A GPU-Based Enhanced Genetic Algorithm for Power-Aware Task Scheduling Problem in HPC Cloud

Nguyen Quang-Hung, Le Thanh Tan, Chiem Thach Phat, Nam Thoai

A GPU-Based Enhanced Genetic Algorithm for Power-Aware Task Scheduling Problem in HPC Cloud

Nguyen Quang-Hung[1], Le Thanh Tan, Chiem Thach Phat, and Nam Thoai[1]

Faculty of Computer Science & Engineering, HCMC University of Technology, VNUHCM
268 Ly Thuong Kiet Street, Ho Chi Minh City, Vietnam
[1]{hungnq2,nam}@cse.hcmut.edu.vn, {50902369, 50901901}@stu.hcmut.edu.vn

Abstract. In this paper, we consider power-aware task scheduling (PATS) in HPC clouds. Users request virtual machines (VMs) to execute their tasks. Each task is executed on one single VM, and requires a fixed number of cores (i.e., processors), computing power (million instructions per second - MIPS) of each core, a fixed start time and non-preemption in a duration. Each physical machine has maximum capacity resources on processors (cores); each core has limited computing power. The energy consumption of each placement is measured for cost calculating purposes. The power consumption of a physical machine is in a linear relationship with its CPU utilization. We want to minimize the total energy consumption of the placements of tasks. We propose here a genetic algorithm (GA) to solve the PATS problem. The GA is developed with two versions: (1) BKGPUGA, which is an adaptively implemented using NVIDIA's Compute Unified Device Architecture (CUDA) framework; and (2) SGA, which is a serial GA version on CPU. The experimental results show the BKGPUGA program that executed on a single NVIDIA® TESLA™ M2090 GPU (512 cores) card obtains significant speedups in comparing to the SGA program executing on Intel® Xeon™ E5-2630 (2.3 GHz) on same input problem size. Both versions share the same GA's parameters (e.g. number of generations, crossover and mutation probability, etc.) and a relative small (10^{-11}) on difference of two finesses between BKGPUGA and SGA. Moreover, the proposed BKGPUGA program can handle large-scale task scheduling problems with scalable speedup under limitations of GPU device (e.g. GPU's device memory, number of GPU cores, etc.).

1 INTRODUCTION

Cloud platforms have become more popular in provision of computing resources under virtual machine (VM) abstraction for high performance computing (HPC) users to run their applications. An HPC cloud is such a cloud platform. Keqin Li [1] presented a task scheduling problems and power-aware scheduling algorithms on multiprocessor computers. We consider here the power-aware task scheduling (PATS) problem in the HPC cloud. The challenge of the PATS problem is the trade-off between minimizing of energy consumption and satisfying Quality of Service (QoS) (e.g. performance or on-time resource availability for reservation requests).

Genetic algorithm (GA) has proposed to solve task scheduling problems [2]. Moreover, GA is one of evolutionary inspired algorithms that are used in green computing [3]. The PATS problem with N tasks (each task requires a VM) and M physical machines can generate M^N possible placements. Therefore, whenever the PATS problem increases its problem size, the computation time of these algorithms to find out an optimal solution or a satisfactory solution is unacceptable.

GPU computing has recently becomes a popular programming model to get high performance on data-parallel applications recently. NVIDIA introduces CUDA parallel computing framework where a CUDA program can run on GeForce®, Quadro®, and Tesla® products. Latest Tesla® architecture is designed for parallel computing and high performance computing. In the newest Tesla architecture, each GPU card has hundreds of CUDA cores and gets multiple Teraflops that targets to high performance computing. For example, a Tesla K10 GPU Accelerator with dual GPUs gets 4.58 teraflops peak single precision [4]. Therefore, study of genetic algorithm on GPU has become an active research topic. Many previous works proposed genetic algorithm on GPU [5][6][7][8]. However, none of these works has studied the PATS. In this paper, we propose BKGPUGA, a GA implemented in CUDA framework and compatible with the NVIDIA Tesla architecture, to solve the PATS problems. The BKGPUGA proposes applying same genetic operation (e.g. crossover, mutation, and selection) and evaluation fitness of chromosomes on whole population in each generation that uses data-parallel model on hundreds of CUDA threads concurrently.

2 Problem Formulation

We describe notations used in this paper as following:

T_i	Task i
M_j	Machine j
$r_j(t)$	Set of indexes of tasks that is allocated on the M_j at time t
$mips_{i,c}$	Allocated MIPS of the c-th processing element (PE) to the T_i by M_j
$MIPS_{j,c}$	Total MIPS of the c-th processing element (PE) on the M_j

We assume that total power consumption of a single physical machine ($P(.)$) has a linear relationship with CPU utilization (U_{cpu}) as mentioned in [9]. We calculate CPU utilization of a host is sum of total CPU utilization on PE_j cores:

$$U_{cpu}(t) = \sum_{c=1}^{PE_j} \sum_{i \in r_j(t)} \frac{mips_{i,c}}{MIPS_{j,c}} \quad (1)$$

Total power consumption of a single host ($P(.)$) at time t is calculated:

$$P\left(U_{cpu}(t)\right) = P_{idle} + (P_{max} - P_{idle}).U_{cpu}(t) \quad (2)$$

Energy consumption of a host (E_i) in period time $[t_i, t_{i+1}]$ is defined by:

$$E_i(\mathrm{t}) = \int_{t_\mathrm{i}}^{t_{\mathrm{i}+1}} [P_{idle} + (P_{max} - P_{idle}).U_{cpu}(t)]dt \quad (3)$$

In this paper, we assume that $\forall \mathrm{t} \in [\mathrm{t_i}, \mathrm{t_{i+1}}]$: $U_{cpu}(t)$ is constant (u_i), then:

$$E_i = [P_{idle} + (P_{max} - P_{idle}).\mathrm{u_i}].(\mathrm{t_{i+1}} - \mathrm{t_i}) \quad (4)$$

Therefore, we obtain the total energy consumption (E) of a host during operation time: $\cup_{i=0,1,2,\ldots}[t_i, t_{i+1}]: E = \sum E_i$

We consider the power-aware task scheduling (PATS) in high performance computing (HPC) Cloud. We formulate the PATS problem as following:

Given a set of n independent tasks to be placed on a set of m physical machines. Each task is executed on a single VM.

The set of n tasks is denoted as: V = $\{T_i(pe_i, mips_i, ram_i, bw_i, ts_i, d_i) | i = 1,...,n\}$

The set of m physical machines is denoted as: M = $\{M_j(PE_j, MIPS_j, RAM_i, BW_j\} | j = 1,...,m\}$

Each i-th task is executed on a single virtual machine (VM_i) requires pe_i processing elements (cores), $mips_i$ MIPS, ram_i MBytes of physical memory, bw_i Kbits/s of network bandwidth, and the VM_i will be started at time (ts_i) and finished at time ($ts_i + d_i$) with neither preemption nor migration in its duration (d_i). We concern three types of computing resources such as processors, physical memory, and network bandwidth. We assume that every M_j can run any VM and the power consumption model ($P_j(t)$) of the M_j has a linear relationship with its CPU utilization as described in formula (2). The objective of scheduling is minimizing total energy consumption in fulfillment of maximum requirements of n tasks (and VMs) and following constraints:

Constraint 1: Each task is executed on a VM that is run by a physical machine (host).

Constraint 2: No task requests any resource larger than total capacity of the host's resource.

Constraint 3: Let $r_j(t)$ be the set of indexes of tasks that are allocated to a host M_j. The sum of total demand resource of these allocated tasks is less than or equal to total capacity of the resource of the M_j. For each c-th processing element of a physical machine M_j (j=1,..,m):

$$\forall c = 1 \ldots \mathrm{PE_j}, \forall i \in \mathrm{r_j(t)}: \sum_{i \in r_j(t)} mips_{i,c} \leq MIPS_{j,c} \quad (5)$$

For other resources of the M_j such as physical memory (RAM) and network bandwidth (BW):

$$\forall i \in r_j(t): \sum_{i \in r_j(t)} ram_i \leq RAM_j, \forall i \in r_j(t): \sum_{i \in r_j(t)} bw_i \leq BW_j \quad (6)$$

HPC applications have various sizes and require multiple cores and submit to system at dynamic arrival rate [10]. An HPC application can request some VMs.

3 Genetic Algorithm for Power-Aware Task scheduling

3.1. Data structures

CUDA framework only supports array data-structures. Therefore, arrays are an easy ways to transfer data from/to host memory to/from GPU. Each chromosome is a mapping of tasks to physical machines where each task requires a single VM. Fig. 1 presents a part of a sample chromosome with six tasks (each task is executed on a single VM), the task ID=0 is allocated to machine 5, the task ID=1 is allocated to machine 7, etc.

Task ID	0	1	2	3	4	5
Machine ID	5	7	8	4	5	9

Fig. 1. A part of chromosome

Finesses of chromosomes are evaluated and stored in an array similar to that in Fig. 2. Chromosome 0 has fitness of 1.892; chromosome 1 has fitness of 1.542, etc.

Chromosome	0	1	2	3	4	5
Fitness	1.892	1.542	1.457	1.358	1.355	1.289

Fig. 2. A part of a sample array of finesses of chromosomes

3.2. Implementing Genetic Algorithm on CUDA

We show the BKGPUGA's execution model that executes genetic operations on both CPU and GPU as shown in the Fig. 3 below.

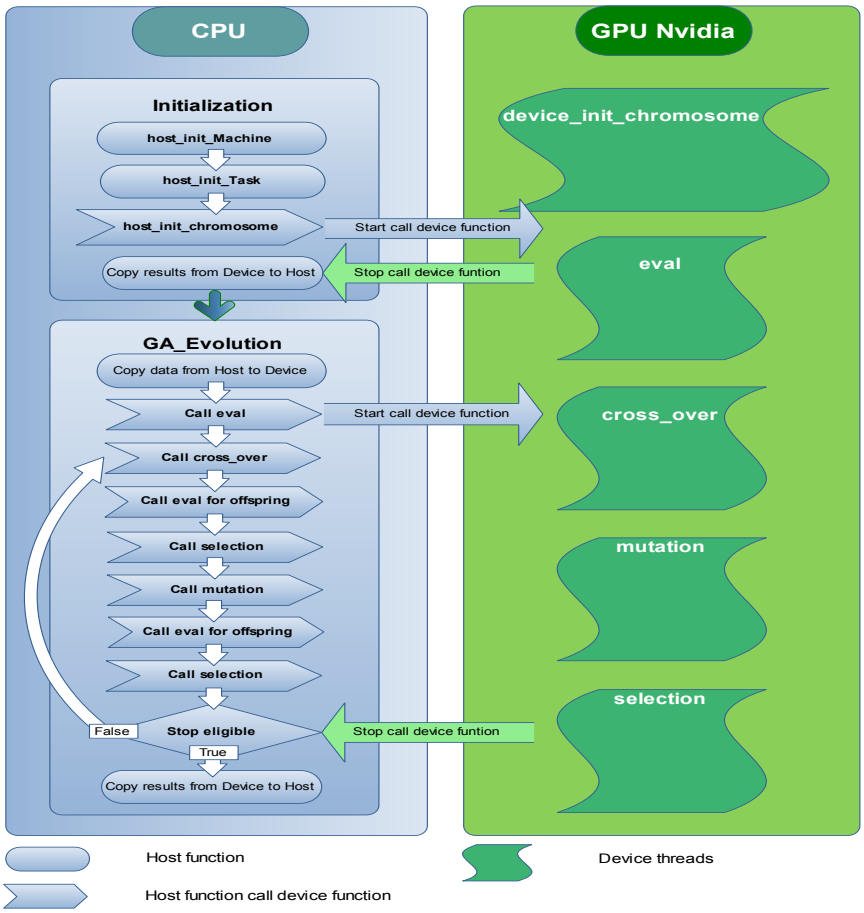

Fig. 3. Execution model of genetic operations on CPU and GPU.

Initialize population: Initial population using CUDA's cuRAND library.
GA_Evolution method: This is the main loop of the GA on CPU-GPU. The Algorithm 1 illustrates the GA_Evolution method.

Algorithm 1: GA_Evolution

```
Input: num_generations, Chromosomes[], Fitness[], TaskInfo[], MachineInfo[]
Output: The best chromosome with highest fitness

size_arr_chrom = A=sizeof(int)*length(chromosome)*pop_size ;
d_NST[], d_tem[]; /* Array of parent and offspring chromosomes on GPU */
size_arr_fitness = B = sizeof(float) * pop_size
d_fitness[], d_tem_fitness[];/* Parent fitness and Offspring  fitness on GPU */
cudaMalloc ( d_NST, d_tem , A);
cudaMalloc ( d_fitness, d_tem_fitness, pop_size);
cudaMemcpy ( d_NST, host_NST, A, HostToDevice) ;
cudaMemcpy ( d_tem, host_NST, A, HostToDevice) ; /* Cloning */
Load tasks and machines information to GPU;
eval_fitness <<< n_chromosomes >>>(d_NST, d_fitness) ;
for c = 1 to num_generations do
    crossover<<<n_chromosomes>>>(d_NST, d_tem, cu_seeds);
    eval_fitness<<<n_chromosomes>>>(d_tem,d_tem_fitness);
    selection<<<n_chromosomes>>>(d_NST,d_tem,d_fitness, d_tem_fitness);
    mutation<<< n_chromosomes x length(chromosome)>>>( d_tem, pop_size, cu_seeds);
    eval_fitness<<<n_chromosomes>>>(d_tem,d_tem_fitness);
    selection<<<num_chromosomes>>>(d_NST,d_tem,d_fitness,d_tem_fitness, cu_seeds);
cudaMemcpy( host_NST, d_NST, DeviceToHost);
cudaMemcpy( host_fitness, d_fitness, DeviceToHost);
cudafree(d_NST, d_tem, d_fitness, d_tem_fitness);
```

Fitness Evaluation

The Fig. 5 shows the flowchart of the fitness evaluation. The placement of each task/VM on a physical machine has to calculate the power consumption increase as the VM is allocated to a physical machine and reduce power consumption when the task/VM is finished its execution.

Selection method

The BKGPUGA does not use random selection method, the BKGPUGA's selection method is rearrangement of chromosomes according to the fitness from high to low, then it pick up the chromosomes have high fitness until reach the limit number of populations. The selection method is illustrated in Fig. 6. After selection or mutation, chromosomes in Parents and Offspring population will have different fitness, size of new population that included both Parents' and Offspring's is double size. Next, the chromosomes are rearranged according to the fitness value, the selection method simply retains high fitness of chromosome in the region, and the population is named after F1, whose magnitude is equal to the original population. To prepare for the next step of the algorithm GA (selection or mutation), F1 will be given a copy of Clones. The next calculation is done on Clones, Clones turn into F1's offspring. After each operation, the arrangement and selection is repeatedly.

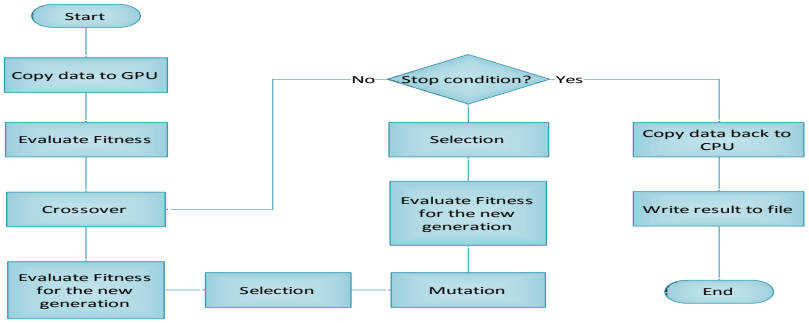

Fig. 4. GA_Evolution method

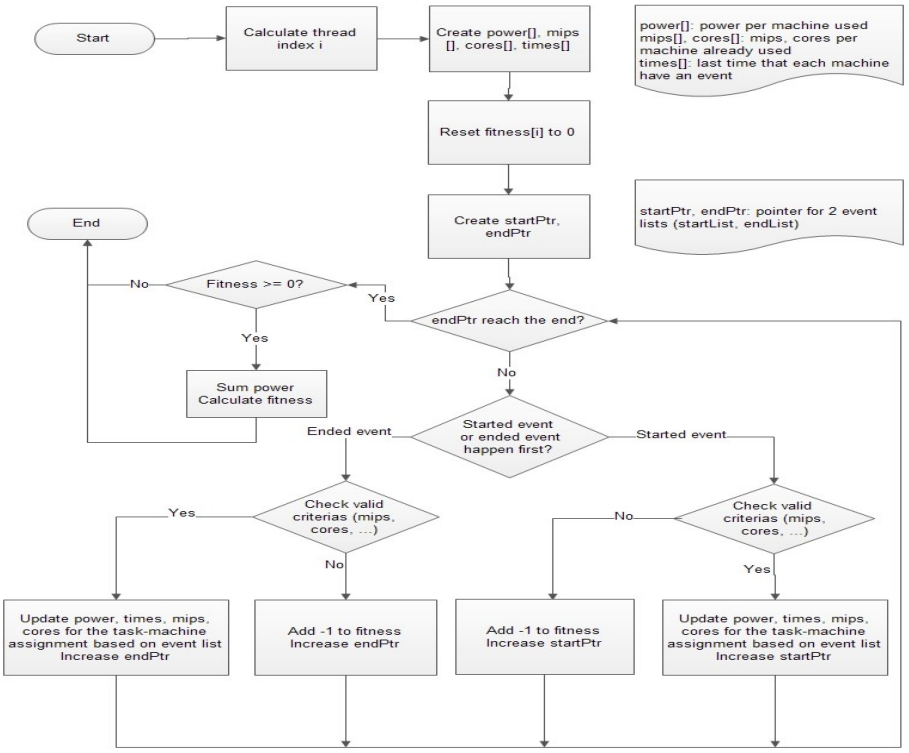

Fig. 5. Flowchart of evaluation fitness of a chromosome on GPU

In order to simplify and speedup the sorting operation, the program has used the CUDA Thrust Library provided by NVIDIA. The selection method keeps the better individuals. This is not only improves the speed of evolution, but also increases the speedup of overall program because of the parallel steps.

Mutation method

Each thread will execute decisions on each cell mutagenic or not based on a given probability. If the decision is *yes*, then the cell will be changed randomly to different values. Fig. 7 shows

mutation with 12.5% probability. Call n is the total number of cell populations, p is the probability change of each cell, q=(1 - p). The probability to have k cells modified Bernoulli calculated by the formula: $P(k) = C_n^k \times p^k \times q^{n-k}$

The cells that are likely to be modified: $(n \times p - q)$ or $(n \times p - q + 1)$

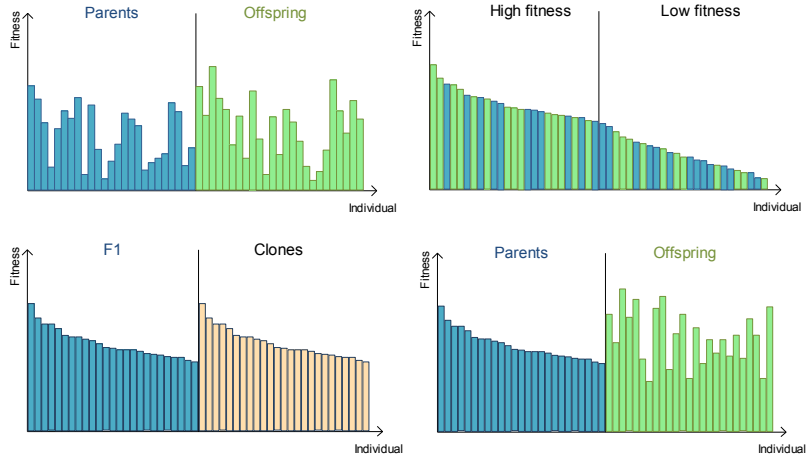

Fig. 6. Fitness of Parents and Offspring populations in Selection method

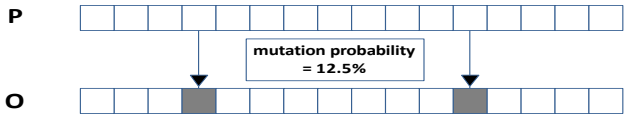

Fig. 7. Mutation process with 12.5% probability

Crossover method

Crossover is the process of choosing two random chromosomes to form two new ones. To ensure that after crossover it allowed sufficient number of individuals to form Offspring population, the probability of it is 100%, which mean all will be crossover.

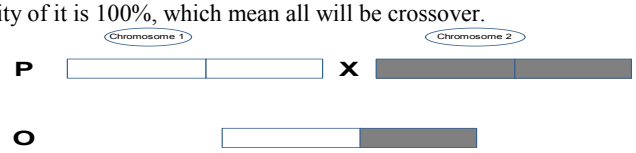

Fig. 8. Selection process between two chromosomes

Crossover process is using one-point crossover and the cross point is randomly chosen. Fig. 8 shows an example of section process between two chromosomes. The result is two new chromosomes. This implementation is simple, ease of illustrating. It creates the children chromosomes randomly but it does not guarantee the quality of these chromosomes. To improve the quality of the result, we can choose the parents chromosomes with some criteria but this makes the algorithm becoming more complex. Thus, the selection with sorting will overcome this drawback.

4 Experimental results

Both serial (SGA) and GPU (BKGPUGA) programs were tested on a machine with one Intel Xeon E5-2630 (6 cores, 2.3 GHz), 24GB of memory, and one Tesla M2090 (512 cores, 6GB memory).

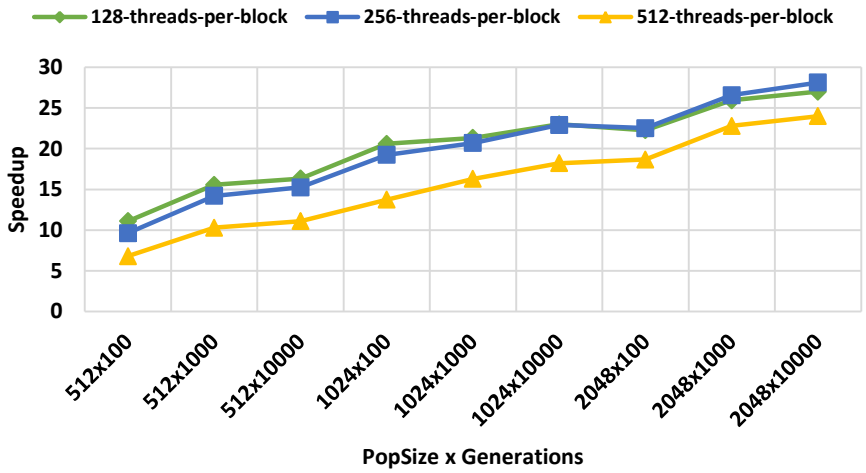

Fig. 9. Speedup of BKGPUGA that executes on NVIDIA Tesla M2090 and computational time SGA that executes on CPU. The X-axis is the size of population and the number of generations. The green/blue/yellow line is the speedup of BKGPUGA with 128/256/512 CUDA threads-per-block.

We generated an instance of the PATS with the number of physical machines and the number of tasks is 500 x 500. On each experiments, mutation probability is 0.005, the number of chromosomes (*popsize* - size of population) is {512, 1024, 2048}, the number of generations is {100, 1000, 10000}, the number of CUDA threads-per-block is {128, 256, 512}. Table 1 shows experimental results of the computation time of the serial GA (SGA) and the computational time of the BKGPUGA. Fig. 9 shows the speedup chart of the BKGPUGA program on configurations of 128, 256 and 512 CUDA threads-per-block (green, blue and yellow lines respectively). The maximum speedup of BKGPUGA is 28.14 when using 256 CUDA threads-per-block to run the GPU GA with 2048 chromosomes and 10,000 generations. The number of generations is the main factor that affects the execution time, when number of generations increases from 100 to 1000 and 10,000 generations the BKGPUGA's average execution time increases approximately ×7.66 and ×71.67 and the SGA's average execution time increases approximately ×9.16 and ×91.39 respectively. The fitness comparison between BKGPUGA and CPU version shows that the difference is relative small (10^{-11}). The fitness values on 1,000 and 10,000 generations are almost equal; that they figure out if it nearly reaches the best solution, the increase of generations makes the fitness is better but not much and a tradeoff is the increased execution time on the BKGPUGA.

Table 1. Experimental result of SGA and BKGPUGA: Problem size is 500x500

Pop. size	Genera-tions	#Threads-per-block	SGA Comp. time (sec.)	BKGPUGA Comp. time (sec.)	Speedup
512	100	128	22.501	2.028	11.10
512	100	256	22.501	2.340	9.61
512	100	512	22.501	3.316	6.79
512	1000	128	255.123	16.390	15.57
512	1000	256	255.123	17.967	14.20
512	1000	512	255.123	24.764	10.30
512	10000	128	2,564.250	157.058	16.33
512	10000	256	2,564.250	168.187	15.25
512	10000	512	2,564.250	231.165	11.09
1024	100	128	60.077	2.918	20.59
1024	100	256	60.077	3.121	19.25
1024	100	512	60.077	4.373	13.74
1024	1000	128	516.906	24.228	21.34
1024	1000	256	516.906	24.971	20.70
1024	1000	512	516.906	31.743	16.28
1024	10000	128	5,351.200	232.682	23.00
1024	10000	256	5,351.200	233.576	22.91
1024	10000	512	5,351.200	293.717	18.22
2048	100	128	114.827	5.156	22.27
2048	100	256	114.827	5.098	22.52
2048	100	512	114.827	6.152	18.66
2048	1000	128	1,035.470	39.912	25.94
2048	1000	256	1,035.470	38.958	26.58
2048	1000	512	1,035.470	45.428	22.79
2048	10000	128	10,124.880	374.850	27.01
2048	10000	256	10,124.880	359.827	28.14
2048	10000	512	10,124.880	421.728	24.01

5 Conclusions and Future Work

Compared to previous studies, this paper presents a parallel GA using GPU computation to solve the power-aware task scheduling (PATS) problem in HPC Cloud. Both BKGPUGA and the corresponding SGA programs are implemented carefully for performance comparison.

Experimental results show the BKGPUGA (CUDA program) executed on NVIDIA Tesla M2090 obtains significant speedup than SGA (serial GA) executed on Intel Xeon E5-2630. The execution time of BKGPUGA depends on the number of generations, size of the task scheduling problems (number of tasks/VMs, number of physical machines). To maximize speedup, when the number of generations is less than or equal to 1,000 we prefer to use 128 CUDA threads per block, and when the number of generations is greater than or equal to 10,000 we prefer to use 256 CUDA threads per block. The limitation on the number of tasks and number of physical machines is the size of local memory on each CUDA thread in internal GPU card.

In the future work, we will concern on some real constraints (as in [11]) on the PATS and we will investigate on improving quality of chromosomes (solutions) by applying EPOBF heuristic in [12] and Memetic methodology in each genetic operation.

Acknowledgments. This research is funded by Vietnam National University Ho Chi Minh (VNU-HCM) under grant number B2012-20-03TĐ.

References

1. Li, K.: Performance Analysis of Power-Aware Task Scheduling Algorithms on Multiprocessor Computers with Dynamic Voltage and Speed. IEEE Trans. Parallel Distrib. Syst. 19, 1484–1497 (2008).
2. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. J. Parallel Distrib. Comput. 61, 810–837 (2001).
3. Kołodziej, J., Khan, S., Zomaya, A.: A Taxonomy of Evolutionary Inspired Solutions for Energy Management in Green Computing: Problems and Resolution Methods. Adv. Intell. Model. Simul. 422, 215–233 (2012).
4. Tesla Kepler GPU Accelerators. (2013).
5. Chen, S., Davis, S., Jiang, H., Novobilski, A.: CUDA-based genetic algorithm on traveling salesman problem. In: Lee, R. (ed.) Computer and Information Science. pp. 241–252. Springer-Verlag Berlin Heidelberg (2011).
6. Luong, T. Van, Melab, N., Talbi, E.-G.: GPU-based island model for evolutionary algorithms. Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10. p. 1089. ACM Press, New York, New York, USA (2010).
7. Arenas, M.G., Mora, A.M., Romero, G., Castillo, P.A.: GPU computation in bioinspired algorithms: a review. In: Cabestany, J., Rojas, I., and Joya, G. (eds.) Advances in Computational Intelligence. pp. 433–440. Springer Berlin Heidelberg (2011).
8. Zhang, S., He, Z.: Implementation of Parallel Genetic Algorithm Based on 2 Classification of Parallel Genetic Algorithms. 4th Int. Symp. ISICA 2009 Huangshi, China. 5821, 24–30 (2009).
9. Fan, X., Weber, W.-D., Barroso, L.A.: Power provisioning for a warehouse-sized computer. ACM SIGARCH Comput. Archit. News. 35, 13 (2007).
10. Garg, S.K., Yeo, C.S., Anandasivam, A., Buyya, R.: Energy-Efficient Scheduling of HPC Applications in Cloud Computing Environments. arXiv Prepr. arXiv0909.1146. (2009).
11. Quang-Hung, N., Nien, P.D., Nam, N.H., Tuong, N.H., Thoai, N.: A Genetic Algorithm for Power-Aware Virtual Machine Allocation in Private Cloud. ICT-EurAsia'13. LNCS 7804, 183–191 (2013).
12. Quang-Hung, N., Thoai, N., Son, N.T.: EPOBF: Energy Efficient Allocation of Virtual Machines in High Performance Computing Cloud. J. Sci. Technol. Vietnamese Acad. Sci. Technol. 51, No. 4B, 173–182 (2013).