter's position;
        check if mine hunter has reached target;
    end
    if all targets reached then
        map is solved;
    end
end
```

Algorithm 1: Basic operation of JMineHunter.

C. Porting JMineHunter to CUDA

Unlike porting the original Java implementation to C, the implementation for CUDA required a paradigm shift in thinking. Initialising the mine and mine hunters with random positions was simple array initialisation and was not implemented in CUDA.

Assigning a target to each mine hunter: The C implementation for this task is described in Algorithm 2, which has complexity of $O(n^2)$. CUDA however, allows us to reduce this significantly (potentially to $O(1)$ complexity) using the Parallel Reduction Algorithm as described in [13].

```
for each mine hunter do
    make the first mine the target of the current mine
    hunter;
    for each mine do
        if the current mine is closer to the mine hunter
        than the target then
            make the current mine the target of the
            current mine hunter;
        end
    end
end
```

Algorithm 2: C implementation of assigning a mine target.

The implementation of the Parallel Reduction algorithm presented here works as follows. The problem decomposes into two sub-tasks. Sub-task one is calculating the distance

of each mine from each mine hunter. This is performed in one kernel call by assigning one thread to calculate the distance of one mine from one mine hunter. That is a total of $128 * 8192$ or $1,048,576$ threads running simultaneously. Each mine hunter has an associated array of distances, with each element storing the distance to each mine.

The second sub-task of the problem is to search for the smallest distance in each mine hunters array of distances. Finding a minimum value in a huge array of values such as this is a classic reduction problem [13]. Fig. 4 illustrates the reduction algorithm operating on a small array of 8 items.

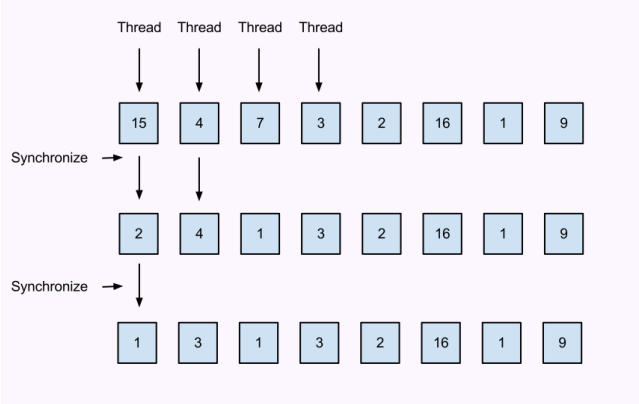

Figure 4. An illustration of reducing a small array.

Data: Initialised array of N values
Result: Minimum value in the array
create N/2 threads and assign each one to an element in the first half of the array;
size = N/2;
while $size > 0$ **do**
    run each thread;
    synchronize threads;
    size = size/2;
end

Algorithm 3: Using reduction to find the minimum value in an array of values.

Algorithm 3 depicts the basic reduction algorithm. When each thread is run it first checks to see if it is in the bottom half of the array. If it is in the bottom half of the array it compares the element in the array for which it is responsible for with the corresponding element in the latter half of the array. The smallest of the two elements will be stored in the element that the thread is responsible for. The variable "size" indicates what constitutes the bottom half of the array. As size is continually being divided by two the number of threads that are actually needed is also being divided by two. However, the same number of threads is run on every iteration of the loop. Some threads are wasted in this process. However, because CUDA compliant boards were designed to create and run threads with little overhead, this is not an issue for concern.

Updating the mine hunters sensors: Updating the mine hunter sensors is simply a parallel array update. Our system has 128 mine hunters with 4 sensors each, giving a total of 512 threads. Each thread is responsible for copying one value into an array element. This seemingly trivial task produced some surprising results in Section IV.

Updating the position of the mine hunters: Updating the mine hunter positions is again simply a parallel array update. Our system has 128 mine hunters with 2 neurons each, giving a total of 256 threads. As illustrated in Fig. 5, each thread is responsible for calculating the output of a single neuron and copying that value into an array element. Again some unexpected results were produced as documented in Section IV.

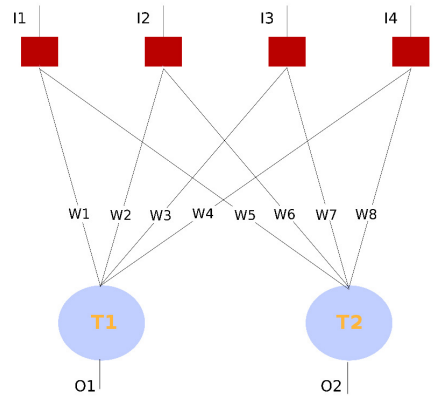

Figure 5. Each neuron in a neural network is updated by a single thread.

IV. Results

This section illustrates the results of our experiments with CUDA. Table III describes the series of functions that were run on the CUDA system. In CUDA terminology these are kernels i.e. functions run specifically on the CUDA device.

Function	Description
A1	Calculating the distance of each Mine from each MineHunter.
A2	Calculating the closest Mine from all MineHunters.
A3	Calculating sensor readings for all MineHunter.
A4	Updating the positions of all MineHunters.

Table III
FUNCTIONS RUN ON CUDA AND ON CPU.

The timing of each function was measured using the CUDA system. These timings are shown in red on the diagrams below.

In order to provide a measure of comparison, equivalent functions were run on the CPU. These CPU functions are shown in blue in the diagrams below. These functions are not equivalent to their CUDA counterparts in terms of complexity, as the parallel nature of the CUDA implementations demands that algorithmic complexity is naturally reduced.

The CPU implementations of these functions are single threaded, non-parallel functions. However, as one of the purposes of this paper was to explore how parallelism on CUDA could enhance performance compared to non-parallel code we feel that these functions provide an appropriate baseline for comparison.

A. Calculate the Distances of Mines from Mine Hunters

The first function that was measured was the process of calculating the distance of each mine from every mine hunter. As shown in Fig. 6 CUDA vastly outperforms its CPU counterpart. From Fig. 6 it can also be seen that the CUDA implementation runs in just 5% of the time used by its CPU counterpart.

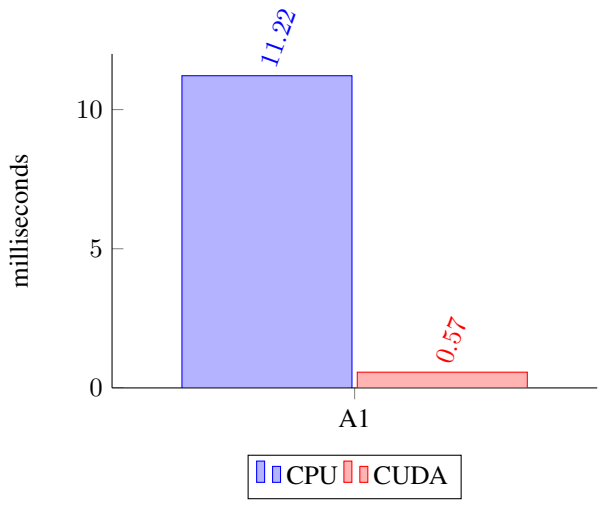

Figure 6. Timings for calculating the distance of each Mine from each Mine Hunter.

B. Calculating the closest Mine to each Mine Hunter

Having calculated the distance of each mine to each mine hunter it was then possible to run the second function, calculating the closest mine to each mine hunter.

As shown in Fig. 7 the speed improvement is not as drastic as the first result, but significant none-the-less, with an improvement of over 65% for the CUDA implementation.

A more impressive illustration of the timings for calculating the closest mine to each mine hunter is presented in Fig. 8, which shows the expected complexity of O(Log n) decay [14].

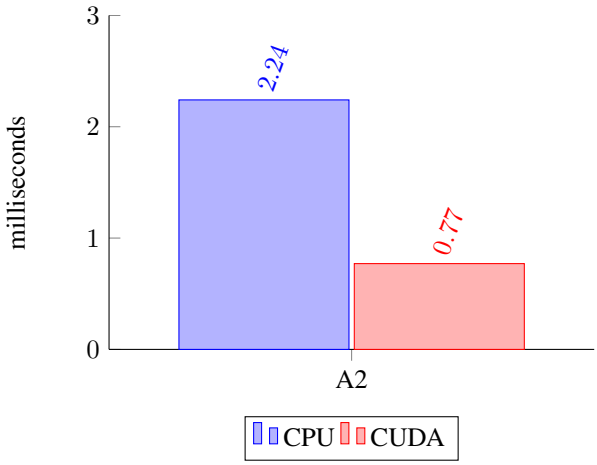

Figure 7. Timings for calculating the closest Mine to each Mine Hunter.

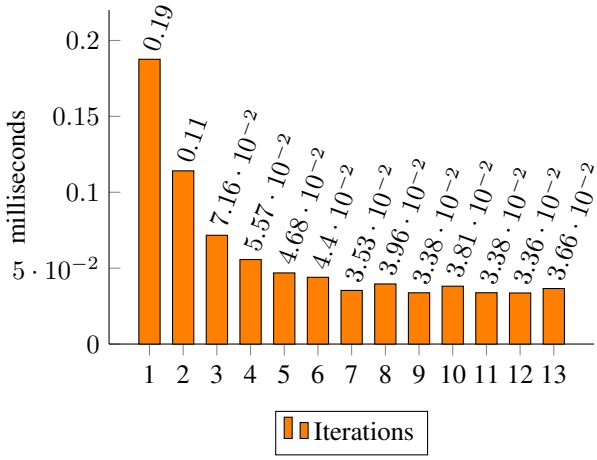

Figure 8. Iteration timings for calculating the closest Mine to each Mine Hunter

C. Calculating sensor readings of all Mine Hunters

The main update loop of our experimental program consisted of two operations, updating the sensor readings of each mine hunter to be used as input to the mine hunter's neural network and updating the positions of the mine hunters by taking the output from their neural network and applying it as the direction in which to move the mine hunter. The results of the former operation are shown in Fig. 9.

These results are disappointing. As can be seen from Fig. 9, the CPU implementation outperforms the CUDA implementation by over 70%. This is likely to be the result of the small data sets involved coupled with an inefficient block/thread configuration resulting in high cache latency. However, more investigation is required to confirm this.

D. Updating the positions of all Mine Hunters.

As with the previous results these results are disappointing. The CPU implementation outperforms the CUDA

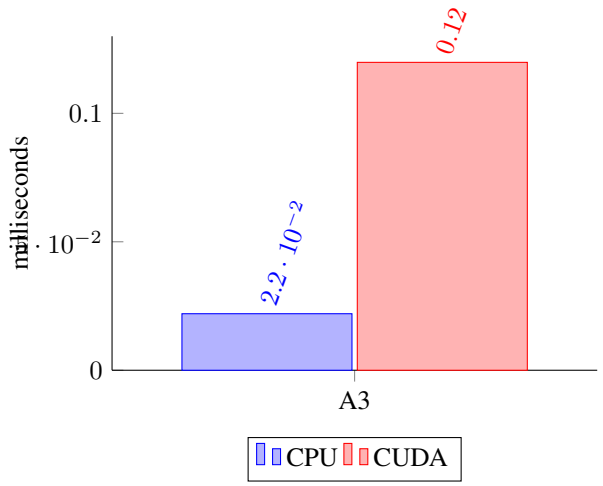

Figure 9. Timings for calculating sensor readings of all Mine Hunters.

implementation by over 70%.

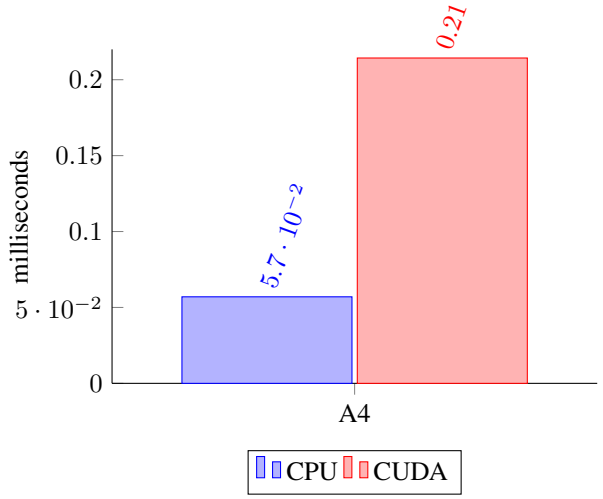

Figure 10. Timings for updating the positions of all MineHunters.

V. Conclusions

The experimental results presented in this paper show that CUDA is an excellent choice for rapid conversion of programs to a parallel architecture. It has a very low learning curve as experienced in implementing the experimental program for this paper. However, the results presented in Fig. 9 and Fig. 10 clearly demonstrate that utilising CUDA without sufficient knowledge of the underlying principles on which it is based can have unexpected results.

Our future work will investigate the reasons why CUDA performed poorly in relation to the CPU solution in the results presented for updating the positions of all mine hunters and calculating sensor readings of all mine hunters.

References

[1] D. Ly, V. Paprotski, and D. Yen, "Neural networks on gpus: Restricted boltzmann machines," Technical Report, Department of Electrical and Computer Engineering, University of Toronto, Tech. Rep., 2008.

[2] A. Guzhva, S. Dolenko, and I. Persiantsev, "Multifold acceleration of neural network computations using gpu," *Artificial Neural Networks–ICANN 2009*, pp. 373–380, 2009.

[3] J. Nageswaran, N. Dutt, J. Krichmar, A. Nicolau, and A. 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.

[4] A. Jahnke, T. Schönauer, U. Roth, K. Mohraz, and H. Klar, "Simulation of spiking neural networks on different hardware platforms," *Artificial Neural NetworksICANN'97*, pp. 1187–1192, 1997.

[5] E. Niebur and D. Brettle, "Efficient simulation of biological neural networks on massively parallel supercomputers with hypercube architecture," *Advances in Neural Information Processing Systems*, pp. 904–904, 1994.

[6] H. Plesser, J. Eppler, A. Morrison, M. Diesmann, and M. Gewaltig, "Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers," *Euro-Par 2007 parallel processing*, pp. 672–681, 2007.

[7] X. Jin, S. Furber, and J. Woods, "Efficient modelling of spiking neural networks on a scalable chip multiprocessor," in *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. IEEE, 2008, pp. 2812–2819.

[8] J. Nageswaran, N. Dutt, J. Krichmar, A. Nicolau, and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using cuda graphics processors," in *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*. IEEE, 2009, pp. 2145–2152.

[9] K. Oh and K. Jung, "Gpu implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, 2004.

[10] H. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 1, pp. 23–38, 1998.

[11] M. Odom and R. Sharda, "A neural network model for bankruptcy prediction," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE, 1990, pp. 163–168.

[12] T. Masters, *Practical neural network recipes in C++*. Morgan Kaufmann, 1993.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[14] M. Harris, "Optimizing cuda," *SC07: High Performance Computing With CUDA*, 2007.