The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 2011, 1–21

RESEARCH ARTICLE

Accelerating Ant Colony Optimization for the Traveling Salesman Problem on the GPU

Akihiro Uchida, Yasuaki Ito, and Koji Nakano*

Department of Information Engineering, Hiroshima University, Higashi-Hiroshima, Japan

(*Received 00 Month 200x; in final form 00 Month 200x*)

Recent Graphics Processing Units (GPUs) can be used for general purpose parallel computation. Ant Colony Optimization (ACO) approaches have been introduced as nature-inspired heuristics to find good solutions of the Traveling Salesman Problem (TSP). In ACO approaches, a number of ants traverse the cities of the TSP to find better solutions of the TSP. The ants randomly select next visiting cities based on the probabilities determined by total amounts of their pheromone spread on routes. The main contribution of this paper is to present sophisticated and efficient implementation of one of the ACO approaches on the GPU. In our implementation, we have considered many programming issues of the GPU architecture including coalesced access of global memory, shared memory bank conflicts, etc. In particular, we present a very efficient method for random selection of next cities by a number of ants. Our new method uses iterative random trial which can find next cities in few computational costs with high probability. This idea can be applied not only GPU implementation, but also CPU implementation. The experimental results on NVIDIA GeForce GTX 580 show that our implementation for 1002 cities runs in 8.71 seconds, while the CPU implementation runs in 190.05 seconds. Thus, our GPU implementation attains a speed-up factor of 22.11.

Keywords: ant colony optimization; traveling salesman problem; GPU; CUDA; parallel processing

1. Introduction

Graphics Processing Units (GPUs) are specialized microprocessors that accelerate graphics operations. Recent GPUs, which have many processing units connected with an off-chip global memory, can be used for general purpose parallel computation. CUDA (Compute Unified Device Architecture) [1] is an architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [2–6].

The Traveling Salesman Problem (TSP) is one of the most well-known NP-hard problems [7, 8]. Since it is quite simple and has many applications, many researchers have been devoted to find good approximation algorithm. *Ant colony optimization* (ACO) is known as a potent method introduced as a nature-inspired meta-heuristic for the solution of combinatorial optimization problems [9, 10]. The idea of ACO is based on the behavior of real ants exploring a path between their colony and a source of food. More specifically, when searching for food, ants initially explore the area surrounding their nest at random. Once an ant finds a food source, it

*Corresponding author. Email: nakano@cs.hiroshima-u.ac.jp

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

evaluates the quantity and the quality of the food and carries some of it back to the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of pheromone will guide other ants to the food source. The indirect communication between the ants via pheromone trails makes them possible to find shortest paths between their nest and food sources. In the ACO, the behavior of real ant colonies is exploited in simulated ant colonies to solve the TSP. Approximation solutions of the TSP can be obtained using the ACO in two steps:

Step 1: Initialization
- Initialize the pheromone trail appropriately,

Step 2: Iteration
- For each ant repeat the following operations until some termination condition is satisfied
  - Construct a solution using the pheromone trail
  - Update the pheromone trail

The first step mainly consists in the initialization of the pheromone trail. In the iteration step, each ant constructs a complete solution for the problem according to a probabilistic state transition rule. The rule depends chiefly on the quantity of the pheromone. Once all ants construct solutions, the quantity of the pheromone is updated in two phases: an evaporation phase in which a fraction of the pheromone evaporates, and a deposit phase in which each ant deposits an amount of pheromone that is proportional to the fitness of its solution. This process is repeated until some termination condition is satisfied.

Several variants of ACO have been proposed in the past. The typical ones of them are Ant System (AS), Max-Min Ant System (MMAS), and Ant Colony System (ACS). AS was the first ACO algorithm to be proposed [9, 10]. The characteristic is that pheromone trails is updated when all the ants have completed the tour shown in the above algorithm. MMAS is an improved algorithm over the AS [11]. The main different points are that only the best ant can update the pheromone trails and the minimum and maximum values of the pheromone are limited. Another improvement over the original AS is ACS [12]. The pheromone update, called local pheromone update, is performed during the tour construction process in addition to the end of the tour construction.

The main contribution of this paper is to implement the AS to solve the TSP on the GPU. The TSP is modeled as a complexly connected, undirected weight graph, such that cities are the vertices, paths are the edges, and costs equated to distances between the cities are associated with the weights of each edge. The TSP is finding the cheapest round-trip route, we call *tour*, which visits each vertex exactly once. In the TSP, a salesman visits n cities, and makes a tour visiting every city exactly once. The goal of the TSP is to find the shortest possible tour. We model the problem as a complete graph with n vertices that represent the cities. Let $v_0, v_1, \ldots, v_{n-1}$ be vertices that represent n cities, $e_{i,j}$ $(0 \leq i, j \leq n-1)$ denote edges between cities, and (x_i, y_i) $(0 \leq i \leq n-1)$ be the location of v_i. Let $d_{i,j}$ be the distance between v_i and v_j. Like many other works, we assume that the distance between two cities is their Euclidean distance. Namely, each distance between cities i and j is $d_{i,j} = d_{j,i} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Given distances between two cities $d_{i,j} (0 \leq i, j \leq n-1)$, the TSP is to find a tour T which minimizes the objective function S:

$$S = \sum_{e_{i,j} \in T} d_{i,j}.$$

TSP is well known as an NP-hard problem in combinatorial optimization and utilized as a benchmark problem for various meta-heuristics such as ACO, genetic algorithm, tabu search, etc.

Many algorithms of ACO for the TSP have been proposed in the past. Manfrin *et al.* have shown a parallel algorithm of MMAS with 4 network-connected computers using MPI [13]. Delisle *et al.* have proposed an efficient and straightforward OpenMP implementation with the multi-processor system [14]. Also, GPU implementations have been proposed. In [15], a GPU implementation of MMAS is shown. Kobashi *et al.* have shown a GPU implementation of AS [16]. The implementation introduces nearest neighbor technique to reduce the computing time of tour construction. Cecilia *et al.* have proposed a GPU implementation of AS [17]. To reduce the computing time of tour construction on the GPU, instead of the ordinary roulette-wheel selection used when ants select a next visiting city, they introduced an alternative method, called *I-Roulette*. This method is similar to the roulette-wheel selection, however, it does not exactly compute the roulette-wheel selection.

In our implementation, we have considered many programming issues of the GPU architecture such as coalesced access of global memory, shared memory bank conflicts, etc. To be concrete, arranging various data in the global memory efficiently, we try to make the bandwidth of the global memory of the GPU maximized. Also, to avoid the access to the global memory as much as possible, we utilize the shared memory that is an on-chip memory of the GPU.

In addition, we have introduced a stochastic method, called *stochastic trial*, instead of the roulette-wheel selection that is used when ants determine a next visiting city. Using the stochastic trial, most prefix-sums computation performed in the roulette-wheel selection can be omitted. Since the computing time of the prefix-sums computation is dominated in that of the AS for TSP, we attained further speed-up of it.

Note that our goal in this paper is to accelerate the AS on the GPU, not to improve the accuracy of the solution. Namely, in this research, we have implemented the original AS on the GPU as it is (Figure 1). Therefore the solution accuracy of our GPU implementation is equal to the original AS. However, the GPU implementation is faster than the CPU implementation.

We have implemented our parallel algorithm in NVIDIA GeForce GTX 580. The experimental results show that our implementation can perform the AS for 1002 cities in 8.71 seconds when the tour construction and pheromone update are repeated 100 times., while the CPU implementation runs in 190.05 seconds. Thus, our GPU implementation attains a speed-up factor of 22.11 over the CPU implementation.

The rest of this paper is organized as follows; Section 2 introduces ant colony optimization for traveling salesman problem. In Section 3, we show the GPU and CUDA architectures to understand our new idea. Section 4 proposes our new ideas to implement the ant colony optimization for traveling salesman problem on the GPU. The experimental results are shown in Section 5. Finally, Section 6 offers concluding remarks.

2. Ant Colony Optimization for the Traveling Salesman Problem

In this section, we review how the TSP can be solved by the ant system (AS). Recall that in the TSP, a salesman visits n cities $v_0, v_1, \ldots, v_{n-1}$ in a 2-dimensional space such that every city is visited exactly once. The goal of the TSP is to find the shortest possible tour. In the AS for the TSP, ants work as agents performing the

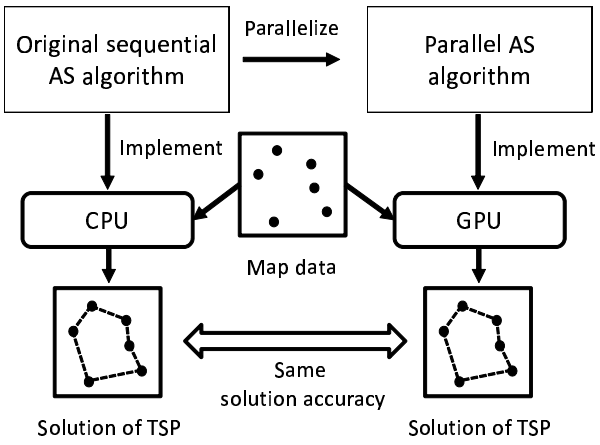

Figure 1. Summary of our research

distributed search. Every edge $e_{i,j}$ connecting a pair of cities v_i and v_j $(0 \leq i, j \leq n)$ has pheromone value $\tau_{i,j}$. Note that every edge is undirected, that is, $e_{i,j}$ and $e_{j,i}$ represent the same edge and $\tau_{i,j}$ and $\tau_{j,i}$ take the same value. Each pheromone value $\tau_{i,j}$ is repeatedly updated according to the behavior of ants in the AS. Thus, we can consider that $\tau_{i,j}$ is a variable, and $\tau_{i,j}$ and $\tau_{j,i}$ represent the same variable.

First, a number of ants are placed at cities independently at random. Recommended in [18], the number of ants is equal to the number n of the cities. Each ant repeatedly selects a next visiting city at random among the unvisited cities. In the selection of a next visiting city, an edge with larger pheromone value is selected with higher probability. After each ant visits all cities, it goes back to the starting city. Clearly, the visiting order of an ant is a solution of the TSP. Thus, after all n ants visit all cities, we have n solutions of the TSP. Based on the n solutions, every pheromone value $\tau_{i,j}$ is updated. This procedure is repeated until some termination condition is satisfied.

We are now in a position to explain the details of the AS. It has three steps *the initialization*, *the tour construction*, and *the pheromone update*. In the initialization step, the pheromone value of every $\tau_{i,j}$ is initialized by the same value. For the reader's benefit, we explain the initialization step after explaining the other steps.

2.1 Tour construction

Each ant selects a starting city independently and uniformly at random. In other words, we generate n independent random integer values in $[0, n-1]$ and place n ants at the corresponding cities. After that, every ant traverses all cities independently. The selection of a next visiting city by a well-known method called *the roulette-wheel selection* [19]. We focus on a particular ant and explain how it traverses all cities using the roulette-wheel selection.

Suppose that the ant is currently visiting the city v_i. Let U denote a set of cities that the ant has not visited. We will show how we select a next visiting city from U. We define *the fitness* $f_{i,j}$ of an edge v_i and v_j with respect to the ant by the following formula:

$$f_{i,j} = \frac{\tau_{i,j}^{\alpha}}{d_{i,j}^{\beta}} \tag{1}$$

where $\tau_{i,j}$ and $d_{i,j}$ denote the pheromone value of an edge between v_i and v_j and

the Euclidean distance of v_i and v_j. Also, $\alpha > 0$ and $\beta > 0$ are fixed values to control the influence of the pheromone and distance values. These values are usually determined by experiments. Clearly, $f_{i,j}$ takes large value if the pheromone value $\tau_{i,j}$ is larger and the distance $d_{i,j}$ is smaller. Hence we can think that the value of $f_{i,j}$ represents the fitness of an edge from v_i to v_j.

We select a next visiting city in U with probability proportional to $f_{i,j}$. In other words, the probability $p_{i,j}$ to select city v_j as a next visiting city is

$$p_{i,j} = \frac{f_{i,j}}{F} \qquad \text{if } v_j \in U \tag{2}$$

$$= 0 \qquad \text{otherwise.} \tag{3}$$

where $F = \sum_{v_j \in U} f_{i,j}$. Clearly, $\sum_{i=0}^{n-1} p_{i,j} = 1$ and v_j is selected as a next visiting city with higher probability if the fitness $f_{i,j}$ is larger.

The reader should no difficulty to confirm that, starting from a randomly selected city, the ant traverses all cities by repeating this procedure to select a next city. We consider that the ant goes back to the stating city after visiting all cities to complete the tour. The tour thus obtained by the ant is an approximation solution of the TSP.

2.2 Pheromone update

After all of the n ants complete the tour construction, every pheromone value $\tau_{i,j}$ is updated. The pheromone update is performed by two steps: *the pheromone evaporation* and *the pheromone deposit*.

Intuitively, the pheromone evaporation is performed to avoid falling into local optima of the TSP. Every pheromone value is decreased by multiplying a fixed constant factor $(1 - \rho)$. More specifically, $\tau_{i,j}$ $(0 \leq i, j \leq n - 1)$ is updated as follows:

$$\tau_{i,j} \leftarrow (1 - \rho)\tau_{i,j} \tag{4}$$

where ρ is a fixed evaporation rate of the pheromone determined by the experiments.

After the pheromone evaporation the pheromone deposit is performed using the tours obtained by the n ants. Let T_k $(0 \leq i \leq n - 1)$ denote a set of n edges in the tour obtained by the k-th ant. Also, let $L(T_k)$ is the total distance of T_k, that is,

$$L(T_k) = \sum_{(v_i, v_j) \in T_k} d_{i,j}.$$

Clearly, T_k is a good approximation solution of the TSP if $L(T_k)$ is small. The pheromone value $\tau_{i,j}$ is updated using $L(T_k)$ as follows:

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{0 \leq k \leq n-1} \sum_{(v_i, v_j) \in T_k} \frac{1}{L(T_k)}. \tag{5}$$

In other words, for every $(v_i, v_j) \in T_k$, we accumulate $\frac{1}{L(T_k)}$ to $\tau_{i,j}$. Thus, the $\tau_{i,j}$ of edge (v_i, v_j) is large if the edge is included in many tours with small total distance.

2.3 Initialization

We will show an appropriate method to determine the initial value of $\tau_{i,j}$.

A simple greedy method [20] can find an approximation solution of the TSP. In this method, an arbitrary city is selected as a starting city. The nearest unvisited city is always selected as a next visiting city. This selection is repeated until all cities are visited. Let T_G be a tour obtained by the greedy method. Suppose that $T_k = T_G$ for all k, that is every ant finds tour T_k as an initial tour. If this is the case, the pheromone update is performed as follows:

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{0 \leq k \leq n-1} \sum_{(v_i,v_j) \in T_k} \frac{1}{L(T_k)} = \sum_{(v_i,v_j) \in T_G} \frac{n}{L(T_G)}.$$

The total value accumulated to $\tau_{i,j}$ is

$$\sum_{0 \leq i,j \leq n-1} \sum_{(v_i,v_j) \in T_G} \frac{n}{L(T_G)} = \frac{n^3}{L(T_G)}.$$

Thus, it is reasonable to equally deposit this value to all $\tau_{i,j}$. In other words, the value $\tau_{i,j}$ is initialized as follows:

$$\tau_{i,j} \leftarrow \frac{n}{L(T_G)}. \tag{6}$$

3. Compute Unified Device Architecture (CUDA)

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [21]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [22, 23]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, threads should perform the coalesced access when they access to the global memory. Figure 2 illustrates the CUDA hardware architecture.

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, as we can see in Figure 2, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories.

CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each block are executed by processor cores in a single streaming processor. In the execution, threads in a block are split into groups of thread called *warps*. Each of these warps contains the same number of threads

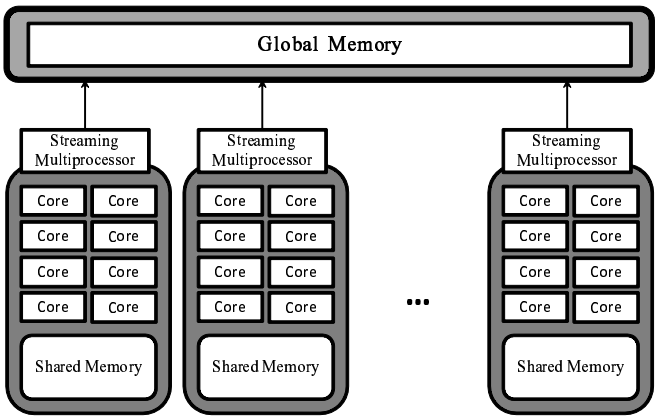

Figure 2. CUDA hardware architecture

and is execute independently. When a warp is selected for execution, all threads execute the same instruction. When one warp is paused or stalled, other warps can be executed to hide latencies and keep the hardware busy.

There is a metric, called *occupancy*, related to the number of active warps on a streaming processor. The occupancy is the ratio of the number of active warps per streaming processor to the maximum number of possible active warps. It is important in determining how effectively the hardware is kept busy. The occupancy depends on the number of registers, the numbers of threads and blocks, and the size of shard memory used in a block. Namely, utilizing too many resources per thread or block may limit the occupancy. To obtain good performance with the GPUs, the occupancy should be considered.

As we have mentioned, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 3, when threads access to continuous locations in a row of a 2-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed at the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed at the same time (*stride access*). From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus, we should avoid the stride access (or the vertical access) and perform the coalesced access (or the horizontal access) whenever possible.

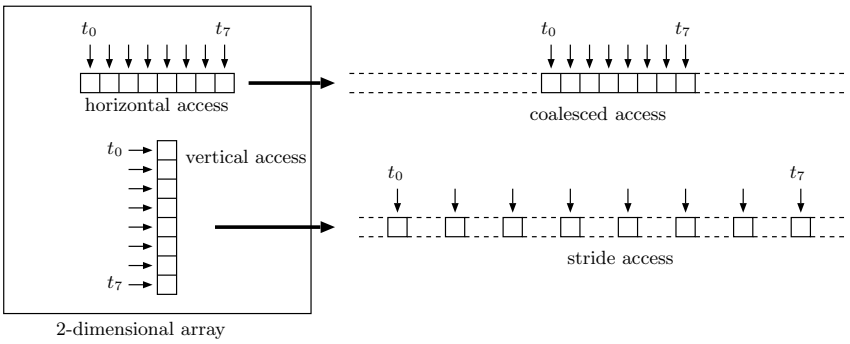

Figure 3. Coalesced and stride access

Just as the global memory is divided into several partitions, the shared memory is also divided into 16 (or 32) equally-sized modules of 32-bit width, called banks

(Figure 4). In the shared memory, the successive 32-bit words are assigned to successive banks. To achieve maximum throughput, concurrent threads of a block should access different banks, otherwise, bank conflicts will occur. In practice, the shared memory can be used as a cache to hide the access latency of the global memory.

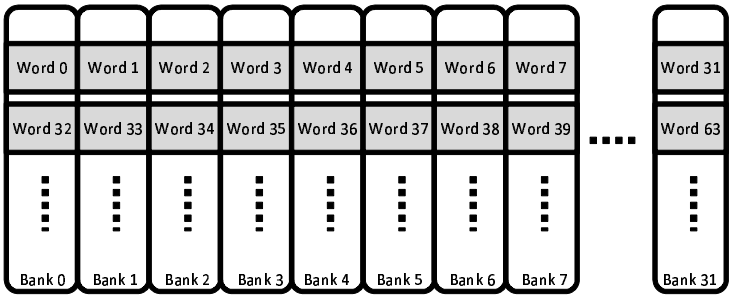

Figure 4. The structure of the shared memory

4. GPU Implementation

The main purpose of this section is to show a GPU implementation of AS for TSP. In the implementation of AS for TSP, much more memory operations are performed than arithmetic operations. Therefore, the performance greatly depends on the memory bandwidth. Since the memory bandwidth of the GPU is much greater than that of the CPU [21], we can accelerate the computation if we sufficiently enjoy the high bandwidth of the GPU. Thus the idea of our implementation is to consider programming issues of the GPU system such as coalesced access of global memory and shared memory bank conflicts in Section 3. To be concrete, arranging various data in the global memory efficiently, we try to make the bandwidth of the global memory of the GPU maximized. Also, to avoid the access to the global memory as much as possible, we utilize the shared memory. Given n coordinates (x_i, y_i) of city v_i $(0 \leq i \leq n-1)$, our implementation computes the shortest possible route that visits each city once and returns to the origin city. Our implementation consists of three CUDA parts, initialization, tour construction, and pheromone update. We use CURAND [24], a standard pseudorandom number generator of CUDA, when we need to generate a random number. The details of our GPU implementation are spelled out as follows.

4.1 Initialization

Given n coordinates (x_i, y_i) of city v_i, each distance $d_{i,j}$ between cities v_i and v_j and a single thread is used to find T_G by the greedy algorithm and computes $\frac{n}{L(T_G)}$, which is the initial value for all $\tau_{i,j}$.

4.2 Tour construction

Recall that in the tour constriction, n ants are initially positioned on n cities chosen randomly. Each ant makes a tour with the roulette-wheel selection independently. Whenever each ant visits a city, it determines a next visiting city with the roulette-wheel selection. To perform the tour construction on the GPU, we consider four

methods, *SelectWithoutCompression*, *SelectWithCompression*, *SelectWithStochasticTrial*, and a hybrid method that is a combination of the above methods. Let us consider the case when ant k is in city v_i. In advance, the fitness values $f_{i,j}$ $(0 \leq i, j \leq n-1)$ are computed by Eq. (1) and stored to the 2-dimensional array in the global memory. Also, the elements related to city v_i, i.e., $f_{i,0}, \ldots, f_{i,n-1}$, are stored in the same row so that the access to the elements can be performed with the coalesced access. In the tour construction, ant k $(0 \leq k \leq n-1)$ makes a tour index array t_k such that element $t_k(i)$ stores the index of the next city from city v_i shown in Figure 5.

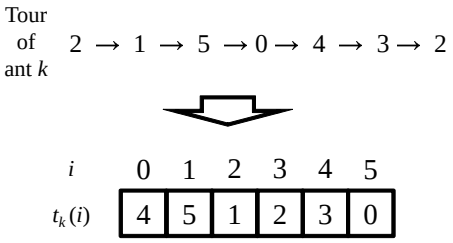

Figure 5. Representation of tour list

4.2.1 SelectWithoutCompression

Each ant has an unvisited list $u_0, u_1, \ldots, u_{n-1}$ such that

$$u_j = \begin{cases} 0 \text{ if city } v_j \text{ has been visited} \\ 1 \text{ otherwise.} \end{cases} \tag{7}$$

To perform the roulette-wheel selection, when ant k is in city v_i, we compute as follows:

Step 1: Compute the prefix sums q_j $(0 \leq j \leq n-1)$ such that

$$q_j = f_{i,0} \cdot u_0 + f_{i,1} \cdot u_1 + \cdots + f_{i,j} \cdot u_j. \tag{8}$$

Step 2: Generate a random number r in $[0, q_{n-1})$.

Step 3: For simplicity, let $q_{-1} = 0$. Find j such that $q_{j-1} \leq r < q_j$ $(0 \leq j \leq n-1)$. City v_j is selected as the next visiting city.

Clearly r is in $[q_{j-1}, q_j)$ with probability $\frac{q_j - q_{j-1}}{q_{n-1}} = \frac{f_{i,j} \cdot u_j}{q_{n-1}}$. Thus, if v_j is unvisited, that is, $u_j = 1$, then the next visiting city is v_j with probability $\frac{f_{i,j}}{q_{n-1}}$. Hence the next visiting city is selected with probability Eq. (2) correctly. Figure 6 shows a summary of SelectWithoutCompression. In Step 1, the values $\tau_{i,j}$ is computed by Eq. (1) are read from the global memory by threads with coalesced access and stored to the shared memory. After that, the prefix-sums in Eq. (8) are computed, where the fitness values of visited cities are 0 not to be selected. To avoid the branch instruction whether the candidate of the next cities has been visited or not, we multiply $f_{i,j}$ and u_j with the unvisited list in Eq. (7). In our implementation, the prefix-sums computation is performed using the parallel prefix-sums algorithm proposed by Harris *et al.* [25], Chapter 39. It is an in-place parallel prefix-sums algorithm with the shared memory on the GPU. Also, it can avoid most bank conflicts by adding a variable amount of padding to each shared memory array index. On the other hand, this method has a fault that the number of elements that it can perform must be power of two. Therefore, when the number of elements

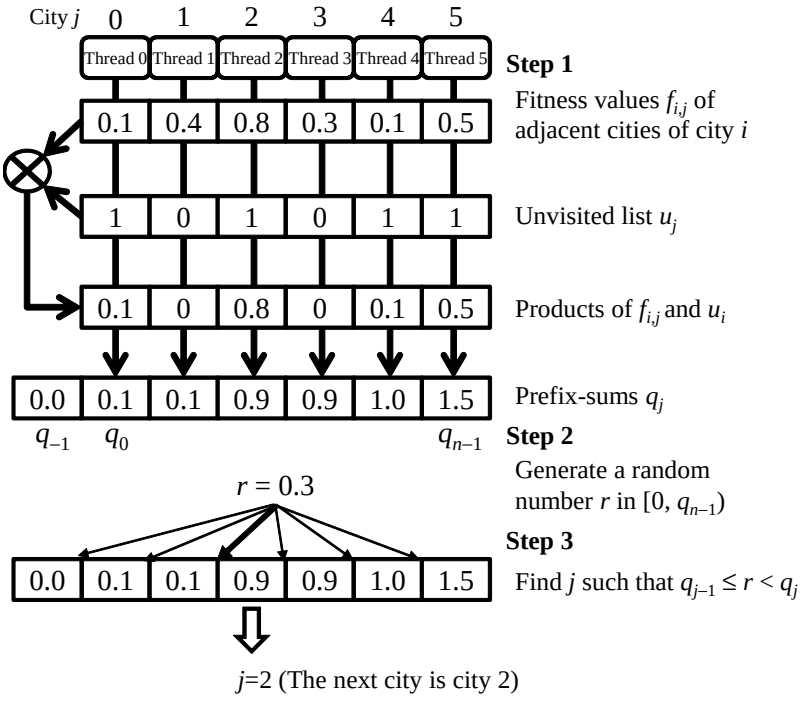

Figure 6. Parallel roulette-wheel selection in SelectWithoutCompression

is a little more than power of two numbers, the efficiency is decreased. For example, if the number of elements is 4097, the method must perform for 8192 elements. This fault can be ignored for small number of elements. However, it cannot be ignored for large number.

After that, a number r in $[0, q_{n-1})$ is generated uniformly at random using CURAND. Using the random number, an index j such that $q_{j-1} \leq r < q_j$ is searched and city v_j is the next visiting city. In this search operation, we use a parallel search method based on the parallel K-ary search [26]. The idea of the parallel K-ary search is that a search space in each repetition is divided into K partitions and the search space is reduced to one of the partitions. In general, Binary search is a special case ($K = 2$) of K-ary search. In our parallel search method, we divide the search space into 32 partitions. Sampling the first elements of each partition, a partition that includes the objective element to search is found by 32 threads, i.e., 1 warp. After that the objective element is searched from the partition by threads whose number is the number of elements in the partition.

The feature of this method is that the fitness values can be read from the global memory with the coalesced access. Although the number of unvisited cities is smaller, in every selection to determine the next city, the roulette-wheel selection has to be performed for both visited and unvisited cities. Namely, the data related to both of the visited and unvisited cities is necessary Although the number of unvisited cities is smaller, computing time cannot be reduced. In other words, it does not depend on the number of visited cities.

4.2.2 SelectWithCompression

The idea of this method is to select only from unvisited cities excluding the visited cities. Instead of the unvisited list in the above method, we use an *unvisited index array* that stores indexes of unvisited cities. When the number of unvisited cities is n', the array consists of elements $v_0, v_1, \ldots, v_{n'-1}$ and each element stores an index of one of the unvisited cities. When a city is visited, the city has to be

removed from the index array. The removing operation takes $O(1)$ time by overwriting the index of the next city with that of the last element, then removing the last element (Figure 7). Using the unvisited index array, it is not necessary to read the data related to the visited cities to compute the prefix-sums in Eq. (8) though SelectWithoutCompression requires data related to both visited and unvisited cities. Therefore, when the number of unvisited cities is smaller, the computing time becomes smaller. However, the global memory access necessary to compute the prefix-sums may not be done with the coalesced access because the contents of the index array are out of order using the above array update. Therefore, when the number of unvisited cities is large, computing time of SelectWithCompression is perhaps slower than that of SelectWithoutCompression.

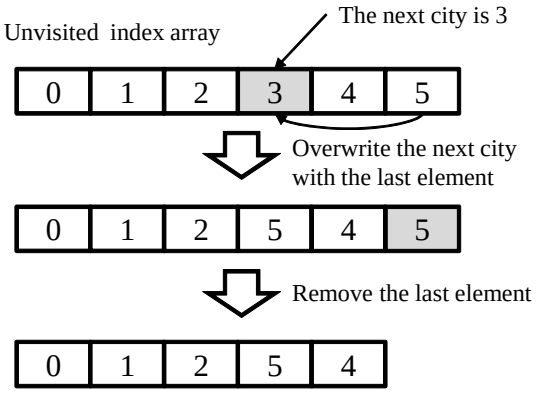

Figure 7. Update of the unvisited index array when city 3 is selected as the next city.

4.2.3 SelectWithStochasticTrial

In the above two methods, whenever each ant visits a city, the prefix-sums computation has to be performed. The prefix-sums computation occupies the most of the computing time of the tour construction. The idea of this method is to avoid the prefix-sums computation as much as possible using *stochastic trial*. The details of the stochastic trial are shown as follows.

Before ants start visiting cities, the prefix-sums $q'_{i,0}$, $q'_{i,1}$, ..., $q'_{i,n-1}$ for every city v_i are computed such that

$$q'_{i,j} = f_{i,0} + f_{i,1} + \cdots + f_{i,j}. \tag{9}$$

As before, let $q'_{i,-1} = 0$ for simplicity. Note that the values of q_i used in SelectWithoutCompression depends on the list of unvisited cities, while the values of $q'_{i,j}$ are independent of it. Hence, once all $q'_{i,j}$ are computed, we do not have to update them. We assume that the values of $q'_{i,j}$ are stored in a 2-dimensional array in the global memory such that $q'_{i,-1}, \ldots, q'_{i,n-1}$ are stored in the i-th row. When an ant is visiting city v_i, the next visiting city is selected as follows:

Step 1: Generate a random number r in $[0, q'_{i,n-1})$.

Step 2: Find j such that $q'_{i,j-1} \leq r < q'_{i,j}$ $(0 \leq j \leq n-1)$. If city v_j is unvisited, it is selected as the next city. Otherwise, these steps are performed until an unvisited city is selected.

In Step 2, the unvisited list (Eq. (7)) is used to find whether the city has been visited or not by the parallel search shown in the above methods.

When almost all of the cities are visited, Step 2 succeeds in select an unvisited city with very small probability. If this is the case, the number of iteration can be very large. Hence, if the next city is not determined in the w-time iteration for some constant w determined by the experiments, we select the next city by SelectWithoutCompression. These steps are similar to the roulette-wheel selection in the above methods. The difference point is not always to determine the next city since a candidate of the next city found by the random selection may have been visited. In followings, the above operation is called *stochastic trial*. SelectWithStochasticTrial repeats the stochastic trial at most w times. If the next city cannot be determined, it is selected by SelectWithoutCompression. When the number of unvisited city is smaller or some of the fitness values of visited cities are larger, almost the trial cannot select the next city. However, the computing time is much shorter than that of the prefix-sums calculation. Therefore, if the next city can be determined in the above steps within w times, the total computing time can be reduced by this method. It is important for this method to determine the value of w. This is because w has to be determined considering the balance between the computing time of the iteration of the stochastic trial and that of SelectWithoutCompression performed when the next city cannot be determined. In Section 5, we will obtain the optimal times w by experiments.

4.2.4 Hybrid Method

In SelectWithStochasticTrial, however, when the number of visited cities is large, a next visiting city may not be determined by the stochastic trial and has to be selected by SelectWithoutCompression. Therefore, we introduce a hybrid method such that when the number of visited city is small, SelectWithStochasticTrial is performed. Then, SelectWithStochasticTrial is switched to SelectWithoutCompression. After that next visiting cities are determined by SelectWithCompression until all the cities are visited. The reason that SelectWithCompression is performed after SelectWithStochasticTrial is that when the number of unvisited cities is small, SelectWithCompression is performed faster than SelectWithoutCompression. In the followings, we call such method *hybrid method*. An important point of this hybrid method is to determine the timing when SelectWithStochasticTrial is switched such that the computing time is minimized. In Section 5, we will obtain the optimal timing by experiments.

4.3 Pheromone update

In the followings, we show a GPU implementation of pheromone update. The idea of the implementation is efficient memory access by coalesced access, the shared memory and avoiding bank conflict. Recall that the pheromone update consists of the pheromone evaporation and the pheromone deposit. In our implementation, the values of pheromone $\tau_{i,j}$ $(0 \leq i \leq j \leq n-1)$ are stored in a 2-dimensional array, which is a symmetric array, that is, $\tau_{i,j} = \tau_{j,i}$, in the global memory and are updated by the results of the tour construction. Making the array symmetric, the elements related to city v_i, i.e., $\tau_{i,0}, \tau_{i,1}, \ldots, \tau_{i,n-1}$, are stored in the same row so that the access to the elements can be performed with the coalesced access. Our implementation consists of two kernels, *PheromoneUpdateKernel* and *SymmetrizeKernel*.

4.3.1 PheromoneUpdateKernel

This kernel assigns n blocks that consist of multiple threads to each row of the array and each block performs the followings independently. Figure 8 shows a summary of the pheromone update on the GPU for a block that performs

the pheromone update for city 0. Threads in block i read $\tau_{i,0}, \tau_{i,1}, \ldots, \tau_{i,n-1}$ in the i-th row with coalesced access, and then store them to the shared memory. When the values are stored to the shared memory, each value is halved in advance since they are doubled in the following kernel, SymmetrizeKernel. After that, the pheromone evaporation is preformed, i.e., each value is reduced by Eq. (4) by threads in parallel. To perform the pheromone deposit, block i reads the values $t_0(i), t_1(i), \ldots, t_{m-1}(i)$ in the i-th row of the tour lists. The read operation is performed with coalesced access by threads. Also, each total tour length of each ant $C_0, C_1, \ldots, C_{m-1}$ stored in the global memory is read. After that threads add a quantity obtained by Eq. (5) to the corresponding values of pheromone in parallel. In the addition, some threads may add to the same pheromone simultaneously. To avoid it, we use the atomic add operation supported by CUDA [21]. After the addition, the values of pheromone are stored back to the global memory. Note that since the 2-dimensional array storing the pheromone values are symmetry, if addition to $\tau_{i,j}$ is performed, that to $\tau_{j,i}$ has to be also performed. However, the above deposit operation adds to either $\tau_{i,j}$ or $\tau_{j,i}$. To obtain the correct results, SymmetrizeKernel is performed.

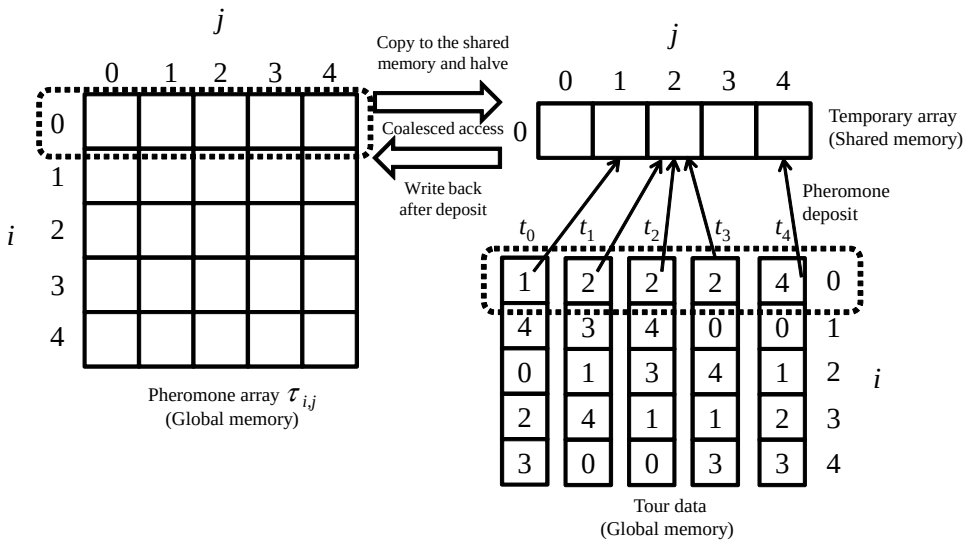

Figure 8. A summary of pheromone update

4.3.2 SymmetrizeKernel

This kernel symmetries the 2-dimensional array storing the pheromone values after performing PheromoneUpdateKernel. More specifically, summing corresponding two elements that are symmetric, each value of symmetric elements is made identical. In this kernel, to make the access to the global memory coalesced, the 2-dimensional array that stores the pheromone values is divided into subarrays of size 32×32. We assign one block to two subarrays that are symmetric or one subarray that includes symmetric elements. Blocks symmetrize the whole array subarray by subarray. To symmetrize the subarrays, one array has to be transposed.

For the transposing, we utilize an efficient method proposed in [27]. The idea of the method is to transpose a 2-dimensional data stored in the global memory via the shared memory with the coalesced access and avoidance of the bank conflict shown in Section 3, as follows. The 2-dimensional array is divided into subarrays whose size is 32×32. One block is assigned to each subarray and each block works

independently. As shown in Figure 9, in each block, values in the corresponding subarray are read in column wise using coalesced access with 32 threads. The read operation is executed row by row. The read values are stored into the shared memory in column wise. After reading 32×32 values from the global memory, the stored values in the shared memory are added to the corresponding transposed position in the global memory in column wise with coalesced access. Using the above transposing via the shared memory, all the access from/to the global memory can be coalesced. However, the use of the shared memory causes another problem, bank conflicts. As given above, the shared memory is used as a 2-dimensional array whose size of 32×32. It means that one column of the subarray mapped into the same bank of the shared memory shown in Section 3. In the above operation, when threads store the values to the shared memory, they access to the same column of the 2-dimensional array, that is, bank conflict occurs (Figure 10(a)). To avoid the bank conflicts, a dummy column to the 2-dimensional array (Figure 10(b)). Adding the dummy column, values of each column are mapped into distinct banks and all the access in the transporting is free from the bank conflicts. Note that when the symmetrization is performed, each value is doubled since the original values are added twice. Therefore they are halved in advance in the previous kernel, PheromoneUpdateKernel.

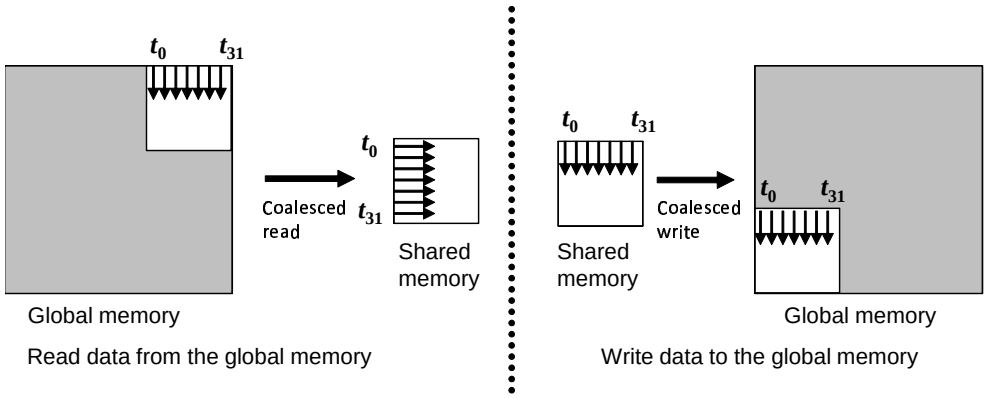

Figure 9. Coalesced transpose with the shared memory

5. Performance Evaluation

We have implemented our AS for the TSP using CUDA C. We have used an NVIDIA GeForce GTX580 with 512 processing cores (16 Streaming Multiprocessors which have 32 processing cores each) running in 1.544GHz and 3GB memory. For the purpose of estimating the speed up of our GPU implementation, we have also implemented a software approach of AS for the TSP using GNU C. In the software implementation, we can apply the idea of the tour construction in the GPU implementation such as the stochastic trial. Each of them will be compared in the followings. We have used Intel Core i7 860 running in 2.8GHz and 3GB memory to run the sequential algorithm for the AS. We have evaluated our implementation using a set of benchmark instances from the TSPLIB library [28]. In the following evaluation, we utilize 8 instances: *d198, a280, lin318, pcb442, rat783, pr1002, pr2392* and *pcb3038* from TSBLIB. Each name consists of the name of the instance and the number of cities. For example, *pr1002* means that the name of the instance is *pr* and the number of cities is 1002. The parameters of ACO, α, β,

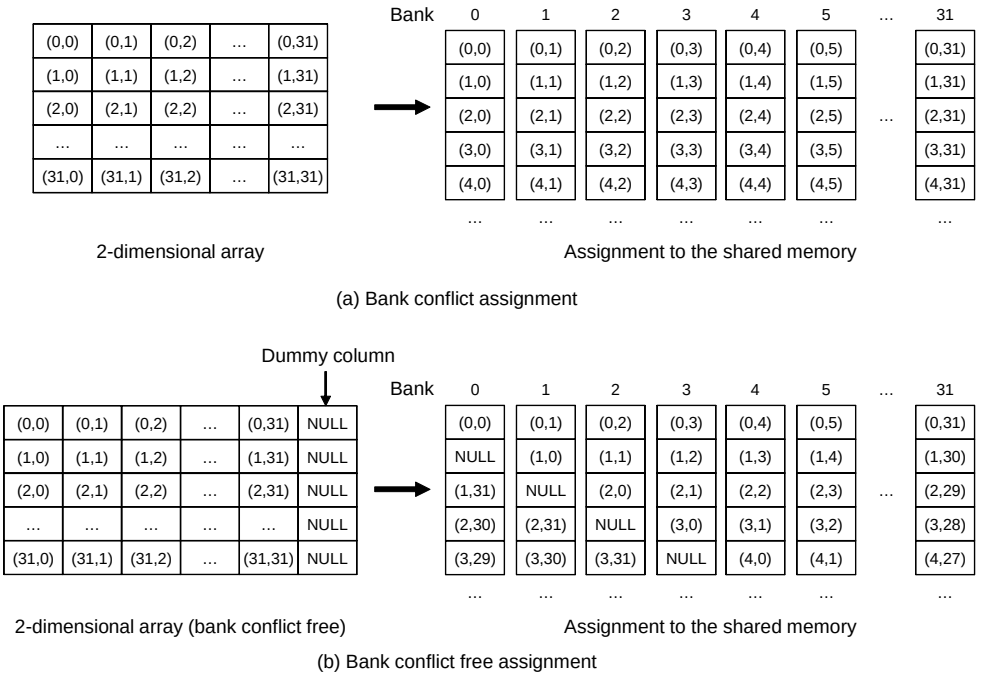

Figure 10. Bank conflict free assignment on the shared memory

and ρ in Eq. (1) and Eq. (4), are set to 1.0, 2.0, and 0.5, respectively. In CUDA, it is important to determine the number of blocks and the number of threads in each block. It greatly influences the performance of the implementation on the GPU. In the followings, we select the optimal numbers obtained by experiments. We first explain the performance of the tour construction and the pheromone update, and then the results of overall performance are shown.

5.1 Evaluation of the tour construction

Before the performance of the tour construction is evaluated, we determine the optimal parameters. One is an upper limit of times of iteration how many times the stochastic trial is repeated if a next visiting city is not determined in SelectWithStochasticTrial. The other is timing when SelectWithStochasticTrial is switched to SelectWithCompression in the hybrid method.

To obtain an optimal upper limit of the iteration of the stochastic trial if a next visiting city is not determined in SelectWithStochasticTrial, we evaluated the number of times necessary to determine next cities in a tour construction for the pheromone values obtained after the tour construction and the pheromone update were repeated 100 times for pr1002. Figure 11 is a graph that shows a histogram of the number of cities and its cumulative histogram of the percentage of cities to the number of times of iteration how many times the stochastic trial is repeated. For example, when the number of times of iteration is 5, the number of cities is 30 and the percentage of cities is about 80%. This means that in 30 cities, the next city was determined by the stochastic trial 5 times and in 80% cities, it was determined by the stochastic trial within 5 times. From the figure, in approximately half of cities, the next city can be determined by the stochastic trial one time. Also, in about 90% cities, the next city can be selected within about 32 times. In several cities, the next city could not be determined when the stochastic trial was repeated more than 2000 times. Considering the balance of computing time between the stochastic

trial and SelectWithoutCompression when the next city cannot be determined, in the following experiments, we set 8 times to the upper limit of times of iteration.

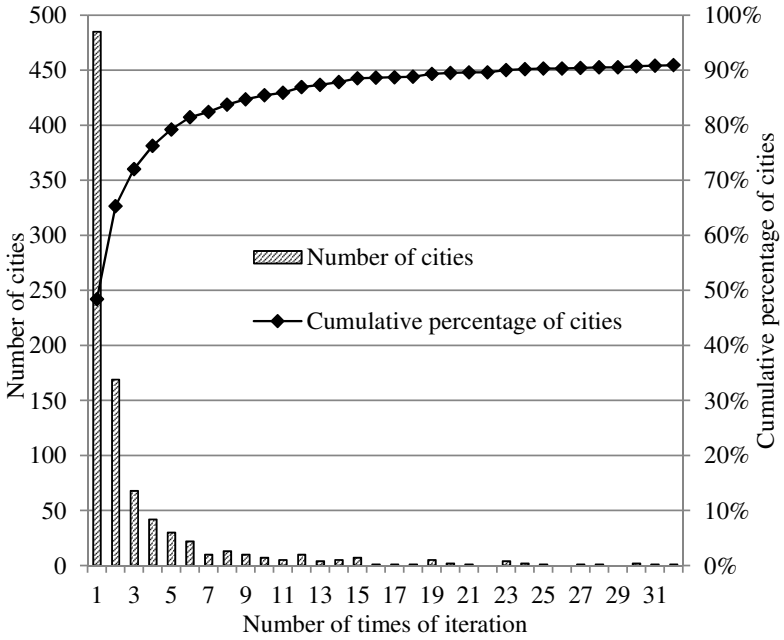

Figure 11. A histogram of the number of cities and its cumulative histogram of the percentage of cities to the number of times of iteration of the stochastic trial

To obtain the timing when SelectWithStochasticTrial is switched to SelectWithCompression in the hybrid method, we have measured the computing time of the tour construction for various percentages when SelectWithStochasticTrial is switched to SelectWithCompression. Figure 12 shows the computing time of tour construction for various instances. According to the figure, the percentage of the visited cities is larger, the computing time is shorter and if it is closed to 100%, it becomes larger. On the other hand, readers can find that the curve for pcb3038 is not smooth around 20%. The reason is that the computing time becomes shorter since the occupancy, described in Section 3, increases. When the number of visited cities is larger, the size of used shared memory storing unvisited cities becomes smaller. In that case, the efficiency of the processors is improved because the occupancy is increased. According to Figure 12, the computing time is minimized by switching the method when about 85% cities are visited. Therefore, we switch from SelectWithStochasticTrial to SelectWithCompression when 85% cities are visited.

To compare the performance among our proposed tour construction methods, we have evaluated the computing time of them. Table 1 shows the computing time of tour construction with various methods for various instances. In both of CPU and GPU implementations, the hybrid method is the fastest. In the CPU implementation, the computing time of SelectWithoutCompression is shorter than that of SelectWithCompression since the computing time of SelectWithCompression is shorter when the number of unvisited cities is small. However, in the GPU implementation, it is a little longer. This is because the global memory access in SelectWithCompression cannot be performed with the coalesced access as shown in Section 4. Compared to the methods without the stochastic trial, for most of the cases, the computing time of the methods with the stochastic trial is shorter than that without it. Additionally, in the CPU and GPU implementation, the computing time of the hybrid method is approximately 40% and 10% shorter than that of SelectWithStochasticTrial, respectively.

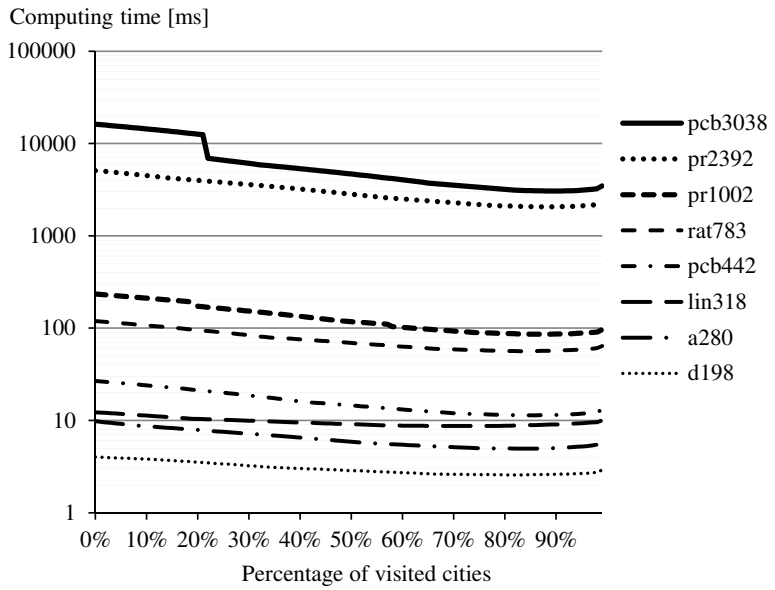

Figure 12. The computing time of tour construction for the various percentages when SelectWithStochasticTrial is switched to SelectWithoutCompression

Table 1. Computing time of tour construction for various instances

	SelectWithoutCompression			SelectWithCompression		
Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	25.01	4.10	6.11	19.84	5.08	3.90
a280 (280)	60.88	10.00	6.09	47.05	15.05	3.13
lin318 (318)	118.37	14.64	8.08	95.15	17.65	5.39
pcb442 (442)	252.72	23.90	10.57	180.61	33.25	5.43
rat783 (783)	1555.49	108.83	14.29	1215.31	153.82	7.90
pr1002 (1002)	4598.04	238.67	19.27	3784.43	253.44	14.93
pr2392 (2392)	68212.89	3823.41	17.84	58452.20	4146.78	14.10
pcb3038(3038)	124208.59	9938.31	12.50	104410.41	10655.35	9.80

	SelectWithStochasticTrial			Hybrid method		
Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	19.19	2.92	6.58	10.88	2.58	4.24
a280 (280)	32.77	5.86	5.60	17.44	4.95	3.98
lin318 (318)	111.57	11.02	10.12	43.79	8.86	8.11
pcb442 (442)	104.81	13.14	7.98	56.33	11.35	6.18
rat783 (783)	955.46	63.87	14.96	346.14	56.38	8.94
pr1002 (1002)	2737.96	96.43	28.39	815.05	86.43	21.87
pr2392 (2392)	34827.20	2302.12	15.13	16511.67	2078.98	12.21
pcb3038(3038)	53987.53	3460.61	15.60	36532.30	3096.72	11.80

5.2 Evaluation of the pheromone update

Table 2 shows the computing time of the pheromone update for various instances. Our GPU implementation can achieve speed-up factors of 22 to 67. Compared to the computing time of the tour construction, the computing time of the pheromone update is much shorter. In other words, the computing time of the tour construction is dominant in the total execution time.

Table 2. Computing time of the pheromone update

Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	0.963	0.042	22.65
a280 (280)	1.384	0.068	20.43
lin318 (318)	2.797	0.076	36.74
pcb442 (442)	4.692	0.127	36.84
rat783 (783)	16.770	0.327	51.24
pr1002 (1002)	34.877	0.530	65.83
pr2392 (2392)	222.762	5.840	38.14
pcb3038(3038)	349.807	8.945	39.11

5.3 Evaluation of overall performance

Table 3 shows overall performance that is the total computing time of AS for various instances. In the tour construction, we used the hybrid method that was the fastest method. Each execution includes the initialization and 100 times iteration of the tour construction and the pheromone update. Since the computing time of the tour construction is much larger than other process, each speed-up factor is similar to that of the tour construction. Our GPU implementation can achieve speed-up factors of 4.51 to 22.11 over the CPU implementation. To attain the same speed-up factor of 22.11 on the CPU implementation with multi-core processors and/or multi-processors, at least 23 processors are necessary. This number of processors is lower bound and it is not easy and not inexpensive to provide this execution environment now. On the other hand, this speed-up factor can be obtained using the proposed method on the commonly used PC with a GPU board that costs several hundred dollars.

Table 3. Total computing time of our implementation when tour constriction and pheromone update are repeated 100 times

Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	1185.52	263.91	4.51
a280 (280)	2102.95	505.51	4.18
lin318 (318)	7400.08	897.29	8.32
pcb442 (442)	7467.31	1153.95	6.50
rat783 (783)	51219.02	5673.15	9.18
pr1002 (1002)	190050.13	8706.32	22.11
pr2392 (2392)	2538336.73	208478.18	12.28
pcb3038(3038)	3688402.99	310613.66	11.87

Table 4 shows a quality comparison for the solutions for the CPU and GPU implementation when the tour construction and the pheromone update are repeated 100 times. From the table, we can find that the difference of the solutions between the CPU and GPU is very small compared with the optimal solutions. This is because we have implemented the sequential algorithm on the GPU as it is. Note that our goal in this paper is to accelerate the AS on the GPU, not to improve the accuracy of the solution.

In the related works of ACO for TSP shown in Section 1, several GPU implementations have been proposed. Table 5 shows the comparison of speed-up factors with the existing approaches. Since those implemented methods, used instance, and utilized GPUs differ, we cannot directly compare our implementation with them. Since the speed-up factor we achieved is 22.11 over our CPU implementation, some readers think that our GPU implementation is as effective as them. However, in-

Table 4. The solution when tour construction and pheromone update are repeated 100 times and the ratio with respect to the optimal solution

Instance (# cities)	CPU	GPU	Optimal
d198 (198)	16796/1.064	16943/1.074	15780
a280 (280)	3174/1.231	3101/1.203	2579
lin318 (318)	48538/1.155	47736/1.136	42029
pcb442 (442)	63312/1.247	62176/1.224	50778
rat783 (783)	11135/1.264	11061/1.256	8806
pr1002 (1002)	335748/1.296	332608/1.284	259045
pr2392 (2392)	514530/1.361	508109/1.344	378032
pr3038 (3038)	186972/1.358	185570/1.348	137694

troducing the hybrid method with the stochastic trial to the conventional CPU implementation, the computing time of our CPU implementation shown in Table 1 is approximately halved. Therefore, our GPU implementation is much more effective.

Table 5. Comparison of speed-up factors with the existing approaches

	Delévacq [15]	Kobashi [16]	Cecilia [17]	This work
GPU	Tesla C2050 $\times 2$	Tesla C2050	Tesla C2050	GeForce GTX580
Speed-up	23.6	23.5	20.0	22.1

6. Conclusions

In this paper, we have proposed an implementation of the ant colony optimization algorithm, especially AS, for the traveling salesman problem on the GPU. In our implementation, we have considered many programming issues of the GPU architecture such as the coalesced access of the global memory and the shared memory bank conflicts. In addition, we have introduced a method with the stochastic trial in the roulette-wheel selection. We have implemented our parallel algorithm in NVIDIA GeForce GTX 580. The experimental results show that our implementation can perform the AS for 1002 cities in 8.71 seconds when the tour construction and pheromone update are repeated 100 times, while the CPU implementation runs in 190.05 seconds. Thus, our GPU implementation attains a speed-up factor of 22.11.

References

[1] NVIDIA Corp., *CUDA ZONE* http://developer.nvidia.com/category/zone/cuda-zone.

[2] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, *Implementations of Parallel Computation of Euclidean Distance Map in Multicore Processors and GPUs*, in *Proceedings of International Conference on Networking and Computing*, 2010, pp. 120–127.

[3] K. Ogawa, Y. Ito, and K. Nakano, *Efficient Canny Edge Detection Using a GPU*, in *Proceedings of International Workshop on Advances in Networking and Computing*, 2010, pp. 279–280.

[4] Y. Ito, K. Ogawa, and K. Nakano, *Fast Ellipse Detection Algorithm using Hough Transform on the GPU*, in *Proc. of International Workshop on Chal-*

lenges on Massively Parallel Processors (CMPP), December, 2011, pp. 313–319.

[5] K. Nishida, Y. Ito, and K. Nakano, *Accelerating the Dynamic Programming for the Matrix Chain Product on the GPU*, in *Proceedings of International Workshop on Challenges on Massively Parallel Processors*, 2011, pp. 320–326.

[6] K. Nakano, *An Optimal Parallel Prefix-sums Algorithm on the Memory Machine Models for GPUs*, in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept., 2012, pp. 99–113.

[7] D.S.J. M. R. Garey, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W H Freeman & Co., 1979.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., The MIT Press, 2001.

[9] M. Dorigo, *Optimization, learning and natural algorithms*, Ph.D. thesis, Dipartimento di Elettronica, Politecnico di Milano, 1992.

[10] M. Dorigo, V. Maniezzo, and A. Colorni, *The ant system: Optimization by a colony of cooperating agents*, IEEE Transactions on Systems, Man, and Cybernetics–Part B 26 (1996), pp. 29–41.

[11] T. Stützle and H.H. Hoos, *MAX–MIN ant system*, Future Generation Computer Systems 16 (2000), pp. 889–914.

[12] M. Dorigo and L.M. Gambardella, *Ant colony system: A cooperative learning approach to the traveling salesman problem*, IEEE Transactions on Evolutionary Computation 1 (1997), pp. 53–66.

[13] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, *Parallel Ant Colony Optimization for the Traveling Salesman Problem*, in *Proc. of 5th International Workshop on Ant Colony Optimization and Swarm Intelligence*, Vol. LNCS 4150, Springer-Verlag, 2006, pp. 224–234.

[14] P. Delisle, M. Krahecki, M. Gravel, and C. Gagné, *Parrallel implementation of an ant colony optimization metaheuristic with OpenMP*, in *Proc. of the 3rd European Workshop on OpenMP*, 2001.

[15] A. Delévacq, P. Delisle, M. Gravel, and M. Krahecki, *Parallel Ant Colony Optimization on Graphics Processing Units*, in *Proc. of the International Conference on Parallel Distributed Processing Techniques and Applications*, 2010, pp. 196–202.

[16] K. Kobashi, A. Fujii, T. Tanaka, and K. Miyoshi, *Acceleration of Ant Colony Optimization for the Traveling Salesman Problem on a GPU*, in *Proc. of the IASTED International Conference Parallel and Distributed Computing and Systems*, December, 2011, pp. 108–115.

[17] J.M. Cecilia, J.M. García, A. Nisbet, M. Amos, and M. Ujaldón, *Enhancing data parallelism for ant colony optimization on GPUs*, Journal of Parallel and Distributed Computing (2012).

[18] M. Dorigo and T. Stützle, *Ant Colony Optimization*, A Bradford Book, 2004.

[19] A. Lipowski and D. Lipowska, *Roulette-wheel selection via stochastic acceptance*, Physica A: Statistical Mechanics and its Applications 391 (2011), pp. 2193–2196.

[20] G. Gutin, A. Yeo, and A. Zverovich, *Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP*, Discrete Applied Mathematics 117 (2002), pp. 81–86.

[21] NVIDIA Corp., *NVIDIA CUDA C PROGRAMMING GUIDE VERSION 5.0* (2012).

[22] NVIDIA Corp., *CUDA C BEST PRACTICE GUIDE VERSION 5.0* (2012).

[23] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, *Implementations of a*

parallel algorithm for computing Euclidean distance map in multicore processors and GPUs, International Journal of Networking and Computing 1 (2011), pp. 260–276.

[24] NVIDIA Corp., *CUDA TOOLKIT 5.0 CURAND GUIDE* (2012).

[25] H. Nguyen, *GPU Gems 3*, Addison-Wesley Professional, 2007.

[26] B. Schlegel, R. Gemulla, and W. Lehner, *k-Ary Search on Modern Processors*, in *Proc. of the Fifth International Workshop on Data Management on New Hardware*, 2009, pp. 52–60.

[27] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, *Implementations of Parallel Computation of Euclidean Distance Map in Multicore Processors and GPUs*, in *Proc. of International Conference on Networking and Computing*, 2010, pp. 120–127.

[28] G. Reinelt, *TSPLIB–a traveling salesman problem library*, ORSA Journal on Computing 3 (1991), pp. 376–384.