Optimizing a Routing Algorithm Based on Hopfield Neural Networks for Graphic Processing Units

Carmelo J. A. Bastos-Filho, Marcos A. C. Oliveira Junior, Dennis R. C. Silva, and Robson A. Santana
Polytechnic School of Pernambuco,
University of Pernambuco,
Recife 50720-001, Pernambuco, Brazil
e-mail: carmelofilho@ieee.org

***Abstract*—Although some interesting routing algorithms based on HNN were already proposed, they are slower when compared to other routing algorithms. Since HNN are inherently parallel, they are suitable for parallel implementations, such as Graphic Processing Units (GPU). In this paper we propose a fast routing algorithm based on Hopfield Neural Networks (HNN) for GPU, considering some implementation issues. We analyzed the memory bottlenecks, the complexity of the HNN and how the kernel functions should be implemented. We performed simulations for five different variations of the routing algorithm for two communication network topologies. We achieved speed-ups up to 55 when compared to the simplest version implemented in GPU and up to 40 when compared to the CPU version. These new results suggest that it is possible to use the HNN for routing in real networks.**

I. Introduction

Routing algorithms have been intensively discussed in the scientific community, mainly because the routing process impacts drastically on the performance of communication networks. An ideal routing algorithm comprises finding the best path between the source and the destination nodes, enabling high quality transmission, avoiding penalties caused by physical layer impairments and reserving resources for future requests. There are different ways to determine a route. Some algorithms determine the routes based on the shortest path (SP) [1], the minor delay [2], the higher Signal to Noise Ratio [3], the better load distribution [4], among others.

A flexible routing algorithm has to calculate the path in real time considering the current available resources and the physical layer impairments. Some Computational Intelligence techniques have been considered in order to provide this ability. Among them, we can cite: Artificial Neural Networks (ANN) [2] [4], Ant Colony Optimization [5], Genetic Algorithms [6]. ANN are suitable for solving the routing problem problem since they present high computational speed and distributed processing capability [2].

Some ANN approaches were already considered for solving the routing problem. Hopfield and Tank described an ANN with feedback connections and showed that this approach can solve the Traveling Salesman Problem. Nowadays, this ANN configuration is known as *Hopfield neural networks* (HNN). Rauch and Winarske [7] were the first to apply HNN to find the shortest path nodes in a communication network. After that, Ali and Kamoun [2] proposed a HNN variation in which the weight matrix just carries convergence information. The link cost and the network topology information are inserted in external bias inputs. Bastos-Filho *et al.* [8] proposed to use a discrete and simple finite difference equation to update the inputs of the neurons in order to simplify and accelerate the convergence of the HNN. Schuler *et al.* [9] optimized the difference equation proposed by Bastos-Filho *et al.* by using a computational intelligence technique calle Particle Swarm Optimization. Recently, Kojic *et al.* [10] and Bastos-Filho *et al.* [11] proposed routing algorithms for optical networks based on HNN.

The average time to find the path between the source and the destination node (T_{path}) is a very important attribute of a routing algorithm. The Dijkstra algorithm can determine a shortest path in some dozens μs. Although Bastos-Filho *et. al.* [11] have optimized the HNN model proposed by Bastos-Filho *et al.* [8] to minimize T_{path}, some dozens ms are necessary to determine the path. It means that the current HNN based routing algorithms can obtain good network performance, but T_{path} is still high when compared to other algorithms.

In the recent years, the use of Graphic Processing Units (GPUs) have been proposed for many scientific applications. The GPU parallel floating point processing capacity allows one to obtain high speed-ups. Nevertheless, there are some aspects that should be considered to adapt an application to run on these platforms, such as memory allocation and communication between blocks. Furthermore, hardware limitations of the GPUs limit the use for general purposes. Naive implementations neglecting theses issues may lead to a poor performance.

One of the major benefits of neural networks is the parallel processing capacity. In a HNN, each neuron can process individually and exchange information by a fully connected network. Therefore, HNN are naturally suitable for a parallel implementation and the execution time may be drastically reduced if the HNN are implemented on a parallel platform. In this paper we present a parallel HNN-based routing algorithm for GPUs. We discuss some important issues regarding the implementation to improve the performance.

This paper is organized as follows: in the next section we briefly review the current best HNN model to solve the routing problem. In Section III, we introduce some basic concepts of the NVIDIA CUDA architecture and GPU computing. We present our contribution and the simulation results in Section IV and V, respectively. In the last section, we present our conclusions and suggest some future works.

II. Hopfield Neural Networks For Routing in Communications Networks

The HNN block diagram is depicted in Figure 1. The neurons are the processing elements. Every output of each neuron is connected to the input of all the other neurons via synaptic weights. Each link in the communication network between two adjacent nodes is associated to one neuron [2]. For example, a link from node x to node i refers to the neuron xi. The output of a neuron V_{xi} depends on the input U_{xi} and is evaluated by (1). The input of each neuron corresponds to the sum of all the outputs of the other neurons yj weighted by a synaptic weights (matrix $T_{xi,yj}$) plus an external bias I_{xi}. The parameter λ determines the computation time to convergence and the correctness of the algorithm. As higher as the parameter λ is, the logistic function tends to a step function.

$$V_{xi} = \frac{1}{1+e^{-\lambda U_{xi}}} \qquad \forall(x,i) \in \bar{N}X\bar{N}/x \neq i \tag{1}$$

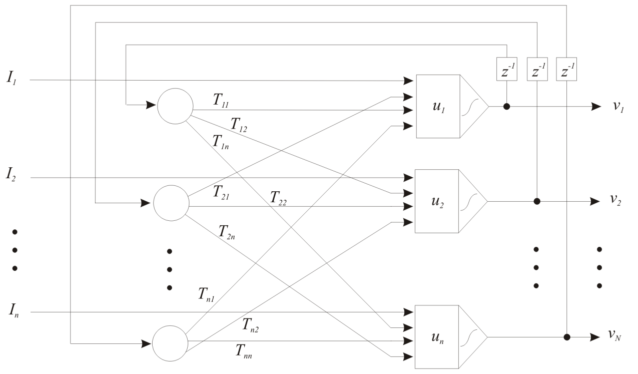

Fig. 1. Hopfield Neural Network Configuration.

If every link in the network has a nonnegative cost associated C_{ij}, the goal of the HNN is to find the path that minimizes the cost from a source node *s* to a destination node *d*. Thus, the HNN should indicate a directed path as an ordered sequence of nodes connecting *s* to *d*. The path that provides minimum cost is defined as L_{sd}. In most cases, the cost matrix is symmetric ($C_{xi} = C_{ix}$). The elements C_{ii} are nulls because on node cannot be connected to itself.

The matrix ρ_{xi} defines if the link *xi* exists in the communication network topology. ρ_{xi} is defined as:

$$\rho_{xi} = \begin{cases} 1, & \text{if the link } xi \text{ does not exist;} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

The HNN converges when the variation of every output values between consecutive iterations (ΔV_{xi}) are below a threshold, $i.e.$, $\Delta V_{th} < 10E^{-5}$. After that, a new matrix (Y_{xi}) is evaluated. If the output has a value greater than 0.5, Y_{xi} is adjusted to "1", otherwise Y_{xi} is adjusted to "0". If $Y_{xi} = 1$, the link $xi \in L_{sd}$, else the link does not belong to the shortest path L_{sd}.

Beside this, each neuron is externally excited by input bias (I_{xi}). I_{xi} are used to include the link costs and the network topology information as shown below:

$$\begin{aligned} I_{xi} \; = \; & -\frac{\mu_1}{2}C_{xi}(1-\delta_{xd}\delta_{is}) - \frac{\mu_2}{2}\rho_{xi}(1-\delta_{xd}\delta_{is}) \\ & -\frac{\mu_4}{2} + \frac{\mu_5}{2}\delta_{xd}\delta_{is}, \, \forall\,(x \neq i)\,, \end{aligned} \tag{3}$$

where δ is the Kronecker function and $\mu1$, $\mu2$, $\mu4$ and $\mu5$ are HNN parameters. μ_1 minimizes the total cost, μ_2 prevents nonexistent links from being included in the chosen path, μ_4 avoids the convergence to a unstable state and μ_5 is introduced to ensure that the source and the destination nodes belong to the solution.

The HNN synaptic matrix $T_{xi,yj}$ is described as [2].

$$T_{xi,yj} \; = \; \mu_4\delta_{xy}\delta_{ij} - \mu_3\delta_{xy} - \mu_3\delta_{ij} + \mu_3\delta_{jx} + \mu_3\delta_{iy}, \quad \forall\,(x \neq i)\,, \forall\,(y \neq j)\,. \tag{4}$$

where $\mu3$ is a HNN parameter and is used to guarantee the convergence.

Therefore, if the system is stable in Liapunov sense, then iterations lead to smaller output changes. As a consequence, after some iterations the HNN converges. Schuler *et al.* used the difference and finite equation described in equation (5) to update the neurons. By using this equation, Schuler *et al.* obtained a lower number of iterations to reach the convergence and a lower number of errors finding the paths.

$$\begin{aligned} U_{xi}[n+1] \; = \; & U_{xi}[n] - AU_{xi}[n-1] + \\ & B\sum_{y=1}^{n}\sum_{\substack{j=1 \\ j\neq y}}^{n} T_{xi,yj}V_{yj}[n] + CI_{xi}. \end{aligned} \tag{5}$$

The pseudocode for the HNN-based routing algorithm is shown in Algorithm 1. U_{xi} is initialized with low and random values ($-0,0002 \leq U_{xi} \leq 0,0002$) in order to speed-up the convergence faster.

III. GPU Computing and CUDA Architecture

GPUs were traditionally designed for image and graphic processing on computers. The use of GPU for general purpose computing was inspired by its ability to process thousands of threads simultaneously. GPUs became popular in recent years through a NVIDIA technology called CUDA (*Compute Unified Device Architecture*). CUDA enables programmers to develop softwares to solve complex computational problems by automatically distributing these threads to many-core simple processors.

The CUDA GPUs have a single-instruction multiple-thread architecture (SIMT) [12], where the same instruction set is

Algorithm 1 Pseudocode of the routing algorithm using Hopfield neural networks.

```
 1: Receive parameters (including $C_{xi}$);
 2: Determine $T_{xi,yj}$;
 3: Evaluate $\rho_{xi}$;
 4: Receive source and destination;
 5: Evaluate $I_{xi}$;
 6: Insert initial noise in $U_{xi}$;
 7: Evaluate $V_{xi}$ (Threshold of $U_{xi}$);
 8: while $\Delta V_{xi} < \Delta V_{th}$ do
 9:     Store $U_{xi}$ and $V_{xi}$;
10:     Upgrade the neurons inputs using (5);
11:     Upgrade the neurons outputs using (1);
12:     Evaluate $\Delta V_{xi}$;
13: end while
14: Determine $Y_{xi}$ (binarization of $V_{xi}$);
15: Get path from $Y_{xi}$;
```

executed on different processors at the same time. This architecture presents less overhead in parallel computing, which is suitable to intensive and repetitive computation.

The CUDA parallel programming model allows one to split the main problem in many sub-problems that can be executed independently in parallel. Each one can be decomposed in many other modules that may have their operations performed cooperatively in parallel. Actually, each sub-problem is equivalent to a block of threads and each thread is a module. The function that is performed to solve a sub-problem is called kernel function. When a kernel function is invoked, it will run on each thread in parallel within the corresponding block.

Each thread executing a kernel function is identified by its thread identifier. One can access it within the kernel through built-in variables provided by the CUDA API [13].

The main bottleneck in the CUDA architecture is the data transferring between the host (CPU) and the device (GPU). This type of operation harms the performance.

Threads within a block can cooperate by sharing data through a dedicated memory, but they need to synchronize the memory access in order to avoid data dependence errors. A barrier placed on the code allows a thread to wait for the other cooperative threads. They guarantee the correctness of the algorithm running on the GPU, but influence on the performance.

Each thread has a private local memory and each block of threads has a shared memory that can be accessed by all threads inside the block. Moreover, all threads can access the same global memory. These memories follow a memory hierarchy: the fastest one is the local memory and the slowest is the global memory; the smallest one is the local memory and the largest is the global memory.

Random number generators are widely used in computational intelligence techniques. They are naturally sequential, since they are based on states. They must be carefully chosen. GPU-based random numbers generators are discussed by [14] and [15]. An approach CPU-free for generating random numbers on demand is presented by [16].

IV. GPU-BASED HNN MODEL

The HNN-based routing algorithm has an embarrassingly parallel behavior since all operations can be executed individually because each one has an independent input. The first step to the adapt the algorithm to a parallel platform is to identify the bottlenecks. The main bottleneck is the update process of the neurons described from the 9^{th} to the 12^{th} line of the algorithm 1. This process only ends when a predefined threshold is reached. In the following subsections, we present how we adapted the algorithm to the parallel platform. First, we show how we call the GPU functions from the host and then we describe the GPU functions themselves.

A. The Host Code

The host code has the duty to call all the kernel functions. It is the slowest part of the implementation, since it does not run in the GPU and can not be implemented in parallel. Thus, one should avoid intensive operations at this point and move as much operations as possible to the kernel functions.

In our first implementation, we developed the naive model described in the Algorithm 2. In each kernel function call (shown in italic), the parameters inside the operands <<>> describe how many threads will run inside a block.

Algorithm 2 Pseudocode of the first version of the Host Code for the HNN-routing algorithm for GPU.

```
1: set-weights <<nodes*nodes>>;
2: initialize <<nodes*nodes>>;
3: converged = false;
4: while !convergedHost do
5:     iteration <<nodes*nodes>>;
6:     copy-from-device (converged);
7:     iterations;
8: end while
9: Determine $Y_{xi}$ <<nodes*nodes>>;
```

Once the hardware limits the maximum number of threads running on a block, the evaluations of the inputs and the outputs of the neurons must be split in blocks. As consequence, the convergence test must be performed in the host. Obviously, this approach harm the performance, since the blocks can not be executed in parallel and the host and the device must establish a communication during the process.

Actually, there is no correct way to get rid of this issue. However, we may assume that there will not be needed more than an exactly number of threads running inside the block. Thus, there will be only one block of threads in the GPU. Assuming this, we could improve the implementation performance. This second version is shown in Algorithm 3.

The kernel function *iterations* runs in the GPU until the threshold is reached. By doing this, no communication between host and device is required during this process. Thus, it is probably the best approach for the host code.

Algorithm 3 Pseudocode of the second version of the Host Code for the HNN-routing algorithm for GPU.

```
1: initialize <<nodes*nodes>>;
2: iteration <<nodes*nodes>>;
3: Determine $Y_{xi}$ <<nodes*nodes>>;
```

B. The Device Code

The functions called by the host code to run in the GPU are known as kernel functions. These functions run in each thread in parallel. The main functions of the algorithm are the ones used to execute all the iterative process until the convergence. They are: the *iteration()* and *iterations()* functions. Both of them call a device function named *calculate-sum()* to evaluate the sum used to update the input of the neurons as shown in the Equation (5).

In the following subsections, the kernel functions and the device function *calculate-sum* are presented. Moreover, we present how we reduced the complexity of the update process and how we avoid some memory bottleneck issues.

1) The Kernel Functions: The initial version of the host code calls the *iteration* function described in the Algorithm 4. It uses a global variable called *converged* that it is copied by the host code from the device to know whether the threshold is reached.

Algorithm 4 The first version of the kernel function *iteration*.

```
1: sum = calculate-sum;
2: update-input-and-output(sum);
3: synchronization-barrier;
4: if threshold is reached then
5:     converged = false;
6: end if
```

The second version of the host code calls the function *iterations* and it is described in Algorithm 5. It runs all iterations of the algorithm until the threshold is reached. It also has a variable *converged* but it is in the shared memory space. This variable can be viewed and modified by all threads inside the block.

Algorithm 5 The second version of the kernel function *iteration*.

```
1: converged = false;
2: while (!converged) do
3:     sum = calculate-sum;
4:     update-input-and-output(sum);
5:     converged = true;
6:     synchronization-barrier;
7:     if threshold is reached then
8:         converged = false;
9:     end if
10: end while
```

2) Reduction of the Complexity of the Update Equation: The function *calculate-sum* is described in the Algorithm 6. The ty and tx variables are the thread identifiers. This description is the simplest way to implement the Equation (5). It has complexity $O(n^2)$ and as it is used by each neuron, the full algorithm has complexity $O(n^3)$.

Algorithm 6 The first version of the device function *calculate-sum*.

```
1: for k = 0 to number-of-nodes do
2:     for l = 0 to number-of-nodes do
3:         if k != l then
4:             sum += weights[ty][tx][k][l] * outputOld[k][l];
5:         end if
6:     end for
7: end for
```

Nevertheless, if the weights are analyzed as they are generated, the algorithm may be described as in Algorithm 7.

Algorithm 7 The optimized version of the device function *calculate-sum*.

```
1: for l = 0 to number-of-nodes do
2:     if l != ty then
3:         sum += weights[ty][tx][ty][l] * output[ty][l];
4:         if l != tx then
5:             sum += weights[ty][tx][l][tx] * output[l,tx];
6:             sum += weights[ty][tx][l][ty] * output[l][ty];
7:         end if
8:     end if
9:     if l != tx then
10:        sum += weights[ty][tx][tx][l] * output[tx][l];
11:    end if
12: end for
```

This approach is $O(n)$ and it might be implemented in a CPU sequential version as well.

3) The Memory Bottleneck: There is a memory hierarchy in the GPU. The closer to the GPU is the memory, the faster the memory access is. Therefore, closer memories should be used when it is possible.

In the *iteration* and the *iterations* kernel functions, the global memory is accessed constantly. Higher performance can be achieved if a faster shared memory is used. It is possible to store some information in the block shared memory, such as the input and output matrices. Unfortunately, the shared memory is limited and it is not possible to store the synaptic weights matrix, once it is a 4-dimensional matrix.

However, the weights may be generated on demand instead generating all of them for after-accessing through the global memory. The code is almost the same, the only difference is that a device function named *weight* is called for each time it is necessary to use a synaptic weight. One should notice that it generates more floating-point operations, but reduces the number of access to the memory.

Algorithm 8 The optimized version of the device function *calculate-sum* using a function to calculate the weight on demand.

```
for l = 0 to number-of-nodes do
    if l != ty then
        sum += weight(ty,tx,ty,l) * output[ty][l];
        if l != tx then
            sum += weight(ty,tx,l,tx) * output[l,tx];
            sum += weight(ty,tx,l,ty) * output[l][ty];
        end if
    end if
    if l != tx then
        sum += weight(ty,tx,tx,l) * output[tx][l];
    end if
end for
```

V. Simulation Setup and Results

The several versions detailed in the previous Section IV were implemented on the CUDA platform. The experiments were executed on an Intel Core Quad 2.40GHz computer with a NVIDIA GeForce 9800GTX+. We performed simulations for two different networks. First, with the famous National Science Foundation Network (NSFNET) with 14 nodes and then with a smaller one with 6 nodes. The networks are depicted respectively in Figure 2 and in Figure 3.

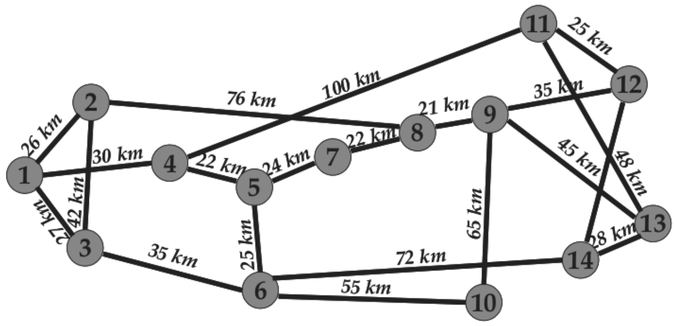

Fig. 2. Topology of the National Science Foundation Network (NSFNET) used in the simulations.

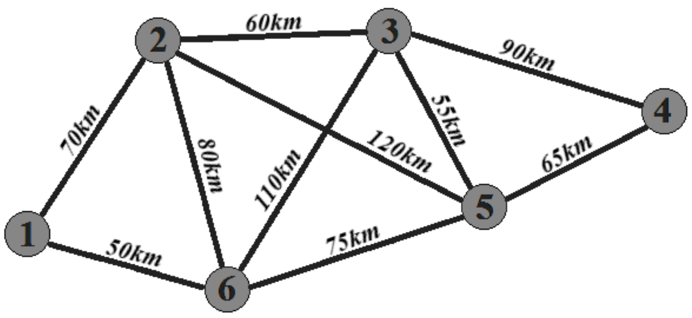

Fig. 3. Topology of a six-nodes network used in the simulations.

We used the following HNN parameters: $\mu_1 = 950.0$, $\mu_2 = 2500.0$, $\mu_3 = 2500.0$, $\mu_4 = 475.0$, $\mu_5 = 2500.0$, $A = 0.001$, $B = 0.001$ and $C = 0.001$.

We analyzed five different GPU versions for the HNN-based routing algorithm: *GPU A* – the initial model with the convergence test in the host code and without any use of the shared memory; *GPU B* – the *GPU A* version updated with a complexity reduction on the neurons update process; *GPU C* – a updated version where all iterations run in the GPU until the threshold is reached; *GPU D* – similar to *GPU C*, but using the shared memory to store the input and the output matrices; and *GPU E* – similar to *GPU D*, but the synaptic weights are generated on-demand.

Each version was called 10,000 times to find a route for randomly chosen pair of nodes for each network. The average execution times are shown in Table I.

TABLE I
Average Execution Time to Define the Route using the CPU Version and and all the GPU Versions.

Algorithm Version	**NSFNET (ms)**	**6-nodes Network (ms)**
CPU	73.461	2.598
GPU A	100.29	22.91
GPU B	98.2	12.69
GPU C	31.71	3.98
GPU D	3.92	0.51
GPU E	1.83	0.48

One can observe huge speed-ups obtained through the HNN implementation versions in both networks. The initial version (*GPU–A*) has an execution time slower than the CPU version, although it is parallel.

In the second version (*GPU–B*), after an analysis on the neurons update process, the first speed-up is reached. the achieved speed-up is higher for the smaller communication network. In the third version (*GPU–C*), all threads run inside an unique block, thus it is completely parallel. This alteration leads to a speed-up higher than three for both networks.

The fourth and higher speed-up was reached by using the shared memory inside the GPU (*GPU–D*). Faster memory access impacts drastically on the performance. Once it is not possible to place the matrix of the synaptic weights in the memory of the block, the synaptic weights are generated on demand in the *GPU–E* version. Surprisingly, it is faster to generate on demand the synaptic weights than to store them for after-accessing.

VI. Conclusions

Despite the algorithm for routing networks using Hopfield Neural Networks implemented on CPUs is slower than other approaches for routing, Hopfield Neural Networks have a parallel behavior that allows faster implementations on parallel platforms.

We presented in this paper a fast Hopfield Neural Networks based algorithm for routing in communications networks suitable for Graphic Processing Units. We also analyzed different versions to show that some aspects must be carefully considered for GPU architectures in order to avoid bottlenecks and even worse performances than sequential approaches. We achieved a speed-up of 55 for the NSFNet communication network topology when compared to a simple approach for GPU.

The high speed-ups shows that HNN are suitable to be implemented in GPUs, whereas the total time to find routes suggest that it is possible to use our approach in real scenarios.

Acknowledgment

The authors would like to thank FACEPE, CNPq, UPE and POLI (Escola Politécnica de Pernambuco).

References

[1] E. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[2] M. K. M. Ali and F. Kamoun, "Neural networks for shortest path computation and routing in computer networks," *Neural Networks, IEEE Transactions on*, vol. 4, no. 6, pp. 941–954, Nov 1993.

[3] H. Pereira, D. Chaves, C. J. A. Bastos-Filho, and J. Martins-Filho, "Osnr model to consider physical layer impairments in transparent optical networks," *Photonic Network Communications*, vol. 18(2), pp. 137–149, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11107-008-0178-2

[4] N. Kojic, I. Reljin, and B. Reljin, "Neural network for finding optimal path in packet-switched network," in *Neural Network Applications in Electrical Engineering, 2004. NEUREL 2004. 2004 7th Seminar on*, Sept. 2004, pp. 91–96.

[5] G. Di Caro and M. Dorigo, "Antnet: Distributed stigmergetic control for communications networks," *Journal of Artificial Intelligence Research*, vol. 9, pp. 317–365, 1998.

[6] S. H. Ngo, X. Jiang, and S. Horiguchi, "Adaptive routing and wavelength assignment using ant-based algorithm," in *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference on*, vol. 2, Nov. 2004, pp. 482–486 vol.2.

[7] H. E. Rauch and T. Winarske, "Neural networks for routing communication traffic," *Control Systems Magazine, IEEE*, vol. 8, no. 2, pp. 26–31, Apr 1988.

[8] C. J. A. Bastos-Filho, R. A. Santana, and A. L. I. Oliveira, "A novel approach for a routing algorithm based on a discrete time hopfield neural network," in *Foundations of Computational Intelligence, 2007. FOCI 2007. IEEE Symposium on*, April 2007, pp. 363–369.

[9] W. H. Schuler, C. J. A. Bastos-Filho, and A. L. I. Oliveira, "A novel hybrid training method for hopfield neural networks applied to routing in communications networks," *Int. J. Hybrid Intell. Syst.*, vol. 6, no. 1, pp. 27–39, 2009.

[10] N. S. Kojic, I. S. Reljin, and B. D. Reljin, "Routing in optical networks by using neural network," in *Neural Network Applications in Electrical Engineering, 2006. NEUREL 2006. 8th Seminar on*, Sept. 2006, pp. 65–68.

[11] C. J. A. Bastos-Filho, R. A. Santana, D. R. C. Silva, J. F. Martins-Filho, and D. A. R. Chaves, "Hopfield neural networks for routing in all-optical networks," in *Transparent Optical Networks, 2010. ICTON '10. 12th International Conference on*, 27 2010-july 1 2010.

[12] NVIDIA, *NVIDIA CUDA Programming Guide 3.1*. NVIDIA Corporation, 2010.

[13] ——, *CUDA C Best Practices Guide 3.2*. NVIDIA Corporation, 2010.

[14] H. Nguyen, *Gpu gems 3*. Addison-Wesley Professional, 2007.

[15] L. D. R. Thomas, Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2009, pp. 63–72.

[16] C. Bastos-Filho, M. Oliveira Junior, D. Nascimento, and A. Ramos, "Impact of the random number generator quality on particle swarm optimization algorithm running on graphic processor units," *Hybrid Intelligent Systems, 2010. HIS '10. Tenth International Conference on*, pp. 85–90, 2010.