Parallel evaluation of Pittsburgh rule-based classifiers on GPUs

Alberto Cano, Amelia Zafra, Sebastián Ventura*

Department of Computer Science and Numerical Analysis, University of Cordoba, Campus Universitario Rabanales, Edificio Einstein, Tercera Planta, 14071 Cordoba, Spain. Tel.: +34 957212218; Fax: +34 957218630

Abstract

Individuals from Pittsburgh rule-based classifiers represent a complete solution to the classification problem and each individual is a variable-length set of rules. Therefore, these systems usually demand a high level of computational resources and run-time, which increases as the complexity and the size of the data sets. It is known that this computational cost is mainly due to the recurring evaluation process of the rules and the individuals as rule sets. In this paper we propose a parallel evaluation model of rules and rule sets on GPUs based on the NVIDIA CUDA programming model which significantly allows reducing the run-time and speeding up the algorithm. The results obtained from the experimental study support the great efficiency and high performance of the GPU model, which is scalable to multiple GPU devices. The GPU model achieves a rule interpreter performance of up to 64 billion operations per second and the evaluation of the individuals is speeded up of up to 3.461× when compared to the CPU model. This provides a significant advantage of the GPU model, especially addressing large and complex problems within reasonable time, where the CPU run-time is not acceptable.

Keywords: Pittsburgh, classification, rule sets, parallel computing, GPUs

*Corresponding author

Email addresses: acano@uco.es (Alberto Cano), azafra@uco.es (Amelia Zafra), sventura@uco.es (Sebastián Ventura)

1. Introduction

Evolutionary computation and its application to machine learning and data mining, and specifically, to classification problems, has attracted the attention of researchers over the last decade [1, 2, 3, 4, 5]. Classification is a supervised machine learning task which consists in predicting class membership of uncategorised examples using the properties of a set of train examples from which a classification model has been inducted [6].

Rule-based classification systems are especially useful in applications and domains which require comprehensibility and clarity in the knowledge discovery process, expressing information in the form of IF-THEN classification rules. Evolutionary rule-based algorithms take advantage of fitness-biased generational inheritance evolution to obtain rule sets, classifiers, which cover the train examples and produce class prediction over new examples.

Rules are encoded into the individuals within the population of the algorithm in two different ways: individual = rule, or individual = set of rules. Most evolutionary rule-based algorithms follow the first approach due to its simplicity and efficiency, whereas the latter, also known as Pittsburgh style algorithms, are not so usually employed because they are considered to perform slowly [7]. However, Pittsburgh approaches comprise other advantages such as providing individuals as complete solutions to the problem and allow of considering relations between the rules within the evolutionary process.

The efficiency, computational cost, and run-time of Pittsburgh rule-based systems is a primary concern and a challenge for researchers [8, 9], especially when seeking their scalability to large scale databases [10, 11], processing vast amounts of data within a reasonable amount of time. Therefore, it becomes crucial to design efficient parallel algorithms capable of handling these large amounts of data [12, 13, 14, 15].

Parallel implementations have been employed to speed up evolutionary algorithms, including multi-core and distributed computing [16, 17], master–slave models [18], and grid computing environments [19, 20]. Over the last few years, increasing attention has focused on graphic processing units (GPUs). GPUs are devices with multi-core architectures and massive parallel processor units, which provide fast parallel hardware for a fraction of the cost of a traditional parallel system. Actually, since the introduction of the computer unified device architecture (CUDA) in 2007, researchers all over the world have harnessed the power of the GPU for general purpose GPU computing (GPGPU) [21, 22, 23, 24].

The use of GPGPU has already been studied for speeding up algorithms within the framework of evolutionary computation and data mining [25, 26, 27, 28], achieving high performance and promising results. Specifically, there are GPU-accelerated genetic rule-based systems for individual = rule approaches, which have been shown to achieve high performance [29, 30, 31]. Franco et al. [29] reported a speedup of up to 58× using the BioHEL system. Cano et al. [30] reported a speedup of up to 820×, considering a scalable model using multiple GPU devices. Augusto [31] reported a speedup of up to 100× compared to a single-threaded model and delivering almost 10× the throughput of a twelve-core CPU. These proposals are all focused on speeding up individual = rule approaches. However, as far as we know, there are no GPU-based approaches to date using an individual = set of rules representation.

In this paper we present an efficient Pittsburgh individuals evaluation model on GPUs which parallelises the fitness computation for both rules and rules sets, applicable to any individual = set of rules evolutionary algorithm. The GPU model is scalable to multiple GPU devices, which allows of addressing larger data sets and population sizes. The rules interpreter, which checks the coverage of the rules over the instances, is carefully designed to maximize its efficiency compared to traditional rules stack-based interpreters. Experimental results demonstrate the great performance and high efficiency of the proposed model, achieving a rules interpreter performance of up to 64 billion operations per second. On the other hand, the individual evaluation performance achieves a speedup of up to 3.461× when compared to the single-threaded CPU implementation, and a speedup of 1.311× versus the parallel CPU version using 12 threads.

This paper is organized as follows. In the next section, genetic rule-based systems and their encodings are introduced, together with the definition of the CUDA programming model on the GPU. Section 3 presents the GPU evaluation model and its implementation in CUDA kernels. Section 4 introduces the experimental study setup, whose results are given in Section 5. Finally, Section 6 collects some concluding remarks.

2. Background

This section introduces the genetic rule-based systems and the encoding of the individuals. Finally, the CUDA programming model on the GPU is presented.

2.1. Genetic rule-based systems

Genetic algorithms (GAs) evolve a population of individuals which correspond to candidate solutions to a problem. GAs have been used for learning rules (Genetic rule-based systems), including crisp and fuzzy rules, and they follow two approaches for encoding rules within a population.

The first one represents an individual as a single rule (individual = rule). The rule base is formed by combining several individuals from the population (rule cooperation) or via different evolutionary runs (rule competition). This representation results in three approaches:

- Michigan: they employ reinforcement learning and the GA is used to learn new rules that replace the older ones via competition through the evolutionary process. These systems are usually called learning classifier systems [32], such as XCS [33], UCS [34], Fuzzy-XCS [35], and Fuzzy-UCS [36].

- Iterative Rule Learning (IRL): individuals compete to be chosen in every GA run. The rule base is formed by the best rules obtained when the algorithm is run multiple times. SLAVE [37], SIA [38] and HIDER [39] are examples which follow this model.

- Genetic Cooperative-Competitive Learning (GCCL): the whole population or a subset of individuals encodes the rule base. In this model, the individuals compete and cooperate simultaneously. This approach makes it necessary to introduce a mechanism to maintain the diversity of the population in order to avoid a convergence of all the individuals in the population. GP-COACH [40] or COGIN [41] follow this approach.

The second one represents an individual as a complete set of rules (individual = set of rules), which is also known as the Pittsburgh approach. The main advantage of this approach compared to the first one is that it allows of addressing the cooperation–competition problem, involving the interaction between rules in the evolutionary process [42, 43]. Pittsburgh systems (especially naive implementations) are slower, since they evolve more complex structures and they assign credit at a less specific (and hence less informative) level [44]. Moreover, one of their main problems is controlling the number of rules, which increases the complexity of the individuals, adding computational cost to their evaluation and becoming an unmanageable problem.

This problem is known as the bloat effect [45], i.e., a growth without control of the size of the individuals.

One method based on this approach is the Memetic Pittsburgh Learning Classifier System (MPLCS) [8]. In order to avoid the bloat effect, they employ a rule deletion operator and a fitness function based on the minimum description length [46], which balances the complexity and accuracy of the rule set. Moreover, this system uses a windowing scheme [47] that reduces the run-time of the system by dividing the training set into many non-overlapping subsets over which the fitness is computed at each GA iteration.

2.2. CUDA programming model

Computer unified device architecture (CUDA) [48] is a parallel computing architecture developed by NVIDIA that allows programmers to take advantage of the parallel computing capacity of NVIDIA GPUs in a general purpose manner. The CUDA programming model executes kernels as batches of parallel threads. These kernels comprise thousands to millions of lightweight GPU threads per each kernel invocation.

CUDA's threads are organised into thread blocks in the form of a grid. Thread blocks are executed in streaming multiprocessors. A stream multiprocessor can perform zero-overhead scheduling to interleave warps (a warp is a group of threads that execute together) and hide the overhead of long-latency arithmetic and memory operations. GPU's architecture was rearranged from SIMD (Single Instruction, Multiple Data) to MIMD (Multiple Instruction, Multiple Data), which runs independent separate program codes. Thus, up to 16 kernels can be executed concurrently as long as there are multiprocessors available. Moreover, asynchronous data transfers can be performed concurrently with the kernel executions. These two features allow of speeding up the execution compared to a sequential kernel pipeline and synchronous data transfers, as in previous GPU architectures.

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

- Global memory: a large long-latency memory that exists physically as an off-chip dynamic device memory. 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: a small low-latency memory that exists physically as on-chip registers and its contents are 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.

- Local memory: 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 multiprocessor, which 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 to execute batches of global accesses, one per thread, exploiting the hardware's warp scheduling to overlap the threads' access latencies.

- Constant memory: this memory is specialised for situations in which many threads will read the same data simultaneously. This type of memory stores data written by the host thread, is accessed constantly, and does not change during the execution of the kernel. A value read from the constant cache is broadcast to all threads in a warp, effectively serving all loads from memory with a single-cache access. This enables a fast, single-ported cache to feed multiple simultaneous memory accesses.

There are some recommendations for improving the performance on a GPU [49]. Memory accesses must be coalesced as with accesses to global memory. Global memory resides in device memory and is accessed via 32, 64, or 128-byte segment memory transactions. It is recommended to perform fewer but larger memory transactions. When a warp executes an instruction which accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly.

To maximise global memory throughput, it is therefore important to maximise the coalescing, by following optimal access patterns, using data types

that meet the size and alignment requirements, or padding data. For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be a multiple of the warp size.

3. Parallel Pittsburgh evaluation on GPU

This section first introduces the encoding of the Pittsburgh individuals on the GPU. Then, it will present the evaluation procedure of an individual's rules. Finally, it will describe the evaluation process of an individual's fitness.

3.1. Pittsburgh individual encoding

Pittsburgh individuals are variable-length sets of rules which may include a default rule class prediction, interesting when using decision lists [50] as individual representation. Rules are one of the formalisms most often used to represent classifiers (decision trees can be easily converted into a rule set [51]). The IF part of the rule is called the antecedent and contains a combination of attribute–value conditions on the predicting attributes. The THEN part is called the consequent and contains the predicted value for the class. This way, a rule assigns a data instance to the class pointed out by the consequent if the values of the predicting attributes satisfy the conditions expressed in the antecedent. Rule specification can be formally defined by means of a context-free grammar [52] as the shown in Figure 1.

$\langle S \rangle \rightarrow \langle S \rangle\ OR\ \langle cmp \rangle$
$\langle S \rangle \rightarrow \langle S \rangle\ AND\ \langle cmp \rangle$
$\langle S \rangle \rightarrow NOT\ \langle S \rangle$
$\langle S \rangle \rightarrow \langle cmp \rangle$
$\langle cmp \rangle \rightarrow \langle op\_num \rangle\ \langle variable \rangle\ \langle value \rangle$
$\langle cmp \rangle \rightarrow \langle op\_cat \rangle\ \langle variable \rangle\ \langle value \rangle$
$\langle op\_num \rangle \rightarrow \geq \mid > \mid < \mid \leq$
$\langle op\_cat \rangle \rightarrow = \mid \neq$
$\langle variable \rangle \rightarrow$ *Any valid attribute in dataset*
$\langle value \rangle \rightarrow$ *Any valid value*

Figure 1: Grammar specification for the rules

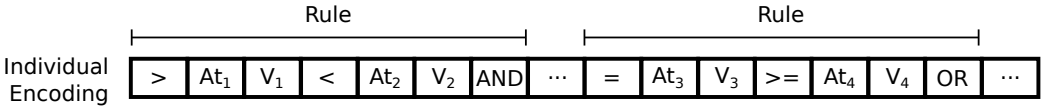

Figure 2: Pittsburgh individual encoding

Figure 2 shows how the rules are stored in the GPU memory. Rules are usually computed by means of a stack-based interpreter [53, 54]. Traditional stack-based interpreters perform push and pop operations on a stack, involving the operator and operands found in the rule. The rule encoding we employ allows the interpreter to achieve maximal efficiency by minimizing the number of push and pop operations on the stack, reading the rules from the left to the right. Attribute–value comparisons are expressed in prefix notation, which places operators to the left of their operands, whereas logical operators are expressed in postfix notation, in which the operator is placed after the operands. This way, the efficiency of the interpreter is increased by minimizing the number of operations on the stack. The interpreter avoids pushing or popping unnecessary operands and behaves as a finite-state machine. For example, the first rule represented in the individual from Figure 2 reads the first element and finds the $>$ operator. The interpreter knows the cardinality of the $>$ operator, which has two operands. Thus, it directly computes $> At_1\ V_1$ and pushes the result into the stack. Then, the next element is $<$, it computes $< At_2\ V_2$ and pushes the result. Finally, the AND operator is found, the interpreter pops the two operands from the stack and returns the AND Boolean computation.

This interpreter model provides a natural representation which allows dealing with all types of logical operators with different cardinalities and operand types while keeping an efficient performance.

3.2. Evaluation of particular rules

Rules within individuals must be evaluated over the instances of the data set in order to assign a fitness to the rules. The evaluation of the rules is divided into two steps, which are implemented in two GPU kernels. The first one, the coverage kernel, checks the coverage of the rules over the instances of the data set. The second one, the reduction kernel, performs a reduction count of the predictions of the rules, to compute the confusion matrix from which the fitness metrics for a classification rule can be obtained.

3.2.1. Rule coverage kernel

The coverage kernel executes the rule interpreter and checks whether the instances of the data set satisfy the conditions comprised in the rules within the individuals. The interpreter takes advantage of the efficient representation of the individuals described in Section 3.1 to implement an efficient stack-based procedure in which the partial results coming from the child nodes are pushed into a stack and pulled back when necessary.

The interpreter behaves as a single task being executed on the Single Instruction Multiple Data (SIMD) processor, while the rules and instances are treated as data. Therefore, the interpreter parallelises the fitness computation cases for individuals, rules, and instances. Each thread is responsible for the coverage of a single rule over a single instance, storing the result of the matching of the coverage and the actual class of the instance to an array. Threads are grouped into a 3D grid of thread blocks, whose size depends on the number of individuals (width), instances (height), and rules (depth), as represented in Figure 3. Thus, a thread block represents a collection of threads which interpret a common rule over a subset of different instances, avoiding a divergence of the kernel, which is known to be one of the major efficiency problems of NVIDIA CUDA programming.

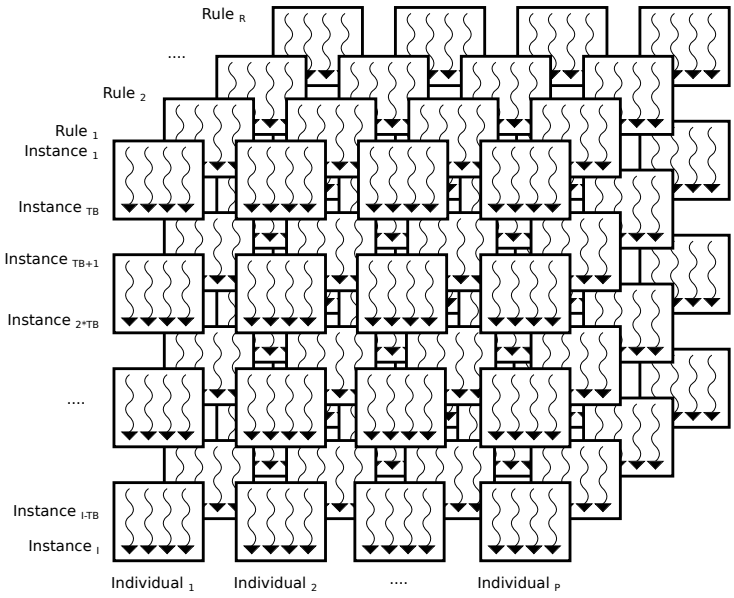

Figure 3: 3D grid of thread blocks

The number of threads per block is recommended to be a multiple of the warp size (a warp is a group of threads that execute together in a streaming multiprocessor), usually being 128, 192, 256, ..., up to 1024 threads per block. This number is important as it concerns the scalability of the model in future GPU devices with a larger number of processors. NVIDIA recommends running at least twice as many thread blocks as the number of multiprocessors in the GPU, and provides an occupancy calculator which reports the GPU occupancy regarding the register and shared memory pressure, and the number of threads per block. Table 1 shows the GPU occupancy to be maximized for different block sizes. 192 threads per block is the best choice since it achieves 100% occupancy and provides more active thread blocks per multiprocessor to hide latency arising from register dependencies, and therefore, a wider range of possibilities given to the scheduler to issue concurrent block to the multiprocessors. Moreover, while the occupancy is maximal, the smaller number of threads per block there is, the higher the number of blocks, which provides better scalability to future GPU devices capable of handling more active blocks concurrently. Scalability to multiple GPU devices is achieved by splitting the population into as many GPUs as available, and each GPU is reponsible of evaluating a subset of the population.

Table 1: Threads per block and GPU occupancy

Threads per block	128	192	256	320
Active Threads per Multiprocessor	1024	1536	1536	1280
Active Warps per Multiprocessor	32	48	48	40
Active Thread Blocks per Multiprocessor	8	8	6	4
Occupancy of each Multiprocessor	67%	100%	100%	83%

Thread accesses to global memory must be coalesced to achieve maximum performance and memory throughput, using data types that meet the size and alignment requirements, or padding data arrays. For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be a multiple of the warp size. Therefore, the results array employs intra-array padding to align the memory addresses to the memory transfer segment sizes [30, 55]. Since the number of threads per block is said to be 192, the results array intra-array padding forces the memory alignment to 192 float values, i.e, 768 bytes. Thus, memory accesses are fully coalesced and best throughput is achieved. Memory alignment and padding details can be found in Section 5.3.2 from the NVIDIA CUDA programming guide.

```
__global__ void coverageKernel(unsigned char* result, double* instancesData,
                        int* instancesClass, float** rules, int** consequent)
{
  int instance = blockDim.y * blockIdx.y + threadIdx.y;
  int memIndex = (blockDim.z*blockIdx.x+blockIdx.z)*nInstances+instance;

  if(covers(&rules[blockIdx.x][blockIdx.z], instance, instancesData))
  {
    if(instancesClass[instance] == consequent[blockIdx.x][blockIdx.z])
      result[memIndex] = 0; // TRUE POSITIVE
    else
      result[memIndex] = 2; // FALSE POSITIVE
  }
  else
  {
    if(instancesClass[instance] != consequent[blockIdx.x][blockIdx.z])
      result[memIndex] = 1; // TRUE NEGATIVE
    else
      result[memIndex] = 3; // FALSE NEGATIVE
  }
}

__device__ bool covers(float* rule, int instance, double* instancesData)
{
  int sp, bufp, attribute;
  float stack[MAX_STACK], op1, op2;

  for(sp = 0, bufp = 0; ;)
  {
    switch(rule[bufp])
    {
      case GREATER:
        attribute = rule[bufp+1];
        op1 = instancesData[numberInstances * attribute + instance];
        op2 = rule[bufp+2];
        if (op1 > op2)                  push(1, stack, &sp);
        else                            push(0, stack, &sp);
        bufp += 3;
        break;
      ...
      case AND:
        op1 = pop(stack, &sp);
        op2 = pop(stack, &sp);
        if (op1 * op2 == 1)             push(1, stack, &sp);
        else                            push(0, stack, &sp);
        bufp++;
        break;
      ...
      case END_RULE:
        return pop(stack, &sp) == 1 ? true : false;
    }
  }
}
```

Listing 1: Rule coverage kernel and interpreter

Threads within a warp shall request consecutive memory addresses that can be serviced in fewer memory transactions. All the threads in a warp evaluate the same rule but over different instances. Thus, the data set must be stored transpose in memory to provide fully coalescing memory requests to the threads from the warp.

The codes for the coverage kernel and the rule interpreter are shown in Listing 1. The coverage kernel receives as input four arrays: an array of attributes values, an array of class values of the instances of the dataset, an array containing the rules to evaluate, and an array containing the consequents of the rules. It computes the matching of the results and return them in an array of matching results. The result of the matching of the rule prediction and the actual class of an instance can take four possible values: true positive (T_P), true negative (T_N), false positive (F_P), or false negative (F_N). Threads and blocks within the kernel are identified by the built-in CUDA variables *threadIdx*, *blockIdx* and *blockDim*, which specify the grid and block dimensions and the block and thread indexes, following the 3D representation shown in Figure 3. Further information about CUDA threads indices can be seen in Section B.4 from CUDA programming guide.

3.2.2. Rule fitness kernel

The rule fitness kernel calculates the fitness of the rules by means of the performance metrics obtained from the confusion matrix. The confusion matrix is a two dimensional table which counts the number of true positives, false positives, true negatives, and false negatives resulting from the matching of a rule over the instances of the data set. There are many well-known performance metrics for classification, such as sensitivity, specificity, precision, recall, F-Measure, etc. The algorithm assigns the fitness values corresponding to the objective or objectives to optimize, e.g, to maximize both sensitivity and specificity at the same time.

The rule fitness kernel is implemented using a 2D grid of thread blocks, whose size depends on the number of individuals (width) and the number of rules (height). The kernel perform a parallel reduction operation over the matching results of the coverage kernel. The naive reduction operation sums in parallel the values of an array reducing iteratively the information.

Our approach does not need to sum the values, but counting the number of T_P, T_N, F_P and F_N. $O(log_2 N)$ parallel reduction is known to perform most efficiently in multi-core CPU processors with large arrays. However, our best results on GPUs were achieved using a 2-level parallel reduction with se-

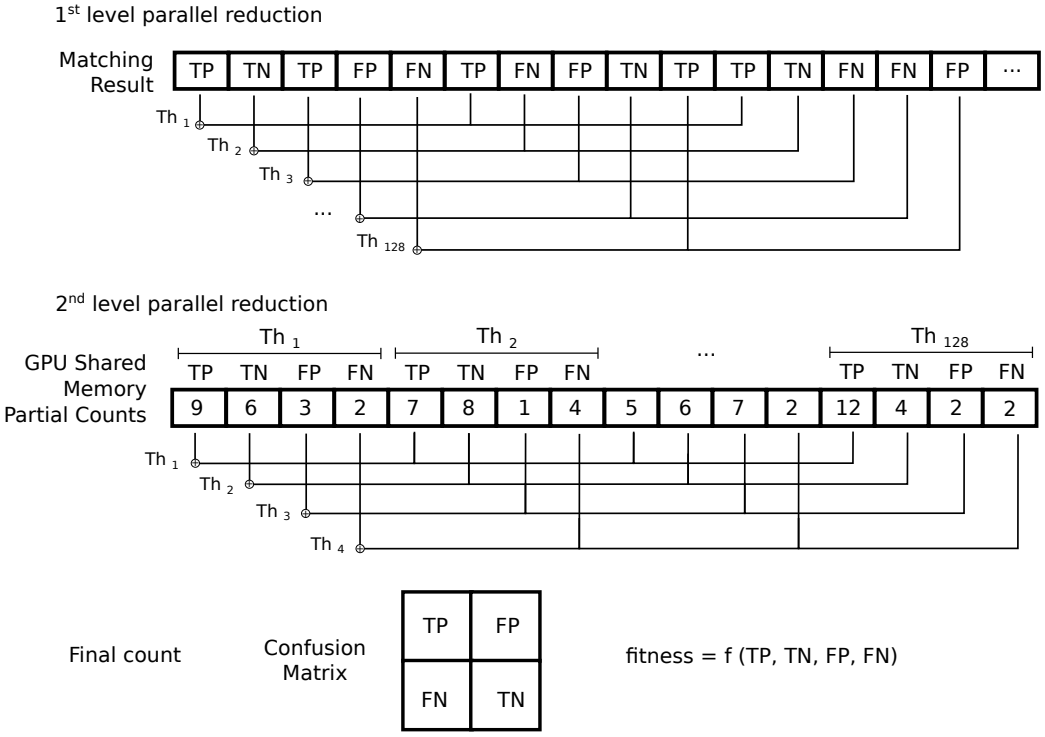

Figure 4: 2-level parallel reduction with sequential addressing

quential addressing using 128 threads per block, which is shown in Figure 4. Accessing sequential memory address in parallel is more efficient than accessing non-contiguous addresses since contiguous data are transferred in a single memory transaction and provides coalesced accesses to threads. Finally, the code for the rule fitness kernel is shown in Listing 2. The input of the kernel is the array of matching results, and returns an array of fitness values. The 2-level parallel reduction takes advantage of GPU shared memory, in which threads within a block collaborate to compute partial counts of the confusion matrix values. Each thread is responsible to count the results from the *base* index to the *top* index. Therefore, contiguous threads address contiguous memory indexes, achieving maximum throughput.

```
__global__ void rulesFitnessKernel(float* fitness, unsigned char* result)
{
  __shared__ int confusionMatrix[512];
  int base = (blockDim.z*blockIdx.x+blockIdx.z)*numberInstances+threadIdx.y;
  int top =  (blockDim.z*blockIdx.x+blockIdx.z+1)*numberInstances-base;

  confusionMatrix[threadIdx.y] = confusionMatrix[threadIdx.y+128] = 0;
  confusionMatrix[threadIdx.y+256] = confusionMatrix[threadIdx.y+384] = 0;

  // Performs the first level reduction of the thread corresponding values
  for(int i = 0; i < top; i+=128)
     confusionMatrix[threadIdx.y*4 + result[base + i]]++;

  __syncthreads();

  if(threadIdx.y < 4)
  {
    // Performs the second level reduction of the half of the sums
    for(int i = 4; i < 512; i+=4)
    {
      confusionMatrix[0] += confusionMatrix[i];   // # true positives
      confusionMatrix[1] += confusionMatrix[i+1]; // # true negatives
      confusionMatrix[2] += confusionMatrix[i+2]; // # false positives
      confusionMatrix[3] += confusionMatrix[i+3]; // # false negatives
    }

    if(threadIdx.y == 0)
    {
      int tp = MC[0], tn = MC[1], fp = MC[2], fn = MC[3];

      float sensitivity, specificity;

      sensitivity = tp / (float) (tp + fn);
      specificity = tn / (float) (tn + fp);

      fitness[blockIdx.z][blockIdx.x] = sensitivity * specificity;
    }
  }
}
```

Listing 2: Rules fitness kernel

3.3. Evaluation of rule sets

Pittsburgh individuals encode sets of rules as complete solutions to the classification problem (classifiers). Many performance measures of a classifier can be evaluated using the confusion matrix. The standard performance measure for classification is the accuracy rate, which is the number of successful predictions relative to the total number of classifications.

The evaluation of the classifiers is divided into two steps, which are implemented in two GPU kernels. The first one, the classification kernel, performs the class prediction for the instances of the data set. The second one, the rule

set fitness kernel, performs a reduction count of the classifier predictions to compute the confusion matrix, from which the fitness metrics for a classifier can be obtained.

3.3.1. Rule set classification kernel

The rule set classification kernel performs the class prediction for the instances of the data set using the classification rules, which are linked as a decision list. An instance is predicted to the class pointed out by the consequent of the first rule which satisfy the conditions of the antecedent. If no rule covers the instance, it is classified using the default class.

In order to save time, the classification kernel reuses the matching results from the rule coverage kernel, and therefore, the rules do not need to be interpreted again. The classifier follow the decision list inference procedure to perform the class prediction. Notice that the class prediction is only triggered when the rule is known to cover the instance (true positive or false positive).

```
__global__ void classificationKernel(unsigned char* result, int* Class,
                                      int defaultClass)
{
  int instance = blockDim.y * blockIdx.y + threadIdx.y;
  int index = blockIdx.x * maxRules * numberInstances + instance;

  for(int i = 0; i < maxRules; i++)
  {
    if(result[index + i*numberInstances] == 0)  // TRUE POSITIVE
    {
      result[index] = 1; // Correctly classified
      return;
    }
    else if(result[index + i*numberInstances] == 2) // FALSE POSITIVE
    {
      result[index] = 0; // Misclassified
      return;
    }
  }

  // If none of the rules covers the instance, apply the default hypothesis
  if(Class[instance] == defaultClass)
    result[index] = 1;  // Correctly classified
  else
    result[index] = 0;  // Misclassified
}
```

Listing 3: Rule set classification kernel

The classification kernel is implemented using a 2D grid of thread blocks, whose size depends on the number of individuals (width) and instances (height). The kernel setup is similar to the rule coverage kernel. The number of threads per block is also 192, to maximize the occupancy of the streaming multiprocessors. Listing 3 shows the code for the classification kernel. The input of the kernel is the array of matching results, an array with information about the instance class and the default class, which applies when none of the rules covers the instance (default hypothesis).

3.3.2. Rule set fitness kernel

```
__global__ void ruleSetFitnessKernel(float* fitness, unsigned char* result)
{
  __shared__ int shmCount[128];

  shmCount[threadIdx.y] = 0;

  int base = blockIdx.x*numberInstances*maxRules + threadIdx.y;
  int top =  blockIdx.x*numberInstances*maxRules + numberInstances - base;

  // Performs the reduction of the thread corresponding values
  for(int i = 0; i < top; i+=128)
  {
    shmCount[threadIdx.y] += result[base + i];
  }

  __syncthreads();

  // Calculates the final amount and the accuracy
  if(threadIdx.y == 0)
  {
    int correctPredictions = shmCount[0];

    for(int i = 1; i < 128; i++)
      correctPredictions += shmCount[i];

    // Compute the accuracy of the classifier
    fitness[blockIdx.x] = correctPredictions / (float) numberInstances;
  }
}
```

Listing 4: Rule sets fitness kernel

The rule set fitness kernel performs a reduction operation over the classifier predictions to count the number of successful predictions. The reduction operation is similar to the one from the rule fitness kernel from Section 3.2.2 and counts the number of correctly classified instances to compute the accuracy of the classifier. The settings for the kernel and the reduction operation

are the same. The kernel is implemented using a 1D grid of thread blocks whose length depends only on the number of individuals. The code for the rule set fitness kernel is shown in Listing 4. The kernel receives as input the array of prediction results from the rule set classification kernel, and returns an array of fitness values which defines the accuracy of the classifiers. Similarly than the rules fitness kernel, shared memory is employed to count partial results and guarantee contiguous and coalesced memory accesses.

4. Experimental setup

This section describes the experimental study setup, the hardware configuration, and the experiments designed to evaluate the efficiency of the GPU model.

4.1. Hardware configuration

The experiments were run on a cluster of machines equipped with dual Intel Xeon E5645 processors running at 2.4 GHz and 24 GB of DDR3 host memory. The GPUs employed were two NVIDIA GTX 480 video cards equipped with 1.5 GB of GDDR5 video RAM. The GTX 480 GPU comprised 15 multiprocessors and 480 CUDA cores. The host operating system was a GNU/Linux Rocks cluster 5.4.3 64 bit together with CUDA 4.1 runtime.

4.2. Problem domains

The performance of the GPU model was evaluated on a series of data sets collected from the UCI machine learning repository [56] and the KEEL data sets repository [57]. These data sets are very varied, with different degrees of complexity. Thus, the number of instances ranges from the simplest, containing 150 instances, to the most complex, containing one million instances. The number of attributes and classes also differ significantly to represent a wide variety of real word data problems. This information is summarized in Table 2. The wide variety of data sets allowed us to evaluate the model performance on problems of both low and high complexity.

4.3. Experiments

The experimental study comprises three experiments designed to evaluate the performance and efficiency of the model. Firstly, the performance of the rules interpreter was evaluated. Then, the times required for evaluating individuals by CPU and GPU were compared. Finally, the efficiency of the model was analysed regarding performance and power consumption.

Table 2: Complexity of the data sets

Data set	#Instances	#Attributes	#Classes
Iris	150	4	3
New-thyroid	215	5	3
Ecoli	336	7	8
Contraceptive	1473	9	3
Thyroid	7200	21	3
Penbased	10992	16	10
Shuttle	58000	9	7
Connect-4	67557	42	3
KDDcup	494020	41	23
Poker	1025010	10	10

4.3.1. Rule interpreter performance

The efficiency of rule interpreters is often reported by means of the number of primitives interpreted by the system per second, similarly to Genetic Programming interpreters, which determine the number of Genetic Programming operations per second (GPops/s) [31, 53, 54].

In this experiment, the performance of the rules interpreter was evaluated by running the interpreter with a different number of rules over data sets with varied number of instances and attributes. Thus, the efficiency of the interpreter was analysed regarding its scalability to larger numbers of rules and instances.

4.3.2. Individual evaluation performance

The second experiment evaluated the performance of the evaluation of the individuals and their rules in order to compute their fitness values. This experiment compared the execution times (these times consider in the case of CPU cluster, data transfers between compute nodes and the GPU times, the data transfer between host and GPU memory) dedicated to evaluate different population sizes over the data sets. The range of population sizes varies from 10 to 100 individuals. This range of population sizes is commonly used in most of the classification problems and algorithms, and represents a realistic scenario for real world data. The number of rules of each individual is equal to the number of classes of the data set, and the length of the rules varies stochastically regarding to the number of attributes of the data set, i.e., rules are created adapted to the problem complexity. Thus, the experiments are not biased for unrealistic more complex rules and individuals which would

obtain better speedups. The purpose of this experiment was to obtain the speedups of the GPU model and check its scalability to large data sets and multiple GPU devices. Extension to multiple GPUs is simple, the population is divided into as many GPUs as available, and each GPU is responsible of evaluating a subset of the population. Therefore, the scalability is guaranteed to larger population sizes and further number of GPU devices.

4.3.3. Performance per Watt

Power consumption has increasingly become a major concern for high-performance computing, due not only to the associated electricity costs, but also to environmental factors [58]. The power efficiency is analysed based on the throughput results on the evaluated cases. To simplify the estimates, it is assumed that the devices work at their full occupancy, that is, at maximum power consumption [31]. One NVIDIA GTX 480 GPU consumes up to 250 W, whereas one Intel Xeon E5645 consumes up to 80 W. The efficiency of the model is evaluated regarding the performance per Watt (GPops/s/W). The power consumption is reported to the CPU or GPU itself and it does not take into account the base system power consumption. We followed this approach because it is the commonly accepted way both in academia and industry [31] to report the performance per watt efficiency.

5. Results

Table 3 shows the rule interpreter execution times and performance in terms of the number of primitives interpreted per second (GPops/s). Each row represents the case of a stack-based interpretation of the rules from the population over the instances of the data sets. The number of rules of each individual is equal to the number of classes of the data set. The number of primitives, Genetic Programming operations (GPops), reflects the total number of primitives to be interpreted for that case, which depends on the variable number of rules, their length, and the number of instances, representing the natural variable length of Pittsburgh problems.

The single-threaded CPU interpreter achieves a performance of up to 9.63 million GPops/s, whereas multi-threading with 4 CPU threads brings the performance up to 34.70 million GPops/s. The dual socket cluster platform allows two 6-core CPUs and a total of 12 CPU threads, which are capable of running up to 92.06 million GPops/s in parallel.

Table 3: Rule interpreter performance

			Interpreter Time (s)					GPops/s (million)				
Data set	Population	GPops	1 CPU	4 CPU	12 CPU	1 GPU	2 GPU	1 CPU	4 CPU	12 CPU	1 GPU	2 GPU
Iris	10	88,560	12	10	8	0.0272	0.0201	7.38	8.86	11.07	3,259.72	4,406.85
	25	225,450	25	12	11	0.0390	0.0276	9.02	18.79	20.50	5,774.85	8,163.75
	50	458,460	49	19	15	0.0526	0.0295	9.36	24.13	30.56	8,719.95	15,522.07
	100	929,340	100	34	21	0.0988	0.0491	9.29	27.33	44.25	9,410.85	18,919.79
New-thyroid	10	126,608	15	12	10	0.0274	0.0204	8.44	10.55	12.66	4,627.49	6,211.15
	25	322,310	35	19	11	0.0400	0.0265	9.21	16.96	29.30	8,064.20	12,164.48
	50	655,428	71	22	13	0.0505	0.0340	9.23	29.79	50.42	12,988.03	19,268.23
	100	1,328,612	144	41	24	0.0967	0.0535	9.23	32.41	55.36	13,734.41	24,832.01
Ecoli	10	539,976	58	18	13	0.0516	0.0392	9.31	30.00	41.54	10,461.41	13,763.66
	25	1,354,168	152	44	23	0.0972	0.0533	8.91	30.78	58.88	13,929.48	25,416.07
	50	2,830,344	299	89	41	0.1813	0.1075	9.47	31.80	69.03	15,607.60	26,339.56
	100	5,658,272	587	166	69	0.3471	0.1708	9.64	34.09	82.00	16,299.87	33,118.75
Contraceptive	10	869,200	96	30	22	0.0550	0.0322	9.05	28.97	39.51	15,801.34	27,000.50
	25	2,212,750	243	70	32	0.1107	0.0625	9.11	31.61	69.15	19,990.88	35,424.40
	50	4,499,700	499	134	102	0.1974	0.1107	9.02	33.58	44.11	22,793.91	40,640.35
	100	9,121,300	989	295	112	0.3857	0.1950	9.22	30.92	81.44	23,646.97	46,773.98
Thyroid	10	4,250,880	488	159	74	0.1625	0.0903	8.71	26.74	57.44	26,165.06	47,089.68
	25	10,821,600	1,204	312	153	0.3810	0.2024	8.99	34.68	70.73	28,406.13	53,474.86
	50	22,006,080	2,394	703	308	0.7508	0.3780	9.19	31.30	71.45	29,308.30	58,219.61
	100	44,608,320	4,824	1,403	616	1.5396	0.7536	9.25	31.79	72.42	28,974.27	59,196.14
Penbased	10	22,712,032	2,422	1,001	550	0.7889	0.3928	9.38	22.69	41.29	28,789.64	57,820.86
	25	55,672,176	6,233	1,829	777	1.9124	1.0215	8.93	30.44	71.65	29,110.43	54,498.50
	50	115,617,696	12,259	3,332	1,371	3.9284	1.9120	9.43	34.70	84.33	29,431.36	60,470.52
	100	228,030,384	24,292	7,447	2,477	7.7361	3.9117	9.39	30.62	92.06	29,476.28	58,294.27
Shuttle	10	80,804,052	9,079	3,310	2,416	2.6057	1.3069	8.90	24.41	33.45	31,010.93	61,826.71
	25	204,515,682	23,116	7,325	3,332	6.5364	3.4067	8.85	27.92	61.38	31,288.50	60,033.02
	50	424,691,064	45,834	14,661	5,839	13.4832	6.5357	9.27	28.97	72.73	31,497.72	64,980.39
	100	852,514,068	90,649	71,206	11,197	27.0089	13.4812	9.40	11.97	76.14	31,564.16	63,237.33
Connect-4	10	39,886,112	5,206	2,026	1,772	1.3552	0.7188	7.66	19.69	22.51	29,431.90	55,491.10
	25	101,539,340	13,164	4,455	2,472	3.3903	1.7277	7.71	22.79	41.08	29,949.92	58,770.99
	50	206,483,592	25,123	8,250	3,714	6.8567	3.3941	8.22	25.03	55.60	30,114.12	60,835.82
	100	418,560,968	54,029	37,498	7,251	13.8397	6.8558	7.75	11.16	57.72	30,243.40	61,051.74
Kddcup	10	2,297,785,824	293,657	99,679	64,722	73.1540	37.1375	7.82	23.05	35.50	31,410.28	61,872.44
	25	5,985,447,516	733,670	208,969	86,570	189.2256	96.7096	8.16	28.64	69.14	31,631.28	61,890.96
	50	11,748,586,032	1,466,624	389,555	145,077	372.2638	189.2589	8.01	30.16	80.98	31,559.84	62,076.80
	100	23,408,248,464	2,900,167	1,926,780	290,873	742.0416	372.2767	8.07	12.15	80.48	31,545.73	62,878.63
Poker	10	2,118,078,368	237,524	104,783	73,069	70.1586	33.9806	8.92	20.21	28.99	30,189.86	62,331.92
	25	5,191,875,024	616,642	191,831	97,471	172.5495	91.4097	8.42	27.06	53.27	30,089.19	56,797.84
	50	10,782,273,504	1,222,919	376,896	162,384	356.6680	172.6337	8.82	28.61	66.40	30,230.56	62,457.54
	100	21,265,654,416	2,404,491	1,649,626	284,182	704.3908	356.5822	8.84	12.89	74.83	30,190.13	59,637.45

Table 4: Individual evaluation performance

		Evaluation time (s)					Speedup vs 1 CPU		Speedup vs 12 CPU	
Data set	Population	1 CPU	4 CPU	12 CPU	1 GPU	2 GPU	1 GPU	2 GPU	1 GPU	2 GPU
Iris	10	0.0096	0.0093	0.0090	0.0010	0.0007	9.60	13.71	9.00	12.86
	25	0.0135	0.0122	0.0106	0.0014	0.0009	9.64	15.00	7.57	11.78
	50	0.0203	0.0200	0.0191	0.0016	0.0010	12.69	20.30	11.94	19.10
	100	0.0420	0.0369	0.0247	0.0019	0.0013	22.11	32.31	13.00	19.00
New-thyroid	10	0.0094	0.0090	0.0088	0.0008	0.0006	11.75	15.67	11.00	14.67
	25	0.0166	0.0129	0.0167	0.0012	0.0008	13.83	20.75	13.92	20.88
	50	0.0300	0.0280	0.0188	0.0017	0.0010	17.65	30.00	11.06	18.80
	100	0.0611	0.0533	0.0294	0.0027	0.0012	22.63	50.92	10.89	24.50
Ecoli	10	0.0323	0.0182	0.0122	0.0015	0.0013	21.53	24.85	8.13	9.38
	25	0.0543	0.0475	0.0303	0.0021	0.0014	25.86	38.79	14.43	21.64
	50	0.1026	0.0818	0.0428	0.0029	0.0025	35.38	41.04	14.76	17.12
	100	0.2090	0.1596	0.0717	0.0042	0.0031	49.76	67.42	17.07	23.13
Contraceptive	10	0.0459	0.0419	0.0328	0.0010	0.0008	45.90	57.38	32.80	41.00
	25	0.1002	0.0828	0.0474	0.0013	0.0010	77.08	100.20	36.46	47.40
	50	0.2036	0.1774	0.1046	0.0017	0.0011	119.76	185.09	61.53	95.09
	100	0.4177	0.3415	0.1617	0.0020	0.0017	208.85	245.71	80.85	95.12
Thyroid	10	0.2112	0.1999	0.1691	0.0010	0.0007	211.20	301.71	169.10	241.57
	25	0.4933	0.4162	0.2185	0.0012	0.0010	411.08	493.30	182.08	218.50
	50	1.0148	0.8749	0.3709	0.0015	0.0011	676.53	922.55	247.27	337.18
	100	2.0318	1.5358	0.6768	0.0029	0.0013	700.62	1,562.92	233.38	520.62
Penbased	10	0.7738	0.7118	0.4548	0.0017	0.0011	455.18	703.45	267.53	413.45
	25	2.0064	1.5883	0.7367	0.0030	0.0019	668.80	1,056.00	245.57	387.74
	50	4.0047	3.0185	1.3909	0.0050	0.0032	800.94	1,251.47	278.18	434.66
	100	8.5293	6.2505	2.4959	0.0094	0.0049	907.37	1,740.67	265.52	509.37
Shuttle	10	2.6452	2.7268	2.2034	0.0028	0.0018	944.71	1,469.56	786.93	1,224.11
	25	7.2141	5.8981	3.4525	0.0057	0.0034	1,265.63	2,121.79	605.70	1,015.44
	50	15.6694	12.8749	5.4326	0.0100	0.0057	1,566.94	2,749.02	543.26	953.09
	100	32.1477	24.6465	9.8108	0.0200	0.0111	1,607.38	2,896.19	490.54	883.86
Connect-4	10	2.2684	1.9459	1.2467	0.0020	0.0014	1,134.20	1,620.29	623.35	890.50
	25	4.9263	4.2588	2.3661	0.0034	0.0021	1,448.91	2,345.86	695.91	1,126.71
	50	10.3063	8.4253	4.7221	0.0064	0.0036	1,610.36	2,862.86	737.83	1,311.69
	100	21.8092	16.0537	5.6394	0.0116	0.0063	1,880.10	3,461.78	486.16	895.14
Kddcup	10	84.7511	78.6884	31.2643	0.0515	0.0267	1,645.65	3,174.20	607.07	1,170.95
	25	208.7099	161.4730	73.0745	0.1236	0.0646	1,688.59	3,230.80	591.22	1,131.18
	50	426.7217	291.6318	123.8365	0.2471	0.1246	1,726.92	3,424.73	501.16	993.87
	100	841.4498	578.6901	213.8102	0.4850	0.2447	1,734.95	3,438.70	440.85	873.76
Poker	10	74.0020	69.8736	46.0978	0.0494	0.0277	1,498.02	2,671.55	933.15	1,664.18
	25	193.5917	162.2887	81.8815	0.1203	0.0648	1,609.24	2,987.53	680.64	1,263.60
	50	385.2833	293.4088	126.4694	0.2325	0.1189	1,657.13	3,240.40	543.95	1,063.66
	100	818.7177	591.6098	229.6941	0.4749	0.2390	1,723.98	3,425.60	483.67	961.06

On the other hand, the GPU implementation obtains great performance in all cases, especially over large scale data sets with a higher number of instances. One GPU obtains up to 31 billion GPops/s, whereas scaling to two GPU devices enhances the interpreter performance up to 64 billion GPops/s. The best scaling is achieved when a higher number of instances and individuals are considered, i.e., the GPU achieves its maximum performance and occupancy when there are enough threads to fill the GPU multiprocessors. Figure 5 shows the GPops/s scaling achieved by the GPU model regarding to the number of nodes to interpret. The higher number of nodes to interpret, the higher the occupancy of the GPU and thus, the higher efficiency.

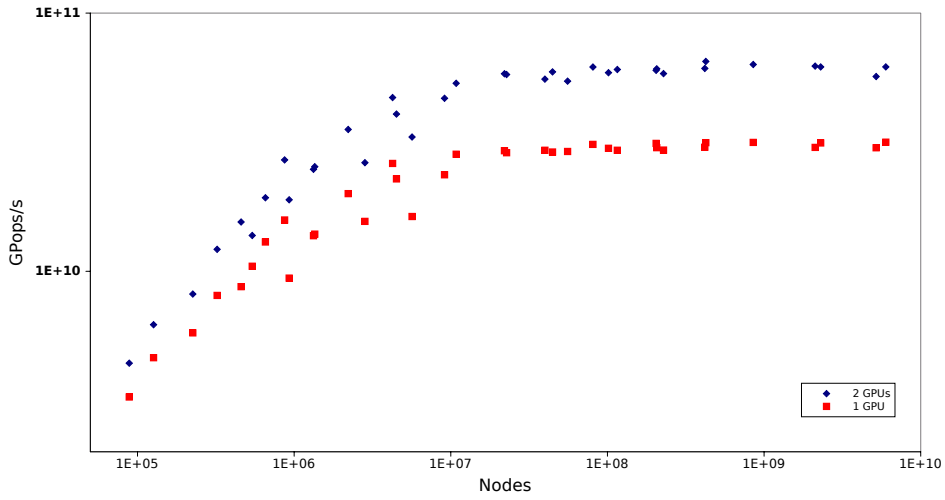

Figure 5: GPU model GPops/s scaling

Table 4 shows the evaluation times and the speedups of the GPUs versus the single-threaded and 12-threaded CPU implementations. The GPU model has high performance and efficiency, which increase as the number of individuals and instances increase. The highest speedup over the single-threaded CPU version is achieved for the Connect-4 data set using 100 individuals (1.880× using one GPU and 3.461× using two GPU devices). On the other hand, compared to the parallel 12-threaded CPU version, the highest speedup is 933× using one GPU and 1.311× using two GPUs. The evaluation times for the Poker data set using 100 individuals are reduced from 818 seconds (13 minutes and 38 seconds) to 0.2390 seconds using two NVIDIA GTX 480 GPUs. Since evolutionary algorithms perform the evaluation of the population each generation, the total amount of time dedicated to evaluate

individuals along generations becomes a major concern. GPU devices allow greatly speeding up the evaluation process and save much time.

Figure 6 shows the speedup obtained by comparing the evaluation time when using two NVIDIA GTX 480 GPUs and the single-threaded CPU evaluator. The figure represents the speedup over the four largest data sets with the higher number of instances. The higher number of instances, the more number of parallel and concurrent threads to evaluate and thus, the higer the occupancy of the GPU.

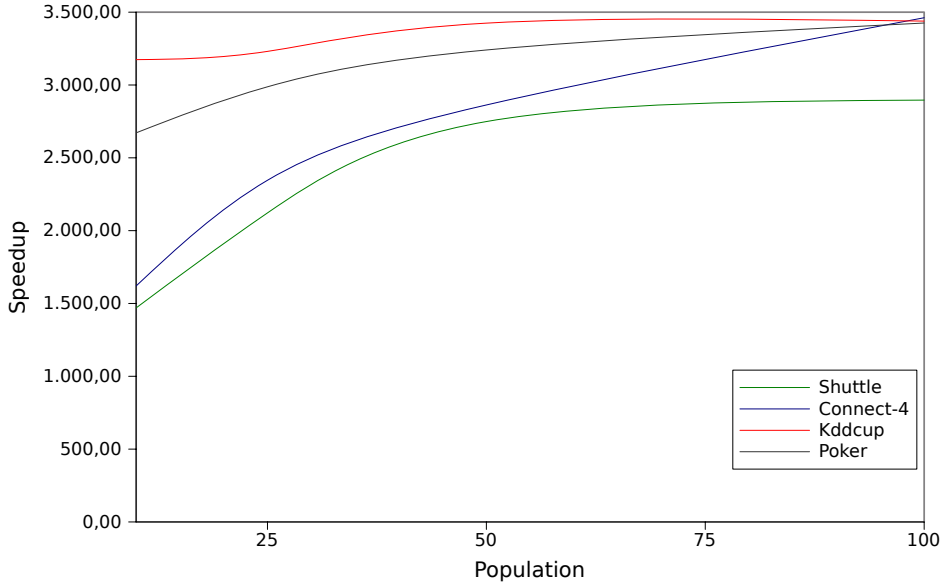

Figure 6: Model speedup using two GPUs

Finally, Table 5 shows the efficiency of the model regarding the computing devices, their power consumption, and their performance in terms of GPops/s. Parallel threaded CPU solutions increase their performance as more threads are employed. However, their efficiency per Watt is decreased as more CPU cores are used. On the other hand, GPUs require many Watts but their performance is justified by a higher efficiency per Watt. Specifically, the single-threaded CPU performs around 0.7 million GPops/s/W whereas using two GPUs increases its efficiency up to 129.96 million GPops/s, which is higher than the efficiency reported in related works [31], which achieve a performance up to 52.7 million GPops/s per Watt.

Table 5: Performance per Watt

Compute device	Watts	GPops/s (million)	GPops/s/W (million)
Intel Xeon E5645 / 1 CPU / 1 core	12.5 W	9.63	0.77
Intel Xeon E5645 / 1 CPU / 4 cores	50 W	34.70	0.69
Intel Xeon E5645 / 2 CPU / 12 cores	160 W	92.06	0.58
NVIDIA GTX 480 / 1 GPU	250 W	31,631.28	126.52
NVIDIA GTX 480 / 2 GPU	500 W	64,980.39	129.96

6. Conclusions

In this paper we have presented a high-performance and efficient evaluation model for individual = rule set (Pittsburgh) genetic rule-based algorithms. The rule interpreter and the GPU kernels have been designed to maximize the GPU occupancy and throughput, reducing the evaluation time of the rules and rule sets. The experimental study has analysed the performance and scalability of the model over a series of varied data sets with different numbers of instances. It is concluded that the GPU implementation is highly efficient, scalable to multiple GPU devices. The best performance was achieved when the number of instances or the population size was large enough to fill the GPU multiprocessors. The speedup of the model was up to 3.461× when addressing large scale classification problems with two GPUs, significantly higher than the speedup achieved by the CPU parallel 12-threaded solution. The rule interpreter obtained a performance above 64 billion GPops/s and even the efficiency per Watt is up to 129 million GPops/s/W.

Acknowledgement

This work was supported by the Regional Government of Andalusia and the Ministry of Science and Technology, projects P08-TIC-3720 and TIN-2011-22408, FEDER funds, and Ministry of Education FPU grant AP2010-0042.

[1] A. Ghosh, L. Jain (Eds.), Evolutionary Computation in Data Mining, volume 163 of *Studies in Fuzziness and Soft Computing*, Springer, 2005.

[2] A. Abraham, E. Corchado, J. M. Corchado, Hybrid learning machines, Neurocomputing 72 (2009) 2729–2730.

[3] E. Corchado, M. Graña, M. Wozniak, New trends and applications on hybrid artificial intelligence systems, Neurocomputing 75 (2012) 61–63.

[4] E. Corchado, A. Abraham, A. de Carvalho, Hybrid intelligent algorithms and applications, Information Sciences 180 (2010) 2633–2634.

[5] W. Pedrycz, R. A. Aliev, Logic-oriented neural networks for fuzzy neurocomputing, Neurocomputing 73 (2009) 10–23.

[6] S. B. Kotsiantis, Supervised machine learning: A review of classification techniques, Informatica (Ljubljana) 31 (2007) 249–268.

[7] A. A. Freitas, Data Mining and Knowledge Discovery with Evolutionary Algorithms, Springer, 2002.

[8] J. Bacardit, N. Krasnogor, Performance and efficiency of memetic pittsburgh learning classifier systems, Evolutionary Computation 17 (2009) 307–342.

[9] J. Bacardit, D. E. Goldberg, M. V. Butz, X. Llor, J. M. Garrell, Speeding-up pittsburgh learning classifier systems: Modeling time and accuracy, Lecture Notes in Computer Science 3242 (2004) 1021–1031.

[10] J. Bacardit, X. Llor, Large scale data mining using genetics-based machine learning, Proceedings of the Genetic and Evolutionary Computation Conference (2011) 1285–1309.

[11] S. Zhang, X. Wu, Large scale data mining based on data partitioning, Applied Artificial Intelligence 15 (2001) 129–139.

[12] E. Cantu-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[13] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Transactions on Evolutionary Computation 6 (2002) 443–462.

[14] M. J. Zaki, C.-T. Ho (Eds.), Large-Scale Parallel Data Mining, volume 1759 of *Lecture Notes in Computer Science*, Springer, 2000.

[15] J. J. Weinman, A. Lidaka, S. Aggarwal, Large-Scale Machine Learning, in: GPU Gems Emerald Edition, Morgan Kaufman, 2011, pp. 277–291.

[16] P. E. Srokosz, C. Tran, A distributed implementation of parallel genetic algorithm for slope stability evaluation, Computer Assisted Mechanics and Engineering Sciences 17 (2010) 13–26.

[17] M. Rodríguez, D. M. Escalante, A. Peregrín, Efficient Distributed Genetic Algorithm for Rule extraction, Applied Soft Computing 11 (2011) 733–743.

[18] S. Dehuri, A. Ghosh, R. Mall, Parallel multi-objective genetic algorithm for classification rule mining, IETE Journal of Research 53 (2007) 475–483.

[19] A. Flling, C. Grimme, J. Lepping, A. Papaspyrou, Connecting Community-Grids by supporting job negotiation with coevolutionary Fuzzy-Systems, Soft Computing 15 (2011) 2375–2387.

[20] P. Switalski, F. Seredynski, An efficient evolutionary scheduling algorithm for parallel job model in grid environment, Lecture Notes in Computer Science 6873 (2011) 347–357.

[21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.

[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU Computing, Proceedings of the IEEE 96 (2008) 879–899.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing 68 (2008) 1370–1380.

[24] D. M. Chitty, Fast parallel genetic programming: Multi-core CPU versus many-core GPU, Soft Computing 16 (2012) 1795–1814.

[25] S. N. Omkar, R. Karanth, Rule extraction for classification of acoustic emission signals using Ant Colony Optimisation, Engineering Applications of Artificial Intelligence 21 (2008) 1381–1388.

[26] K. L. Fok, T. T. Wong, M. L. Wong, Evolutionary computing on consumer graphics hardware, IEEE Intelligent Systems 22 (2007) 69–78.

[27] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, Y. Shi, Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA), The Journal of Supercomputing (2011) 1–26.

[28] A. Cano, A. Zafra, S. Ventura, A parallel genetic programming algorithm for classification, Lecture Notes in Computer Science 6678 (2011) 172–181.

[29] M. A. Franco, N. Krasnogor, J. Bacardit, Speeding up the evaluation of evolutionary learning systems using GPGPUs, Proceedings of the Genetic and Evolutionary Computation Conference (2010) 1039–1046.

[30] A. Cano, A. Zafra, S. Ventura, Speeding up the evaluation phase of GP classification algorithms on GPUs, Soft Computing 16 (2012) 187–202.

[31] D. A. Augusto, H. Barbosa, Accelerated parallel genetic programming tree evaluation with OpenCL, Journal of Parallel and Distributed Computing (2012) Article in Press.

[32] P. L. Lanzi, Learning classifier systems: Then and now, Evolutionary Intelligence 1 (2008) 63–82.

[33] M. V. Butz, T. Kovacs, P. L. Lanzi, S. W. Wilson, Toward a Theory of Generalization and Learning in XCS, IEEE Transactions on Evolutionary Computation 8 (2004) 28–46.

[34] E. Bernadó-Mansilla, J. M. Garrell-Guiu, Accuracy-Based Learning Classifier Systems: Models, Analysis and Applications to Classification Tasks, Evolutionary Computation 11 (2003) 209–238.

[35] J. Casillas, B. Carse, L. Bull, Fuzzy-XCS: A Michigan genetic fuzzy system, IEEE Transactions on Fuzzy Systems 15 (2007) 536–550.

[36] A. Orriols-Puig, J. Casillas, E. Bernadó-Mansilla, Fuzzy-UCS: A Michigan-style learning fuzzy-classifier system for supervised learning, IEEE Transactions on Evolutionary Computation 13 (2009) 260–283.

[37] A. González, R. Pérez, SLAVE: a genetic learning system based on an iterative approach, IEEE Transactions on Fuzzy Systems 7 (1999) 176–191.

[38] G. Venturini, SIA: A Supervised Inductive Algorithm with Genetic Search for Learning Attributes based Concepts, in: European Conference on Machine Learning, pp. 280–296.

[39] J. S. Aguilar-Ruiz, R. Giraldez, J. C. Riquelme, Natural Encoding for Evolutionary Supervised Learning, IEEE Transactions on Evolutionary Computation 11 (2007) 466–479.

[40] F. J. Berlanga, A. J. R. Rivas, M. J. del Jesús, F. Herrera, GP-COACH: Genetic Programming-based learning of Compact and Accurate fuzzy rule-based classification systems for high-dimensional problems, Information Sciences 180 (2010) 1183–1200.

[41] D. P. Greene, S. F. Smith, Competition-based induction of decision models from examples, Machine Learning 13 (1994) 229–257.

[42] R. Axelrod, The Complexity of Cooperation: Agent-based Models of Competition and Collaboration, Princeton University Press, 1997.

[43] A. Palacios, L. Sánchez, I. Couso, Extending a Simple Genetic Cooperative-Competitive Learning Fuzzy Classifier to Low Quality Datasets, Evolutionary Intelligence 2 (2009) 73–84.

[44] T. Kovacs, Genetics-based machine learning, in: G. Rozenberg, T. Bäck, J. Kok (Eds.), Handbook of Natural Computing: Theory, Experiments, and Applications, Springer Verlag, 2011.

[45] W. B. Langdon, Fitness Causes Bloat in Variable Size Representations, Technical Report CSRP-97-14, University of Birmingham, School of Computer Science, 1997.

[46] J. Rissanen, Minimum description length principle, in: Encyclopedia of Machine Learning, 2010, pp. 666–668.

[47] J. Bacardit, Pittsburgh Genetics-Based Machine Learning in the Data Mining era: Representations, generalization, and run-time, Ph.D. thesis, Ramon Llull University, Barcelona, Spain, 2004.

[48] NVIDIA Corporation, NVIDIA CUDA Programming and Best Practices Guide, http://www.nvidia.com/cuda, 2012.

[49] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, IEEE Micro 28 (2008) 13–27.

[50] R. L. Rivest, Learning decision lists, Machine Learning 2 (1987) 229–246.

[51] J. Quinlan, C4.5: Programs for Machine Learning, 1993.

[52] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Chapter 4: Context-Free Grammars, Addison-Wesley, 2006.

[53] W. B. Langdon, W. Banzhaf, A SIMD interpreter for genetic programming on GPU graphics cards, Lecture Notes in Computer Science 4971 (2008) 73–85.

[54] W. B. Langdon, A many threaded cuda interpreter for genetic programming, Lecture Notes in Computer Science 6021 (2010) 146–158.

[55] G. Rivera, C. W. Tseng, Data transformations for eliminating conflict misses, ACM SIGPLAN 33 (1998) 38–49.

[56] D. J. Newman, A. Asuncion, UCI machine learning repository, 2007.

[57] J. Alcalá-Fdez, A. Fernandez, J. Luengo, J. Derrac, S. García, L. Sánchez, F. Herrera, KEEL Data-Mining Software Tool: Data Set Repository, Integration of Algorithms and Experimental Analysis Framework, Analysis Framework. Journal of Multiple-Valued Logic and Soft Computing 17 (2011) 255–287.

[58] W.-C. Feng, X. Feng, R. Ge, Green supercomputing comes of age, IT Professional 10 (2008) 17–23.