A Massive Parallel Cellular GPU Implementation of Neural Network to Large Scale Euclidean TSP

Hongjian Wang, Naiyu Zhang, and Jean-Charles Créput

IRTES-SeT, Université de Technologie de Belfort-Montbéliard, 90010 Belfort, France
{hongjian.wang,naiyu.zhang,jean-charles.creput}@utbm.fr

Abstract. This paper proposes a parallel model of the self-organizing map (SOM) neural network applied to the Euclidean traveling salesman problem (TSP) and intended for implementation on the graphics processing unit (GPU) platform. The plane is partitioned into an appropriate number of cellular units, that are each responsible of a certain part of the data and network. The advantage of the parallel algorithm is that it is decentralized and based on data decomposition, rather than based on data duplication, or mixed sequential/parallel solving, as often with GPU implementation of optimization metaheuristics. The processing units and the required memory are with linear increasing relationship to the problem size, which makes the model able to deal with very large scale problems in a massively parallel way. The approach is applied to Euclidean TSPLIB problems and National TSPs with up to 33708 cities on both GPU and CPU, and these two types of implementation are compared and discussed.

Keywords: Neural network, Self-organizing map, Euclidean traveling salesman problem, Parallel cellular model, Graphics processing unit.

1 Introduction

A classical and widely studied combinatorial optimization problem is the Euclidean traveling salesman problem (TSP). The problem is NP-complete [1]. The self-organizing map (SOM), originally proposed by Kohonen [2], is a particular kind of artificial neural network (ANN) model. When applied in the plane, SOM is a visual pattern that adapts and modifies its shape according to some underlying distribution. The SOM has been applied to the TSP since a long time [3–5] and it was shown that this artificial neural network model is promising to tackle large size instances since it uses $O(N)$ memory size, where N is the instance size, i.e. the number of cities. In the light of its natural parallelism, we propose a parallel cellular-based SOM model to solve the Euclidean TSP and implement it on the graphics processing units (GPU) platform. From our knowledge, we did not find such type of SOM application to the Euclidean plane and implementation on GPU in the literature.

In recent years, the graphic hardware performance is improved rapidly and GPU vendors make it easier and easier for developers to harness the computation power of GPU. Some methods for computing SOM on GPU have been proposed [6,7]. All these methods accelerate SOM process by parallelizing the inner steps in each basic iteration, of which mainly focus on two aspects as follows, firstly, to find out the winner neuron in parallel, secondly, to move the winner neuron and its neighbors in parallel. In our model, we use each parallel processing unit to do SOM iterations independently in parallel to a constant part of the data, instead of using many parallel processing units to cooperatively accelerate a sequential SOM procedure iteration by iteration. The processing units and the required memory are with linear increasing relationship to the problem size, which makes the model able to deal with very large scale problems in a massively parallel way. The theoretical computation time of our model is based on a parallel execution of many spiral search of closest points, each one having a time complexity in $O(1)$ in average when dealing with a uniform, or at most a bounded data distribution [8]. Then, one of the main interests of the proposed approach is to allow the execution of approximately N spiral searches in parallel, where N is the problem size. Thus, what would be done in $O(N)$ computation time in average for a sequential spiral search able to find N closest points, is performed in constant time $O(1)$ theoretical complexity for a parallel algorithm in the average case, for bounded distributions. This is what we intend by "massive parallelism", the theoretical possibility to reduce average computation time by factor N, when solving a Euclidean NP-hard optimization problem.

The rest of this paper is organized as follows. We briefly introduce the Euclidean traveling salesman problem and the self-organizing map in Section 2. After that, we present our parallel cellular-based model in Section 3 and give the detailed GPU implementation in Section 4. Our experimental analysis on both small and large scale problems is outlined in Section 5, before we summarize our work and conclude with suggestions for future study.

2 Background

2.1 Traveling Salesman Problem

The travelling salesman problem (TSP) can be simply defined as a complete weighted graph $G = (V, E, d)$ where $V = \{1, 2, \cdots, n\}$ is a set of vertices (cities), $E = \{(i, j)|(i, j) \in V \times V\}$ is a set of edges, and d is a function assigning a weight (distance) d_{ij} to every edge (i, j). The objective is to find a minimum weight cycle in G which visits each vertex exactly once. The Euclidean TSP, or planar TSP, is the TSP with the distance being the ordinary Euclidean distance. It consists, correspondingly, of finding the shortest tour that visits N cities where the cities are points in the plane and where the distance between cities is given by the Euclidean metric.

2.2 The Kohonen's Self-organizing Map

The standard self-organizing map [2] is a non directed graph $G = (V, E)$, called the network, where each vertex $v \in V$ is a neuron having a synaptic weight vector

$w_v = (x, y) \in \Re^2$, where $\Re^2$ is the two-dimensional Euclidean space. Synaptic weight vector corresponds to the vertex location in the plane. The set of neurons N is provided with the d_G induced canonical metric $d_G(v, v') = 1$ if and only if $(v, v') \in E$, and with the usual Euclidean distance $d(v, v')$.

In the training procedure, a fixed amount of T_{max} iterations are applied to a graph network (a ring network in TSP applications), the vertex coordinates of which being randomly initialized into an area delimiting the data set. Here, the data set is the set of cities. Each iteration follows three basic steps. At each iteration t, a point $p(t) \in \Re^2$ is randomly extracted from the data set (extraction step). Then, a competition between neurons against the input point $p(t)$ is performed to select the winner neuron n^* (competition step). Usually, it is the nearest neuron to $p(t)$. Finally, the learning law (triggering step) presented in Equation 1 is applied to n^* and to the neurons within a finite neighborhood of n^* of radius σ_t, in the sense of the topological distance d_G, using learning rate $\alpha(t)$ and function profile h_t. The function profile is given by the Gaussian in Equation 2. Here, learning rate $\alpha(t)$ and radius σ_t are geometric decreasing functions of time. To perform a decreasing run within T_{max} iterations, in each iteration t, coeffients $\alpha(t)$ and σ_t are multiplied by $exp(ln(\chi_{final}/\chi_{init})/T_{max})$ with respectively $\chi = \alpha$ and $\chi = \sigma$, χ_{init} and χ_{final} being respectively the values in starting and final iteration. Examples of a basic iteration with different learning rates and neighborhood sizes are shown in Fig.1.

$$w_n(t + 1) = w_n(t) + \alpha(t) \times h_t(n^*, n) \times (p(t) - w_n(t)) \; . \tag{1}$$

$$h_t(n^*, n) = exp(-d_G(n^*, n)^2/\sigma_t^2) \; . \tag{2}$$

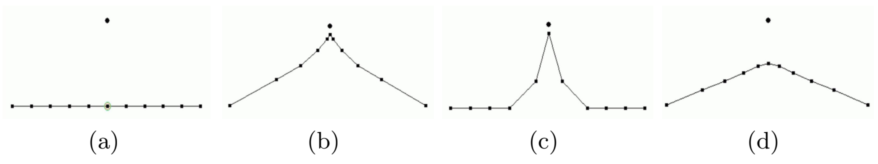

Fig. 1. A single SOM iteration with learning rate α and radius σ. (a) Initial configuration. (b) $\alpha = 0.9, \sigma = 4$. (c) $\alpha = 0.9, \sigma = 1$. (d) $\alpha = 0.5, \sigma = 4$.

3 Parallel Cellular Model

3.1 Cell Partition

It is intuitive that TSP and SOM can be connected by sharing the same Euclidean space. As a result, the input data distribution of SOM is the set of cities of TSP. The application consists of applying iterations to a ring structure with a fixed number of vertices (neurons) M. Specifically, M is set to $2N$, N being the number of cities. After training procedure, the ring transforms into a possible solution for the TSP along which a determined tour of cities can be obtained.

Fig.2 illustrates an example of training procedure on the $pr124$ instance from TSPLIB [9] at different steps of a long simulation run. Black dots are the city points of TSP. Small red circles and the black lines that connect them constitute the ring network of neurons. Execution starts with solutions having randomly generated neuron coordinates into a rectangular area containing cities, as shown in Fig.2(a). After 100 iterations, the ring network as shown in Fig.2(b) has roughly deployed towards cities. After 10000 iterations, the ring network has almost completely been mapped onto cities, as shown in Fig.2(c).

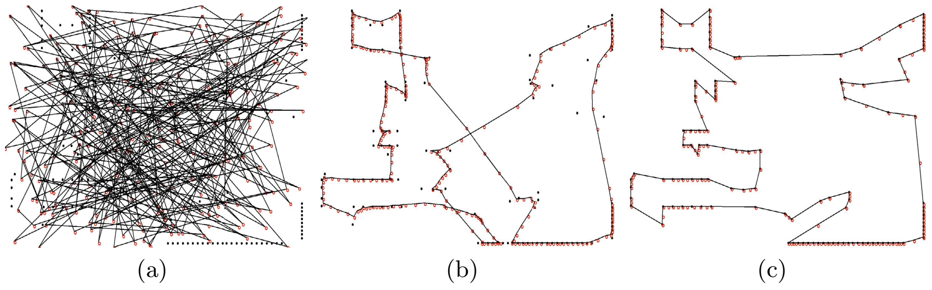

Fig. 2. Different steps of training procedure on the $pr124$ instance

In order to implement the parallel level at which parallel execution will take place, we introduce a supplementary level of decomposition of the ring network plane and input data. We uniformly partition the Euclidean space into small cells with the same size that constitute a two-dimensional cellular matrix. The scale of cell partition is decided by the number of cities. Specifically, the size of each dimension is set to $\lceil\sqrt{N \times \lambda}\rceil$, where N is the number of cities and parameter λ, which we set to 1.1 in the later experiments, is used to adjustment. The three main data structures of the parallel model are illustrated in Fig.3. This intermediate cellular matrix is in linear relationship to the input size. Its role will be to memorize the ring network in a distributed fashion and authorize many parallel closest neuron searches in the plane by a spiral search algorithm [8]. Each cell is then viewed as a basic training unit and will be executed in parallel. Thus, in each parallel iteration we conduct a number of parallel training procedures instead of carrying out one only. Each cell corresponds to a processing unit or GPU thread.

Each processing unit, that corresponds to a cell, will have to perform the different steps of the sequential SOM iteration in parallel. A problem that arises is then to allow many data points extracted at first step by the processing units, at a given parallel iteration, to reflect the input data density distribution. As a solution to this problem, we propose a particular cell activation formula stated in Equation 3 to choose those cells that will execute or not the iteration. Here, p_i is the probability that the cell i will be activated, q_i is the number of cities in the cell i, and num is the number of cells. The empirical preset parameter δ is used to adjust the degree of activity of cells/processing units. As a result, the more cities a cell contains, the higher is the probability this cell to be activated to carry out

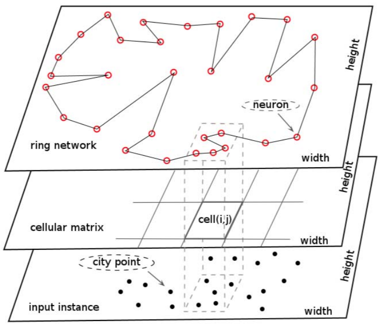

Fig. 3. Parallel cellular model

the SOM execution at each parallel iteration. In this way, the cell activation depends on a random choice based on the input data density distribution.

$$p_i = \frac{q_i}{\max\{q_1, q_2, \ldots, q_{num}\}} \times \delta \ . \tag{3}$$

3.2 Cellular-Based Parallel SOM

Based on the cell partition, the parallelized SOM training procedure carries out four parallel steps: cell activation step, extraction step, competition step and triggering step. Then, this parallel process is repeated T_{max} times. Note that T_{max} now represents the number of parallel iterations.

For each processing unit which is associated to a single cell, a cell is activated or not depending on the activation probability. If the cell is activated, the processing unit will continue to perform the next three parallel operations, otherwise it does nothing and directly skips to the end of the current iteration.

In the parallel extraction step, the processing unit randomly chooses a city from its own cell, unlikely the original sequential SOM which randomly extracts a point from the entire input data set.

In the competition step, the processing unit carries out a spiral search [8] based on the cell partition model to find the nearest neuron to the extracted city point. The cell in which this point lies will be searched first. If this cell is empty of neuron (ring node), then the cells surrounding it are searched one by one in a spiral-like pattern until a neuron is found. Once one neuron is found, it is guaranteed that only the cells that intersect a particular circle, which is centered at the extracted point and with the radius equal to the distance between the first found neuron and the extracted point, have to be checked before finishing searching. When performed on a uniform data distribution, or bounded density distribution [8], a single spiral search process takes $O(1)$ computation time according to the instance size. Then, one of the main interests of the method would be to perform $O(N)$ (the cell number) spiral searches in parallel, then in a theoretical constant time $O(1)$ for bounded density distribution, if $O(N)$ physical cores were available. This is what we call "massive parallelism".

In the triggering step, each processing unit moves its closest neuron and several neurons within a finite neighborhood toward the extracted city, according to the rule of Equation 1. All the processing units share one unique ring network of neurons in the Euclidean space. The coordinates of neurons are therefore stored into a shared buffer which is simultaneously accessed by all the parallel processing units.

After all the parallel processing units have finished their jobs in one single iteration, the learning rate α and radius σ are decreased, getting ready for the next parallel iteration.

To establish our cellular-based parallel SOM model, the scale of cell partition is $\lceil\sqrt{N \times \lambda}\rceil^2$, with N the number of cities. Hence, the number of parallel processors needed is $O(N)$. Since only one ring network is stored in memory, the memory complexity is also $O(N)$. Moreover, the parallel spiral search by every processor takes constant time $O(1)$ theoretically for bounded density distribution. For T_{max} parallel iterations, the maximum number of single SOM iterations is $T_{max} \times \lceil\sqrt{N \times \lambda}\rceil^2$, which corresponds to the extreme case where all the processing units are activated at the same time.

4 GPU Implementation

4.1 Platform Background

We use GPU to implement our parallel model with the compute unified device architecture (CUDA) programming interface. In the CUDA programming model, the GPU works as a SIMT co-processor of a conventional CPU. It is based on the concept of kernels which are functions written in C executed in parallel by a given number of CUDA threads. These threads will be launched onto GPU's streaming multi-processors and executed in parallel [10]. Hence, we apply CUDA threads as the parallel processing units in our model.

All CUDA threads are organized into a two level concepts: CUDA grid and CUDA block. A kernel has one grid which contains multiple blocks. Every block is formed of multiple threads. The dimension of grid and block can be one-dimension, two-dimension or three-dimension. Each thread has a *threadId* and a *blockId* which are built-in variables defined by the CUDA runtime to help user locate the thread's position in its block as well as its block's position in the grid [10, 11].

4.2 CUDA Code Design

In the CUDA program flow in Algorithm 1, Lines 2, 4, 7, 8, 11, and 13 are implemented with CUDA kernel functions that will be executed by GPU threads in parallel. The kernel function in Line 2 is used for calculating each cell's density value, i.e. the number of city points in each cell. After all the cells' density values are obtained, the maximum one is found. This last work in Line 3 is done on CPU since it is done only one time and does not directly concern the main behavior.

Note that computing a maximum value is a trivial job even when done on GPU. Then, the cells' activation probabilities are computed according to the activation formula of equation 3 by the kernel function of Line 4. In each iteration of the program, each cell needs two random numbers: one is used for cell activation and the other is used to extract input point in the activated cell. With respect to the large scale input instances with huge cellular matrix and numerous iterations, the random numbers generated via kernel functions shown in Line 7 and Line 8 are stored in a fixed size area due to the limited GPU global memory. Every time these random numbers are used out, a new set of random numbers are generated at the beginning of the next iteration, depending on a constant rate factor called *memory_reuse_set_rate*. The random number generators we use in Line 7 and Line 8 are from Nvidia CURAND library [10]. Line 10 and Line 11 concern the cell refreshing. Each cell has data structures where to deposit information of the number and indexes, in the neuron ring, of the neurons it contains. This information may change during each iteration, but it appears that it can be sufficient to make the refreshing based on a refresh rate coefficient called *cell_refresh_rate*. The cell contains are refreshed via kernel function in Line 11. Note that neurons' locations are moved in the plane at each single iteration, whereas the indexes in cells are refreshed based on a lower rate. Then, the parallel SOM process takes place with kernel function of Line 13 (see below). After the parallel SOM process is done, the SOM parameters will be modified getting prepared to do the next iteration.

Algorithm 1. CUDA program flow

```
 1: Initialize data;
 2: Calculate cells' density values;
 3: Find the max cell density value;
 4: Calculate cells' activated probabilities;
 5: for $ite \leftarrow 0$ to $max\_ite$ do
 6:    if $ite \% memory\_reuse\_set\_rate == 0$ then
 7:       Set seeds for random number generators;
 8:       Generate random numbers;
 9:    end if
10:    if $ite == 0 \| ite \% cell\_refresh\_rate == 0$ then
11:       Refresh cells;
12:    end if
13:    Parallel SOM process;
14:    Modify SOM parameters;
15: end for
16: Save results;
```

Overall, the host code (CPU side) of the program is mainly used for flow control and the entire GPU threads synchronization by sequentially calling separate kernel functions. For all the kernel functions, one thread handles one cell and the number of threads launched by each kernel is no less than the number of cells.

The parallel SOM kernel function of Line 13 of Algorithm 1 is further illustrated by Algorithm 2. Firstly, it locates the cell's position by its *threadId* and *blockId*. Then, the thread checks if the cell is activated or not, by comparing the cell's activated probability to a random number with value between 0 and 1. If the cell is activated, the thread randomly selects a city point in the cell by using a second random number with value between 0 and the cell's density value (number of cities in that cell). After that, the thread performs a spiral search within a certain range on the grid for finding the closest neuron to the selected city point. The maximum number of cells a thread has to search equals $(range \times 2 + 1)^2$. After finding the winner neuron, the thread carries out learning process via modifying positions of the winner neuron and its neighbors. All the neurons' locations are stored in GPU global memory which is accessible to all the threads. Like all the multi-threaded applications, different threads may try to modify one same neuron's location at the same time, which causes race conditions. In order to guarantee a coherent memory update, we use the CUDA atomic function which performs a read-modify-write atomic operation without interference from any other threads [10].

Algorithm 2. GPU parallel SOM kernel flow

```
1: Locate cell position associated to current thread
2: Check if the cell is activated;
3: if the cell is activated then
4:     Randomly select a city point in the cell;
5:     Perform a spiral search within a certain range;
6:     Modify positions of the winner neuron and its neighbors;
7: end if
```

5 Experimental Analysis

5.1 Warp Divergence Analysis

In the CUDA architecture, a warp refers to a collection of 32 threads that are "woven together" and get executed in lockstep [11]. At every line in kernel function, each thread in a warp executes the same instruction on different data. When some of the threads in a warp need to execute an instruction while others in the same warp do not, this situation is known as warp divergence or thread divergence. Under normal circumstances, divergent branches simply result in performance degradation with some threads remaining idle while the other threads actually execute the instructions in the branch. The execution of threads in a warp with divergent branches are therefore carried out sequentially, resulting in performance degradation.

According to our trial tests, the most time consuming kernel function is the parallel SOM kernel. One of the reasons is that there exists warp divergence when this kernel is being executed because it has an unpredictable spiral search process in it. The spiral search is carried out in each cell of the search range, one by one, and it stops immediately when the thread finds a nearest neuron. As a result, different threads may stop at different times. Also, the more cells each thread is going to search in, the severer this problem gets. Hence, different search range settings have different influences on warp divergence. When the block size is set to 256 which is usually enough to fulfill the streaming multi-processor with adequate warps for the GPU device with CUDA capability 2.0, the highest branch efficiency (ratio of non-divergent branches to total branches [10]) of all executions with search range set to 1, 2, and 3 is 90.1%, 87.2%, and 85.9% respectively as collected by NVIDIA Visual Profiler. In theory, the less threads are put in one block, the less warp divergence occurrences will appear. Extremely, if there is only one thread in a block, then there will definitely not be warp divergence. However, the decrease of threads in each block implies the decrease of the CUDA cores usage associated to each streaming multi-processor. In order to analyze the tradeoff between performance and number of threads in a block, we have tested a set of different combinations of grid size and block size for the parallel SOM kernel. The configuration which makes the kernel run fastest is with block size of 8 with highest branch efficiency of 96.9%.

5.2 Comparative Results on GPU and CPU

During our experimental study, we have used the following platforms:

- *On the CPU side:* An Intel(R) Core(TM) 2 Duo CPU E8400 processor running at 2.67 GHz and endowed with four cores and 4 Gbytes memory. It is worth noting that only one single core executes the SOM process in our implementation.
- *On the GPU side:* A Nvidia GeForce GTX 570 Fermi graphics card endowed with 480 CUDA cores (15 streaming multi-processors with 32 CUDA cores each) and 1280 Mbytes memory.

Table 1. Experiment parameters

	α_{init}	α_{final}	σ_{init}	σ_{final}	iterations	δ	CRR[a]	SSR[b]	MRSR[c]
GPU[1]	1	0.01	12	1	100000	1	1	1	1000
CPU[1]	1	0.01	12	1	$100000 \times N$	–	100	1	–
GPU[2]	1	0.01	100	1	100000	1	1	3	1000
CPU[2]	1	0.01	100	1	$10000 \times N$	–	100	3	–

[1] Tests of small size instances. [2] Tests of large size instances.
[a] Cell refresh rate. [b] Spiral search range. [c] Memory reuse set rate.

We have done our tests with two groups of instances from either National TSPs (http://www.math.uwaterloo.ca/tsp/world/countries.html) and TSPLIB database [9]. One group consists of four small size instances from 124 cities to 980 cities, while the other consists of four large size instances from 8246 cities to 33708 cities. The parameter settings for the two groups are shown in Table

1. As discussed in Section 3.2, $T_{max} \times \lceil \sqrt{N \times \lambda} \rceil^2$ parallel SOM operations will be carried out as an extreme case by the GPU SOM program, with N the input instance size and λ set to 1.1. For the tests of small size instances, we set the total number of sequential iterations of the CPU version to $T_{max} \times N$, in order to make the total SOM operations approximately similar between GPU version and CPU version, and to reach similar quality results. Whereas for the tests with large size instances, we set it to $T_{max} \times N/10$, also to achieve similar quality results and because GPU operations depend on the cell activation probabilities and may be less than N at each GPU parallel iteration.

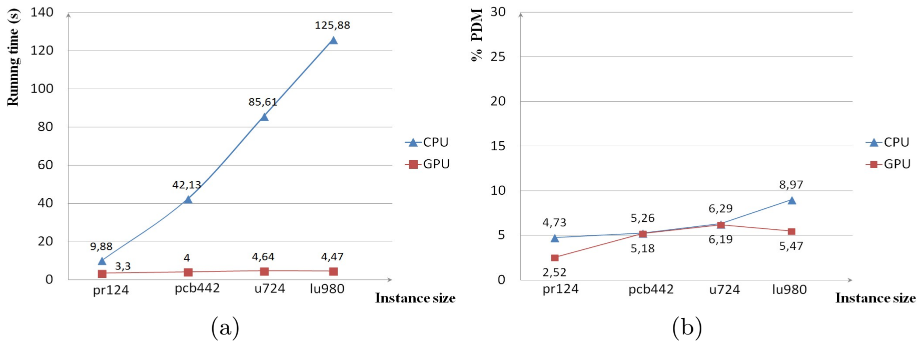

Fig. 4. Test results of small size instances

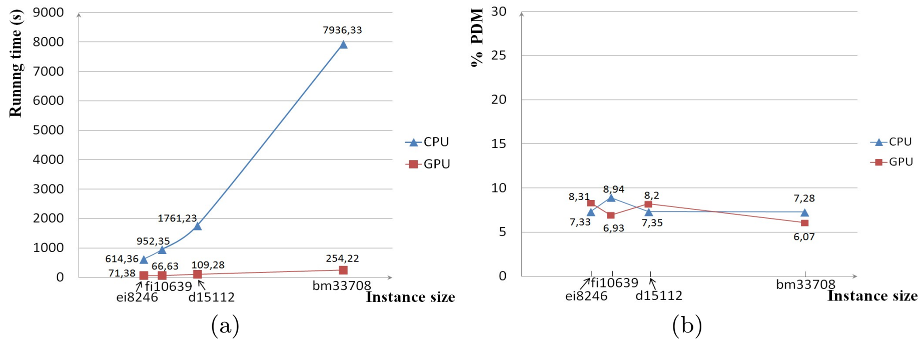

Fig. 5. Test results of large size instances

All the tests are done on a basis of 10 runs per instance. For each test case is reported the percentage deviation, called "%PDM", to the optimum tour length of the mean solution value obtained, i.e. %PDM = $(mean\ \ length - optimum) \times 100/optimum$. As well, is reported the percentage deviation from the optimum of the best solution value found over 10 runs, called "%PDB". Finally, is also reported the average computation time per run in seconds, called "Sec".

Table 2. Test results of small size instances

		GPU			CPU		
Problem	Optimal	%PDM	%PDB	Sec	%PDM	%PDB	Sec
pr124	59030	2.52	1.07	3.30	4.73	1.85	9.88
pcb442	50778	5.18	3.41	4.00	5.26	3.24	42.13
u724	41910	6.19	4.96	4.64	6.29	4.67	85.61
lu980	11340	5.47	3.40	4.47	8.97	4.58	125.88
Average		4.84	3.21	4.10	6.31	3.59	65.88

Table 3. Test results of large size instances

		GPU			CPU		
Problem	Optimal	%PDM	%PDB	Sec	%PDM	%PDB	Sec
ei8246	206171	8.31	7.12	71.38	7.33	6.88	614.36
fi10639	520527	6.93	6.49	66.63	8.94	8.10	952.35
d15112	1573084	8.20	7.66	109.28	7.35	7.14	1761.23
bm33708	959304	6.07	5.85	254.22	7.28	7.04	7936.33
Average		7.38	6.78	125.38	7.73	7.29	2816.07

As shown in Fig.4 and Fig.5, and in Table 2 and Table 3, respectively for the two instance groups, our GPU parallel SOM approach outperforms its counterpart CPU sequential version both on small size and large size instances, for similar tour length results. For small size instances, the ratio of CPU time by GPU time (called acceleration factor) varies from roughly factor 3 to factor 28, as the instance size grows. For large size instances, it varies from roughly factor 9 to factor 31 for the maximum size instance with up to 33708 cities. We think that the acceleration factor augmentation indicates a better streaming multiprocessor occupancy as the instance size grows. We can note that the execution time of GPU version increases in a linear way with a very weak increasing coefficient, when compared to the CPU version execution time. We consider that such results are encouraging in that the parallel SOM model should really exploit the benefits of multi-processors, as the number of physical cores will augment in the future.

6 Conclusion

In this paper we propose a cellular-based parallel model for the self-organizing map and apply it to the large scale Euclidean traveling salesman problems. We did not find in the literature GPU implementations to such large size problems with up to 33708 cities. We think that this is because current GPU applications to the TSP concern memory consuming algorithms, such as ant colony, genetic algorithm or k-opt local search, which generally require $O(N^2)$ memory size. Whereas, our approach is dimension with $O(N)$ memory size. We implement our model on a GPU platform and compare the results with its counterpart CPU version. Test results shows that our GPU model has linear increasing execution time with a very weak increasing coefficient when compared to the CPU version, for both small size instances and large size instances.

Future work should deal with verification of effectiveness of the algorithm as the number of physical cores augments. More precisely, we should verify the possibility to design a weakly linear increasing, or ideally a near constant time algorithm, for bounded or uniform distributions, when the number of physical cores really increases as the instance size increases. It should be of interest also to study more CUDA programming techniques, for a better memory coalescing, or the use of shared memory. Moreover, implementations of the model to other parallel computing systems are also potential areas of research.

References

1. Papadimitriou, C.H.: The euclidean travelling salesman problem is np-complete. Theoretical Computer Science 4, 237–244 (1977)
2. Kohonen, T.: Self-organizing maps, vol. 30. Springer (2001)
3. Angeniol, B., de La Croix Vaubois, G., Le Texier, J.Y.: Self-organizing feature maps and the travelling salesman problem. Neural Networks 1, 289–293 (1988)
4. Cochrane, E., Beasley, J.: The co-adaptive neural network approach to the euclidean travelling salesman problem. Neural Networks 16, 1499–1525 (2003)
5. Créput, J.C., Koukam, A.: A memetic neural network for the euclidean traveling salesman problem. Neurocomputing 72, 1250–1264 (2009)
6. McConnell, S., Sturgeon, R., Henry, G., Mayne, A., Hurley, R.: Scalability of self-organizing maps on a gpu cluster using opencl and cuda. Journal of Physics: Conference Series 341, 012018 (2012)
7. Yoshimi, M., Kuhara, T., Nishimoto, K., Miki, M., Hiroyasu, T.: Visualization of pareto solutions by spherical self-organizing map and its acceleration on a gpu. Journal of Software Engineering and Applications 5 (2012)
8. Bentley, J.L., Weide, B.W., Yao, A.C.: Optimal expected-time algorithms for closest point problems. ACM Transactions on Mathematical Software (TOMS) 6, 563–580 (1980)
9. Reinelt, G.: Tsplib a traveling salesman problem library. ORSA Journal on Computing 3, 376–384 (1991)
10. NVIDIA: CUDA C Programming Guide 4.2, CURAND Library, Profiler User's Guide (2012), `http://docs.nvidia.com/cuda`
11. Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional (2010)