Fast QAP Solving by ACO with 2-opt Local Search on a GPU

Shigeyoshi Tsutsui  
Management and Information Sciences  
Hannan University  
Matsumura, Osaka 580-8502, Japan  
Email: tsutsui@hannan-u.ac.jp

Noriyuki Fujimoto  
Graduate School of Science  
Osaka Prefecture University  
Sakai, Osaka 599-8531, Japan  
Email: fujimoto@mi.s.osakafu-u.ac.jp

Abstract—This paper proposes a parallel ant colony optimization (ACO) for solving quadratic assignment problems (QAPs) on a graphics processing unit (GPU) by combining fast, 2-opt local search in compute unified device architecture (CUDA). In 2-opt for QAP, 2-opt moves can be divided into two groups based on computing cost. In one group, the computing cost is \( O(1) \) and in the other group, the computing cost is \( O(n) \). We compute these groups of 2-opt moves in parallel by assigning the computations to threads of CUDA. In this assignment, we propose an efficient method that can reduce disabling time in each thread of CUDA. The results show GPU computation with 2-opt produces a speedup of \( x_{24.6} \) on average, compared to computation with CPU.

I. INTRODUCTION

General-purpose computing on graphics processing units (GPGPU) has become popular with great success, especially in scientific fields such as fluid dynamics, image processing, and visualization using particle methods [1]. These parallel computations are reported to see a speedup of tens to hundreds of times compared to CPU computations [2]. However, GPU computations have only just started to be used in the evolutionary computation field as described in Section II-B. In previous studies [3], [4], we applied GPU computation to solve quadratic assignment problems (QAPs) using a distributed parallel GA model on GPUs. However, in those studies no local searches were applied. Ant colony optimization (ACO) shows high performance in solving combinatorial optimizations [5]. In this study, we propose a parallel ACO for solving QAPs on a GPU by combining fast, 2-opt local search [6] in CUDA (Compute Unified Device Architecture [2]).

In a QAP, a solution \( \phi \) is presented by a permutation of \( \{0, 1, 2, \cdots, n-1\} \) where \( n \) is the problem size. There are a total number of \( n(n-1)/2 \) neighbors of \( \phi \) in 2-opt. Depending on the pair value \( (i, j) \), 2-opt moves can be divided into two groups based on computing cost. In one group, the computing cost is \( O(1) \) (constant) and in the other group, the computing cost is \( O(n) \) [7]. We compute these groups of 2-opt moves in parallel by assigning the computations to threads of CUDA. In this assignment, we devised an efficient method that reduces disabling time, to the degree possible, in each thread of CUDA. As with the ACO algorithm, we use the Cunning Ant System (cAS) which is one of the most promising ACO algorithms [8].

In the remainder of this paper, Section II gives a brief review of GPU computation and its application to evolutionary computation. Then, cAS and how we apply it to solving QAPs are described in Section III. Section IV describes implementation of cAS with 2-opt on a GPU in detail. In Section V, results are presented. Finally, Section VI concludes the paper.

II. A BRIEF REVIEW OF GPU COMPUTATION AND ITS APPLICATION TO EVOLUTIONARY COMPUTATION

A. GPU Computation with CUDA

In terms of hardware, CUDA GPUs are regarded as two-level shared-memory machines [2]. Processors in a CUDA GPU are grouped into multiprocessors (MPs). Each MP consists of thread processors (TPs). TPs in an MP exchange data via fast shared memory (SM). On the other hand, data exchange among MPs is performed via VRAM. VRAM is also like main memory for processors. So, code and data in a CUDA program are basically stored in VRAM. Table I shows examples of specifications of GPUs which are recently available [9]. In this study, we use a single GTX 480.

<table>
<thead>
<tr>
<th>GPU</th>
<th>GTX 285 Architecture</th>
<th>Fermi Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of cores per MP</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>No. of MPs</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>Total No. of cores</td>
<td>240</td>
<td>480</td>
</tr>
<tr>
<td>Max No. of thread per block</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Clock rate</td>
<td>1477MHz</td>
<td>1400MHz</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>L1 cache</td>
<td>16KB</td>
<td>16KB / 48KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>16KB / 48KB</td>
<td>768KB</td>
</tr>
</tbody>
</table>

The CUDA programming model is a multi-threaded programming model. In a CUDA program, threads form two hierarchies: a grid and thread blocks. A block is a set of threads. A block has a dimensionality of 1, 2, or 3. A grid is a set of blocks with the same size and dimensionality. A grid has dimensionality of 1 or 2. Each thread executes the same code specified by the kernel function. A kernel function call generates threads as a grid with given dimensionality and size.
To obtain high performance, here we need to know how each thread runs in parallel. The approach is called single instruction, multiple threads (SIMT) [10]. In SIMT each MP executes threads in groups of 32 parallel threads called warps.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. However, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

In our implementation to be described in Section IV, we designed the kernel function so that the threads that belong to the same warp, to the degree possible, will not have branches.

B. GPU Computation in Evolutionary Computation

Studies on parallel evolutionary computation with GPU computation were performed genetic programming (GP), genetic algorithms (GAs), evolutionary programming (EP), ACO, and others.

The feasibility of evaluating GP populations directly using a GPU is shown in [11]. A SIMD C++ GP system on a GPU to predict a ten-year-plus outcome of breast cancer from a dataset containing a million inputs was built in [12]. A GP system on a commercial video game console was developed in [13]. In [14], accelerating GP through GPUs was also presented.

A parallel hybrid GA (HGA) on consumer-level graphics cards was presented in [15]. HGA extends the classical genetic algorithm by incorporating the Cauchy mutation operator from evolutionary programming. To evaluate and evolve neural models, a distributed adaptive GA (DAGA) was proposed in [16]. A parallel multi-objective evolutionary algorithm (MOEA) within the CUDA environment on an nVidia GPU was proposed in [17]. In [18], a hybrid approach which combines CPU and GPU was proposed. In [19], several types of GPU-based island models were proposed and applied to function optimization using evolutionary strategies (ESs) showing promising results. Studies solving travelling salesman problem (TSP) on GPUs were performed in [20], which used GAs and in [21], which used ACO.

Studies solving QAP on GPUs using an evolutionary model are found in [3], [4], [22], [23]. In [3], [4], distributed GA models were used and no local searches were applied. In [22], a cella GA model was used and no local searches were used. In [23], parallel hybrid evolutionary algorithms over CPU and GPU were proposed and applied to QAPs.

III. cAS and Its Application to Solving QAPs

Ant colony optimization (ACO) has been applied with great success to a large number of hard problems, mainly combinatorial optimization problems such as TSP, QAP, scheduling problems, and vehicle routing problems [5].

In a previous study we proposed a new ACO algorithm called the Cunning Ant System (cAS), and it was applied to solving TSP. In this research we apply cAS with 2-opt local search to QAPs on the GPU.

A. A Brief Introduction of cAS

cAS introduced two important schemes. One is a scheme to use partial solutions, which we call cunning. In constructing a new solution, cAS uses pre-existing partial solutions. With this scheme, we may prevent premature stagnation by reducing strong positive feedback to the trail density. The other is to use the colony model, dividing colonies into units, which has a stronger exploitation feature, while maintaining a certain degree of diversity among units.

cAS uses an agent called the cunning ant (c-ant). It constructs a solution by borrowing a part of an existing solution (we call it a donor ant (d-ant)). The remainder of the solution is constructed based on \( \tau_{ij}(t) \) probabilistically as usual. Let \( l_s \) represent the number of nodes of partial solution that are constructed based on \( \tau_{ij}(t) \) (i.e., \( l_s \), the number of nodes of partial solutions from its \( d-ant \), is \( n - l_s \), where \( n \) is the problem size). Then cAS introduces the control parameter \( \gamma \) which can define \( E(l_s) \) (the average of \( l_s \)) by \( E(l_s) = n \times \gamma \). Using \( \gamma \) values in [0.2, 0.5] is a good choice in cAS.

The colony model of cAS is shown in Fig. 1. It consists of \( m \) units. Each unit consists of only one \( ant^*_{k,t} \) (\( k = 1, 2, \ldots, m \)). At iteration \( t \) in unit \( k \), a new \( c-ant_{k,t+1} \) is generated using the existing ant in the unit (i.e., \( ant^*_k(t) \)) as the \( d-ant_{k,t} \). Then, the newly generated \( c-ant_{k,t+1} \) and \( d-ant_{k,t} \) are compared, and the better one becomes the next \( ant^*_{k,t+1} \) of the unit. Thus, in this colony model, \( ant^*_{k,t} \), the best individual of unit \( k \), is always reserved. Pheromone density \( \tau_{ij}(t) \) is then updated with \( ant^*_{k,t} \) (\( k = 1, 2, \ldots, m \)) and \( \tau_{ij}(t+1) \) is obtained as:

\[
\tau_{ij}(t+1) = \rho \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta^* \tau^k_{ij}(t),
\]

\[
\Delta^* \tau^k_{ij}(t) = 1/C^*_{k,t} : \text{ if } (i,j) \in ant^*_{k,t} : 0 : \text{ otherwise},
\]

where the parameter \( \rho (0 \leq \rho < 1) \) models the trail evaporation. \( \Delta^* \tau^k_{ij}(t) \) is the amount of pheromone by \( ant^*_{k,t} \) and \( C^*_{k,t} \) is the fitness of \( ant^*_{k,t} \).

Values of \( \tau_{ij}(t+1) \) are set to be within \([\tau_{min}, \tau_{max}]\) as in MAX-MIN ant system (MMAS) [6]. Here, \( \tau_{max} \) and \( \tau_{min} \) for cAS is defined as

\[
\tau_{max}(t) = \frac{1}{1 - \rho} \times \sum_{k=1}^{m} C^*_{k,t},
\]

\[
\tau_{min}(t) = \frac{(n/2 - 1) \cdot \sqrt{p_{best}}}{p_{best}},
\]

where \( p_{best} \) is a control parameter introduced in MMAS.

B. cAS on QAP

1) Quadratic Assignment Problem (QAP): The QAP is the problem which assigns a set of facilities to a set of locations and can be stated as a problem to find permutations \( \phi \) which minimize

\[
\text{cost}(\phi) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_{ij} b_{\phi(i)\phi(j)},
\]

where \( A = (a_{ij}) \) and \( B = (b_{ij}) \) are two \( n \times n \) matrices and \( \phi \) is a permutation of \{0, 1, \ldots, n-1\}. Matrix \( A \) is a
flow matrix between facilities $i$ and $j$, and $B$ is the distance between locations $i$ and $j$. Thus, the goal of the QAP is to place the facilities on locations in such a way that the sum of the products between flows and distances are minimized.

2) $c$-ant and $d$-ant in QAP: The $c$-ant in QAP acts in a slightly different manner from $c$-ant in TSP. In TSP, $\tau_{ij}(t)$ are defined on each edge between city $i$ and $j$. In QAP, the pheromone trails $\tau_{ij}(t)$ correspond to the desirability of assigning a location $i$ to a facility $j$. Fig. 2 shows how the $c$-ant acts in QAP. In this example, the $c$-ant uses part of the node values at positions 0, 2, and 4 of the $d$-ant, where these positions are determined randomly. The $c$-ant constructs the remainder of the node values for positions 1 and 3 according to the following probability:

$$p_{ij} = \frac{\tau_{ij}(t)}{\sum_{k \in F(i)} \tau_{ik}}, \quad (6)$$

where $F(i)$ is the set of facilities that are yet to be assigned to some locations.

In cAS for TSP, the number of nodes to be sampled, $l_s$, is determined probabilistically so that $E(l_s) = n \times \gamma$ [8]. In cAS for QAP in this study, we simply determine $l_s$ as $l_s = n \times \gamma$. Then we copy the number of nodes, $l_c = n - l_s$, from $d$-ant at its random positions and sample $l_s$ number of remaining nodes according to Eq. (6) with random sequence.

IV. IMPLEMENTATION OF cAS WITH 2-OPT ON A GPU

A. Structure of cAS based on CUDA

In this study, we apply 2-opt local search to newly generated solutions by cAS. Sequential algorithm cAS with 2-opt can be summarized as shown in Fig. 3.

Fig. 4 shows the configuration of the algorithm on a GPU based on CUDA. In this implementation, we coded the process of each step in Fig. 3 as a kernel function of CUDA. All of the data of the algorithm are located in VRAM. They include the population pools ($ant_{k,t}^*$, $c$-ant$_{k,t}$), the pheromone matrix $\tau_{ij}(t)$, the working memory for 2-opt (neighbor move costs), and the flow matrix $A$ and distance matrix $B$ of the QAP instance being solved solving. Since matrices $A$ and $B$ are constant data, reading them is performed through texture fetching. On SM, we located only working data which are shared among threads tightly in a block.

These kernel functions are called from the CPU for each ACO iteration. In these iterations of the algorithm, only the
The neighborhood size of $N$ of $\phi$ is $2$-opt iterative local search; given a solution $B$. An Efficient Implementation of 2-opt with CUDA can also be seen to increase with problem size. Thus, the efficient implementation of 2-opt is the most important factor in increasing speedup of this algorithm.

**TABLE II**

<table>
<thead>
<tr>
<th>Instances</th>
<th>Total</th>
<th>2-opt</th>
<th>Sampling</th>
<th>Trail update</th>
</tr>
</thead>
<tbody>
<tr>
<td>tai50b</td>
<td>100%</td>
<td>99.88%</td>
<td>0.11%</td>
<td>0.01%</td>
</tr>
<tr>
<td>tai60b</td>
<td>100%</td>
<td>99.91%</td>
<td>0.09%</td>
<td>0.01%</td>
</tr>
<tr>
<td>tai80b</td>
<td>100%</td>
<td>99.94%</td>
<td>0.06%</td>
<td>0.00%</td>
</tr>
<tr>
<td>tai100b</td>
<td>100%</td>
<td>99.96%</td>
<td>0.04%</td>
<td>0.00%</td>
</tr>
<tr>
<td>tai150b</td>
<td>100%</td>
<td>99.98%</td>
<td>0.02%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

**B. An Efficient Implementation of 2-opt with CUDA**

1) 2-opt Local Search in QAP: In this research, we use a 2-opt iterative local search; given a solution $\phi$, it explores its neighbourhood $N(\phi)$ and accepts a solution according to a given pivoting rule. Here we use the best improvement pivoting rule. The process is repeated until an $IT_{\text{max}}$ number of iterations is reached or no improvement solutions are found. The neighbourhood size of $N(\phi)$ ($|N(\phi)|$) is $n(n-1)/2$ where $n$ is the problem size.

When we exchange values in positions $r$ and $s$ of $\phi$ (i.e., $\phi(r), \phi(s)$), the change of cost($\phi$), $\Delta(\phi, r, s)$, can be computed in computational cost $O(n)$ as follows:

$$\Delta(\phi, r, s) = a_{rr}(b_{\phi(s)}(r) - b_{\phi(r)}(s)) + a_{rs}(b_{\phi(r)}(s) - b_{\phi(s)}(r)) + a_{sr}(b_{\phi(s)}(r) - b_{\phi(r)}(s)) + a_{ss}(b_{\phi(r)}(r) - b_{\phi(s)}(s)) + \sum_{k=0, k \neq r, s}^{n-1} \left( a_{kr}(b_{\phi(k)}(r) - b_{\phi(k)}(r)) + a_{ks}(b_{\phi(k)}(s) - b_{\phi(s)}(k)) + a_{ks}(b_{\phi(s)}(k) - b_{\phi(s)}(k)) + a_{sk}(b_{\phi(r)}(s) - b_{\phi(s)}(k)) \right)$$  \hfill (7)

Let $\phi'$ be results from $\phi$ by exchanging values in position $r$ and $s$ of $\phi$, then fast computation of $\Delta(\phi', u, v)$ is possible in computational cost $O(1)$ if $u$ and $v$ satisfy the condition $\{u, v\} \subseteq \{r, s\} = \emptyset$, as follows:

$$\Delta(\phi', u, v) = \Delta(\phi, u, v) + \left( a_{uv} - a_{vu} + a_{wu} - a_{uw} \right) \times \left( b_{\phi'\phi'(u)} - b_{\phi\phi'(u)} \right) + \left( a_{ur} - a_{ru} + a_{vr} - a_{rv} \right) \times \left( b_{\phi'\phi'(u)} - b_{\phi\phi'(u)} \right) + \left( a_{us} - a_{su} + a_{vs} - a_{sv} \right) \times \left( b_{\phi'\phi'(u)} - b_{\phi\phi'(u)} \right) + \left( a_{su} - a_{us} + a_{sv} - a_{us} \right) \times \left( b_{\phi'\phi'(u)} - b_{\phi\phi'(u)} \right)$$  \hfill (8)

However, to use this fast update, additional memorization of the $\Delta(\phi, i, j)$ values for all pairs $(i, j)$ in a table are required.

2) A Simple Threads Assignment of Move Cost Computations in a Block: In 2-opt in this study, we use a table which contains move costs so that we can compute the move cost in $O(1)$ by Eq. (8). For each move, we assign an index number as shown in Fig. 5. In this example, we assume a problem size of $n = 8$. Thus, the neighbourhood size $|N(\phi)|$ is $8 \times 7/2 = 28$. The simplest approach to computing the move costs in parallel in a block is to assign each move indexed $i$ to the corresponding sequential thread indexed $i$ in a block.

![Fig. 5. Indexing of moves ($n = 8$)](image)

Here, consider a case in which a current solution $\phi'$ is obtained by exchanging positions 2 and 4 of a solution $\phi$ in a previous 2-opt iteration. Then the computation of $\Delta(\phi', r, s)$, shown in bold face numbers, must be performed in $O(n)$ by Eq. (8). The computation of the remaining moves are performed in $O(1)$ quickly by Eq. (8).

Thus, if we simply assign each move to the block thread, threads of a warp diverge via the conditional branch ($\{r, s\} \cap \{2, 4\} = \emptyset$) into two calculations, threads in one group run in $O(n)$ of Eq. (7) and threads in the other group run in $O(1)$ of Eq. (8). As a result, the computation time of each thread in a warp becomes longer (please see Section II-A), and we cannot receive the benefit of the fast calculation by Eq. (8).

3) An Efficient Threads Assignment of Move Cost Computations in a Block: In general, for problem size $n$, the number of moves having move cost in $O(1)$ is $(n-2)(n-3)/2$ and the number of moves having move cost in $O(n)$ is $2n - 3$. Table III shows these values for various problem sizes $n$. For larger size problems, ratios of $|N(\phi)|$ in $O(1)$ to $|N(\phi)|$ have smaller values.

In this research, we assign move cost computations of a solution $\phi$ which are in $O(1)$ and in $O(n)$ to threads which...
belong to different warps in a block as described as follows and the code of the kernel function two_opt() is shown in Fig. 6.

Since the computation of a move cost which is $O(1)$ is smaller than the computation which is $O(n)$, we assign multiple number $N_S$ of computations which are $O(1)$ to a single thread in the block.

It is necessary to allocate multiple calculations of the moves to a thread also from the reason that the maximum number of threads in a block is limited (see Table I). Let $C$ be $|N(\phi)|$ ($C = n(n-1)/2$). Here, each neighbour is numbered by $\{0, 1, 2, \cdots, C-1\}$ (see Fig. 5). Then, thread indexed as $t = [k/N_S]$ computes moves for $k \in \{tN_S, tN_S + 1, \cdots, tN_S + N_S - 1\}$ (lines 23–33, and 38–52).

In this computation, if $k$ is a move in $O(n)$, then the thread indexed as $t$ skips the computation (line 48). The total number of threads assigned for computations in $O(1)$ is $T_S = \lceil C/N_S \rceil$.

For each thread indexed as $t$, we need to know the move pair values $(i, j)$ corresponding to each move assigned to it. In a thread indexed as $t$, if the pair $(i, j)$ for its initial move $tN_S$ is given, move pairs for $tN_S + 1, \cdots, tN_S + N_S - 1$ can be easily calculated (lines 30–32, and 49–51). So, we prepared a lookup table to provide the pair values only for initial move in each $t$ (move indexed as $tN_S$) (lines 22 and 37).

For the computation in $O(n)$, we assign only one computation of move cost to one thread in the block. Although the total number of moves in $O(n)$ is $2n - 3$, we used $2n$ threads for these parallel computations for implementation convenience. Since the threads for these computations must not share the same warp with threads used for computations in $O(1)$, the starting thread index should be a multiple of warp size (32), which follows the index of the last thread used for computation in $O(1)$. Thus, the first index of thread that computes move in $O(n)$ is $T_{L-\text{START}} = \lfloor T_S/32 \rfloor \times 32$. Among these $2n$ threads, 3 threads do nothing as seen in lines 59 and 70. Thus, the total number of threads $T_{\text{TOTAL}} = T_{L-\text{START}} + 2n$ and this kernel function is called from CPU by kernel call “two_opt<<< m, T_{\text{TOTAL}} >>>(...argument...);”. Hereafter, we refer to this thread structure as Move-Cost Adjusted Thread Assignment, or MATA for short. In Table IV, we summarize all kernel functions of this algorithm.

---

**TABLE III**

| Problem size $n$ | Neighborhood size $|N(n)|$ | $|N(n)|$ in $O(1)$ | $|N(n)|$ in $O(n)$ | $|N(n)|$ in $O(n)$ |
|------------------|---------------------|-----------------|-----------------|-----------------|
| 40               | 780                 | 703             | 77              | 0.099           |
| 80               | 3160                | 3003            | 157             | 0.050           |
| 120              | 7140                | 6903            | 237             | 0.033           |
| 160              | 12720               | 12403           | 317             | 0.025           |
| 200              | 19900               | 19503           | 397             | 0.020           |

Fig. 4. Configuration of the algorithm on a GPU based on CUDA
V. Experiments

A. Experimental Conditions

In this study, we used a PC which has one Intel Core i7 965 (3.2 GHz) processor and a single nVidia GeForce GTX480 GPU. The OS was Windows XP Professional with nVidia graphics driver Version 258.96. For CUDA program compilation, Microsoft Visual Studio 2008 Professional Edition with optimization option /O2 and CUDA 3.1 SDK were used.

The instances on which we tested our algorithm were taken from the QAPLIB benchmark library at [24]. QAP instances in the QAPLIB can be classified into 4 classes; (1) randomly generated instances, (2) grid-based distance matrix, (3) real-life instances, and (4) real-life like instances [25]. In this experiment, we used the following 9 instances which were classified as either (1) or (4) with the problem size ranging from 50 to 150; tai50b, tai60b, tai80b, tai100b, tai150b, tai50a, tai60a, tai80a, and tai100a.

10 runs were performed for each instance. We measured the performance by the average time to find acceptable solutions $T_{avg}$ over 10 runs. Instances classified into class (1) are much harder to solve than those in class (4). Thus, for acceptable solutions for class (1), we set them to be within 1.0% of the known optimal solutions. For class (4) instances, except tai150b, we set them to be known optimal solutions. For tai150b, we set them to be within 0.2% of the known optimal solution. We represent the average time over 10 runs as $T_{avg}$. Control parameter values were tuned by changing their combination values and their values are summarized in Table V.

We did the following three types of experiments:

1) Runs on a GPU with the MATA in Section IV-B3.
2) Runs on a GPU with the simple assignment in Section IV-B2. One modification is that we assign to one thread $N_S$ number of move calculations, similar to how it is done in MATA. Thus, in these runs we used a total number of $T_S = \lceil C/N_S \rceil$ threads in a block. Hereafter we will refer to this as non-MATA.
3) Sequential runs on a CPU using a single thread. The CPU is Intel Core i7 965 (3.2 GHz) processor. The parameter values are the same as those of (1) and (2).

### Table IV

<table>
<thead>
<tr>
<th>Kernel functions</th>
<th>dim3</th>
<th>block</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>pheromone update</td>
<td>initialize pheromone()</td>
<td>$n$</td>
<td>$T_{c0}(t)$</td>
</tr>
<tr>
<td>add pheromone()</td>
<td>$m$</td>
<td>$1$</td>
<td>$T_{c0} = \frac{1}{cost}$</td>
</tr>
<tr>
<td>max_min pheromone()</td>
<td>$n$</td>
<td>$n$</td>
<td>adjust $r_p$ in $[min, \text{tnax}]$</td>
</tr>
<tr>
<td>sampling</td>
<td>sampling()</td>
<td>$m$</td>
<td>$1$</td>
</tr>
<tr>
<td>local search</td>
<td>two_opt()</td>
<td>$m$</td>
<td>$T_{total}$</td>
</tr>
</tbody>
</table>

Fig. 6. The kernel function two-opt() code in CUDA.
B. Results

The results are summarized in Table VI.

1) Effect of Approach Including MATA: Here, we see the effect of approach including MATA. As seen in the table, the run time results with this approach in $T_{avg}$ (experiment (1)) were faster than those of the non-MATA run time in $T_{avg}$ (experiment (2)), although these speedup values are different among instances. For example, on tai60a, $T_{avg}$ of (1) is 25.7 and $T_{avg}$ of (2) is 150.7, respectively. The speedup ratio of (1) to (2) is x5.9. On tai60b, the speedup ratio of (1) to (2) is x8.2. These speedup values range in from x2.8 to x8.2 and the average value of the speedup values over 9 instances is x5.6.

2) GPU Computation with MATA versus CPU Computation: Now see the results of GPU computation of the proposed approach (MATA) compared to the results with CPU computation. On tai100a, for example, GPU computation of (1) is obtaining $T_{avg}$ of 909.9 and CPU computation is obtaining $T_{avg}$ of 20062.5 showing a speedup ratio of x22.0. Note here if we compare $T_{avg}$ of GPU with non-MATA and $T_{avg}$ of CPU, the speedup ratio on this instance is only x4.6. The speedup ratios of the GPU with MATA to CPU are in the in the range from x14.8 to x55.5, showing their average is x24.6.

3) Distribution of Computing Time with MATA: Table VII shows distribution of computing time with MATA. We can see that the distribution is similar to that was shown CPU computation in Table II. In this table, kernel function two_opt() uses over 99% of the computation time most of the time with MATA. This value can also be seen to increases with problem size. In implementing kernel function for creating new solutions (sampling()), we used a single thread in creating a single solution in a block (see Table IV). However, the kernel function sampling() uses from 0.06% to 2.3% of the computation time, and these values are less than 1% for instances which problem size $n$ are larger than or equal 80. Thus, we can see that the parallelization of kernel function sampling() in a block is not necessary.

VI. Conclusions

In this paper, we propose a parallel ACO for solving quadratic assignment problems (QAPs) on a GPU by combining fast, 2-opt local search in CUDA. In 2-opt on QAPs, there are a total number of $n(n−1)/2$ neighbours in a candidate solution. These 2-opt moves can be divided into two groups based on computing cost. In one group, the computing cost is $O(1)$ (constant) and in the other group, the computing cost is $O(n)$.

We compute these groups of 2-opt moves in parallel by assigning the computations to threads of CUDA. In this assignment, we proposed an efficient method which we call Move-Cost Adjusted Thread Assignment (MATA) that can reduce disabling time, to the degree possible, in each thread of CUDA. For the ACO algorithm, we use the Cunning Ant System (cAS). The results showed that the MATA in IV-B3 is a useful approach for fast execution in solving QAPs by ACO with 2-opt local search on a GPU. The GPU computation with MATA showed a speedup of x24.6 on average, compared to computation with CPU. The following remains for future work which tightly related to this research.

1) We used a single GPU. Study on colony models using multiple GPUs is an interesting research direction.

2) To use powerful local searches is an important issue. MATA can be applicable to a taboo search heuristics. To extend the 2-opt with MATA to a taboo search with MATA is also another interesting research direction.

ACKNOWLEDGMENT

This research is partially supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research No. 22500215.

REFERENCES

### TABLE VI
**Summary of Results. The effect of approach including MATA and GPU computation with MATA versus CPU computation over 9 QAP instances are presented.**

<table>
<thead>
<tr>
<th>QAP instances</th>
<th>Acceptable solution in error (%)</th>
<th>GPU Computation</th>
<th>CPU Computation</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MATA (E)</td>
<td>non-MATA (I)</td>
<td>T(<em>{\text{avg}}) of (2) / T(</em>{\text{avg}}) of (1)</td>
</tr>
<tr>
<td></td>
<td>T(_{\text{avg}}) (sec)</td>
<td>#OPT</td>
<td>T(_{\text{avg}}) (sec)</td>
<td>#OPT</td>
</tr>
<tr>
<td>tai50a</td>
<td>1.0</td>
<td>7.2</td>
<td>10</td>
<td>31.3</td>
</tr>
<tr>
<td>tai60a</td>
<td>1.0</td>
<td>25.7</td>
<td>10</td>
<td>150.7</td>
</tr>
<tr>
<td>tai80a</td>
<td>1.0</td>
<td>26.1</td>
<td>10</td>
<td>719.4</td>
</tr>
<tr>
<td>tai100a</td>
<td>1.0</td>
<td>909.9</td>
<td>10</td>
<td>4337.0</td>
</tr>
<tr>
<td>tai50b</td>
<td>0.0</td>
<td>0.2</td>
<td>10</td>
<td>1.6</td>
</tr>
<tr>
<td>tai60b</td>
<td>0.0</td>
<td>0.5</td>
<td>10</td>
<td>4.4</td>
</tr>
<tr>
<td>tai80b</td>
<td>0.0</td>
<td>6.8</td>
<td>10</td>
<td>39.0</td>
</tr>
<tr>
<td>tai100b</td>
<td>0.0</td>
<td>11.8</td>
<td>10</td>
<td>69.2</td>
</tr>
<tr>
<td>tai150b</td>
<td>0.2</td>
<td>98.8</td>
<td>10</td>
<td>612.8</td>
</tr>
<tr>
<td>average</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>


