An Analytical Study of GPU Computation for Solving QAPs by Parallel Evolutionary Computation with Independent Run

Shigeyoshi Tsutsui, *Member, IEEE* and Noriyuki Fujimoto, *Member, IEEE*

***Abstract*— This paper proposes an evolutionary algorithm for solving QAPs with parallel independent run using GPU computation and gives a statistical analysis on how speedup can be attained with this model. With the proposed model, we achieve a GPU computation performance that is nearly proportional to the number of equipped multi-processors (MPs) in the GPUs. We explain these computational results by performing statistical analysis. Regarding performance comparison to CPU computations, GPU computation shows a speedup of x4.4 and x7.9 on average using a single GPU and two GPUs, respectively.**

I. Introduction

Recently, parallel computations using GPUs (Graphics Processing Units) have 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 more than tens to hundreds of times compared to CPU computations. However, GPU computations have only just started to be used in the evolutionary computation field.

In evolutionary computation, many applications consume their run time for evaluation of individuals. Thus, in evolutionary computation, the approach in which populations are managed by CPUs and evaluation tasks are assigned to GPUs is a good approach though some overhead time for the transfer of individuals between the CPU and GPU is needed. A typical example of this approach is reported by Maitre et al. [2], in which they showed a successful speedup of x60 in solving a real-world material-science problem.

On the other hand, there are applications where the evolutionary algorithm itself must be performed fast. Typical examples of such applications are combinatorial optimization problems such as routing problems and assignment problems. In this class of problems, computation time for the evaluation of individuals is not dominant. In this case, it is useful that evolutionary operations are performed on GPUs using parallel evolutionary models, such as fine-grained or coarse-grained models [3], [4]. However with this approach, the speedup is at most x10.

In a previous study [5], we applied GPU computation to solve quadratic assignment problems (QAPs) with parallel evolutionary computation using a coarse-grained model on a single GPU. The results in that study showed that parallel evolutionary computation with the NVIDIA GeForce GTX285 GPU produce a speedup of x3 to x12 compared to the Intel Core i7 965 (3.2 GHz). However, the analysis of the results was postponed for future work.

In this study, we propose a simplified parallel evolutionary model with independent run and analyze how the speedup is obtained using a statistical model of parallel runs of the algorithm. The basic idea of the parallel independent run is as follows. A set of small-size subpopulations is run in parallel in each block in CUDA [1]. The computation time to get an acceptable solution is different in each block because the computations are performed probabilistically using pseudo random numbers with different seeds. If one subpopulation finds an acceptable solution first, then we stop executions of all subpopulations. This time is shorter than the average time where we run all subpopulation until the acceptable solution is obtained.

In the remainder of this paper, Section II describes a brief review of GPU computation and its application to evolutionary computation. Then, the parallel GA model with independent run to solve QAPs on GPUs is described in Section III. Section IV describes computational results and gives analysis. Finally, Section V 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 as shown in Fig. 1. Processors in a CUDA GPU are grouped into multiprocessors (MPs). Each MP consists of 8 thread processors (TPs). TPs in a MP exchange data via fast 16KB shared memory (SM). On the other hand, data exchange between 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. Although processors have no data cache for VRAM (except for the constant and texture memory areas), SM can be used as manually controlled data cache.

The CUDA programming model is a multi-threaded programming model. In Fig. 2, we describe an overview of the CUDA programming model. In a CUDA program, threads form two hierarchies: a *grid* and *thread blocks*. A thread block is a set of threads. A thread 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*

Shigeyoshi Tsutsui is with Department of Management and Information Sciences, Hannan, 5-4-33, Amammihigashi, Matsubara, Osaka 580-8502 Japan (phone: +81 72 332 1224; email: tsutsui@hannan-u.ac.jp).

Noriyuki Fujimoto is with Graduate School of Science, Osaka Prefecture University, 1-1 Gakuenmachi, Nakaku, Sakai, Osaka 599-8531, Japan (phone: +81 72 252 1161; email: fujimoto@mi.s.osakafu-u.ac.jp).

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

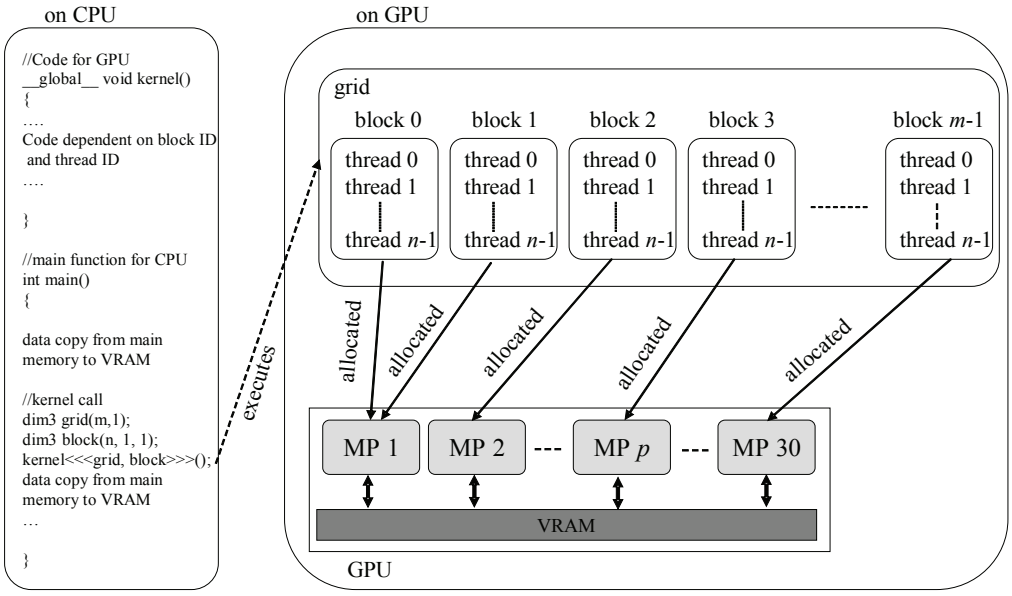

Fig. 2. An overview of CUDA programming model

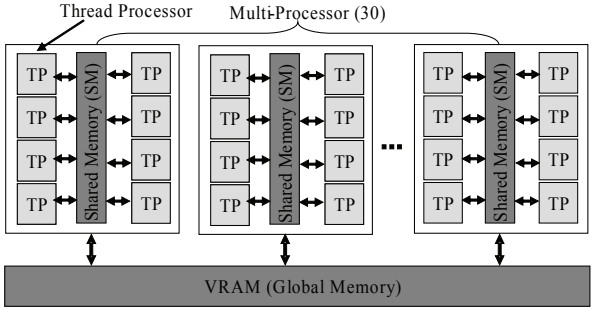

Fig. 1. CUDA architecture as a two-level shared-memory machine

function. A kernel function call generates threads as a grid with given dimensionality and size.

VRAM bandwidth is very high at 159 GB/sec. However, when accessing VRAM, there is memory latency as large as 100 to 150 arithmetic operations [1]. 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 MP are divided among concurrent thread blocks allocated for the MP. VRAM has a read-only region of size 64KB [1]. The region is called the *constant memory*. The constant memory space is cached. The cache working set for constant memory is 8 KB per MP [1].

B. Applications of GPU Computation to the Evolutionary Computation

There are many parallel computations in the scientific computation fields [1]. For example, cloud tracking [6], statistical static timing analysis [7], biomedical image analysis [8], AES cryptography encoding and decoding [9], medical image construction [10].

Some studies on GPU computation can be found in the evolutionary computation field, especially in genetic programming (GP), as shown in Banzhaf et al.'s intensive survey [11]. Langdon et al. showed the feasibility of evaluating genetic programming populations directly using a GPU [12]. Langdon & Harrison built a SIMD C++ GP system on a GPU to predict a ten year+ outcome of breast cancer from a dataset containing a million inputs [13]. Robilliard et al. showed it is possible to efficiently interpret several GP programs in parallel [14]. Wilson & Banzhaf implemented a GP system on a commercial video game console [15].

A few studies on parallel evolutionary computation with GPU computation are reported. Fok et al. showed early results using an nVidia GeForceFX 6800 with parallel Evolutionary Programming (EP) [3]. It was found that the speedup factor of their parallel EP ranges from x1.25 to x5.02, when the population size is large enough. Wong, M. L. & Wong, T. T reported on a parallel hybrid genetic algorithm (HGA) on consumer-level graphics cards [16]. HGA extends the classical genetic algorithm by incorporating the Cauchy mutation operator from evolutionary programming.

To evaluate and evolve neural models, Clayton et al. proposed a Distributed Adaptive Genetic Algorithm (DAGA), which is suitable to the large population sizes promoted by the GPU architecture [17]. Wong proposes implementing a parallel MOEA within the CUDA environment on an nVidia GPU [18]. The speedups of his parallel MOEA range from x5.62 to x10.75.

In [2], Maitre, et al. propose a hybrid approach which combines CPU and GPU. They showed successful speedup of x60 in solving a real-world material-science problem (see Section I). Implementing a fast pseudo-random number generator is important in implementing parallel evolutionary

computation on GPU. Langdon proposed a fast, high-quality pseudo-random number generator for nVidia CUDA [19].

III. Parallel GA model with Independent Run to Solve Quadratic Assignment Problems on GPU

Here, we propose a parallel independent evolutionary model to solve quadratic assignment problems (QAPs).

A. Quadratic Assignment Problem (QAP)

In QAP, we are given L locations and L facilities and the task is to obtain an assignment Φ which minimizes cost as defined in the following equation.

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

where d_{ij} is the distance matrix which represents distance between locations i and j and , f_{ij} is the flow matrix which represents flow between facilities i and j. The QAP is an NP-hard optimization problem [20] and it is considered one of the hardest optimization problems.

B. Evolutionary Model for GPU Computation for QAP

1) The base GA model for QAP: The base evolutionary model for QAP in GPU computation is the same as was used in our previous study [5]. Fig. 3 shows the base evolutionary 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 store individuals of the current population, and W is a working pool to store newly generated offspring individuals until they are selected to update P for the next generation. The algorithm is performed 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_j ($j \neq i$) 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 I_j 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 [21].

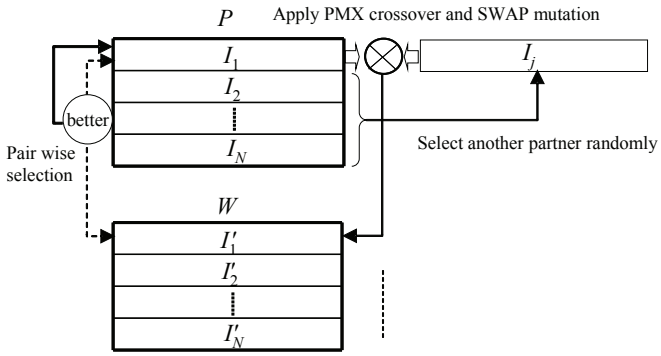

Fig. 3. The base evolutionary mode for QAP

From the viewpoint of parallelization, this model also has advantage because a process for one individual in a generation cycle can be implemented in a parallel thread easily. Using one child from two parents was already proposed for designing the well known GENITOR algorithm by Whitley et al. [22]. For crossover operators, we performed preliminary tests using two well known operators, i.e, the order crossover (OX) [23] and the partially mapped crossover (PMX) [24] by implementing the GA model both on CPU and GPU. The results showed the PMX operator worked much better than the OX operator on QAPs. Thus, in the experiment in IV, we will use 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 a mutation rate of p_m. Applying local search in solving QAP is very common in evolutionary algorithms [25], [26], [27]. One of the main purpose of this research is to analyze the effect of the parallel evolutonary computation on GPUs, we will not apply any local search in this study.

2) Parallel evolutionary model for GPU computation with independent run: The NVIDIA GeForce GTX285 GPU which we use in this study has 30 MPs and each MP has 8 TPs sharing 16KB high speed SM among them. For each subpopulation, we use the base evolutionary model described in III.B. So, we allocate the population pools P and W described in Fig. 3 to the SM of each MP.

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, and maximum problems sizes of QAP are at most 150. 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 pool P and working pool W is $2L \times N$ where N is the number of individuals allocated to each MP at the same time; i.e. subpopulation size. We have only 16KB shared memory per MP. To maximize both L and N, we chose N as 128 under the assumption that L is at most 56. Consequently, for W and P, our implementation consumes $2L \times N = 2 \times 56 \times 128 =$

14336 bytes in SM at most. From the CUDA programming scheme, we generate p blocks and each block consists of 128 threads.

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

In our previous study [5], individuals in each block are exchanged among blocks every 500-generation interval as follows: (1) In the host machine, all individuals are shuffled. Then, they are sent to the VRAM of the GPU. (2) Each MP selects a block not yet processed and copies the corresponding individuals from VRAM to its SM, performs the generational process up to 500 generations, and finally copies the evolved individuals from its SM to VRAM. (3) The above process is repeated until all blocks 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.

In this study, we evolve subpopulations of each block independently without exchanging individuals among blocks. If a subpopulation in a block has found an acceptable solution, then it sets "FoundFlag" to 1 (the initial value being set to 0) in VRAM which can be shared by all blocks, sends the solution to the VRAM, and then terminates the execution. All subpopulations in other blocks check the flag at every generation whether the flag is set or not. If they find the flag is set by another block, then they terminate their executions, otherwise, they continue. In this way, with this parallel independent run model, execution of the algorithm terminates if one of the subpopulations in the blocks find an acceptable solution.

The subpopulation size of 128 is relatively small for solving QAP and independent run with this population size often causes stagnation of evolution in the subpopulations. To prevent the stagnation in the subpopulation, we introduce a *restart strategy* as described in the following subsection.

3) Implementing restart strategy: Restart strategies have been discussed elsewhere. The delta coding method used by Mathias and Whitley is an iterative genetic search strategy that sustains search by periodically reinitializing the population [28]. It also remaps the search hyperspace with each iteration and it is reported that it shows good performance especially when used with Gray coding. The CHC method by Eshelman is a safe search strategy where restart of the search process is done if it gets stuck at local optima by re-initializing the population with individuals generated by mutating the best solution obtained so far (also keeping the best one) [29].

In another study, Tsutsui et al. proposed a search space division and restart mechanism to avoid getting stuck in local traps in the framework referred to as *the bi-population GA scheme (bGA)* [30]. Although the use of a restart strategy is a feature of the bGA, its main purpose is to maintain a suitable balance between exploration and exploitation during the search process by means of two populations.

The approach taken in this study is similar to CHC. Let the best functional value in a subpopulation be represented by f_{c-best}. In each generation, we count the number of individuals whose functional values are equal to f_{c-best}, and represent the number by B_{count}. If B_{count} is lager than or equal to $N \times B_{rate}$, then we assume the subpopulation is trapped at local solutions and the subpopulation is re-initialized. According to a preliminary study, we found that the approach of keeping the best individual during re-initialization causes ill effect. So, in the re-initialization, all member of the subpopulation are generated anew. We used $B_{rate} = 0.6$.

In a block, each individual is processed as an independent thread, so to get B_{count}, we need to use instructions which control exclusive and synchronous executions. In this study, we used *atomic instructions* which work on the SM. These instructions are available in a GPU with compute capability 1.2 or later. Since this counting takes some additional overhead time, the checking current B_{count} is performed only every 50 generations. Fig. 4 shows the pseudo code for each block.

TABLE I shows the effect of the restart strategy. Results of this table were obtained by emulating a single block run using a CPU (Intel Core i7965 3.2 GHz). We used a mutation rate of $p_m = 0.1$. From this table, we can see a clear effect as a result of using the restart strategy.

TABLE I
EFFECT OF RESTART STRATEGY

QAP Instances	no restart				restart			
	#OPT	T_{avg}	Min	Max	#OPT	T_{avg}	Min	Max
tai25b	6	0.07	0.05	0.10	30	0.07	0.05	3.77
kra30a	1	0.27	0.27	0.27	30	10.03	0.27	61.13
kra30b	0	-	-	-	30	25.20	0.45	92.24
tai30b	0	-	-	-	30	3.96	0.39	12.53
kra32	0	-	-	-	30	10.70	0.33	59.52
tai35b	0	-	-	-	30	29.69	2.53	137.72
ste36b	0	-	-	-	30	17.00	0.73	45.10
tai40b	1	0.30	0.30	0.30	30	6.96	0.30	20.96
tai50b	0	-	-	-	30	48.07	1.12	147.37

#OPT: Number of success run in 30 runs
T_{avg}: Average time to find acceptable solutions in success runs in second

IV. COMPUTATIONAL RESULTS AND ANALYSIS

A. Experimental Conditions

In this study, we used a PC which has one Intel Core i7 965 (3.2 GHz) processor and two NVIDIA GeForce GTX285 GPUs. The OS was Windows XP Professional with NVIDIA graphics driver Version 195.62. For CUDA program compilation, Microsoft Visual Studio 2008 Professional Edition with optimization option /O2 and CUDA 2.3 SDK were used. The two GPUs were controlled using thread programming provided in the Win32 API.

The instances on which we tested our algorithm were taken from the QAPLIB benchmark library at [31]. QAP instances

```
//Code for GPU
__global__ void kernel()
{
        int threadID = POOL_SIZE * blockIdx.x + threadIdx.x;
        get seed of random number generator for threadID from VRAM;
        t=0;
        initialize string with the seed;
        evaluate the string;
        for(int t=1; t<MaxGeneration; t++){
                __syncthreads();
                reset memory contents related restart;
                Evolutionary Code for the thread threadID;
                ...

                if (this thread (individual) found an acceptable solution){
                      set FoundFlag in VRAM to 1 using "atomicCAS" instruction;
                      if(this thread has set)
                            write the solution (string) to VRAM;
                }
                __syncthreads();
                if(FoundFlag in VRAM ==1)
                      break; //an acceptable solution has found
                if(t%50==0){
                       atomicMin(fc_best, perf);//get minimum value at s_minValue in SM
                        __syncthreads();
                       if(perf == *fc_best)//if this thread have minimum functional value
                            atomicAdd(B_count, 1);//add 1 to B_count in SM
                            __syncthreads();
                            if(threadIdx.x==0){//check restart condition by thread with id 0
                                  if(*B_count > (int)(POOL_SIZE*B_rate+0.5))//satisfy?
                                        *restart_flag = 1;
                                        //inform other thread that the restart condition has satisfied
                                  }
                                  __syncthreads();
                                  if(*restart_flag == 1){//restart?
                                        initialize string of this thread;
                                        evaluate;
                                        __syncthreads();
                                  }
                            }
                            else
                                  __syncthreads();
                }
        }
}
//main function for CPU
int main()
{
         ....
        data copy from main memory to VRAM
        //kernel call
        dim3 grid(128,1);
        dim3 block(P, 1, 1);
        kernel<<<grid, block, SMsize >>>();
        copy from main memory to VRAM
          ....
}
```

Fig. 4. Pseudo code for parallel independent run in CUDA

in the QAPLIB 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 [25]. In this experiment, we used the following 9 instances which were classified as either iii) or iv) with the problem size ranging from 25 to 50; tai25b, kra30a, kra30b, tai30b, kra32, tai35b, ste36b, tai40, and tai50b.

30 runs were performed for each instance. We measured the performance by the average time to find acceptable solutions as measured by CPU (wall clock). Acceptable solutions for all instances except tai50b were set to known optimal values. For tai50b, we set the acceptable solution be within 0.06% of the known optimal solution. We represent the average time over 30 runs as $T_{p,avg}$ where p is the number of MPs. We used a mutation rate of $p_m = 0.1$ in all experiments.

B. Distribution of Computation Time on a Single Block

Here we performed 100 runs on one GPU using a single block until acceptable solutions were obtained. Fig 5 shows the distribution of the run time until an acceptable solution was obtained for each instance. Each asterisks indicates the time in each run and black boxes indicates their average time ($T_{1,avg}$). Please refer to TABLE II for each value of $T_{1,avg}$ and its standard deviation. From this figure, if we run the algorithm using $p(p > 1)$ blocks simultaneously it is clear that the average runtime is reduced, i.e., $T_{p,avg} < T_{1,avg}$ for $p > 1$.

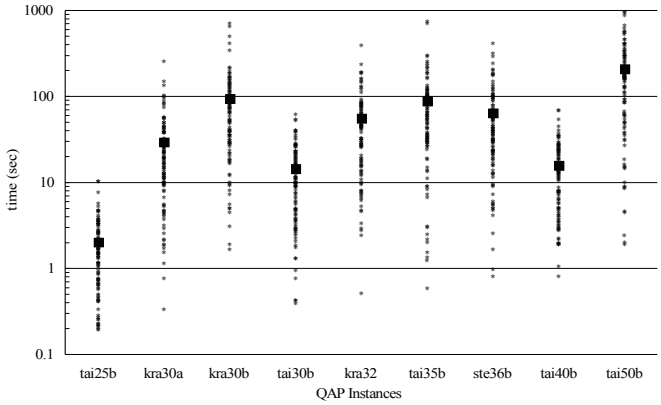

Fig. 5. Distribution of time to find acceptable solution on a single block runs

C. Numerical Results

Since our system has two GPUs, we performed two experiments, one was to use a single GPU with 30 blocks ($p = 30$) and the other was to use two GPUs with 60 blocks ($p = 60$). In the CUDA programming model, it is possible to run our algorithm using many blocks such that the number of blocks is more than the number of equipped MPs. However, doing this could cause side effects in performance in our parallel evolutionary model due to hardware resource limits. The results are summarized in TABLE II.

In the table, $gain_p$ indicates $T_{1,avg}/T_{p,avg}$, the run time gain obtained by p-block parallel runs to single block runs (please note that the single block runs are taken from the experiment in IV-B and are an average over 100 runs). For example, on tai25b, $gain_{30} = 9.56$ and $gain_{60} = 10.85$, respectively, showing not so high a gain. However, on kra30a, $gain_{30} = 25.43$ and $gain_{60} = 49.10$, respectively. The values of $gain_p$ are different from instance to instance, they are in the range [10, 35] for $p = 30$, and [10, 70] for $p = 60$, and are nearly proportional to p, except for instances tai25b and tai40b. In these two instances, the gains are smaller. Thus, with some exception, we can clearly confirm the effectiveness of the parallel independent run model.

D. Analytical Estimation of Speed with Parallel Independent Runs

In this subsection, we analyze the results in IV-C from a statistical perspective. Let the probability density function of

TABLE II

RESULTS OF PARALLEL INDEPENDENT RUNS AND STATISTICAL ESTIMATION

GPU	Instances	tai25b					kra30a					kra30b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p
GPUx 1	1	2.02	1.89	1.00	-	-	34.25	36.79	1.00	-	-	113.69	113.36	1.00	-	-
	30	0.21	0.06	9.56	0.03	0.18	1.35	1.23	25.43	0.79	0.56	3.17	2.52	35.85	2.92	0.25
GPUx 2	60	0.19	0.02	10.85	0.00	0.18	0.70	0.47	49.10	0.34	0.36	1.63	1.48	69.59	1.21	0.42
GPU	Instances	tai30b					kra32					tai35b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p
GPUx 1	1	14.31	13.64	1.00	-	-	56.18	61.11	1.00	-	-	92.08	75.90	1.00	-	-
	30	0.71	0.46	20.24	0.45	0.25	2.13	1.80	26.35	1.78	0.35	3.67	3.56	25.12	3.25	0.41
GPUx 2	60	0.46	0.16	30.88	0.16	0.30	1.12	0.93	50.11	0.83	0.29	1.65	1.38	55.68	1.66	-0.01
GPU	Instances	ste36b					tai40b					tai50b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p	$T_{p,avg}$	σ	$gain_p$	$M(P)$	Δ_p
GPUx 1	1	70.82	73.23	1.00	-	-	19.07	16.43	1.00	-	-	212.55	203.64	1.00	-	-
	30	2.57	1.89	27.54	1.63	0.94	1.15	0.62	16.56	0.46	0.69	8.75	8.65	24.29	6.19	2.56
GPUx 2	60	1.35	0.77	52.40	0.66	0.70	0.90	0.16	21.27	0.12	0.78	4.28	3.84	49.62	2.72	1.56

σ : standard deviation

Γ : Results with Gamma distribution estimation

Values of $T_{p,avg}$ *and M(P)* are in second

the run time on a single block be represented by $f(t)$ and probability distribution function of $f(t)$ by $F(t)$, where

$$F(t) = \int_0^t f(t)dt. \tag{2}$$

Here, consider a situation where we run p blocks in parallel independently and if a block finds an acceptable solution while other blocks are still running, we stop the GPU computation immediately as described in III-B.2. Let the probability density function of the runtime on p blocks be represented by $g(p, t)$, probability distribution function of $g(p, t)$ by $G(p, t)$, and the mean time by $M(p)$. Then, $g(p, t)$ can be simply obtained as follows. Since the probability that all p blocks cannot find acceptable solutions at t is $(1-F(t))^p$, the probability distribution function $G(p, t)$ can be obtained as

$$G(p,t) = 1 - (1 - F(t))^p, \tag{3}$$

and its probability density function $g(t)$ is obtained as

$$g(p,t) = \frac{dG(p,t)}{dt}, \tag{4}$$

and $M(p)$ is

$$M(p) = \int_0^\infty t \cdot g(p,t)dt. \tag{5}$$

We estimated the distributions in Fig. 5 by the normal distribution and the gamma distribution using the least-square method. Results showed that the gamma distribution reflected the distributions well as shown in Fig. 6. As we can see in the distributions in Fig 5, the distributions are not symmetrical against mean values i.e., they distribute tighter in smaller t (please note the vertical axis is log scale). This is the reason that the normal distribution does not reflect the distributions. By applying Eq. (5) to the estimated distributions in Fig. 6, we calculated $M(p)$ for $p = 30$ and 60. These results are shown in TABLE II. Seeing these results, we can see that they reflect the results obtained by GPU computation ($T_{p,avg}$). However, the values of $M(p)$ are slightly smaller than the corresponding $T_{p,avg}$ with the exception of tai35b ($p = 60$). We showed the differences of both values by Δ_p. Although the Δ_p values are different among instances, we can see larger size instances have larger Δ_p values than smaller size instances.

Now consider here why these differences arise. In our estimation in Eq. (3), we assumed that there is no additional overhead time even if we run p ($p > 1$) blocks in parallel. Detailed hardware performance information from the manufacturer is not open to us. But we can recognize that some additional overhead time must arise if we run multiple blocks simultaneously. One possible scenario we can consider is overhead caused by constant memory access conflict. In our implementaion, QAP data (distance matrix and flow matrix, please see III-A) are stored in the constant memory. The constant memory has 8KB cache memory for each MP, but

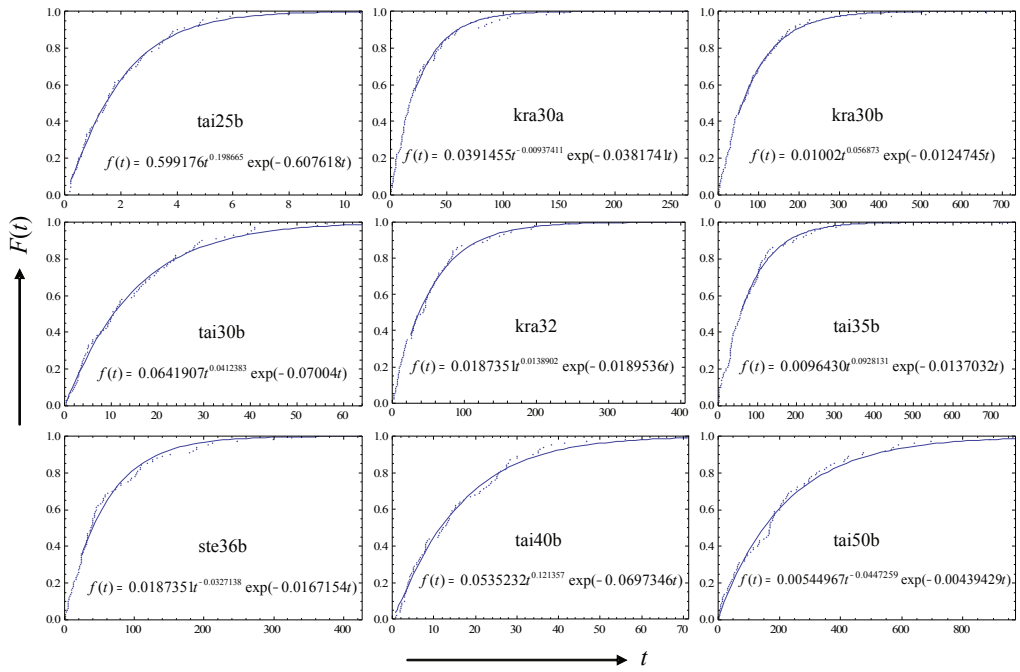

Fig. 6. Estimation of distribution by Gamma distribution

some access conflict occurs when data are transferred from constant memory to cache memory.

Although there exist some differences (Δ_p) between real GPU computation and our statistical analysis values, this analysis does represent the theoretical elements of speedup by the parallel independent run.

E. Comparison with CPU Computation

To compare the results of GPU computation and CPU computation, we measured the CPU computation time of the same tasks. The CPU was an Intel Core i7 965 (3.2 GHz) processor running Windows XP Professional. For CPU computation, we used the same evolutionary model as described in Fig. 3. Other conditions, such as crossover operator, mutation operator, and restart strategy, are also the same. A single population was used on the CPU. We tuned population size for each instance, by testing population sizes of 64, 128, 256, 512, 1024, and 2048. We used a population size which shows the shortest mean time to get an acceptable solution over 30 runs (T_{avg}). We refer to this CPU computation as Model I. We also show the results of CPU computation which we refer to as Model II, where population size is fixed to 128, which is the same as in GPU computation.

Results are summarized in TABLE III. On tai50b, the speedup of the GPU is x3.9 and x4.4 compared against Model I, using a single GPU and two GPUs, respectively. These results are not so promising. On kra30b however, the speedup of the GPU is x7.9 and x15.4 compared against Model I, using a single GPU and two GPUs, respectively, showing promising results. On average, we got a speedup of x4.4 and x7.9 compared against Model I, using a single GPU and two GPUs, respectively. The speedup values of the GPU against Model II are larger than or equal to those against Model I.

TABLE III
COMPARISON BETWEEN GPU COMPUTATION AND CPU COMPUTATION WITH A SINGLE THREAD

QAP instances	GPU Computaion		CPU Computation			*speedup* to Model I		*speedup to* Model II	
			Model I		Model II				
	GPUx 1 ($T_{30,avg}$)	GPUx 2 ($T_{60,avg}$)	CPU (T_{avg})	Population Size	CPU (T_{avg}) with Population size 128	GPUx 1	GPUx 2	GPUx 1	GPUx 2
tai25b	0.21	0.19	0.82	128	0.82	3.9	4.4	3.9	4.4
kra30a	1.35	0.70	6.64	1024	21.45	4.9	9.5	15.9	30.7
kra30b	3.17	1.63	25.20	128	25.20	7.9	15.4	7.9	15.4
tai30b	0.71	0.46	2.05	512	3.96	2.9	4.4	5.6	8.5
kra32	2.13	1.12	10.70	128	10.70	5.0	9.5	5.0	9.5
tai35b	3.67	1.65	12.16	512	29.64	3.3	7.4	8.1	17.9
ste36b	2.57	1.35	15.07	256	16.99	5.9	11.1	6.6	12.6
tai40b	1.15	0.90	4.44	512	6.98	3.9	5.0	6.1	7.8
tai50b	8.75	4.28	18.76	512	48.07	2.1	4.4	5.5	11.2

Values of $T_{30,avg}$, $T_{60,avg}$ *and* $M(P)$ are in second

To obtain higher speedup values, we need to improve the implementation of genetic operators used in each thread in the blocks. Fig. 7 shows the code of the PMX operator used in this study. The flow is the same as is used in CPU implementation. Here, cut1 and cut2 are cut-points and have different values among threads. Thus, the loop numbers in the *for* statements in each thread are different from each other. Further, whether the branch condition holds or not in each thread is also different from thread to thread. These differences among threads increase the computation time in a block because each warp of 32 threads is essentially run in a SIMD fashion in a MP; high performance can only be achieved if all of a warp's threads execute the same instruction.

```
//cut1 are cut2 are random number in [0, L-1]. cut1 < cut2 is assumed.
//unsigned char *parent1, *parent2 are strings of the parents.
//unsinged char *child is a new string to be generated.
for (int j = 0; j <L; j++)
   child[j] = parent1[j];
for(int i = cut1; i < cut2 ; i++){
   for(int j = 0 ; j < L; j++){
      if (parent2[i] == child[j]){
         unsigned char tmp = child[i]; child[i] = child[j]; child[j] =tmp;
         break;
      }
   }
}
```

Fig. 7. Code of the PMX operator for each thread

To use the same cut-points among threads in one generation may result in some speedup of the execution of the operator, but we need further consideration of operators suitable for more efficient GPU computation.

V. Conclusions

In this paper, we proposed an evolutionary algorithm for solving QAPs with parallel independent runs using GPU computation and gave an analysis of the results. In this parallel model, a set of small-size subpopulations was run in parallel in each block in CUDA independently. With this scheme, we got a performance of GPU computation that is almost proportional to the number of equipped multiprocessors (MPs) in the GPUs.

We explained these computational results by performing statistical analysis. Regarding performance comparison to CPU computations, GPU computation showed a speedup of x4.4 and x7.9 on average using a single GPU and two GPUs, respectively. We can consider many parallel evolutionary models for GPU computation. To implement these models and analyse them remain for future work.

References

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

[2] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet, "Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA," in *Proc. of the 2009 Genetic and Evolutionary Computation Conference (GECCO-2009)*. Morgan Kaufmann Publishers, 2009.

[3] K. Fok, T. Wong, and M. Wong, "Evolutionary computing on consumer-level graphics hardware," *IEEE Intelligent Systems*, vol. 22, no. 2, 2007.

[4] J.-M. Li, X.-J. Wang, R.-S. He, and Z.-X. Chi, "An efficient fine-grained parallel genetic algorithm based on gpu-accelerated," in *Proc. of the Network and Parallel Computing Workshops*, 2007.

[5] S. Tsutsui and N. Fujimoto, "Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study," in *Proc. of the Genetic and Evolutionary Computation Conference, GECCO (Companion)*. ACM, 2009.

[6] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan, "GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction," in *Proc. of the 5th IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS)*, 2008.

[7] K. Gulati and S. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *Proc. of the 2009 Conference on Asia and South Pacific Design Automation*, 2009.

[8] 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 *Proc. of the 22nd annual international conference on Supercomputing*, 2008.

[9] S. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for cryptography," in *Proc. of the IEEE International Conference on Signal Processing and Communication)*, 2007.

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

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

[12] W. Langdon and W. Banzhaf, "A SIMD interpreter for genetic programming on GPU graphics cards," in *Proc. of the 11th European Conference on Genetic Programming*. Springer, 2008.

[13] 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*, vol. 12, no. 12, 2008.

[14] D. Robilliard, V. Marion-Poty, and C. Fonlupt, "Population parallel GP on the G80 GPU," in *Proc. of the 11th European Conference on Genetic Programming*. Springer, 2008.

[15] G. Wilson and W. BanzhafLinear, "Linear genetic programming GPGPU on Microsoft's Xbox 360," in *Proc. of the IEEE Congress on Evolutionary Computation 2008 (CEC'08)*, 2008.

[16] 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.

[17] T. F. Clayton, L. N. Patel, G. Leng, A. F. Murray, and I. A. B. Lindsay, "Rapid evaluation and evolution of neural models using graphics card hardware," in *Proc. of the 10th Conference on Genetic and Evolutionary Computation*, 2008.

[18] M. L. Wong, "Parallel multi-objective evolutionary algorithms on graphics processing units," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO (Companion)*. ACM, 2009.

[19] W. B. Langdon, "High quality pseudo random number generator for nVidia CUDA," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO (Companion)*. ACM, 2009.

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

[21] S. Mahfoud, "A comparison of parallel and sequential niching methods," in *Proc. of the Six International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

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

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

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

[25] T. Stützle and H. Hoos, "Max-min ant system," *Future Generation Computer Systems*, vol. 16, no. 9, 2000.

[26] V. Maniezzo and A. Colorni, "The ant system applied to the quadratic assignment problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 5, 1999.

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

[28] K. E. Mathias and L. D. Whitley, "Changing representation during search: a comparative study of delta coding," *Evolutionary Computation*, vol. 2, no. 3, 1997.

[29] L. J. Eshelman, "The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination," in *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991.

[30] S. Tsutsui, A. Ghosh, D. Corne, and Y. Fujimoto, "A real coded genetic algorithm with an explorer and an exploiter populations," in *Proc. of the 7th International Conference on Genetic Algorithm (ICGA97)*. Morgan Kaufmann Publishers, 1997.

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