Speeding up the BioHEL evolutionary learning system using GPGPUs

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

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

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

1 Introduction

The BioHEL system is an evolutionary learning system designed to cope with large-scale datasets. This system have several characteristics focused on tackling this kind of problems, such as special representation to determine the relevant attributes in a rule, the usage of a windowing system, among others. Recently, we have extended the system to perform the rule evaluation process inside NVIDIA GPGPUs using CUDA (Compute Unified Device Architecture)[3]. This improvement speeded up the whole learning process up to 58.1X. Moreover, the CUDA-based evaluation was successfully combined with the rest of efficiency enhancement mechanisms within BioHEL. The total speedup using theses techniques was cumulative, obtaining a maximum combined speedup of 765.3X. The following sections will explain how the BioHEL system works and how the CUDA-based fitness function was implemented and incorporated within BioHEL.

2 The BioHEL system

The BioHEL (Bioinformatics-oriented Hierarchical Evolutionary Learning) system is an evolutionary learning system proposed by Bacardit et al.[2] to handle large scale bioinformatic datasets. BioHEL uses iterative rule learning to generate a set of rules. The rules in the solution are evolved, one at the time, using a standard genetic algorithm. Each time the system learns a new rule, adds it to the theory and removes all covered examples from the training set. This process is repeated iteratively until all examples are covered. Moreover, BioHEL incorporates a windowing system to improve its efficiency called Incremental Learning with Alternative Strata (ILAS)[2]. This technique separates the training set into equally distributed strata. In each iteration, the GA chooses a different stratum for its fitness computations based on a simple round-robin policy. For a complete description of the system, please see [2].

3 CUDA-based evaluation process

The evaluation process is the most computationally expensive part of BioHEL. Specifically, it consists of three stages. The first stage consists of matching all the classifiers with all the examples in the training set. This is the most computationally expensive part of the evaluation. The second stage is the computation of the classifier metrics from the match process: a) the number of instances matching the rule condition, b) the number of instances matching the rule class and c) the number of instances matching both the class and the condition at the same time. The third stage is the calculation of the fitness of each rule based on these three metrics.

The architecture chosen for our implementation is CUDA[1]. Our CUDA-based fitness computation process performs stages 1 and 2 of the evaluation inside the GPU. The most intuitive strategy would be to perform only the match process inside the card (as it is the most expensive stage). However, this option is impractical when trying to tackle large-scale problems. Considering that the population size is n, and the training set size is m, we intend to perform $n \times m$ match operations in parallel. This means that it would 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 for memory copy operations is proportional to the structure size. A better strategy is to also calculate in parallel the final result for each classifier (the three performance metrics) inside the GPU. By doing this, we will only need to copy a structure of size $O(n)$ containing the final results.

Although we are only using CUDA to speed up the fitness function, our specific context (supervised learning for large-scale datasets) is much more challenging than a simple parallel evaluation of a fitness formula, as we have to handle very large amounts of memory, not just code. Thus, we need to minimize memory transfers in/out of the device, starting with the training set. If the GPU has enough memory to hold all of it, we only need to copy it once before the GA starts running, and reuse it iteration after iteration. If the complete training set cannot fit in the device memory, then the system will divide the classifiers and instances in a way that minimises the number of memory copy operations.

Moreover, as BioHEL already has other efficiency enhancement mechanisms (the ILAS windowing mechanism), we also tested their combination to check if the individual speedups of the CUDA evaluation function and ILAS were cumulative. This combination has an extra challenge, as copying one stratum to the GPU at each iteration (instead of reusing the same training set iteration after iteration) has some overhead. The solution is simple: all the strata are generated before the GA run and copied together (one after the other) to the GPU. At each iteration, it is only necessary to tell the GPU the offset and size of each stratum.

Considering these two stages, we implemented two kernels which correspond to the main task on each stage. To summarise, our CUDA fitness calculation involves five stages as shown in Figure 1. The following section will explain with greater detail how the kernel functions were implemented.

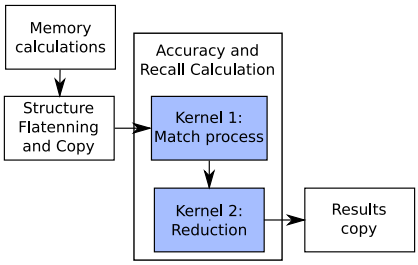

Figure 1: Stages of the CUDA evaluation in each iteration of the GA inside BioHEL

4 Kernel functions

To perform all the calculations, CUDA uses two kernel functions. The first kernel is in charge of performing the match operations between all the rules and all the examples. Each thread will carry out a single match operation. The output of this part are three bits corresponding to the thread's contribution to the three metrics that need to be computed for each classifier. It is possible to store these values directly into global memory and reduce (count) all the values using the second kernel. However, this means that the threads would copy into global memory a large amount of data, which has a large impact in the run-time. Therefore, these values are stored in a shared memory structure (shared among all threads in a single block) over which a one-level parallel reduction is performed. This reduction algorithm is based on the parallel reduction algorithm proposed by NVIDIA[4], but rather than 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 that CUDA uses.

Preliminary experimentation showed that performing the complete reduction of only one value at the same time is faster than performing reduction over three values. Therefore, the second kernel only performs one value reductions. To prepare the data accordingly, we copy back the results at the end of the first kernel into three separate memory areas. Afterwards, the second kernel iteratively reduces the information in these memory areas. This second kernel will perform the CUDA parallel reduction algorithm[4] without any major modification. This process is applied independently over each one of the three areas. Moreover, this kernel is called iteratively until the number of blocks used is equal to one. However, when this kernel finishes, we will have our results scattered through these three memory areas. Thus, we reorder again the data in memory at the end of the kernel's execution in order to place together the results. This makes possible to copy this information back to host memory using a single memory copy operation.

Moreover, to perform reduction efficiently in both kernels, we place 1 classifier and 512 instances per block. This allows to reduce the maximum amount of data in each iteration.

5 Results

Our CUDA experiments were performed using a Tesla C1060 inside a Pentium 4 of 3.6GHz and 2GB of RAM. On the other hand, the serial version of the algorithm was run in the HPC facility at the University of Nottingham, each node with 2 quad-core processors (Intel Xeon E5472 3.0GHz). We wanted to compare with the most likely architecture a user would use if they do not have access to GPU technology.

We tested the whole learning process using different problems that varied the number of instances and the number of attributes. Moreover, we implemented different functions for problems with continuous and discrete attributes to handle more efficiently each case.

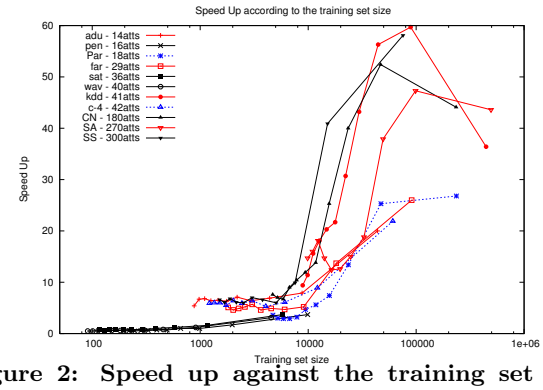

Figure 2: Speed up against the training set size. Black = Continuous, Red = Mixed, Blue = Discrete.

Figure 2 shows the results using the CUDA-based fitness function integrated inside BioHEL using different problems and training set sizes. We obtained up to 58.1X speed up over a problem with 300 continuous attributes and 75583 instances. In absolute terms, this means reducing the runtime of the system from 14 days to 8 hours in the largest dataset. For problems with at least 100000 instances, the speedups are higher than 25X. The problems that benefited the most from the usage of CUDA were the ones with more attributes and more examples in the training set, thus the most challenging ones. 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. Furthermore, the profile information showed that the GPU's idle time was almost negligible in all datasets but the smallest ones, showing a full usage of the card. For more detailed results and profiler information, please see [3] and `http://cs.nott.ac.uk/~mxf/ biohel/`.

6 Conclusion

The implementation of this CUDA-based fitness function for BioHEL was a necessary step in order to tackle efficiently large-scale datasets. This improvement will help us to perform experiments that we could not perform before because the execution times were impractical. This will help us do more exhaustive research on large-scale datasets pushing forward the boundaries of evolutionary learning.

Furthermore, the methodology used can be extended to other evolutionary learning algorithms, specially the ones that apply supervised learning in which the application would be straight forward.

7 References

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

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

[3] M. Franco, N. Krasnogor, and J. Bacardit. Speeding up the evaluation of evolutionary learning systems using gpgpus. In *Proc. of the 12th Annual Conference on Genetic and Evolutionary Computation*, 2010. To appear.

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