Speeding Up the Evaluation of Evolutionary Learning Systems using GPGPUs

María A. Franco
ASAP Research Group
School of Computer Science
University of Nottingham,
Jubilee Campus
Nottingham NG8 1BB
mxf@cs.nott.ac.uk

Natalio Krasnogor
ASAP Research Group
School of Computer Science
University of Nottingham,
Jubilee Campus
Nottingham NG8 1BB
nxk@cs.nott.ac.uk

Jaume Bacardit
ASAP Research Group
School of Computer Science
University of Nottingham,
Jubilee Campus
Nottingham NG8 1BB
jqb@cs.nott.ac.uk

ABSTRACT

In this paper we introduce a method for computing fitness in evolutionary learning systems based on NVIDIA's massive parallel technology using the CUDA library. Both the match process of a population of classifiers against a training set and the computation of the fitness of each classifier from its matches have been parallelized. This method has been integrated within the BioHEL evolutionary learning system. The methodology presented in this paper can be easily extended to any evolutionary learning system. The method has been tested using a broad set of problems with varying number of attributes and instances. The evaluation function by itself achieves speedups up to 52.4X while its integration with the entire learning process achieves speedups up to 58.1X. Moreover, the speedup increases when the CUDA-based fitness computation is combined with other efficiency enhancement mechanisms.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning—*Concept Learning, Induction*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Algorithms, Experimentation, Performance

Keywords

Evolutionary Algorithms, Learning Classifier Systems, Rule Induction, Large-scale Datasets, GPGPUs

1. INTRODUCTION

The match process is the stage that takes most of the execution time in most evolutionary learning systems. Moreover, this situation gets aggravated when trying to handle large-scale datasets. Consequently, there is a need to develop more powerful tools to successfully perform data mining over these datasets. There is extensive efficiency-enhancement work in the evolutionary computation context [18]. For instance, speedups in the match process have been achieved by using processor vector instructions[12] or by using alternative rule encodings[9].

Recently, the usage of general-purpose graphics processing units (GPGPU) has become a popular practice in high performance computing. By exploiting hardware originally designed to render 3D graphics at high speed it is possible to perform highly parallel general purpose computations. GPGPUs have been used already to speed up the evaluation process in genetic algorithms[14], genetic programming[11] and learning classifier systems (LCS)[13].

In this work we present an efficient CUDA-based fitness computation process for the BioHEL system[3], an evolutionary learning system which was conceived to improve the scalability of LCS in large scale bioinformatic datasets[6, 21]. Specifically, we use NVIDIA's Compute Unified Device Architecture (CUDA)[1]. Our implementation calculates the matching between all the rules in the population and all the examples in the training set in parallel. Afterwards we also compute in parallel the accuracy and coverage metrics of the whole population which are the base of the BioHEL fitness function.

The methodology presented in this paper can be easily extended to speedup the match or the evaluation process* in other evolutionary learning systems. Since BioHEL uses supervised learning approach, we will refer to the match process as the evaluation process for the rest of the paper.

We tested our parallel implementation against the serial version using eleven different problems representing a broad set of characteristics. To determine the impact of the new evaluation process first we tested it independently and then we tested it incorporated in the learning system. We also evaluated the combination between the CUDA implementation and the ILAS windowing scheme[4], another efficiency enhancement mechanism already introduced in BioHEL. This comparison determines whether the speedup obtained by both methods is complementary and thus can be successfully combined, or whether there is a trade off that might constraint their integration.

*For Pittsburgh-like LCS and Iterative Rule Learning systems the evaluation process involves the match process and the fitness calculation of the classifiers while for Michigan LCS it only involves the creation of the *match set*

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

The rest of the paper is organised as follows. Section 2 describes the related work along with the architecture used and the BioHEL system. Section 3 describes the methodology and implementation of the new evaluation process. Section 4 explains the experimental design followed. Section 5 shows the results of the evaluation process independently and the results of the evaluation process integrated with BioHEL and Section 6 presents the final remarks and further work.

2. BACKGROUND MATERIAL

2.1 Efficiency enhancements in evolutionary learning systems

The rule matching is the stage that takes the majority of the execution time in most evolutionary learning systems[9, 12]. Reducing this cost has been an important research topic for several years. For instance, Llorà and Sastry[12] used the bitset encoding to paralellize the rule matching in XCS[23]. They used vector operations that have been widely implemented in different processors architectures. In this work they analysed only the evaluation process and obtained up to 90X speedup for binary attributes. Afterwards, Butz et al. [9] presented an extension of the previous work considering that the specificity and the generality of the rules varies during the learning process. They analysed a "specificity encoding" which consists in expressing only the attributes with a value different to "don't care" to reduce the number of comparisons. They compared this encoding with the bitset encoding used in [12] during the entire execution of XCS. They achieved speedups up to 10X with the bitset approach and up to 2X with the specificity encoding.

Mellor and Nicklin[15] presented a population-based approach to find the matchset in XCS. This approach indexes the binary rules (using the 3-ary alphabet {0,1,#}) in a tree structure that allows matching several rules at the same time. The results showed that the number of comparisons made using this technique is between 13% and 37% of the comparisons made with standard matching.

2.2 The BioHEL evolutionary learning system

BioHEL[3] is an evolutionary learning system that uses the Iterative Rule Learning approach[22]. It was designed to deal with large scale bioinformatics datasets. The system inherited most of its components from a previous Pittsburgh Learning Classifier System called GAssist[2]. BioHEL generates sets of rules, each of them learnt sequentially using a standard genetic algorithm. Each time the system learns a new rule, it is added to the rule set and all the examples covered by the rule are removed from the training set.

BioHEL includes a windowing system to improve its efficiency called incremental learning with alternating strata (ILAS)[2]. In ILAS, the training set is divided into a number of non-overlapping strata. Afterwards, each GA iteration uses only one stratum for its fitness computations. These strata are selected using a simple round robin policy. The usage of ILAS redefines the way elitism works within BioHEL. Instead of inserting into the population the best individual from the previous iterations, BioHEL inserts the best individual of the generation where the current stratum was used for the last time. Please see [3] for a complete description of BioHEL.

2.3 GPGPUs and CUDA

The efficiency enhancement method for BioHEL presented in this paper uses CUDA (Computer Unified Device Architecture)[1], a parallel computing architecture developed by NVIDIA. CUDA allows the user to exploit in a general purpose manner the computing capacity inside the NVIDIA GPGPUs (General Purpose Graphic Processor Units).

GPGPUs have been widely used in the last few years for high performance computations. For instance, they have been already used to speedup self organising maps for pattern classification[17], decision trees[19] , neural networks[20], and support vector classification[10]. Genetic Programming and Genetic Algorithms have also benefited from the availability of GPGPU hardware. For instance, Langdon et al.[11] implemented the evaluation process of GP trees for bioinformatic purposes using GPGPUs achieving an speedup around 8X. Moreover, Maitre et al. presented an implementation of a genetic algorithm which performs the evaluation function using CUDA[14]. The major difference between that work and ours is that they do not have a training set but a function instead, which they run in parallel over the different individuals. Our fitness computation is based on managing a training set within the device which implies memory occupancy, while they use a compact mathematical (algorithmic) representation as the fitness function.

CUDA follows the SIMD (Single Instruction Multiple Data) paradigm which consists in running the same operations in each one of *threads* over different data. The thread is the most granular processing unit inside a GPGPU. Each one of the threads runs within a block. A block is the minimum unit processed by a single multiprocessor inside the GPGPU. All the threads inside the same block are able to share information without global synchronisation through the *shared memory*. CUDA provides an extension of the C language to implement the *kernel functions*, which is the code that will be run by the threads.

The wise usage of the memory in CUDA is crucial for the performance of the algorithms. There are five types of memory in CUDA, each of them with a different aim. If any of these types is not properly used, the performance drops dramatically.

- **Shared memory:** is the memory shared by all threads in one block. The maximum amount of shared memory per block is 16Kb. If the access to this memory is coalesced[†] then the access speed will be as fast as accessing registers.
- **Device memory:** is the part of the memory where all the structures passed as parameters are stored. This memory is readable and writable from the device and from the host, and depends on the amount of memory the video card has.
- **Global memory:** is a part of the device memory visible by all the blocks. It is not cached so it is writable by the host and by all the threads. Coalescense is also important when accessing global memory.
- **Constant memory:** is a part of the device memory visible by all the blocks but not writable by the threads. This part of the memory can only be written by the host. The maximum amount of constant memory is 64Kb. This type of memory should only be used to store information that is accessed constantly and does not change during the execution of the program.

[†]Consecutive threads in a block accessing consecutive position in memory

- **Local thread memory:** is the part of the memory only visible by the thread itself, also known as registers. The variables created inside the thread code are stored in this memory.

3. DESIGN OF A CUDA-BASED EFFICIENT FITNESS COMPUTATION PROCESS

As it was mentioned in the introduction, the fitness calculation is one of the major bottlenecks in terms of execution time for an evolutionary learning system; within the fitness calculation computing the accuracy and recall of each rule is the most expensive code[3]. To compute these two values we need three metrics:

1. The number of instances in the training set that match the condition of the rule.
2. The number of instances in the training set that match the class of the rule.
3. The number of instances in the training set that match the class and the condition at the same time.

The evaluation process of this system involves a huge computational cost, because it involves comparing all the rules with all the examples in the instance set or strata (if using ILAS). Our goal is to parallelize the match of all the rules in the population with each one of the examples and then to calculate the results for each classifier in parallel.

Considering that the population size is n and the training set size is m, we intend to make $n \times m$ operations in parallel. The most naive way to do this is to perform the match operations in the GPGPU, then copy back the results of each match to the host and sum up everything here to avoid synchronisations. This approach would be very slow because every time we calculate the fitness it will be necessary to copy a structure of size $O(n \times m)$ from device memory to host memory. This is very undesirable because the execution time of the copy operations is proportional to the structure size. Therefore, the most feasible way to solve this is to calculate the final result for each classifier inside the GPGPU (and in parallel). By doing this, we will only need to copy a structure of size $O(n)$ containing the final results.

At this point our calculation involves two steps which correspond to the main task of each kernel:

1. Calculating the match between all the classifiers and all the instances
2. Reducing[‡] the results for each classifier

Implementing this parallelism involves a greater challenge than the one presented in [14], because it involves not only individuals (rules in this case) but the training set. Moreover, this data does not stay constant during the execution of the algorithm. This makes the implementation even more complex because data will need to be copied multiple times into device memory and these operations are very slow. However, this scheme has potentially a higher degree of parallelism. Our implementation address this problem by attempting to make full usage of the global memory. Considering that in each GA the instances remain constant, and the algorithm will only use a subset of the instances in each iteration, our implementation tries to store in device memory all the instances at the beginning of each GA. Then the windowing is handled inside the device passing the window offset as a parameter of the fitness function. If the global memory is not big enough to store all the instances then further memory calculations are performed as shown in Section 3.1. On the other hand, the classifiers need to be copied into device memory in each iteration. However, this does not involve a considerable computational cost because in most cases the populations are relatively small (i.e. ≤ 500).

The CUDA evaluation is integrated inside BioHEL in two different stages: during the evaluation process and during the elitism. Even though the latter only involves few classifiers, preliminary experiments showed improvements in the performance when this part of the code was parallelized.

The CUDA fitness calculation involves five stages as shown in Figure 1. The following subsections explains in greater detail the strategies followed in the memory calculation stage and evaluation stage.

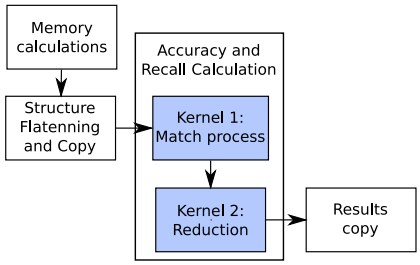

Figure 1: Stages of the CUDA evaluation in each iteration of the GA inside BioHEL

3.1 Memory calculations

Both at the beginning and at each iteration of the GA it is necessary to calculate how many classifiers and instances fit in memory. This allows us to determine whether it is possible to copy all the instances at once. On the other hand, if all the instances do not fit in memory, we calculate if it is possible to fit all the classifiers at once and revisit the instances in several iterations. If none of this is possible, the algorithm calculates the number of classifiers and instances that minimise the memory copy operations.

The heuristic we used to do this consist in assigning half of the memory for storing classifier information and the other half for storing instance information. To calculate this we solve a simple quadratic formula. So considering MI the memory occupied by each instance, MC the memory occupied by each classifier, A the size of the largest output structure used (See Section 3.2.1) and MD the global device memory available, we try to calculate x the number of classifiers and y the number of instances to fit in memory. Given the memory occupancy formula

$$MD = xMC + yMI + x\lceil \frac{y}{512} \rceil A \tag{1}$$

we assume the following

$$xMC = yMI \tag{2}$$

and eliminate the ceiling by assuming the worst case.

$$\lceil \frac{y}{512} \rceil < \frac{y}{512} + 1 \tag{3}$$

Substituting both equations in the equation (1) we find out the we need to solve the following quadratic equation.

$$MD = x(2MC + A) + \frac{x^2 AMC}{512\, MI} \tag{4}$$

The memory calculation function also computes the grid size based on the global amount of device memory. The

[‡]Add all the elements within an array in parallel

amount of threads per block remains constant (512) as it is the maximum possible number of threads inside a grid. Considering that our CUDA program is memory bound, using this number of threads will allow maximising the number of instances that are reduced in a kernel, and minimise the amount of data that we copy back into global memory. Thus, each block will have 512 instances and 1 classifier and the grid size can be easily calculated as $(\lceil y/512 \rceil, x)$.

3.2 Evaluation process

To perform all the calculations CUDA uses two kernel functions. The first kernel is in charge of performing the match operations between the rules and the examples. Each thread will carry out a single match operation. The second kernel is in charge of counting the matches and mismatches (as specified at the beginning of this section) of a rule. That is, counting (in a general sense, reducing) the results calculated in the previous kernel.

The instances are spread among several blocks, and threads between different blocks do not share high speed access memory. Therefore, it takes more than one step and global synchronisation to reduce the information of all the instances. The most intuitive way to do it is to perform all the reductions in the second kernel. However, this means copying back to global memory a large amount of data at the end of the first kernel, which has a large impact in the run-time. To minimise the volume of data the first stage of the reduction is already done in the first kernel, because this avoids copying back a very large structure into global memory. This first reduction is slower because it reduces three values at the same time. Then the subsequent reductions (for each of the three metrics to be computed) are performed independently in the second kernel in a more efficient way. At the end of the execution of the second kernel we will have three values per classifier to copy to host memory. Also, at the end of each kernel the information is reorganized in order to minimise the run-time of subsequent kernels and the memory copy operations from device to host. In the following subsections will explain more deeply how each one of the kernel functions work.

3.2.1 Kernel 1

The first kernel is in charge of performing, in parallel, the match operations between all the rules and all the examples. Algorithm 1 shows the pseudo-code for this function. After calculating the three match values in each thread it is possible to store these values directly into global memory and continue to the execution of second kernel. But this structure is too big and preliminary experiments showed that storing it in global memory was very slow. This is why these values are stored in a shared memory structure over which a one-level parallel reduction will be performed. This reduction algorithm is based on the parallel reduction algorithm proposed by NVIDIA[16] but instead of reducing only one value it reduces three values at the same time. Now instead of having an output structure of size $O(n \times m)$ we have one of size $O(b \times n)$ where b is the number of blocks. This reduces the amount of time spent in writing in global memory as well as the amount of global memory needed.

At the end of the execution of this kernel, the result values are copied back into three separate blocks as shown in Figure 2. This allows having three separate areas over which we can perform an efficient reduction with the second kernel.

Algorithm 1 KERNEL 1: COMPUTE MATCH

Require: Number of instances $numIns$, number of classifiers $numClass$, array of classifiers $class$, array of instances ins
1: Determine the thread index T_i, the instance index I_i and the classifier index C_i regarding the global memory.
2: Declare $cond, action$ and $match$ arrays as shared memory
3: **if** $I_i < numIns \wedge C_i < numClass$ **then**
4: $cond[T_i] \leftarrow 1$
5: **for** att in $class[C_i].atts \wedge match$ **do**
6: $cond[T_i] \leftarrow !match(ins[I_i].atts[att], class[C_i].atts[att])$
7: **end for**
8: $action[T_i] \leftarrow ins[I_i].action = class[C_i].action \ ? \ 1 : 0$
9: $match[T_i] \leftarrow action[T_i] \wedge cond[T_i]$
10: **else**
11: $match[T_i] \leftarrow action[T_i] \leftarrow cond[T_i] \leftarrow 0$
12: **end if**
13: Perform reduction over $match[T_i], action[T_i], cond[T_i]$
14: **if** $T_i = T_0$ **then** {Only the first thread of a block copies the final value into global memory}
15: Write $match[T_i], action[T_i]$ and $cond[T_i]$ into global memory
16: **end if**

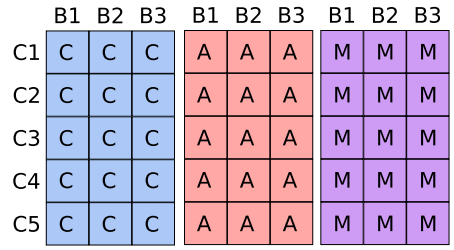

Figure 2: Distribution of the global memory by the end of the first kernel function. C = condition counter, A = action counter, M = match counter.

3.2.2 Kernel 2

The second kernel is in charge of reducing iteratively the information that was previously calculated in the first kernel. This kernel will perform the CUDA parallel reduction algorithm[16] without any major modification. Figure 3 shows an example of the reduction process for the match information of one classifier. This reduction is applied independently over each one of the three memory areas created by the previous kernel. Preliminary experimentation showed that performing reduction of only one value at the same time is faster than performing reduction over three values.

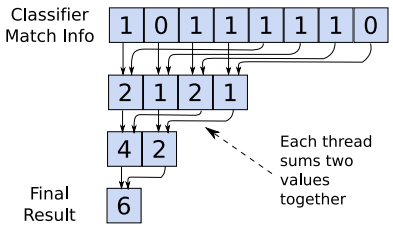

Figure 3: CUDA parallel reduction algorithm.

This kernel is called iteratively until the number of blocks used is equal to one. In the last iteration the kernel will reorder again the data in memory copying all the information of each classifier next to each other. After that, the information of the classifiers will be pack together to make possible copying it back to host memory using a single memory copy operation.

3.3 Handling the ILAS Windowing system

In this paper we also study the integration of the CUDA-based fitness computation and the ILAS windowing scheme, to verify if both efficiency-enhancement techniques can be combined efficiently. As we explained before, if all the instances left in the training set fit in global memory we copy them into global memory at the beginning of each GA. In case the window system is activated, the CUDA fitness computation is called passing the offset of the window as an extra parameter. This is possible because the windows are created at the beginning of the GA and all the windows are stored continuously in memory. It is just necessary to specify the offset to know which window the system is using.

Since the instances tend to be much bigger than the classifiers, we save a lot of computational effort by copying them into global memory once per GA and handling the windowing. However, when the GA finishes the instances that are covered by the new rule are removed from the training set and the windows are created again. At this point it is necessary to copy the instances again into global memory.

3.4 Mixed attributes representation

In order to work with a mix of continuous and nominal attributes at the same time we implemented a slightly different match function. In [5] an efficient CPU-based match process for mixed attributes was proposed, which worked by separating the match of both kinds of attributes in two separate loops. We evaluated that approach in our CUDA implementation. Our preliminary experiments showed that in CUDA we can achieve more speedup by using only one loop that checks the attributes sequentially with a condition inside that checks if the attribute is nominal o real instead of using two loops. This is because the divergent code performs better than the non-coalesced memory access produced by the two loops. In CUDA, the patterns we use to access the memory affect directly the execution time of the kernel functions[16]. When we access memory in a non-coalesced way the performance drops dramatically.

4. EXPERIMENTAL DESIGN

To test the performance of our implementation we decided to perform two stages of experiments. First we evaluate the speedup in the evaluation process independently. Afterwards, we evaluate the speedup of the overall system after incorporating the CUDA evaluation process inside BioHEL. For the first stage, we ran the evaluation process independently of one GA run of 50 iterations (with the rest of the learning mechanisms disabled). We ran a complete GA per execution to test the advantage of copying the instances once at the beginning of each GA and use this information during the subsequent iterations. We test this against the independent evaluation process inside BioHEL.

We compared the speedup over eleven different problems. Table 1 contain a detailed list of relevant characteristics of the datasets regarding our analysis along with the coverage break which is a problem-dependent parameter of BioHEL. The adult (adu), kddcup (kdd), waveform (wav) and connect-4 (c-4) datasets were taken from the UCI repository of machine learning datasets[7]. The CN, SS and SA are protein structure prediction benchmarks already used in [3]. The ParMX (par) dataset is an hybrid parity-multiplexer dataset already used in [8]. The FARS - Fatality Analysis Reporting System dataset (far) was taken from the U.S National Highway Traffic Safety Administration. For each type of problem, we partitioned it using ten-fold cross validation. Over each training set we ran the evaluation process with 25 different seeds. To determine the impact of using different number of windows for the ILAS windowing scheme we did experiments with 1 2 4 6 8 10 15 20 25 30 35 40 45 and 50 windows. The rest of BioHEL's parameters remain the same as the ones used in [5].

	Name	|T|	#Att	#Dis	#Con	#Cl	Cov
Cont.	sat	5790	36	0	36	6	0.1
	wav	4539	40	0	40	3	0.1
	pen	9892	16	0	16	10	0.1
	SS	75583	300	0	300	3	0.0025
	CN	234638	180	0	180	2	0.001
Mixed	adu	43960	14	8	6	2	0.01
	far	90868	29	24	5	8	0.1
	kdd	444619	41	15	26	23	0.1
	SA	493788	270	26	244	2	0.0025
	Par	235929	18	18	0	2	0.001
	c-4	60803	42	42	0	3	0.0025

Table 1: Characteristics of the datasets. \|T\| = Training set size, #Att = Number of atts., #Dis = Number of discrete atts., #Con = Number of continuous atts., #Cl = Number of classes, Cov = Coverage breakpoint

For the second stage of experiments we compare the new version of BioHEL using CUDA with the approach presented in [5]. We used the same problems as in the previous stage and we performed experiments with 1 5 10 25 20 25 30 35 40 45 and 50 windows. Because of the computational cost of each experiment, we ran each one of the ten-fold cross validation training sets only once in this stage.

For the CUDA experiments we used Pentium 4 of 3.6GHz with hyperthreading, 2GB of RAM and a Tesla C1060 with 4GB of global memory and 30 multiprocessors. For the serial experiments we used the High Performance Computing facility of the University of Nottingham each node with 2 quad-core processors (Intel Xeon E5472 3.0GHz), as we wanted to compare our implementation against the most likely architecture a user would use if they do not have access to the GPGPU technology. All the code and datasets used for these experiments are public for replication purposes at `http://www.infobiotics.org/software/`.

5. RESULTS

The following subsections present the results of both stages of experiments. The speed up of the CUDA evaluation process over the serial algorithm is reported in order to determine the advantage of using this parallel architecture. For interpretation and replication purposes we also report the execution time of the two baseline cases (1 window) in Table 2. This table shows that the variance in the execution time of the CUDA algorithm is always smaller than the serial one, which indicates that the average speedup reported is statistically significant. The accuracy of the system is not reported since both implementations behave identical and obtain the same accuracy under similar circumstances.

5.1 Performance of the evaluation process

In Table 3 we present the net speedup (speedup of the CUDA evaluation against the serial version using the same window size) and the total speedup (speedup of the CUDA evaluation against the serial version without using window-

	Prob.	Evaluation Serial	Evaluation CUDA	Integration Serial	Integration CUDA
Cont.	sat	3.6± 0.2	1.9±0.0	95.7± 19.9	25.9± 2.5
	wav	2.6± 0.1	1.6±0.0	75.5± 9.4	24.7± 0.8
	pen	4.9± 0.2	2.2±0.0	149.7± 19.9	40.0± 2.9
	CN	1555.9± 452.8	42.4±0.6	821464.7±167542.0	18644.3±944.0
	SS	770.6± 119.5	14.7±0.2	347979.8± 60982.7	5992.3±247.5
Mixed	adu	147.9± 30.9	10.4±0.1	5422.8± 1410.7	271.7± 26.0
	Par	863.7± 163.1	60.0±0.6	524706.7± 98949.5	19559.8±671.7
	kdd	1715.7± 632.4	95.9±1.4	76442.3± 23533.2	2102.4±191.3
	far	420.8± 90.6	23.1±1.0	2471.3± 701.8	95.0± 41.5
	c-4	343.8± 71.9	17.9±0.2	52917.9± 8059.6	2417.8±170.2
	SA	3776.4±1212.8	90.5±1.2	1252976.8±203186.6	28759.7±552.0

Table 2: Execution time (s) of the evaluation and the integration of both versions in each one of the problems using 1 window

ing) for all the problems. In this table we can notice that the CUDA-based evaluation process achieves speedups for all the problems without using windowing (using 1 window). Moreover, the speedup gets considerably higher in the problems with more than 40000 instances. The highest net speedup achieved was 52.4X in the SS problem. Combined with the ILAS windowing scheme the highest total speedup is 637X in the SA problem. We also observe that the speedup varies with the number of windows and for a higher number of windows the speedup tend to decrease. This is because the CUDA evaluation gets advantage of using large training sets since it does all the example comparisons in parallel. The same happens with the problems that are very small from the start (pen, sat and wav). In these datasets, the overhead of parallelising the fitness function is greater that the advantage obtained.

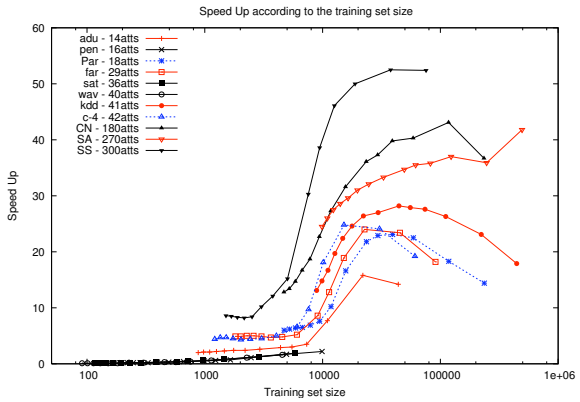

Figure 4: Speed up against the training set size. Problems: Black = Continuous, Red = Mixed, Blue = Discrete.

It is important also to analyse the relation between the speedup and the number of attributes and instances of the problem. Figure 4 shows the relation between training set size and speedup. For the different number of windows x we consider the training set size T_x to be equal to the size of the strata ($T_x = {}^{|T|}/_{x}$). We observe that the relationship between the speedup and the training set is not linear and the behaviour between both implementations (for mixed and real problems) is slightly different. Between 25000 and 50000 the speedup gets its maximum value for most of the problems and then it gets steady or decreases. A possible explanation for the decrease in the speed up when the training set gets bigger is that the number of blocks needed in kernel 1 increase. The size of the structure copied into global memory at the end of this kernel is dependant on the number of blocks, and using more blocks might increase the execution time in this kernel. More experimentation needs to be carried out to validate this hypothesis. Moreover, the decrease in the speed up when the training set is less than 25000 examples is because up to this point the serial algorithm can fit the examples in cache memory. Also the usage of CUDA when the training set is small is less beneficial, because the overhead produced by the memory copy operations is not compensated by the speed up gained.

Also the speedup seems to depend on the number of attributes in the problem. The problems with more attributes get more speedup than the problems with few attributes. Also the large continuous problems (SS and CN) achieve more speedup than the large mixed problems (SA) despite the number of attributes. This is a consequence of the usage of a divergent code to handle mixed attributes. Thus, it makes sense to have an independent CUDA fitness function to handle real problems as simple as possible and handle the rest of problems with a different function.

Even though the speedup varies depending on the window size, the number of attributes of the problem and other problems parameters such as number of iterations per GA, we have shown that the methodology presented improves the performance of the evaluation process when the training set is sufficiently big. In the next section we are going to present the results of the performance of BioHEL after integrating the CUDA-based fitness function.

5.2 BioHEL using CUDA-based evaluation

In Table 4 we present the speedup of the BioHEL system using the CUDA evaluation over the serial version. In this table we can notice that the speedup increases compared to the results of the raw evaluation. The reason for this is that the CUDA evaluation only copies the instance set once for each rule learned. BioHEL runs a GA n times with the same set of instances and learns a rule out of each GA. Then, it selects the best of these rules and incorporates it into the rule set, removes the covered instances from the training set and starts again. Instances do not need to be copied back into the graphics card for each of these GA runs. In the evaluation of the fitness function we were simulating only a single GA run, but is this stage of experiments $n = 2$. Hence, the increase in speedup is expected. However, in all datasets the speedup obtained is less than twice. This is due to the use of the iterative rule learning approach because each time we learn one rule the training set size decreases, while in the evaluation all experiments were done with the full training set. When the training set is too small the overhead of performing the necessary memory copy operations overcomes the advantage obtained by using a GPU. This aspect is not considered in the experiments of the previous stage. Using a large number of strata amplifies this problem because the system performs less calculations in parallel but the memory copy overhead remains constant as the whole training set is copied.

The SS dataset obtained the best results, both using CUDA only (with a maximum speedup of 58.1X) and in combination with ILAS (with a maximum speedup of 765.3X). In most datasets, specially the large ones, we can observe that as the number of windows increases, the total speedup (compared to the serial non-windowed BioHEL) also increases. Thus, we have shown that both efficiency-enhancement mechanisms can be successfully combined.

		Number of Windows													
		1	2	4	6	8	10	15	20	25	30	35	40	45	50
Cont. problems	sat	1.9	1.3	0.8	0.6	0.5	0.4	0.3	0.2	0.2	0.2	0.2	0.1	0.1	0.1
		1.9	2.5	2.8	2.9	2.9	3.0	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1
	wav	1.6	1.1	0.6	0.4	0.3	0.3	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1
		1.6	1.9	2.1	2.1	2.1	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2
	pen	2.2	1.7	1.1	0.7	0.6	0.5	0.4	0.3	0.3	0.2	0.2	0.2	0.2	0.2
		2.2	3.1	3.7	3.6	3.5	3.6	4.1	4.2	4.2	4.2	4.2	4.2	4.2	4.2
	CN	36.7	43.1	40.3	39.8	37.3	36.1	31.6	27.3	22.7	18.7	16.7	14.7	13.4	12.8
		36.7	69.9	128.4	178.3	220.7	258.4	327.8	389.3	432.9	463.3	488.7	506.8	522.3	536.7
	SS	52.4	52.5	50.0	46.1	38.6	30.3	15.2	12.1	10.2	8.4	8.2	8.3	8.5	8.6
		52.4	96.9	168.0	222.7	264.1	293.7	344.7	374.8	394.3	406.9	416.8	424.4	430.0	435.2
Mixed problems	adu	14.2	15.8	7.7	3.5	3.0	2.9	2.6	2.4	2.4	2.3	2.2	2.1	2.1	2.0
		14.2	25.3	41.6	52.9	61.5	68.0	78.4	84.3	88.0	90.8	92.2	93.2	94.5	95.1
	Par	14.4	18.3	22.5	23.1	22.9	21.8	16.6	10.2	7.6	6.9	6.5	6.4	6.2	6.0
		14.4	27.8	51.7	72.7	91.0	107.2	141.1	168.4	189.5	208.3	222.9	235.1	245.1	255.6
	kdd	17.9	23.1	26.3	27.6	27.9	28.2	27.0	26.4	24.6	22.4	19.7	16.7	14.8	13.1
		17.9	34.9	65.3	91.9	115.1	138.3	182.5	218.9	248.1	272.7	291.7	311.4	324.8	339.7
	far	18.2	23.4	24.0	18.9	12.8	8.6	5.2	4.8	4.7	4.9	5.0	5.0	4.9	4.9
		18.2	33.7	60.4	81.3	99.6	115.8	144.6	164.5	176.0	186.4	191.2	197.0	200.9	203.5
	c-4	19.2	24.1	24.8	18.1	9.7	6.6	5.0	4.5	4.4	4.3	4.5	4.7	4.7	4.4
		19.2	35.6	63.3	86.4	104.5	120.0	146.5	162.7	173.1	180.6	184.9	189.4	192.1	194.1
	SA	41.8	35.9	37.0	35.8	35.5	34.7	33.3	32.1	31.0	29.6	28.6	27.5	26.0	24.5
		41.8	79.5	146.2	203.7	252.8	296.3	383.7	449.0	502.3	542.7	578.0	605.8	630.4	649.9

Table 3: Speed up of the CUDA evaluation compared to the serial version using the same window (first row) and the serial version without using windowing (second row)

Figure 5 shows the relation between the training set size and speedup of the integrated system. We can observe similar behaviours in the speedup as the ones explained in the previous section. In the case of the integration the maximum speedup for most of the problems can be found around 50000 and 100000 examples.

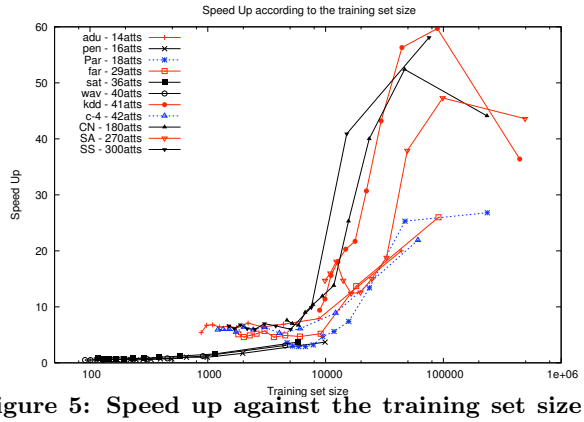

Figure 5: Speed up against the training set size of the integrated learning process. Black = Continuous, Red = Mixed, Blue = Discrete.

6. CONCLUSIONS

We have successfully implemented a CUDA-based evaluation process for the BioHEL evolutionary learning system that achieves a maximum speedup of 52.4X (when evaluated on its own) and up to 58.1X when integrated within BioHEL. Even though these values are dependant on the characteristics of the dataset, we have shown that the CUDA architecture can be used to successfully speed up the evaluation process of LCS by a considerable amount in a broad range of problems. This implementation exploits the intrinsic parallelism in the evaluation process and can be easily extended to any learning classifier system. The speedup obtained with CUDA in the evaluation process helps the system to handle larger problems. Moreover, the combination of the CUDA-based evaluation and the ILAS windowing scheme also showed to be beneficial, obtaining a maximum combined speedup of 765.3X

As further work we would like to extend this implementation so it can use more than one GPGPU at the time. The system could schedule the matching of different groups of classifiers and instances in different GPGPUs, if more than one device is available. This could help handling even much larger problems without the need of using a card with enough global memory which could be very expensive.

Also, it would be interesting to use synthetic problems where the training set size and the number of attributes varies gradually to develop speedup models that can explain in which conditions it is worth to use a CUDA-based fitness computation. Based on these results we could automatically switch from the serial and the CUDA mechanisms based on the training set size and the number of attributes in the problem. It could be also interesting to develop an implementation that copies the information only once into device memory and synchronises the population and the instances when there are changes. This way, the system would only perform small memory copy operations which would be interesting to compare with the current system.

Finally, the use of GPGPUs opens the door to perform much extensive experiments that we could not afford to perform before. For instance, it is known that increasing the number of strata of the ILAS windowing scheme makes the problem more difficult to learn[4]. Nevertheless, it was not possibly to exhaustively determine when this was creating a significant impact in the system's performance in large scale domains because the experiments were too computationally demanding. GPGPUs allow us to perform much larger experiments than before, thus we are able to push forward the boundaries of evolutionary data mining.

7. REFERENCES

[1] *NVIDIA CUDA Programming Guide 2.0*. 2008.

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

[3] J. Bacardit, E. Burke, and N. Krasnogor. Improving the scalability of rule-based evolutionary learning. *Memetic Computing*, 1(1):55–67, March 2009.

		Number of Windows										
		1	5	10	15	20	25	30	35	40	45	50
Cont. problems	sat	3.7	1.6	1.2	1.0	0.8	0.8	0.7	0.7	0.7	0.6	0.8
		3.7	5.6	6.3	8.4	8.3	9.5	8.2	9.4	9.7	8.8	10.6
	wav	3.1	1.2	0.8	0.7	0.7	0.6	0.6	0.5	0.6	0.5	0.5
		3.1	4.7	6.2	8.6	8.9	9.3	9.3	9.2	9.9	9.4	9.3
	pen	3.7	1.7	1.0	1.0	0.7	0.7	0.6	0.6	0.5	0.6	0.5
		3.7	8.1	8.6	11.1	9.8	11.3	11.3	10.6	10.8	11.3	11.5
	CN	44.1	52.4	40.0	25.3	13.8	11.9	10.4	9.0	6.7	7.0	7.6
		44.1	188.9	298.8	394.1	437.0	466.4	504.2	552.5	565.8	588.0	615.5
	SS	58.1	40.9	9.9	6.0	6.6	7.0	6.0	6.1	6.8	6.1	6.6
		58.1	256.0	390.9	498.7	546.1	620.1	635.3	671.9	708.5	714.0	765.3
Mixed problems	adu	20.0	7.9	6.9	6.4	7.1	6.3	6.5	6.4	6.8	6.7	5.4
		20.0	83.8	132.0	165.2	176.3	201.1	198.1	204.9	211.2	218.1	224.3
	Par	26.8	25.3	13.4	7.4	5.6	4.7	3.2	2.9	2.9	3.0	3.6
		26.8	77.6	99.8	118.6	130.7	133.0	133.8	140.3	141.7	138.2	139.9
	kdd	36.4	59.7	56.3	43.2	30.7	21.7	20.3	18.1	15.6	11.4	9.4
		36.4	161.7	292.7	376.7	457.7	480.8	528.9	571.4	578.6	608.6	642.3
	far	26.0	13.7	5.2	4.7	4.9	4.6	5.7	5.2	4.9	4.6	5.1
		26.0	89.1	123.4	159.2	166.3	178.7	181.9	194.7	180.1	192.0	182.0
	c-4	21.9	8.9	6.1	5.3	6.4	5.9	6.4	5.5	6.0	6.0	5.9
		21.9	92.9	157.1	188.8	220.9	238.7	257.4	259.9	271.0	265.3	277.5
	SA	43.6	47.3	37.9	18.8	15.0	12.6	12.4	14.7	18.1	16.0	14.7
		43.6	148.9	214.2	259.6	285.3	309.1	322.8	338.1	347.6	350.2	359.3

Table 4: Speed Up of the BioHEL system using the CUDA evaluation over the serial version

[4] J. Bacardit, D. Goldberg, M. Butz, X. Llora, and J. Garrell. *Speeding-up pittsburgh learning classifier systems: Modeling time and accuracy*. 2004.

[5] J. Bacardit and N. Krasnogor. A mixed discrete-continuous attribute list representation for large scale classification domains. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1155–1162, New York, NY, USA, 2009. ACM.

[6] J. Bacardit, M. Stout, J. Hirst, A. Valencia, R. Smith, and N. Krasnogor. Automated alphabet reduction for protein datasets. *BMC Bioinformatics*, 10(1):6, 2009.

[7] C. Blake, E. Keogh, and C. Merz. *UCI repository of machine learning databases*. 1998. (www.ics.uci.edu/mlearn/MLRepository.html).

[8] M.V. Butz. *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*, volume 109 of *Studies in Fuzziness and Soft Computing*. Springer, 2006.

[9] M.V. Butz, P.L. Lanzi, X. Llorà, and D. Loiacono. An analysis of matching in learning classifier systems. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1349–1356, New York, NY, USA, 2008. ACM.

[10] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*, pages 111, 104, 2008.

[11] W.B. Langdon and A.P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Comput.*, 12(12):1169–1183, 2008.

[12] X. Llorà and K. Sastry. Fast rule matching for learning classifier systems via vector instructions. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1513–1520, New York, NY, USA, 2006. ACM.

[13] D. Loiacono and P. Lanzi. Speeding up matching in XCS. In *12th International Workshop on Learning Classifier Systems*, 2009.

[14] O. Maitre, L.A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410, Montreal, Québec, Canada, 2009. ACM.

[15] D. Mellor and S.P. Nicklin. A population-based approach to finding the matchset of a learning classifier system efficiently. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1267–1274, Montreal, Québec, Canada, 2009. ACM.

[16] NVIDIA. Data-Parallel algorithms. http://developer.download.nvidia.com/compute/cuda /sdk/website/Data-Parallel_Algorithms.html#reduction, September 2009.

[17] R. Prabhu. SOMGPU: an unsupervised pattern classifier on graphical processing unit. In *IEEE Congress on Evolutionary Computation*, pages 1011–1018, 2008.

[18] K. Sastry. Principled efficiency enhancement techniques, 2005. Genetic and Evolutionary Computation Conference - GECCO 2005- Tutorial.

[19] T. Sharp. Implementing decision trees and forests on a GPU. In *Computer Vision – ECCV 2008*, pages 595–608. 2008.

[20] D. Steinkraus, I. Buck, and P.Y. Simard. Using GPUs for machine learning algorithms. In *Proc. of the Eighth International Confernece on Document Analysis and Recognition*, volume 2, pages 1115–1120, 2005.

[21] M. Stout, J. Bacardit, J.D. Hirst, and N. Krasnogor. Prediction of recursive convex hull class assignments for protein residues. *Bioinf.*, 24(7):916–923, 2008.

[22] G. Venturini. SIA: a supervised inductive algorithm with genetic search for learning attributes based concepts. In *Machine Learning: ECML-93 - Proc.\ of the European Conference on Machine Learning*, pages 280–296. Springer-Verlag, 1993.

[23] S.W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, June 1995.