HAL
archives-ouvertes.fr

Parallel Implementations of Hopfield Neural Networks On GPU

Li Liang

▶ To cite this version:

Li Liang. Parallel Implementations of Hopfield Neural Networks On GPU. Distributed, Parallel, and Cluster Computing [cs.DC]. 2011. dumas-00636458

HAL Id: dumas-00636458
https://dumas.ccsd.cnrs.fr/dumas-00636458

Submitted on 27 Oct 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parallel Implementations of Hopfield Neural Networks On GPU

Li LIANG

August 12, 2011

Abstract

In recent years the multi-cores and General-Purpose GPU (GPGPU) architectures have become general platforms for various of parallel applications, with lots of parallel algorithms being proposed for this interesting persperctive. In this report, we study and develop a particular kind of artificial neural network (ANN), in hopfield model, to solve some optimization problems, since it has a highly parallel nature. Section 1 introduces the context of the problem and the linked topics of this internship. In Section 2 we synthesize some previous work in this domain. In section 3 we study the parallel mechanism and propose the algorithm to realize this neural network. Section 4 interprets the method we use to map the neural network structure. Section 5 discusses the implementation details of simulations of hopfield networks on GPU. In section 6, we comment on the results and the performances compared with the CPU implementation.

Contents

1 Introduction **2**
- 1.1 Context . . . 2
- 1.2 Problem Description . . . 3
- 1.3 Objective . . . 4
- 1.4 Background . . . 4
- 1.5 The parallel algorithms design method for neural networks . . . 7

2 State of Art **7**

3 Basic implementation of neural network **10**
- 3.1 Network adapting to an optimization problem . . . 12
- 3.2 Typical k-out-of-N Rule Networks . . . 13
  - 3.2.1 The k-out-of-N rule in the simulations . . . 14
  - 3.2.2 Considering equality constraints . . . 14
- 3.3 Rules and Algorithms . . . 15
  - 3.3.1 Host/Device Data Preparation . . . 18

4 Parallelization of neural network **19**
- 4.1 Internal parallelization . . . 20
- 4.2 External parallelization . . . 20

5 Implementations **24**
5.1 Platform . . . 24
5.2 Simulation infrastructure . . . 25
5.2.1 Mapping Network Structure on GPU . . . 25
5.3 Random neuron selection on GPU . . . 26
5.4 Implementation of internal parallelization . . . 26
5.5 Implementation of the external parallelization . . . 28
5.5.1 Store Network Data in Blocks . . . 28
5.5.2 Global state calculation . . . 28

6 Result/Performance Analyses **30**
6.1 Energy Function . . . 30
6.2 Complexity Analysis . . . 30
6.3 Comsuming Time Comparision . . . 32

7 Conclusion **33**
7.1 Future work . . . 34

8 Acknowledgement **34**

1 Introduction

There are increasing realizations of parallel algorithms on multi-cores recently, especially for the GPUs, a programmable architecture that can be used in a data parallel computing. We create a particular GPU application : the artificial neural network in hopfiel model. In this report, we will explain the method we use according to the models and the parallel evolution process realized by GPUs.

1.1 Context

The Cairn team focus its reasearch on the computer architectures, especially the reconfigurable architectures such as reconfigurable system-on-chip (SoC), i.e. hardware systems whose configuration may change before or even during execution. CAIRN team intends to approach reconfigurable architectures from the invention of new reconfigurable platforms, the tool development, and the relative algorithms. This subject, the optimization algorithms using neural networks on parallel architectures, is one of the solutions relative to the real-time scheduling in reconfigurable architectures. Another solution is realized in the FPGA (Field Programmable Gate Array) architectures. We will focus on the first solution to specify the neural network algorithms and state the implementation details, as integrated in the heterogeneous SoC architecture scheduling optimation reasearch.

One of the research topic of the CAIRN team concerns the study of the management of reconfigurable system on chip. And this management of the execution of a set of tasks on the different execution resources is an import problem in the domain of multiprocessor sysytem on chip design. This type of system is based on a complex application to execute on a complex architecture. In general, this type of system embeds an operating system which makes global system management

easier. For this type of platform, the scheduling is more complex because it consists in defining not only the temporal but also the spatial execution of each task of the application. Indeed, in a reconfigurable system, the tasks can be dynamically and partially configured on the reconfigurable part. Each task is defined by its geographical position in the reconfigurable part, and the scheduling must ensure that the task are not spatially overlapped.

On important service that the operating system must ensure is the scheduling of the tasks on the different execution resources. This scheduling must be done under constraint and due to the unpredictable environment events, an online scheduling is required. This scheduling of a set of tasks within a large number of resources is a complex problem, and can be view as an optimization problem. To solve this type of problem, an efficient algorithm needs to be implemented in the OS and this algorithm needs to produce solution on line, i.e. during the execution of the applications.

This problem is addressed by a PhD student who has defined a spatial temporal scheduling and currently on the hardware implementation of his proposal. His proposal is based on neural network structure to solve the spatial temporal scheduling optimization problem. The use of neural network in this context is interesting in particular because this structure offers interesting performance of convergence towards solutions.

We will follow this direction to solve optimizations of management problems of reconfigurable systems by artificial neural networks, and thus try to explore some methods and to evaluate the performance of the implementations.

1.2 Problem Description

The model retained for the scheduling problem is the hopfield neural network. This type of neural network has been demonstrated as an interesting solution to solve optimization problem. To converge towards valid solutions, the convergence of the hopfield neural network must be asynchronous and sequential. This means that the neurons are evaluated one by one. and each neuron evolves asynchronously, i.e. no global synchronous clock.

A special kind of neural network based model is proposed for the design of heterogeneous multiprocessor architectures scheduler, taking the platform heterogeneity into account using a neural network model with inhibitor neurons which differs from hopfield model. However, we want to check the classic hopfield model in GPU architecture since this kind of neural network is naturally parallel.

With the parallel nature of this structure type, it seems that an hardware implementation can be efficient. But with a convergence constraint, the parallel evolution is not so simple. To implement an hardware solution of the network, the cairn team proposes a specific parallel evaluation of the neurons evaluate the performance of an GPGPU implementation. Indeed, the GPGPU processors are very efficient structure for parallel programming. In order to compare the performance of the hardware implementation with the GPGPU processors, the cairn team wants to implement the neural network structure in this type. Rather than focusing on the scheduling problem, the objective of this internship is to develop a general implementation of the hopfield neural network. The scheduling problem will be just a simple example of implementation, but the general case

of hopfield model is really important due to the possibility to use this type of structure for other problems.

The work presented in this report concerns the development of a general algorihtm which supports the simulation of a general hopfield neural network. A generic simulation core has been developed, which is completely independent from any specific neural network problem. This generic simulation core reads a hopfield neural network description file, which contains all the information about the neurons and their connections.

1.3 Objective

One application of artificial neural networks is solving the scheduling problem in a heterogeneous computing system, which is one of Cairn team of IRISA. We aim to adapt one special kind of artificial neural network for the solutions to optimize the heterogeneous computing systems.

The objective of the internship is to study and optimize the algorithms in the architecture of GPGPU. In the implementation, we will define and develop an artificial neural network in hopfield model in CUDA. Then we will compare the performance in GPU with the same size on CPU. Both the simulations on CPU and GPU should show that the hopfield neural network will converge in the same condition, because of the same evolution prototypes. We study the impact of parallelizations of the hopfield neural network, to check if it has a better performance/cost rate or not.

1.4 Background

Artificial Neural Network

The artificial neural network intends to grasp the meaning of the essential computations of the central nervous systems of humain beings, which is made up of interconnected neurons and synapses. In 1943, McCulloch and Pitts proposed a model based on simplified binary neurons. Each single neuron implements a simple thresholding function, whose state is either “active” or “not acvtive”. Between the neurons, the synapses are the bridge to connect them and have a connection weight. This “active” or “non-active” state is determined by calculating the weighted sum of the states of its connected neurons. If the sum exceeds the threshold, the state will change to active, otherwise, the neuron will be non-active.

This basic feature of neural network, sometimes can make an useful mathematical function, if we design some kind of neuron connections and active threshold functions. If we add more features in the neural network, we will create various of neural network models for different uses.

There are some advantages of using the neural networks:

- *Adaptive learing* : An ability to learn how to do tasks based on the data given for training or initial experience.
- *Self-Organisation* : An ANN can create its own organisation or representation of the information it reveives during learning time.

- *Real Time Operation* : ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.

- *Fault Tolerance via Redundant Information Coding* : Partial destruction of a network leads to the corresponding degradation of performance. However,some network capabilities may be retained even with major network damage.

For solving the specific problems on the parallel architecture, we will emphasize the third capacity, that is the neural network can run a better performance in parallel hardware and with a optimized algorithms.

In mathematical view, there are three basic components in a simple neural network. Firstly the synapses of the neuron are modeled as weights. The strength of the connection between an input and a neuron is noted by the weight. Secondly are the activities within the neurons. One is summing up all the inputs modified by their respective weights. Another is an activation function controls the amplitude of the output of the neurons.

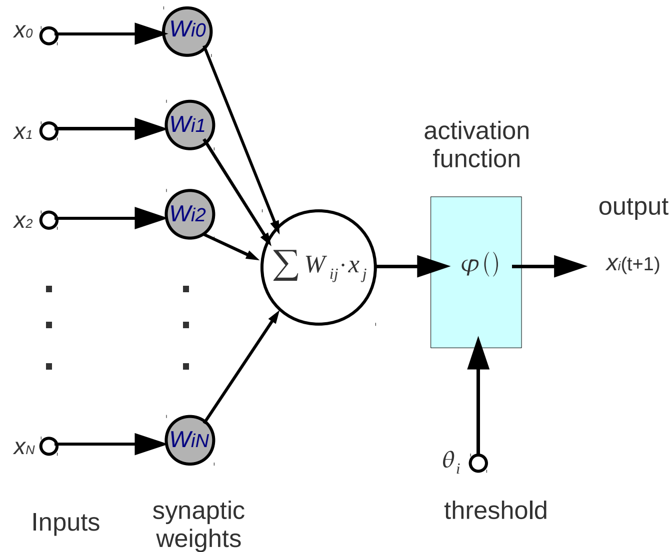


Figure 1: Modeling the process of the neurons

The figure above shows the process of the neurons in mathematical model, $x_0, x_1, x_2, ..., x_N$ denotes the input value of each neuron. A summing operation is executed with the results through synaptic weights $W_{i0}, W_{i1}, W_{i2}, ..., W_{iN}$. The summing value $\sum W_{ij}x_j$ will be sent to the activation function $\phi(\bullet)$ with a threshold value θ_i, then we get the output value of neuron i, or called the state value of neuron i.

This is a simple view of one neuron's update. In a global view, the whole network's update process is called the evolution of the network, which represents

the changes of variables in different concrete problems. Using varies of synaptic weight value between neurons and specific threshold function, we can create all kinds of artificial neural network models adapting many applications. In our reasearch, we focus on the hopfield network model, with dicrete connection and state values.

The Hopfield Model

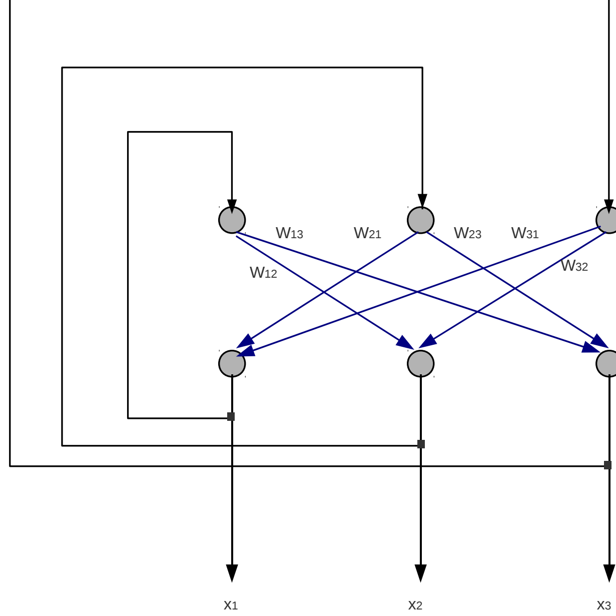


Figure 2: A hopfield model network with diagonal weights zero

In 1982, John Hopfield[3] invented *Hopfield network*, a special kind of recurrent artificial neural network. The hopfield network consistes of a set N interconnected neurons which update their activation values asynchronously and independently of other neurons. All neurons are both input and output neurons. The activation values are binary (1/0, or +1/-1).

The state of the system is given by the activation value y_k. The input $s_k(t+)$ of a neuron k at time $(t+1)$ is a weighted sum:

$$s_k(t+1) = \sum_{j \neq k} y_j(t) w_{jk} + \theta_k. \tag{1}$$

A simple threshold function is applied to the net input to obtain the new activation value $y_k(t+1)$ at time $t+1$:

$$y_k(t+1) = \begin{cases} +1 & \text{if } s_k(t+1) > U_k \\ 0 & \text{if } s_k(t+1) \leq U_k \end{cases}$$

In hopfield model, the value of U_k is *zero*. An associative memory is a primary application in the Hopfield network. In this case, the weights of the connections between the neurons have to be set that the states of the system corresponding with the patterns which are to be stored in the network are stable.

1.5 The parallel algorithms design method for neural networks

As shown in the structure of a neural network, the parallel properties exsit in the simulation process, which is highly depend on the model we use. Generally speaking, when we design a parallel algorithms for a neural network, we should firstly assure the correctness of a learning process. If a deadlock happen, then we will lost a lot of efficient and even get a false result.

Creating an effective parallel program requires evaluating cost as well as performance. Besides the cost of the hardware, we should consider the resource requirements of the program on the architecture, for example ,its memory usage. While costs and their impact are difficult to quantify than performance, we often decide to compromise performance to reduce them. In both sides, as in algorithms designs, we should favor high performance solutions that keep the resource requirements of the algorithm small, and as in architecture designs, we should try to aim to high performance systems that facilitate resource-efficient algorithms and reduce the programming effort. The general process of a parallel algorithm can be illustrated in Table 1.

The hopfield network structure mappings and the implementation details will be discussed later.

Table 1: The Parallel Algorithms Design Process (Source: *Parallel Computer Architecture : a hardware/software approach. 1999*

Step	Architecture Dependent	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload
		Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality
		Reduce communication and synchronization cost
		Reduce serialization at shared resources
		Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary
		Exploit locality in network topology

2 State of Art

Compared with sequential programs, that are generally executed on a central processing unit (CPU), the parallel computing uses multiple processing elements with multiple processors, such as in networked computers , specialized hardware, or in GPUs. A multi-core processor is a single component with two or more independent processors which is usually used in heterogeneous computing and other multithreading applications. Those applications also include domains of general-purpose, and the NVIDIA's graphics processing unit (GPU) is specially adopted designed for that use. A GPU is generally present on a video card, for accelarating the graphics, and beside this, many data intensive programs are

developed in General Purpose GPU (GPGPU) architecture with a delicate parallel algorithm.

In the domain of artificial neural networks, there are various reasearch papers on expressions of the neural network's natural parallel architectures, and some of them talked about the realizations on GPUs, but mainly in a plain neural back-propagation model. As in hopfield model, which is after the author's name, the principle ideas are created and presented in his paper that concerns the neural network and the optimization problems, and concludes a new way of solving problems through dedicated networks. A TSP (Traveling-Salesman Problem) solution is also simulated in this kind of neural system.

Nordstrom and Svensson [5] proposed some levels of paralleslism in the neural network, each of them defines different parallel computations:

- training-session parallelism (simultaneous execution of different sessions);
- training-example parallelism (simultaneous learning on different training examples);
- layer parallelism (concurrent execution of layers within a network);
- node parallelism (parallel execution of nodes for a single input);
- weight parallelism (simultaneous weighted summation within a node);

The first two levels of parallelisms are prepared for the training of neural networks, when they are learning a series of examples and executed by a quantities of sessions. The third parallelism is suitable for the multiple-layer perceptron, but there are still some dependency problems to solve. The node parallelism and weight parallelism exist in the nature activation process of a neural network. In the reality, they are independently executed from the other neurons, and the summation of the weights is biologically gathered rather than sequentially computed. It's reasonable to employ a software or hardware parallel technique to realize a artificial neural network.

Nikola B. Serbedzija[7] made a summary about the techiques in the parallelisation of simulating the artificial neural network, especially in the specific parallel architectures. Several theoratical parallelization methods for neurosimulations are proposed and analysed his paper. As discussed in the paper, those simulations on general-purpose parallel computers are done in late eighties. The main techniques follow the neuroparadigm of interpreting the neural network by matrix-vector operations. Among the results obtained in several types of parallel architectures, the best one is the data-parallel machine. And the data-parallel program is customized in this kind of machine to mapping efficiently the neural network data structure. And they also showed that the best performance should appear in particular neural model design, because the mapping and executing are highly dependent upon the particular algorithms.

Udo Seiffert[8] initially implemented a multiple-layer perceptron network on a massively parallel computer hardware, to show the possibility of accelerating the evolution process of a neural network on several types of parallel architectures, including heterogeneous computer clusters and multi-processors machines.

Till now, people tend to use a general-purpose parallel architecture, to have a flexibility of an artificial neural network designs. And for each kind of neural

models, they developed different algorihms and implementation methods to get a better performance. They are mainly data-parallel style as mentioned before. The data of a neural network is distributed to all processors, and they execute the same program with a centralized synchronization. That means the multiprocessors execute a single set of instructions (called SIMD, Single instruction, multiple data). Different threads will execute the same program but different pieces of data. For instance, F.Valafar and O.K.Ersoy explored a parallel implementation of back-propagation model network without hidden layer on MasPar MP-1[6], which is one of the SIMD architecture supports data parallelism.

Some people also reviewed the implementation of a range of neural network models on SIMD architecture, especially the hopfield network model[4]. On the Distributed Array Processor (DAP), a 4096-processor SIMD machine, they proved the parallel algorithm of hopfield model can solve some problems like image restorations, and it appeared to be an effective solution for other purposes.

However, in the simulation of neural network, people trend to design various of implementations for general purpose, so the GPU is usually chosen as a typical SIMD architecture to do neural networks simulations.

Mario Martinez-Zarzuela et al.[9] presented an implementation of a Fuzzy ART Neural Network on the GPU. Their method can reach 33 times on a GeForce 7800GT graphics card. Jayrom M. Nageswaran et al.[10]simulated a large-scale Spiking Neural Network running on an NVIDIA GTX-280. Their GPU's spiking neural network model simulated 26 times faster than CPU version with 100K neurons with 50 million synaptic connections, but only 1.5 times slow than real-time with a network of 100K neurons and 10 million synaptic connections. Ryanne Dolan and Guilherme Desouza[11] simulated a massive parallelism of cellular neural network in General-purpose GPUs, and proposed the abstraction for the cellular neural networks and tested an image processing library performance using GPUs, showing that their parallel algorithm can give a significant acceleration. These works focus on different kind of neural network for particular use, thus the implementations more or less lack of flexibility to apply the algorithm to other applications.

For some popular models of neural network realized in GPU, such as back-propagation model, Noel Lopes and Bernardete Ribeiro[13] described a parallel implementation of the Multiple Back-Propagation (MBP) algorithm and presented the results on benchmarks. They showed that their implementation on GPU can reduce the computational cost compared with the CPU version when solving the classification and regression problem. A. Guzhva et al.[12] implemented standard back-propagation algorithm for training multiple perceptrons simultaneously on GPU, and they proved that the parallel computations had lead to 50x speed increase compared to a CPU-based program. These training multiple layer perceptron networks have 1 hidden layer, 1648 inputs, 8 neurons in the hidden layer and one neuron in the output layer.

And in recent years, people trend to use algebra method in GPU architecture, those methods can be extended to a range of applications with similar parallelism trainings. For example, Dominik Scherer, Hannes Schulz and Sven Behnke[24] create a multi-level parallelism of convolutional neural network for Nvidia's CUDA GPU architecture to accelerate the training process. Daniel L.Ly, Volodymur Paprotski, Danny Yen[14] investigated the advantage of GPUs in the inherent

parallelisim of neural network in type of Restricted Boltizmann machine, and showed the algorithm has a better performance running on GPUs.

In the previous work, there's no specific parallel program of hopfield neural network training for solving optimization problems on GPU architectures. We note that these implementations have only took measures in particular field, such as pattern recognition, or simply proved the advantage of using parallel architecture. The possibility of using hopfield network on GPU towards optimization problems is still an interesting topic, while the new parallel mechanism has been discovered, and the new computational techniques have been developed. So it is quite necessary to discuss the potential of employing the newly parallel algorithms of hopfield network towards an optimization problem.

So we would study relative algorithm design methods, the particular data structure of this artificial neural network focusing on hopfield model, and to employ it with CUDA. We use several techniques for accelerating the parallel algorithm performance, to show an advantage to the traditional CPU implementations.

3 Basic implementation of neural network

Firstly we take the data structure into account in hopfield model.

In a artificial neural network, the neurons are basic units that each of them has a state. The active state is triggered by a certain level of activation, which is transmited by synapses (connections between neurons). So the network is represented by a vector of every states of neurons, a vector of external input, and a matrix of weights of connection.

With hopfield network model, we create a CPU program to simulate the evolution of the network. A typical network data can be designed as two vectors and a matrix structure:

```
typedef struct
{
  //Number of neurons
  int nbNeurons;
  //State of neurons
  int *neuron;
  //Matrix of the weights of the connections;
  int **connection;
  //Table of the external inputs of the neurons;
  int *input;
}NNetwork;
```

And the executing function for one neuron is represented as follows. We pull out a neuron randomly for calculating its new state. The initial activation value is the input value of this neuron. Then we collect the augumentation of every activation through the weighted synapse and the states from the other neurons. The new state of this neuron is a biased value after a threshold function. (if the activation value less or equal to 0, then the neuron will be non-active, otherwise, the neuron will become an active neuron.

We note every global state for a sample network of 18 neurons.

```
Update(NNetwork *pN)
{
  int activation=0;

  //may introduce parallelism
  int randomIndex=(int)((pN->nbNeurons*1.0)
                        *rand()/(RAND_MAX+1.0));
  activation=pN->input[randomIndex];

  for(j=0;j<pN->nbNeurons;j++)
    activation+=(pN->connection[randomIndex][j])
      *pN->neuron[j];

  pN->neuron[randomIndex]=threshold(activation);
  //end
}
```

Then we get the convergent times of a 18 neurons network. In the beginning, at time 0, the states of neurons are initialized randomly, with value 0 or 1, which represents “not active” or “active”. And with the calculation for a certain index neuron’s new state, the global state of network changes. For example we set 6 as the number of our expected active neurons, then finally we get 6 active neurons and the global state of network does not change any more at time 33, and from time 34, the neural network will keep the same state, and we regard this global state as a stable state.

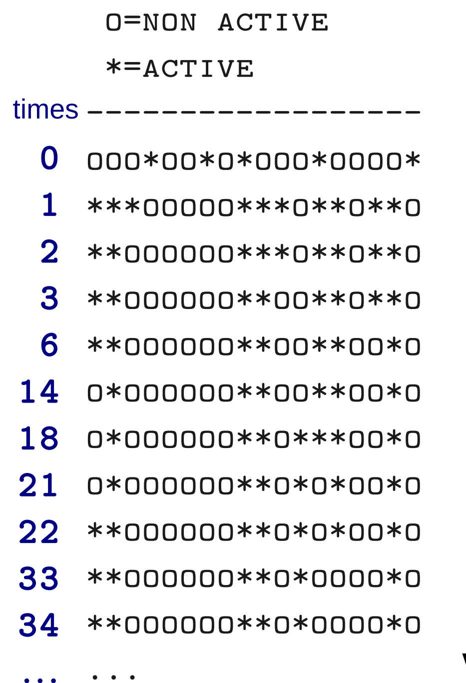


Figure 3: convergence times of hopfield neural network

3.1 Network adapting to an optimization problem

Actually, the optimization problems are proposed in the time-scheduling on heteogeneous system as we mentioned before, which is similar to the job-shop scheduling problem: to assign the resources at particular times. We aim to design an opimization algorithms to make a scheduler for the design of heterogeneous multiprocessor architectures. There are some kinds of scheduling algorithms such as round-robin, fair queuing, fixed priority pre-emptive scheduling. However, the neural network method is proved to be an efficient one, especially in a hopfield model.

Hopfield neural networka are originally applied in optimization problems such as *Traveling Salesman Problem*(TSP), proposed by J.J.Hopfield[2].

The solution algorithms of the optimization problems are often in forms of evolutionary computations. It follows the mechanisms from biological evolution. One example is *genetic algorithm* in a process of natural evolution. Scheduling problem is one of the appropriate problems for genetic algorithms. But sometimes the genetic algorithms can not get a convergence to the solution and fail to get the right direction to the optimal.

Compared with the genetic algorithms, the *branch and bound algorithms*, which is proposed by A.H.Land and A.G.Doig, seems to be more suitable to solve the local optimization problems, while the GA is preferred to find the global optimization. Genetic algorithm is based on a heuristic method to search the optimal solution. While the branches are prunned, we will finally get the optimal element until the lower bound reach the upper bound. Because the feature of probability dependency, it's more suitable for a game strategy decision rather than a hardware-based scheduling problem.

Another possible solution of evolutionary algorithms is *artificial neural networks*. The evolutionary computation of neural network can perform efficiently by GPU architectures, since the computations can be distributed into quantity of threads in GPU resources. Similar to genetic programming, the evlolutionary process of neural network follows the rules in nature parallelism, and GPU architecture supports well to this kind of applications. Thus we can obtain a evolutionary accelaration with these speeding-up features on GPUs. The essential acceleration parts exist in the execution of small programs simultaneously. In the field of time-scheduling optimization problem, hopfield model neural networks have a special feature with which we can find an optimal by a convergence method. The network will converge to a stable state with the energy function decrease.

The most unique aspect of the hopfield model is the introduction of energy function of Lyapunov. An artifical neural network evolves to the stable state under some conditions below:

- The matrix that represents the weights of the connections between the neurons is non-positive and symetric;
- The diagonal elements are zeros;
- The activation function is always increasing.

That means the system always evolves without increasing the Lyapunov function value, which assures us a stable destination of the system under these restrictions.

And the energy function can be presented as follows:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} W_{ij}x_i x_j - \sum_{i=1}^{N} I_i x_i \quad (2)$$

Where W_{ij} denotes the weights of connections between neuron i and neuron j; x_i denotes the states of the neuron i (0 or 1); I_i denotes the external input to the neuron i.

With this hopfield model, we want to find a neural network that the minimum of the energy function is associated with the optimization problem. So we should choose elaborately the weights W_{ij} of the connections and the external inputs I_i of the network.

Then the solution of the optimization problem is decomposed into four steps:

- **1**. Find a topology of the network in which the outputs of the neurons are interpreted as the solution of the problem.
- **2**. Construct an energy function in which the minimum corresponds to the best solution.
- **3**. Calculate the weights of the connections between the neurons and the inputs to apply to each neuron, from the energy function that already found.
- **4**. Since the corresponding neural network are found, we let it evolve to stable state.

After the iteration, if the network tends to a stable state and the energy value is at minimum, then we get the satisfying constraints and we find a solution of the problem; If not, we can still try some different initial states or re-execute many times to find a solution.

3.2 Typical k-out-of-N Rule Networks

Because of the discrete properties of hopfield network, some integer related engeering problems can be defined with neural network in certain cases. According to the hopfield model, the value of connection weights are -2s, that means, every neuron connects with other neurons by weight value of -2 and there's no self-connections. For initialization, the network will start with external inputs to each neurons, which differs from neuron to neuron. One neuron belongs to one or serveral sub-network in which a *k-out-of-N* rule applies (k represents the expected number of active neurons, and N represents the size of the sub-network). Different parts of the whole network are applied different rules.

As in the figure below, we have a 3 tasks $\times$ 6 cycles network, in which every neuron has an input. The input value is defined by the applied rule. We made a number of active neurons for each task and each cycle. The input value of a neuron is calculated by $input[i] = \sum 2k_i - 1$. k_i is the ith rule applied in a certain neuron. Every neuron is applied by two rule both in tasks and in cycles.

The objective is to get a developed network in which all the rules have been applied and all the neurons will not change their states any more. This developing process is called the evolution of a neural network. Evolution is usually described

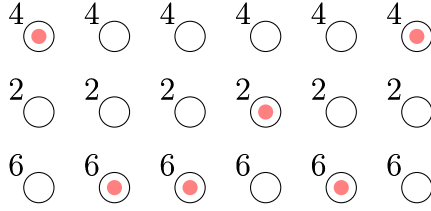


Figure 4: 3 tasks × 6 cycles network

as the training processes in some other kinds of neural models. So we need to design a way that the neural network evolves.

The hopfield network is a kind of recurrent neural network, which is different from feedforward neural network (has only one forward direction, such as multi-layer perceptron). These neurons within the hopfield network update their states asychronously and independently from other neurons.

3.2.1 The k-out-of-N rule in the simulations

For constructing this hopfield neural network, Gene A.Tagliarini, J.Fury Christ and Edward W.Page[27] have introduced a rule called *k-out-of-N* which specifies the weights of the connections and the external inputs to apply to the neurons, both considering the equality and inequality constraints.

Imagine a network in which each neuron represents a hypothesis that can be true or flase. If a neuron is active when the network reach its equilibrium, its corresponding hypothesis is true, if not, the corresponding hypothesis is false.

The optimization problems in which the solution indicating the values subjecting to binary values are known as the *zero-one programming problem*, usually have the constraints as the following type:

$$\sum_{i=1}^{N} x_i = k \tag{3}$$

where x_i represents the variable of the decision binaire, k, N are positive integers satisfying $0 \leq k \leq N$.

3.2.2 Considering equality constraints

Thus, when the hypothesis are represented by the neurons, the equality above will calls for exactly k neurons in the set of N neurons may be active when the network reach a stable state.

The cost function that we want to minimize in an optimization problem can be shown as follow:

$$E = (k - \sum_{i=1}^{N} x_i)^2 \tag{4}$$

Only when the sum of the neurons of state 1 is k, this cost function will be 0. In other case, it will be positive. This function can be actually taken as the form of an energy function of a neural network, which respects all the convergence properties. The energy function can be expressed as:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1,j\neq i}^{N}(-2)x_i x_j - \sum_{i=1}^{N}(2k-1)x_i \quad (5)$$

This equation specifies the weights of the connections and the required inputs for realizing a network of N neurons respecting all the convergence properties and having some stable states with exactly k active neurons. The matrix of the connection and the corresponding input vectors are shown as:

$$W_{ij} = \begin{cases} 0 & \text{if i=j} \\ -2 & \text{if i}\neq\text{j} \end{cases} \quad I_i = 2k-1$$

The state reflects if one neuron is active or not, we define an active state as 1, and a non-active state as 0, thought in natural neural network each neuron has a continuous level of activation. For simplifying the calculation, we made it binary.

Therefore, for a hopfield network, the data structure can be described as a vector of the states of neurons $Neurons[\ ]$ (with values 0/1), a vector of external inputs $Inputs[\ ]$ (with values $2k_i - 1$), a matrix of the connection weights between the neurons $Connection[\ ][\ ]$ (with values $-2/0$).

3.3 Rules and Algorithms

Here we define the essential data of a hopfiel neural network prepared for evolutions:

NUM 18 defines the total number of neurons

NUM_RULES 9 defines the number of rules(sub-networks)

RULE 1 : [0,1,2,3,4,5] → 2

RULE 2 : [6,7,8,9,10,11] → 1

RULE 3 : [12,13,14,15,16,17] → 3

RULE 4 : [0,6,12] → 1

RULE 5 : [1,7,13] → 1

...

RULE 9 : [5,11,17] → 1

These rules are created for producing proper input values and a connection matrix for a hopfield neural network. In the network of 18 neurons mentioned before, we expect there are eventually 6 active neurons, that distribute in every column, and 2, 1, 3 active neurons in the rows separately. Then 9 $k-out-of-N$ rules are needed to create suitable input values and its connection matrix.

A PHP script is developed to generate the input values and connection matrix by the algorithm in the last subsection. If the size of the network (number of neurons) is large or there are too many rules to be applied, we can also use C script to generate the input value and connection matrix data.

The size of network (number of neurons), vector of input value and the connection matrix are stored in an input file. Once the input file is created, we have sufficient data and thus the whole neural network is established. The simulators, the executing kernels on CPU/GPU, are able to read the input file of an established hopfield neural network.

1 int i,j,N,$index$,$neurons$
2 int $state[N]$; %Table of states of a neuron
3 int $connection[N][N]$; %Matrix of the weights of connections
4 int $input[N]$; %Table of external inputs

Algorithm 1: Definitions in algorithm of Hopfiel Models

Then we apply a rule of *k-out-of-N* for several subsets of the network. these subsets represent one part of the network, in some cases, they represent the tasks and cycles of a rectangle-shape network.

```
foreach ($neurones[$setname] as $neurone1) {
  $neuroneinput[$neurone1] += 2* $valeur -1;
  foreach ($neurones[$setname] as $neurone2) {
    if ($neurone1 == $neurone2) {
      $Matrix[$neurone1][$neurone2] = 0;
    }
    else {
      $Matrix[$neurone1][$neurone2] = -2;
      }
    }
}
```

1 **Procedure** ApplyKoutofN(int k, int N) **for** $i = 1$ **to** N **do**
2 $input[i] \leftarrow (2k - 1)$; %initialization of the inputs **for** $j = 1$ **to** N, $j \neq i$ **do**
3 $connection[i][j] \leftarrow -2$; %initialization of connection weights;
4 **end**
5 **end**
6 **End of Procedure**;

Algorithm 2: Apply *k-out-of-N* rule to subnetworks

```
1  %Read parameters of the network from input file
2  repeat
3      %Simulation of the evolution of network
4      for i = 1 to N do
5          index ← Rand(N);  % a neuron is pulled for calculating its state
           activation ← Input[index];
6          for j = 1 to N do
7              activation ← activation + Weight[index, j] * Stat[j];
8               % summing calculation from the products of weighted connections
               and states
9          end
10          %Application of a sigmoid of infinite gain
11         if activation < 1/2 then
12             Stat[index] ← 0;
13             the new state is not active
14         else
15             Stat[index] ← 1;
16             the new state is active Stable(index);
17              %Detect whether the network change after nbStable iterations
18         end
19     end
20 until (Energy(Stat[]) = 0) ;
21  % Energy() is the function that calculate the network energy, after the
   neurons states and applied rules, it will return an integer.
22  % As for the stop condition of the process, a function of Stable() can be used
   with the state vector to detect the stabilization of the network, it will return a
   boolean of false if more than one neuron has been detected with a change of
   state, and true if there isn't any change since nbStable.
23 ShowAllResults()  $ Show All the final states of the neurons.
```

Algorithm 3: Principal Algorithms

3.3.1 Host/Device Data Preparation

CUDA assumes the threads on GPU may execute on a physically separate *device* that operates as a co-processor to the *host*. And CUDA also assumes that both the host and device maintain their own DRAM, referred to as *host memory* and *device memory*.

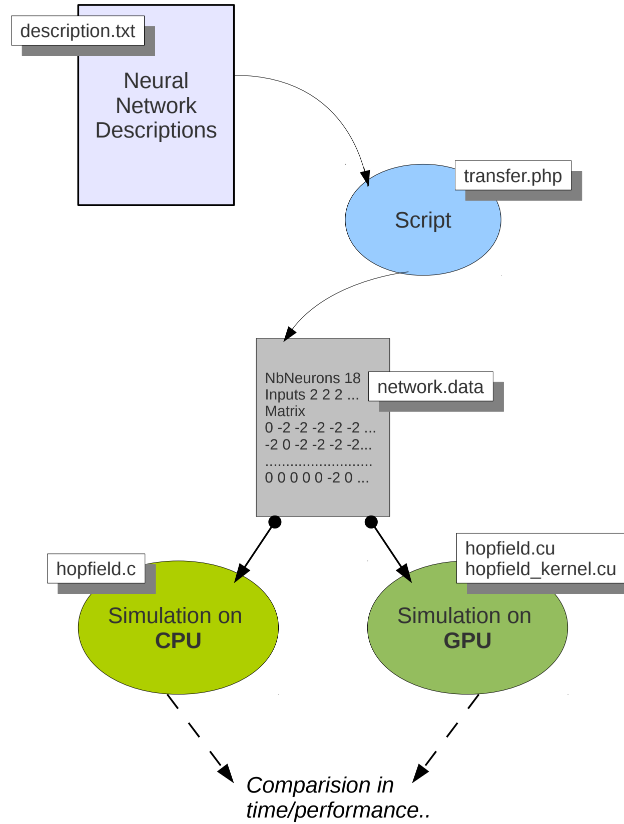


Figure 5: Data Transaction and Processing

The conventional way is to allocate the memory on CPU for the whole network at first, and tranfer the data (initial states, inputs vector, and connection weights matrix) to GPU. Before this, we have to prepare all the data that both CPU and GPU need. The data processing transaction is illustrated as follows:

To ease the description of some optimization problem, we have also developed a script which is dedicated to the scheduling problem. This script enables to define the problem at high level and to produce the neural network description.

Firstly we put the description referring the index and rule information in file, and transfer this description information into a hopfield network data file. The data file contains all the information about a hopfield network : the neuron number, input values, and the connection values between neurons. The transfer script is written in PHP/C and runs on CPU for generating the proper data. Then we design the simulator on CPU and on GPU separately, using the same generated data. On the side of CPU, corresponding the *host* side, will execute the programs

using the naive network data. On the side of GPU, corresponding the *device* side, will launch the kernels using the data allocated from CPU. So we should consider this CPU/GPU data transfer time when comparing the complete time. And also, we should assure both the simulators get the correct result to give a fine solution to the optimization problem, which will be discussed later.

4 Parallelization of neural network

In this section, we introduce the details of parallelizations of the evolution process of hopfield network. This parallel algorithm contains several aspects of parallelism, using different levels of GPU memory. And also, the partitions of network data in each method are different according to the problem size and the algorithm to use. These kernels will accelerate the whole evolution process since they will built in the GPU simulator of the hopfield network.

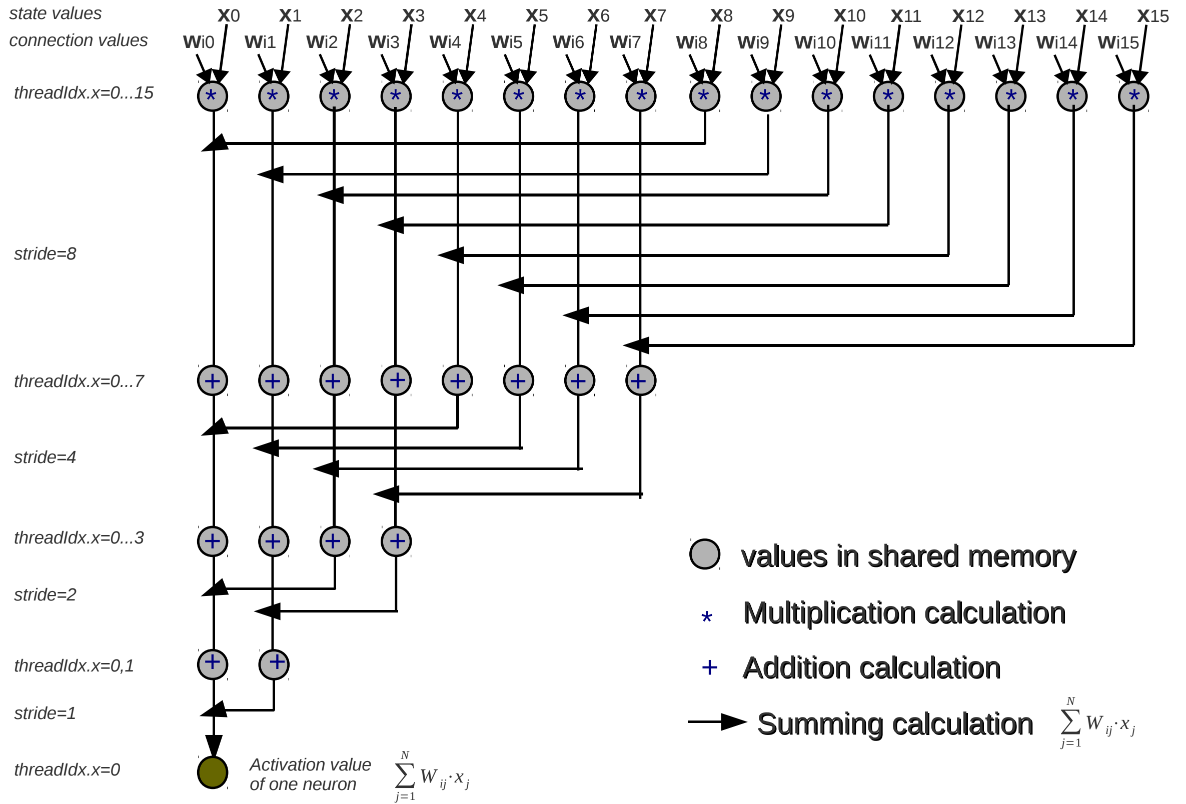


Figure 6: The reduction process within one block

4.1 Internal parallelization

Firstly we consider the parallelism of update of one single neuron, which is independently proceed from other neurons. In this calculation, the result is not relative with other calculations in other neurons. Thus an internal parallelization is proposed to the computations within one neuron.

As introduced by previous section, the data for one neuron update is stored in a block on GPU. Thus, the state update of one single neurons uses the technique of "scanning-sum" reduction within one block.

An important accelerating part of the process is the reduction of vector scanning. This is a typical data-parallel programming in a loop-level. In GPU we use multiple thread blocks to execute a scanning and a summing up calculation.

Here we have an object to optimize the algorithm efficiency, which means to reach a peak performance level to get a full use of the parallel advantage. In CUDA there is no global synchronization across all thread blocks, because that will cause more expense in hardware, moreover, using too much blocks will also produce some deadlocks. So in this case, we assign the task into multiple kernels, which serves as a global synchronization point. Thus, decomposing computation into multiple kernel invocations will avoid global synchronizations.

A performance measuring technique will be employed during the optimization of parallelizations:

- **GFPLOP/s** : for compute-bound kernels
- **Bandwidth** : for memory-bound kernels

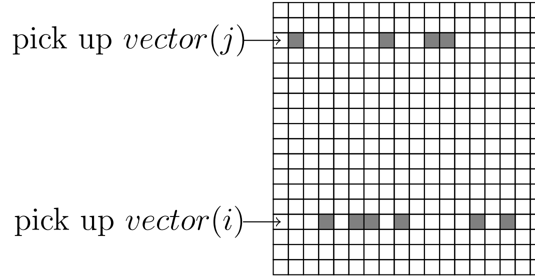
To reach this performance, a mechanism of parallelization of neural network computations is proposed: reduction of calculations of a new activation value to one neuron. For example, in the figure below, we have $N = 16$ neurons in the network, and we want to calculate the new activation value of neuron i, which is represented as $\sum_{j=1}^{N} W_{ij} \bullet x_j$.

Two step of calculation is needed: one is multiplication calculation between the connection value and the state value in every neuron, another is the addition (summing) calculation in the network. The product of W_{ij} and x_j in each neuron is calculated in parallel by threads, then we get $N = 16$ products. In the second step, we add all the 16 products in reduction, as shown in the figure below. The temporary summing data, including the first N products, are stored in the shared memory on GPU.

4.2 External parallelization

The previous calculation within one block only concerns one neuron update by dot production of two vectors. However, the neurons can update independently with other neurons since there's no synchronization among the update steps. In this point of parallelism, we propose to update the neurons in the same time, and then to constate whether this network converge or not.

But to evaluate neurons in exeternal parallelization, normally we have to select the neurons that are independent, that is to say, there's no connection/synapse between the neurons selected. Oherwise, the connection between neurons can affect on the convergence of the network.



In this process, The connection matrix is interpreted in a 2D texture. We note that the connection matrices in hopfield networks, are populated primarily with zeros, so called sparce matrix. Only the pixels which represents the connections between neurons by rules are populated with -2. We have to give special optimization to sparce matrix since zero elements indexing and execution will waste lots of computing resources.

Because of the limitations of GPU data storage, we need to transfer the raw data of neural network to some formats, avoiding overloading the matrix and vector data on GPU. There are several formats to store this connection sparce matrix, with different specifications, and methods of manipulations of the matrices [15]:

- **Diagonal Format**(DIA) Type that all the non-zero values are restricted to several matrix diagonals. In this format, the row and column indices of each nonzero entries are implicitly defined by the positions within diagonals and the corresponding offsets. So the diagonal format is stored in two arrays of `connection` (nonzero values) and `offset` (the offset of each diagonal from the main diagonal). But this format is not suitable for a general sparce matrix like hopfield connection matrix whose non-zero values are not alway restricted in the diagonals, though many of them are symmetric.

- **ELLPACK Format**(ELL) Type that the non-zero values are stored in a dense $M-by-K$ array data, for an $M-by-N$ matrix with a maximum of K non-zeros per row. The corresponding column indices are stored in `indices`. This format is more general than the diagonal format.

- **Coordinate Format**(COO) Type that the values are represented simply by row and column indices. This type is the most general format for any kind of matrix, which is represented in arrays of `row, col` and `connection`. We use this type for storing the raw data of the connection matrix of hopfield network.

- **Compressed Sparse Row Format**(CSR) Type stores column indices and non-zero values and an array of row pointers, seperately represented as `indices, connection` and `ptr`. This is an efficient storing format for connection matrix when we operate sparse matrix multiplications.

Among these formats of sparce matrix, the compressed sparse row format is the most popular and general purpose representation for the use of a connection matrix of hopfield model. One reason is that compression method is comparitively simple and efficient, another reason is that, for a great size of neural network, it

require less storage space for the vectors and connections values in CPU and GPU, so that the loading data on GPU will decrease dramatically, and more importantly, less calculation time for updating the network states.

In CSR format, we represent the $N \times N$ connection matrix non-zero values in a `connection` vector full of -2. The vector of `indices` indicates the column indces. The vector `ptr` indicates the row pointer (neuron index) and it has a length of $N + 1$. For example, `ptr[i]` stores the offset of the i_{th} neurons. The vector of `start_neuron` and `end_neuron` are the beginning and ending indices of the non-zero neurons connections in the matrix. The following arrays can be checked as a typical CSR format matrix storage that will be integrated in multiplication operations on GPU.

The connection matrix of 6×6 of a hopfield network of 6 neurons is as follows:

$$\begin{bmatrix} 0 & -2 & -2 & -2 & -2 & -2 \\ -2 & 0 & -2 & -2 & -2 & -2 \\ -2 & -2 & 0 & -2 & -2 & -2 \\ -2 & -2 & -2 & 0 & -2 & -2 \\ -2 & -2 & -2 & -2 & 0 & -2 \\ -2 & -2 & -2 & -2 & -2 & 0 \end{bmatrix} \tag{6}$$

And this connection matrix will be resolved in compressed sparse row format with a vector of matrix data, a vector of indices and a vector of pointers.

$$\begin{cases} \textit{nonzero values} & \texttt{connection[126]} = \texttt{[-2 -2 -2 -2 ... -2]}; \\ \textit{colomn indices} & \texttt{indices[18]} = \texttt{[1 2 3 4 5 6 12 ...]}; \\ \textit{row pointers} & \texttt{ptr[19]} = \texttt{[0 7 14 21 ... 126]} \end{cases} \tag{7}$$

$$\begin{bmatrix}
0 & -2 & -2 & -2 & -2 & -2 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\
-2 & 0 & -2 & -2 & -2 & -2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\
-2 & -2 & 0 & -2 & -2 & -2 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 \\
-2 & -2 & -2 & 0 & -2 & -2 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\
-2 & -2 & -2 & -2 & 0 & -2 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 \\
-2 & -2 & -2 & -2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 \\
-2 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -2 & -2 & -2 & -2 & -2 & 0 & 0 & 0 & 0 & 0 \\
0 & -2 & 0 & 0 & 0 & 0 & -2 & 0 & -2 & -2 & -2 & -2 & 0 & -2 & 0 & 0 & 0 & 0 \\
0 & 0 & -2 & 0 & 0 & 0 & -2 & -2 & 0 & -2 & -2 & -2 & 0 & 0 & -2 & 0 & 0 & 0 \\
0 & 0 & 0 & -2 & 0 & 0 & -2 & -2 & -2 & 0 & -2 & -2 & 0 & 0 & 0 & -2 & 0 & 0 \\
0 & 0 & 0 & 0 & -2 & 0 & -2 & -2 & -2 & -2 & 0 & -2 & 0 & 0 & 0 & 0 & -2 & 0 \\
0 & 0 & 0 & 0 & 0 & -2 & -2 & -2 & -2 & -2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & -2 \\
-2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -2 & -2 & -2 & -2 \\
0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & -2 & 0 & -2 & -2 & -2 & -2 \\
0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & -2 & -2 & 0 & -2 & -2 & -2 \\
0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & -2 & -2 & -2 & 0 & -2 & -2 \\
0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -2 & -2 & -2 & -2 & 0 & -2 \\
0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & -2 & -2 & -2 & -2 & -2 & -2 & 0
\end{bmatrix}$$

Figure 7: A complet 18 neurons hopfield network connection values

For the dense linear algebra programs on GPU, people use **BLAS** (The Basic Linear Algebra Subprograms) library to compute algebra computations such as dot product, vector sums, and matrix-matrix multiply. This is a basic algorithmic library intergrated in CUDA and a most general one.

Based on this, NVIDIA has provided a *CUDA Data Parallel Primitives Library*(CUDPP) in data-parallel algorithm primitives such as parallel prefix-sum

scan, sort, and reduction. We use its one of the data-parallel algorithm called *segmented scan* to complete the scanning process, which is based in a tree structure. However, the implementation of CUDPP has inefficient global memory accesses, and there are some bank conflicts in the shared memory accesses. Dotsenko et al.[16] have proposed an improved matrix-based segmented scan algorithm which reduces the shared memory bank conflicts and synchronizations.

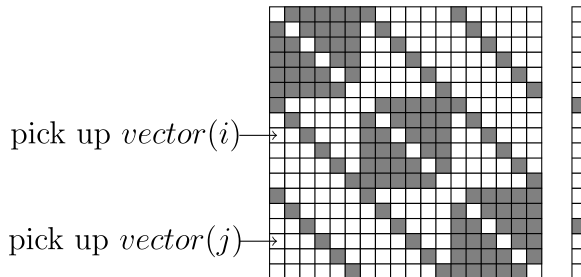
And also, particularly for Sparce Matrix parallel computing, NVIDIA has implemented another library called *CUSP*, a generic parallel algorithms for sparce matrix and graph computations. Along with a C++ STL integrated library *THRUST*, they provide a efficiency and flexibility interface in the implementation of matrix parallel oriented applications.

In the simulation of hopfield network, we propose an iterative method: steps of evolution of the network are regarded as iterations of matrix-vector multiply operations.

At a moment t, we have a state vector of hopfield network, noted as $\mathbf{S}$. We select one group of neurons to update them simulateneously (in the beginning, we choose all the neurons as a group), after the one iteration, we will get a new global state at the moment $t+1$:

$$\mathbf{S}_{t+1} \leftarrow Sigmoid(\mathbf{I} + \mathbf{W} \bullet \mathbf{S}_t) \tag{8}$$

Where $\mathbf{W}$ denotes the connection weight matrix of the hopfield network, $\mathbf{I}$ denotes the input vector, and $\mathbf{S}_{t+1}$ denotes the updated state after $\mathbf{S}_t$. The new global state is calculate by the matrix-vector product operation. We should note that in one iteration, all the neurons participate the updating process.



The computational kernel of this matrix-vector multiply is executed by fragment programs, which computes the inner product of one neuron's connection vector (row) and global state vector. In the GPU architecture, this is realized by rendering groups of rows with an equal number of non-zero entries,[17], that means each group is a rectangle associated to a fragment program specialized for a particular number of non-zero entries.

Thus, for accessing these non-zero entries in a given row and the associated elements of the vector, the connection matrix $\mathbf{W}$ and state vector $\mathbf{S}$ are stored in textures requiring appropriate indirections. The texture $\mathcal{X}^x$ holds vector $\mathbf{S}$ with one element per pixel.

The sparce connection matrix $\mathbf{W}$ of neural network is stored in two textures. One is the diagonal entries $\mathcal{W}_i^x$ are laid out the same as $\mathcal{X}^x$, i.e., W_{ii} is at the same coordinate as x_i. Another is the off-diagonal,non-zero entries of matrix $\mathbf{W}$, they are stored in a texture $\mathcal{W}_j^a$, with one segment per matrix row. Each segment's

address is stored in an indirection texture $\mathcal{R}^x$. Then we have a texture $\mathcal{C}^a$ keeping the correspondence between the addresses of w_{ij} in $\mathcal{W}_j^a$, maching x_j in texture $\mathcal{X}^x$.

Firstly we read the elements of connection matrix **W**, with a high memory bandwidth provided by GPU. After reading the non-zero elements and explicitly stored zero elements, we read the stored element coordinates.

5 Implementations

5.1 Platform

Our implementation and performance analyses are based on the device of NVIDIA's video card GeForce GTX 480".

- CUDA Driver Version : 3.20
- CUDA Runtime Version : 3.20
- Total amount of global memory : 1610285056 bytes
- Multiprocessors × Cores/MP = Cores : 15(MP) × 32(Cores/MP) = 400(Cores)
- Warp size : 32
- Maximum number of threads per block : 1024
- Maximum sizes of each dimension of a block : 1024 × 1024 × 64
- Maximum sizes of each dimension of a grid : 65535 × 65535 × 1
- Clock rate : 1.40 GHz
- Cuda compilation tools : NVCC , release 3.2, V0.2.1221

Table 2: CPU & GPU platform specifications

CPU	4 Cores of Intel(R) Xeon(R) CPU, W3540 @ 2.93GHz
Memory	6111428 kB
PCI Device	Intel Cprposration 5550
OS	Ubuntu 10.04.2 LTS
GCC Compiler	gcc version 4.3.4

This video card is equiped within the server machine. And the profiler of the programs will be measured by the application itself.

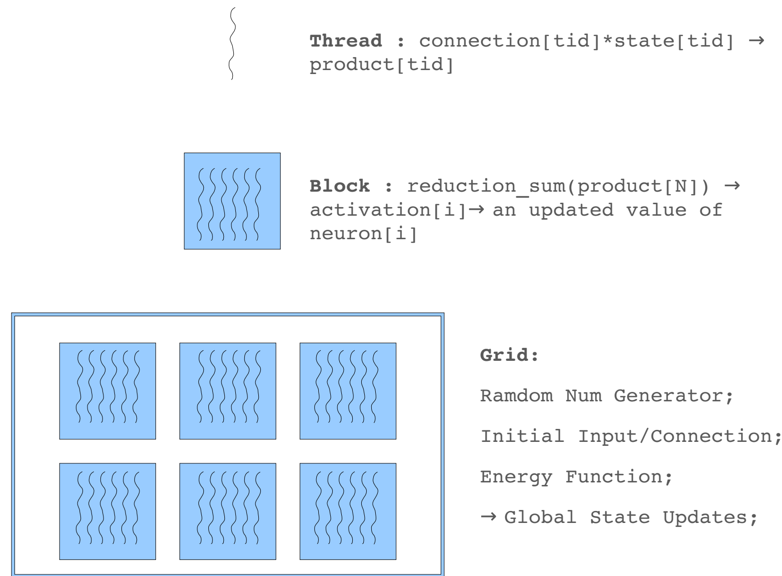


Figure 8: CUDA memories use in GPU execution

5.2 Simulation infrastructure

5.2.1 Mapping Network Structure on GPU

There are several levels of memory of GPU which have different access and process speed : the global memory , shared memory and registered memory, which is called the memory hierarchy. The *thread* data is handled by its *per-thread local memory*, while the *thread block* uses a *per-block shared memory*. And for all the *grid*, the *global memory* is accessible and it has the lowest speed.

Execution Configuration

In host function, we must configure the function that called by CPU. The form is `<<<gridDim,blockDim,Ns,S>>>`. `gridDim` is of type `dim3` specifies the size of the grid. Here we use one dimension of grid, and in our platform, $gridDim = 65536$. `blockDim` is also of type `dim3` which specifies the size of each block, where we have $blockDim.x = 1024$ threads in each block. `Ns` is of type `size_t` that specifies the number of bytes in shared memory, and `S` is of type `cudaStream_t` that specifies the associated stream. The last two parameters are optional and both of them have a default value of 0.

The kernel function runs on GPU (device) is described as :

```
evolveNetwork(Inputs,States,Connections)<<<gridDim,blockDim>>>;
```

The function type qualifiers play important roles in indicating the calling properties:

- **__device__**: Executed on the device and callable from the device only.

`VectorDotProduct(vectorA[N],vectorB[N]), RandomGenerator()`

- **__global__**: Executed on the device and callable from the host only.
  `NetworkEvolve(Network)`

- **__host__**: Executed on the host and callable from the host only.

For the access of different levels of memory, the variable type qualifiers define the data location:

- **__device__**: In global memory space,is accessible from all the threads within the grid.
  `global state [N], externel input [N], connection matrix [N × N]`

- **__constant__**: In constant memory space,is accessible from all the threads within the grid.

- **__shared__**: In a shared space of a thread block,is accessible from all the threads within the block.
  `reduction cache [threadsPerBlock]`

5.3 Random neuron selection on GPU

The updating mechanism of the neural network is that each neuron will change its state independently, both in sequential and parallel evolution process. But which neuron to choose to update? It's decided by system random variable, because in some time every neuron has a possiblity to update its state. So the system should give a random number index that indicate which neuron will be updated and this function should be built in GPUs.

We note that there is not a in-built random number generator (RNG) in CUDA. One possible solution is to choose the neuron by using the Mersenne Twister(MT) method to generate a neuron index, which is one of the best available pseudorandom number generator in CUDA parallel model.

This RNG is directly integrated on GPU programs and will be lanched when the network starts to evolve. It will generate a serie of index number that tell which block will execute. Thus we have a parallelization between blocks, that is to say, the neurons can also evolve in group at one time. In the implementation of the two levels of parallelism: within Block and within Grid, we should look carefully at the synchronization operation for avoiding an execution failure and loss of effciency.

Before executing the kernel on GPU, we can also generate a serie of random numbers on CPU at first, as a random vector, with a length superior than the potential convergence times. That is to say, this random vector is generated by system function and is long enough to give the index data in the evolution process.

5.4 Implementation of internal parallelization

Initially, a variable of *stride* is set to adjust the pace of the summing calculation, after $stride = \frac{N}{2}$ of threads of adding the products finish executing, the value of *stride* turns to be its half. The addition of two temporary data always occurs

between neurons whose index distance is one *stride*. In each level of reduction, a `syncthreads()` operation is required to assure the current reduction is finished and is ready for the next reduction. Finally, the *stride* turns to be 1 and the last temporary data is the activation value $\sum_{j=1}^{N} W_{ij} \bullet x_j$.

If the size of network N is larger than the number of threads per block *threadsPerBlock*, that one block can not process more than 1024 threads in parallel, we should split the product result vector ($product[N]$) into pieces of vector that has a limited size of 1024.

When the process within the block proceeds to reach a time that there are no more than 32 executing threads, we have only one *warp* left. In fact, the instructions are SIMD synchronous within a warp. Then we don't need to place a `syncthreads` within a warp and we can unroll the last 6 iterations of an inner loop.

```
__global__ void reduction(int *connection,
                          int *state,
                          int *product)
{
  __shared__ int cache[threadsPerBlock];

  int tid=threadIdx.x + blockIdx.x*blockDim.x;
  int gridSize=blockDim.x*gridDim.x;
  int cacheIndex=threadIdx.x;
  int temp=0;

  while(tid<NUM)
        {
          temp=connection[tid]*state[tid];
          tid+=gridSize;
        }

  cache[cacheIndex]=temp;

  __syncthreads();
  int i=blockDim.x/2;
  while(i!=0)
        {
          if(cacheIndex<i)
                cache[cacheIndex]+=cache[cacheIndex+i];
          __syncthreads();
          i/=2;
        }
  if(cacheIndex==0)
        product[blockIdx.x]=cache[cacheIndex];
}
```

5.5 Implementation of the external parallelization

5.5.1 Store Network Data in Blocks

We consider to use *Blocked Compressed Sparse Row Format* to describe our hopfield network. Elements of the network are sorted by row index. The column index is explicitly stored for each element, and so stored an array of indices of the first element in each row. In GPU realizations, the matrix is divided into blocks of size $B_r \times B_c$ in dense forms. One block stores the column coordinate and encode the row coordinates. Then we divide the matrix into strips of S_r rows, in which blocks are stored in coordinate format.

The kernel of computations of Sparce matrix in CSR format is as in the following algorithms: (in which the state vector and connection matrix have already been allocated)

```
__global__ void
hopfield_spmv_csr_kernel(const int groupsize,
const int *ptr,
const int *indices,
const int *connection,
const int *state, int activation)
{
  int neuron_index = blockDim.x*blockIdx.x+threadIdx.x;
  if(neuron_index<groupsize)
  {
    int product=0;
    int start_neuron=ptr[neuron_index];
    int end_neuron=ptr[neuron_index+1];
    for(int i=start_neuron;i<end_neuron;i++)
      product+=connection[i]*state[indices[i]];

    activation[neuron_index]+=product;
  }
}
```

5.5.2 Global state calculation

According to the hierachy structure of CUDA memory, we distribute the network data into different part of GPU loads and assign them to thread blocks. The principle of this partition is avoiding the communications among the computation tasks. One task is exectly executed by one block on GPU, and the threads within a block. So we can process one strip of matrix by one thread block. Threads within one block are divided into groups os size $B_r \times B_c$. Each group reads a matrix block while each thread reads one element of the block. Then we multiply the corresponding element of the vectors. After all the block finish processing, we will get the result vector y.

The size of Block and Grid is defined as follows:

```
#define imin(a,b) (a<b?a:b)
const int threadsPerBlock=256;
```

```
const int blocksPerGrid=
imin(32,(NUM+threadsPerBlock-1)/threadsPerBlock);
```

When the program runs on GPU, there are 256 threads executes in parallel with one block. And the initial number of blocks per grid is `(NUM+threadsPerBlock-1)/threadsPerBlock` (the upper bound of times). If the total number of threads is less than 32×256, we still take 32 as the value of `blocksPerGrid`.

The process is unrolled using a 32-thread warp per matrix row. And when the program is executed in one block, the temporary data is stored for reduction operation in the shared memory. This segment of kernel program is shown as follow:

```
__global__ void
hopfield_kernel_warps(const int groupsize,
                      const int *ptr,
                      const int *indices,
                      const int *connection,
                      const int state,int activation)
{
  __shared__ float sdata[];
  //global thread index
  int tid=blockDim.x*blockIdx.x+threadIdx.x;
  int wid=tid/32; //global warp index
  int lane=tid & (32-1); //thread index within the warp
  //one warp per row
  int row=wid;

  if(row<groupsize){
    int start_neuron=ptr[row];
    int end_neuron=ptr[row+1];

    //compute running sum per thread
    sdata[threadIdx.x]=0;
    for(int i=start_neuron+lane;i<end_neurons;i+=32)
      sdata[threadIdx.x]+=connection[i]*state[indices[i]];

    //parallel reduction in shared memory
    if(lane<16) sdata[threadIdx.x]+=sdata[threadIdx.x+16];
    if(lane<8) sdata[threadIdx.x]+=sdata[threadIdx.x+8];
    if(lane<4) sdata[threadIdx.x]+=sdata[threadIdx.x+4];
    if(lane<2) sdata[threadIdx.x]+=sdata[threadIdx.x+2];
    if(lane<1) sdata[threadIdx.x]+=sdata[threadIdx.x+1];

    //get the result in the first thread
    if(lane==0)
      activation[row]+=sdata[threadIdx.x];
  }
}
```

Another kernel will be used to compute the sum of the two vectors of input

values and activation values, so called **AXPY**, involving computations like $y := \alpha x + y$ with two vectors. Here we simply define the value of α is 1 and use a single AXPY operation based on the BLAS library interface. This computes the increamental activation values of the neural network.

Then we send this activation value vector to a sigmoid function as we mentioned before, handled by each threads within the block by neurons. After a synchronization of assuring all the threads finish their processing, we will get a length n vector of new state $\mathbf{S}$.

6 Result/Performance Analyses

6.1 Energy Function

An energy function is used to evaluate the stability of the neural network for hopfiel model, and it's also associated with the actual optimization problem. We defines the energy function as:

And the energy function can be presented as follows:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} W_{ij}x_i x_j - \sum_{i=1}^{N} I_i x_i \tag{9}$$

where W_{ij} denotes the weights of connections between neurons i and j x_i denotes the states of the neuron i (0 or 1) I_i denotes the external input to the neuron i

As the network evolves, the energy function decrease. At last, the energy function gets its minimum and the whole network becomes stable. We have different number of updating times with different size of networks and different expected active neuron numbers.Generally, the bigger size a neural network has, a more evolution times needed to get its stable state.

The two figures below show a 1152 neurons network's energy function value in its evolution process. We can see that in the beginning, the speed of decreasing is larger than the last period of time of evolution. The trend of the energy function value can be regarded as the changement of neuron state values in the direction of an expected global stable state.

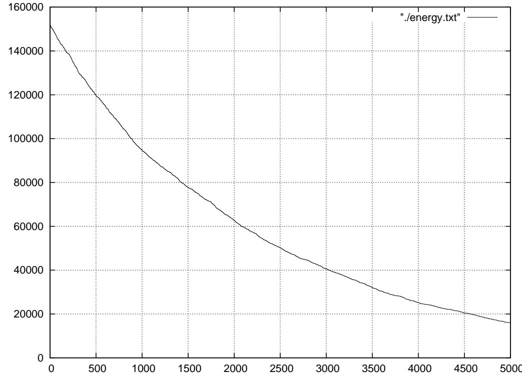
6.2 Complexity Analysis

The optimizations are based on some levels of the performance. First of all we should evaluate to performance, which contains some properties: One is *latency*, the time taken for the operations; *bandwidth*, the rate that operations are performed; and *cost*, the impact these operations have on the execution time of the program.

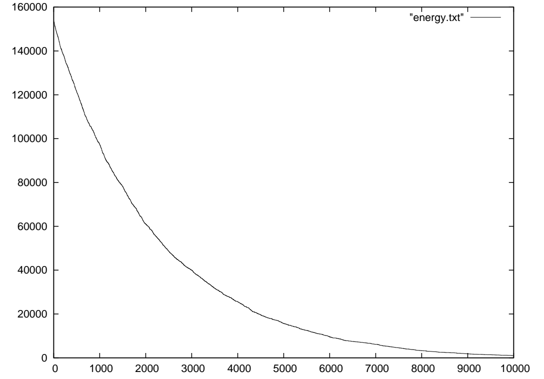
Data Transfer Time

The time for a data transfer operation is generally described by a linear model:

$$TransferTime(n) = T_0 + \frac{n}{B} \tag{10}$$



(a) The decreasing Energy Function



(b) The decreasing Energy Function (10000 times evolution)

where n is the amount of data, B is the transfer rate of the component moving the data in compatible units, and T_0 is the constant of the startup cost. This is a very convenient model used to describe a diverse collection of operations, such as messages deliveries, memory accesses, but transactions, and vector operations. For memory operation, it is the access time. For any sort of pipelined operation, such as vector operations, it is the time to fill the pipeline.

When using this model, the bandwidth of a data transfer operation depends on the transfer size. As the transfer size increases, it approaches the asymptotic rate of B, referred as r_∞. So the startup cost decide how quickly it will approach this rate. We can easily get the size at which half of the peak bandwidth :

$$n_{\frac{1}{2}} = T_0 B \tag{11}$$

This value of the data size is called the *half power point*. However ,this linear model does not give any indication of when the next operation can be initiated, nor does it indicate whether other useful work can be performed during the transfer. That depends on the way that the data are transferred.

Calculation Complexity

In the first level of parallelization of reduction for one neuron's update, We can check that beside the warp unroll, the calculating time will be $O(\log_2(NUM))$, where NUM is the neuron number of the whole hopfield network. Considering the last inner loop unroll of 32 threads, the calculation complexity will be $O(\log_2(NUM)/32 + 5)$, where unroll the warp threads needs 5 steps.

In the second level of group parallelization for the whole network update, the calculation complexity depends on the degree of sparse (from structed or unstructed connection matrice and state vectors), because they are related to the compressed rate.

If the whole connection matrix of one hopfield is of size $M \times N$, where M is the number of rows (tasks number), and N means the number of columns (cycle number), with every row and every column being applied a $k - out - of - N$ rule. So there are totally $M + N$ rules present in the network simulation. The tasks connections number is $M * N * (N - 1)/2$, and the cycles connections number is

$N * M * (M-1)/2$, then the total connections within the network is $M * N * (M + N - 2)/2$, here we get a filling rate of the compressed matrix :

$$DenseRate(Matrix(M \times N) = \frac{M+N}{2} - 1 \tag{12}$$

The length of `connection` is relative with this dense rate. The length of `indice` and `ptr` are relative with the neuron number $M * N$. So the data transfer complexity of the second parallelization is $O(M * N * \frac{M+N}{2} + 1)$, And the second kernel computing complexity is the group length $M * N$ (when we pick up all the neurons as one group).

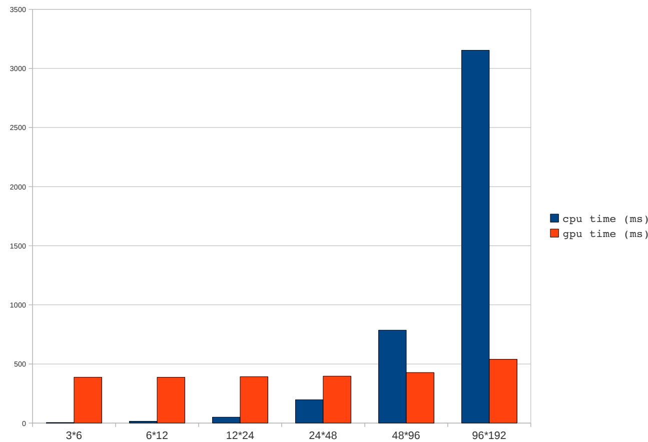
6.3 Comsuming Time Comparision

The CUDA provides an *Event API* to evaluate the execution time of programs on GPU. This API contains the methods of creating and destroying the events, so we can obtain the consuming time by calculating the timestamp difference of the event cycles. The time is measured in millisecond.

As in the reduction of computing one neuron's state, the time consuming highly depends on the problem size: we take an 3×6 hopfield network as an initial size network, and enlarge the size (task number $\times$ cycle number) by doubling both the tasks and cycles. The following figure shows the comparision of time in CPU and GPU by updating one neuron's state.

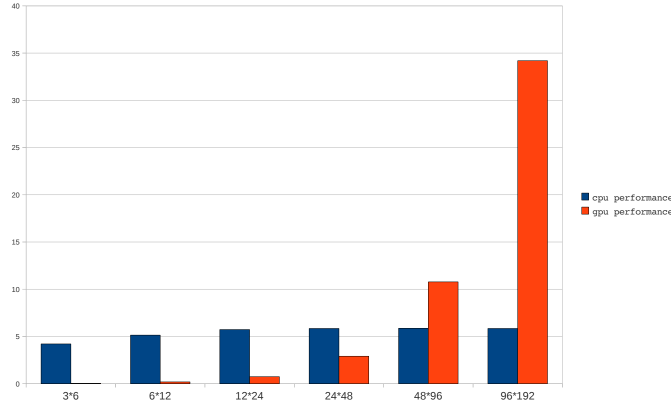
The evaluation of time indicates that GPU have succeeded in acceleration with a large size of hopfield network. The CPU program's running time increase with the size of network, while GPU program use almost the same time in whatever size of network. But in this step, we should note that we only update one neuron's state, we can parallize the reduction processing threads as many as possible, as there are plenty of threads capacity on GPU (one grid can support 65536*1024 threads).

Figure 9: CPU/GPU Time Consuming of Different Size Networks



In Figure above, the performance values are obtained by calculating the rates of number of neurons in a network per time in millisecond. In a network of 24×48

Figure 10: CPU/GPU Performance: Network Size/Execution Time



neurons, the GPU programe has not shown a better performance, and if the size of network is superior than 48×96 neurons, the program on GPU has shown an advantage in calculating global state than the program on CPU. The results tells us that for small size of network, we can simulate firstly on CPU, and if the size of the network is bigger than a certain number, for instance, 2000 neurons, (this number can vary from the expected active neurons in a model), it is better to simulate the artificial neural network on GPU.

And for the number of neurons of evolution to a global stable state, the program on CPU and GPU show a same result. since the acceration of the calculation on GPU does not change an individuel neuron's update mechanism.

7 Conclusion

In this work, we studied the hopfield neural network mechanism and explained the data structure arrangement on GPU architecture. The hopfield structure is proved to be an efficient tool to get a optimization solution. We made preparation network data through an hopfield network generator on CPU, according to the rules applied on the subset of the whole network.

And we explored several parallel techniques on CUDA architecture for the hopfield neural network. Once the network data is prepared, we sent two networks in the same size separately to CPU and GPU simulator. There are two facts we need to observe: One is the network state in the evolution process, another is the energy function of hopfield networks. In most cases both the network will get a stable state that the neuron value will not change any more. The different point of the simulation process is the performance run on different architectures. The sequantial program runs on CPU proceed this evolution process sequantially in every aspect, while the parallel program runs on GPU partly parallel in the calculating kernels.

The result shows that the parallelization of reduction of updating one neuron of the hopfield network is worthy since it can get a much better performance compared with the traditional sequantial program. And along with the size of the

network, the accerating level increase when there are more neurons. The energy function value dicreases when the network evolves forward. After a finite period of time, the network state will not change and the energy function gets its minimum.

Another techinique for the parallelisation is the parallelization among the neuron states updates. In this step, we use a matrix-vector multiplication as the important kernel to reduce the network storage and the calculating part. The evolution process is represented to be a series of multiplication sigmoid function operations. After this, we found that this can efficiently enhance the performance of the evolution simulation. The CUDA also provides the libraries for this kind of algebra calculations.

The final analysis shows the speeding-ups between CPU and GPU programs, the acceleration sources, and different levels of accelarations in different sizes of network. The highly parallel nature of the hopfield network is proved in these implementations. And thus we can use this parallel neural network on GPU to solve some optimization problem speficied in varies areas such as heteogeneous computing resource assignments.

7.1 Future work

We aim at a better realization of neural network for optimization problems. Till now we found some useful clues to implement a typical hopfield neural network, but in future, there are still some topics need to be studied:

1 Other artificial neural network simulations for solving time scheduling optimization problems.

2 Different hopfield neural network structures and different mechanisms for specified optimization problems.

3 Higher speeding-ups for the implementation of hopfield neural network to get a peak bandwidth usage of the graphics card.

4 Realizations of ANN for other platforms, such as FPGA, or other parallel archtectures.

8 Acknowledgement

I am grateful to my superviser Daniel Chillet, for his advices of research and guides of the implementations, and correctness of this report, and thank Antoine Eiche, for his introductions to relative knowledge. And I would like to thank Cairn team for providing NVIDIA graphics card for the implementations and performance analysis, and I'm also thankful to Arnaud Tissant for helping me set up this hardware in his server and to other lab staffs for installing the basic software enviroments.

References

[1] D.Chillet, A.Eiche, S.Pillement, O.Sentieys, "Real-time scheduling on heterogeneous system-on-chip architectures using an optimised artificial neural network", *Journal of Systems Architecture* 57, 340-353, (2011).

[2] J.J.Hopfield and D.W.Tank, "'Neural' Computation of Decisions in Optimization Problem", *Biol.Cybern.*52,141-152(1985).

[3] J.J.Hopfield, Neural networks and physical systems with emergent collective computational abilities, *Proc. Natl. Acad. Sci.USA* Vol.79, pp. 2554-2558, April 1982.

[4] B.M.Forrest, D.Roweth, N.Stroud, D.J.Wallace and G.V.Wilson. "Implementing Neural Network Models on Parallel Computers". *The Computer Journal*, VOL.30, NO.5, 1987.

[5] T.Nordstrom and B.Svensson, "Using and Designing Mssively Parallel Computers for Artificial Neural Networks", *Journal of Parallel and Distributed Computing*, Vol.14, No.3, 1992, pp.260-285.

[6] F.Valafar, O.K.Ersoy. "A Parallel Implementation of Backprogagation Neural Network On MASPAR MP-1", *ECE Tchinical Reports*. Paper 223, 1993.

[7] Nikola B. Serbedzija, "Simulating Artificial Neural Networks on Parallel Architectures", *Computer*,Vol.29, No.3, 1996, 56-63.

[8] Udo Seiffert, "Artificial Neural Networks on Massively Parallel Computer Hardware", *ESANN'2002 proceedings-European Symposium on Artificial Neural Networks Bruges(Belgium)*, April 2002.

[9] Mario Martinez-Zarzuela, Francisco Javier Diaz Pernas, Jose Fernando Diez Higuera, and Miriam Anton Rodriguez. "Fuzzy ART Neural Network Parallel Computing on the GPU". *IWANN 2007*, LNCS 4507, pp.463-470.

[10] J.M., Dutt, N., Krichmar, J.L., Nicolau, A., & Veidenbaum, A, "Efficient simulation of large-scale spiking neural networks using CUDA graphics processors". *International conference on neural networks* ,2009.

[11] R.Dolan, G.Desouza. "GPU-Based Simulation of Cellular Neural Networks for Image Processing". *Proceedings of International Joint Conference on Neural Networks* June 14-19,2009.

[12] A. Guzhva, S.Dolenko, and I. Persiantsev. "Multifold Acceleration of Neural Network Computations Using GPU". *ICANN 2009*, Part I, LNCS 5768, pp.373-380,2009.

[13] N.Lopes and B.Ribeiro, "GPU Implementation of the Multiple Back-Propagation Algorithms", *IDEAL 2009*, LNCS 5788, pp.449-456, 2009.

[14] Daniel L. Ly, Volodymyr Paprotski, Danny Yen. "Neural Networks on GPUs: Restricted Boltzmann Machines". *University of Toronto*, gpucomputing.net, 2010.

[15] Nathan Bell and Michael Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA", *NVIDIA Technical Report* NVR-2008-004, Dec.2008.

[16] Y.Dotsenko, N.K.Govindaraju, P.-P.Sloan, C.Boyd, and J.Manferdelli. "Fast Scan Algorihms On Graphics Processors", *ICS '08 : Proceedings of the 22nd annual International Conference on Supercomputing*, Page 205-213, 2008.

[17] J.Bolz, I.Farmer, E.Grinspun, P.schroder, "Sparse Matrix Solvers on the GPU : Conjugate Gradients and Multigrid", *ACM TRANSACTIONS OF GRAPHICS*, 2003.

[18] A.Monakov and A.Avetisyan, "Implementing Blocked Sparse Matrix-Vector Multiplication on NVDIA GPUs", *SAMOS 2009*, LNCS 5657, pp.289-297, 2009.

[19] A.Monakov, A.Lokhmotov, A.Avetisyan, "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures", *HiPEAC 2010*, LNCS 5952, pp.111-125, 2010.

[20] Sander Marcel Bohte. Spiking Neural Networks. *Thesis Universiteit Leiden.* ISBN 90-6734-167-3, 2003.

[21] David E. Culler, Jaswinder Pal Singh. Parallel computer architecture : a hardware/software approach. Morgan Kaufmann Publishers,Inc. 1999.

[22] NVIDIA CUDA compute unified device architecture, the programming guide.

[23] CS 112 Lab 10: Shared-Memory Parallelism Using OpenMP, Calvin College.

[24] Dominik Scherer, Hannes Schulz, and Sven Behnke. Accelerating Large-Scale Convolutional Neural Networks with Parallel Graphics Multiprocessors. *ICANN 2010* ,Part III, LNCS 6354, pp. 82-91

[25] Ben Krose, Patrick van der Smagt. An introduction to Neural Networks,Eighth edition. pp. 50-52, November 1996.

[26] I. BENKERMI. PhD Thesis : Modele et algorithme d'ordonnancement pour architectures reconfigurables dynamiquement,University of Rennes 1.

[27] Gene A.Tagliarini, J. Fury Christ and Edward W.Page. Optimization Using Neural Network, *IEEE TRANSACTIONS ON COMPUTERS, VOL.40 NO.12* DECEMBER 1991.

[28] Honghoon Jang, Anjin Park, Keechul Jung. Neural Network Implementation using CUDA and OpenMP.

[29] Kyong-Su Oh,Keechul Jung,GPU implementation of neural networks. *Pattern Recognition Society* 2004.

[30] D. Chillet, S. Pillement, and O. Sentieys. RANN: A Reconfigurable Artificial Neural Network Model for Task Scheduling on Reconfigurable System-on-Chip. *Algorithm-Architecture Matching for Signal and Image Processing, Springer*,2010.

[31] D. Chillet, S. Pillement, and O. Sentieys. Ordonnancement de taches par rseaux de neurones pour architectures de soc htrognes. *Traitement du signal*, 26(1):p77-89, 2009.

[32] Janardan Misra, Indranil Saha, Artificial neuralnetworksinhardware: A survey of two decades of progress, *Neurocomputing*74 (2010)239-255.

[33] Christopher Edward Davis, Master degree thesis: Graphics Processing Unit Computation of Neural Networks, The University of New Mexico, 2005

[34] Zheng He and Koichi Harada,Balance Algorithm for Point-Feature Label Placement Problem, *ICANN 2005, LNCS 3696*, pp.179-184, 2005.

[35] JON CHRISTENSEN, An Empirical Study of Algorithms for Point-Feature Label Placement, *ACM Transactions on Graphics*,Vol.14, No,3, July, 1995, pp.203-232.