An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem

Akihiro Uchida, Yasuaki Ito, Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima, 739–8527 Japan
{uchida, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—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. 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. The experimental results on NVIDIA GeForce GTX 580 show that our implementation for 1002 cities runs in 8.71 seconds, while a conventional CPU implementation runs in 381.95 seconds. Thus, our GPU implementation attains a speed-up factor of 43.47.**

***Index Terms*—Ant Colony Optimization, Traveling Salesman Problem, GPU, CUDA, Parallel Processing**

I. 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], [3].

Ant colony optimization (ACO) was introduced as a nature-inspired meta-heuristic for the solution of combinatorial optimization problems [4], [5]. 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 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. Indirect communication between the ants via pheromone trails makes them possible to find shortest paths between their nest and food sources. In ACO, the characteristic of real ant colonies is exploited in simulated ant colonies to solve problems. The generic ACO algorithm consists of the following two steps:

Step 1: Initialization
- Initialize the pheromone trail

Step 2: Iteration
- For each ant repeat until stopping criteria
  - 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 update 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 stopping criteria.

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 [4], [5]. 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 [6]. 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 [7]. 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 *traveling salesman problem* (TSP) [8] on the GPU. In TSP, a salesman visits n cities, and makes a tour visiting each city exactly once to try 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

978-0-7695-4893-7/12 $26.00 © 2012 IEEE
DOI 10.1109/ICNC.2012.22

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. In this paper, 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 a tour T, TSP is to find a tour 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 [9]. Delisle *et al.* have proposed an efficient and straightforward OpenMP implementation with the multiprocessor system [10]. Also, GPU implementations have been proposed. In [11], a GPU implementation of MMAS is shown. Kobashi *et al.* have shown a GPU implementation of AS [12]. The implementation introduces nearest neighbor technique to reduce the computing time of tour construction. Cecilia *et al.* have proposed a GPU implementation of AS [13]. To reduce the computing time of tour construction on the GPU, instead of the ordinary roulette-wheel selection used when ants select a next city to visit, they introduced an alternative method, called *I-Roulette*. The 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 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 the next city to visit. Using the stochastic trial, most prefix sum computation that is performed in the roulette-wheel selection can be omitted. Since the computing time of the prefix sum 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. The solution obtained by our implementation is basically the same as that by the original AS for the TSP. 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, that repeats tour construction and pheromone update 100 times, in 8.71 seconds, while a conventional CPU implementation runs in 381.95 seconds. Thus, our GPU implementation attains a speed-up factor of 43.47 over the conventional CPU implementation.

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

II. Ant Colony Optimization for the Traveling Salesman Problem

In this section, we describe a solution for TSP with ant colony optimization. Specially, we explain an algorithm solving this problem by ant system (AS). Recall that in TSP, a salesman visits n cities. and the salesman makes a tour visiting each city exactly once to try to find the shortest possible tour. In AS for TSP, ants are used as agents that perform distributed search. Each ant visits each city exactly once, ending up back at the starting city and then offers the tour as its solution. Each ant has the following characteristic:

- An ant selects which city to visit, using a transition rule that is a function of the distance to the city and the quantity of pheromone present along the connecting path.
- Transitions to already visited cities are added to a *visited list* and not allowed.
- When a tour is complete, the ant deposits a pheromone trail along paths visited in the tour.

Using the characteristic of ants, AS performs the following three steps; (i) *initialization*, (ii) *tour construction* and (iii) *pheromone update*. First of all, initialization is performed, and tour construction and pheromone update are repeated until stopping criteria. Given n cities, the distances between the cities, and m ants, the details of these three steps are spelled out as follows.

A. Initialization

In the initialization step, the initial quantities of all the pheromone trail are determined using the greedy manner [14] as follows:

$$\tau(i,j) = \frac{n}{C_g} \quad \forall (i,j) \in L, \tag{1}$$

where L denotes all edges between cities and C_g is the total length of a tour obtained by the greedy algorithm such that starting from an arbitrary city as current city, the shortest edge that connects current city and an unvisited city is selected. The quantities of pheromone assigned to each edge between two cities are initially set to a reciprocal of the average of C_g.

B. Tour construction

In tour constriction, m ants independently visit each city exactly once. Each ant starts at a city decided randomly, and selects which city to visit probabilistically. A probability

$p_k(i,j)$ to visit city j from city i for ant k is computed by Eq. (2).

$$p_k(i,j) = \begin{cases} \frac{f(i,j)}{\sum_{l \in N_k(i)} f(i,l)} & \texttt{if } j \in N_k(i) \\ 0 & \texttt{otherwise}, \end{cases} \quad (2)$$

where $N_k(i)$ is a set of unvisited adjacent cities for ant k in city i, and $f(i,j)$ is a fitness between cities i and j

$$f(i,j) = [\tau(i,j)]^{\alpha}[\eta(i,j)]^{\beta}, \quad (3)$$

where $\tau(i,j)$ denotes a quantity of pheromone between cities i and j, $\eta(i,j)$ represents heuristic information which is a reciprocal of the distance between cities i and j, and α and β control the relative influence of pheromone versus distance. These equations mean that when the quantity of pheromone between cities i and j is large and the distance between cities i and j is short, the probability to visit city j becomes large. Using this probability, each ant visits each city exactly once, ending up back at the starting city. The method such that ants select which city to visit using the above probability is well-known as *roulette-wheel selection* [15]. Visiting cities with the roulette-wheel selection, each ant constructs a tour.

C. Pheromone update

When all the ants complete tour construction, the pheromone assigned between cities is updated using information of each tour. The update consists of pheromone evaporation and pheromone deposit.

Pheromone evaporation is utilized to avoid falling into local optima. Every quantity of pheromone is reduced with the following equation;

$$\tau(i,j) \leftarrow (1-\rho)\tau(i,j) \quad \forall (i,j) \in L, \quad (4)$$

where ρ is an evaporation rate of pheromone.

After the pheromone evaporation, for every pheromone between cities, pheromone deposit is performed with the results of the tour construction as follows;

$$\tau(i,j) \leftarrow \tau(i,j) + \sum_{k=1}^{m} \Delta\tau_k(i,j) \quad \forall (i,j) \in L, \quad (5)$$

where $\Delta\tau_k(i,j)$ is a quantity of pheromone between cities i and j which is deposited by ant k. The quantity is computed by

$$\Delta\tau_k(i,j) = \begin{cases} \frac{1}{C_k} & \texttt{if } e_{i,j} \in T_k \\ 0 & \texttt{otherwise}, \end{cases} \quad (6)$$

where C_k is the tour length of ant k, and T_k is the tour of ant k. This equation means that when an edge is included in shorter tours and is selected by more ants in the tour construction, the quantity of additional pheromone is larger.

III. Compute Unified Device Architecture (CUDA)

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [16]. 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 [17], [18]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory. Figure 1 illustrates the CUDA hardware architecture.

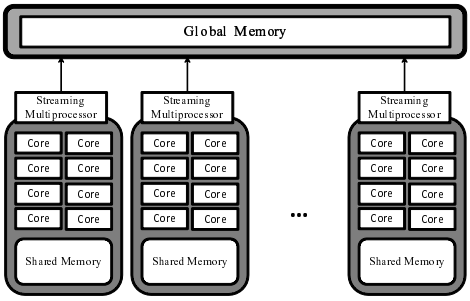

Fig. 1. CUDA hardware architecture

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 1, 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 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.

As we have mentioned, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 2, 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 in the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed in 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.

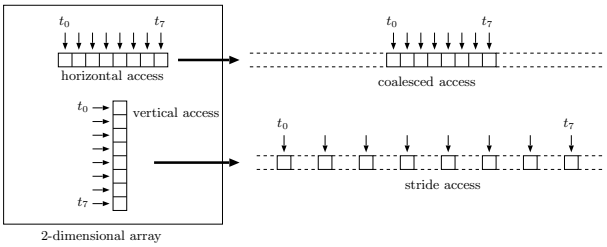

Fig. 2. Coalesced and stride access

Just as the global memory is divided into several partitions, shared memory is also divided into 16 (or 32) equally-sized modules of 32-bit width, called banks (Figure 3). In the shared memory, the successive 32-bit words are assigned to successive banks. To achieve maximum throughput, concurrent threads of a thread 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.

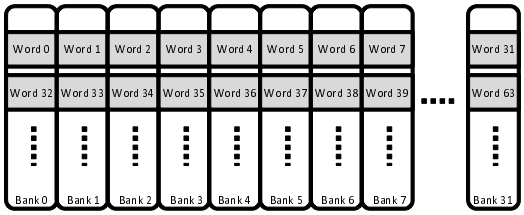

Fig. 3. The structure of the shared memory

IV. GPU Implementation

The main purpose of this section is to show a GPU implementation of AS for TSP. The ideas of our implementation to consider programming issues of the GPU system such as coalesced access of global memory and shared memory bank conflicts in Section III. Given n coordinates (x_i, y_i) of city 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 describe the details of them as follows.

A. Initialization

This part is an initialization process for the followings. Given n coordinates (x_i, y_i) of city i, each distance $d(i,j)$ between cities i and j and initial values of pheromone $\tau_{i,j}$ in Eq. (1) are computed. Also, initializing random seeds for CURAND used in the following process is performed. CURAND is a library that provides a pseudorandom number generator on the GPU by NVIDIA [19].

B. Tour construction

Recall that in the tour constriction, m ants are initially positioned on n cities chosen randomly. Each ant makes a tour with roulette-wheel selection independently. Whenever each ant visits a city, it determines which city to visit with roulette-wheel selection. To perform the tour construction on the GPU, we consider four methods, *SelectionWithoutCompression*, *SelectionWithCompression*, *SelectionWithStochasticTrial*, and a hybrid method that is a combination of the above methods. Let us consider the case when ant k is in city i. In advance, the fitness values $f(i,j)$ $(0 \leq i,j \leq n-1)$ are computed by Eq. (3) and stored to the 2-dimensional array in the global memory. Also, the elements related to city i, i.e., $f(i,0),\ldots,f(i,n-1)$, are stores in the same row so that the access to the elements can be performed with coalesced access. In the tour construction, ant k $(0 \leq k \leq m-1)$ makes a tour index array t_k such that element $t_k(i)$ stores the index of the next city from city i shown in Figure 4.

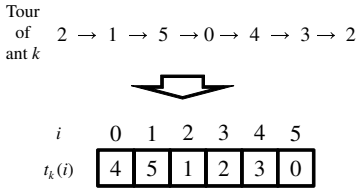

Fig. 4. Representation of tour list

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

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

To perform the roulette-wheel selection, when ant k is in city i. we compute as follows;

Step 1: Calculate the prefix sums $q_j (0 \leq j \leq n-1)$ of the fitness values for adjacent cities and a sentinel q_{-1} such that

$$q_j = \begin{cases} \sum_{s=0}^{j} f(i,s) \cdot u_j & 0 \leq j \leq n-1 \\ 0 & j = -1. \end{cases} \tag{8}$$

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

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

Figure 5 shows a summary of SelectionWithoutCompression. In Step 1, values $\tau(i,j)$ and $\eta(i,j)$ $(0 \leq j \leq n-1)$ to compute

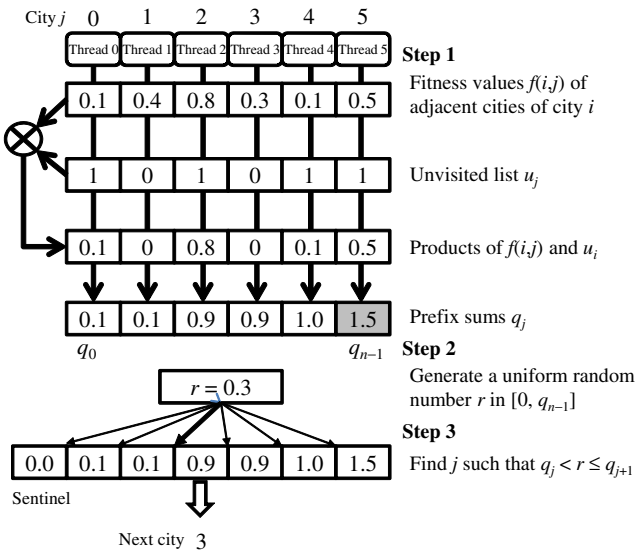

Fig. 5. Parallel roulette-wheel selection in SelectionWithoutCompression

the fitness function in Eq. (3) are read from the global memory by threads with coalesced access and stored to the shared memory. After that, 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 sum computation is performed using the parallel prefix sum algorithm proposed by Harris *et al.* [20], Chapter 39. It is an in-place parallel prefix sum 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 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 uniform random number r in $[0, q_{n-1}]$ is generated with by CURAND. Using the random number by CURAND, an index j such that $q_{j-1} < r \leq q_j$ is searched and city j is the next city to visit. In the search, we use a parallel search method based on the parallel K-ary search [21]. The idea of the parallel K-ary search is that a search space in each iteration 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 coalesced access. Although the number of unvisited cities is smaller, in every selection to determine the next city to visit, 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 When the number of unvisited cities is smaller, computing time is not reduced. In other words, it does not depend on the number of visited cities.

2) SelectionWithCompression: 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 6). Using the index array of unvisited cities, it is not necessary to read the data related to the visited cities to compute the prefix sums in Eq. (8) though SelectionWithoutCompression requires data related to both visited and unvisited cities. Therefore, when the number of unvisited cities is smaller, the computing time becomes shorter. However, the global memory access necessary to compute the prefix sums may not be done with 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 SelectionWithCompression is perhaps slower than that of SelectionWithoutCompression.

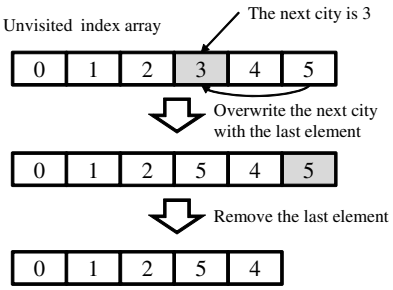

Fig. 6. Update of the unvisited index array when city 3 is selected as the next city.

3) SelectionWithStochasticTrial: In the above two methods, whenever each ant visits a city, the prefix sum calculation has to be performed. The prefix sum calculation occupies the most of the computing time of the tour construction. The idea of this method is to avoid the prefix sum calculation 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 for each city

are calculated such that

$$q'(i,j) = \begin{cases} \sum_{s=0}^{j} f(i,s) & 0 \le i,j \le n-1 \\ 0 & j = -1, \end{cases} \quad (9)$$

where all the cities have been unvisited, i.e., $u_j = 1$ $(0 \le j \le n-1)$ in Eq. (8). The results are stored to the 2-dimensional array in the global memory such that the prefix sums for city i to each city, $q'(i,0), \ldots, q'(i,n-1)$, are stored to the same row to be read with coalesced access. When an ant is in city i, to select the next city, the following steps are repeated until the next city is determined or the number of the iteration exceeds w.

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

Step 2: Find j such that $q'(i,j-1) < r \le q'(i,j)$ $(0 \le j \le n-1)$. If city j is unvisited, it is selected as the next city. If not, these steps are performed again.

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. If the next city is not determined after the w-time iteration, the next city is selected by SelectionWithoutCompression. These steps are similar to the roulette-wheel selection in the above methods. The difference point is that it 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*. SelectionWithStochasticTrial repeats the stochastic trial at most w times. If the next city cannot be determined, it is selected by SelectionWithoutCompression. 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 sum 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 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 SelectionWithoutCompression performed when the next city cannot be determined. In Section V, we will obtain the optimal times w by experiments.

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

C. Pheromone update

In the followings, we show a GPU implementation of pheromone update. Recall that pheromone update consists of pheromone evaporation and pheromone deposit. In our implementation, the values of pheromone $\tau(i,j)$ $(0 \le i \le j \le n-1)$ 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 i, i.e., $\tau(i,0), \tau(i,1), \ldots, \tau(i,n-1)$, are stores in the same row so that the access to the elements can be performed with coalesced access. Our implementation consists of two kernels, *PheromoneUpdateKernel* and *SymmetrizeKernel*.

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 7 shows a summary of the pheromone update on the GPU for a block that perform 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, pheromone evaporation is preformed, i.e., each value is reduced by Eq. (4) by threads in parallel. To perform 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. (6) 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 [16]. After the addition, the values of pheromone are stored back to the global memory. Note that since the 2-dimensional array that stores 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.

2) SymmetrizeKernel: This kernel symmetrizes the array for the results of 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 pheromone values is divided into subarrays whose size is 32×32. We assign one block to two subarrays that are symmetric or one subarray that includes symmetric element. 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 [22]. The method transposes a 2-dimensional data stored in the global memory via the shared

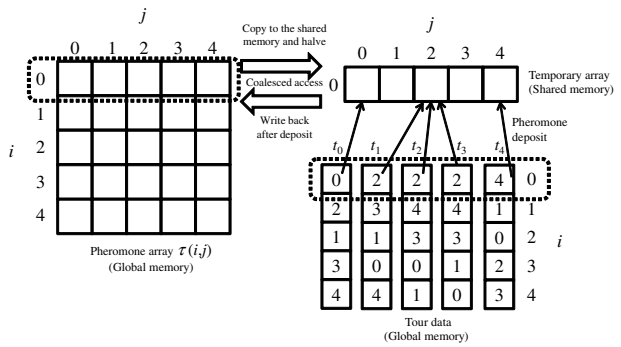

Fig. 7. A summary of PheromoneUpdateKernel

memory with coalesced access and avoidance of bank conflict on the GPU. 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.

V. Performance Evaluation

We have implemented our AS for the TSP using CUDA C. We have used 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 conventional software approach of AS for the TSP using GNU C. 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 [23]. In the following evaluation, we utilize 8 instances: *d198, a280, lin318, pcb442, rat783, pr1002,* and *pr2392* 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, and ρ in Eq. (3) and Eq. (4), are set to 1.0, 2.0, and 0.5, respectively. Also, the number of used ants m is set to the number of cities. Those parameters are recommended in [24]. 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 tour construction and pheromone update, and then the results of overall performance are shown.

A. Evaluation of tour construction

Before performance of 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 the next city is not determined in SelectionWithStochasticTrial. The other is timing when SelectionWithStochasticTrial is switched to SelectionWithCompression in the hybrid method.

To obtain an optimal upper limit of iteration of the stochastic trial if the next city is not determined in SelectionWithStochasticTrial, we evaluated the number of times necessary to determine the next city in a tour construction for the pheromone values obtained after the tour construction and pheromone update were repeated 100 times for pr1002. Figure 8 is a graph that shows a histogram of the number of city 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 about 25 and the percentage of cities is about 84%. This means that in about 500 cities, the next city was determined by the stochastic trial 5 times and in 84% 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 cannot be determined when the stochastic trial has to be repeated more than 2000 times. Considering the balance of computing time between the stochastic trial and SelectionWithoutCompression when the next city cannot be determined, in the following experiments, we set 8 times to the upper limit of times of iteration.

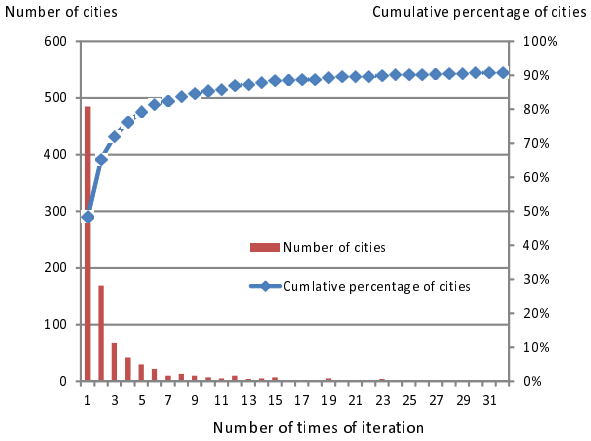

Fig. 8. A histogram of the number of city and its cumulative histogram of the percentage of cities to the number of times of iteration of the stochastic trial

To obtain the timing when SelectionWithStochasticTrial is switched to SelectionWithCompression in the hybrid method, we measured the computing time of tour construction for various percentages when SelectionWithStochasticTrial is switched to SelectionWithCompression. Figure 9 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. According to Figure 9, computing time is minimized by switching the method when about 85% cities are visited. Therefore, we switch from SelectionWithStochasticTrial to SelectionWithCompression when 85% cities

are visited.

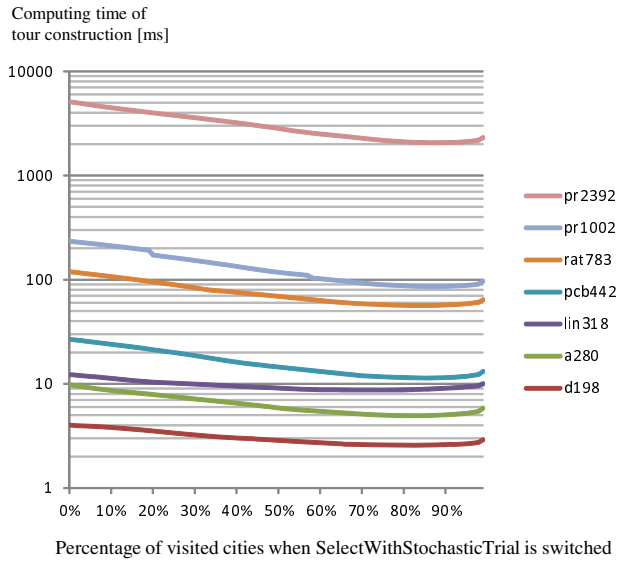

Fig. 9. Computing time of tour construction for the various percentages when SelectionWithStochasticTrial is switched to SelectionWithoutCompression for pr1002 with various w

To compare the performance among our proposed tour construction methods, we have evaluated the computing time of them. Table I shows the computing time of tour construction with various methods for pr1002. The computing time of SelectWithCompression is a little shorter than that of SelectWithoutCompression. Since the computing time of SelectWithCompression becomes shorter when the number of unvisited cities is small, the total computing time becomes shorter. Compared to the methods without the stochastic trial, the computing time of the methods with the stochastic trial is approximately halved. In addition, the computing time of the hybrid method is approximately 10% shorter than that of SelectWithStochasticTrial.

TABLE I
COMPUTING TIME OF TOUR CONSTRUCTION FOR PR1002

Tour construction method	Time[ms]
SelectWithoutCompression	235.43
SelectWithCompression	217.94
SelectWithStochasticTrial	96.37
Hybrid method	86.43

Table II shows the computing time of tour construction for various instances. From 198 to 1002 cities, when the number of cities is larger, the speed-up factor is larger. However, The speed-up factor for 2392 cities is smaller than that for 1002 cities. This is because in the parallel prefix sum computation shown in Section IV can be performed only for power of two numbers. Therefore, for the instance of which number of cities is 2392, the parallel prefix sum computation for 4096 elements must be performed. Therefore, approximate half of elements are redundant. Since in CPU implementation, the redundant elements are not necessary to compute the prefix sum, the computing time of GPU implementation becomes longer, that is, the speed-up factor becomes smaller.

TABLE II
COMPUTING TIME OF TOUR CONSTRUCTION FOR VARIOUS INSTANCES

Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	19.84	2.58	7.69
a280 (280)	47.05	4.95	9.50
lin318 (318)	95.15	8.86	10.74
pcb442 (442)	180.61	11.35	15.92
rat783 (783)	1215.31	56.38	21.56
pr1002 (1002)	3784.43	86.43	43.79
pr2392 (2392)	58452.20	2078.98	28.12

B. Evaluation of pheromone update

Table III shows the computing time of pheromone update for various instances. The computing time of both the CPU and GPU implementation is Our GPU implementation can achieve speed-up factors of 22 to 67. Compared to the computing time of tour construction, the computing time of pheromone update is much shorter.

TABLE III
COMPUTING TIME OF PHEROMONE UPDATE

Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	0.963	0.036	26.64
a280 (280)	1.384	0.060	22.92
lin318 (318)	2.797	0.070	39.88
pcb442 (442)	4.692	0.113	41.43
rat783 (783)	16.770	0.320	52.37
pr1002 (1002)	34.877	0.520	67.08
pr2392 (2392)	222.762	5.411	41.17

C. Evaluation of overall performance

Table IV shows overall performance that is the total computing time of AS for various instances. Each execution includes the initialization and 100 times iteration of tour construction and pheromone update. Since the computing time of tour construction is much larger than other process, each speed-up factor is similar to that of tour construction. Our GPU implementation can achieve speed-up factors of 7.52 to 43.47 over the CPU implementation.

In the related works of ACO for TSP shown in Section I, several GPU implementations have been proposed. Since those implemented methods, used instance, and utilized GPUs differ, we cannot directly compare our implementation with them. However, three implementations proposed in papers [11], [12], [13] achieved the maximum speed-up factor of 23.9, 23.5, and 20.0 over their CPU implementations, respectively. Since the speed-up factor we achieved is 43.47, our GPU implementation is more effective than them.

TABLE IV
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)	2080.72	263.91	7.52
a280 (280)	4844.59	505.51	9.31
lin318 (318)	9797.03	897.29	10.61
pcb442 (442)	18534.37	1153.95	15.66
rat783 (783)	123220.58	5673.15	21.43
pr1002 (1002)	381949.72	8706.32	43.47
pr2392 (2392)	5867605.87	208478.18	28.04

VI. 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 coalesced access of global memory and 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, that repeats tour construction and pheromone update 100 times, in 8.71 seconds, while a conventional CPU implementation runs in 381.95 seconds. Thus, our GPU implementation attains a speed-up factor of 43.47.

Future works include GPU implementations for various algorithms of ant colony optimization such as MMAS, ACS, and AS with the idea of nearest neighbor to obtain further acceleration and accuracy. In addition to TSP, other combinatorial optimization problems such as the quadratic assignment problem, etc. are applied by our method.

REFERENCES

[1] NVIDIA Corp., "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] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, 1992.

[5] 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*, vol. 26, no. 1, pp. 29–41, 1996.

[6] T. Stützle and H. H. Hoos, "MAX–MIN ant system," *Future Generation Computer Systems*, vol. 16, no. 8, pp. 889–914, 2000.

[7] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, April 1997.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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 Distribited Computing and Systems*, December 2011, pp. 108–115.

[13] 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.

[14] G. Gutin, A. Yeo, and A. Zverovich, "Traveling salesman shoud not be greedy: domination analysis of greedy-type heuristics for the tsp," *Discrete Applied Mathematics*, vol. 117, pp. 81–86, 2002.

[15] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Machanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, March 2011.

[16] NVIDIA Corp., *NVIDIA CUDA Programming Guide Version 4.1*, 2011.

[17] ——, *CUDA C Best Practice Guide Version 4.1*, 2012.

[18] 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*, vol. 1, no. 2, pp. 260–276, 2011.

[19] NVIDIA Corp., *CUDA Toolkit 4.1 CURAND Guide*, 2011.

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

[21] 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.

[22] 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.

[23] G. Reinelt, "Tsplib–a traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, pp. 376–384, 1991.

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