Speeding Up Evolutionary Learning Algorithms using GPUs

Alberto Cano Amelia Zafra Sebastián Ventura

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

Abstract

This paper propose a multithreaded Genetic Programming classification evaluation model using NVIDIA CUDA GPUs to reduce the computational time due to the poor performance in large problems. Two different classification algorithms are benchmarked using UCI Machine Learning data sets. Experimental results compare the performance using single and multithreaded Java, C and GPU code and show the efficiency far better obtained by our proposal.

Keywords: GPUs, CUDA, Genetic Programming, Classification.

1 INTRODUCTION

Evolutionary Algorithms (EA) are good method to find a reasonable solution for data mining and knowledge discovery [1], but they can be slow at converging with complex, high dimensional problems. To solve this problem, different ways of parallelization have been studied. Specifically, if we focus on genetic programming (GP), we can find multiple ways to take advantage both of different types of parallel hardware and of different features of particular problem domains. Most of the parallel algorithms during the last two decades deal with the implementation over clusters or Massively Parallel Processing architectures (MPPs). More recently, the work about parallelization using graphics processing units (GPUs) [3], [5] and [10] provide fast parallel hardware for a fraction of the cost of a traditional parallel system.

The purpose of this paper is to analyze the scalability of NVIDIA GPUs using Compute Unified Device Architecture (CUDA) framework by the implementation of two Genetic Programming (GP) classification models and benchmark them using UCI datasets. Experimental studies compare our proposal with C and Java single and multithreaded code proving the viability of GPUs to speedup EAs solving very large problems.

The remainder of this paper is organized as follows. Section 2 provides an overview of the GP classification algorithms to find out which parts of the algorithms are more susceptible to performance improvement. Section 3 presents information on the GPU computing architecture. Section 4 discusses the parallelization analysis and implementation. Section 5 describes the experimental models benchmarked. Section 6 presents computational experiment results. Conclusions of our investigation and future research tasks are summarized in Section 7.

2 GP CLASSIFICATION ALGORITHMS

GP is a kind of mainstream Evolutionary Algorithms paradigm that presents more variants from EAs and its goal is to develop programs automatically. GP's individuals are expressions or syntax trees that represent the structure of a program that solves the problem, in our case, the correct classification of a set of instances.

There are different proposals using the paradigm of the GP to represent sets of expressions (decision trees, discriminant functions, classification rules, etc). Classification rules are widespread employed for GP and the formalism they use to represent the classifier is IF-THEN rules [8]. The antecedent of the rule (IF) consists of a combination of conditions on the attributes predictors. The consequent (THEN) contains the predicted value for the class. Thus, a rule assigns a pattern to the class if the values of predictor attributes satisfy the conditions assessed in the antecedent.

GP algorithms have the same structure as other EAs. Deb K. [4] propose a model where an EA starts with a set of individuals named algorithm's population. The initial population is generated randomly. For each iteration, the algorithm evaluates each individual using the fitness function. The algorithm terminates if it finds acceptable solutions or the generation count reaches a limit. Otherwise, it selects several individuals and copies them to replace individuals in the population that were not selected for reproduction so that the population size remains constant. Then, the algorithm manipulates individuals by applying different evolutionary operators such as crossover and mutation. This model initializes the population at first. The consecutive processes of selection, generation, replacement, update and control constitute an iteration of the algorithm.

The most computationally expensive step is generation since it involves the evaluation of all individuals generated. For each individual its expression must be interpreted or translated into an executable format and then it is evaluated for each training pattern. The result of an evaluation can be: true positive (t_p), false positive (f_p), true negative (t_n) or false negative (f_n). With these values the confusion matrix of the individual is constructed to get its quality index, its fitness.

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

The experience using GP algorithms proves that almost 98% of the time is taken by generation step. The execution time of the algorithm is mainly linked to the evaluation of individuals due to the population size and the number of patterns. Thus, the most significant improvement is obtained by accelerating the evaluation phase, and this is what we do in our proposal.

3 CUDA ARCHITECTURE

The CUDA programming model consists of a host that is a traditional CPU and one or more massively data-parallel coprocessing compute devices such as a GPU.

The runtime system 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 per each kernel invocation. To make the most of the hardware, creating enough threads is required; for example, the kernel might compute each array's element result in a separate thread.

CUDA's threads are organized into a two-level hierarchy, 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 three different main memory spaces: global, shared and local. GPU's memories are specialized and have different access times and output limitations.

Global memory is a large, long-latency memory that exists physically as an off-chip dynamic RAM. The threads can read and write global memory to share data and must write the kernel's output to be readable after the kernel terminates. However, a better way to share data and improve performance is to take advantage of shared memory.

Shared memory is a small, low-latency memory that exits physically as an on-chip registers and its content only maintained during thread block execution and are discarded when the thread block completes. Kernels that read or write a known range of global memory with spatial or temporal locality can employ shared memory as a software-managed cache. Such caching potentially reduces global-memory bandwidth demands and improves overall performance.

Each thread also has its own local memory space as registers, so the number of registers a thread uses determines the number of concurrent threads executed in the multiproccesor, that is called multiprocessor occupancy. To avoid wasting hundreds of cycles while a thread waits for a long-latency global-memory load or store to complete, a common technique is executing batches of global accesses, one per thread, exploiting the hardware's warp scheduling to overlap the threads's access latencies.

4 IMPLEMENTATION

To take advantage of the GPUs architecture we decided to evaluate all individuals over 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 represents that one thread is the evaluation of one individual over one pattern.

To achieve full performance, we have to maximize the multiprocessor ocupancy, so each block represents the evaluation of one individual over 512 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 512.

This configuration shown at figure 1 allows up to 65535 individuals and 33.553.920 patterns per GPU, large enough for all tested data sets. Larger populations can be evaluated using several GPUs in the system up to 4 devices and 8 GPU cores per host.

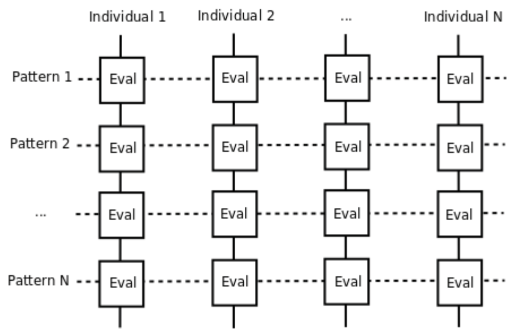

Figure 1: Parallel Evaluation Model.

Before running the evaluator on the GPU, the individuals's rules must be copied to the device memory using PCI-E bus. The CUDA programming model is an extension of the C language, so kernels are easy to write. We use Java Native Interface (JNI) to call Java natives methods and get the individuals's expression tree. JNI, Java's foreign function interface for executing native C code, is used to bridge the Java code with the kernels. In JNI, the programmer declares selected C functions as native external methods that can be invoked by a Java program. The native functions are assumed to have been separately compiled into host-specific binary code using nvcc compiler and also supports callback functions to enable native code to access Java objects.

In the following host code at figure 2, all variables preceded by *h_* are stored at host memory space and *d_* are stored at device memory space. A *base* pointer is used to know which subset of the population must be evaluated by each thread.

The evaluation takes place in two steps. In the first kernel at figure 3, threads check if the rule covers or not the pattern using a stack and stores a value for that thread depending on the classification algorithm. Generally, if the rules covers the pattern and the consequent matches the pattern's class or the rule does not cover the pattern and the consequent does not match the pattern's class we get a success, otherwise we get a fail.

The second kernel counts the results by subsets of 512, the actual maximum number of threads per block, to get the total number of t_p, f_p, t_n and f_n and build the confusion matrix. This way, the kernel calculates in parallel the fitness of individuals using the confusion matrix and the quality metrics described for each classification model. Finally, results are copied to host memory and set to individuals for the next generation.

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.

```
int threadPopulationSize = ceil(
    populationSize/numThreads);
int base = plan->device *
    threadPopulationSize;

for(int i=0; i<threadPopulationSize; i++)
  cudaMemcpy(d_exprTree[i], h_exprTree[
      base + i], sizeof(char)*strelen(
      h_exprTree[base + i]),
      cudaMemcpyHostToDevice);

dim3 threads_eval(1, 512);
dim3 grid_eval(threadPopulationSize, ceil(
    Matrix_H/512.0));

evaluate_kernel
  <<<grid_eval, threads_eval>>>
  (d_fitness, d_exprTree, Matrix_W
   , Matrix_H, classifiedClass);

cudaThreadSynchronize();

dim3 threads_cm(1, 512);
dim3 grid_cm(threadPopulationSize, 1);

confusion_matrix_kernel
  <<< grid_cm, threads_cm >>>
  (d_fitness, Matriz_H);

cudaMemcpy(h_fitness + base, d_fitness,
    threadPopulationSize*sizeof(float),
    cudaMemcpyDeviceToHost );
```

Figure 2: Host code, kernel invocation.

While there are remaining tokens, it checks what it has to do next. A stack is used to store numerical values. Finally, we check the last 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.

```
__global__ void evaluate_kernel()
{
int row = 512*blockIdx.y + threadIdx.y;
int position = blockIdx.x*Matrix_H + row;
int sp=0,bufp=length(expr[blockIdx.x]);

while(1)
{
switch(getop(s, &bufp, expr[blockIdx.x]))
    {
case NUMBER:
  push(atof(s), stack, &sp);
  break;
case VARIABLE:
  push(vars[Matrix_W*(row) + atoi(s)],
      stack, &sp);
  break;
case AND:
  if (pop(stack, &sp) * pop(stack, &sp))
              push(1, stack, &sp);
  else          push(0, stack, &sp);
  break;
case OR:
  if (pop(stack, &sp) || pop(stack, &sp))
             push(1, stack, &sp);
  else          push(0, stack, &sp);
  break;
case NOT:
  if(pop(stack, &sp) == 0)
              push(1, stack, &sp);
  else          push(0, stack, &sp);
  break;
case '>':
  op1 = pop(stack, &sp);
  if (op1 > pop(stack, &sp))
              push(1, stack, &sp);
  else          push(0, stack, &sp);
  break;
case '<':
  op1 = pop(stack, &sp);
  if (op1 < pop(stack, &sp))
              push(1, stack, &sp);
  else          push(0, stack, &sp);
  break;
...
case END:
  arg = pop(stack, &sp);
  if(arg == 1) { // The antecedent is true
  if(classifiedClass == knownClass[row])
    result[position] = 0;        // tp
  else
    result[position] = 2;        // fp
  }
  else {
  if(classifiedClass != knownClass[row])
    result[position] = 1;        // tn
  else
    result[position] = 3;        // fn
  }
  return;
default:  break;
}}}
```

Figure 3: GPU evaluation kernel.

This implementation allows scalability for different GPUs multiprocessors count. The more SM the device has, the faster it runs because of the parallelized blocks. Future devices will also take advantage of the programming model by the implementation of L1 core and L2 SM cache. The antecedent of the classification rule is the same for all the threads within a block so it can be cached in a single memory access.

5 EXPERIMENTATION

This paper presents an implementation of a GPU GP evaluator for data classification using JCLEC [9]. JCLEC is a software system for Evolutionary Computation (EC) research, developed in the Java programming language. It provides a high-level software environment to do any kind of Evolutionary Algorithm, with support for genetic algorithms, genetic programming and evolutionary programming.

Experiments were run on a PC equipped with an Intel Core i7 processor running at 2.66GHz with one and two NVIDIA GeForce 285 GTX video card equipped with 2GB of GDDR3 video RAM. No overclock was made to any of the hardware.

UCI Machine Learning Repository provides a repository of databases, domain theories and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms. We have selected two of them 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.

Two different GP classification models proposed in the literature are listed. The evaluation from each model is detailed for parallelization purpose. Full description can be obtained from their references.

Falco model

Falco, Della y Tarantino [6] propose a method to get the fitness of the rule by evaluating the antecedent over all the patterns within the data set. The adjustment function calculates the difference between the number of examples where the rule correctly predicts the membership or not of the class and number of examples where the opposite occurs, then the prediction is wrong. Finally fitness is expressed as:

$$fitness = I - (successes - fails) + \alpha * N \qquad (1)$$

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

Table 1: Execution time results

		Shuttle data set								Poker-I data set							
		Falco Model				Bojarczuk Model				Falco Model				Bojarczuk Model			
	Pop	100	200	400	800	100	200	400	800	100	200	400	800	100	200	400	800
Time	Java	102	190	403	822	1832	2315	3652	6193	293	569	1093	2100	1447	2757	5177	9720
	C1	19	23	77	165	97,7	185	328	655	64	115	196	430	261	488	888	1711
	C2	10	12	38	82	51,1	94,1	165	342	33	58	99	216	136	250	446	863
	C4	5	6	20	42	28,1	48,9	89,2	184	17	30	51	111	71,4	133	230	436
	C8	5	6	19	40	27,9	49,4	90,1	177	16	28	47	80	66,4	125	213	408
	GPU	17	23	67	142	3,9	7,2	14,2	28,4	55	99	172	362	20,8	40	∞	∞
	GPUs	10	13	35	73	2,1	3,7	7,1	14	29	52	88	183	10,5	20	37,5	∞
Speedup	Java	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	C1	5,4	8,1	5,2	5,0	18,8	12,5	11,2	9,5	4,6	5,0	5,6	4,9	5,5	5,7	5,8	4,7
	C2	10,6	15,9	10,5	10,1	35,9	24,6	22,1	18,1	9,0	9,8	11,1	9,7	10,6	11,0	11,6	11,3
	C4	19,7	30,3	20,5	19,8	65,2	47,3	40,1	33,7	16,8	18,9	21,6	18,9	20,3	20,7	22,5	22,3
	C8	19,9	30,1	21,2	20,6	65,7	46,8	40,5	34,6	18,5	20,5	23,3	26,4	21,8	22,0	24,2	23,8
	GPU	349	491	361	347	468	325	256	218	319	343	380	348	69,7	69,6	∞	∞
	GPUs	636	901	690	672	890	625	513	442	611	662	749	688	138	136	138	∞

of nodes. The closer α to 1 is, the more importance is given to simplicity.

Bojarczuk model

Bojarczuk, Lopes y Freitas [2] proposal presents a method in which each rule is evaluated for all of the classes simultaneously for a pattern. The classifier is formed by taking the best individual for each class generated during the evolutionary process. Instead of successes and fails it measures t_p, f_p, t_n and f_n.

The fitness function used combines two indicators that are commonplace in the domain, namely the sensitivity (Se) and the specifity (Sp), defined as follows:

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

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

$$Sy = \frac{maxnodes - 0.5 * numnodes - 0.5}{maxnodes - 1} \tag{3}$$

where numnodes is the current number of nodes of and individual and maxnodes is the maximum allowed size of a tree. Finally, the fitness function is calculated as the product of the indicators of the predictive accuracy and simplicity:

$$fitness = Se * Sp * Sy \tag{4}$$

6 RESULTS

The results of the two GP classification algorithms benchmarked using UCI data sets are shown at table 1 and figures 4 and 5. Rows in the first half represents the generation time and rows in the second half represents the speedup compared to Java 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 GPUs devices. Infinite values mean memory requirements excedded. In Falco table, GPU time is expressed in seconds and CPU time is indicated in minutes. Bojarczuk table times are in seconds.

Benchmarks 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. To go beyond, GPUs work much better. Its parallelized model allows to speed up classification problem from a month to an hour. Real classification training usually needs dozens of evaluations to get an accurate result, so the absolute time saved is a great deal of time. The highest speedup is obtained with Falco model, and it increased performance by a factor up to 700 over the Java standard CPU solution and up to 130 over the C single threaded CPU implementation.

7 CONCLUSION

Massive parallelization using NVIDIA CUDA framework provides a speedup of hundred of times over Java and C implementation. GPUs are best for thousands

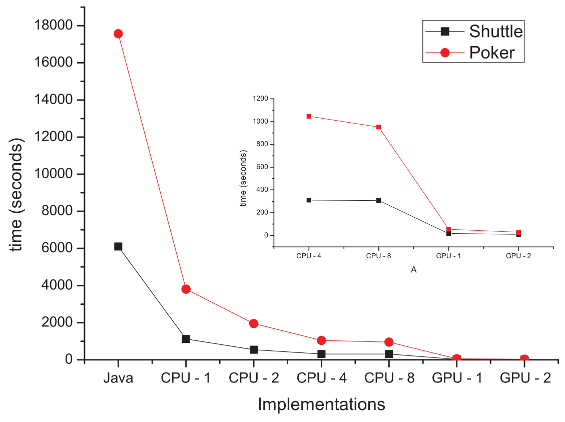

Figure 4: Falco implementation time comparison.

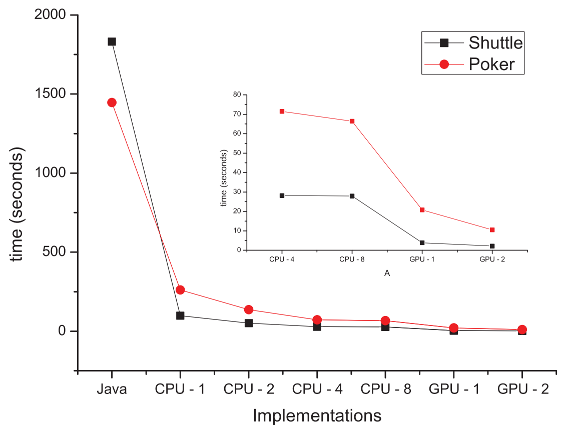

Figure 5: Bojarczuk implementation time comparison.

threads tasks where each thread does its job but all of them collaborate in the execution of the program.

CPU solution is lineal complex. However, the GPU groups the threads in a block, then a grid of blocks is exectued in SMs multiprocessors, one per SM. Thus, linearity is approximately given by the number of 30-blocks grid. This implementation allows future scalability for GPUs with more cores. Next NVIDIA GPU codenamed Fermi doubles the number of cores available up to 512. Make note that i7 CPU scores are 2.5 times faster than 3.0 GHz PIV.

Further work will implement the whole algorithm inside the GPU so selection, crossover and mutation steps will be parallelized and data transfers between CPU and GPU memory are expected to minimize.

ACKNOWLEDGMENTS

This work has been financed in part by the TIN2008-06681-C06-03 project of the Spanish Inter-Ministerial Commission of Science and Technology (CICYT), the P08-TIC-3720 project of the Andalusian Science and Technology Department and FEDER funds.

References

[1] A. A. Freitas, “Data Mining and Knowledge Discovery with Evolutionary Algorithms”, Springer-Verlag, 2002.

[2] C.C. Bojarczuk, H.S. Lopes, A.A. Freitas, and E.L. Michalkiewicz, “A constrained-syntax genetic programming system for discovering classification rules: application to medical data sets”, *Artificial Intelligence in Medicine*, vol. 30, no. 1, pp. 27-48, January 2004.

[3] Chitty, D., “A data parallel approach to genetic programming using programmable graphics hardware”, *GECCO’07: Proceedings of the Conference on Genetic and Evolutionary Computing*, pp. 1566 - 1573, 2007.

[4] Deb K, “A population-based algorithm-generator for real-parameter optimization”, *Soft Computing* 9(4):236253, 2005.

[5] Harding, S., Banzhaf, W., “Fast genetic programming and artificial developmental systems on gpus”, *HPCS’07: Proceedings of the Conference on High Performance Computing and Simulation*, pp. 2 - 2, 2007.

[6] I. De Falco, A. Della Cioppa, and E. Tarantino, “Discovering interesting classification rules with genetic programming”, *Applied Soft Computing Journal*, vol. 1, no.4, pp. 257-269, 2002.

[7] M.G. Arenas, J.G. Castellano, J. Carpio, P.A. Castillo, M. Cillero, J.J. Merelo, A. Prieto, V.Rivas, G. Romero. “Speedup measurement for a distributed evolutionary algorithm that uses Jini”, *In XI Jornadas de Paralelismo*, pp. 247 -252, ISBN:84-699-3003-6, Granada, Spain, September, 2000.

[8] P.G. Espejo, S. Ventura and F. Herrera, “A Survey on the Application of Genetic Programming to Classification”, *IEEE Transactions on Systems, Man and Cybernetics*: Part C, (accepted).

[9] S.Ventura, C. Romero, A. Zafra, J.A. Delgado and C. Hervás. “JCLEC: a Java framework for evolutionary computation”, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 2007.

[10] William B. Langdon, Andrew P. Harrison, “GP on SPMD parallel graphics hardware for mega Bioinformatics data mining”, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12:1169–1183, 2008.