GPU based Parallel Cooperative Particle Swarm Optimization using C-CUDA: A Case Study

Jitendra Kumar, Lotika Singh, Sandeep Paul
Department of Physics & Computer Science
Dayalbagh Educational Institute, India
Email: {jitendrakumar, lotikasingh, spaul.dei}@ieee.org

***Abstract*—The applications requiring massive computations may get benefit from the Graphics Processing Units (GPUs) with Compute Unified Device Architecture (CUDA) by reducing the execution time. Since the introduction of CUDA, applications from different areas have been benefited. Evolutionary algorithms are one such potential area where CUDA implementation proves to be beneficial not only in terms of the speedups obtained but also the improvement in convergence time. In this paper we present a detailed study of parallel implementation of one of the existing variants of Particle Swarm Optimization which is Cooperative Particle Swarm Optimization (CPSO). We also present a comparative study on CPSO implemented in C and C-CUDA. The algorithm was tested on a set of standard benchmark optimization functions. In this process, some interesting results related to the speedup and improvements in the time in convergence were obtained. The differences in randomizing procedures used in CUDA seem to contribute towards the diversity in population leading to better solution in contrast with the serial implementation. It also provides motivation for further research on neural network architecture and weight optimization using CUDA implementation. The results obtained in this paper therefore re-emphasize the utility of CUDA based implementation for complex and computationally intensive applications.**

Keywords*—*Cooperative Particle Swarm Optimization (CPSO), Graphics Processing Unit, Compute Unified Device Architecture

I. Introduction

Particle Swarm Optimization (PSO) is one of the most well known stochastic optimization algorithms introduced by Kennedy and Eberhart [1], [2]. The algorithm's methodology follows the behaviour of flock of the birds or the sociological behaviour of a group of people. It has been successfully used in various problems including vehicle routing [3], [4], neural networks [5], [6], optimization [7].

Like in any other evolutionary algorithm the initial population in PSO is a set of randomly chosen candidate solutions. The particles try to discover a good solution with the help of information provided by the neighbors. Since the introduction of the classical PSO algorithm a wide range of variants of PSO have been reported in the literature. One of the earliest variants include [8] which introduces the concept of inertia weight. There are variants to inertia weight implementation itself including [9], in which the inertia weight is dynamically learned using a fuzzy system. Other variants include the Conventional PSO (cPSO) and global-local best values based PSO (GLbest-PSO) [10]. Hybrid versions of PSO also exist which use standard particle swarm optimization along with classical genetic operators like crossover, one of them is in [11]. Some more variants of the particle swarm optimization include [12]–[16].

Various successful attempts to parallelize the PSO algorithm and variants also exist in the literature. In [17] the authors have done a parallel implementation of modified PSO using Message Passing Interface (MPI), PSO has been implemented using CUDA on GPU in [18], [19] and [20] and effectiveness of parallel PSO is shown by testing the algorithm on standard benchmark functions. Other evolutionary algorithms have also been implemented on GPU like Differential Evolution using C-CUDA [21] . In the paper authors have shown the reduction in execution time of parallel implementation using C-CUDA. Since the parallel implementations reduce the execution time dramatically, numerous applications are benefitting from it. Some of them include [22] in which a parallel implementation of K-Nearest Neighbor (KNN) algorithm is presented, in [23] CUDA based PSO was used for geophysical inversion, in [24] authors developed a parallel algorithm for option pricing based on PSO and implemented it using MPI, [25] presents a GPU based optimization in traffic signal coordination using evolutionary algorithms like Genetic Algorithms (GA), Ant Colony Optimization (ACO), and PSO.

CPSO (Cooperative Particle Swarm Optimization) is also one of the existing variants of PSO introduced by Van den Bergh and Engelbrecht [26]. CPSO uses the idea of cooperation between individuals of a population for solving a problem. The computational intensive nature of CPSO makes it an extremely suitable candidate for implementation with high performance computing. In this paper we are presenting the C-CUDA implementation of CPSO-S [26]. The implementation was carried out using five standard benchmark functions. A comparison of the parallel and sequential version of CPSO is then performed which shows the efficiency of the CUDA implementation both qualitatively and quantitatively.

II. Compute Unified Device Architecture (CUDA)

The Graphics Processing Units (GPUs) have recently emerged as one of the popular computing devices. The GPUs are designed by having the graphics processing architecture and parallel computing architecture [27]. Since GPUs' parallel computing architecture is optimized for computationally intensive problems, this technology may be evident for problems having a large amount of computation. A software and hardware architecture namely CUDA was introduced by NVIDIA to use GPUs for general purpose computing. CUDA enabled GPUs are a set of streaming multiprocessors (SM) and these SMs are a set of stream processors (SP) also known as

cores. GPUs have their own memory hierarchy having global memory, shared memory, registers etc. as shown in Fig. 1.

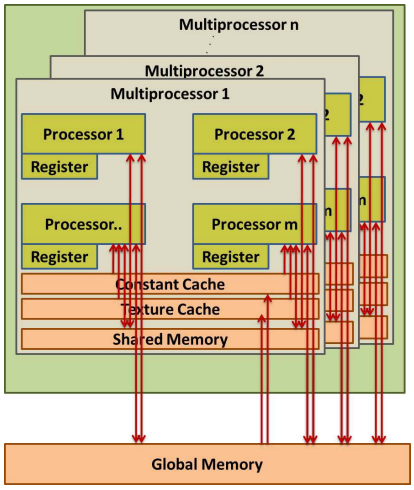

Fig. 1. GPU memory hierarchy model

CUDA provides an environment for CPU+GPU heterogeneous computing. CUDA is well suited for data parallel programs like SIMD (Single Instruction Multiple Data) programming model. Since CUDA programs can be executed using massive number of threads in parallel with the help of parallel architecture of GPU, it is also known as SIMT (Single Instruction Multiple Threads) programming model. The device code is called *kernel*. The kernel is not the complete program, it is a functional unit of the code to be executed on GPU. The CUDA describes thread architecture using the concepts of *grid, block,* and *threads* as shown in Fig. 2.

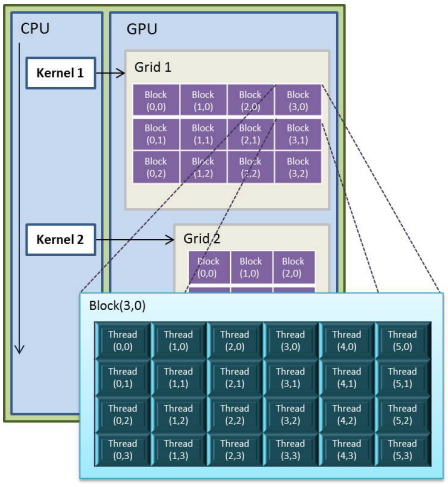

Fig. 2. Grid, block and threads within GPU

CUDA is a combination of C and some extended libraries including optimized functions for GPU which provides APIs to control the GPU. There are various optimized libraries provided by NVIDIA like cuBLAS, cuFFT which enable the fast execution of the included functions on the GPU. Thus CUDA programming model permits the programmers to better use the parallel power of the GPUs for general purpose computing. It therefore is becoming an extremely important and efficient tool when dealing with problem which are computationally very intensive. Multiple threads working simultaneously makes the C-CUDA working environment an ideal tool for data parallel applications specifically. Evolutionary algorithm is one potential area in which the inherent parallel nature of the algorithms makes them ideally suitable for C-CUDA implantation.

A. Evolutionary Computing using CUDA

Today evolutionary algorithms (EAs) are being used widely in research and applications. Justification for the parallel implementation of EA's comes from the fact that the fitness evaluation of one individual does not depend on the others, thereby giving rise to data parallelism which therefore makes it capable to be evaluated in parallel. Fitness evaluation of individuals is usually considered to be the most expensive operation in EAs. A detailed survey on parallelism of evolutionary algorithms has been presented in [28].

There are several papers in literature which include parallel implementation of the evolutionary algorithm(s) some of these include [18], [19], [21], [23], [29]. The algorithm can be implemented using any of the parallel programming model like Message Passing Interface (MPI), OpenMP, CUDA. Some papers and applications which uses CUDA for parallel implementation include [18], [19], [21]–[23], [29]–[31].

III. The CPSO Algorithm

The Cooperative Particle Swarm Optimization (CPSO) [26] is one of the variants of Particle Swarm Optimization (PSO) which uses the concept of *cooperation*. The idea floated through this algorithm focuses on letting all the particles reach the global best in cooperative fashion instead of competing with other particles to search the global best. Clearwater et al. [32] define cooperation as follows: "Cooperation involves a collection of agents that interact by communicating information to each other while solving a problem." Though PSO also involves the cooperation where each particle send and receive information to/from its neighbors. But in that case the cooperation is performed when each particle communicates its local best (complete solution vector) to its neighbors in form of fitness values while in CPSO each particle exchange the information with its neighbors in the form of fitness value of a *context vector*. A context vector is formed by replacing one position in the global best solution vector by the current or local best position in one dimension at a time.

In CPSO instead of trying to optimize all n dimensions or components of the problem by a single vector, the population is divided into n different subpopulations each of 1-D and each of them now try to optimize only one component of the problem. The detailed pseudocode for the CPSO algorithm first introduced by Van den Bergh and Engelbrecht can be found in [26]. Though this algorithm does not guarantee to converge in global optima and may also stuck into local optima but the main advantage of this algorithm is to overcome the *two step forward, one step backward* phenomena [26].

IV. Implementation Details

CPSO permits dimension wise update in the population rather than string wise update as in the case of classical EAs. This causes the number of fitness evaluations to be

very large in the case of CPSO. The total number of fitness evaluation involved in one complete generation is $4ns$ (two fitness evaluations for $pBestPosition$ update and two for $gBestPosition$ update in each dimension for each string) where n is number of dimensions and s is population size. Fitness evaluation is considered to be one of the most expensive operations in evolutionary algorithms. CPSO is therefore highly suitable for parallel algorithm owing to the large number of fitness evaluations involved. The pseudocode of parallel implementation using GPU is given in Table I.

TABLE I. PSEUDOCODE OF GPU BASED CPSO ALGORITHM

```
1.  Position_Velocity_Init_Kernel();
        // Initializes positions and velocities
2.  Fitness_Evaluation_Kernel();
        // Evaluates the fitness of individuals
3.  Initialize personal best positions (pBestPosition) and personal best fitness
    (pBestFitness) with initial position and fitness values respectively
4.  Find_Global_Best_Kernel();
    repeat
      for each dimension j ∈ [1...n]
5.      pBestPosition_Update_Kernel();
            // Updates pBestPosition in j^th dimension of each particle
6.      gBestPosition_Update_Kernel();
            // Updates global best position (gBestPosition) in j^th dimension
7.      Position_Velocity_Update_Kernel();
            // Updates position and velocity of j^th dimension using
               standard PSO equations - (1) and (2)
    until (termination criteria is met)
```

In implementation scheme shown in Table I the *kernel* functions are device functions which are called by host. A brief description of each function is given below:

1) *Position_Velocity_Init_Kernel()* function
Initialization of position and velocity was performed on GPU instead of CPU mainly to save the data transfer but in addition it also saves the time by initializing the values in parallel. One block of k threads initializes the values of one individual. Each thread was responsible to initialize maximum $\lceil n/k \rceil$ elements for position and velocity each (where n is number of dimensions).
2) *Fitness_Evaluation_Kernel()* function
In fitness evaluation one block of k threads was mapped to evaluate the fitness of one individual. Here also each thread was responsible to evaluate a maximum of $\lceil n/k \rceil$ elements and the result was stored in shared memory of the respective block using *atomic* operations (atomic operations are used to ensure that only one thread is writing at one time). As shown in Fig. 3 j^{th} thread of i^{th} block evaluates the $\{j^{th}, (j+k)^{th}, (j+2k)^{th}, ..., (j+pk)^{th}\} < n$ elements of i^{th} individual (where $p = (int)(n/k)$).
3) *Find_Global_Best_Kernel()* function
This kernel function uses only one thread and finds out the global best fitness value (*gBestFitness*) and the global best vector (*gBestPosition*).
4) *pBestPosition_Update_Kernel()* function
This kernel function updates the personal best position (*pBestPosition*) in j^{th} dimension of each particle. It also generates *context vectors* and performs their evaluations. The update in *pBestPosition* takes place based on the fitness values of context vectors generated by the current *position* and the current *pBestPosition*. The context vector generation using current *position* for 0^{th} dimension is shown in Fig. 4. If the fitness value of context vector produced by current *position* in j^{th} dimension is better than the fitness value of context vector formed by *pBestPosition* in j^{th} dimension then the *pBestPosition* in j^{th} dimension is replaced by the current *position* in j^{th} dimension. Finally the context vectors formed by updated *pBestPosition* vectors are evaluated and the fitness values are stored.
5) *gBestPosition_Update_Kernel()* function
This kernel function invokes one block of k threads and is responsible to update the *gBestPosition* in j^{th} dimension. The global best position *gBestPosition* vector is evaluated and the fitness value of global best *gBestValue* is compared with the best context vector (based on fitness values) formed by the updated personal best position *pBestPosition* vectors. The better value survives in j^{th} dimension of *gBestPosition*.
6) *Position_Velocity_Update_Kernel()* function
Positions and velocities are updated using classical PSO equations - (1) and (2). The update in j^{th} dimension is done in parallel using s threads, one thread is responsible to update one element from position and one element from velocity vectors.

From Fig. 5 it can be noticed that in this scheme of implementation the device (GPU) performs most of the computations while the host (CPU) is involved in managing the GPU. The tasks for computation to GPU was allotted by the CPU and the termination criteria was also checked on CPU towards the end of each generation. Position and velocity initializations were done by device because of two main reasons. First it saved time by initializing it in parallel and secondly the data transfer was not required as opposed to initialization on CPU which involved the data transfer. The fitness evaluation which mainly causes the evolutionary algorithms to be time consuming was performed on the device in parallel. Each individual was evaluated by multiple threads in parallel and multiple individuals were evaluated simultaneously in separate blocks. Updates in personal, global, and current positions, and velocity were also done by the threads in parallel.

While working with CUDA, the first thing to do is memory allocation on the graphics card. Memory allocation on the GPU was done using *cudaMalloc()* which is an inbuilt function provided by CUDA libraries. Since CPSO is stochastic evolutionary algorithm, the positions and velocities are required to be randomly initialized. The positions and velocities were being initialized on the GPU in parallel. The implementation of fitness function in C-CUDA is different from a sequential implementation. The difference being that in C-CUDA the objective function was applied over multiple dimensions of each individual in parallel using multithreading and *atomic* operations.

There is a concept of existence of multiple swarm in the CPSO algorithm, where each swarm consists of the informa-

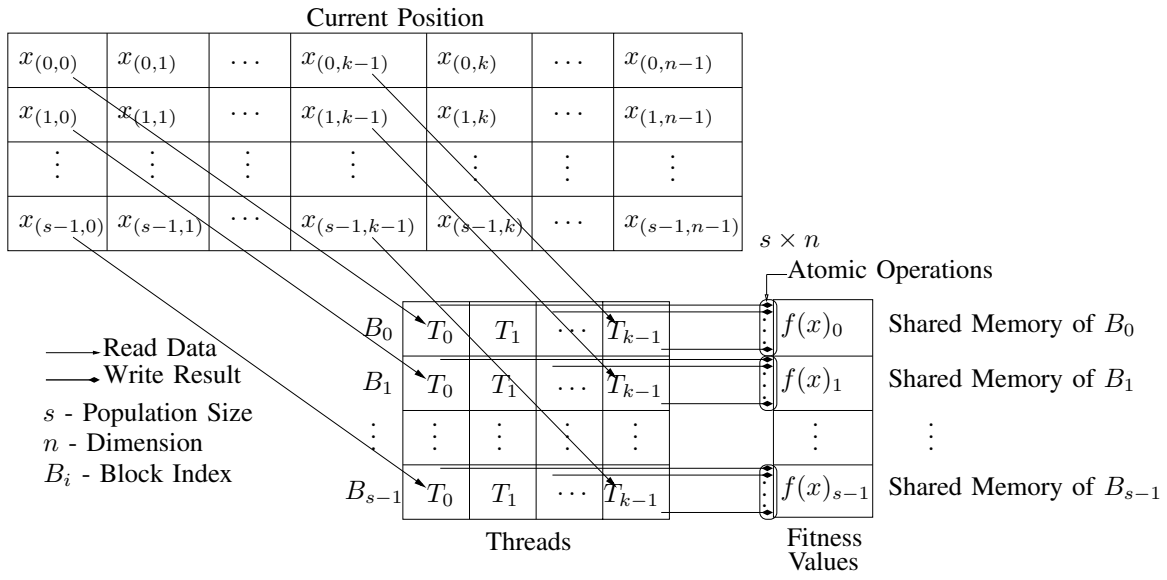

Fig. 3. Parallel implementation of fitness evaluation

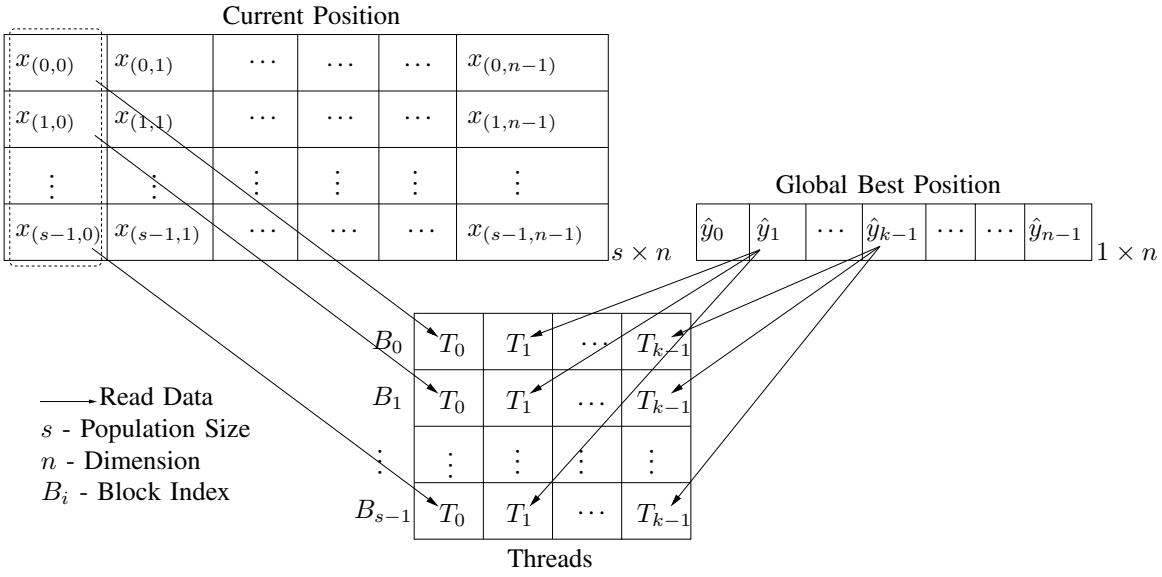

Fig. 4. Diagrammatic representation of context vector generation

tion about a particular dimension from all the individuals in the population. Therefore in CPSO each individual cooperates in finding the best result in each dimension. In order to update the *global best position* in j^{th} dimension, the *personal best positions* in j^{th} dimension are updated first. In this implementation the personal best positions of each individuals in j^{th} dimension were updated in parallel which if run sequentially would take a considerable amount of time. Then the global best positions were updated in j^{th} dimension. After updating the global best position in j^{th} dimension the positions and velocities of same dimension were updated using standard position and velocity update equations - (1) and (2) as referred to in [8].

$$v_{i,j}^{(t+1)} = wv_{i,j}^{t} + c_1 r_1 [y_{i,j}^{t} - x_{i,j}^{t}] + c_2 r_2 [\hat{y}_j^{t} - x_{i,j}^{t}] \quad (1)$$

$$x_i^{t+1} = x_i^t + v_i^{t+1} \quad (2)$$

Where c_1 and c_2 are positive constants, r_1 and r_2 are random number in range (0,1), and w is inertia weight. For all experiments in this paper we have taken $c_1 = c_2 = 2.05$ and $w = 0.25$.

V. Experimental Setup

The experiments were performed on the machine having one Tesla M-2070 graphics card. This graphics card has a global memory of 6GB. It consists of 14 SMs (streaming multiprocessors) each having 32 CUDA cores, which counts to total 448 cores on the GPU. The processor cores of this GPU has the frequency of 1.15 GHz. The GPU is capable to work with 1024 threads per block. Sequential experiments were done on the same machine (DELL PowerEdge M610) having 6 quad core Intel®Xeon® X5650 processors with global memory (RAM) of 24GB.

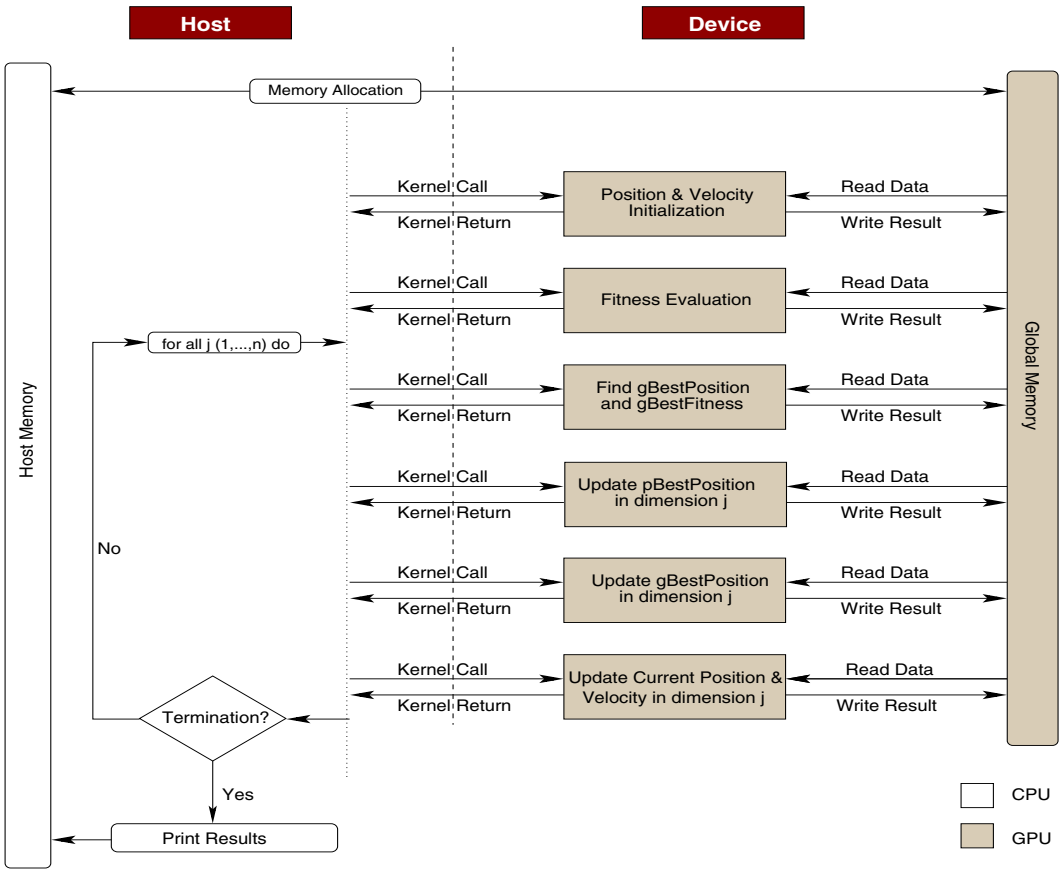

Fig. 5. Implementation diagram of CPSO on GPU using C-CUDA

VI. Results and Discussion

Experiments were performed on both unimodal and multimodal standard benchmark functions. These benchmark functions are shown in Table II [33] [34]. Each of the experiments were run 10 times and average result is reported.

The experiments were performed on problems of 1000 dimensions. Such large dimensions of the problem were taken to utilize the computational capability of the machine. In the case of simple and less computationally expensive problem, communication time increases as compared to computational time, thereby taking more time in communicating tasks to the threads rather than actually performing effective computation. This study therefore helps us to conclude that this kind of implementation is more suitable for problems with large data sets and computationally intensive.

To show the effectiveness of parallel over sequential implementation, speedup is one of the measures. It is the ratio of the execution time of sequential and parallel implementations.This measure gives us an estimate of the time which we save while performing parallel implementation of an algorithm. It is therefore helpful in solving problems involving large datasets in real time, which might otherwise take a lot of time. In this context the results of the experiments for our implementation are presented in Table III. In addition to the speedups, the subtle changes involved in the generation of random numbers due to the parallel implementation, the convergence rates and hence the performance of the algorithm itself improves. A detailed study on this aspect is covered in Table IV and V.

The effect of communication to computation ratio can be seen from the results presented in Table III and Fig. 6. These experiments were performed with functions $f_1(x)$ and $f_3(x)$ for population sizes of 500, 1000, and 1500. The simulations were carried out for 100 generations and the average over 10 runs is reported. The effectiveness of communication to computation can be seen by the results obtained from running the experiments with different number of threads per block. The results show that as we increase the number of threads from 16 to 96 threads per block the speedup increases, but from 128 threads per block onwards it starts decreasing as depicted in Fig. 6.

The explanation for this lies in CPSO's C-CUDA implementation itself. C-CUDA implementation not only split the entire population but also individuals. In this implementation we assigned one individual to one block of threads and each thread was assigned to evaluate at most $\lceil n/k \rceil$ elements of an individual where n is number of dimensions and k is number of threads in block. Since multiple blocks can run simultaneously multiple individuals were being evaluated using different blocks.

It can also be noticed from the Fig. 6 that speedup increases when we increase number of threads and keep problem size constant and vice-versa. But the obtained speedup by increasing the number of threads starts to decrease after a certain point. One of the possible reasons for this could be the GPU

TABLE II. TEST FUNCTIONS USED FOR THE EXPERIMENTS[1]

	Name	Function	D	Range
1	Shifted Sphere	$f_1(x) = \sum_{i=1}^{D} z_i^2 + f\_bias_1$	1000	$[-100, 100]^D$
2	Shifted Elliptic	$f_2(x) = \sum_{i=1}^{D} (10^6)^{\frac{i-1}{D-1}} z_i^2$	1000	$[-100, 100]^D$
3	Shifted Rastrigin	$f_3(x) = \sum_{i=1}^{D} \left(z_i^2 - 10cos(2\pi z_i) + 10\right) + f\_bias_3$	1000	$[-5, 5]^D$
4	Shifted Rosenbrock	$f_4(x) = \sum_{i=1}^{D-1} \left(100(z_i^2 - z_{i+1})^2 + (z_i - 1)^2\right) + f\_bias_2$	1000	$[-100, 100]^D$
5	Shifted Ackley	$f_5(x) = -20exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} z_i^2}\right) - exp\left(\frac{1}{D}\sum_{i=1}^{D} cos(2\pi z_i)\right) + 20 + e + f\_bias_4$	1000	$[-32, 32]^D$

z = (x-o), **x** = $[x_1, x_2, ..., x_D]$, **o** = $[o_1, o_2, ..., o_D]$; D: dimensions

$f\_bias_1 = -450.00$, $f\_bias_2 = 390.00$, $f\_bias_3 = -330.00$, $f\_bias_4 = -140.00$

architecture itself. When we use 16 threads per block, each thread was responsible to evaluate maximum of 63 elements while working with 96 threads per block each thread was responsible to evaluate at most 11 elements. As the number of threads per block increases the number of elements to evaluate for each thread decreases; hence reduction in computation time. But increment in number of threads results the increment in thread management time e.g. thread initialization, thread switching etc. In these experiments the ratio of communication time to computation time is found to be balanced with 96 threads per block hence achieving a higher speedup.

Similar experiments were performed with $f_3(x)$ (which is a multimodal function) and a similar trend as describe above for $f_1(x)$ can be observed in Fig. 6. For $f_1(x)$ we achieved the **11**$\times$ speedup while in the case of $f_3(x)$ it was $\approx$**60**$\times$. Based on these results shown in Fig. 6 it can be said that the speedup also depends on the complexity of the problem which means as the problem complexity increases the speedup also increases. The reason being that with increase in the problem complexity, the computational capability of the threads are being fully utilized in performing complex calculations.

TABLE III. SPEEDUP RESULTS USING C-CUDA AND C FOR $f_1(x)$ WITH 500, 1000, AND 1500 INDIVIDUALS AND 100 GENERATIONS

		Population Size		
		500	1000	1500
Sequential Execution Time (sec)		923.30	1847.10	2779.20
16 Threads/Block	Time (sec)	229.71	394.49	639.84
	Speedup	4.02	4.68	4.34
32 Threads/Block	Time (sec)	139.42	236.89	379.95
	Speedup	6.62	7.80	7.31
64 Threads/Block	Time (sec)	114.08	192.65	307.55
	Speedup	8.09	9.59	9.04
96 Threads/Block	Time (sec)	96.52	161.93	261.46
	Speedup	**9.57**	***11.41***	**10.63**
128 Threads/Block	Time (sec)	108.11	182.92	294.34
	Speedup	8.54	10.10	9.44
160 Threads/Block	Time (sec)	108.92	184.04	295.44
	Speedup	8.48	10.04	9.41
192 Threads/Block	Time (sec)	120.87	206.36	329.52
	Speedup	7.64	8.95	8.43
224 Threads/Block	Time (sec)	121.70	225.90	349.02
	Speedup	7.59	8.18	7.96

We now compare the execution time of sequential and parallel version the program for the five standard benchmark functions (see Table II). These results were obtained for a maximum of 1000 generations or when the population converged

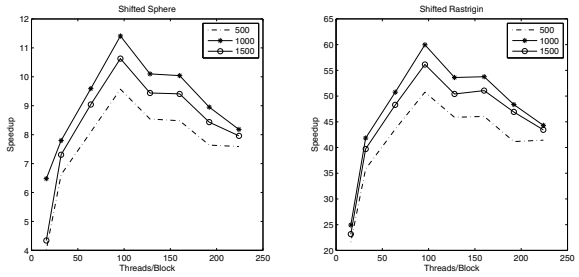

Fig. 6. Speedup results using C and C-CUDA for $f_1(x)$ and $f_3(x)$ for Population size 500, 1000, and 1500

(whichever being the earliest). It is evident from the Table IV, V that the performance with parallel implementation improves with respect to sequential version of the program in terms of convergence and hence execution time in almost all cases.

On taking a smaller population size of 50, it can be observed from Table IV that the population has converged to the optimal solution. The table depicts the time in which the convergence is achieved or maximum generation has elapsed, whichever is earliest. In this paper we define it to be the *Solution Finding Time (SFT)*. The highest speedup (in terms of SFT) is obtained for $f_5(x)$ which is one of the most complex functions used in these experiments. Precisely the number of generations in which convergence was achieved for functions $f_1(x)$ through $f_5(x)$ were 89, 89, 195, 1481 and 72 respectively. On the other hand for the sequential implementation only function $f_2(x)$ was able to obtain an optimal value and reach convergence in 43 generations. For all the other serial implementations of the functions, convergence was not attained till 1000 generations.

TABLE IV. RUN TIME RESULTS USING C-CUDA AND C FOR THE BENCHMARK OPTIMIZATION FUNCTIONS WITH 50 INDIVIDUALS

Function	Solution Finding Time (SFT) in sec	
	C	C-CUDA
$f_1(x)$	923.00	96.76
$f_2(x)$	170.70	101.95
$f_3(x)$	5116.30	222.77
$f_4(x)$	1316.20	1211.48
$f_5(x)$	5227.60	122.03

Table V gives an interesting result which presents the fitness values obtained by two versions of the implementation

TABLE V. CONVERGENCE RESULTS USING C-CUDA AND C FOR THE BENCHMARK OPTIMIZATION FUNCTIONS WITH 50 INDIVIDUALS

Function	Global Best	C		C-CUDA	
		Obtained Fitness	Standard Deviation	Obtained Fitness	Standard Deviation
$f_1(x)$	-450.00	-449.99	0.69	-449.99	0.00
$f_2(x)$	0.00	0.00	0.00	0.00	0.00
$f_3(x)$	-330.00	-329.99	0.34	-329.99	0.00
$f_4(x)^a$	390.00	2160.89	227.11	581.60	0.06
$f_5(x)$	-140.00	-139.99	0.01	-140.00	0.00

[a]Converged using C-CUDA in 1481 generations with Obtained Fitness 390.01 and Standard Deviation 0.003

after convergence or after a maximum of 1000 generations as previously described. As is evident from the table that for all the functions C-CUDA implementation has converged (the standard deviation being zero) while in the case of sequential C implementation the algorithm has converged only for function $f_2(x)$ and $f_5(x)$, and still has large standard deviation for $f_4(x)$. This re-enforces the utility of C-CUDA implementation both in terms of time for convergence and overall speedup.

A detailed study on the convergence of population for function $f_1(x)$ and $f_3(x)$ were also carried out. These experiments were also performed for population size of 500, 1000, and 1500. These experiments were performed for a maximum of 250 generations or till convergence was achieved (whichever is first). The population in the sequential (C) code implementation was not able to achieve convergence in 250 generations in either case but with parallel (C-CUDA) implementation convergence was achieved within 250 generations for both $f_1(x)$ and $f_3(x)$. The reason for this may be due to the better randomness generated by the CUDA random number generator. In a sequential implementation, the random numbers are generated on a single CPU, while in the case of a C-CUDA implementation, different random numbers are generated with the help of different random states on the GPU. This provides more diversity in the population leading to better results.

Figure 7 shows the population convergence plot using 96 threads/block in C-CUDA against the sequential implementation for a population of 500, 1000 and 1500 individuals. (Only for better visibility of results in the plot, the standard deviations plotted in the figures are taken as the natural logs of the exact values calculated).

It can be noticed from the Fig. 7 that with the C-CUDA implementation for all the population sizes convergence was achieved in 150 generations. Here also the experiments were performed with different number of threads per block (96, 512, and 1000). More precisely, the standard deviations of $f_1(x)$ for population sizes 500, 1000, and 1500 were 0.001. The convergence was achieved after 94, 95, and 102 generations with 96 threads per block; 101, 111, and 115 generations with 512 threads per block; and 97, 104, and 103 generations with 1000 threads per block. While in the sequential case the standard deviations for population sizes 500, 1000, and 1500 after 150 generations were 29.48, 29.65, and 28.85 respectively. In the case of $f_3(x)$ the standard deviations were 0.001 with 96 threads per block for population sizes 500, 1000, and 1500. The number of generation elapsed to achieve convergence were 213, 232, and 207 respectively; with 512 threads per block the standard deviations were 0.002, 0.002, and 0.03 for population

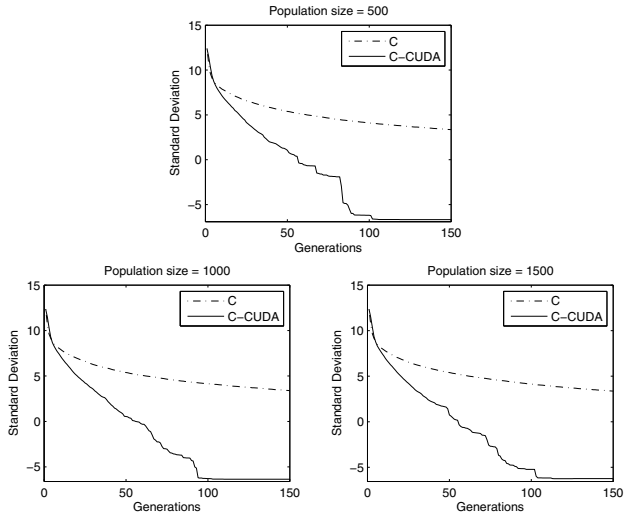

Fig. 7. Plot of Standard deviation (taken in natural log) versus number of generations for $f_1(x)$

sizes 500, 1000, and 1500 respectively after 250 generations; with 1000 threads per block standard deviations were 0.002 for population sizes 500, 1000, and 1500 after 250 generations. While in the case of sequential implementation the standard deviations were 4.029, 4.137, and 4.071 for population sizes 500, 1000, and 1500 respectively. For the sake of comparison we also used the Mersenne Twister Random Number Generator (MTRNG) [35] and similar results were obtained for sequential implementation. The obtained standard deviations after 250 generations for $f_1(x)$ were 10.867, 11.109, and 11.087 for population sizes 500, 1000, and 1500 respectively. In the case of $f_3(x)$ after 250 generations 4.069, 4.147, and 4.162 were the obtained standard deviation for population sizes 500, 1000, and 1500 respectively. It is clear from the obtained results that the parallel (C-CUDA) implementation was able to converge faster than the sequential implementation in both the cases when we were using the MTRNG and also on using the standard C random number generator.

VII. CONCLUSION

Parallel computing is being used in various areas particularly where large number of computations are involved e.g. bioinformatics, medical imaging, neural networks etc. This detailed case study shows that the problems involving massive computation can be solved efficiently using parallel computing models. During the experiments carried out for this paper we noticed an effective reduction in the execution time of C-CUDA over sequential C implementation. Parallel implementation therefore not only improves performance in terms of speedup but is also better qualitatively. The improvement in convergence can be attributed to the fact that the generation of random number for the algorithm take place differently (each streaming processor (SP) or core is generating random number in parallel using different CUDA random states), which may bring more randomness and hence diversity in the population. Although, this still needs to be studied further. C-CUDA implementation can be carried out further for studies related

to evolution of neural network weights, and the structure itself. The results in this paper also opens up avenues for further research related to more complex and computationally intensive real world problem.

ACKNOWLEDGMENT

The authors wish to thank Late Prof. Satish Kumar, former Head, Department of Physics & Computer Science, Dayalbagh Educational Institute for his constant encouragement and guidance. We wish to thank Dayalbagh Educational Institute, Agra, India for providing with the necessary infrastructure. We would also like to thank the reviewers for their detailed comments and suggestions that helped us improve the quality of paper significantly.

REFERENCES

[1] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proc. of the 6th Int'l Symposium on Micro Machine and Human Science (MHS)*, Oct. 1995, pp.39-43.

[2] R. C. Eberhart et al., *Computational Intelligence PC Tools*: Academic, 1996, Chapter 6, pp.212-226.

[3] Li Jian, "Solving Capacitated Vehicle Routing Problems via Genetic Particle Swarm Optimization," in *3rd Int'l Symposium on Intelligent Information Technology Application (IITA)*, vol.3, Nov. 2009, pp.528-531.

[4] Wu Bin et al., "Particle Swarm Optimization method for Vehicle Routing Problem," in *5th World Congress on Intelligent Control and Automation (WCICA)*, vol.3, June 2004, pp.2219-2221.

[5] W. Zhang et al., "Application of neural network based on particle swarm algorithm for the results of students," in *8th World Congress on Intelligent Control and Automation (WCICA)*, July 2010, pp.1218-1221.

[6] W. Xuan et al., "A Hybrid Particle Swarm Optimization Neural Network Approach for Short Term Load Forecasting," in *4th Int'l Conf. on Wireless Communications, Networking and Mobile Computing (WiCOM)*, Oct. 2008, pp.1-5.

[7] I. Ibrahim et al., "A Novel Multi-state Particle Swarm Optimization for Discrete Combinatorial Optimization Problems," in *4th Int'l Conference on Computational Intelligence, Modelling and Simulation (CIMSiM)*, Sept. 2012, pp.18-23.

[8] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *IEEE World Congress on Computational Intelligence, The 1998 IEEE Int'l Conference on Evolutionary Computation Proc.*, 1998, pp.69-73.

[9] Y. Shi and R.C. Eberhart, "Fuzzy adaptive particle swarm optimization," in *Proceedings of the Congress on Evolutionary Computation*, 2001, vol.1, pp.101-106.

[10] Zhan and Xiaojuan, "A Hybrid Variants Particle Swarm Optimization Algorithm," in *Int'l Conference on Machine Vision and Human-Machine Interface (MVHI)*, April 2010, pp.33-36.

[11] J. Park et al., "A Hybrid Particle Swarm Optimization Employing Crossover Operation for Economic Dispatch Problems with Valve-point Effects," in *Int'l Conf. on Intelligent Systems Applications to Power Systems (ISAP)*, Nov. 2007, pp.1-6.L

[12] X. Hu and R. Eberhart, "Multiobjective otimization using dyanmic neighborhood particle swarm optimization," in *Proc. of the Congress on Evolutionary Computation*, vol.2, 2002, pp.1677-1681.

[13] B. Jiao et al., "A dynamic inertia weight particle swarm optimization algorithm," in *Chaos, Solitons & Fractals*, vol.37, Aug. 2008, pp.698-705.

[14] Xinghua Wu, "A density adjustment based particle swarm optimization learning algorithm for neural network design," in *Int'l Conference on Electrical and Control Engineering (ICECE)*, Sept. 2011, pp.2829-2832. 2011.

[15] Y. Zhe-ping et al., "A novel two-subpopulation particle swarm optimization," in *10th World Congress on Intelligent Control and Automation (WCICA)*, July 2012, pp.4113-4117.

[16] M. Chen et al., "A Hybrid Particle Swarm Optimization Improved by Mutative Scale Chaos Algorithm," in *4th Int'l Conference on Computational and Information Science (ICCIS)*, Aug. 2012, pp.321-324.

[17] K. Deep et al., "Modified parallel particle swarm optimization for global optimization using Message Passing Interface," in *IEEE 5th Int'l Conf. on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, Sept. 2010, pp.1451-1458.

[18] L. de P. Veronese and R. A. Krohling, "Swarm's Flight: Accelerating the Particles using C-CUDA," in *IEEE Congress on Evolutionary Computation (CEC)*, May 2009, pp.3264-3270.

[19] Li Wenna and Z. Zhenyu, "A CUDA-based Multi-Channel Particle Swarm Algorithm," in *Int'l Conf. on Control, Automation and Systems Engineering (CASE)*, July 2011, pp.1-4.

[20] Z. You and T. Ying, "GPU-based parallel particle swarm optimization," in *IEEE Congress on Evolutionary Computation (CEC)*, May 2009, pp.1493-1500.

[21] L. de P. Veronese and R. A. Krohling, "Differential Evolution Algorithm on the GPU with C-CUDA," in *IEEE Congress on Evolutionary Computation (CEC)*, July 2010, pp.1-7.

[22] S. Liang et al., "A CUDA-based Parallel Implementation of K-Nearest Neighbor Algorithm," in *Int'l Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Oct 2009, pp.291-296.

[23] D. Datta et al., "CUDA based Particle Swarm Optimization for Geophysical Inversion,", in *1st Int'l Conf. on Recent Advances in Information Technology (RAIT)*, March 2012, pp.416-420.

[24] H. Prasain et al., "A parallel Particle swarm optimization algorithm for option pricing," in *IEEE Int'l Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010, pp.1-7.

[25] K. Wang and Z. Shen, "GPU based ordinal optimization for traffic signal coordination," in *IEEE Int'l Conference on Service Operations and Logistics, and Informatics (SOLI)*, July 2012, pp.166-171.

[26] F. van den Bergh and A. P. Engelbrecht, "Cooperative learning in neural networks using particle swarm optimizers," *South African Comput. J.*, vol. 26, Nov. 2000, pp.84-90.

[27] NVIDIA CUDA Programming Guide Version 3.2 2010

[28] E. Alba and M. Tomassini, "Parallelism and Evolutionary Algorithms," in *IEEE Trans. on Evolutionary Computation*, vol.6, no.5, Oct 2002, pp.443-462.

[29] C. Tsai, et al., "Parallel Elite Genetic Algorithm and Its Applicatoin to Global Path Planning for Autonomous Robot Navigation," in *IEEE Trans. on Industrial Electronics*, vol.58, no.10, Oct. 2011, pp. 4813-4821.

[30] A. Bustamam et al., "Fast Parallel Markov Clustering in Bioinformatics using Massively Parallel Computing on GPU with CUDA and ELLPACK-R Sparse Format," in *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, vol.9, no.3, May/June 2012, pp.679-692.

[31] C. Juang et al., "Speedup of Implementing Fuzzy Neural Networks with High-Dimensional Inputs Through Parallel Processing on Graphic Processing Units," in *IEEE Trans. on Fuzzy Systems*, vol.19, no.4, Aug. 2011, pp.717-728.

[32] S. H. Clearwater et al., "Cooperative problem solving," in *Computation: The Micro and Macro view, Singapore: World Scientific*, 1992, pp.33-70.

[33] K. Tang et al., "Benchmark Functions for the CEC'2008 Special Session and Competition on Large Scale Global Optimization," http://sci2s.ugr.es/programacion/workshop/Tech.Report.CEC2008.LSGO.pdf, 2008.

[34] Ke Tang et al., "Becnchmark Functions for the CEC'2010 Special and Competition on Large-Scale Global Optimization," http://sci2s.ugr.es/eamhco/cec2010_functions.pdf, 2010.

[35] "Mersenne Twister: A random number generator," http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.