GPU-Accelerated Standard and Multi-Population Cultural Algorithms

Jianqiang Dong and Bo Yuan

Intelligent Computing Lab, Division of Informatics

Graduate School at Shenzhen, Tsinghua University
Shenzhen 518055, P. R. China
513712287@qq.com, yuanb@sz.tsinghua.edu.cn

***Abstract*—In this paper, we present three parallel cultural algorithms using CUDA-enabled GPUs. Firstly, we used the GPU to accelerate an expensive fitness function. Next, the parallel versions of both standard and multi-population CAs were presented. Experiments show that the standard CA with an expensive fitness function was made more than 600 times faster. On lightweight benchmark problems, the speedups were only 3-4 times for the standard CA while the multi-population CA can still achieve 30-50 times speedups.**

Keywords-GPU; CUDA; acceleration; cultural algorithms; multi-population; clustering;

I. INTRODUCTION

With the rapid development of evolutionary algorithms (EAs) since 1970s [1, 2], many population based algorithms inspired by the nature have been proposed such as genetic algorithms (GAs), which are loosely based on the concepts of genes and natural selection, and particle swarm optimization (PSO), which mimics the behaviour of a swarm of birds. By contrast, cultural algorithms (CAs) imitate the human society and maintain an extra level of evolutionary information called belief space [3]. In this way, the population can be guided by the belief space and generate new knowledge to modify the belief space. With this unique belief space, CAs can have better understanding of the problem and may converge faster with a better solution.

A major inherent issue of EAs compared to classical optimization techniques is their computational complexity. Usually, the population may contain hundreds of individuals and the evolutionary process may also need hundreds of iterations, resulting in tens of thousands fitness evaluations. This is particularly problematic if the objective function itself is computationally expensive.

The situation has been getting significantly better with the emergence of the groundbreaking GPU (Graphics Processing Unit) computing technology in the past decade. GPUs were originally designed for handling the intensive parallel computation required by high quality computer graphics and later were gradually applied to solving general computing problems traditionally handled by CPUs, referred to as GPGPU. In its early stage, GPU computing was tedious and created a daunting challenge for researchers, requiring detailed knowledge of the sophisticated graphics pipeline. The introduction of CUDA (Compute Unified Device Architecture) in 2007, a heterogeneous parallel programming framework developed by NVIDIA [4, 5], improved the programmability of GPUs dramatically and it is now convenient for researchers in various disciplines to make full use of the power of GPUs. Currently, the latest CUDA version is 5.0 and Tesla K20X, the most advanced GPU produced by NVIDIA, features 2688 CUDA cores and 6 GB memory for conducting massively parallel computing**,** boosting its theoretical performance to 3.95 TFLOPS (single precision)**,** one order of magnitude faster than existing mainstream multi-core CPUs [6].

Furthermore, many-core GPUs are purposefully designed for multithreaded computing while CPUs need to handle complex logic control operations and various computing tasks. The memory architecture of GPUs is also customized for intensive and data parallel computing tasks with much higher memory bandwidth [7]. In nowadays, CUDA-enabled GPUs have been incorporated in many Top500 supercomputers [8] and widely applied in fluid dynamics, life science, signal processing, physics and finance etc. with speedups ranging from one to three orders of magnitude.

In this paper, we presented three GPU implementations of CAs to demonstrate the great potential and possibility of advanced GPU computing in this class of algorithms:

- Firstly, we used the GPU to accelerate an expensive fitness function (each thread calculated part of the function). In this experiment, a maximum 614 times speedup was realized.
- Secondly, we used five lightweight fitness functions and the entire population was parallelized (each thread calculated the fitness of an individual). Due to the impact of the sequential components of the CA and the underutilization of GPU resources, a moderate 3-4 times speedup was obtained.
- Thirdly, with the same fitness functions, the multi-population CA managed to achieve 30-50 times speedups, as it is more suitable for GPU computing.

In the rest part of this paper, Section II presents the details of CAs and analyses its potential in parallel computing. Section III introduces CUDA technology and Section IV specifies the three experiments and GPU implementations. The experimental results are shown in Section V and this paper is concluded in Section VI.

II. CULTURAL ALGORITHMS

Cultural algorithms are a branch of evolutionary algorithms using both population space and belief space to perform evolution. They are inspired by the spread, evolution and influence of culture in human society.

A. Framework

After problem encoding (representation) and initializing the population space and the belief space, each iteration in CAs consists of 4 steps: i) acquisition of new knowledge from the population; ii) updating the belief space; iii) using the belief space to alter the population space; iv) evolving the population space by population based algorithms such as GAs and PSO. It is easy to see that CAs are extensions to traditional EAs, with an extra knowledge component.

B. Development

CAs were originally introduced in 1994 and a formal definition of CAs was given in 1999 [3, 9]. To improve the performance of CAs in multi-optimal or multi-objective scenarios, the multi-population CA was proposed where individuals from different populations can be exchanged [10]. In 2009, Guo et al. presented a scheme called knowledge migration to make populations exchange information much more efficiently and effectively [11]. In the meantime, CAs have been widely applied in optimization [12], neural networks [13], data mining [14] and industry [15].

C. Parallel CAs

Both the standard CA and multi-population CA can be parallelized, in a way similar to other population based algorithms [16-18]. To the best our knowledge, there has not been any work on parallel CAs using GPUs. Since the fitness function in many tasks is the most computationally intensive component, the fitness of each individual in the population can be evaluated in parallel (population-level). Alternatively, for expensive fitness functions, it is also possible to parallelize the functions themselves (function-level).

Other parts of CAs can be also parallelized such as new population generation, selection scheme, accept function, influence function and the evolution of the belief space. In CAs, the new population generation and the influence function always work together. Each child has only one parent while all children are influenced by a common knowledge component. The selection procedure, accept function and the evolution of the belief space consist of large amount of function calls for sorting, random number generation and searching for the maximum, and all of these procedures can be parallelized with GPUs.

Note that the standard CA has only one population and only a single thread block can be used due to the issue of synchronization and communication. For multi-population CAs, each population can evolve within one thread block and, after several iterations, the migration process can be executed in the CPU, making full use of the GPU computing power and enjoying the benefits of knowledge migration.

III. CUDA

In CUDA, the computing resources of GPUs are divided into SMs (Streaming Multiprocessor) each of which consists of a group of SPs (Streaming Processor) or cores, the basic computing units in GPU. Threads are organized into blocks, which can be freely scheduled among SMs. Threads within the same block can synchronize and communicate via shared memory while threads in different blocks are independent. For parallel EAs, each thread usually corresponds to an individual and each block works as a population, and the whole grid holds the information of multiple populations. In the GPU, tens of thousands threads can be executed concurrently with zero-overhead scheduling (in contrast to only a few threads supported by the CPU), which can significantly reduce the idle time of processors and achieve high computing efficiency.

The memory architecture in CUDA is also very friendly to population based algorithms. For example, frequently used data such as the iteration information can be stored in register files while the population can be stored in the shared memory with fast access speed. In the meantime, nearly all population based algorithms have a heavy load of creating random numbers. CUDA provides random number generators in the `curand` library [19] and with the help of this library, various types of high quality random numbers can be generated efficiently and conveniently.

IV. EXPERIMENT SETTING

In this section, we present the specification of three experiments on parallel CAs from different perspectives.

A. Parallel Fitness Function

In the first experiment, a standard CA was used in a clustering task. The goal was to find a set of cluster centres so that the sum of the distances between each data point and its nearest centre is minimized. The CA itself was run in the CPU (each individual was evaluated sequentially) while the fitness function was parallelized and accelerated by the GPU as it can be very expensive for large datasets.

The CPU version pseudo code of the fitness function is shown as follows:

```
for i=1:num_sample
    for j=1:num_centre
       Compute distance[i][j]
       if j=1
          min_dist=distance[i][j]
       else
          if distance[i][j]<min_dis
            min_dist=distance[i][j]
          endif
       endif
    endfor
    total_dist+=min_dist
endfor
```

The total number of data points (num_sample) to be clustered, the number of blocks (Dim_b) and the number of threads in each block (Dim_t) were set in the CPU (step 1). In the GPU, each thread was responsible for evaluating num data points (num can take various values for different threads, depending on the specific configuration) to obtain the partial result (step 2). Finally, the CPU combined the partial results from each thread to obtain the final result (step 3). Note that the data and results need to be transferred between the CPU and the GPU. The pseudo code is shown as follows:

```
In CPU:
1.  Set Dim_b, Dim_t

In GPU:
2.  For the j^th thread in the i^th block, store
    its result in part_dist[i*Dim_t+j]

In CPU:
3.  total_dist=sum(part_dist)
```

B. Standard CA on GPU

In this experiment, a standard CA was executed on the GPU, and was compared with the CPU version in terms of running time. The focus was on the parallel implementation of the CA itself and the benchmark functions (Table I) were not parallelized. This choice is plausible when the fitness functions cannot be effectively parallelized. Note that only a single thread block can be used in this situation. The pseudo code is shown as follows:

```
In CPU:
1.  Initialize the population space
2.  Initialize the belief space
3.  Allocate GPU memory and copy population
    and belief info into GPU memory

In GPU (for each generation):
4.  Generate offspring under the guide of
    belief space and add them to the
    population (thread)
5.  Calculate the fitness value of each
    individual (thread)
6.  Compare each individual with other
    individuals and calculate the winning
    times(thread)
7.  Sort the winning times and get the
    selected individuals(block)
8.  Use the best individuals to update the
    situation knowledge(block)
9.  Update the norm knowledge (thread)

In CPU:
10. Copy the population info, situation info
    and norm info back to main memory
```

All data related to population, fitness, situation knowledge, norm knowledge and the winning times were stored into the shared memory inside the block, and other variables were stored in the registers of each thread.

TABLE I. BENCHMARK PROBLEMS

$$F_1(x) = \sum_{i=1}^{n} x_i^2$$

$$F_2(x) = \sum_{i=1}^{n} [x_i - 10\cos(2\pi x_i) + 10]$$

$$F_3(x) = \sum_{i=1}^{n} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

$$F_4(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}})^2 + 1$$

$$F_5(x) = 418.9829n - \sum_{i=1}^{n} x_i \sin(|x_i|^{0.5})$$

C. Multi-Population CA on GPU

In the standard CA experiment, only one of the many SMs in the GPU was used while other SMs were left idle. For multi-population CAs, each block holds the information of one population and several blocks can be executed in parallel, making much better use of the GPU resources and higher speedups can be expected. The pseudo code of this experiment is shown as follows:

```
In CPU:
1.  Initialize the population space of each
    population
2.  Initialize the belief space of each
    population
3.  Allocate GPU memory and copy population
    and belief information into GPU memory

In GPU (for each generation):
4.  Generate offspring under the guide of
    belief space and add them to the
    population (thread)
5.  Calculate the fitness values of all
    individuals in each population (thread)
6.  Compare each individual with other
    individuals and calculate the winning
    times within its population(thread)
7.  Sort the winning times and get the
    selected individuals in each population
    (block)
8.  Use the best individuals to update the
    situation knowledge (block)
9.  Update the norm knowledge(thread)
10. If the migration criteria are satisfied,
    go to step 11
```

```
In CPU:
11. Finish the GPU kernel and copy the
    population info, situation info and norm
    info to main memory
12. Migrate the best individuals between
    populations
13. If the stopping criteria are not met, go
    to step 4
14. Copy the population info, situation info
    and norm info to main memory and find the
    best solution from each population
```

V. EXPERIMENT RESULTS

In all three experiments, we used a Fermi-based NVIDIA GeForce GTX 560 Ti (8 SMs ×48 SPs = 384 cores) GPU, CUDA 5.0, Intel i5-2300 CPU, and Visual Studio 2010.

A. Parallel Fitness Function

For this experiment, we used the well known KDD CUP 1999 dataset [20] with 4,898,431 samples and 42 attributes. The number of clusters was set to 100. Note that this is a rather arbitrary number as the focus was not on finding the best clustering pattern but on the efficiency of clustering. The number of blocks and the number of threads in each block were fixed to 32 and 256, respectively. Table II shows the time in milliseconds required to evaluate the fitness function for a single time.

TABLE II. FITNESS EVALUATION BY CPU AND GPU

Sample Size	CPU	GPU	Speedup
400	50	86	0.58
4000	526	93	5.66
40000	4867	119	40.90
400000	48924	162	302.00
4898431	572994	761	752.95

From Table II, we can see that, with the help of GPU, the fitness function can be made up to 753 times faster than the CPU version. Also, the advantage of GPU computing became more evident as the size of the problem increased. Note that launching GPU kernels from the GPU also involves some overhead, which may compromise the performance of GPU for simple tasks. Also, for simple tasks, not all threads were functional.

For the full experiment, we used the following parameter settings: population size: 100, number of iterations: 10, possibility of acceptance: 0.7. We used the situation and norm knowledge as the belief space information. The results (in milliseconds) are shown in Table III and the speedup was up to 614 times.

Note that the overheads in GPU computing include transferring data between the CPU and the GPU as well as the launch of kernel functions. For a single function call, these overheads may be non-neglectable but for practical scenarios, they can often be averaged out over iterations. This is the reason that in some cases higher speedups were obtained even when the CA itself was not parallelized.

It should be mentioned that, for the last case with 4,898,431 data points, the CA was only allowed a single iteration on the CPU due to its prohibitively long running time (125,807,915 milliseconds or 35 hours). The value shown in Table III is the estimated value for 10 iterations.

TABLE III. RESULTS ON THE CLUSTERING TASK

Sample Size	CPU	GPU	Speedup
400	109074	2710	40.25
4000	1093754	4843	225.84
40000	10188151	19574	520.49
400000	100674532	166756	603.72
4898431	1258079150^	2048129	614.26

^Estimated Value

B. Standard CA on GPU

In this experiment, we investigated the efficiency of the CA with full GPU implementation. We used five 10D benchmark problems listed in Table I. The parameter values were: possibility of acceptance: 0.6, population size: 256, number of iterations: 5000 and the number of individuals for competition: 30. The results (in milliseconds) are shown in Table IV. Since all fitness functions are quite simple in computation, the sequential components of the CA may be relatively substantial (Amdahl's law) and the GPU was severely underutilized due to the single thread block in play. As a result, the speedups were not comparable to Table III.

TABLE IV. STANDARD CA ON FIVE BENCHMARK PROBLEMS

Function	CPU	GPU	Speedup
F1	7272	1760	4.13
F2	8778	2748	3.19
F3	7983	2583	3.09
F4	9763	2521	3.87
F5	10470	2393	4.38

C. Multi-Population CA on GPU

In this experiment, we showed how multi-population CAs can benefit from GPU computing. All parameters values were the same as previous. From the results in Table V and Table VI with different number of populations, it is clear that multi-population GAs can significantly benefit from GPU computing, mainly due to the effectiveness of multiple thread blocks (better utilization of GPU resources).

TABLE V. MULTI-POPULATION CA ON FIVE BENCHMARK PROBLEMS (NUMBER OF POPULATIONS = 32)

Function	CPU	GPU	Speedup
F1	233943	6318	37.03
F2	286149	7424	38.54
F3	257164	7016	36.65
F4	307432	7179	42.82
F5	280141	6861	40.83

TABLE VI. MULTI-POPULATION CA ON FIVE BENCHMARK PROBLEMS (NUMBER OF POPULATIONS = 128)

Function	CPU	GPU	Speedup
F1	977638	22847	42.79
F2	1068943	23296	45.89
F3	1024596	25339	40.44
F4	1193047	25442	46.89
F5	1147641	23787	48.25

VI. CONCLUSION

CAs are a branch of population based evolutionary algorithms that have been successfully applied to many problems. Due to the extra level of belief space, which needs to be updated during the evolution, the computational complexity is generally higher than some other EAs. In fact, it is not unusual in many disciplines that certain algorithms can produce superior performance but at the cost of efficiency. For large scale real-world problems, the running time of these algorithms may be intolerable.

With the popularity of GPU computing techniques such as CUDA and advanced GPU hardware, it is now possible to enjoy both effectiveness and efficiency at unprecedented convenience. Computationally intensive algorithms can be parallelized and accelerated by GPUs, running potentially hundreds times faster than their CPU counterparts. In data mining, this is particularly important for handling massive datasets, as most sequential algorithms cannot produce satisfactory results within reasonable amount of time.

In this paper, we have demonstrated empirically how GPU computing can significantly accelerate standard and multi-population CAs on both expensive and lightweight benchmark problems. We showed that CAs can be parallelized in two aspects: the parallel evaluation of the population and the parallel evaluation of the fitness function. The maximum speedup was achieved when the fitness function was significantly expensive. For lightweight fitness functions, the speedups were quite moderate for standard CAs as only a single thread block was in play. By contrast, multi-population CAs were able to take the advantage of GPU computing with multiple thread blocks.

In the future, we will further investigate the potential for parallelism of other EAs and data mining algorithms and there are many subtle performance factors to be carefully addressed in order to achieve the best possible speedup. Furthermore, with the introduction of the revolutionary Kepler-based Tesla K20 series GPUs (Compute Capability: 3.0), it is now possible to launch new kernel functions inside a kernel function, a mechanism referred to as dynamic parallelism [6], which we believe will open a whole new horizon for parallel algorithm research.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (No. 60905030) and NVIDIA Academic Partnership Program.

REFERENCES

[1] K. A. De Jong, Evolutionary Computation: A Unified Approach. MIT Press, 2006.
[2] D. B. Fogel, Evolutionary Computation: Toward a New Philosophy of Machine Intelligence. John Wiley & Sons, 2006
[3] R. G. Reynolds, "An introduction to cultural algorithms," In Proceeding of the Third Annual Conference on Evolutionary Programming, 1994, pp. 131-139.
[4] CUDA. Available: https://developer.nvidia.com/cuda-toolkit
[5] NVIDIA. Available: http://www.nvidia.com/
[6] NVIDIA Tesla K20-K20X GPU Accelerators Benchmarks. Available: http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf
[7] D. B. Krik and W. M. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.
[8] TOP500 Supercomputer Sites. Available: http://www.top500.org/
[9] X. Jin and R. G. Reynolds, "Using knowledge-based evolutionary computation to solve nonlinear constraint optimization problems: a cultural algorithm approach," In Proceeding of the 1999 Congress on Evolutionary Computation, 1999, pp. 1672-1678.
[10] J. G. Digalakis and K. G. Margaritis, "A multipopulation cultural algorithm for the electrical generator scheduling problem," Mathematics and Computers in Simulation, vol. 60(3-5), pp. 293-301, 2002.
[11] Y. Guo, Y. Cao and H. Wang, "Knowledge migration based multi-population cultural algorithm," In Proceedings of the Fifth International Conference on Natural Computation, 2009, pp. 331-335.
[12] B. Franklin and M. Bergerman, "Cultural algorithms: concepts and experiments," In Proceedings of the 2000 Congress on Evolutionary Computation, 2000, pp. 1245 - 1251.
[13] C. J. Lin, C. H. Chen, Y. C. Liu and C. T. Lin, "A hybrid of cooperative particle swarm optimization and cultural algorithm for neural fuzzy networks and its prediction applications," IEEE Transactions on Systems, Man, and Cybernetics, Part C, vol. 39(1), pp. 55-68, 2009.
[14] X, Jin and R. G. Reynolds, "Data mining using cultural algorithms and regional schemata," In Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, 2002, pp. 33-40.
[15] N. Rychtyckyj, D. Ostrowski, G. Schleis and R. G. Reynolds, "Using cultural algorithms in industry," In Proceedings of the 2003 Swarm Intelligence Symposium, 2003, pp. 187-192.
[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer and K. Skadron, "A performance study of general purpose applications on graphics processors," Journal of Parallel and Distributed Computing, vol. 68(10), pp. 1370-1380, 2008.
[17] Y. Zhou and Y. Tan, "GPU-based parallel particle swarm optimization," In Proceedings of the 2009 IEEE Congress on Evolutionary Computation, 2009, pp. 1493-1500.
[18] R, Cabido, A. S. Montemayor and J. J. Pantrigo, "High performance memetic algorithm particle filter for multiple object tracking on modern GPUs," Soft Computing, vol. 16(2), pp. 217-230, 2012.
[19] CUDA Toolkit 5.0 Document - CURAND Guide. Available: http://docs.nvidia.com/cuda/curand/index.html
[20] KDD CUP 1999 Dataset. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html