Solving Classification Problems Using Genetic Programming Algorithms on GPUs

Alberto Cano, Amelia Zafra, and Sebastián Ventura

Department of Computing and Numerical Analysis, University of Córdoba
14071 Córdoba, Spain
{i52caroa,azafra,sventura}@uco.es

Abstract. Genetic Programming is very efficient in problem solving compared to other proposals but its performance is very slow when the size of the data increases. This paper proposes a model for multi-threaded Genetic Programming classification evaluation using a NVIDIA CUDA GPUs programming model to parallelize the evaluation phase and reduce computational time. Three different well-known Genetic Programming classification algorithms are evaluated using the parallel evaluation model proposed. Experimental results using UCI Machine Learning data sets compare the performance of the three classification algorithms in single and multithreaded Java, C and CUDA GPU code. Results show that our proposal is much more efficient.

1 Introduction

Evolutionary Algorithms (EA) are a good method, inspired by natural evolution, to find a reasonable solution for data mining and knowledge discovery [1], but they can be slow at converging and have complex and great dimensional problems. Their parallelization has been an object of intensive study. Concretely, we focus on the Genetic Programming (GP) paradigm.

GP has been parallelized in many ways to take advantage both of different types of parallel hardware and of different features in particular problem domains. Most parallel algorithms during the last two decades deal with implementation in clusters or Massively Parallel Processing architectures. More recently, the studies on parallelization focus on using graphic processing units (GPUs) which provide fast parallel hardware for a fraction of the cost of a traditional parallel system.

The purpose of this paper is to improve the efficiency of GP classification models in solving classification problems. Once it has been demonstrated that the evaluation phase is the one that requires the most computational time, the proposal is to parallelize this phase generically, to be used by different algorithms. An evaluator is designed using GPUs to speed up the performance, receiving a classifier and returning the confusion matrix of that classifier to a database. Three of the most popular GP algorithms are tested using the parallel evaluator proposed. Results greatly speed algorithm performance up to 1000 times with respect to a non-parallel version executed sequentially.

E.S. Corchado Rodriguez et al. (Eds.): HAIS 2010, Part II, LNAI 6077, pp. 17–26, 2010.

The remainder of this paper is organized as follows. Section 2 provides an overview of GPU architecture and related experiences with Evolutionary Computation in GPUs. Section 3 analyzes GP classification algorithms to find out which parts of the algorithms are more susceptible to performance improvement, and discusses their parallelization and implementation. Section 4 describes the experimental setup and presents computational results for the models benchmarked. Conclusions of our investigation and future research tasks are summarized in Section 5.

2 Overview

In this section, first, the CUDA programming model on GPU will be specified. Then, the more important previous studies of GPU on GP will be described.

2.1 CUDA Programming Model on GPU

The CUDA programming model [13] executes kernels as batches of parallel threads in a Single Instruction Multiple Data (SIMD) programming style. These kernels comprise thousands to millions of lightweight GPU threads for each kernel invocation.

CUDA threads are organized into a two-level hierarchy represented in Figure 1. At the higher one, all the threads in a data-parallel execution phase form a grid. Each call to a kernel initiates a grid composed of many thread groupings, called thread blocks. All the blocks in a grid have the same number of threads, with a maximum of 512. The maximum number of thread blocks is 65535 x 65535, so each device can run up to 65535 x 65535 x 512 = $2 \cdot 10^{12}$ threads.

To properly identify threads, each thread in a thread block has a unique ID in the form of a three-dimensional coordinate, and each block in a grid also has a unique two-dimensional coordinate.

Thread blocks are executed in streaming multiprocessors (SM). A SM can perform zero-overhead scheduling to interleave warps and hide the latency of long-latency arithmetic and memory operations.

There are four different main memory spaces: global, constant, shared and local. GPU's memories are specialized and have different access times, lifetimes and output limitations.

2.2 Related GP Works with GPUS

Several studies have been done on GPUs and Massively Parallel Processing architectures within the framework of evolutionary computation. Concretely, we can cite some studies on Genetic Programming on GPUs [7].

D. Chitty [4] describes the technique of general purpose computing using graphics cards and how to extend this technique to Genetic Programming. The

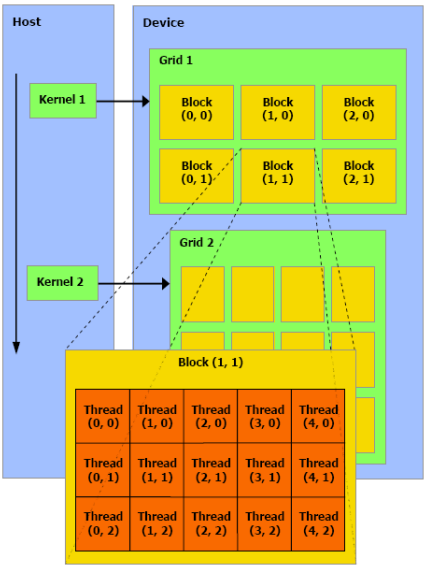

Fig. 1. CUDA threads and blocks

improvement in the performance of Genetic Programming on single processor architectures is also demonstrated. S. Harding [8] goes on to report on how exactly the evaluation of individuals on GP could be accelerated.

Previous research on GPUs have shown evaluation phase speedups for large training case sets like the one in our proposal. Furthermore, D. Robilliard [14] proposes a parallelization scheme to exploit the power of the GPU on small training sets. To optimize with a modest-sized training set, instead of sequentially evaluating the GP solutions parallelizing the training cases, the parallel capacity of the GPU are shared by the GP programs and data. Thus, different GP programs are evaluated in parallel, and a cluster of elementary processors are assigned to each of them to treat the training cases in parallel. A similar technique but using an implementation based on the Single Program Multiple Data (SPMD) model is proposed by W. Landgdon and A. Harrison [12]. The use of SPMD instead of Single Instruction Multiple Data (SIMD) affords the opportunity to achieve increased speedups since, for example, one cluster can interpret the *if* branch of a test while another cluster treats the *else* branch independently. On the other hand, performing the same computation inside a cluster is also possible, but the two branches are processed sequentially in order to respect the SIMD constraint: this is called divergence and is, of course, less efficient.

These reports have helped us to design and optimize our proposal to achieve maximum performance in data sets with different dimensions and population sizes.

3 Genetic Programming Algorithms on GPU

In this section the computation time of the different phases of the GP generational algorithm is evaluated to determine the most expensive part of algorithm. Then, it is specified how parallelize using GPU the evaluation phase of a GP algorithm.

3.1 GP Classification Algorithm

Genetic Programming, introduced by Koza [10], is a learning methodology belonging to the family of EA [17]. Among successful EA implementations, GP retains a significant position due to such valuable characteristics as: its flexible variable length solution representation, the fact that a priori knowledge is not needed about the statistical distribution of the data (data distribution free), data in their original form can be used to operate directly on them, unknown relationships that exist among data can be detected and expressed as a mathematical expression and finally, the most important discriminative features of a class can be discovered. These characteristics convert these algorithms into a paradigm of growing interest both for obtaining classification rules [2],[18], and for other tasks related to prediction, such as feature selection and the generation of discriminant functions.

The evolutionary process of Genetic Programming algorithms [6], similar to other EA paradigms, is represented in Figure 2 and consists of the following steps. The initial population is created randomly and evaluated. For each generation, the algorithm performs a selection of the best parents. This subset of individuals is recombined and mutated by applying different genetic operators,

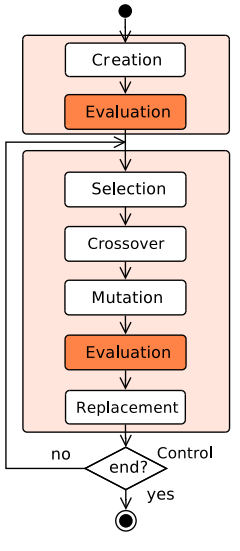

Fig. 2. GP Evolution Model

Table 1. CPU GP classification time test

Phase	Time (ms)	Percentage
Initialization	8647	8.96%
Creation	382	0.39%
Evaluation	8265	8.57%
Generation	87793	91.04%
Selection	11	0.01%
Crossover	13	0.01%
Mutation	26	0.03%
Evaluation	82282	85.32%
Replacement	26	0.03%
Control	5435	5.64%
Total	96440	100 %

thus obtaining offspring. These new individuals must now be evaluated using the fitness function. Different replacement strategies can be employed on parents and offspring so that the next generation population size remains constant. The algorithm performs a control stage in which it terminates if it finds acceptable solutions or the generation count reaches a limit. The consecutive processes of selection of parents, crossover, mutation, evaluation, replacement and control constitute a generation of the algorithm.

Experiments have been conducted to evaluate the computation time of the different phases of the generational algorithm. The experiment using GP algorithms represented in table 1 proves that around 94% of the time is taken up by the evaluation phase. This percentage is mainly linked to the population size and the number of patterns, increasing up to 98% in large problems. Thus, the most significant improvement is obtained by accelerating the evaluation phase, and this is what we do in our proposal. The most computationally expensive phase is evaluation since it involves the test of all individuals generated throughout all the patterns. Each individual's expression must be interpreted or translated into an executable format which is then evaluated for each training pattern. The result of the individual's evaluation throughout all the patterns is used to build the confusion matrix. The confusion matrix allows us to apply different quality indexes to get the individual's fitness.

The evaluation process of individuals within the population consists of two loops, where each individual iterates each pattern and checks if the rule covers that pattern. These two loops make the algorithm really slow when the population or pattern size increases.

3.2 Implementation on GPU

To parallelize the evaluation phase, the designed implementation is meant to be as generic as possible to be employed by different classification algorithms. The implementation receives a classifier and returns the confusion matrix that is used by most algorithms for the calculation of the fitness function.

To take advantage of GPU architecture [15], all individuals are evaluated throughout all the patterns simultaneously. An easy way to do that, is to create a grid of thread blocks sized as follows: one dimension is sized as the number of individuals and the other dimension is sized as the number of patterns in the data set. This organization means that one thread is the evaluation of one individual in one pattern. To achieve full performance, we have to maximize multiprocessor occupancy, so each block represents the evaluation of one individual in 128 patterns. This way, each thread within the block computes one single evaluation, then the size of the second dimension of the grid is the number of patterns divided by 128. This configuration allows up to 65536 individuals and 8.388.608 patterns per GPU and kernel call, large enough for all the data sets tested. Larger populations can be evaluated using several GPUs, up to 4 devices per host or iterating kernel calls.

Before running the evaluator on the GPU, the individuals's rules must be copied to the memory device using the PCI-E bus. Full performance can be

obtained by copying the rules into constant memory and pointing all the threads in a warp toward the same rule, resulting in a single memory instruction access. Constant memory provides a 64 KB cache written by the host and read by GPU threads.

The evaluation takes place in two steps. In the first kernel, each thread checks if the antecedent of the rule covers the pattern and the consequent matches the pattern class and stores a value, generally a hit or a miss. This implementation allows reusability for different classification models by changing the value stored depending on whether the antecedent does or does not cover the pattern, and the resulting matches or non-matches of the pattern class.

The kernel function must analyze the expression working with Polish notation, also known as prefix notation. Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets. While there remain tokens to be analyzed in the individual's expression, it checks what it has to do next by using a stack in order to store numerical values. Finally, we check the top value of the stack. If this value is true, that means the antecedent was true, so depending on the algorithm used, we compare this value to the known class given for the pattern.

The second kernel performs a reduction [5] and counts the results by subsets of 128, to get the total number of hits and misses. Using the total number of hits and misses, the proposed kernel performs the calculation of the confusion matrix on GPU, which could be used by any implementation of GP classification algorithms or even any genetic algorithm. This way, the kernel calculates the fitness of individuals in parallel using the confusion matrix and the quality metrics required by the classification model. Finally, results are copied back to the host memory and set to individuals for the next generation.

4 Experimental Results

Experiments carried out compare the performance of three different GP algorithms in single and multithreaded Java, C and CUDA GPU code. This section explains several experimentation details related with the data sets and the algorithms and then the speedup of the different implementations is compared.

4.1 Experimental Setup

This paper presents an implementation of a GPU GP evaluator for data classification using JCLEC [16]. JCLEC is a software system for Evolutionary Computation (EC) research, developed in Java programming language. It provides a high-level software environment for any kind of Evolutionary Algorithm, with support for Genetic Algorithms, Genetic Programming and Evolutionary Programming. We have selected two databases from the UCI repository for benchmarks, shuttle and poker hand inverse. The shuttle data set contains 9 attributes, 58000 instances and 7 classes. The poker hand inverse data set contains 11 attributes, 10^6 instances and 10 classes.

Experiments were run on a PC equipped with an Intel Core i7 processor running at 2.66GHz with two NVIDIA GeForce 285 GTX video cards equipped with 2GB of GDDR3 video RAM. No overclocking was done for any of the hardware.

Three different Grammar-Guided Genetic-Programming classification algorithms that were proposed in the literature are tested using our evaluator proposal. The evaluation concerning each algorithm is detailed for parallelization purposes.

Falco, Della and Tarantino [9] propose a method to achieve rule fitness by evaluating the antecedent throughout all the patterns within the data set. The adjustment function calculates the difference between the number of examples where the rule does or does not correctly predict the membership of the class and number of examples; if the opposite occurs, then the prediction is wrong. Specifically it measures the number of true positives t_p, false positives f_p, true negatives t_n and false negatives f_n. Finally fitness is expressed as:

$$fitness = I - ((t_p + t_n) - (f_p + f_n)) + \alpha * N \tag{1}$$

where I is the total number of examples from all training, α is a value between 0 and 1 and N is the number of nodes.

Tan, Tay, Lee and Heng [11] propose a fitness function that combines two indicators that are commonplace in the domain, namely sensitivity (Se) and specifity (Sp), defined as follows:

$$Se = \frac{t_p}{t_p + f_n} \qquad Sp = \frac{t_n}{f_p + t_n} \tag{2}$$

Thus, fitness is defined by the product of these two parameters.

$$fitness = Se * Sp \tag{3}$$

The proposal by Bojarczuk, Lopes and Freitas [3] presents a method in which each rule is simultaneously evaluated for all the classes in a pattern. The classifier is formed by taking the best individual for each class generated during the evolutionary process.

GP does not produce simple solutions. The comprehensibility of a rule is inversely proportional to its size. Therefore Bojarczuk defines the simplicity (Sy) and then the fitness of a rule:

$$Sy = \frac{maxnodes - 0.5 * numnodes - 0.5}{maxnodes - 1} \qquad fitness = Se * Sp * Sy \tag{4}$$

4.2 Comparing the Performance of GPU and Other Proposals

The results of the three GP classification algorithms are benchmarked using two UCI data sets that are shown in Tables 2 and 3. The rows represent the

Table 2. Shuttle generation speedup results

		Tan Model				Falco Model				Bojarczuk Model			
	Pop	100	200	400	800	100	200	400	800	100	200	400	800
Speed up	Java	1	1	1	1	1	1	1	1	1	1	1	1
	C1	2,8	3,1	3,2	2,9	5,4	8,1	5,2	5,0	18,8	12,5	11,2	9,5
	C2	5,5	6,1	6,3	5,7	10,6	15,9	10,5	10,1	35,9	24,6	22,1	18,1
	C4	10,1	11,5	12,5	10,7	19,7	30,3	20,5	19,8	65,2	47,3	40,1	33,7
	C8	11,1	12,4	13,4	10,3	19,9	30,1	21,2	20,6	65,7	46,8	40,5	34,6
	GPU	218	267	293	253	487	660	460	453	614	408	312	269
	GPUs	436	534	587	506	785	1187	899	867	1060	795	621	533

Table 3. Poker-I generation speedup results

		Tan Model				Falco Model				Bojarczuk Model			
	Pop	100	200	400	800	100	200	400	800	100	200	400	800
Speed up	Java	1	1	1	1	1	1	1	1	1	1	1	1
	C1	2,7	3,2	3,1	3,0	4,6	5,0	5,6	4,9	5,5	5,7	5,8	4,7
	C2	5,5	6,5	6,7	5,5	9,0	9,8	11,1	9,7	10,6	11,0	11,6	11,3
	C4	10,3	11,1	12,8	10,5	16,8	18,9	21,6	18,9	20,3	20,7	22,5	22,3
	C8	11,2	12,9	14,0	10,2	18,5	20,5	23,3	26,4	21,8	22,0	24,2	23,8
	GPU	155	174	234	221	688	623	648	611	142	147	148	142
	GPUs	288	336	500	439	1275	1200	1287	1197	267	297	288	283

speedup compared to Java execution time. Each column is labeled with the algorithm execution configuration from left to right: Population size, Java single CPU thread, C single CPU thread, C two CPU threads, C four CPU threads, C eight CPU threads, 1 GPU device, 2 GPU devices.

Benchmark results prove the ability of GPUs to solve GP evaluation. Intel i7 quadcore performs linear scalability from 1 to 2 and 4 threads, but not any further. After that point, GPUs perform much better. Their parallelized model allows the time spent on a classification problem to drop from one month to only one hour. Real classification training usually needs dozens of evaluations to get an accurate result, so the absolute time saved is actually a great deal of time. The greatest speedup is obtained with the Falco model which increases performance by a factor up to 1200 over the Java solution and up to 150 over the C single threaded CPU implementation. The speed up obtained is actually similar when considering different population sizes, for example a population of 200 individuals is selected to represent the figures. Fig. 3 and Fig. 4 display the speed up obtained by the different proposals with respect to the sequential Java version in the Shuttle and Poker data sets respectively. Note the progessive

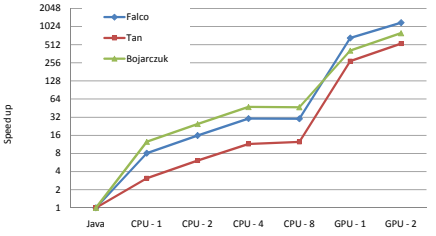

Fig. 3. Shuttle data set speed up

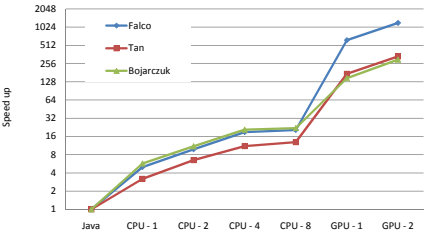

Fig. 4. Poker-I data set speed up

improvement obtained by threading C implementation and the significant increase that occurs when using GPUs.

5 Conclusions

Massive parallelization using the NVIDIA CUDA framework provides a hundred or thousand time speedup over Java and C implementation. GPUs are best for massive multithreaded tasks where each thread does its job but all of them collaborate in the execution of the program.

The CPU solution is lineal and complex. However, the GPU groups the threads into a block; then a grid of blocks is executed in SMs multiprocessors, one per SM. Thus, linearity is approximated by the number of 30-block grids. This implementation allows future scalability for GPUs with more processors. Next the NVIDIA GPU code-named Fermi doubles the number of cores available to 512 and performs up to 1.5 TFLOPs in single precision. It is noteworthy that i7 CPU scores are 2.5 times faster than 3.0 GHz PIV in our benchmarks.

Further work will implement the whole algorithm inside the GPU, so that selection, crossover, mutation, replacement and control phases will be parallelized, reducing data memory transfers between CPU and GPU devices.

Acknowledgments

The authors gratefully acknowledge the financial support provided by the Spanish department of Research under TIN2008-06681-C06-03, P08-TIC-3720 Projects and FEDER funds.

References

1. Freitas, A.A.: Data Mining and Knowledge Discovery with Evolutionary Algorithms. Springer, Heidelberg (2002)
2. Tsakonas, A.: A comparison of classification accuracy of four Genetic Programming-evolved intelligent structures. Information Sciences 176(6), 691–724 (2006)

3. Bojarczuk, C.C., Lopes, H.S., Freitas, A.A., Michalkiewicz, E.L.: A constrained-syntax Genetic Programming system for discovering classification rules: application to medical data sets. Artificial Intelligence in Medicine 30(1), 27–48 (2004)
4. Chitty, D.: A data parallel approach to Genetic Programming using programmable graphics hardware. In: GECCO 2007: Proceedings of the Conference on Genetic and Evolutionary Computing, pp. 1566–1573 (2007)
5. Kirk, D., Hwu, W.-m.W., Stratton, J.: Reductions and Their Implementation. University of Illinois, Urbana-Champaign (2009)
6. Deb, K.: A population-based algorithm-generator for real-parameter optimization. Soft Computing 9(4), 236–253 (2005)
7. Genetic Programming on General Purpose Graphics Processing Units, GP GP GPU, http://www.gpgpgpu.com
8. Harding, S., Banzhaf, W.: Fast Genetic Programming and artificial developmental systems on GPUS. In: HPCS 2007: Proceedings of the Conference on High Performance Computing and Simulation (2007)
9. De Falco, I., Della Cioppa, A., Tarantino, E.: Discovering interesting classification rules with Genetic Programming. Applied Soft Computing Journal 1(4), 257–269 (2002)
10. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
11. Tan, K.C., Tay, A., Lee, T.H., Heng, C.M.: Mining multiple comprehensible classification rules using Genetic Programming. In: CEC 2002: Proceedings of the Evolutionary Computation on 2002, pp. 1302–1307 (2002)
12. Langdon, W., Harrison, A.: GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Computing. A Fusion of Foundations, Methodologies and Applications 12(12), 1169–1183 (2008)
13. NVIDIA Programming and Best Practices Guide 2.3, NVIDIA CUDA Zone, http://www.nvidia.com/object/cuda_home.html
14. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines 10(4), 447–471 (2009)
15. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-m.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73–82 (2008)
16. Ventura, S., Romero, C., Zafra, A., Delgado, J.A., Hervás, C.: JCLEC: A Java framework for evolutionary computation. Soft Computing 12(4), 381–392 (2007)
17. Back, T., Fogel, D., Michalewicz, Z.: Handbook of Evolutionary Computation. Oxford University Press, Oxford (1997)
18. Lensberg, T., Eilifsen, A., McKee, T.E.: Bankruptcy theory development and classification via Genetic Programming. European Journal of Operational Research 169(2), 677–697 (2006)