CUDA-Based Genetic Algorithm on Traveling Salesman Problem

Chapter · January 1970

DOI: 10.1007/978-3-642-21378-6_19

4 authors:

Su Chen

8 PUBLICATIONS **117** CITATIONS

SEE PROFILE

Spencer Davis

Arkansas State University - Jonesboro

8 PUBLICATIONS **60** CITATIONS

SEE PROFILE

Hai Jiang

Arkansas State University - Jonesboro

192 PUBLICATIONS **4,800** CITATIONS

SEE PROFILE

Andy Novobilski

University of North Georgia

15 PUBLICATIONS **76** CITATIONS

SEE PROFILE

CUDA-based Genetic Algorithm on Traveling Salesman Problem

Su Chen, Spencer Davis, Hai Jiang, Andy Novobilski
Department of Computer Science
Arkansas State University
{su.chen, spencer.davis}@smail.astate.edu, {hjiang, anovobilski}@astate.edu

***Abstract*—Genetic algorithm is a widely used tool for generating searching solutions in NP-hard problems. The genetic algorithm on a particular problem should be specifically designed for parallelization and its performance gain might vary according to the parallelism hidden within the algorithm. NVIDIA GPUs that support the CUDA programming paradigm provide many processing units and a shared address space to ease the parallelization process. A heuristic genetic algorithm on the traveling salesman problem is specially designed to run on CPU. Then a corresponding CUDA program is developed for performance comparison. The experimental results indicate that a sequential genetic algorithm with intensive interactions can be accelerated by being translated into CUDA code for GPU execution.**

I. Introduction

Genetic algorithm (GA) and other stochastic searching algorithms are usually designed to solve NP-hard problems [3]. The traveling salesman problem (TSP) is a famous NP-hard problem [5][10]. It aims to get the shortest wrap-around tour path for a group of cities. Since NP-hard problems cannot be solved in acceptable time, people aim to find acceptable solutions in acceptable time instead. To achieve this, various heuristic algorithms, such as the genetic algorithm, ant algorithm, tabu search, neural network, etc., are designed.

Genetic algorithm was inspired by the evolvement of chromosomes in the real world, which includes crossover, mutation, and natural selection. Viewing chromosomes as solutions to a TSP problem, crossover and mutation are changing phases for chromosomes, while natural selection is a sifting phase that will wash out the worst solutions so that better ones will stay.

To simulate this process in a computer program, programmers have to design sequences of numbers to represent chromosomes and perform certain operation on them. Different Problem will have different types of chromosome designs. For example, select participants from a group will have a design of $a_1a_2a_3...a_n$ as its chromosome, where a_i are either 0 or 1, where 0 means unselected and 1 means selected.

As the problem size increases, it takes a very long time to reach an optimum solution, or even a less-optimum but satisfying solution. In order to shorten the convergence time, artificial intelligence is usually introduced to make algorithms efficient. For the traveling salesman problem, 2-opt is a specifically designed mutation operator which takes longer time than ordinary operators, but guarantees fast and steady convergence. However, even with this efficient operator, computing time is still quite long when problem size is large. Recently, NVIDIA's CUDA programming paradigm enables GPU as a new computing platform [1][2]. Many-core GPUs can explore parallelism inside Genetic Algorithms for execution speedup and provide a cost effective method of implementing SIMD type solutions.

This paper intends to develop a heuristic genetic algorithm on TSP and then parallelize it with CUDA on GPUs for performance gains. The rest of the paper is organized as follows: Section II discuss the deployment of genetic algorithm on TSP problem. Section III addresses the issues of genetic algorithm implementation on GPUs with CUDA. Section IV provides performance analyses on both CPU and GPU. Section V gives the related work. Finally, our conclusions and future work are described.

II. Genetic Algorithm on TSP

GA's input usually includes a waypoint number and a distance table. The Output of GA should be an optimized chromosome chain that represents the order of cities that the traveling salesman should follow. The general process of GA is given in Fig. 1.

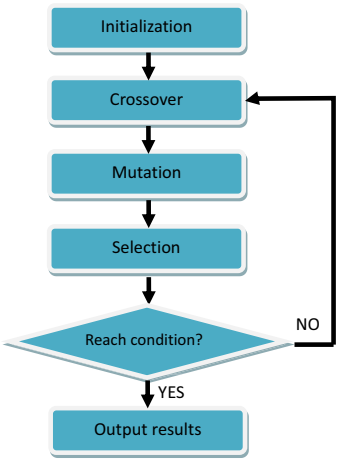

Figure 1. Flowchart of Genetic Algorithm

The Initialization phase generates a group of chromosomes as shown in Fig. 2. The group size can influence quality of the final result and running time. Therefore, it needs to be properly chosen. Generally, when the group size increases, results are potentially better whereas the running time increases. Factors for both good results and a reasonable running time should be considered.

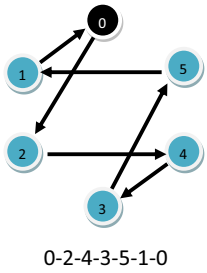

Figure 2. Chromosome sequence in GA

Crossover phase is an important part in GA to simulate the action where two chromosome individuals exchange partial sections of their bodies. This process helps increase diversity as well as exchange better genes within population. Crossover on real chromosomes is illustrated in Fig. 3. Unfortunately, in TSP, there are no two same numbers in one chain. Therefore, it is impossible to do crossover directly, as chromosomes do in the real world. However, there are alternative ways to simulate this process. The strategy used by this paper is based on sequence orders not values, as shown in Fig. 4 where the crossover of the selected portion of chromosomes is reasonably done.

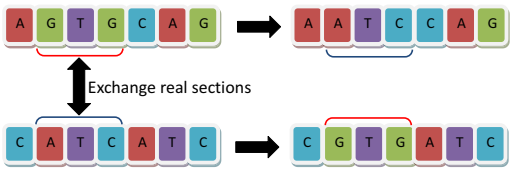

Figure 3. Crossover in the real world

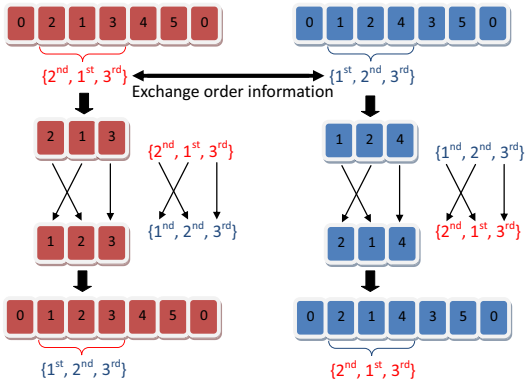

Figure 4. Crossover in the GA on TSP

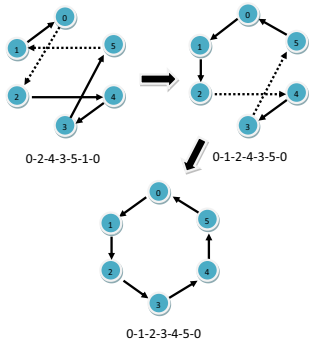

Figure 5. Mutation in the GA

Since good chromosomes are forced to stay in population and pass down their heritage information by crossover, after generations, chromosomes will assimilate each other. The mutation phase is designed to make unpredictable changes on chromosomes in order to maintain the variety of the population. Mutation operators can be arbitrarily designed

but the effects taken by them will be hard to tell. Some mutation operators will slow down the convergence process, while others will accelerate it. In this paper, we select 2-opt as the mutation operator, which can make the algorithm converge much faster than ordinary GA. The 2-opt mutation operator is specifically designed to solve TSP and guarantees both diversity and steady evolvement [5][10]. However, this operator takes $O(n)$ time, and has larger time cost than that of simple operators. Details in 2-opt is given in Fig. 5.

Selection phase is usually placed after crossover and mutation. In this paper, simplest selection method is adopted and only better solutions are accepted. Each new chromosome will be compared with the older one and the better of the chromosomes stay in the population.

A termination condition should be set to stop the evolution process. In this paper, when the best result has stopped evolving for some generations, algorithm will stop and output the result. Though better solutions are expected when execution time becomes longer, after solutions are convergent, the probability for GA to update the best result becomes extremely small.

III. Genetic Algorithm Implementation with CUDA

A. CUDA Platform and GPU Architecture

CUDA (Compute Unified Device Architecture), developed by NVIDIA, is a parallel programming paradigm [1][2]. While graphics cards were originally designed only to process image and video flows, CUDA provides a platform to solve any general purposed problem on GPU. Rather than fetching image pixels concurrently, now threads in the GPU can run common tasks in parallel; however, as in other parallel programming platforms, task dependency problems should be considered by programmers themselves.

Besides hundreds of threads, Fermi (The latest GPU Architecture in 2010) provides shared memory that can be accessed by threads within the same block extremely fast. Shared memory can be thought of as a cache that can be directly manipulated by users. When the input of the problem is small and all its intermediate results can be loaded into shared memory, Fermi will do excellent job. On the other hand, if the input size is relatively large, the utilization of shared memory should be carefully considered.

Limitations of CUDA cannot be ignored. Recursion and pointers for functions are still not supported and debugging is a tedious job. The bus latency between the CPU and GPU exhibits as a bottleneck. All of these limitations should be avoided or considered during programming, and CUDA's architecture should be taken advantage of in their code.

B. Opportunities for GA with CUDA

Usually, in order to guarantee the diversity of species, GA maintains a group, or population, consisting of a good number of chromosomes. This can be thought of as the desire of the problem solver to create more directions in order to search a bigger area. It can be understood as that if more ants are dispatched to different directions, chance to find food becomes greater. In GA, these ants communicate with each other frequently and change their searching directions based on the information they get. Though it contains many interactions and dependencies, it is possible to be parallelized for performance gains.

The most reasonable way to parallel this process is to map activities of chromosome individuals to separated threads. Since all chromosomes will do the same job, they roughly finish at the same time. This property prevents cores from being idle. Otherwise, synchronization will drag down performance severely.

CUDA platform and Fermi architecture provide good tools to parallelize the algorithm. First, GPU supplies hundreds of cores for executing threads in parallel. Second, threads can talk to each other easily and fast because they share address space in several levels such as shared memory and global memory levels. Third, as an extension of C, CUDA eases the programming task.

C. Random Number Generation in CUDA

Since GA is a stochastic searching algorithm, a random numbers generation strategy is required. Unfortunately, CUDA does not provide one yet. However, a pseudo random number generator can be easily simulated in different ways. In this paper, bit shifting, multiplication and module operators are used to generate random numbers.

CUDA programs on GPUs may slow down when threads compete for random seeds. To solve this problem, random

seeds are generated in the CPU and assigned to each GPU thread. Equipped with a simple random number generator function, threads in the GPU can generate random numbers simultaneously without blocking or false sharing.

D. Data Management for GA

In CUDA architecture, threads can be arranged in blocks and grids to fit applications. In the latest Fermi architecture, cache and shared memory co-exist to enable GPU cores to behave as CPU. This provides a greater chance to get better performance. Shared memory is as fast as cache and can be directly manipulated by users. However, shared memory space can only be accessed by threads within one block.

If shared memory is big enough for everything, programmers do not have to spend too much time on data manipulation. However, with limited shared memory size, only few frequently used variables and arrays have priorities to reside in it. For the GA on TSP problem, the distance table and chromosome group occupy the majority of the space and both are too large for shared memory. Since distance between two cities is Euclidean distance in this paper, the distance table can be discarded and coordinate arrays are used instead. This change will definitely harm CPU's performance because of duplicated calculations. However in GPU, such computation redundance is encouraged since there always are many idle cores due to memory access latency. This design is proved to be valid by experimental results.

E. Parallelization of GA

Because each thread has an independent seed for random number generation, different threads can initialize chromosomes simultaneously. Since mutation of one chromosome has nothing to do with other chromosomes in this paper, there are no task dependencies between any two threads in these phases.

For the crossover part however, threads tend to find a peer to exchange information. Since threads are working on this part together, it is not possible for them to work on original chromosomes directly. Copies have to be made before crossover phase starts. However, these copies still cannot be changed directly because it is possible that two chromosomes choose the same target to communicate with. Since this operation not only reads, but also changes data, working directly on the copies is still not allowed. Therefore, each thread should make another temporary copy for the target chromosome to work on.

After crossover phase, the copy for previous group will be used as the group of the last generation. After mutation phase, selection phase needs to compare current chromosome and previous version and decide which one is better, and therefore stays. This process can be directly parallelized since no communication and task dependency exist among threads.

Updating the best chromosome needs to search for the minimum one in the adaptive value array for the new chromosome group. This can be implemented in complexity of $O(logn)$ using n processors instead of $O(n)$ done sequentially. The existence of shared memory and cache can reduce this time to an insignificant level, even doing it sequentially in one thread.

F. Synchronization in CUDA

Based on the task dependency analysis, necessary synchronization points have been detected and inserted as in Fig. 6. Synchronization needs to be addressed at four positions:

1. The copy phase should wait till best value is updated.

2. The crossover phase should wait till copy phase finishes.

3. Updating the best chromosome should wait until the selection phase finishes.

4. On the CPU side, the programmer should place a CUDA synchronization call to wait till all threads are idle, and then output the result. If this is not done, the result will be wrong since CPU does not know what is going inside the GPU.

Also, from crossover phase to selection phase, synchronization is not necessary due to the introduction of chromosome copy and algorithm design.

IV. Experimental Results and Discussion

Both sequential and parallel programs were tested on a machine with two Intel Xeon E5504 Quad-Core CPUs (2.00GHz, 4MB cache) and two NVIDIA Tesla 20-Series C2050 GPUs.

Tests have been carefully made to determine how many chromosomes should be generated as a group and how large the termination generations should be. It turns out that we can get better solutions by setting 200 as the chromosome number and 1000 as the termination generation. Values that are larger than these two numbers do not provide further significant improvment to our solutions but increase the running time in a linear speed.

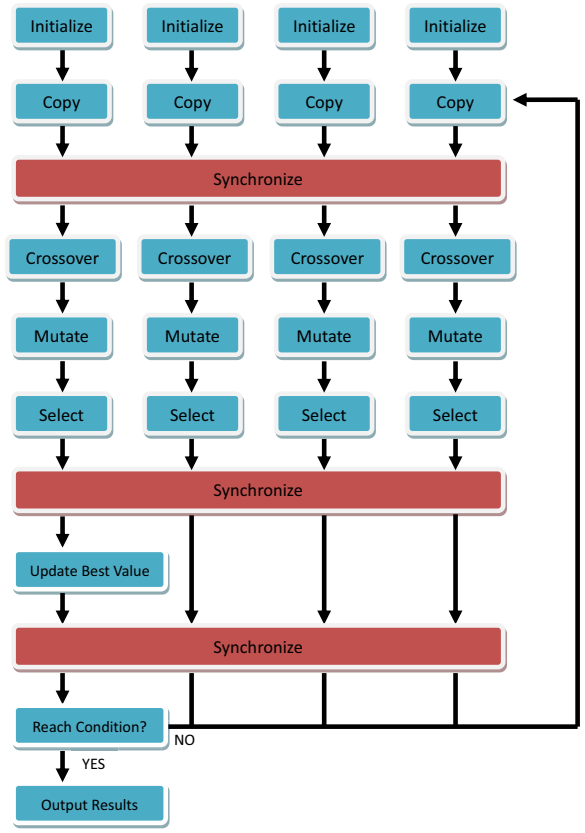

Figure 6. Task dependency and synchronization points

In general, the test data can be classified into two types: randomized and clustered, as shown in Figs. 7 and 8, respectively. Both of them occur in real life. However, it is easier for people to tell if clustered cities are well routed than random data through their intuitive observations. Even in programs, the work load of clustered data is smaller than randomized data. When a good solution is found, it is harder to find a better one for clustered data than for randomized data since only few tiny specific changes can update present the best solution for a cluster, while many more possible changes exist for randomized data. This inherent property associated with these two types of data makes their potential work load different in this paper. When dealing with clustered data, the algorithm will find it hard to update best value after it approaches some sort of line. Hence, the program ends early. On the other hand, the best value tends to update more times for randomized data, which causes a longer average running time. Comparison results are illustrated in Fig. 9. Both GPU and CPU give positive results for the above hypothesis, that is, algorithm on clustered data terminates earlier than that on randomized data. Another fact is that, for both types of data, GPU beats CPU.

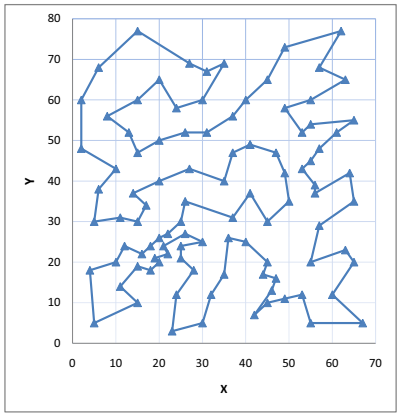

Figure 7. An example of randomized data

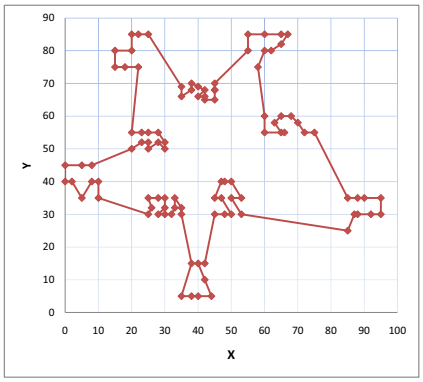

Figure 8. An example of clustered data

Questions may be raised about why GA in this paper only gets such insignificant speed up on GPU. As mentioned before, for the synchronization purpose, we only generate one block to run this program. Under CUDA architecture, threads in one block can only be served by one Streaming Multiprocessor (SM). However, in C2050, each GPU has 16 SMs and shared memory and cache are evenly assigned to each SM, which means we only used about 1/16 computing resource on one GPU, and the performance is still better than using CPU (single processor). In future work, we will try to expand the

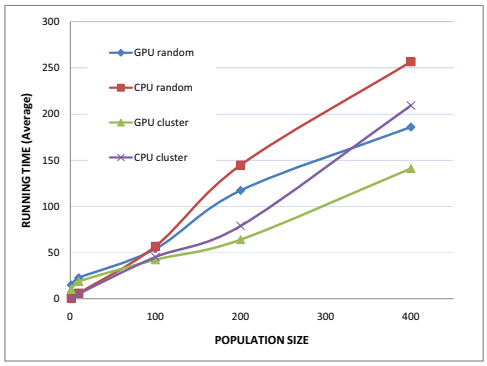

Figure 9. Performance comparison with randomized and clustered data

problem scale and keep the whole GPU or clusters busy, and the speed-up will increase significantly.

V. Related Work

Computer simulation of evolution started in 1950s with the work of Nils Aall Barricelli [3][4]. Since 1957, Alex Fraser has published a series of papers on simulation of artificial selection of organisms [7][8]. Based on this work, computer simulation of evolution became more popular in 1960s and 1970s. All essential elements of modern genetic algorithms were included in the book by Fraser and Burnell (1970) [9]. Goldberg (1989) first used genetic algorithm to solve the traveling salesman problem [10]. As a method for solving traveling salesman problems, 2-opt was raised by G. A. Croes (1958) in 1950s [5].

Muhlenbein (1989) brought up the concept of PGA (parallel genetic algorithm) [11], which aimed to implement GA on computer clusters. Ismail (2004) implemented PGA using MPI library [12]. In 2008, NVIDIA released latest CUDA SDK2.0 version, which bestowed CUDA much wider range of applications. Stefano et al. (2009) presented a paper about implementing a simple GA with CUDA architecture, where sequential code for same algorithm was taken for comparison [6]. Another paper from Petr Pospicha and Jiri (2009) presented a new PGA and implemented it on CUDA [13]. However, performance comparison in Stefano's work was not based on the same algorithm. In 2010, NVIDIA developed latest version of its GPU architecture, which is called Fermi, for Tesla M2050 and M2070 [2] and corresponding programming guide under these architectures was released [1].

VI. Conclusions and Future Work

Compared to Stefano's work in 2009 [6], this paper presents a more complex but parallelizable Genetic Algorithm (not specifically designed for certain GPU architecture) to solve TSP problem. Corresponding sequential C code for the same algorithm is carefully written for the performance comparison. Experimental results show the CUDA program with new Fermi architecture achieves some performance gains, although not so significant. However, considering the massive random memory accesses brought in by this much more complex algorithm and its relatively shorter execution time, this insignificant acceleration indicates that the current GPU architecture may have great potentials in speeding up the existing simulations of group evolution. More advanced performance tuning techniques such as asynchronous communication and zero copy will be applied for further performance gains in the future.

References

[1] Nvidia cuda c programming guide 3.1, 2009.
[2] Nvidia fermi tuning guide, 2009.
[3] Nils Aall Barricelli. Esempi numerici di processi di evoluzione. *Methodos*, pages 45–68, 1954.
[4] Nils Aall Barricelli. Symbiogenetic evolution processes realized by artificial methods. *Methodos*, 9:143–182, 1957.
[5] G. A. Croes. A method for solving travling salesman problems. *Operations Res.*, 6(1):791–812, 1958.
[6] Stefano Debattisti. Implementation of a simple genetic algorithm within the cuda architecture. *The Genetic and Evolutionary Computation Conference*, 2009.
[7] Alex Fraser. Simulation of genetic systems by automatic digital computers. *Australian Journal of Biological Science*, 10:484–499, 1957.
[8] Alex Fraser and Donald Burnell. Computer models in genetics. *Computers and Security*, 13:69–78, 1970.
[9] Alex Fraser and Donald Burnell. *Computer Models in Genetics*. New York: McGraw-Hill, 1970.
[10] David E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Lkuwer Academic Publishers, 1989.
[11] Muhlenbein H. Parallel genetic algorithm, population dynamic and combinational optimization. *Proc. 3rd, International Conference on Genetic Algorithms*, 1989.
[12] M. A. Ismail. *Parallel genetic algorithms (PGAs): master slave paradigm approach using MPI*. E-Tech, 2004.
[13] Petr Pospichal and Jiri Jaros. Gpu-based acceleration of the genetic algorithm. *Genetic and Evolutionary Computation Conference*, 2009.