Accelerating Ant Colony Optimization for the Vertex Coloring Problem on the GPU

Ryouhei Murooka, Yasuaki Ito, and Koji Nakano
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan
Email: {murooka, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—Vertex coloring is an assignment of colors to vertex of an undirected graph such that no two vertices sharing the same edge have the same color. The vertex coloring problem is to find the minimum number of colors necessary to color a graph given, which is an NP-hard problem in combinatorial optimization. Ant Colony Optimization (ACO) is a well-known meta-heuristic in which a colony of artificial ants cooperates in exploring good solutions to a combinatorial optimization problem. Several methods applying ACO to the vertex coloring problem have been proposed. The main contribution of this paper is to propose a GPU implementation to accelerate the computation of the ACO algorithm for the vertex coloring problem. In our implementation, we have considered programming issues of the GPU architecture, such as coalescing access of the global memory, bank conflict of the shared memory, etc. The experimental results show that on NVIDIA GeForce GTX 1080, our implementation for 1000 vertices runs in 2.740s, while the CPU implementation on Intel Core i7-4790 runs in 100.866s. Thus, our GPU implementation attains a speed-up factor of 36.81.**

***Keywords*-GPU; CUDA; Vertex coloring problem; Ant colony optimization**

I. Introduction

Vertex coloring is an assignment of labels called *colors* to vertex of a graph. Given an undirected graph without self-loops, colors are assigned to vertices such that no two vertices sharing the same edge have the same color as illustrated in Figure 1. *The vertex coloring problem* is to find the minimum number of colors needed to color a given graph. This problem is well-known as an NP-hard problem in combinatorial optimization. Therefore, approximation algorithms for this problem have been proposed, such as greedy algorithms [1]–[4], tabu search [5], genetic algorithm [6], ant colony optimization [7]–[9], among others. Also, parallel algorithms have been introduced [10]–[12].

Ant colony optimization (ACO) was introduced as a nature-inspired meta-heuristic for the solution of combinatorial optimization problems [13], [14]. 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 of the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of the 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 genetic ACO algorithm consists of the following two steps:

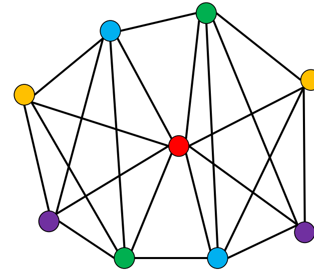


Figure 1: An example of vertex coloring

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 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 stopping criteria.

GPUs (Graphics Processing Units) are specialized microprocessors that accelerate graphics operations. Many processing units in recent GPUs can be used for general purpose parallel computation. CUDA (Compute Unified Device Architecture) [15], [16] is an architecture for general purpose parallel computation on NVIDIA's GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many researches have been devoted using CUDA [17], [18].

The main contribution of this paper is to accelerate the ACO for the vertex coloring problem using the GPU. Espe-

cially, we have implement the ACO based on the Recursive Largest First method (RLF) proposed by Costa *et al.* in [7] on the GPU. In our GPU implementation, we have considered many programming issues of the GPU architecture such as coalescing access of the global memory, bank conflict of the shared memory, etc. We note that our goal in this paper is to accelerate the computation of the ACO 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 ACO for the vertex coloring problem. We have implemented our parallel algorithm in NVIDIA GeForce GTX 1080. The experimental results show that our proposed GPU implementation can run 19.68 to 36.81 times faster than the sequential CPU implementation, where the solutions of the both implementation are almost the same.

This paper is organized as follows; Section II introduces the ACO algorithm for the vertex coloring problem. Section III briefly describes about GPUs and CUDA architecture. In Section IV, our GPU implementation of the ACO algorithm for the vertex coloring problem is proposed. Experimental results are shown in Section V. Finally, Section VI concludes the paper.

II. Ant Colony Optimization for the Vertex Coloring Problem

In this section, we describe a solution for the vertex coloring problem using the ACO. There are several approaches for the vertex coloring problem using the ACO [7]–[9]. Especially, in this work, we use ACO based on *the Recursive Largest First method* (RLF) [19] which is one of the greedy algorithms [7]. We show the RLF method first, then the algorithm of the ACO based on the RLF method.

A. The Recursive Largest First method

The Recursive Largest First (RLF) method is a heuristic algorithm for the vertex coloring problem [1]. Given a graph $G = (V, E)$ with V the vertices set, and E the edge set, this algorithm sequentially finds a sequence of stable sets C_k $(k = 0, 1, \dots)$ each of which is assigned color k. The number of stable sets corresponds to the number of colors necessary to be assigned. The algorithm of the RLF method is shown in Algorithm 1, where U denotes the set of uncolored vertices.

Algorithm 1 The RLF method

Input: A graph $G = (V, E)$
Output: A coloring of V
1: $U \leftarrow V$
2: $k \leftarrow 0$
3: **while** $U \neq \phi$ **do**
4: $\quad C_k \leftarrow$ FindStableSet(U)
5: $\quad U \leftarrow U \setminus C_k$
6: $\quad$ Assign C_k color k
7: $\quad k \leftarrow k + 1$
8: **end while**

To explain the algorithm, in the following, we describe how a stable set C_k is built when $C_0, \dots, C_{k-1}$ have been found and have assigned color $0, \dots, k-1$, respectively. Let N and W denote the set of uncolored vertices neighboring C_k and a candidate set of vertices that can be colored in U, respectively. Note that C_k and N are initially empty and W is initialized to U. In the algorithm, vertices are selected from W one by one. Whenever a vertex is selected, the vertex and its neighbors are removed from W, and the neighbors are added to N. The algorithm first selects a vertex $v \in W$ which has the largest number of neighbors in W. The vertex v is added to C_k, and W and N are updated. After that, the rest of vertices in W is found by the following selection is repeated until W is empty: the next vertex in W is selected such that it has the largest number of neighbors in N. The vertex is added to C_k, and W and N are updated. When W becomes empty, the next stable set C_{k+1} is built from U. Algorithm codes of the above operation is shown in Algorithm 2, where $d_U(w)$ and $d_N(w)$ denote the number of neighbors of w in U and N, respectively. Figure 2 shows an example of finding a stable set by Algorithm 2. Due to the page limitation, the explanation of the figure is omitted, but the reader can easily understand it.

Algorithm 2 Procedure: FindStableSet

Input: A set of uncolored vertices U
Output: A stable set C_k
1: $N \leftarrow \phi$
2: $C_k \leftarrow \phi$
3: $W \leftarrow U$
4: $v \leftarrow \arg\max_{w \in W} d_U(w)$
5: Add v to C_k
6: Add all neighbors in W of v to N
7: Remove v and its neighbors from W
8: **while** $W \neq \phi$ **do**
9: $\quad v \leftarrow \arg\max_{w \in W} d_N(w)$
10: $\quad$ Add v to C_k
11: $\quad$ Add all neighbors in W of v to N
12: $\quad$ Remove v and its neighbors from W
13: **end while**
14: Return C_k

B. ACO algorithm based on the RLF method

We review the ACO algorithm based on the RLF method proposed in [7]. Recall that in the vertex coloring problem, given a graph $G = (V, E)$ with V the set of n vertices $v_0, v_1, \dots, v_{n-1}$, colors are assigned to vertices. In this method, pheromone values are defined between every pair of two vertices. Let $\tau_{i,j}$ be a pheromone value between v_i and v_j. Note that $\tau_{i,j}$ and $\tau_{j,i}$ take the same variable. In the

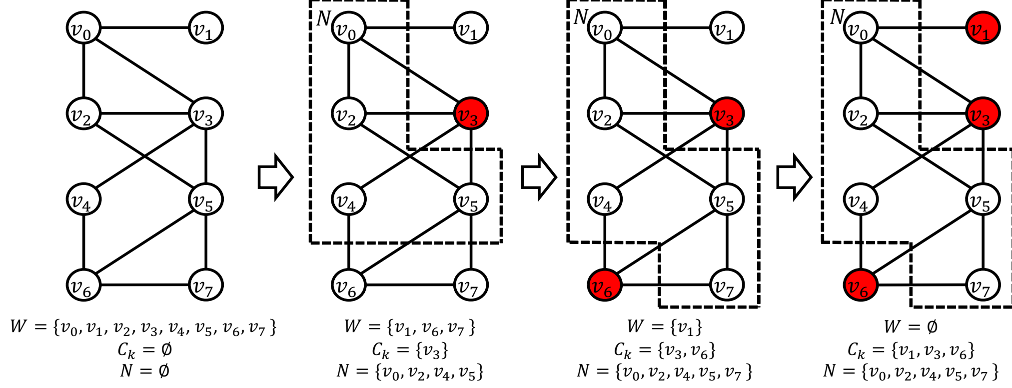


Figure 2: An example of finding a stable set in the RLF method

algorithm, when the value of pheromone $\tau_{i,j}$ is larger, two vertices v_i and v_j are assigned the same color with higher probability. Based on the pheromone, ants work as agents performing the distributed search. Recommended in [7], first, a number of ants are placed at vertices independently at random. Each ant builds a stable set by visiting vertices and its vertices are assigned a same color. The ant repeatedly selects a next visiting vertex at random among the candidate vertices that can be colored currently. In the selection of a next visiting vertex, the next vertex is stochastically selected by following the pheromone values. Since the pheromone values of adjacent two vertices are zero, neighboring vertices are not selected in the selection. The ant assigns the color to the selected vertex and repeats this selection until no vertices are selected. After that, the ant restart visiting vertices to build a stable set with a new color for uncolored vertices. When all the vertices are colored, the number of colors is a solution of this problem. Thus, after all m ants color all vertices, we have m solutions of this problem. This procedure is repeated until some termination condition is satisfied.

In the following, we explain the details of the algorithm consisting of three steps *the initialization*, *the tour construction*, and *the pheromone update*.

Initialization: As shown in the above, every pair of two vertices has a pheromone value. To avoid assigning the same color to adjacent vertices, every value of neighboring two vertices is set to zero. Thus, the pheromone values are initialized as follows;

$$\begin{aligned} \tau_{i,j} &= 1 \qquad \text{if } e_{i,j} \notin E \\ &= 0 \qquad \text{otherwise.} \end{aligned}$$

Tour construction: Each ant basically performs the RLF method in Algorithm 1 independently. The difference is the selection of vertices in Algorithm 2. More specifically, to find a stable set, in Algorithm 2, a first vertex and next visiting vertices are selected in lines 4 and 9, respectively. On the other hand, in the ACO algorithm, a first vertex is selected from uncolored vertices at random. Also, a next vertex is selected by a well-known method called *roulette-wheel selection* [20]. Algorithm codes of procedure of finding a stable set in the ACO algorithm is shown in Algorithm 3.

Algorithm 3 Procedure: FindStableSet in the ACO

Input: A set of uncolored vertices U
Output: A stable set C_k
1: $N \leftarrow \phi$
2: $C_k \leftarrow \phi$
3: $W \leftarrow U$
4: v is selected from W at random
5: Add v to C_k
6: Add all neighbors in W of v to N
7: Remove v and its neighbors from W
8: **while** $W \neq \phi$ **do**
9: v is selected from W by the roulette-wheel selection
10: Add v to C_k
11: Add all neighbors of v to N
12: Remove v and its neighbors from W
13: **end while**
14: Return C_k

We focus on a particular ant and explain how it traverses vertices using the roulette-wheel selection. More specifically, we show how the ant selects a next visiting vertex from W when the ant in v_i assigns vertices color k. We select a next visiting vertex in W with *the fitness* $f_{i,j}$ between v_i and v_j with respect to the ant by the following formula:

$$f_{i,j} = (d_N(v_j))^\alpha \cdot (\tau_{i,j})^\beta \tag{1}$$

where $d_N(v_j)$ denote the number of neighbors of v_j in N. Also, $\alpha > 0$ and $\beta > 0$ are fixed values to control the influence of the pheromone.

We select a next visiting vertex in W with probability proportional to $f_{i,j}$. In other words, the probability $p_{i,j}$ to

select vertex v_j as a next visiting vertex is

$$p_{i,j} = \frac{f_{i,j}}{F} \quad \text{if } v_j \in W \\ = 0 \quad \text{otherwise} \tag{2}$$

where $F = \sum_{v_j \in W} f_{i,j}$. Clearly, $\sum_{v_j \in W} p_{i,j} = 1$ and v_j is selected as a next visiting vertex with higher probability if the fitness value $f_{i,j}$ is larger. In this paper, the parameters α and β are set to 2 and 4, respectively, that are the most accurate shown in [7].

This tour construction is performed until no vertices are selected, that is, W becomes empty. After that tour construction is performed for uncolored vertices by visiting them and assigning the next color $k+1$. When all the vertices are colored by repeating the tour construction, the assignment of colors obtained by the ant is an approximation solution of the vertex coloring problem.

Pheromone update: After all m 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 vertex coloring. Every pheromone value is decreased by multiplying a fixed constant factor ρ $(0 < \rho < 1)$. More specifically, $\tau_{i,j}$ $(0 \leq i, j \leq n-1)$ is updated as follows:

$$\tau_{i,j} \leftarrow \rho \cdot \tau_{i,j}, \tag{3}$$

where ρ is a fixed evaporation rate of the pheromone determined by the experiments. In this work, we set $\rho = 0.5$ that is the optimal parameter shown in [7].

After the pheromone evaporation, the pheromone deposit is performed using the tours obtained by the m ants. Let q_a and T_a denote the number of colors and a set of tours obtained by the a-th ant $(0 \leq a \leq n-1)$, respectively. Each T_a consists q_a tours since the ant makes one tour for each color. The pheromone value $\tau_{i,j}$ is updated as follows:

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{0 \leq a \leq n-1} \left(\frac{1}{q_a} + \sum_{(v_i,v_j) \in T_a} \frac{L}{q_a} \right) \quad \text{if } e_{i,j} \notin E, \tag{4}$$

where L is a fixed value to control the quantity of deposit of the pheromone determined by the experiments.

III. GPU AND CUDA ARCHITECTURE

GPUs consist of many processing cores, called Streaming Multiprocessor (SM), and hierarchical memories. In the GPU computation, it is necessary to consider the characteristics of the memories to accelerate it. Figure 3 illustrates the hardware architecture of the GPU. The GPU has two types of memories, the global memory and the shared memory.

The global memory has a large capacity, 1.5-12 Gbytes, implemented by off-chip DRAMs. All the processing cores in the SMs can access the global memory, but its access latency is very long. In particular, we need consider *the coalesced access* of the global memory access. The continuous location in address space of the global memory are accessed at the same time by a group of threads, called *warp*. However, if threads does not access to continuous location, called *stride access*, the memory access need to be repeated several times. Therefore, from such the structure of the global memory, the coalesced access maximizes the bandwidth of memory access.

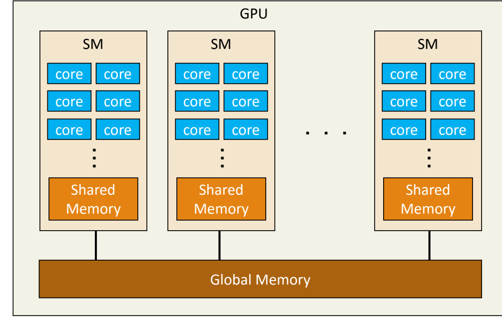


Figure 3: GPU hardware architecture

The shared memory can be used as a cache to hide the access latency of the global memory. To access to the shared memory, the structure needs be considered. The shared memory is divided into 32 equally-sized banks of 32 (or 64)-bit width. In the shared memory, the successive 32 (or 64)-bit words are assigned to successive banks. To achieve maximum throughput, threads in a warp should access different banks, otherwise, *bank conflicts* will be occurred.

IV. GPU IMPLEMENTATION

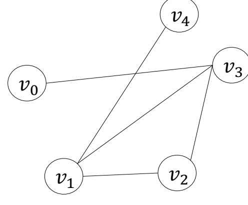
The main purpose of this section is to show a GPU implementation of ACO algorithm based on the RLF method. In the GPU implementation, as described in Section II, n ants independently color vertices by visiting vertices using pheromone. Our implementation consists of three CUDA parts, initialization, tour construction, and pheromone update. We use CURAND [21], 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.

A. Initialization

Given an input graph $G = (V, E)$, we use an adjacency matrix of size $n \times n$ to represent the graph, where n denotes the number of vertices in Figure 4. The matrix is stored in the global memory as a 2-dimensional array. In the initialization, the pheromone values are initialized using the adjacent matrix.

B. Tour construction

Recall that in the tour construction, m ants are initially positioned on m vertices randomly. We assign each ant to



(a) Input graph

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

(b) Adjacent matrix of a graph (c) Initial pheromone values

Figure 4: An example of data representation of the graph and pheromone values

one block with multiple threads, that is, m blocks are used in total. Each ant builds a sequence stable sets by visiting vertices independently. Each ant has an candidate list $W = \{w_0, w_1, \ldots, w_{n-1}\}$ such that

$$\begin{aligned} w_j = & \quad 1 \qquad \text{if vertex } v_j \text{ can be colored} \\ = & \quad 0 \qquad \text{otherwise.} \end{aligned} \tag{5}$$

The values of the list denote availability of vertices to be colored, that is, if $w_j = 0$, when an ant assigns vertices color k, vertex v_j has been colored or neighbors vertices assigned color k. When the ant visits a vertex, the vertex is colored. Therefore, whenever the ant visits a vertex, this list is updated. Additionally, sets of vertices N and C_k in Algorithm 3 are also updated for each visit.

Whenever each ant visits a vertex, it determines a next visiting vertex with the roulette-wheel selection. To perform the tour construction on the GPU, we use two selection methods; *the selection with prefix-sums*, *the selection with stochastic trial* and *the hybrid method*. In these methods, we use the basic idea proposed in [22].

Selection with prefix-sums: To perform the roulette-wheel selection, in the selection with prefix-sums, when ant a is in vertex v_i, we execute the following three steps:

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

$$q_j = \tau_{i,0} \cdot w_0 + \tau_{i,1} \cdot w_1 + \cdots + \tau_{i,j} \cdot w_j \tag{6}$$

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

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

Clearly, r is in $[q_{j-1}, q_j)$ with probability $\frac{q_j - q_{j-1}}{q_{n-1}} = \frac{\tau_{i,j} \cdot w_j}{q_{n-1}}$. Thus, if v_j is a candidate to be colored, that is, $w_j = 1$, then the next visiting vertex is v_j with probability $\frac{\tau_{i,j}}{q_{n-1}}$. Hence the next visiting vertex is selected with probability Eq. 2 correctly.

We show a parallel computation of the selection with prefix-sums, as follows. In Step 1, to compute the values $f_{i,j}$ computed by Eq. 1, the n values $\tau_{i,0}, \tau_{i,1}, \ldots, \tau_{i,n-1}$ are read from the global memory by threads with coalescing access and stored to the shared memory. After that, the prefix-sums in Eq. 6 are computed, where the fitness values of vertices that cannot be selected are 0. To avoid the branch instructions, we multiply $f_{i,j}$ and w_j with the candidate list W in Eq. 5. In our implementation, the prefix-sums computation is performed using the parallel prefix-sums algorithm proposed Harris *et al.* [23], 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.

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 selected and vertex v_j is the next visiting vertex. In this search operation, we use a parallel search method based on the parallel K-ary search using multiple threads in a block [24]. This 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 element of each partition, a partition that includes the objective element is searched is found by 32 threads. After that, the objective element is searched from the partition by threads of which the 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 coalescing access. On the other hand, in every selection to determine the next vertex, the roulette-wheel selection has to be computed for all vertices regardless of whether they can be selected or not. Namely, although the number of candidate vertices is smaller, the computing time cannot be reduced. In other words, it does not depend on the number of candidate vertices.

Selection with stochastic trial: This selection is based on the roulette-wheel selection with stochastic trial proposed in [22]. In the above selection, whenever every ant visits a city, the prefix-sums computation needs 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 this method are shown as follows.

Before ants start visiting vertices, the prefix-sums $q'_{i,0}, q'_{i,1}, \ldots, q'_{i,n-1}$ for every vertex v_i $(0 \leq i \leq n-1)$ are computed such that

$$q'_{i,j} = f'_{i,0} + f'_{i,1} + \cdots + f'_{i,j}$$

For simplicity, let $q'_{i,-1} = 0$. Note that once all $q'_{i,j}$ are computed, we do not have to update them during the tour construction. We assume that the values of $q'_{i,j}$ are stored into 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 vertex v_i, the next visiting vertex 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 vertex v_j is a candidate in W, it is selected as the next vertex. Otherwise, these steps are performed until the next vertex is selected.

In Step 2, the candidate list (Eq. 5) is used to find whether the vertex has been selected or not by the parallel search shown in the above methods. Using this idea, the number of prefix-sums computation can be reduced.

Hybrid method: In the selection with stochastic trial, when the number of candidates in W is small, Step 2 succeeds in selecting an candidate vertex with small probability. If this is the case, the number of iteration of the above steps can be large. Hence, if the next vertex is not determined in the t-time iteration for some constant value t determined the experiments, we select the next visiting vertex by the selection with prefix-sums. When the number of candidates in W is smaller of some of the fitness values of non-candidate vertices are larger, almost the trial cannot select the next vertex. However, the computing time is much shorter than that of the prefix-sums computation. Therefore, if the next vertex can be determined in the above steps within t times, the total computing time can be reduced by this method. In our implementation, we set t to 4 that is the optimal times by our experiments.

C. Pheromone update

We show a GPU implementation of the pheromone update as follows. The idea of the implementation is efficient memory access by coalescing access of the global memory. Recall that the pheromone update consists of the pheromone evaporation and the pheromone deposit. In the GPU implementation, the pheromone values $\tau_{i,j}$ $(0 \leq i, j \leq n-1)$ are stored in a 2-dimensional array, which is a symmetric array, that is, $\tau_{i,j} = \tau_{j,i}$, in the global memory. The values are updated by the results of the tour construction by ants. Arranging the array symmetric, the elements related to vertex v_i, i.e., $\tau_{i,0}, \ldots, \tau_{i,n-1}$ are stored in the same row so that the access to the elements can be performed using the coalescing access.

A kernel that performs the pheromone update assigns m blocks that consists of n threads to each row of the array, where m and n denote the number of ants and vertices, respectively. Each block performs the following operations independently. Threads in a block i $(0 \leq i \leq n-1)$ read $\tau_{i,0}, \ldots, \tau_{i,n-1}$ in the i-th row with coalescing access, and then store them to the shared memory. Each pheromone value is updated from Eqs. 3 and 4 as follows. First, all pheromone values are updated such that

$$\tau_{i,j} \leftarrow \rho \cdot \tau_{i,j} + \sum_{0 \leq a \leq n-1} \frac{1}{q_a}$$

After that, pheromone values of vertices along all the tours obtained by all ants in the tour construction is updated such that

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{(v_i, v_j) \in T_a} \frac{L}{q_a}$$

In the above updating, making every block of threads accesses coalesced, the memory access can be performed efficiently.

V. Performance Evaluation

This section shows the performance evaluation of our proposed GPU implementation of the AS for the vertex coloring problem using CUDA C. We have implemented it on NVIDIA GeForce GTX 1080 with 2560 cores running in 1.733GHz and CUDA version 7.5. For the purpose of estimating the speed-up of our GPU implementation, we have also implementation a sequential CPU implementation of the AS for the vertex coloring problem using GNU C on Intel Core i7-4790 running in 3.66GHz. In the sequential CPU implementation, we can apply the same idea of the tour construction in the GPU implementation such as the selection with stochastic trial. We have evaluated our implementations using a set of benchmark instances from the bench mark set [25]. We have used 6 instances from the set consisting of 500 and 1000 vertices each of which has distinct densities, 10%, 50%, and 90%. In this paper. we set the number of ants to 20% of the number of the vertices n, that is, 100 and 200 ants have been used for instances with 500 and 1000 vertices, respectively. Every execution repeated the process consisting of the tour construction and the pheromone 50 times.

Table I shows the computing time and solutions of the CPU and GPU implementations. Regarding solutions, i.e., the number of colors assigned, the solutions of both implementations are almost the same because the GPU implementation is accelerating the computation of the ACO algorithm. On the other hand, for the computing time, when the number of vertices is larger, the computing time is longer. Also, when the density of graph is higher, the computing time is shorter. Our proposed GPU implementation can run 19.68 to 36.81 times faster than the sequential CPU implementation.

VI. Conclusions

In this paper, we have proposed an implementation of accelerating the ant colony optimization for the vertex coloring problem using a GPU. In our implementation, we have considered programming issues of the GPU architecture such

Table I: The computing time in seconds and solutions of the ACO for the vertex color problem

instance in [25]	# vertices	# edges	CPU time	CPU # colors	GPU time	GPU # colors	speed-up
DSJC500.1.col	500	12458	12.452	18	0.464	18	26.84
DSJC500.5.col	500	62624	10.454	68	0.448	68	23.32
DSJC500.9.col	500	112437	10.232	167	0.426	167	24.03
DSJC1000.1.col	1000	49629	100.866	28	2.740	28	36.81
DSJC1000.5.col	1000	249826	84.922	121	2.642	121	32.14
DSJC1000.9.col	1000	449449	49.551	309	2.517	308	19.68

as the coalescing access of the global memory and the bank conflict of the shared memory, etc. We have implemented it on NVIDIA GeForce GTX 1080. The experimental results show that our GPU implementation attains a speed-up factor of at most 36.81 over the sequential CPU implementation on Intel Core i7-4790.

REFERENCES

[1] F. T. Leighton, "A graph coloring algorithm for large scheduling problems," *Journal of Research of the National Bureau of Standards*, vol. 84, no. 6, pp. 489–506, 1979.

[2] D. Brélaz, "New methods to color the vertices of a graph," *Communications of the ACM*, vol. 22, no. 4, pp. 251–256, 1979.

[3] H. Al-Omari and K. E. Sabri, "New graph coloring algorithms," *American Journal of Mathematics and Statistics*, vol. 2, no. 4, pp. 439–441, 2006.

[4] M. Chiarandini, G. Galbiati, and S. Gualandi, "Efficiency issues in the RLF heuristic for graph coloring," in *Proc. of Metaheuristics International Conference*, 2011, pp. 461–469.

[5] A. Hertz and D. de Werra, "Using tabu search techniques for graph coloring," *Computing*, vol. 39, pp. 345–351, 1987.

[6] C. Fleurent and J. A. Ferland, "Genetic and hybrid algorithms for graph coloring," *Annals of Operations Research*, vol. 63, pp. 437–461, 1996.

[7] D. Costa and A. Hertz, "Ants can colour graphs," *Journal of the Operational Research Society*, vol. 48, no. 3, pp. 295–305, May 1997.

[8] T. N. Bui, T. H. Nguyen, C. M. Patel, and K.-A. T. Phan, "An ant-based algorithm for coloring graphs," *"Discrete Applied Mathematics*, vol. 156, no. 2, pp. 190–200, 2008.

[9] P. Consoli, A. Collerà, and M. Pavone, "Swarm intelligence heuristics for graph coloring problem," in *Proc. of IEEE Congress on Evolutionary Computation*, 2013, pp. 1909–1916.

[10] E. G. Boman, D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, and F. Manne, "A scalable parallel graph coloring algorithm for distributed memory computers," in *Proc. of 11th International Euro-Par Conference*, 2005, pp. 241–251.

[11] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proc. of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, 2014, pp. 167–177.

[12] G. Rokos, G. Gorman, and P. H. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures," in *European Conference on Parallel Processing*, 2015, pp. 414–425.

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

[14] 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.

[15] *CUDA C Programming Guide Version 7.0*, NVIDIA Corporation, 2015.

[16] *CUDA C Best Practice Guide Version 7.0*, NVIDIA Corporation, 2015.

[17] Y. Ito and K. Nakano, "A GPU implementation of dynamic programming for the optimal polygon triangulation," *IEICE Transactions on Information and Systems*, vol. E96-D, no. 12, pp. 2596–2603, 2013.

[18] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[19] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning," *Operations Research*, vol. 39, no. 3, pp. 378–406, 1991.

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

[21] NVIDIA Corp., *cuRAND Library PG-05328-050 v7.5*, 2015.

[22] A. Uchida, Y. Ito, and K. Nakano, "Accelerating ant colony optimisation for the travelling salesman problem on the GPU," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 4, pp. 401–420, 2014.

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

[24] 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.

[25] T. H. Nguyen and T. Bui, "Graph coloring benchmark instances," https://turing.cs.hbg.psu.edu/txn131/graph-coloring.html.