ACO on Multiple GPUs with CUDA for Faster Solution of QAPs

Shigeyoshi Tsutsui

Hannan University, Matsubara Osaka 580-8502, Japan
tsutsui@hannan-u.ac.jp

Abstract. In this paper, we implement ACO algorithms on a PC which has 4 GTX 480 GPUs. We implement two types of ACO models; the island model, and the master/slave model. When we compare the island model and the master/slave model, the island model shows promising speedup values on class (iv) QAP instances. On the other hand, the master/slave model showed promising speedup values on both classes (i) and (iv) with large-size QAP instances.

1 Introduction

Recently, GPU (Graphics Processing Unit) computation has become popular with great success, especially in scientific fields such as fluid dynamics, image processing, and visualization using particle methods [1]. As for parallel ACO on GPU, Bai et al. [2], Fu et al. [3], and Delévacqa et al. [4] implemented MMAS on GPU with CUDA and applied it to solve TSP. In [5], Diego et al. proposed a parallelization strategy to solve the VRP with ACO on a GPU.

In a previous paper [6], we proposed an ACO to solve large scale quadratic assignment problems (QAPs) on a GPU (GTX480) with CUDA. We used tabu search (TS) as a local search of solutions obtained by the ACO. In the implementation, we proposed a novel threads assignment method in CUDA, which we call *MATA* (Move-Cost Adjusted Thread Assignment), to reduce the idling time of threads caused by thread divergence in a warp (see Section 2.1). We tested the ACO using several large-size benchmark instances in QAPLIB [7]. The ACO was able to solve the QAP instances successfully with about 20x speedup compared with CPU computation (i7 965, 3.2GHz). As for the ACO algorithm, we use the Cunning Ant System (cAS) [8].

In this paper, we implement the previous ACO algorithm on a PC which has 4 GTX 480 GPUs. We implement two types of ACO models using multiple GPUs. One is the island model, and the other is the master/slave model. In the island model, we implement one colony on each GPU, agents (solutions) are exchanged among colonies at defined ACO iteration intervals using several types of topologies. In the master/slave model, we have only one colony in the CPU, and only local search (TS) processes are distributed to each GPU.

In the remainder of this paper, Section 2 reviews of the previous study of ACO on a GPU with MATA and shows revised results using newly tuned parameter

C.A. Coello Coello et al. (Eds.): PPSN 2012, Part II, LNCS 7492, pp. 174–184, 2012.

settings. Then, Section 3 describes how the ACO is implemented on a PC with multiple GPUs in detail. In Section 4, experimental results and their analysis are given. Finally, Section 5 concludes this paper.

2 A Review of an ACO on a GPU with MATA and Revised Results

2.1 GPU Computation with CUDA

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. In a CUDA program, threads form two hierarchies: the *grid* and *thread blocks*. A block is a set of threads. A grid is a set of blocks with the same size. Each thread executes the same code specified by the *kernel function*.

Threads in a block are executed through a mode called *single instruction, multiple threads* (SIMT) [9]. 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.

2.2 ACO with TS on a GPU for Solving 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 a permutation ϕ which minimizes

$$cost(\phi) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_{ij} b_{\phi(i)\phi(j)} \quad (1)$$

where $A = (a_{ij})$ and $B = (b_{ij})$ are two $n \times n$ matrices and ϕ is a permutation of $\{0, 1, \cdots, n-1\}$. 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 is minimized.

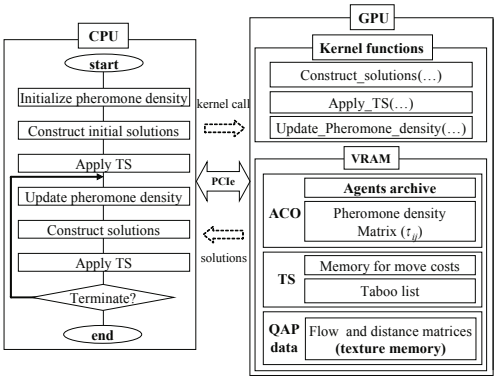

Fig. 1. ACO with TS on a GPU

Fig. 1 shows the configuration of the ACO with TS to solve GAPs on a GPU in [6]. As shown in the figure, each step of the algorithm is coded as a kernel function of CUDA. All of the data of the algorithm are located in VRAM of the GPU. As for the local search in Fig. 1, we implement TS based on Ro-TS [10].

Construction of a solution is performed by the kernel function "Construct_solutions$(\cdots)$" in a single block. Then each m solutions are stored in VRAM. In the kernel function "Apply_tabu_search$(\cdots)$", m solutions are distributed in m thread blocks. This function performs the computation of move-cost in parallel using a large number of threads in each block. Kernel function "Pheromone_update$(\cdots)$" consists of 4 separate kernel functions to ease implementation. Thus, in this configuration, the CPU performs only loop control of the algorithm.

2.3 Move-cost Adjusted Thread Assignment (MATA)

More than 99% of computation time was used for execution of TS when we ran the algorithm using CPU with a single thread (see Table 3 in Section 3.2). MATA was proposed for efficient implementation of TS on a GPU.

As is well known, in TS we need to check all solutions neighboring the current solution to obtain the best move. This move-cost calculation is costly. Let $N(\phi)$ be the set of neighbors of the current solution ϕ. A neighbor, $\phi' \in N(\phi)$, is obtained by exchanging a pair of elements (i, j) of ϕ. Then, we need to compute move-costs $\Delta(\phi, i, j) = cost(\phi') - cost(\phi)$ for all the neighboring solutions. The neighborhood size of $N(\phi)$ ($|N(\phi)|$) is $n(n-1)/2$ where n is the problem size. When we exchange r-th and s-th elements of ϕ (i.e., $\phi(r)$ and $\phi(s)$), $\Delta(\phi, r, s)$ can be calculated in computing cost $\mathcal{O}(n)$ [6].

Let ϕ' be obtained from ϕ by exchanging r-th and s-th elements of ϕ, then fast computation of $\Delta(\phi', u, v)$ is obtained in computing cost $\mathcal{O}(1)$ if u and v satisfy the condition $u, v \cap r, s = \emptyset$ [11]. To use this fast update, additional memorization of the $\Delta(\phi, i, j)$ values for all pairs (i, j) in a table are required. For each move, we assign an index number as shown in Fig. 2. In this example, we assume a problem size of $n = 8$. Thus, the neighborhood size $|N(\phi)|$ is $8 \times 7/2 = 28$. As described in Section 2.2, each set of move-cost calculations of an solution is being done in one block. 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.

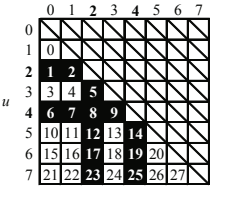

Fig. 2. Indexing of moves ($n = 8$)

Here, consider a case in which a solution ϕ' is obtained by exchanging positions 2 and 4 of a current solution ϕ in a previous TS iteration. Then the computation of $\Delta(\phi', u, v)$, the numbers shown in white font in black squares in Fig. 2, must be performed in $\mathcal{O}(n)$. The computation of the remaining moves are performed in $\mathcal{O}(1)$ fast. Thus, if we simply assign each move to the block thread, threads of a warp diverge via the conditional branch $(u, v \cap 2, 4 = \emptyset)$ into two calculations; threads in one group run in $\mathcal{O}(n)$ and threads in the other group run in $\mathcal{O}(1)$. In threads of CUDA, all instructions are executed in SIMT (see Section 2.1). As a result, the computation time of each thread in a warp becomes longer, and we cannot receive the benefit of the fast calculation of $\mathcal{O}(1)$ in [11].

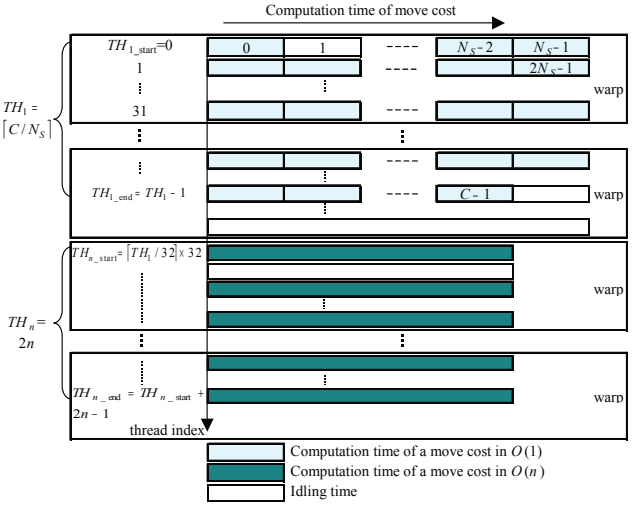

Fig. 3. Move-Cost Adjusted Thread Assignment

Thus, we should remove a situation where threads which run in $\mathcal{O}(1)$ and threads which run in $\mathcal{O}(n)$ co-exist in the same warp. In MATA, we assign move-cost computations of a solution ϕ which are in $\mathcal{O}(1)$ and in $\mathcal{O}(n)$ to threads which belong to different warps in a block, as shown in Fig. 3. Since the computation of a move-cost which is $\mathcal{O}(1)$ is smaller than the computation which is $\mathcal{O}(n)$, we assign a multiple number of N_S computations which are $\mathcal{O}(1)$ to a single thread in the block. Also, it is necessary to assign multiple calculations of the move-costs to a thread, because the maximum number of threads in a block is limited (1024 for GTX 480). Let C be $|N(\phi)|$ ($C = n(n-1)/2$). Here, each neighbor is numbered by $0, 1, 2, \cdots, C-1$ (see Fig. 2). Then, the thread indexed as $t = \lfloor k/N_S \rfloor$ computes moves for $k \in tN_S, tN_S + 1, \cdots, tN_S + N_S - 1$. In this computation, if k is a move in $\mathcal{O}(n)$, then the thread indexed as t skips the computation. The total number of threads assigned for computations in $\mathcal{O}(1)$ is $TH_1 = \lceil C/N_S \rceil$.

For the computation in $\mathcal{O}(n)$, we assign only one computation of move-cost to one thread in the block. Although the total number of moves in $\mathcal{O}(n)$ is $2n-3$, we used $TH_n = 2n$ threads for these computations for implementation convenience. Since the threads for these computations must not share the same warp with threads used for computations in $\mathcal{O}(1)$, the starting thread index should be a multiple of warp size (32), which follows the index of the last thread used for computation in $\mathcal{O}(1)$. Thus, the total number of threads in a block TH_{total} is $\lceil TH_1/32 \rceil \times 32 + TH_n$.

2.4 Revised Results

In this section, we present revised results from a previous study [6]. We tuned TS parameters so that we can get better performance as shown in Table 1. The machine is the same as before; i.e., 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. We updated CUDA 4.0 SDK from previous 3.1 SDK.

The instances on which we tested our algorithm were taken from the QAPLIB benchmark library [7]. QAP instances in the QAPLIB can be classified into 4 classes; (i) randomly generated instances, (ii) grid-based distance matrix,

(iii) real-life in-stances, and (iv) real-life like instances [11]. In this revised experiment, we used 10 instances which were classified as either (i) or (iv) (please see Table 2). Here, note that instances classified into class (i) are much harder to solve than those in class (iv). 25 runs were performed for each instance. In Table 1, values in parentheses are values used in [6].

Let IT_{TS} be the length of TS applied to one solution which is constructed by ACO, and IT_{ACO} be the iterations of ACO, respectively. Then, $IT_{TOTAL} = m \times IT_{ACO} \times IT_{TS}$ represents a total length of TS in the algorithm. We define a value $IT_{TOTAL-MAX} = m \times n \times 3200$. In this revised experiment, if IT_{TOTAL} reaches $IT_{TOTAL-MAX}$ or a known optimal solution is found, the algorithm terminates. This $IT_{TOTAL-MAX}$ is larger than the $IT_{TOTAL-MAX}$ in [6]. T_{avg} and $Error(\%)$ are mean run time and mean error over 25 runs, respectively. The revised results are summarized in Table 2. The effectiveness of using MATA is clearly observed as was shown in [6]. Values of *Error* are smaller and values of *Speedup* in T_{avg} are larger than observed in [6] due to revised parameter settings and longer runs.

Table 1. Revised parameter values (γ is a control paremeter of cAS [8])

Parameters		Values	
		class (i) QAPs	class (iv) QAPs
ACO	Number of ants: m	n	n ($4n$)
	Evaporation factor: ρ	0.2 (0.5)	0.2 (0.5)
	γ	0.4	0.5
TS	Length of TS: IT_{TS}	$64n$ ($16n$)	n ($4n$)
	Taboo list size: $T_{list\ size}$	$4n$ (n)	$n/2$ (n)
	N_S	$n/4$	$n/4$

3 Implementation of ACO on Multiple GPUs

Since there are four PCIe x16 slots in our system. we added an additional 3 GTX 480 GPUs and constructed a multi-GPU environment with a total of 4 GTX 480 GPUs. In this section, we propose two types of multi-GPU models for ACO with MATA, to attain a fast computation speed in solving QAPs. They are the island model and the master/slave model, the most popular parallel EAs [12, 13].

3.1 Island Model

Island models for EAs in a massively parallel platform are intensively studied in [14]. The cAS, which is used as our ACO model in this study, has an archive which maintains m solutions (see Fig. 1). This archive is similar to a population in EAs. In our implementation, we exchange (immigrate) the solutions among GPUs. In our implementation, one ACO model in a GPU in Section 2 composes one island. In the configuration of ACO on a GPU in Fig. 1, all m solutions are maintained in VRAM of the GPU. In an island model, we need to exchange solutions among islands (GPUs) depending on its topology.

Table 2. Revised results with MATA

QAP instances		GPU Computation (GTX 480) T_{avg} (sec) MATA	GPU Computation (GTX 480) T_{avg} (sec) non-MATA	GPU Computation (GTX 480) T_{avg} (sec) non-MATA / MATA	GPU Computation (GTX 480) *Error* (%)	CPU Computation T_{avg} (sec)	CPU Computation *Error* (%)	Speedup in T_{avg} CPU / MATA
class (i)	tai40a	9.5	70.6	7.4	0.14	191.0	0.14	20.0
	tai50a	18.3	136.0	7.4	0.34	463.7	0.35	25.3
	tai60a	32.6	269.1	8.3	0.32	962.8	0.36	29.6
	tai80a	154.9	728.2	4.7	0.41	3108.7	0.35	20.1
	tai100a	431.8	2352.9	5.4	0.36	7894.3	0.33	18.3
class (iv)	tai50b	0.3	1.7	6.4	**0**	6.8	**0**	24.9
	tai60b	0.5	4.0	7.9	**0**	15.4	**0**	30.5
	tai80b	5.5	30.6	5.6	**0**	145.0	**0**	26.5
	tai100b	14.6	80.8	5.5	**0**	374.4	**0**	25.7
	tai150b	2893.2	16348.8	5.7	0.07	48948.6	0.05	16.9
average		-	-	6.4	-	-	-	23.8

It is possible to exchange solutions among GPUs using "cudaMemcpyPeer(···)" function with CUDA 4.x. without via CPU. However, to perform exchange solutions depending on a defined topology, the CPU needs to know which data should be exchanged. This means that the CPU can't execute the cudaMemcpyPeer(···) function without having solutions from each GPU. Since the data needs to be sent to the CPU anyway, it is most efficient to exchange this data through the CPU rather than doing a direct exchanged between GPUs. Thus, in our implementation of island models, solutions in VRAM are transferred between GPU and CPU using usual "cudaMemcpy(···)" function when immigrations are required as shown in Fig. 4. As for control multiple GPUs in the CPU, we use OpenMP API.

Although there are many topologies for island models [13], in this study we implement the following 4 models:

(1) Island model with independent runs (IM_INDP): Four GPUs are executed independently. When at least one ACO in a GPU finds an acceptable solution, then the algorithm terminates.

(2) Island model with elitist (IM_ELIT): In this model, at defined ACO iteration interval $I_{interval}$ the CPU collects the global best solution from the 4 GPUs, and then distributes it to all GPUs except the GPU that produced that best solution. In each GPU, the worst solution in each archive is replaced with the received solution.

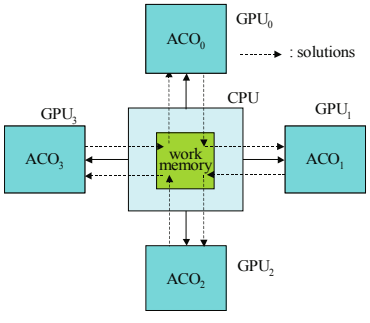

Fig. 4. Island model with 4 GPUs

(3) Island model with ring connected (IM_RING): The best solution in each GPU g ($g = 0, 1, 2, 3$) is distributed to its neighbor GPU $(g + 1)\ Mod\ 4$ at $I_{interval}$. In each GPU, the worst solution in each archive is replaced with the received solution if the received one is better than the worst one.

(4) Island model with elitist and massive ring connected (IM_ELMR): In this model, first the global best solution is distributed, as performed in IM_ELIT. Then, in addition to this immigration operation, randomly selected $m \times d_{rate}$ of solutions in the archive in each GPU are distributed to its neighbor. Received solutions in each GPU are compared with randomly selected, non-duplicate solutions. We use d_{rate} of 0.5 in this study.

3.2 Master/Slave Model

As mentioned in Section 2.3, more than 99% of computation time was used for execution of TS when we ran the algorithm using CPU with a single thread (see Table 3). In the master/slave model in this study, the ACO algorithm is executed in the CPU as shown in Fig. 5. Let m be number of agents in the archive of cAS, then we assign $m/4$ number of solutions to each GPU. When new solutions are generated in the CPU, first, $m/4$ number of solutions are transferred to each

GPU, then "Apply_tabu_search$(\cdots)$" kernel function is lunched to apply the TS with MATA to these solutions. The improved solutions are send back to the CPU from each GPU.

Note here that in practical implementation of the master/slave model, the value of m must be divisible by the number 4. So, we assigned $\lfloor m/4 \rfloor$ number of agents to each slave GPU and we used an agents number of $m' = \lfloor m/4 \rfloor \times 4$ instead of m.

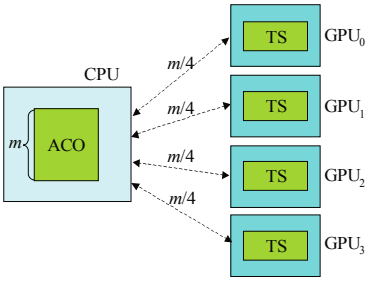

Fig. 5. Master/slave model with 4 GPUs

4 Experiment of Multiple GPUs in Solving QAPs

4.1 Experimental Setup

The machine is the same as in Section 2.4. We used 4 GTX 480 GPUs in 4 PCIe slots. We use the control parameter values shown in Table 1. We used the same QAP instances described in Section 2.4.

Table 3. Computation time with sequential CPU run

QAP Instances	Construction of solusions	TS	Updating Trail
tai60a	0.004%	99.996%	0.000%
tai100a	0.002%	99.998%	0.000%
tai100b	0.008%	99.991%	0.000%
tai150b	0.005%	99.995%	0.000%

Termination criteria are slightly different from those in Section 2.4. When we perform a fair comparison of different algorithms, sometimes it is difficult to determine their termination criteria. In this experiment, we run the algorithms until predetermined acceptable solutions are obtained and effectiveness of using 4 GPUs is measured by average time ($T_{avg,4}$) to obtained the solutions. We obtain the speedup by $T_{avg,1}/T_{avg,4}$, where $T_{avg,1}$ is average time to obtain acceptable solutions by ACO using a single GPU configured as described in Section 3. We performed 25 runs for each experiment.

We determined acceptable solutions as follows. For the class (i) instances, since it is difficult to obtain known optimal solutions 25 times in 25 runs with reasonable run time, we set the their acceptable solutions to be within 0.5% of the known optimal solutions. For the class (iv) instances, except tai150b, we set them to known optimal solutions. We set tai150b to be within 0.2% of the known optimal solution. We used $I_{interval}$ value of 1.

4.2 Results of the Island Models

Results of the island models are summarized in Table 4. The IM_INDP is the simplest of the island models. Thus, we use results of IM_INDP as bench marks for other island models. Except for the results from tai40a, all other island models had improved speedup values compared to IM_INDP. In the table, we showed the average number of iterations of the ACO to obtain the acceptable solutions (IT_{ACO}). On tai40a, this value is only 1.7. Thus, on this instance, there was no benefit from immigration operations.

The speedup values of IM_RING and IM_ELIT showed very similar results with each other. On tai80b and tai150b, we can see super-linear speedup values. We performed t-test between IM_ELIT and IM_INDP, showing a clear effect of using this topology, especially for class (iv) instances. Since we used long-tabu search length for class (i) instances (see Table 1), values of IT_{ACO} are smaller than those of class (iv) instances. This could have caused the reduced effect of immigration on class (i) instances, compared with (iv) instances.

Among the four island models, IM_ELMR showed the best speedup, except for tai40a. However, the t-test between IM_ELIT and IM_ELMR shows that the advantage of using IM_ELMR over IM_RING and IM_ELIT on class (i) instances again becomes smaller than on class (iv) instances. The speedup values are different among instances. Consider why these difference occur using IM_INDP as a parallel model. Let probability density function of the run time on a single GPU be represented by $f(t)$ and probability distribution function of $f(t)$ be $F(t)$. Here, consider an IM_INDP with p GPUs. Let the probability distribution function of run time of the IM_INDP with p GPUs be represented by $G(t, p)$. Since there is no interaction among GPUs in IM_INDP, the $G(t, p)$ can be obtained as

Table 4. Results of the island models with 4 GPUs

	QAP instances	Acceptable Error (%)	$T_{avg.1}$ (sec)	Average IT_{ACO}	*Speedup* ($T_{avg.1}/T_{avg.4}$)				*p-values*	
					IM_INDP	IM_RING	IM_ELIT	IM_ELMR	IM_INDP v.s. IM_ELIT	IM_ELIT v.s. IM_ELMR
class (i)	tai40a	0.5	0.4	1.7	1.7	**1.8**	1.7	1.7	0.90	0.85
	tai50a	0.5	4.2	11.4	2.1	2.6	2.9	**3.3**	0.07	0.44
	tai60a	0.5	10.6	14.8	1.9	2.2	2.3	**2.5**	0.03	0.61
	tai80a	0.5	58.3	18.9	2.4	2.5	2.7	**2.9**	0.45	0.75
	tai100a	0.5	125.9	14.6	1.7	2.1	2.2	**2.5**	0.03	0.13
class (iv)	tai50b	0	0.2	31.8	1.5	2.3	2.5	**3.0**	6.50E-05	0.08
	tai60b	0	0.4	25.7	1.2	1.4	1.9	**2.6**	4.68E-05	5.83E-05
	tai80b	0	6.6	121.3	1.5	4.7	4.3	**6.5**	9.16E-11	4.00E-09
	tai100b	0	10.1	67.0	1.4	2.3	2.3	**3.2**	6.00E-08	1.24E-05
	tai150b	0.2	105.9	116.6	1.8	4.2	4.3	**4.9**	2.10E-05	0.22

$$G(p, t) = 1 - (1 - F(t))^p, \tag{2}$$

and the average run time $T_{avg,p}$ is obtained as

$$T_{avg,p} = \int_0^\infty t \cdot G'(p, t)dt \tag{3}$$

Thus, the speedup with p GPUs is obtained as $Speedup(p) = T_{avg,1}/T_{avg,p}$. Table 5 shows the values of $Speedup(p)$ and $Speedup(4)$ for assuming various functions of $f(t)$. This analysis gives us a good understanding of the results of IM_INDP in Table 4. But for more detail analysis, we need to identify $f(t)$ by sampling the data of run times.

Table 5. Estimation of *Speedup*

$f(t)$	*Speedup* (p)	*Speedup* (4)
$f(t) = 1, \quad 0 \le t \le 1$	$(p+1)/2$	2.5
$f(t) = 2(1-t), \quad 0 \le t \le 1$	$(2p+1)/3$	3
$f(t) = 3(1-t)^2 \quad 0 \le t \le 1$	$(3p+1)/4$	3.25
$f(t) = \lambda e^{-\lambda t} \quad 0 \le t$	p	4

4.3 Results of the Master/Slave Model

Since computation times of TS occupy more than 99% of the algorithm (Table 3), we expected the master/slave model to show good results in the speedup. However, as shown in Fig. 6, results on the small-size instances in this study (tai40a, tai50a, tai60a, tai50b, tai60b) showed relatively small speedup values against the ideal speedup value of 4. In the figure, the *Speedup* values are defined in Section 4.1. Results on large-size instances (tai80a, tai100a, tai80b, tai100b tai150b), the speedup values were nearer to ideal speedup values.

Now we will analyze why these results were obtained on the master/slave model. Fig. 7 shows the average computation times of tai60a and tai150b over 10 runs for 10 ACO iterations by changing the number of agents m from 1 to 150 with step 1. Here, the ACO algorithm is the master/slave model in Section 3.2 with a GPU number setting of 1, and the computation time is normalized by the time of $m = 1$. Since the number of MPs of GTX 480 is 15, we can see the computation times increases nearly 15 interval of m. However, the increasing times are different between these two instances.

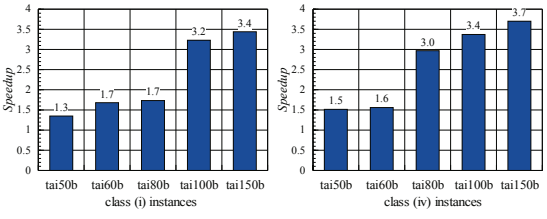

Fig. 6. Results of the master/slave model with 4 GPUs

On tai60a ($n = 60$) instance, the difference of computation times among $1 \leq m \leq 15$, $16 \leq m \leq 30$, and $31 \leq m \leq 45$, and $46 \leq m \leq 60$ is very small. In our implementation of TS on a GPU, we assigned one thread block to each agent (solution), and thus number of agents is identical to number of thread blocks. In CUDA, multiple blocks are allocated to one MP if computation resources, such as registers, are available. In the execution of tai60a ($n = 60$), this situation occurs and multiple blocks are executed in parallel in one MP. Since in this experiment, we set $m = 60$ ($:m = n$, see Section 4.1 and Table 1), the solutions assigned to one slave GPU is only 15. This means the speedup using the master/slave model becomes very small as was seen in Fig. 6.

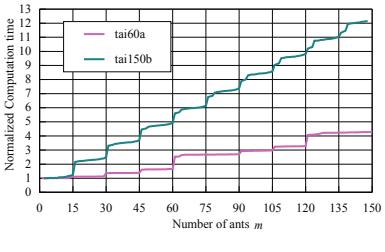

Fig. 7. Computation times for various number of agents

On the other hand, on tai150b ($n = 150$) instances, computation times proportionally increase according to m with every 15 intervals. This means that on tai150b, a single thread block is allocated to one MP at the same time, with the resulting speedup shown in Fig. 6. Note here that on this instances, the number of agents assigned to one GPU is $\lfloor 150/4 \rfloor = 37$ and the total agent number of $37 \times 4 = 148$ was used in this experiment.

5 Conclusion

In a previous paper, we proposed an ACO for solving QAPs on a GPU by combining TS local search in CUDA. There, we implemented an efficient thread assignment method, MATA. Based on this implementation, in this paper we implemented the algorithm on multiple GPUs to solve QAPs fast. We implemented two types of models on multiple GPUs; the island model and the the master/slave model. For these models, we experimented using QAP benchmark instances, and gave analysis on the results.

As for the island model, we used 4 types of topologies. Although the results of speedup much depend on the instances we used, we showed that the island model IM_ELMR has a good speedup feature. As for the master/slave model, we observed reasonable speedups for large-size of instances, where we used large number of agents.

When we compared the island model and the master/slave model, the island model showed promising speedup values on class (iv) instances of QAP. On the other hand, the master/slave model consistently showed promising speedup values both on classes (i) and (iv) with large-size QAP instances with large number of agents. As regards to this comparison, a more intensive analytical study is an interesting future research direction. Implementation using an existing massively parallel platform such as EASEA [14] is also an interesting future research topic.

References

1. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., Mei, W., Hwu, W.: Program optimization carving for GPU computing. J. Parallel Distrib. Comput. 68(10), 1389–1401 (2008)
2. Bai, H., OuYang, D., Li, X., He, L., Yu, H.: MAX-MIN ant system on GPU with CUDA. In: Innovative Computing, Information and Control, pp. 801–804 (2009)
3. Fu, J., Zhou, G., Lei, L.: A parallel ant colony optimization algorithm with GPU-acceleration based on all-in-roulette selection. In: Workshop on Advanced Computational Intelligence, pp. 260–264 (2010)
4. Delévacqa, A., Delislea, P., Gravelb, M., Krajeckia, M.: Parallel ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing (in press, 2012)
5. Diego, F., Gómez, E., Ortega-Mier, M., García-Sánchez, Á.: Parallel CUDA architecture for solving de VRP with ACO. In: Industrial Engineering and Industrial Management, pp. 385–393 (2012)
6. Tsutsui, S., Fujimoto, N.: ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment. In: GECCO 2011, pp. 1547–1554. ACM (2011)
7. Burkard, R.E., Çela, E., Karisch, S.E., Rendl, F.: QAPLIB - a quadratic assignment problem library (2009), www.seas.upenn.edu/qaplib
8. Tsutsui, S.: *c*AS: Ant Colony Optimization with Cunning Ants. In: Runarsson, T.P., Beyer, H.-G., Burke, E.K., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN IX. LNCS, vol. 4193, pp. 162–171. Springer, Heidelberg (2006)
9. NVIDIA: (2010), developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

10. Taillard, É.: Robust taboo search for quadratic assinment problem. Parallel Computing 17, 443–455 (1991)
11. Taillard, É.: Comparison of iterative searches for the quadratic assignment problem. Location Science 3(2), 87–105 (1995)
12. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. John Wiley and Sons (2005)
13. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kuwer Academic Publishers (2000)
14. Maitre, O., Krüger, F., Querry, S., Lachiche, N., Collet, P.: EASEA: specification and execution of evolutionary algorithms on GPGPU. Soft Comput. 16(2), 261–279 (2012)