Comparison of Parallel CUDA and OpenMP Implementations of Particle Swarm Optimization

*1O. Tolga Altinoz and 2A. Egemen Yılmaz
*1Department of Electrical and Electronics Engineering, TED University, Kolej, Ankara, Turkey
2Department of Electrical and Electronics Engineering, Ankara University, Golbasi, Ankara, Turkey

Abstract:

Since the physical constraints on micro computing devices have forced the researchers to design next generation chips, the significance of the parallelization and distributed computing grow in importance. In this study, a sequential implementation of the Particle Swarm Optimization algorithm is converted into a concurrent version, which is executed on the cores of both CPU and GPU. For this reason, CUDA and OpenMP libraries are operated on the parallel algorithm to make a concurrent execution on CPU and GPU, respectively. The aim of this study is to compare CPU and GPU implementation of the PSO algorithm as regards the average execution time of independent Monte Carlo runs and computation architecture. For this purpose, nine benchmark functions are selected as test problems, and the parallel algorithm is executed on different processing units. The results show that parallel performance of the algorithm on different architectures outperforms to some application oriented computation units.

Key words: Particle swarm optimization, parallel programming, CUDA, OpenMP

1. Introduction

Many engineering problems are based on finding the best possible solution for any specific configuration/system. For this purpose, beginning with the mathematicians who solved optimization problems that are related to geometric studies, new algorithms have been developed in order to find better solutions. Since the real-world engineering problems are very complicated when compared to geometric studies, the classical optimization algorithms are not able to solve these problems demanding complex mathematical computations. Therefore, some methods which are greatly developed by observation of the nature such as relations between matters and living beings are introduced, and they are known as heuristic approaches. Genetic Algorithm [1], Ant Colony Optimization [2], Differential Evolution [3], and Particle Swarm Optimization (PSO) [4] can be mentioned as examples of the heuristic approaches. Even though these algorithms yield better results for many engineering problems, the execution time of sequential codes or implementations restrict the research in this are as regards the time limitations; so that a mid-level problem can be solved in a couple of days with a sequential algorithm, and consequently the problem becomes too costly to be solved. Therefore, especially in the last decade, engineers have applied parallelization techniques and libraries in order to decrease the cost of a study/project and the execution time of the parallelized algorithms. Hence, in this study, the sequential PSO algorithm is parallelized in two different manners (i.e. via different techniques aimed for

*Corresponding author: Address: Faculty of Engineering, Department of Electrical and Electronics Engineering, TED University, Kolej, Ankara, TURKEY. E-mail: tolga.altinoz@tedu.edu.tr

deployment on different architectures), and these two parallel PSO (PPSO) algorithms are executed on both CPU and GPU.

Currently, two major parallelization techniques can be mentioned: These are
a) Distributed computing [5] via the Message passing Interface (MPI), in which the algorithm is divided/duplicated into sub-codes, and then these codes transferred to the distributed computational units. Each unit executes relatively tiny piece of the code, then they transfer the response to the main computer. There is not any restriction related to the distance between computation units, so that any unit could be installed elsewhere (e.g. even in another country and/or continent),
b) Multicore processing [6]; for which parallel processing units are at a single device such as personal computer, workstation or server, then the code is executed on different cores of on-board devices like CPU and/or GPU. In this paper, the multicore processing of PPSO algorithm is investigated with respect to the parallel processing unit.

There are two main architectures (CPU and GPU) in a personal computer device. The tasks assigned for these two units differ each other. The CPU composes of only a few numbers of very fast cores, which have the ability to execute relatively complex operations. On the other hand GPU consists of many numbers of slow cores that are able to execute fundamental operations. Previously, the authors have demonstrated that the PPSO code executed on different GPU architectures is approximately 20% faster than sequential code run on CPU [6]. However, the performance of PPSO executed on the cores of different CPU architecture and comparison with GPU is a question that should be investigated. Therefore, in this study, the PPSO algorithm is implemented on different CPU and GPU architectures. The average execution time of ten independent Monte Carlo runs (executions) is considered as a performance indicator.

This paper is composed of 3 sections in addition to the conclusion. In Section 2, parallelization of the PSO algorithm with CUDA and OpenMP is discussed. In Section 3, the performance of the PPSO algorithms on different architectures is investigated with respect to complexity of the computation units.

2. Parallelization of Particle Swarm Optimization

Since it is a population based algorithm, Particle Swarm Optimization (PSO) is a good candidate for parallelization. The behavior of the algorithm greatly depends on the interactions between particles, where the members of the population are called particles (N). Each particle has velocity (v_i, $i=1,2,\ldots,N$) and position (x_i, $i=1,2,\ldots,N$). At the end of each iteration (k), the particles change their position based on their current velocity. The well-known physical position update rule, which is applied on this algorithm, is given in (1):

$$x_i[k+1] = x_i[k] + v_i[k] \times \Delta t \tag{1}$$

Equation (1) implies that the new position is the sum of the previous position with multiplication of velocity and time interval, which is assumed to be the time interval for each consecutive iteration (|(k+1) – k|) and usually considered to be unity (Δt=1). Hence, it can be extracted from (1) that, the movement of any particle in the swarm greatly depends on its velocity. The velocity update formulation is presented in (2):

$$v_i[k] = v_i[k] + c_1 \times R \times \left(p_i[k] - x_i[k]\right) + c_2 \times R \times \left(p_g[k] - x_i[k]\right) \tag{2}$$

where c_1 and c_2 are the control parameters of the algorithm (c_1=c_2=1.498), p_i is the personal best position of i'th particle, and p_g is the global best position among the population [4].

The PSO algorithm can be summarized as follows:
a) the position and velocity values of each particle are assigned randomly,
b) At the beginning of iteration, cost function is evaluated ($f(x_i)$) for each particle's position (x_i),
c) Personal (the best position of each individual between the iteration 1 and k) and global best (the best position among the population at the iteration k) positions are determined based on their cost values,
d) The position of each particles is altered by using (1) and (2),
e) if the maximum number of iteration is not equal to current iteration k, then step b is revisited; else the algorithm is terminated.

The structure of PSO algorithm is based on the repeated action so that same mathematical expression is executed for all particles, which makes the algorithm quite suitable for parallelization. Figure 1 presents only the general idea beneath the parallelization of particle swarm optimization, since the parallelization greatly depends on the hardware/architecture. For example, the core indices and the number of cores depend on the architecture. Ideally, it is desired to execute each particle's operations in a separate core. But in most cases, the number of cores is not sufficient to achieve this since some cores are already occupied by the operating system. Anyway, Figure 1 presents a good description for explaining PPSO. PPSO can be briefly summarized as follows:
a) The position and velocity values of each particle are assigned randomly at different cores, if there are sufficient number of cores, then these parts is executed only at a single instruction run instead of N instruction runs.
b) At the beginning of the iteration, the cost function is evaluated ($f(x_i)$) for each position (x_i) at each core,
c) Personal (the best position of each individual between the iteration 1 and k) and global best (the best position among the population at the iteration k) positions are determined according to the cost values. This step includes calculation of taking the minimum of a vector, which is implemented by association of many cores. However, due to some subroutines (comparisons and statements), all cores might not be uses efficiently.
d) The position of each particle is altered by using (1) and (2) at each core
e) if the maximum number of iteration is not equal to current iteration k, then step b is revisited; else the algorithm is terminated.

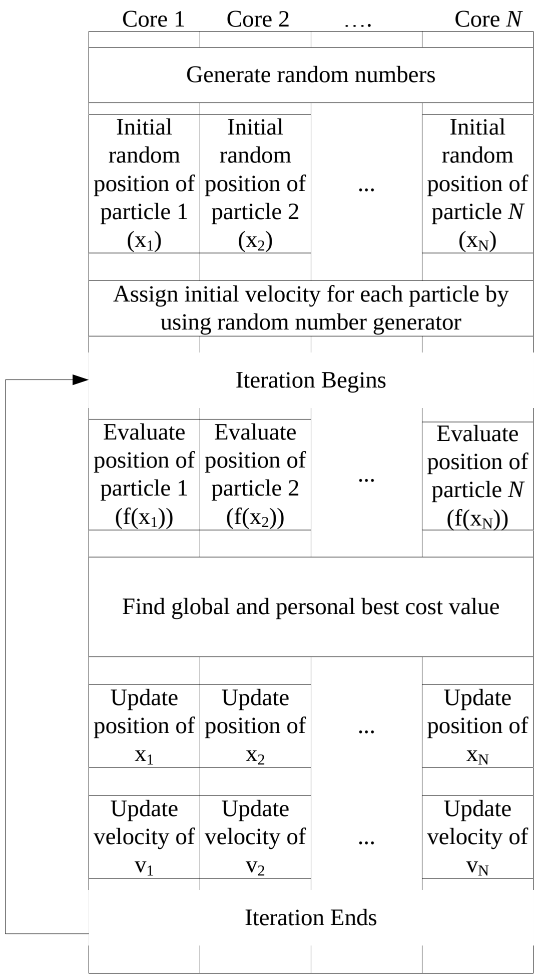
Core 1
Core 2
….
Core *N*
Generate random numbers
Initial random position of particle 1 (x_1)
Initial random position of particle 2 (x_2)
...
Initial random position of particle *N* (x_N)
Assign initial velocity for each particle by using random number generator
Iteration Begins
Evaluate position of particle 1 ($f(x_1)$)
Evaluate position of particle 2 ($f(x_2)$)
...
Evaluate position of particle *N* ($f(x_N)$)
Find global and personal best cost value
Update position of x_1
Update position of x_2
...
Update position of x_N
Update velocity of v_1
Update velocity of v_2
...
Update velocity of v_N
Iteration Ends

Figure 1. A sample demonstration for parallel particle swarm optimization

2.1. Parallelization on GPU with CUDA

In the last decade, graphical boards are preferred for complex computations since they can have more than a thousand cores. Hence, the graphical processing units (GPUs) have become the dominant and leading parallel computation units in the market [7]. Even though numerous cores allow the problem in hand to be solved in a relatively short time, programming of GPU oriented codes becomes harder and more complex. However, with the introduction of the CUDA package, which is a set of libraries composed of well-written codes and algorithms, programming becomes much easier and straightforward for researchers. In this study, the general framework of PPSO algorithm, which is demonstrated on Figure 1, is implemented on the GPU board with the aid of CUDA libraries [6].

2.2. Parallelization on CPU with OpenMP

In a general manner, the codes which are written in C/C++ environment are executed on CPU on a single core. Since the numbers of the cores in CPU have increased, programmers desire to use these cores with simple modifications on sequential codes. Therefore, the OpenMP package was introduced for this purpose [8]. This package contains compiler directives, run time routines and environment variables. Since all data is recorded on the globally shared memory on the CPU, OpenMP programming is much simpler compared to CUDAAs part of this study, PPSO is also implemented on different CPU architectures.

3. Results

The aim of this study is demonstrate the performance of PPSO algorithm on different hardware units. Therefore, some benchmark problems are selected. Table 1 gives these nine benchmark problems, which contains unimodal and multimodal functions. The algorithm has the following parameter settings: The swarm size is 100, number of iterations is 1000, the problem dimension (*n*) is 10, and the number of independent Monte Carlo runs is 10.

Table 1. Benchmark functions

Function	**Search Space**	**Optimum Value**
$f_1 = \sum_{i=1}^{n} x_i^2$	$[-5.12,5.12]^n$	0
$f_2 = \sum_{i=1}^{n} \left(i.x_i^2\right)$	$[-5.12,5.12]^n$	0
$f_3 = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j\right)^2$	$[-5.12,5.12]^n$	0
$f_4 = \sum_{i=1}^{n-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + \left(x_i - 1\right)^2\right]$	$[-2.048,2.048]^n$	0
$f_5 = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i)\right)$	$[-5.12,5.12]^n$	0
$f_6 = 20 + e - 20\exp\left(-0.2\sqrt{\frac{1}{2}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right)$	$[-15,30]^n$	0
$f_7 = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right)$	$[-600,600]^n$	0
$f_8 = \sum_{i=1}^{n} \lvert x_i \rvert^{i+1}$	$[-1,1]^n$	0
$f_9 = -\sum_{i=1}^{n} \sin(x_i)\left[\sin\left(\frac{ix_i^2}{\pi}\right)\right]^{20}$	$[0,\pi]^n$	-4.687

Instead of recording the execution time for each independent run, in this study, the total execution time is recorded, and then the average time is calculated. Hence, the average execution time is taken as a metric of this study. The performance comparison among different hardware (CPU: Intel Core i5-2450M, AMD Athlon X2 270, Intel Core i5-3470s; GPU: GTX550Ti, GT610, GT 520MX, Quadro K5000, Tesla K20) is practiced on two different setups. In the first setup, sequential PSO algorithm is implemented on single core CPU, and the results are compared with the average execution time of PPSO on GPU. Table 2 presents this comparison.

Table 2. Average execution times of 10 independent Monte Carlo runs for sequential and parallel implementations of the PSO algorithm (ms)

Function	C/C++ - Sequential			CUDA - Parallel				
	Intel Core i5-2450M	AMD Athlon X2 270	Intel Core i5-3470S	GTX 550Ti	GT 610	GT 520MX	Quadro K5000	Tesla K20
f_1	2824.5	672	1038.15	236.63	361.39	554.91	456.51	170.89
f_2	2777.35	696	1025.5	244.09	384.72	573.66	450.4	170.32
f_3	3162.89	919	1284.3	258.72	411.652	608.13	497.69	192.46
f_4	1933.5	590	908.04	354.03	433	355.6	496.86	191.16
f_5	2195.1	691.5	1261	295.44	488.28	360.58	591.53	200.76
f_6	1924.15	729.5	1034.55	306.52	509.25	353.78	590.59	203.6
f_7	1603.44	758.5	574.95	312.37	516.3	331.66	591.02	203.55
f_8	3195.89	1001.5	1349.65	357.32	624.33	425.36	763.43	263.78
f_9	3928.35	1213.5	1569.8	466	884.16	633.43	1331.83	419.43

Table 2 clearly presents that the GPU implementation outperforms to the sequential code. For a professional computation device NVIDIA Tesla K20, execution on the GPU is approximately 100 times faster than the mobile CPU processor. The results also demonstrate that even the performance of the slowest GPU device is better than all CPU architecture discussed in this paper. Also from Table 2, it is clear that the application oriented GPU device NVIDIA Quadro K5000 demonstrated almost the same performance with the low cost GPU architecture GT 610. The reason behind this unexpected result that the Quadro device is a highly application oriented device, so that that device contains many fast image processing functions. Therefore, it can be claimed that this device is not suitable for optimization algorithm parallelization.

In the second setup, PPSO is implemented on two different parallel devices, which are CPU and GPU. The average execution times for these implementations are presented in Table 3. It is clear from the table that the fastest device for parallelization is the NVIDIA Tesla K20 GPU board, which is approximately 5 times faster than other devices. The results demonstrate that the mobile devices perform the slowest execution times. Also, from the results, a good CPU configuration performs almost the same as mid-level GPU devices. Even if the PPSO implementation on CPU produces almost same performance as same GPU boards, CPUs are sometimes unreliable so that the operating system assigns some tasks, which interrupt the parallel operations.

The results presented in Tables 2 and 3 indicate that for some algorithms, the parallelization on CPU may be better than or comparable to GPU implementations. The performance of PPSO is outperforms only at a high-level computing device NVIDIA Tesla K20.

Table 3. Average execution times of 10 independent Monte Carlo runs for CUDA parallel and OpenMP parallel implementations of the PSO algorithm (ms)

Function	OpenMP - Parallel			CUDA - Parallel				
	Intel Core i5-2450M	AMD Athlon X2 270	Intel Core i5-3470S	GTX 550Ti	GT 610	GT 520MX	Quadro K5000	Tesla K20
f_1	1437.09	534.5	324	236.63	361.39	554.91	456.51	170.89
f_2	1482.34	509.5	297.04	244.09	384.72	573.66	450.4	170.32
f_3	1640.3	635.5	347.95	258.72	411.652	608.13	497.69	192.46
f_4	1195.69	460	220.35	354.03	433	355.6	496.86	191.16
f_5	1343.19	554	258.54	295.44	488.28	360.58	591.53	200.76
f_6	1225.09	572.5	222.8	306.52	509.25	353.78	590.59	203.6
f_7	1293.39	609	233.14	312.37	516.3	331.66	591.02	203.55
f_8	2038.4	781	327.85	357.32	624.33	425.36	763.43	263.78
f_9	1893.09	842	439.85	466	884.16	633.43	1331.83	419.43

Conclusions

In this study, PPSO is implemented on eight different hardware devices (CPU: Intel Core i5-2450M, AMD Athlon X2 270, Intel Core i5-3470s; GPU: GTX550Ti, GT610, GT 520MX, Quadro K5000, Tesla K20), and three different software models (sequential, OpenMP and CUDA). Nine benchmark functions are selected as test problems, and average execution time is taken as a performance criterion. The results show that:
a) parallel implementations are faster than sequential codes as expected,
b) application oriented hardware might demonstrate unexpectedly the worst performance;
c) in general, GPU implementations outperform to CPU implementations, and
d) instead of mid-level GPU devices, the programmers can select CPUs as target devices, which prove to be more cost-effective.

As a future study, codes on high-level CPU architectures will be implemented, and hybrid implementations (using CPU and GPU cores) will be performed.

Acknowledgements

This study is made possible by a joint grant from TUBITAK (with Grant Nr. 112E168) and ANCS-UEFISCDI (with Grant Nr. 605/01.01.2013). The authors would like to express their gratitude to these institutions for their support.

References

[1] Holland J. Adaptation in nature and artificial systems An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press; 1992.
[2] Dorigo M. Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italy, 1992.
[3] Storn R, Price K. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization 1997; 11: 341–359.
[4] Kennedy J, Eberhart R. Particle Swarm Optimization. Proceedings of IEEE International Conference on Neural Networks IV 1995;1942–1948.
[5] Arora S, Barak B. Computational Complexity – A Modern Approach, Cambridge; 2009.
[6] Altinoz OT, Yilmaz AE, Ciuprina G. Comparison of particle swarm optimization on various GPUs designed for add-on graphic boards. International Symposium on Computing in Science and Engineering; 2013, 297-301.
[7] NVidia Corporation CUDA dynamic parallelism programming, NVidia, 2012.
[8] OpenMP, Architecture Review Board (http://www.openmp.org), version 3.0 has been released in 2008.