2011 10th International Symposium on Parallel and Distributed Computing

Solving NP-Complete Problems on the CUDA Architecture using Genetic Algorithms

Mihai Calin Feier
Technical University of Cluj-Napoca
mihaif@student.utcluj.ro

Camelia Lemnaru
Technical University of Cluj-Napoca
Camelia.Lemnaru@cs.utcluj.ro

Rodica Potolea
Technical University of Cluj-Napoca
Rodica.Potolea@cs.utcluj.ro

***Abstract*—This paper focuses on solutions to two NP-Complete problems: k-SAT and the knapsack problem. We propose a new parallel genetic algorithm strategy on the CUDA architecture, and perform experiments to compare it with the sequential versions. We show how these problems can benefit from the GPU solutions, leading to significant improvements in speedup while keeping the quality of the solution. The best performance obtained in terms of speedup is 67 times. The solution presented in this paper suggests a general strategy for finding fast and robust solutions to complex problems.**

I. Introduction

In the early 1970s, a group of (seemingly) intractable (or computationally demanding) problems, such as 3-SAT or sub-graph isomorphism, emerged as a new category in computational complexity, the NP-Complete problems [2]. Since the formalization of this new concept by Richard Karp in 1972, a series of problems in many research areas have been shown to be NP-Complete. All known algorithms for NP-Complete problems require super-polynomial time, and it is yet to be proven whether faster algorithms can be developed. In practice, however, there exist several techniques – employed generally in computational problems, which can produce substantially faster algorithms: approximation, randomization, restriction, parametrization and heuristic approaches.

This paper presents both sequential and parallel evolutionary algorithms, based on the approximation technique, for two well-known NP-Complete problems: 3-SAT and the knapsack problem. The parallel versions of the algorithms have been implemented on GPU using CUDA [5].

We aim to show that genetic mechanisms can be employed to provide good solutions for NP-complete problems. Also, according to the GPGPU (*General-Purpose computation on Graphics Processing Units*) paradigm, video cards with massive parallelism capabilities, easily available to virtually anyone nowadays, can be used to speed up various computationally intensive tasks. Since NP-Complete problems fit perfectly in this category, we study how CUDA and the GPU capabilities can be exploited to deploy fast and effective parallel implementations for NP-Complete problems.

Other significant parallel genetic algorithm implementations on CUDA can be found in [3,6,7]. They deal with the mapping of a parallel island-based genetic algorithm, with and without migration. They show that this approach leads to significant improvements over the sequential implementation, up to a few thousand times faster running time in the best configuration, without taking into account the driver and data transfers overhead. This ensures that the parallel approach to genetic algorithms is a promising one if the hardware resources are properly used.

II. Theoretical Background

This section reviews the most important theoretical aspects related to genetic algorithms – basic form, combination and population techniques – and presents a short overview of NP-Completeness [4].

A. *NP-Complete Problems*

The NP-Complete problem set is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time, but for which there is no known efficient way to locate a solution.

Instances of NP-complete problems are ubiquitous in everyday life, whether it is the TSP employed to model transportation systems, or the knapsack problem utilized in resource allocation matters. Even though these problems are theoretically intractable, there exist a series of techniques employed generally in computational problems which can produce substantially faster algorithms. Approximation techniques focus on providing an "almost optimal" solution. Randomization allows for a certain amount of randomness to achieve a faster running time at the cost of the algorithm failing with a small probability. Restriction methods impose certain restrictions on the input and parametrization sets certain parameters of the input to fix values. Last, but not least, heuristic algorithms are not proven to be neither fast

978-0-7695-4540-0/11 $26.00 © 2011 IEEE
DOI 10.1109/ISPDC.2011.50

nor to produce a good result in all cases, but they are known to work reasonably well in most cases.

In this paper we focus on the SAT problem and the knapsack problem, in a comparison of a sequential implementation on the CPU versus a parallel implementation on the GPU.

B. *Genetic Algorithms*

The concept of a genetic algorithm (from now on referred to as GA) refers to a heuristic search process that mimics the process of natural evolution in order to find approximate solutions for a specified problem. GAs are contained in the larger class of evolutionary algorithms. They operate with techniques such as selection, crossover and mutation (over a number of generations) in order to generate candidate solutions, called individuals. GAs are generally employed in complex optimization and search problems – usually NP.

The *islands* approach is important for GAs, allowing several populations to evolve independently; small fractions from an island's population can migrate to another island, systematically. This approach may lead to better results due to the evolution of (semi)-independent (sub)-populations that approach the solution from different directions. However, this comes at the price of higher computation time, justified by a larger population. The option of having multiple populations evolving independently may increase the chances of closing in to the actual solution, even though some of the subpopulations (i.e. some islands) get stuck at a local optimum.

III. Selected Problems Specifics

For the sequential version of the GA we have employed the GALib framework [10]. It is provided with several genetic algorithm implementations, the most important of them being the basic single population GA, and an island based GA. We have used both these approaches in comparison, and thus the island-based type has proven to give the best results. For this reason we have chosen this type of GA for the parallel approach as well.

A. *The SAT and Knapsack Problems*

The ***SAT problem*** is searching for an assignment for the variables in a boolean expression to satisfy it. The 3-SAT problem, a version to which all boolean expressions can be transformed in, provides an easier representation as a 3-CNF, but with a certain overhead given by the increased expression size [8]. This conjunctive form has a straightforward representation and it is easy to represent and evaluate. In the context of a genetic algorithm, the common fitness function has an intuitive implementation; counting the number of true clauses within the statement gives a correct measure of how close to the true statement we are positioning.

The ***knapsack problem*** has a simple representation in genetic algorithms. Each chromosome is a binary string of bits, with each value encoding whether object i has been considered as part of the solution (i belongs to current knapsack) or not. For the fitness function, the sum of the values for each object in the solution can be considered and scaled. There is a discussion on what happens when an individual that exceeds the maximum admissible weight is generated. This can be avoided at initialization or in the mutation process, if no such individuals can be generated, or by assigning small fitness scores to such individuals.

B. *Parallel solutions*

For the parallel versions, a CUDA based genetic algorithm has been implemented from grounds up. The nature of a genetic algorithm is very well suited for a massive parallel architecture like this, because each individual (encoding a candidate solution in the search space) can be independently computed and analyzed. In order to make better use of the CUDA architecture, we have selected the island based implementation for the genetic algorithm, to isolate the populations within a block and minimize inter-block communication [11]. This is desirable for both the algorithm itself, as subsets of individuals locally search for an optimum, and for the GPU implementation, as no inter-block synchronization is required, each island evolving in a separate block. The mapping on the architecture was done by mapping each population on a CUDA block, and each genome in a population on a thread.

It is desired to run the entire algorithm on the GPU in order to minimize the communication between the GPU and CPU. This communication is usually slow and will cause extra delays in the genetic algorithm that are not used for computation. Unfortunately, in an island-based GA there is migration to be done after a number of generations have evolved, and usually this is done after each generation. In order to do that on the CUDA architecture, where inter-block synchronization is a problem, this can only be done using communication with the CPU.

Because of the (possibly) large number and size of the chromosomes, they are not kept in the cache memory, which is only 16 KB in our case, or 48 KB in the newer devices. This also ensures the scalability of the solution for future developments. The shared memory is used as a fast user-managed cache memory for the data in the operations of the genetic algorithm, like sorting or scaling.

As a random number generator that is needed in generating the initial population and within the GA operators on the kernel code, we used the CURAND Library that comes with the NVidia SDK. This library

includes two generators, out of which we used the Pseudorandom sequence generator.

For the selection step in the parallel implementation we have chosen tournament selection [1], which can have a customized behavior by implementing some kind of scaling of the fitness scores.

IV. Experimental Results

This section presents the experiments we have conducted for both the sequential and the parallel implementation. We have studied, through extensive experiments, how certain implementation decisions affect the efficiency of the algorithms.

The tests shown for the SAT problem are from the the DIMACS Benchmark Instances, namely some of those described in [9], which are satisfiable. These are, as described, random problems with hard generator, no single clauses and hard density. Each test is formed of roughly 800 CNF statements of 100 distinct variables. We used 15 different tests, each being run an average of 5 times, and took the average of those results for each configuration tested, in order to minimize the effects of randomness and have a good approximation of how good the algorithm works.

For the knapsack problem, the tests were 100 different knapsack instances having the number of objects set to 40, with weights and values being integers less than 1000.

A. *Testing parameters*

We have considered the following default configuration for the experiments: a single point crossover with a probability of 90%, and the mutation probability of 0.01. The chromosome representation is a binary string that is well suited for our experiments. The default mutation operation is random bit swap, but the fitness function has been specifically defined for each our specific problem: counting the number of true clauses within the statement for SAT and the summation of the values for each object in the solution for knapsack. By default, a linear scaling of the fitness scores is applied. Elitism is enabled, but set to one single individual per each generation in the default setting. The other parameters, like the population size and number of islands were varied in order to compare the running time performance obtained.

B. *Parallel version*

The tests were run on a computer with a Q6600 quad core processor running at 2.4 GHz and 2 GB RAM, and the video card is a NVidia GeForce GTX 260 with 216 processing cores. All tests were performed on the described SAT and knapsack instances, both on the CPU and the GPU. The knapsack problem is also interesting because it has a very lightweight fitness function, so it can be employed to benchmark the GA implementation.

In table 1 we show the comparison between the CPU and the GPU implementations, ran in the same configurations and the same test data. The quality of the solutions obtained by both versions is roughly the same, the only difference only coming from the random number generators employed and the elitism implementation.

Table 1 – Time performance of the sequential and parallel versions, with different GA settings

Islands	1		16		32		128	
Pop size	32	128	32	128	32	128	32	128
	SAT problem							
$T_{seq(ms)}$	125	143	1255	1717	2548	3091	10240	12117
$T_{par(ms)}$	267	360	298	387	317	421	340	1014
Speedup	0.47	0.39	4.21	4.43	8.04	7.34	**30.12**	11.95
	Knapsack problem							
$T_{seq(ms)}$	17	25	131	281	255	551	1008	2237
$T_{par(ms)}$	10	17	11	19	12	23	15	65
Speedup	1.7	1.47	11.9	14.8	21.3	23.9	**67.2**	34.4

We compare different configurations of number of islands and population size. It shows the difference in running time between the sequential version (T_{seq}) and the parallel one (T_{par}), for both problems. One can observe a good speedup obtained in almost all cases (time including the data transfers and the synchronization between CPU and GPU).

It can be observed that the GPU implementation is only efficient when the CUDA architecture is properly used, i.e. using enough blocks in order to hide the memory latency by alternating execution between blocks, thus providing better parallelism.

In Figure 1, we can observe how the parallel implementation behaves in terms of speed compared to the sequential version, for the knapsack problem. We kept the population count per island fixed at 64 individuals, and varied the number of islands used.

As shown in Figure 2, the quality of the solution in the case of the knapsack problem is not affected, in this case even obtaining better convergence to the solution.

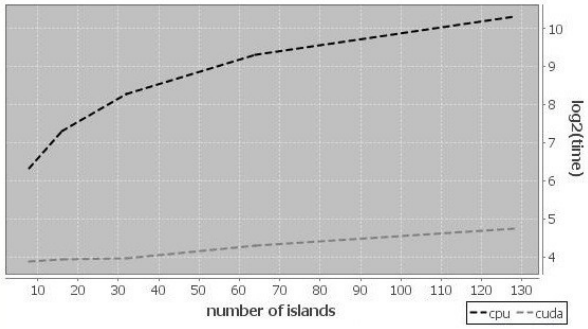

Fig. 1 – Time performance for the knapsack problem

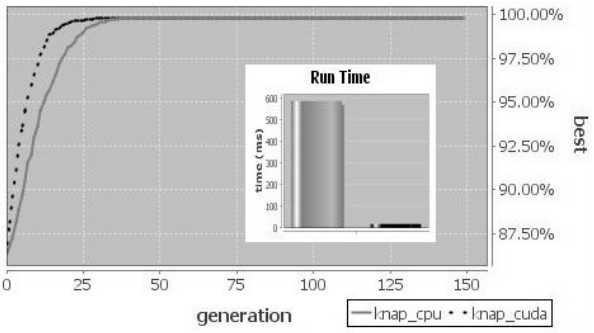

Fig. 2 – Quality of solution for the knapsack problem

V. Conclusions

This paper presents parallel, genetic-based solutions for two NP-Complete problems: k-SAT and the knapsack problem. They have been implemented on the GPU, using CUDA and the GPGPU paradigm.

The results of the extensive experiments performed validate the idea of generating solutions to NP-Complete problems with genetic algorithms; we have shown that there is a feasible solution of running the GAs on the GPU, and we have comparatively evaluated the results of the sequential and parallel implementations. In terms of speedup, our results range between 0.39 and 67.2 factor of improvement, while improving the performance criteria for specific parameter settings.

Our current interest in the evolution of our research considers several directions: (1) identifying settings for the parameters of the GA on various NP-Complete problems, and cluster problems by type of generic solution; (2) defining a set of generic constraints for a GA solution on a problem so that it could be efficiently implemented on the GPU; (3) defining a set of best practices for the implementation of GAs on the GPU.

References

[1] T. Blickle, L. Thiele, A mathematical analysis of tournament selection, *Proc. of the Sixth ICGA*, Morgan Kaufmann Publishers, San Francisco, Ca., 1995, pp. 9-16.

[2] Cook, S.A. (1971). "The complexity of theorem proving procedures". Proceedings, Third Annual ACM Symposium on the Theory of Computing, ACM, New York. pp. 151–158. doi:10.1145/800157.805047

[3] S. Debattisti, N. Marlat, L. Mussi, S. Cagnoni, Implementation of a Simple Genetic Algorithm within the CUDA Architecture, Università degli Studi di Parma, Italy

[4] M.R. Garey, D.S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, W.H.Freemanand Company, New York, 1979.

[5] Nvidia Corp., *Nvidia CUDA Programming Guide*, version 3.2, January 2011

[6] Pospichal, P., Jaros, J., Schwarz, J., Parallel Genetic Algorithm on the CUDA Architecture, *Applications of Evolutionary Computation*, Springer, 2010, pp. 442-451

[7] P. Pospichal, Jo. Schwarz, J. Jaros, Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU, *16th International Conference on Soft Computing MENDEL* 2010, pp. 64-70

[8] E. Rodriguez-Tello, J. Torres-Jimenez, ERA: An Algorithm for Reducing the Epistasis of SAT Problems, *Genetic and Evolutionary Computation-GECCO 2003,* pp. 1283-1294

[9] B. Selman, D.G. Mitchell, H.J. Levesque, Generating Hard Satisfiability Instances, *Artificial Intelligence*, Vol. 81, pp. 17-29, 1996

[10] M. Wall, GAlib: A C++ Library of Genetic Algorithm Components, MIT, August 1996

[11] S. Xiao, W. Feng, *Inter-Block GPU Communication via Fast Barrier Synchronization*, Technical Report TR-09-19, Computer Science, Virginia Tech, 2009