International Journal of Research and Engineering
ISSN: 2348-7860 (O) | 2348-7852 (P) | Vol. 04 No. 06 | June 2017 | PP. 177-183

Particle Swarm Optimization with CUDA

Somayeh Taherian Dehkordi,Amid Khatibi Bardsiri
Department of Computer Engineering, Kerman Branch, Islamic Azad University
Kerman, Iran

Abstract—**In recent years, particles' optimization algorithm has highly been used as an effective method in solving complex and difficult optimization problems. Since particles algorithm is based on recurring population and it can be very inefficient in terms of the time required for implementation and speed to solve optimization problems with large-scale including ones which need a very large population to search in problem solution space. The main reason for this issue is that this algorithm optimization process requires a large number of function evaluations which are usually run serially. This article aims to implement particles' optimization algorithm in parallel on graphics processing unit and to improve running efficiency and speed. The implementation results on the graphics processor show that the performance of this algorithm has greatly increased as to its implementation in parallel and with change in kernel implementation. In fact, in this study, implementation and velocity evaluation of particles algorithm implementation in parallel and based on CUDA framework has been investigated and compared. Then, there have been efforts to improve acceleration in this method in part and a new method will be proposed in CUDA framework to improve acceleration in particles algorithm and graphic processor setting.**

Keywords—**Parallel processing; Evolutionary algorithms; graphics processing unit(GPU); particle swarm optimization (PSO)**

I. Introduction

Evolutionary algorithms are considered as one of the most effective and efficient methods to solve optimization problems. The working method of evolutionary algorithms is modeled and defined based on the evolution and natural selection principles. In these algorithms, interactions between creatures and biological activities related to evolution are used to create computational algorithms which are able to calculate the optimum of problems. The behavior of biomolecules and transfer of inheritance from one generation to the next generation in the genetic algorithm[1], the mass and collective behavior of fish in the artificial fish algorithm[2], the escape from predators and the rush to food sources in the Krill algorithm[3], cooperation of spiders in social processes for [illegible] behavior of a pack of wolves and their leadership hierarchy to hunt animals in the gray wolf algorithm[5], migration patterns of animals from an area of low water and grass to an area rich in food in the animal migration algorithm[6], the behavior and illumination of fireflies to attract the opposite sex in the firefly algorithm[7], the pollination of flowers and plants for survival in the flower pollination algorithm[8], the proliferation and competition of weeds for survival and their natural selection to rush into bio-optimal areas in the invasive weed optimization (IWO) algorithm[9], and hundreds of other cases are among the cases which have been used to create evolutionary algorithms, and solve optimization problems.

The particle swarm optimization (PSO) algorithm [10] is an evolutionary algorithm with a swarm intelligence approach, where each member of the group can move towards the optimum of problems with the aid of other group members. In these algorithms, a single component, alone, is less able to solve the problem. But interacting with other members of the group, especially with more competent members of the group, who are responsible for leading the group, they can converge near the solutions of the problem. The particle swarm optimization (PSO) algorithm was presented and modeled by two social behavior psychologists through studying the group behavior of birds for finding food. In this algorithm, each member of the population has two components: speed and position, which are updated in each iteration by the best members and the movement history of the current member. When facing with large populations or large-scale problems, evolutionary algorithms such as: the PSO algorithm, experience difficulties and need a lot of time to converge on the optimal solution. Given that each member of the population, by itself, can calculate its position with the aid of the best global member and the best position, thus these algorithms can be parallelized and accelerated. One of the disadvantages of the PSO algorithm is the need for a large population to converge to more accurate solutions of optimization problems. In an optimization problem which is modeled using an objective function, if the complexity and dimensions of the problem increases, it is necessary to consider the number of the population members proportionally large enough, so that the entire search space of the objective function can be searched to find the optimum of the problem. In more complex and multi-dimensional objective functions, this process requires long execution time, which is inappropriate and undesirable in many applications. One of the methods to increase the speed of executing evolutionary algorithms, such as: the PSO algorithm, is the use of parallelization methods. And the parallel nature of the PSO algorithm has made the structure of this algorithm ideal for parallelization. There are various methods to parallelize and increase the speed of the PSO algorithm, among which we can refer to grid computing and parallelization thread packages [illegible]ch of these methods has its [illegible]e, in the grid computing technology, the implementation costs and high complexity makes this parallelization method unavailable to the public. Or in the thread parallelization method in the main processor, due to the presence of frequent switches in different processes in the main processor, this method does not have high performance.

Contrary to the two said methods, NVIDIA company which provides graphics processors, has recently offered a parallelization platform called CUDA, in order to parallelize complex computations in the multiple cores of a graphics processor. The CUDA platform can be considered as a library and an interface between the programmer and the graphics hardware, which greatly reduces the complexity of parallel computing, and puts various features and commands at the programmer's disposal to perform the parallelization operation [11].

In this paper, we try to use the parallel cores of the graphics card processor, the number of which is far greater than that of the main processor, in order to increase the speed of the PSO algorithm. The architecture and structure of the parallel cores of the graphics card processor allow each particle to be implemented independently of other particles, by one of the graphics card's cores. In this method, each particle searches part of the problem space in parallel with other particles.

In the proposed method, particles are placed inside the main memory in the form of a one-dimensional array, and this structure is placed inside the graphics processor straight away. The advantage of this method is a reduction in the number of sequential switches between the main processor and graphics processor for transferring the particles. In this method, each particle is placed inside a processing core of the graphics processor. Using this technique, the particles search the problem space, in parallel. In this method, the particles do not need to be placed in the main processor queue to be executed, which reduces the particles' search time. Compared with its serial version, this parallel structure increases the acceleration of the PSO algorithm. In the proposed method, to better search the problem space, we add a new mechanism to the PSO algorithm, in a way that if any particle is placed in the undesirable space, the particle can jump from the current location, and be transferred to a place with a higher chance of finding the optimal solution. This mechanism can help the accuracy of the proposed algorithm. In the proposed method, using a series of standard functions called "benchmark functions", the speed and acceleration of the proposed algorithm which is executable in the graphics processor, are compared with the standard PSO executable in the main processor. In this paper, first, we review the PSO algorithm and the architecture of graphics processors, then, we will introduce and evaluate the proposed method.

A. particle swarm optimization (PSO)

The PSO algorithm is an evolutionary algorithm of optimization based on the laws of probability and the collective behavior of organisms. The idea of the evolutionary algorithm was introduced by Dr. Russell Eberhard; a computer scientist, and Dr. James Kennedy; a social issues psychologist in 1995[12].

The PSO algorithm is modeled on the social and swarm behavior of birds or a group of fish searching for food. In this algorithm, the members of the population are modeled in the form of particles, and these particles try to move towards areas where more food is found, with the aid of their previous information and the best position of other particles.

When observing the behavior of birds which fly in groups, we can see the following two key behaviors [13]:

- Each particle has a velocity, whose value is determined based on the population's current speed and best position, and the best previous position of the desired particle.
- The position of each particle is updated based on the current position of the particle and its new velocity.

According to Figure 1, a particle can move differently in the search space of the problem, but it chooses a path which is more in line with the outcome of the best global position and its own best position [13]:

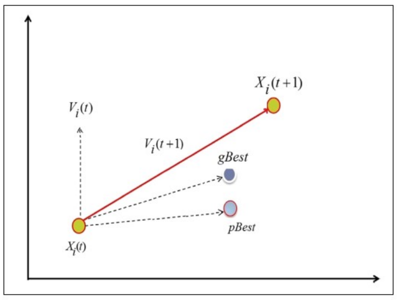


Fig. 1. The direction of motion of a particle

In this picture, V_i (t) and V_i (t+1) are the current and new velocity vectors of the i-th particle, respectively, $X_i(t)$ and $X_i(t+1)$ the current and new positions of the i-th particle, respectively, $pBest_i(t)$ the best position that the i-th particle has so far achieved, and gBest(t) the best position of all the particles in the population. As shown in the picture, in order to calculate Vector $X_i(t+1)$, it is required to obtain the vector sum of the two vectors: $X_i(t)$ and $V_i(t+1)$, according to Equation (1) [13]:

$$X_i(t+1) = X_i(t) + V_i(t+1) \quad (1)$$

The current velocity vector, the current position vector, the best global particle, and the best history of the current particle in the search space of the problem can be used to calculate the new velocity in according with Equation (2) [13]:

$$V_i(t+1)=\omega * V_i(t) + q_1 r_1(pBest_i(t)-X_i(t)) + q_2 r_2(gBest(t)-X_i(t)) (2)$$

Where; ω is the inertia coefficient, q_1 and q_2 the learning coefficients of the PSO algorithm, and r_1 and r_2 two random numbers in the interval [0,1]. It has been proved that if the speed of the particles is limited within a certain range, such as: $[-V_{MAX},V_{MAX}]$, the convergence of the PSO algorithm will be guaranteed.

It has been proved that if the values of ω, q_1, q_2 are chosen correctly, the speed of the particles will fall within a specified range, and there will be no need to limit the speed of the particles. Typically, if the values of learning coefficients in the PSO algorithm, are chosen from between 1.5 and 2, the algorithm will act properly. The convergence of the PSO algorithm is greatly dependent on the value of w, and it is better to dynamically choose this value in each iteration. One of the methods to choose the inertia coefficient is to reduce its value repeatedly in each iteration within a certain range such as (0.0, 2.9) until it is finally converged to 0.2.

In other words, choosing a larger value for the inertia coefficient in the early stages of the PSO algorithm, causes the search space of the problem to be scanned properly, and its smaller value in the final iterations causes the places around the optimal solutions to be searched properly.

II. Parallelism in Graphics Processor

A graphics processing unit (GPU) can be considered as a processor for graphics card to perform graphics operations and render images. The number of cores is far greater in a graphics processor than in the main processor, and the number of cores sometimes reaches up to hundreds and thousands. The architecture of graphics processors is in such a way that their architecture has been mainly used to design the processing and computing unit, and less focus has been put on the design of the main and cache memories. Because the processing operations performed in the graphics processor, are mainly simple but high-volume processing operations, in which the arithmetic logic unit (ALU) needs to be more developed than other units. Instead of storing data in the temporary or cache memory, a graphics processor tries to perform processing operations on the data. Thus, its circuits are more needed to be designed for computing, and are less needed to be used for data storage.

Whereas, in the main processor, most of the circuits have been prepared to design the hidden data storage unit, memory, control unit, etc. [14]. Figure (2) shows the difference between the hardware systems of a graphics processor and main processor. As can be seen in this figure, the number of cores used in the architecture of the graphics processor, is far greater than the number of those used in the main processor. And contrarily, the arithmetic logic unit (ALU) of the main processor has been more developed than that of the graphics processor, which is due to more complex calculations in the main processor [15]:

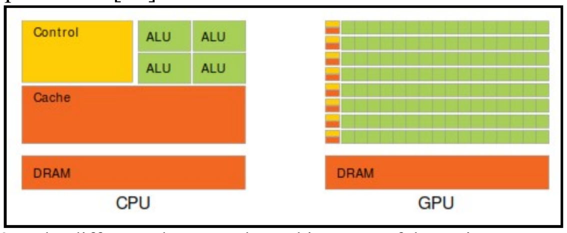


Fig. 2. The difference between the architectures of the main processor and graphics processor

By comparing the architectures of the main processor and graphics processor, the following points can be found out [16]:

- The number of cores is much greater in the graphics processor than in the main processor, which indicates that the architectures of the graphics processor and main processor are appropriate for computation on large and small data, respectively.
- The size of cores is smaller in the graphics processor than in the main processor, which indicates that the graphics processor is optimized for large but simple calculations, while the opposite is true in the main processor.
- The control unit is larger in the main processor architecture than in the graphics processor, which is due to the high complexity of the main processor architecture compared with that of the graphics processor, which needs its controlunit to be designed precisely, so that the main processor can be controlled and managed properly.
- A large part of the main processor's architecture has been allocated to the cache memory architecture. Whereas the cache memory does not play a very key role in the graphics processor, which is because each core acts individually, and process switching rarely occurs between different cores in the graphics processor. Thus the data are accessible by the processor. But in the main processor, due to the small number of cores, switching occurs frequently between processes, which requires the intermediate data to be stored in a fast memory such as the cache memory, so that in switching at the next stage, the process can have access to its own data.

A. *The parallel execution of programs in the graphics processor*

For execution on the graphics processor, each parallel program is divided into the following two main parts:

- The Host Code (executable in the main processor; CPU)
- The Machine Code (executable in the graphics processor; GPU)

The host code is in fact the serial and nonparallel part of a program code, which can only be executed in the main processor. Whereas the machine code forms the parallel part of the program code and is only executable in the graphics processor. The program code may consist of several serial and parallel parts, which are alternately repeated in the program code. Figure 3 shows the synchronous execution of a program in the CPU and GPU [17]:

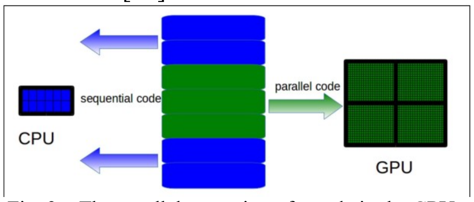


Fig. 3. The parallel execution of a code in the GPU

In this picture, the main code is divided into two parts: parallel (green) and serial (blue). And as shown in the picture, first, the serial part is executed in the CPU, and then, the parallel part is executed by both the CPU and GPU. And eventually, the serial code is executed in the CPU. The execution of serial and parallel parts can be done synchronously or asynchronously. In case of the latter, a synchronization process between the threads is necessary. For instance, the parallel processing of data in the GPU, can be done asynchronously by copying data from the CPU to the GPU, or synchronously during the transfer of data and information between the two processors.

B. *Comparing the processing power of GPUs and CPUs*

There are different criteria to measure the performance of CPUs and GPUs, among which the two main criteria: the computing power and data transfer bandwidth have been used here. The computing power is typically calculated based on the number of floating point operations that a processor can execute per unit time, and is usually measured in giga-operations per unit time, whose higher value is considered as a good sign for the performance of a processor. Figure (4) shows the number of floating point operations executed by two types of NVIDIA's graphics processors and two types of Intel's non-graphics processors in the first 11 months of 2013. The analysis of the graph shows that the increasing trend in the floating-point arithmetic is much bigger in the graphics processors than in main and non-graphics processors.

The results of investigations show that main processors' ability to perform floating-point operations has not changed a lot during a year, but the architecture of graphics processors, with its enormous growth during this period, could strongly increase their ability to perform floating-point arithmetic [17].

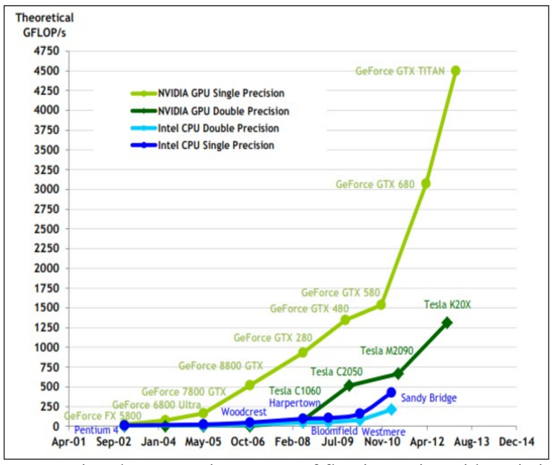


Fig. 4. Comparing the processing power of floating-point arithmetic in CPUs and GPUs

Performing floating-point arithmetic by itself cannot determine the performance of a processor. Because the computing power of a processor is also greatly dependent on an important factor; that is, memory bus bandwidth. Even if a GPU has a greater power in floating-point arithmetic, but the communication lines with the CPU are weak, it cannot enjoy high efficiency. Because the high bandwidth between the CPU memory and GPU memory, acts as a data transfer bridge, and the greater its width is, the more data it can transfer between the two processors during a certain period of time. Figure (5) shows a comparison between the memory bandwidths used in CPUs and GPUs and their cores between 2003 and 2013. This chart shows the bandwidths of two GPUs and a CPU during a period of 10 years. The analysis of the desired chart suggests that the sizes of computational bandwidths in GPUs, have had the greatest growth possible:

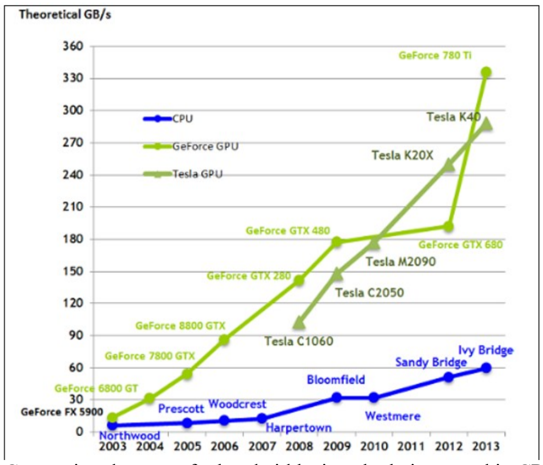


Fig. 5. Comparing data transfer bandwidths in calculations used in CPUs and GPUs

In 2007, NVIDIA Company which manufactures graphics processors and cards, introduced the first version of a software platform, called CUDA platform, for parallel programming with the aid of the company's GPUs. The CUDA platform is a software layer consisting of a series of library commands, which make it difficult for programmers to use the GPU.

C. *CUDA framework*

Through this platform, programmers can develop programs using common languages, especially Visual C++, for parallelization and execution of complex algorithms. In programming in the CUDA platform, a programmer needs to have specialized information about the hardware of GPUs, which makes programming in this platform a little difficult. CUDA is not a programming language, but it is rather considered as an interface between the programmer and GPU. The CUDA platform only works on NVIDIA's GPUs, and in this sense, the development and transfer of these programs are facing some difficulties. For programming in CUDA, it is necessary to install the packs of this platform and new updates of the GPU on the system, and add the path of CUDA compilers to the integrated development environment (IDE). A CUDA program is made up of two main parts; serial and parallel, which are regularly and alternately transferred between the CPU and GPU, and are executed [18]. The CUDA platform uses the software units: thread, block, and grid, to parallelize different data and algorithms. Each thread can be executed on a core in parallel, or a series of threads can be timed on a single core. The threads are kernel execution units, in a way that each thread executes a version of kernel. Threads are timed in larger units such as blocks, and a series of threads are defined in a block, and are able to communicate with each other with the aid of a shared memory. In case of substrate change, the threads contained in a block, store their data block in the local memory, so that they can perform their previous parallel operations again. A grid is an execution unit to call the kernels, and each kernel is executed by a grid which consists of a series of blocks. There is also a key concept in CUDA called the multiprocessing flow, which means a series of CUDA cores serving a parallel process, which will be further explained in what follows. Figure 6 shows the structures of a grid, block, and a thread in two-dimensional forms in Image (a), and several multiprocessing flows in CUDA in Image (b) [19]:

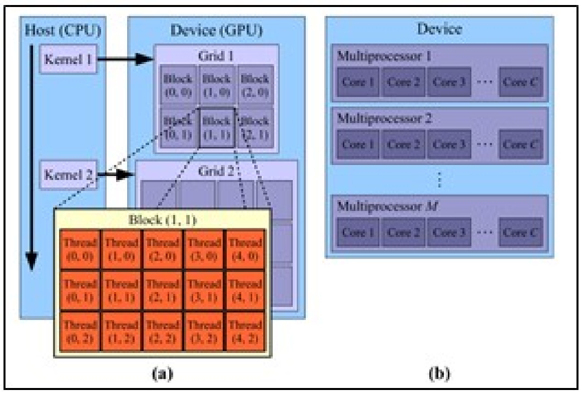


Fig. 6. The main components used in the CUDA platform

In a state where they are placed two-dimensionally inside the blocks, the structures of threads have two ID numbers; vertical and horizontal in the direction of each block. And these thread numbers are only valid in a single block and are considered to be local. With the aid of the intra-block horizontal and vertical numbers of each thread, and horizontal and vertical dimensions of each block, a unique number can be provided to the threads belonging to a block, in a way that no two threads in the flower of a grid have identical numbers. In the blocks contained in a grid in the two-dimensional mode, the situation is like that of two-dimensional threads. And each block has two intra-grid numbers; unique and local. And with the aid of the dimensions of a grid and the block number, the unique number of the blocks of a grid can be calculated.

D. *Maintaining the Integrity of the Specifications*

The template is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin in this template measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

III. THE PROPOSED METHOD TO INCREASE THE SPEED THROUGH PARALLELIZING THE PSO ALGORITHM

In order to find the optimum of problems, evolutionary algorithms, such as the PSO algorithm, need a population of a certain size to look for the global optimum of the objective function in the search space. And if the shapes of these functions are more complex, it is necessary to consider the population size large enough, so that the entire search space of the objective function can be searched to find the global optimum. In the PSO algorithm, if the initial population size (the size given by the problem) is small, the search space of the problem will not be properly searched to find the optimum of the problem. Hence, it is necessary to increase the population size to cover the entire space of the problem. And in the meantime, in order to prevent the proposed algorithm from slowing, we execute it in parallel in the GPU through the CUDA platform. Not any algorithm can be executed in parallel, but it must rather have a structure and function, which can be easily parallelized. The PSO algorithm is a population-based algorithm, where each member of its population can search the search space of the objective function to find the optimum, with the aid of its best global member and the best history of its motion. And this mechanism allows the PSO algorithm to be executed in parallel. The proposed method to parallelize the enhanced PSO algorithm has the following main stages:

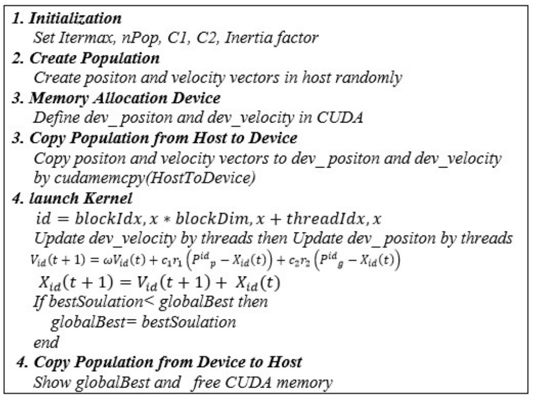
- The particles are encoded in the form of two vectors; the velocity vector and position vector. Each part of the position and velocity vectors is dedicated to a particle.
- The initial parameters of CUDA, including the number of blocks and threads of each block, and the parameters of the proposed algorithm, including the number of particles, and the number of iterations, are initialized.
- The values of the position and velocity vectors of the particles are randomly initialized.
- A proper memory is allocated for the velocity and position vectors of the particles in CUDA.
- Copying the velocity and position vectors from the host memory space to the machine's global memory
- Executing a kernel, which has the parallelization code of the proposed PSO algorithm. This kernel is executed by a series of threads contained in a block.
- Sequential iteration of the kernel to change the position of particles in the search space of the problem
- Updating the best global particle in each execution of the kernel
- Transferring the best global particle from the machine's global memory to the host main memory
- Evaluating the runtime
- Releasing the memories used in the definition of the velocity and position vectors of the particles.

In the method proposed to parallelize the PSO algorithm, we can define two arrays: (V_1^1, V_2^1,….,V_D^1, V_1^n, V_2^n, ….,$V_D^{n)}$ and (x_1^1, $x_2^{1,\ ….,}$ x_D^1, f^1, x_1^n, x_2^n …, V_D^n,f^n), which store the speed and position of all the particles in themselves, and then using the cudaMalloc command, their required memory can be allocated. In this structure of the desired vectors, n is the number of particles, D the dimensions of the evaluation function, f^i the fit of the i-th particle, v_j^i the velocity of the j-th particle, and x_j^i the position of the j-th particle.The position and velocity vectors are equal to n×(D+1) and n×D, respectively.

Then the values of these two vectors are randomly initialized in the main memory, and using the Cudamemcpy command and Cudamemcpyhosttodevice switch, the values of the two arrays are transferred from the host main memory to the GPU global memory. Then, each thread executes a version of the kernel, which contains the parallel commands of the proposed algorithm to determine the new position of the particles. To execute the kernel of changing the particle positions, CUDA considers the number of threads to be equal to the number of the initial population members. After determining the position of each particle at the end of the current iteration, it is necessary to calculate the best global particle of the population by calling the second kernel, and then update its position. In this situation, the previous kernel exits from the memory of threads and takes action for the parallel execution of the second kernel. To execute the kernel of updating the position or updating the best global particle, each thread obtains its unique ID number from Equation (3). And through this ID number, it can gain access to cells i to i+D+1 of the position vector and i+D of the velocity vector, which are in the machine's global memory.

$$i = blockIdx.x * blockDim.x + threadIdx.x \qquad (3)$$

After the completion of iteration, using the Cudamemcpy command and Cudamemcpydevicetohost switch, the best global particle algorithm, which is in the machine's global memory, is transferred into the host main memory, and is taken to the outlet, and eventually, the memories used by CUDA, are released. Figure (7) shows the pseudo-code of the proposed method for the parallelization and acceleration of the PSO algorithm.



```
1. Initialization
   Set Itermax, nPop, C1, C2, Inertia factor
2. Create Population
   Create positon and velocity vectors in host randomly
3. Memory Allocation Device
   Define dev_positon and dev_velocity in CUDA
3. Copy Population from Host to Device
   Copy positon and velocity vectors to dev_positon and dev_velocity
   by cudamemcpy(HostToDevice)
4. launch Kernel
   $id = blockIdx, x * blockDim, x + threadIdx, x$
   Update dev_velocity by threads then Update dev_positon by threads
   $V_{id}(t+1) = \omega V_{id}(t) + c_1 r_1 \left(P^{id}{}_p - X_{id}(t)\right) + c_2 r_2 \left(P^{id}{}_g - X_{id}(t)\right)$
   $X_{id}(t+1) = V_{id}(t+1) + X_{id}(t)$
   If bestSoulation< globalBest then
      globalBest= bestSoulation
   end
4. Copy Population from Device to Host
   Show globalBest and  free CUDA memory
```

Fig. 7. The pseudo-code of the proposed method in the acceleration of the PSO algorithm

IV. ANALYSIS

To assess the accuracy and speed of convergence in evolutionary algorithms, such as: the PSO algorithm, there are mathematical functions called "benchmark functions", which are considered as the objective function of the problem. In this objective functions, which represent an optimization problem, finding the global minimum is the final goal of the desired evolutionary algorithm. Table (1) shows a number of benchmark functions, which are used to assess the accuracy and speed of the proposed algorithm, along with the criteria and values of variables, for which the function becomes minimum:

TABLE I. THE BENCHMARK FUNCTIONS USED IN THE SIMULATION

CRITERION	Benchmark Function
$f = \sum_{i=1}^{n} x_i^2$	Sphere
$f = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$	Rosenbrock
$f = 1 + \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$	Griewank
$f = 10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i))$	Rastrigin

Three-dimensional graphs for benchmark functions have been shown in Figures (8), (9), (10), and (11):

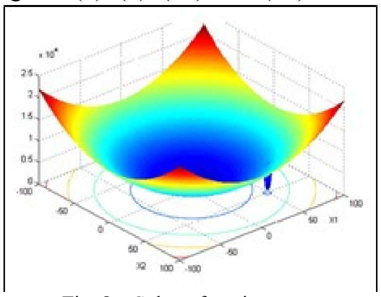


Fig. 8. Sphere function

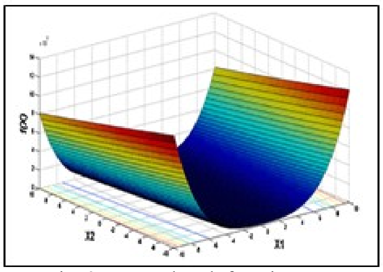


Fig. 9. Rosenbrock function

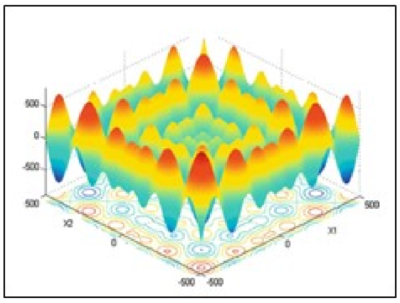


Fig. 10. Griewank function

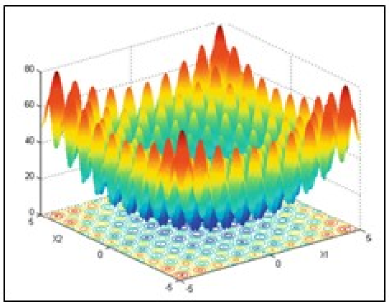


Fig. 11. Rastrigin function

In the second stage of the study, to improve the speed of the proposed algorithm, it is necessary to implement it in parallel in the CUDA platform, and determine the efficiency of the algorithm by calculating the runtime of its serial and parallel versions. The benchmark functions: Sphere, Rosenbrock, Three Hump, Easom, Beale, Sum Squares, Griewank, and Rastrigin are also used in this section, to assess the speed of the parallel version relative to the serial version. The two important criteria: runtime and acceleration are usually used in CUDA to assess the efficiency of parallel algorithms. In Equation (4), the GPU acceleration is defined as the CPU to GPU runtime ratio:

$$\text{Speedup} = \frac{Runtime\ CPU}{Runtime\ GPU} \qquad (4)$$

Pieces of hardware including:

CPU: Intel i5-3337U CPU@1.80GHz,

4GB of RAM, and

GPU: GeForce 710M with 1GB of memory were used for the simulation and implementations.

To calculate the acceleration of the proposed algorithm, which is the same CPU to GPU runtime ratio, we consider the initial population as the variable input of the problem, and add to the number of its members, then calculate the acceleration values successively. In this section, for each benchmark function, we carried out 50 different tests in CUDA, and calculated the runtime of the parallel bat and standard algorithms, then we calculated the mean time, and used it to calculate the average acceleration in each benchmark function. In evolutionary algorithms, the population size plays an important role in their accuracy and of course the speed of their execution. And the greater the size of their initial population is, the more the accuracy of the algorithm increases, and of course the run time also increases, thus reducing the speed of the algorithm. The average acceleration graph for populations with sizes of 1024, 2048, 4096, and 8192, shows that the efficiency of a GPU increases with the increased size of data, and the increased size of data, or the same initial population, can provide the proposed algorithm with an appropriate acceleration.

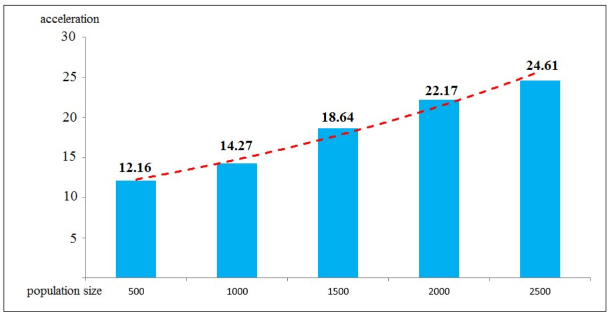


Fig. 12. Increasing the acceleration by increasing the population size

Analyzing the graph of acceleration versus initial population size shows that an increase in the population size causes an increase in the runtime of the bat algorithm both in the CPU and GPU. But the increase is much higher in the CPU than in the GPU, which causes the CPU to GPU run time ratio to constantly increase, in a way that it causes the acceleration rate to have an ascending trend as the population increases. In general, analyzing the simulation outputs and acceleration results in different benchmark functions, shows that by increasing the population size of particles as the input data of the problem, the runtime constantly increases in the GPU and CPU, in such a way that the runtime increase rate is much greater in the CPU than in the GPU, which causes the CPU to GPU runtime ratio to increase as the population size increases. To put it in better words, the efficiency of a GPU shows itself better, when our data are large enough.

V. Conclusion

The PSO algorithm is an evolutionary algorithm with a swarm intelligence approach, where a collection of particles can search the search space of the problem in parallel. Nonetheless, by increasing the initial population size in the PSO algorithm, its runtime increases too much. Therefore, in this paper, we presented a parallelization method based on CUDA, to improve the speed of the PSO algorithm. The results of implementing the proposed method in the GPU using the CUDA platform, show that an increase in the population size, increases the runtime in the proposed algorithm and PSO algorithm.

But, this increase is smaller in the GPU than in the CPU, which causes an increase in the acceleration, which is the GPU to CPU runtime ratio, as the population size increases. In future studies, we are going to use the CUDA parallel computing platform to accelerate other swarm intelligence algorithms, so that the optimal solutions of a problem are calculated faster.

References

[1] Zang, H., Zhang, S., & Hapeshi, K. (2010). A review of nature-inspired algorithms. *Journal of Bionic Engineering*, *7*, S232-S237.

[2] Tsai, H. C., & Lin, Y. H. (2011). Modification of the fish swarm algorithm with particle swarm optimization formulation and communication behavior.*Applied Soft Computing*, *11*(8), 5367-5374.

[3] Wang, G. G., Gandomi, A. H., & Alavi, A. H. (2014). Stud krill herd algorithm. *Neurocomputing*, *128*, 363-370.

[4] James, J. Q., & Li, V. O. (2015). A social spider algorithm for global optimization. *Applied Soft Computing*, *30*, 614-627.

[5] Emary, E., Yamany, W., Hassanien, A. E., & Snasel, V. (2015). Multi-Objective Gray-Wolf Optimization for Attribute Reduction. *Procedia Computer Science*, *65*, 623-632.

[6] Li, X., Zhang, J., & Yin, M. (2014). Animal migration optimization: an optimization algorithm inspired by animal migration behavior. *Neural Computing and Applications*, *24*(7-8), 1867-1877.

[7] Abdullah, A., Deris, S., Mohamad, M. S., & Hashim, S. Z. M. (2012). A new hybrid firefly algorithm for complex and nonlinear problem. In *Distributed Computing and Artificial Intelligence* (pp. 673-680). Springer Berlin Heidelberg.

[8] Łukasik, S., & Kowalski, P. A. (2015). Study of flower pollination algorithm for continuous optimization. In *Intelligent Systems' 2014* (pp. 451-459). Springer International Publishing.

[9] Ahmadi, M., & Mojallali, H. (2012). Chaotic invasive weed optimization algorithm with application to parameter estimation of chaotic systems.*Chaos, Solitons & Fractals*, *45*(9), 1108-1120.

[10] Wang, G. G., Hossein Gandomi, A., Yang, X. S., & Hossein Alavi, A. (2014). A novel improved accelerated particle swarm optimization algorithm for global numerical optimization. *Engineering Computations*, *31*(7), 1198-1220.

[11] Harish, P., & Narayanan, P. J. (2007). Accelerating large graph algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007* (pp. 197-208). Springer Berlin Heidelberg.

[12] Chunming; D. Simon; Aug.2005, "A new particle swarm optimization technique", 18th International Confereces System Engineering,2 .ICSEng 2005 ,pp.164-169.

[13] Alam, S., Dobbie, G., Koh, Y. S., Riddle, P., & Rehman, S. U. (2014). Research on particle swarm optimization based clustering: a systematic review of literature and techniques. Swarm and Evolutionary Computation,17, 1-13.

[14] Cheng, J., Grossman, M., & McKercher, T. (2014). Professional Cuda C Programming. John Wiley & Sons.

[15] Nvidia, C. U. D. A. (2013). CUDA C Programming Guide 5.5. NVIDIA Corporation, Jul.

[16] Verdonck, S., Meers, K., & Tuerlinckx, F. (2015). Efficient simulation of diffusion-based choice RT models on CPU and GPU. Behavior research methods, 1-15.

[17] Manconi, A., Manca, E., Moscatelli, M., Gnocchi, M., Orro, A., Armano, G., & Milanesi, L. (2015). G-CNV: a GPU-based tool for preparing data to detect CNVs with read-depth methods. Frontiers in bioengineering and biotechnology, 3.

[18] Kasim, H., March, V., Zhang, R., & See, S. (2008). Survey on parallel programming model. In Network and Parallel Computing (pp. 266-275). Springer Berlin Heidelberg.

[19] Mejía-Roa, E., Tabas-Madrid, D., Setoain, J., García, C., Tirado, F., & Pascual-Montano, A. (2015). NMF-mGPU: non-negative matrix factorization on multi-GP