Solving Quadratic Assignment Problems by Genetic Algorithms with GPU Computation: A Case Study

Shigeyoshi Tsutsui
Department of Management and Information Science, Hannan University
5-4-33 Amamihigashi, Matsubara, Osaka 580-8502, Japan
tsutsui@hannan-u.ac.jp

Noriyuki Fujimoto
Graduate School of Science, Osaka Prefecture University
1-1 Gakuen-Cho, Naka-ku, Sakai-Shi, Osaka, 599-8531, Japan
fujimoto@mi.s.osakafu-u.ac.jp

ABSTRACT

This paper describes designing a parallel GA with GPU computation to solve the quadratic assignment problem (QAP) which is one of the hardest optimization problems in permutation domains. For the parallel method, a multiple-population, coarse-grained GA model was used. Each subpopulation is evolved by a multiprocessor in a GPU (NVIDIA GeForce GTX285). At predetermined intervals of generations all individuals in subpopulations are shuffled via the VRAM of the GPU. The instances on which this algorithm was tested were taken from the QAPLIB benchmark library. Results were promising, showing a speedup ration from 3 to 12 times, compared to the Intel i7 965 processor.

Categories and Subject Descriptors

I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—Heuristic Methods; D.1.3 [Programming Techniques]: Concurrent Programming—Distributed programming, Parallel programming

General Terms

Algorithms

Keywords

Genetic Algorithm, Distributed Genetic Algorithm, Quadratic Assignment Problem, Parallel Computation, GPU Computation

1. INTRODUCTION

Parallel genetic algorithms (GAs) have been recognized for a long time and have been successfully applied to solve many hard tasks to reduce the time required to reach acceptable solutions. According to Cantú-Paz (2000) [3], parallel GAs can be classified into the following three main types: (1) global single-population master-slave GAs, (2) single-population fine-grained, and (3) multiple-population coarse-grained GAs. In a master-slave GA, there is a single main GA, but the evaluation of fitness is distributed among several processors. Fine-grained parallel GAs are suited for massively parallel computers and consist of one spatially-structured population. Selection and mating are restricted to a small neighborhood, but neighborhoods overlap, permitting some interaction among all the individuals. Multiple-population GAs are the most popular approach. They consist of several subpopulations which exchange individuals occasionally (migration). Since multiple-population coarse-grained GAs are usually implemented on distributed memory MIMD computers, they are known also as *distributed* GAs.

Although parallel GAs are very promising, it is true that there are many barriers to be conquered in designing parallel GAs, depending on the computational environment available at hand and also there are many design factors, such as determining parallel topology, parameter setting, reducing communication overhead, etc. As a new parallel computation scheme, recently there has been a growing interest in developing parallel algorithms using graphic processing units (hereafter, we refer to this as GPU computation).

In this study, we designed a parallel GA with GPU computation to solve the quadratic assignment problem (QAP) based on multiple-population coarse-grained GAs using CUDA GPU programming architecture. We chose QAP for the following reasons: First, problem sizes of QAPs in real life problems are relatively small compared with other problems in permutation domains such as the traveling salesman problem (TSP) and the scheduling problem. This enables us to use the shared memory of a GPU effectively. Second, QAP is one of the most difficult problems among problems in permutation domains. Thus, QAP is a good test bed to evaluate an optimization algorithm (please refer to 3.1 for details of QAP). Although there is still much future work to go related to this study, our results were promising, showing a speedup ration from 3 to 12 times, compared to the Intel ® Core™ i7 965 processor, which is one of the fastest processors available currently.

In the remainder of this paper, we give a brief overview of GPU computation and its applications to evolutionary computaion in Section 2. Then, in Section 3 the proposed parallel GA on a GPU for solving QAP is described in detail. In Section 4, experimental results and their analysis are performed. Finally, Section 5 concludes this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'09, July 8–12, 2009, Montréal Québec, Canada.
Copyright 2009 ACM 978-1-60558-505-5/09/07 ...$5.00.

2. A BRIEF REVIEW ON GPU COMPUATION AND ITS APPLICATION TO EVOLUTIONARY COMPUTATION

2.1 GPU Computation

In terms of hardware, CUDA GPUs are regarded as two-level shared-memory machinesas shown in Fig. 1. Processors in a CUDA GPU are grouped into multiprocessors. Each multiprocessor consists of 8 processors. Processors in a multiprocessor can exchange data via 16KB fast shared memory. On the other hand, data exchange between multiprocessors can be performed via VRAM. VRAM is also like main memory for processors. So, code and data in a CUDA program are basically stored in VRAM. Although processors have no data cache for VRAM, shared memory can be used as manually controlled data cache.

CUDA programming model is one of multi-threaded programming models. In the following, we describe an overview of CUDA programming model using Fig. 2. In a CUDA program, threads form two hierarchy: a grid and thread blocks. A thread block is a set of threads. A thread block has dimensionality 1, 2, or 3. A grid is a set of blocks with the same size and dimensionality. A grid has dimensionality 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. The reason why threads form two hierarcy is as follows. Threads in a thread block can share data efficiently via shared memory. However, the maximum number of threads per block is limited to 1024. So, if more than 1024 threads are required, we have to partition threads into several thread blocks with the same size.

VRAM bandwidth is very high at 159 GB/sec. However, when accessing VRAM, there are memory latency as large as 100 to 150 arithmetic operations [23]. This VRAM latency can be hidden if there are a sufficient number of threads by overlapping memory access of a thread with computation of other threads. The number of threads that run concurrently on a multiprocessor is restricted by the fact that shared memory and registers in a multiprocessor are divided among concurrent thread blocks allocated for the multiprocessor. NVIDIA recommends at least 192 threads per block and at least 2 blocks per multiprocessor [23].

VRAM has a read-only region of size 64KB [23]. The region is called the *constant memory*. The constant memory space is cached. The cache working set for constant memory is 8 KB per multiprocessor [23].

2.2 Applications of GPU Computation to the Evolutionary Computation

There are many parallel computations in the scientific computation fields [23]. For example, cloud tracking (Grauer-Gray et al., 2008) [9], statistical static timing analysis (Gulati & Khatri, 2008) [10], biomedical image analysis (Hartley et al., 2008) [11], AES cryptography encoding and decoding (Manavski, 2007) [20], medical image construction (Schellmann, 2008) [28] and so on. These studies show promising results in computation time. One of such promising results is our fast implementation of matrix-vector multiplication on the CUDA architecture by Fujimoto (2008) [7, 6]. According to subsequent research [18, 13, 22, 17], our implementation has been proven to be still the fastest with significant performance over other studies.

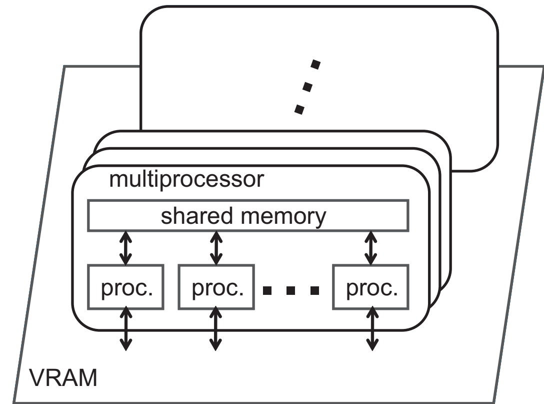

Figure 1: CUDA architecture as a two-level shared-memory machine

Some studies with CUDA can be found in the evolutionary computation field, especially in genetic programming (GP), as shown in Banzhaf et al.'s intensive survey [2]. Langdon et al. showed the feasibility of evaluating genetic programming populations directly using a GPU (Langdon & Banzhaf, 2008) [14]. Langdon & Harrison (2008) built a SIMD C++ GP system on a GPU to predict ten year+ outcome of breast cancer from a dataset containing a million inputs [15]. Robilliard et al. (2008) showed it is possible to efficiently interpret several GP programs in parallel [26]. Wilson & Banzhaf (2008) implemented a GP system on a commercial video game console [33].

A few studies on parallel GAs with GPU computation are reported. Fok et al. (2007) showed early results that a range from 1.25 to 5 times speedup can be achieved [5] using traditional GPGPU, which reduces general computations to a series of rasterization problems on a GPU using OpenGL or DirectX 3D. Wong, M. L. & Wong, T. T (2006) reported on a parallel hybrid genetic algorithm (HGA) on consumer-level graphics cards [34]. HGA extends the classical genetic algorithm by incorporating the Cauchy mutation operator from evolutionary programming. To evaluate and evolve neural models, Clayton et al. (2008) proposed a Distributed Adaptive Genetic Algorithm (DAGA), which is suitable to the large population sizes promoted by the GPU architecture [4].

In GAs, algorithms are performed introducing randomness. As a result, program flow for each individual is much different from other individuals because if statement condition and for or $while$ loop iteration numbers among individuals are different. This reduces the efficiency of parallel executions in each multiprocessor (MP) in a GPU, because each thread in an MP is executed in SIMD fashion. Further, efficient data sharing among threads is restricted due to the relatively small size of shared memory in each MP (16KB for GeForce GTX285, for example).

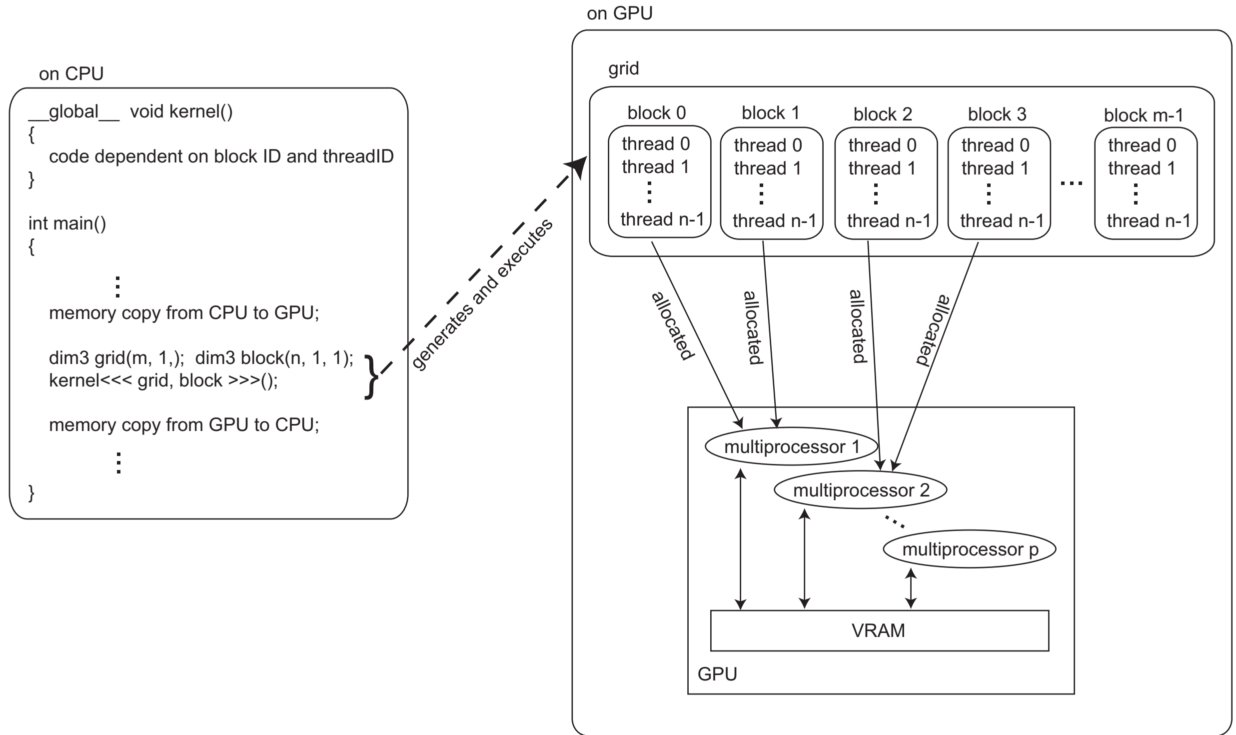

Figure 2: An overview of CUDA programming model

3. IMPLEMENTATION OF PARALLEL GA FOR GPU COMPUTATION

3.1 Quadratic Assignment Problem (QAP)

In QAP [12], we are given l locations and l facilities and the task is to assign the facilities to the locations to minimize the cost. For each pair of locations i and j, we are given their distance d_{ij}. For each pair of facilities i and j, we are given the flow f_{ij} between these facilities (regardless of their locations). The cost of each assignment, ϕ, is given by the sum over-all pairs of locations of the product of the distance between these locations and the flow between the facilities assigned to these locations. Assignments of facilities to locations can be encoded as permutations. The location for facility i is given by the value at position i of the permutation. The cost for such an assignment is given by

$$cost(\phi) = \sum_{i=1}^{l}\sum_{j=1}^{l} f_{ij} d_{\phi(i)\phi(j)}. \tag{1}$$

The task is to minimize $cost(\phi)$ over all possible assignments (permutations). Intuitively, $f_{ij}d_{\phi(i)\phi(j)}$ represents the cost contribution of simultaneously assigning facility i to location $\phi(i)$ and facility j to location j.

Figure 3 shows a simple example of a QAP with problem size l of 4.The distance matrix d_{ij} and the flow matrix f_{ij} are 4×4 matrices. In general, they are asymmeterical but in the figure we assumed they are symmetrical for simplicity. As seen in the figure, the permutation ϕ={2, 1, 4, 3} means that facility 1 is assigned to location 2, facility 2 is assigned to location 1, facility 3 is assigned to location 4, and facility

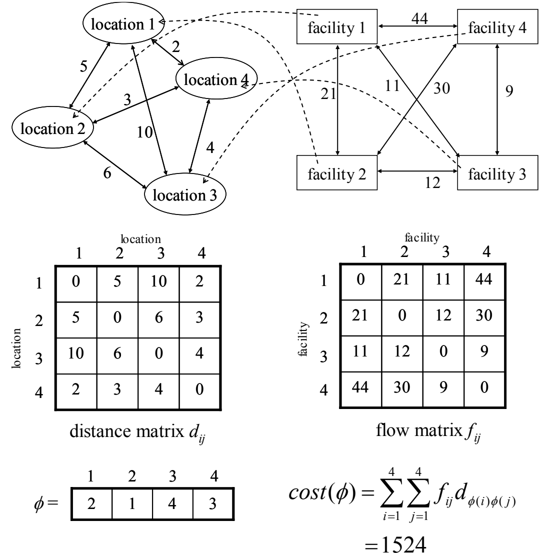

Figure 3: A simple example of QAP

4 is assigned to location 3, respectively. For this assignment ϕ, the value of $cost(\phi)$ according to Eq. (1) is 1524.

The QAP is an NP-hard optimization problem [27] and it is considered one of the hardest optimization problems as described in Section 1. The instances on which we will test the parallel GA with GPU computation are taken from the QAPLIB benchmark library at [1].

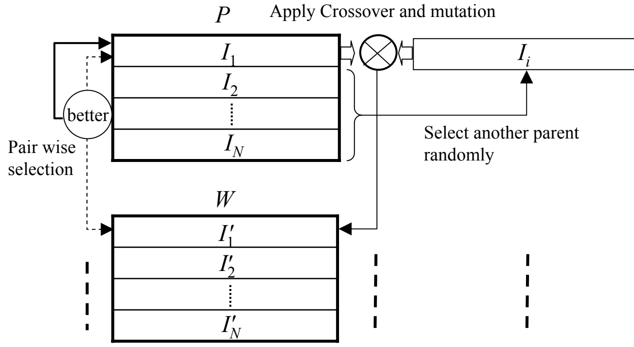

Figure 4: The base GA model for QAP

3.2 GA Model for GPU Computation for QAP

3.2.1 *The base GA model for QAP*

We describe the base GA model for QAP which is common both for GPU computation and CPU computation. In addition to GA, there are several evolutionary computation approaches to solve QAP such as the ant colony optimization (ACO) [29, 21, 31] and the estimation of distribution algorithm (EDA) [25]. However, in this study we use GA using commonly used crossover operators. Figure 4 shows the base GA model for QAP in this research. Let N be the population size. We use two pools P and W of size N. P is the population pool to keep individuals of the current population, and W is a working pool to keep newly generated offspring individuals until they are selected to update P for the nest generation. Then, the algorithm is as follows:

Step 1 Set generation counter $t \leftarrow 0$ and initialize P.

Step 2 Evaluate each individual in P.

Step 3 For each individual I_i in P, select its partner I_i ($i \neq j$) randomly. Then apply a crossover to the pair (I_i , I_j) and generate one child I'_i in position i in W.

Step 4 For each I'_i, apply a mutation with probability p_m.

Step 5 Evaluate each individual in W.

Step 6 For each i, compare the costs of I_i and I'_i. If I'_i is the winner, then replace I_i with I'_i.

Step 7 Increment generation counter $t \leftarrow t + 1$.

Step 8 If the termination criteria are met, terminate the algorithm. Otherwise, go to Step 3.

Usually, a crossover operator generates two offspring from two parents. However, in this model we generate only one child from two parents. In Step 6, the comparison of costs is performed like a tournament selection with size 2. However, each comparison is performed between individuals I_i and I'_i which have the same index i. Please remember here that I'_i is generated from I_i as one of its parents (the other parent was chosen randomly). Since a parent and a child have partly similar substrings, this comparison scheme can be expected to maintain population diversity like the deterministic crowding proposed by Mahfoud [19]. From the viewpoint of the parallelization, this model has also advantage because a process for one individual in a generation cycle can be implemented in parallel thread easily. Using one child from two parents is already proposed for designing the well known GENITOR algorithm by Whitley et al. [32].

For crossover operators, we perform preliminary tests using two well known operators, i.e, the order crossover (OX) [24] and the partially mapped crossover (PMX) [8] by implementing the GA model both on CPU and GPU. The results showed PMX operator worked much better than OX operator on QAPs. Thus, in the experiment in Section 4, we used PMX operator. For mutation operator, we used a swap mutation where values of two randomly chosen positions in a string are exchanged. Strings to which the mutation are applied are probabilistically determined with mutation rate of p_m.

Applying local search in solving QAP is very common in evolutionary algorithms [29, 21, 31]. Popular local searches used in solving QAP are the taboo search (TS) and 2-OPT heuristic. However, to apply these local searches efficiently, we need a large memory for each parallel thread. For example, the size needed for taboo search [30] is $32l^2 + l$ bytes. To take this size space for each thread in the shared memory with current GPU is almost impossible. The main purpose of this research is to implement an efficient parallel GA on a GPU, and thus to see the effectiveness of the parallel GA itself, we will not apply any local search in this study.

3.2.2 *Parallel GA model for GPU Computation*

The GPU NVIDIA GeForce GTX285 which we use in this study has 30 multiprocessors (MPs) and each MP has 8 stream processors (SPs) sharing 16KB high speed memory among them. To use this machines features efficiently, we divide individuals into subpopulations of size 128 each (see 3.2.3). For each subpopulation, we use the base GA model described in 3.2.1. So, we allocate the population pools P and W described in 3.2.1 to the shared memory of each MP. We define the total number of subpopulation as $30 \times k (k = 1, 2, ...)$. So, the total individuals number with this model is $128 \times 30 \times k$. The parameter k finally determines the total thread number in CUDA programming and the total thread number is the same with the total individuals number.

The procedure of the parallel GA model for GPU computation is as follows:

(1) In the host machine, all individuals are shuffled. Then, they are sent to the VRAM of the GPU.

(2) For a subpopulation not yet processed, each MP selects one of such subpopulations, and copies the corresponding individuals from VRAM to its shared memory, and performs the generational process in Figure 3 up to $G_{interval}$ generations, and finally copies the evolved individuals from its shared memory to VRAM.

(3) The above process is repeated until all subpopulations are processed.

(4) Then, all individuals are copied back to the memory of its host machine and merged.

(5) These processes are repeated until termination criteria are satisfied.

As seen in these descriptions, this parallel GA model is a variant of distributed GAs.

3.2.3 *Implementation Details for GPU Computation*

To accelerate a CUDA program, the following optimizations are significant:

(1) Maximizing utilization of shared memory. Device memory, except constant memory space, is not cached. We have only 64KB constant memory space. Latency of VRAM access is 100 to 150 times longer than the execution time of a basic arithmetic operation. On the other hand, latency of shared memory access is the same as that of registers if bank conflict does not occur. Shared memory is just like manually controlled cache memory.

(2) Maximizing the number of threads running on an MP. The CUDA GPU chip can hide latency of VRAM access by overlapping the access with computation of other threads. The overhead of thread switching is zero clock [16]. To use this ability effectively, the existence of many *other threads* is essential.

(3) Coalescing access to VRAM. VRAM bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction of 32, 64, or 128 bytes [23]. Our preliminary experiments showed that coalesced access is about 10 times faster than non-coalesced access.

To load as many individuals as possible in shared memory, we represent an individual as an array of type unsigned char, rather than type int. This restricts the problem size we can solve to at most 255. However, this does not immediately interfere with solving QAP because QAP is fairly difficult even if the problem size is relatively small.

Let l be the problem size of a given QAP instance. Then, the size of each individual is l bytes. So, the size of population pools P and working pool W is $2l \times N$ where N is the number of individuals allocated to each MP at the same time. We have only 16KB shared memory per MP. To maximize both l and N with loading and storing individuals between shared memory and VRAM coalesced, we chose N as 128 and the shape of a thread block as 16 times 8 under the assumption that l is at most 56.

Figure 5 shows how the thread block loads/stores 128 individuals in coalesced access manner. In our implementation, we represent each individual as an array of L elements of type unsigned char and L is fixed to 56 regardless of l. Consequently, for W and P, our implementation consumes $2L \times N = 2 \times 56 \times 128 = 14336$ bytes in shared memory.

We stored distance matrix d_{ij} and flow matrix f_{ij} in the constant memory space so that they can be accessed via cache. To save the memory space size for these matrices, unsigned short was used for the elements of these matrices.

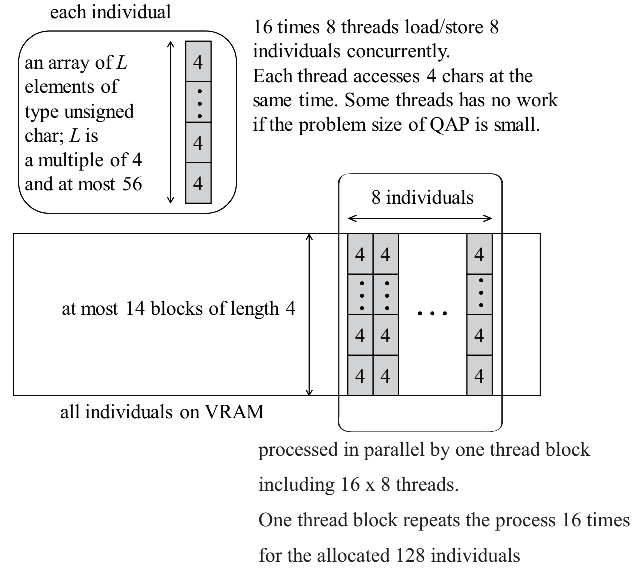

Figure 5: Coalesced access for a population pool

We implemented a simple random number generator for each thread with independent seed number.

4. EXPERIMENTS

4.1 Experimental Conditions

In this study, we used a PC which has one Intel ® Core™ i7 965 processor and one NVIDIA GeForce GTX285. The OS was Windows XP Professional with NVIDIA graphics driver Version 181.20. For CUDA program compilation, Microsoft Visual Studio 2005 Professional Edition with optimization option /O2 and CUDA 2.1 SDK were used.

We used test instances in QAPLIB [36]. QAP described in 3.1 can be classified into 4 classes; i) randomly generated instances, ii) grid-based distance matrix, iii) real-life instances, and iv) real-life like instances [34]. In this experiment, we use the following 8 instances which are classified into iii) and iv) with the problem size ranging from 20 to 40; tai20b, tai25b, kra30a, kra30b, tai30b, tai35b, ste36b, and tai40b.

10 runs were performed for each instance. We measured the performance by the average time to find optimal solutions in successful runs (T_{avg}) using CPU wall clock. We increased parameter k (please see 3.2.2) from 1 to 6 with step size 1 until the $\#OPT$ 9 is obtained where $\#OPT$ is the number of runs in which the algorithm succeeded in finding the optimal solution. The maximum execution time for each run is set to 15 seconds for tai20b, 30 seconds for tai25b, and 120 seconds for kra30a, kra30b, tai30b, tai35b, ste36b, respectively.

$G_{interval}$ in 3.2.2 is also an important parameter. Small $G_{interval}$ increases data transfer time between the shared

memory and the VRAM, and VRAM and main memory of the host computer. In this study we set $G_{interval} = 500$.

To compare results with GPU computation, we designed two types of GA on CPU, GA-1 and GA-2. GA-1 is logically identical with parallel GA on the GPU described in Section 3. It consists of $30 \times k (k = 1, 2, ...)$ subpopulations and each subpopulation has 128 individuals. Every 500 generation interval, all individual of $128 \times 30 \times k$ are shuffled. GA-2 also consists of $30 \times k (k = 1, 2, ...)$ subpopulations. But only 5% of individuals are randomly exchanged among subpopulations at every 50 generation interval. Please note here that all of these subpopulations are processed by a single thread on the CPU in GA-1 and GA-2. We did not use the SSE instructions in CPU computation. In all GA models, we used mutation rate of p_m=0.1.

In the GPU computation, We used *float*, whereas in the GA-1 and GA-2 CPU computation, we used *double*. However, in our GAs, using floating point values are limited. They were used only for mutation rate (p_m) and we guess there are no meaningful difference in reaching results between using *float* and *double*.

4.2 Results

Results are summarized in Table 1. Before seeing the results of GPU computation, let's review the difference in values of T_{avg} between GA-1 and GA-2. As described in 4.1, GA-1 has the same distributed GA model with GA on the GPU though it runs as a single thread on the CPU. As a distribution GA, GA-2 is a more common approach than GA-1. We can see values of T_{avg} of GA-1 are bigger than GA-2 with one exception (kra30a).

Now let's see the results of GPU computation compared to the results with CPU computations. On tai20b, for example, GPU computation used $128 \times 30 \times 1$ threads (=total number of individuals), obtaining T_{avg} of 0.064. GA-1 and GA-2 used the same number of total individuals. Values of T_{avg} for GA-1 and GA-2 are 0.428 and 0.422, respectively. GPU computation is 6.7 and 6.6 times faster than GA-1 and GA-2, respectively. Although the speedup ratios of GPU computation are different depending on QAP instances and are in the range from 2.9 to 12.6 against GA-2, we can observe a definite speedup with GPU computation on instances in this study.

5. CONCLUSIONS

As a new parallel computation scheme, recently there has been a growing interest in GPU computation. In this paper, we designed a parallel GA with GPU computation to solve the quadratic assignment problem (QAP) based on multiple-population coarse-grained GAs. Although there is still much future work to go related to this study, our results were promising, showing a speedup ration from 3 to 12 times, compared to the Intel ® Core™ i7 965 processor, which is one of the fastest processors available currently. The following remains for future work:

(1) In the current model, we shuffled all individuals in GPU with interval $G_{interval} = 500$. This shuffling was performed by CPU. This is time consuming and increases computation overhead. It should be possible to implement a more efficient distributed GA model easily.

(2) Although in this implementation we did not use the *texture memory*, using it for working memory would make the program run faster.

(3) Although we used multiple-population coarse-grained GAs in this study, using a single-population, fine-grained model still remains a promising direction.

(4) In this study, we did not apply any local search. However, using local search should increase the performance a great deal. Thus, to incorporate local search is another promising research direction.

(5) An additional direction might be using a master-slave GA, in which GPU performs mainly the local search.

From our early experience, we want to make some proposals for future GPU configuration as it applies to GAs. First, larger shared memory would greatly improve the performance and the applicability of GPUs for larger problems. Further, it would enable us to use a wider variety of GA models. Second, current CUDA allocates local array variables in VRAM. Thus, using a local array reduces the computation speed a great deal. We propose that the array can be located in the registers. In evolutionary computations, much data are represented by array, this would make it easier to develop more efficient programs.

6. ACKNOWLEDGMENTS

We would like to thank Dr. M. Umano, Prof. Of Osaka Prefecture University, for his useful comments on this research. This research is partially supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research number 19500199 and was supported in part by Grant-in-Aid for Young Scientists (B)(18700058) from the Japan Society for the Promotion of Science.

7. REFERENCES

[1] QAPLIB - a quadratic assignment problem library, 2009. http://www.seas.upenn.edu/qaplib.

[2] W. Banzhaf, S. Harding, W. Langdon, and G. Wilson. Accelerating Genetic Programming through Graphics Processing Units. Springer, 2008.

[3] E. Cantú-Paz. Efficient and Accurate Parallel Genetic Algorithms. Kuwer Academic Publishers, 2000.

[4] T. Clayton, L. Patel, G. Leng, A. Murray, and I. Lindsay. Rapid evaluation and evolution of neural models using graphics card hardware. In Proceedings of the 10th annual conference on Genetic and evolutionary computation, 2008.

[5] K. Fok, T. Wong, and M. Wong. Evolutionary computing on consumer-level graphics hardware. IEEE Intelligent Systems, 22(2), 2007.

Table 1: Experimental results of GPU computation. The results are compared with CPU computation with single thread.

QAP instances	GPU computation				CPU computation with single thread								Speedup ratio to CPU computation	
					GA-1				GA-2					
	Total population	*#OPT*	T_{avg} *(sec)*	*std*	Total population	*#OPT*	T_{avg} *(sec)*	std	Total population	*#OPT*	T_{avg} *(sec)*	std	GA-1	GA-2
tai20b	128×30×1	10	0.064	0.005	128×30×1	10	0.428	0.039	128×30×1	10	0.422	0.042	**6.7**	**6.6**
tai25b	128×30×1	10	0.169	0.015	128×30×1	10	1.386	0.135	128×30×1	10	1.286	0.145	**8.2**	**7.6**
kra30a	128×30×5	10	2.002	1.741	128×30×2	9	9.651	4.541	128×30×4	9	11.870	3.115	**4.8**	**5.9**
kra30b	128×30×5	9	1.332	0.732	128×30×5	8	23.399	11.492	128×30×4	9	16.745	11.164	**17.6**	**12.6**
tai30b	128×30×3	10	0.947	0.576	128×30×3	10	22.649	6.830	128×30×1	10	7.203	6.274	**23.9**	**7.6**
tai35b	128×30×4	10	2.510	0.740	128×30×3	10	22.649	6.830	128×30×1	10	7.203	6.274	**9.0**	**2.9**
ste36b	128×30×4	10	3.337	1.056	128×30×4	10	33.274	13.062	128×30×2	10	14.675	3.836	**10.0**	**4.4**
tai40b	128×30×1	10	1.088	0.087	128×30×1	9	6.016	0.486	128×30×1	10	5.811	0.482	**5.5**	**5.3**

#OPT : the number of runs in which the algorithm succeeded in finding the optimal solution
T_{avg} : the average time to find optimal solutions in successful runs in second
std : standard deviation of T_{avg}

[6] N. Fujimoto. Dense matrix-vector multiplication on the CUDA architecture. Parallel Processing Letters, 18(4), 2008.

[7] N. Fujimoto. Faster matrix-vector multiplication on GeForce 8800GTX. In Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2008.

[8] D. Goldberg. Genetic algorithms in search, optimization and machine learning. Addison-Wesley publishing company, 1989.

[9] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan. Gpu implementation of belief propagation using cuda for cloud tracking and reconstruction. In Proceedings of the 5th IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS), 2008.

[10] K. Gulati and S. Khatri. In Proceedings of the 2009 Conference on Asia and South Pacific Design Automation, 2009.

[11] T. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of gpus and multicores. In Proceedings of the 22nd annual international conference on Supercomputing, 2008.

[12] T. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. Econometrica, 25, 1957.

[13] S. Lahabar and P. Narayanan. Singular value decomposition on gpu using cuda. In Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS), (to appear), 2009.

[14] W. Langdon and W. Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In Proceedings of the 11th European Conference on Genetic Programming. Springer, 2008.

[15] W. Langdon and A. Harrison. Gp on spmd parallel graphics hardware for mega bioinformatics data mining. Soft Computing - A Fusion of Foundations, Methodologies and Applications, 12(12), 2008.

[16] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. IEEE Micro, 28(2), 2008.

[17] Y. Liu, E. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS) (to appear), 2009.

[18] P. Maciol and K. Banas. Testing tesla architecture for scientific computing: The performance of matrix-vector product. In Proceedings of the International Multiconference on Computer Science and Information Technology, 2008.

[19] S. Mahfoud. A comparison of parallel and sequential niching methods. In Procedings of the Six International Conference on Genetic Algorithms. Morgan Kaufmann, 1995.

[20] S. Manavski. Cuda compatible gpu as an efficient hardware accelerator for cryptography. In Proceedings of the IEEE International Conference on Signal Processing and Communication (ICSPC 2007), 2007.

[21] V. Maniezzo and A. Colorni. The ant system applied to the quadratic assignment problem. IEEE Transactions on Knowledge and Data Engineering, 11(5), 1999.

[22] S. Matsuoka. The rise of the commodity vectors. In Proceedings of the 8th International Meeting High Performance Computing for Computational Science (VECPAR), volume LNCS 5336, 2008.

[23] NVIDIA, 2009. http://www.nvidia.com/page/home.html.

[24] I. Oliver, D. Smith, and J. Holland. A study of permutation crossover operators on the travel salesman problem. In Proceedings of the 2nd International Conference on Genetic Algorithms. L. Erlbaum Associates Inc., 1987.

[25] M. Pelikan, S. Tsutsui, and R. Kalapala. Dependency trees, permutations, and quadratic assignment problem. In Proceedings of the Genetic and

Evolutionary Computation Conference (GECCO 2007). ACM, 2007.

[26] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel gp on the g80 gpu. In Proceedings of the 11th European Conference on Genetic Programming. Springer, 2008.

[27] S. Sahni and T. Gonzalez. P-complete approximation problems. Journal of the ACM, 23, 1976.

[28] M. Schellmann, J. Vörding, S. Gorlatch, and D. Meiländer. Cost-effective medical image reconstruction: from clusters to graphics processing units. In Proceedings of the 2008 conference on Computing frontiers, 2008.

[29] T. Stützle and H. Hoos. Max-min ant system. Future Generation Computer Systems, 16(9), 2000.

[30] É. Taillard. taboo search code. 2004. http://mistic.heig-vd.ch/taillard/codes.dir/tabou_qap.cpp.

[31] S. Tsutsui. Parallel ant colony optimization for the quadratic assignment problems with symmetric multi processing. In Proceedings of the Sixth International Conference on Ant Colony Optimization and Swarm Intelligence. Springer, 2008.

[32] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesman problem: The genetic edge recombination operator. In Proceeding of the 3rd International Conference on Genetic Algorithms. Morgan Kaufmann, 1989.

[33] G. Wilson and W. BanzhafLinear. Linear genetic programming gpgpu on microsoft's xbox 360. In Proceedings of the IEEE Congress on Evolutionary Computation 2008 (CEC'08), 2008.

[34] M. Wong and T. Wong. Parallel hybrid genetic algorithms on consumer-level graphics hardware. In Proceedings of IEEE Congress on Evolutionary Computation 2006 (CEC'06), 2006.