GPU-accelerated eXtended Classifier System

Mani Abedini*, Michael Kirley*, Raymond Chiong†‡, and Thomas Weise§
*Department of Computing and Information Systems, The University of Melbourne,
Parkville, Victoria 3010, Australia · Emails: mabedini@student.unimelb.edu.au, mkirley@unimelb.edu.au
†School of Design, Communication and IT, Faculty of Science and IT,
The University of Newcastle, Callaghan, NSW 2308, Australia · Email: Raymond.Chiong@newcastle.edu.au
‡Faculty of Higher Education, Swinburne University of Technology,
Lilydale, Victoria 3140, Australia · Email: rchiong@swin.edu.au
§Nature Inspired Computation and Applications Laboratory (NICAL),
Joint USTC-Birmingham Research Institute in Intelligent Computation and Its Applications,
University of Science and Technology of China, Hefei 230027, Anhui, China · Email: tweise@ustc.edu.cn

***Abstract*—XCS – the eXtended Classifier System – combines an evolutionary algorithm with reinforcement learning to evolve a population of condition-action rules (classifiers). Typically, population-based approaches are slow and increasing the problem size (in terms of the number of features/samples) poses a real threat to the suitability of XCS for real-world applications. Thus, reducing the execution time without losing accuracy is highly desirable. Profiling of the execution of off-the-shelf XCS implementations suggests that the rule matching process is the most computational demanding step. A solution to this is parallelization, i.e., using parallel processing techniques to speed up the matching process (and thus the entire XCS learning process). There are many ways to achieve that, using Graphic Processing Units (GPUs) is one option. Originally, GPUs were designed to conduct a sequence of graphics operations in a massively parallel fashion. Today, GPUs can be used for all sorts of general purpose calculations that are normally handled by the CPU. In this paper, we propose a hybrid rule matching process using both CPU and GPU simultaneously for a maximum performance gain. Our experimental results indicate that this approach does speed up the XCS learning process, and that the GPU is the dominant powerful computing resource in the model.**

This is a preview version of the conference paper [1] (see page 10 for the reference). Read the full piece at http://dx.doi.org/10.1109/CIDM.2013.6597250.

I. Introduction

Since the first publication of XCS [2], it has emerged as one of the successful Michigan-style learning classifier systems [3]. XCS is a successor of the original work by Holland (CS-1) [4]. It is a Genetic-Based Machine Learning (GBML) method, which maintains a population of condition-action rules called classifiers. One part of the learning process is matching corresponding rules to a given data input. The matching process is a highly computational demanding stage. It is a limiting factor to the scalability of XCS. For problems with a large number of features, i.e., so-called *high dimensionality*, XCS can get inconveniently slow.

An analytical investigation on the XCS components has revealed that the matching process consumes 65% to 85% of the overall execution time [3]. Hence, improving the performance of this function has a great potential for decreasing the overall execution time. While using CPU-integrated vector instructions can reduce the matching time [3], XCS can also benefit from using the Graphic Processing Unit (GPU). For example, an enhanced Compute Unified Device Architecture (CUDA)-based matching process can produce a speed-up by factor 3 to 50 [5].

The CUDA library was introduced by NVIDIA, which allows for running general CPU commands on a General-Purpose Graphic Processor Unit (GPGPU) with a massive parallel processing capability. Initially, graphic cards were designed to render 3D graphics. Recent advances in GPU technologies, however, also encourage developers to use the massive parallel cores inside a GPU to run simple instructions on a vast amount of data simultaneously. This idea pushed graphic card manufacturers to provide developing tools and libraries. With this technology, we can employ the high performance computation capacity of a GPGPU. Moreover, nowadays we can afford to have GPGPU-enabled graphic cards in typical desktop computers or laptops. Even high performance computing servers also provide GPU-enabled nodes.

In the literature, however, there is a lack of related work that explores the efficient use of both GPGPUs and CPUs at the same time. Usually, the CPU remains idle while delegating processing to the GPU. Our work bridges this gap by introducing a new hybrid GPGPU and CPU-based XCS matching process. In this paper, we present two different models. Both models are analyzed on various GPU/CPU load configurations and a broad range of real-world data sets. The first tested model is using a trivial parallelization architecture. This model breaches the CUDA best practice guidelines and thus does not reach a reasonable speed-up level. The second model, on the other hand, is designed to follow the CUDA guidelines and achieves speed-up with a factor up to 18. The experimental results therefore show that a good parallelization strategy has a great impact on the performance of a GPGPU-enabled approach.

The rest of the paper is organized as follows. Section II introduces general background information and related work. The experimental design and two proposed hybrid CPU-GPU based matching functions are presented in Section III. Section IV explains the results of the experiments and finally

Algorithm 1 High-level pseudocode of XCS

Require: Input data:σ, Population:[Δ], MaxPopulationSize:Φ

repeat
  $\sigma \leftarrow env$
  $[M] \leftarrow GetMatchSet(\sigma,[\Delta])$
  $[PA] \leftarrow CreatePredictionArray([M])$
  $act \leftarrow SelectAnAction([PA])$
  $[A] \leftarrow CreateActionSet([M], act)$
  $R \leftarrow ExecutingActionOnENV(act)$
  $[A] \leftarrow UpdateSet([A], R)$
  $[\Delta] \leftarrow RuleDiscovery([A],[\Delta])$
  while SizeOf(Δ) $>$ Φ **do**
    DeleteStochClassifier(Δ)
  end while
until terminating conditions are not met

the conclusion and future work are presented in Section V.

II. Background

In this section, the base-line XCS model is briefly explained. Then, the GPGPU programming style and the CUDA framework are introduced. Lastly, we review studies that are related to our work.

A. eXtended Classifier System

The eXtended Classifier System, or XCS [2] in short, is built based on the original learning classifier system (CS-1) introduced by Holland [4]. The XCS learning process is an evolutionary process, which includes the following steps: At each time step, the classifier system receives a problem instance – this input is in the form of a vector of features – which requires a decision (called action) to be performed next. A *match set* $[M]$ is created consisting of rules (classifiers) that can be "triggered" by the given data instance. A covering operator is used to create new matching classifiers when $[M]$ is empty. A prediction array $[PA]$ is constructed for $[M]$ that contains an estimation of the corresponding rewards for each of the possible actions.

Based on the values in the prediction array, an action, act, is selected. Those classifiers that support the predicted action built the *Action Set* $[A]$ (see Algorithm 1).

In response to act, the reinforcement mechanism is invoked and the prediction (p), prediction error (ϵ), accuracy (k), and fitness (F) of the classifiers are updated. The corresponding numerical reward is distributed to the rules accountable for it so as to improve the estimates of the action values.

This matching process is the limiting factor for the scalability of XCS. The overall execution time of the matching process is of linear order of the feature size and population size. Hence, for high-dimensional data sets (e.g., data sets with tens of thousands of features) XCS can be inconveniently slow.

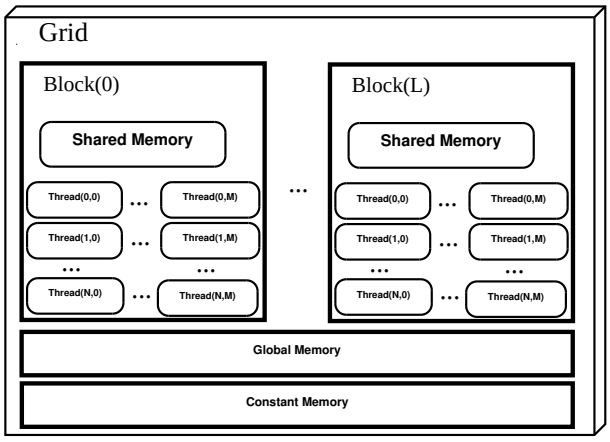


Fig. 1. A typical modern GPU architecture.

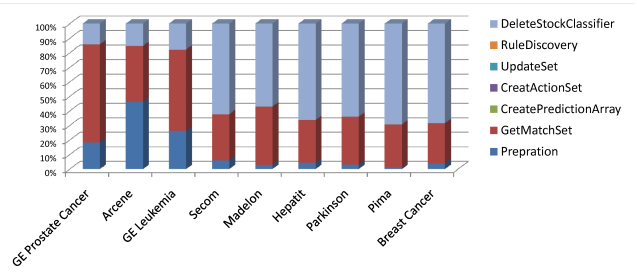
1) Real Value Encoding: The condition of classifiers in the freely available XCS implementations, such as XCS-C by Butz [6], was described by a character based ternary representation for binary inputs $\{0, 1,$ and # for Don't Care$\}$. Later, Wilson et al. [7] revised the XCS and proposed XCSR for getting real input values. In this model, each condition covers a real interval predicate, defined by two real values ($Centre, Spread$): a center point and a spread range around it to describe a range of accepted values between $Centre \pm Spread$. Although our proposed method is implemented as an XCSR version, it can be easily extended to any other XCS-based models or even, with a little more effort, to other GBMLs, as the strategy we propose to parallelize the matching process is independent of the rest of the learning algorithm, including the encoding approach.

2) XCS Execution Profile: The profiling analysis of the freely available XCS implementations (xcslib by Pier Luca Lanzi and XCS-C by Martin Butz) suggests that 65%- 85% of the overall XCS runtime are spent on the rule matching phase [3]. That is, the process of finding whether any rule in the population can match the input values or not.

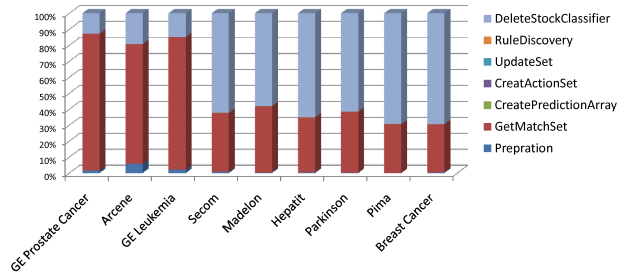
In this paper, we use a modified version of Butz' XCSR implementation. In order to avoid data structure conversion between CPU and GPU, we have replaced the linked-list based population with an array-based population (see Section III-B for more details). We then examine the execution profile of our XCSR version. Our XCSR has been analyzed over

Algorithm 2 The XCS GetMatchSet() function

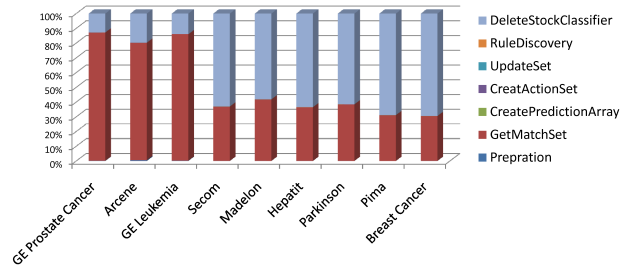
Require: Input data:σ, Population:Δ, MaxPopulationSize:Φ
  for $\forall cl \in \Delta$ **do**
    if match(cl,σ) **then**
      $M \leftarrow M$ + cl
    end if
  end for
  if $M = \emptyset$ **then**
    $M \leftarrow$ createNewCoverMatch(σ)
  end if



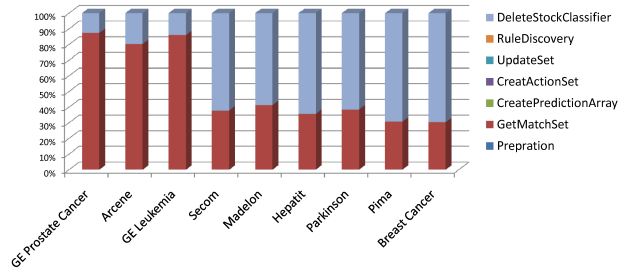
(a) after 1 iteration



(b) after 10 iterations



(c) after 100 iterations



(d) after 1000 iterations

Fig. 2. The execution profile of XCS (a) after 1 iteration, (b) after 10 iterations, (c) after 100 iterations, and (d) after 1000 iterations. These figures support the statement that in a long run, GetMatchSet() and DeleteStockClassifier() functions are the high computing demanding processes among XCS components.

nine different real data sets with different characteristics (see Section III-A).

The execution profile of our XCSR reveals that the functions GetMatchSet() and DeleteStockClassifier() are the most computational demanding processes in XCS. For large-scale data sets, initially the preparation phase, which includes reading the data sets and converting the data into appropriate data structures, is spending considerable amount of time as well – but it runs only once. Eventually, after a few iterations, GetMatchSet() is spending more than 70% of the XCS execution time.

For small-size data sets most of the execution time (up to 65%) is spent by the function DeleteClassifier() to select a classifier and delete it in order to maintain the bounded population size. This is another piece of evidence showing that high-dimensional data sets (with thousands of features) will force XCS to create large classifiers, which leads to a highly expensive rule matching function. The GetMatchSet() function has more potential for massive parallelization than DeleteStockClassifier(). In this paper (as well as most of the related work) the focus is on improving the matching process.

B. General-Purpose Graphic Processing Units

The underlying architecture of GPUs allows them to do floating point calculation at least ten times faster than current multi-core CPUs [5]. They provide massively parallel threads to compute in a Single Instruction on Multiple Data (SIMD) fashion. The sophisticated control logic unit and integrated cache have been replaced by hundreds of additional Arithmetic and Logic Units (ALUs). Therefore, once the data is loaded, hundreds of independent threads can run simultaneously.

The threads are organized into *Blocks* and *Grids*. A Grid contains multiple Blocks. Each Block is handled by a single multiprocessor. The threads inside a Block share a common memory called *Shared Memory*. A couple of very fast 32-bit local *Registers* are also provided for each thread. Each GPU has an integrated memory (*Device Memory*). The data should be transferred from the main memory (*Host Memory*) to the Device Memory.

1) CUDA Framework: NVIDIA introduced the Compute Unified Device Architecture (CUDA) as an extension to C, which allows developers to move intensive computing processes to the GPGPU. The piece of codes run by GPU threads are called *Kernel Functions*.

CUDA library provides instructions to allocate Device Memory, copy data into Device Memory, define the Kernel Functions, and retrieve the results back to the Host Memory. To improve the memory latency operations between Host and Device Memory, CUDA provides two additional memory allocation modes: *Page-locked Host Memory*, also called *Pinned memory*, and *Zero-copy Host Memory* modes. The former prevents the operating system to page the allocated memory to the hard disk, which allows the GPU to use Direct Memory Access (DMA) to speed up the memory operations. In the later mode, a new feature added to CUDA 2.2, the GPU can physically access the Host Memory, which will prevent memory latency operations.

In this work, we have used the Zero-copy Host Memory mode, where both CPU and GPU can access the same physical memory allocated for the XCS population. This is an essential feature since, in our design, both CPU and GPU matching functions are working on the same population.

2) Performance Issues: Understanding of CUDA and hardware architecture is important when designing efficient kernel functions. As we will see later, following the CUDA guidelines would improve the performance of the system.

According to NVIDIA's best practice guidelines [8], data transfer between Host and Device Memory is expensive. Therefore, it is essential to avoid frequent transferring data between kernel and the main algorithm. Also, due to cumulative data transfer overheads, it is better to run one large transfer instead of many small ones. Another important issue is the memory access pattern used by the kernel functions. It is fastest when k GPU threads access k words in the memory, all sequentially aligned. Strided and misaligned access will effectively decrease the performance.

As mentioned in Section II-B, kernel functions can access different types of memories, here ordered from the fastest to the slowest: Register, Local Memory, Shared Memory, Constant Memory, and Global Memory. Using registers leads to zero latency between the kernel instructions and accessing the global memory will cause a major delay.

The other major concern in designing kernel functions is avoiding any flow control instructions, such as *if*, *switch*, *for*, or *while*. Each flow control commands will cause branching and divergence of the execution flow and will force the GPU to turn the parallel execution plan into serial execution. Therefore, sometimes one extra condition (maybe used to detect early termination) can lead to a huge delay in the overall execution.

C. Related Work

Pioneer work in the area of speeding up classifier systems was conducted by Llorà and Sastry [3]. They proposed using an efficient condition encoding and a fast rule matching process based on CPU-implemented vector instruction set. The standard character-based ternary condition alphabet, widely used to construct classifiers for binary input values, does not efficiently use the allocated memory. They used a compact bit-set representation, which can save 75% of the allocated memory; because a character is eight bits long while only two bits are required for bit-set encoding. They also took the advantage of hardware-integrated 32-bit vector instructions to provide a small degree of parallelization. More precisely, four binary conditions can be checked at the same time. Their experimental results on the Multiplexer test bed problems showed an up to 90 times faster matching process.

Lanzi and Loiacono [5] used the CUDA library to revise the XCS matching function. Their experimental analysis of the independent matching process suggested a speed-up by a factor of 20 to 50 for the ternary-based XCRS matching function. Later, Loiacono [9] extended the work and implemented computing a prediction array that is calculated on GPGPU units by using the CUDA library. He reported a 2 to 32 times speed-up of the new CUDA-based prediction array function. This high speed-up level was achieved because the matching process runtimes were compared independently to the other functions in the learning system. Yet, the XCS execution profile suggests that the matching process only occupies for 65% to 80% of the whole process [3]. Moreover, memory operations latency between Device Memory and Host Memory effectively increases the runtime.

In all these studies, only parts of the whole learning system have been analyzed and reported. In recent work, Franco et al. [10] enhanced BioHEL – a Pittsburgh-style learning classifier system – fitness computation process by using the CUDA library. They analyzed the proposed model independently and incorporated it in the learning system and reported a 52-58 times faster learning method. 11 data sets with different characteristics were selected for their experiments. In their model, two separate kernel functions are implemented – one for conducting the matching process and the other for gathering the results of the first kernel and making the final conclusion in a parallel reduction manner.

TABLE I
A SUMMARY OF THE DATA SETS USED IN OUR EXPERIMENTS. † DATA SETS BELONG TO THE UCI REPOSITORY.

Data Set	number of features	number of samples
GE Prostate Cancer	12600	136
Arcene †	10000	200
GE Leukemia	7129	72
madelon †	500	2600
parkinson †	23	197
Hepatitis †	19	155
Pima †	8	768
Breast Cancer †	9	286

III. Design of Experiments

To establish the hybrid CPU/GPU based matching process idea, we have designed a series of experiments on some real-world data sets. Although only the XCS matching process has been enhanced, we have not analyzed the matching process separately but the whole of XCS. The values reported in the result section reflect the mean execution time over 30 trials, each time over 100 iterations, for every single data set.

A. Data Sets

In our experiments, data sets with different characteristics have been selected: those with a small number of features to more than 100000 and from a small number of samples to a large number of samples (see Table I). The data sets marked with † stem from the UCI repository [11], a very well-known freely available machine learning data set repository. We also include two Gene-Expression (GE) data sets: GE Prostate Cancer [12] and GE Leukemia [13]. GE profiling using DNA microarrays allows us to analyze multiple gene markers simultaneously. Consequently, it is a convenient test to identify abnormal biological processes such as cancer [14]. However, from our perspective, these data sets represent an important real-world application for analyzing high-dimensional data sets.

B. CPU/GPU Hybrid Models

The original GetMatchSet() function of XCS has been presented in Algorithm 2. In our experiments, we developed two

Algorithm 3 CUDA-enabled GetMatchSet(): execute either of matching models

Require: Input data:σ, Classifier:cl, Condition Length:τ, Population:Δ, PopulationSize:φ, Maximum thread number in a block:MAXThreads
cudaMemcpy(Δ, cudaMemcpyHostToDevice)
cudaMemcpy(σ, cudaMemcpyHostToDevice)
if mode = 1 **then**
NBlocks $\leftarrow \varphi /$ MAXThreads
cuMatchBitModel1 $\lll$ NBlocks, MAXThreads $\ggg$ (Δ,σ)
else
dim3 dimBlock(MAXThreads,$\tau /$ MAXThreads)
dim3 dimGrid(φ)
cuMatchBitModel2 $\lll$ dimGrid , dimBlock $\ggg$ (Δ,σ)
end if

Algorithm 4 First CUDA-enabled GetMatchSet() kernel: Each thread assigned to check matching of one classifier

Require: Input data:σ, Classifier:cl, Condition Length:τ, Population:Δ, PopulationSize:φ
iThread $\leftarrow$ blockIdx.x $\times$ blockDim.x + threadIdx.x
for iCond = 0 to τ **do**
if Δ[iThread].condition[iCond] $\neq$ # **then**
if σ[iThread] $\leq \Delta$[iThread].condition[iCond].min OR σ[iThread] $\geq \Delta$[iThread].condition[iCond].max **then**
return
end if
end if
iCond++
end for
[M] $\leftarrow$ [M] + cl

different CUDA-enabled GetMatchSet() functions. To execute a CUDA-enabled function, the data should be copied to the Device Memory (GPU internal memory) and call a *kernel* method. The execution parameters of a kernel method are defined by the size of Grid and the size of each Block (as $\lll$ dimGrid, dimBlock $\ggg$). The following Algorithm 3 prepares data and executes one of our two models. The results should be retrieved from the Device Memory to the Host Memory – even for integrated GPUs that basically share a physical memory with the CPU. From CUDA version 2.2 on, GPU threads have the ability to read and write directly on the Host Memory via the *zero copy* memory allocation mode. This feature eliminates the data transfer latency and enables the GPU to access bigger-sized data.

The matching process calculations are split between GPU and CPU (based on the GPU load parameter). For example, if the GPU load is 70%, then 70% of the macro classifiers in the population are checked by the GPU and the remaining 30% are checked by the CPU. The list of matched classifiers is combined at the end. The following subsections describe our two different strategies to use GPGPU threads for finding the matching rules.

1) Task-Oriented Parallelization: In the first model, we designed a very traditional task-oriented parallelization approach, which emphasizes on using multiple threads each for one task. Each thread is designated to matching one rule (classifier) to the input values. This model violates many CUDA best practice guidelines. In other words, both the memory access strategy and simple execution flow are poorly considered. Both loops and condition statements create divergence, which significantly increase the execution time.

2) Data-Oriented Parallelization: In the second model, we used a guideline-conform way to design the kernel function, this time based on a data parallelization approach. Each thread is designated to check only one condition on a data value, while each block is responsible to check one classifier and investigate if that rule matches the input values (a row of values) or not. Although the number of threads in a block is physically limited, fortunately, ordinary GPUs provide tens of thousands physical threads.

We also used a reduction approach to eliminate loops by using a shared flag. Initially, the flag is set; if any condition in a classifier is violated, then the flag would be reset, which means that the classifier is not matched with data anymore. If all conditions matched with the input value, the flag retains the initial value. At the end, by checking the flag, it becomes clear if the rule (classifier) is matched or not. This technique eliminates the need of collecting and summarizing the result of all threads, which saves a lot of GPU execution time.

C. Hardware and Software

The proposed models in this paper were implemented on the latest CUDA library (version 4.0), and were deployed and tested on two major types of NVIDIA Graphic cards (GeForce and Tesla) over five different hardware configurations, which include two of different GPU architectures on *VPAC*[1] Tango

[1]Victorian Partnership for Advanced Computing: vpac.org

Algorithm 5 Second CUDA-enabled GetMatchSet() kernel: Each block for matching of one classifier and each thread within that block is assigned to a condition of the related classifier

Require: Input data:σ, Population:[Δ]
$\_\_shared\_\_$ $matched \leftarrow 1$
$iBlock \leftarrow blockIdx.x$
$iThread \leftarrow threadIdx.y \times blockDim.x + threadIdx.x$
$cond \leftarrow \Delta[iBlock].condition[iThread]$
checkRes $\leftarrow cond = \#$ OR ($cond.min \geq \sigma[iThread] \leq cond.max$)
if checkRes = 0 **then**
matched $\leftarrow 0$
end if
if iThread = 0 AND checkRes = 1 **then**
$M \leftarrow M + cl$
end if

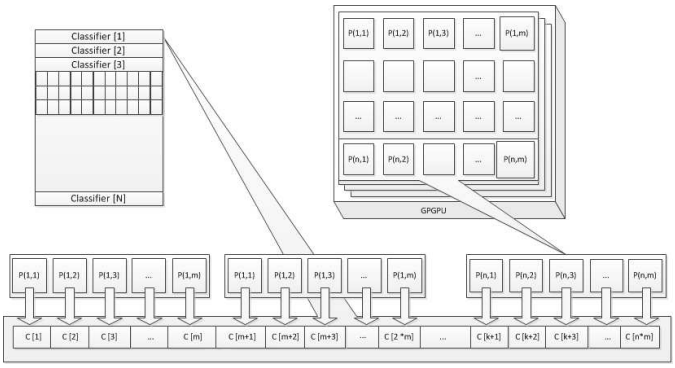


Fig. 3. XCS GetMatchSet() on CUDA.

Cluster server, GPU units on *MASSIVE*[2] cluster server and two desktop PCs equipped with low budget NVIDIA GPU cards:

1) Tango server with one Enrico-based node with a 2.27 GHz quad-core Nehalem server equipped with 12GB RAM and two of the latest NVIDIA Fermi architecture Tesla C2050.
2) Tango server with a small 4-node based GPU partition, each one has a Tesla 1060, 1 quad-core processor with 4GB of RAM and a single 320GB hard drive.
3) Massive cluster server with 1008 CPU-cores and 168 GPUs. Each node has a 12-core CPU and 48 GB RAM. GPU nodes are equipped with 2 NVIDIA M2070 GPUs with 6GB GDDR5.
4) One desktop PC with Intel Core 2 Due Centrino 2.5 GHz, 4GB memory, and a NVIDIA GeForce 8400M GS.
5) One desktop PC with Intel Core i7 1.60 GHz, 6GB memory, and a NVIDIA GeForce GT 330M GPU.

The operating system of the Tango and Massive servers is CentOS 5.6, a Linux server edition. Both desktop PCs use Ubuntu 10.04 LTS, a Linux desktop edition.

IV. Results

The results of running two different GPGPU-CPU hybrid models, realized into two different kernel functions, are shown in Figures 5 and 6 respectively. In both figures, the speed-ups of the corresponding model are presented based on the GPU load. For example, in Figure 5, the first graph depicts the fact that only in two cases the task-oriented model has demonstrated close to 2X speed-up; in three cases, the model has not only increased the execution time (negative speed-up) but also that by increasing the GPU load the overall performance degrades. This figure also suggests that the task-oriented model performs poorly in high-dimensional data sets, but for small-size data sets it could provide a competitive learning system. Also, because the desktop hardware using GeForce330M GS possesses a fairly powerful CPU, increasing the GPU loads, in most cases, actually decreases the overall speed. However, this trend is not followed by the server-based devices, which pose fairly powerful GPUs and less powerful

[2]The Multi-modal Australian ScienceS Imaging and Visualization Environment: massive.org.au

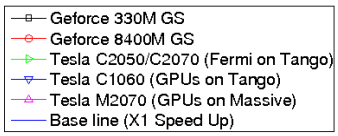


Fig. 4. Legend for Figures 5 and 6.

CPUs. Overall, in this model shifting the computing load from CPUs to GPUs does not effectively increase the speed performance.

Figure 6 for the data-oriented parallelization model presents more encouraging results. The graphs show an effective GPU/CPU distribution trend. Basically, by increasing the GPU load, in the majority of cases, the execution time decreases significantly, hence a speed-up of up to 18 times could be reached. It also appears that in all scenarios, a higher (90%) GPU load would result in a faster learning system. However, if we shift all the computation burden to GPU (100%), then we will lose a significant portion of the leverage of using the CUDA-enabled matching process.

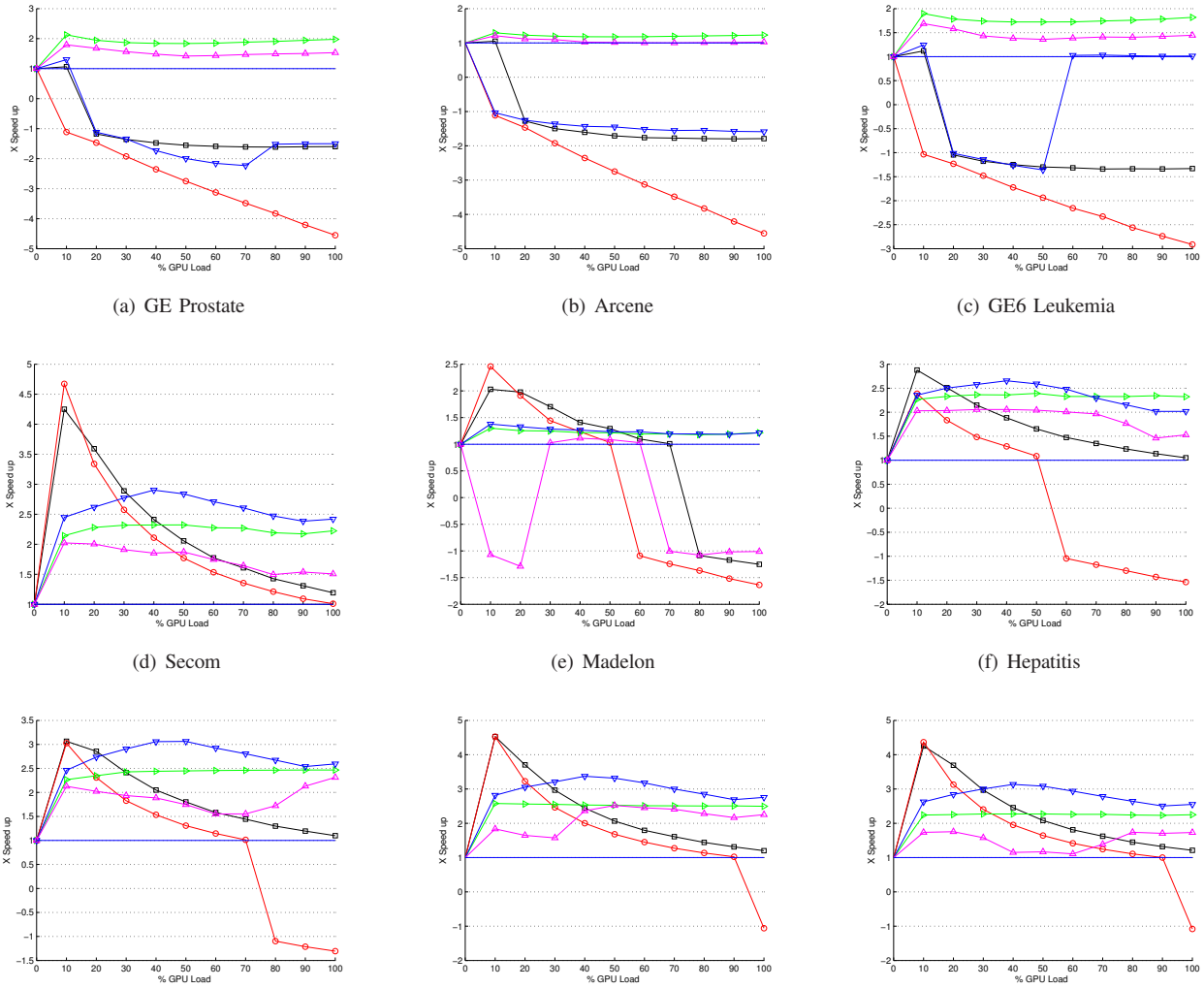
The graphs also emphasize on the main aspect of the GPGPU technology: Highly massive parallel computing capacity. It appears that in all cases, whether high-dimensional data sets with tens of thousands of features or small-scale data sets with few features, shifting the matching process to the GPU will increase the speed. Still, the XCS matching process would benefit most from using GPU computing for high-dimensional data sets. The GPU's massively parallel architecture can easily absorb the increasing resource demands, which is a threat for CPU-only models. However, for the data sets with a large number of samples, our proposed models do not provide significant improvements. For example, the Madelon data set has 500 features and 2600 samples and the data-oriented model can only slightly improve the matching speed.

The second proposed model (based on a data parallelization approach) was focused on using the massive parallel GPU thread capacity to handle the thousands of features, and iterative policy to check all samples. Therefore, it is safe to conclude that this model does not provide any leverage for large sample size data sets.

We should be aware that the graphs are in speed-up scale. In other words, it is not visible that in Figure 6's GE Prostate case, the desktop PC with GeForce 8400M GS video card could finish the experiments three times faster than cluster servers. An interesting observation reveals that, although the second PC with GeForce 330M GS has i7 cores CPU, the first PC could beat the second one, when only the CPU was used for the matching process, because it has a faster CPU. But, gradually the second PC could fill the gap because it has a more powerful GPGPU.

V. Conclusion and Future Work

In this paper, we have proposed two hybrid GPGPU-CPU based rule matching processes to improve XCS's runtime



(a) GE Prostate (b) Arcene (c) GE6 Leukemia

(d) Secom (e) Madelon (f) Hepatitis

(g) Parkinson (h) Pima (i) Breast Cancer

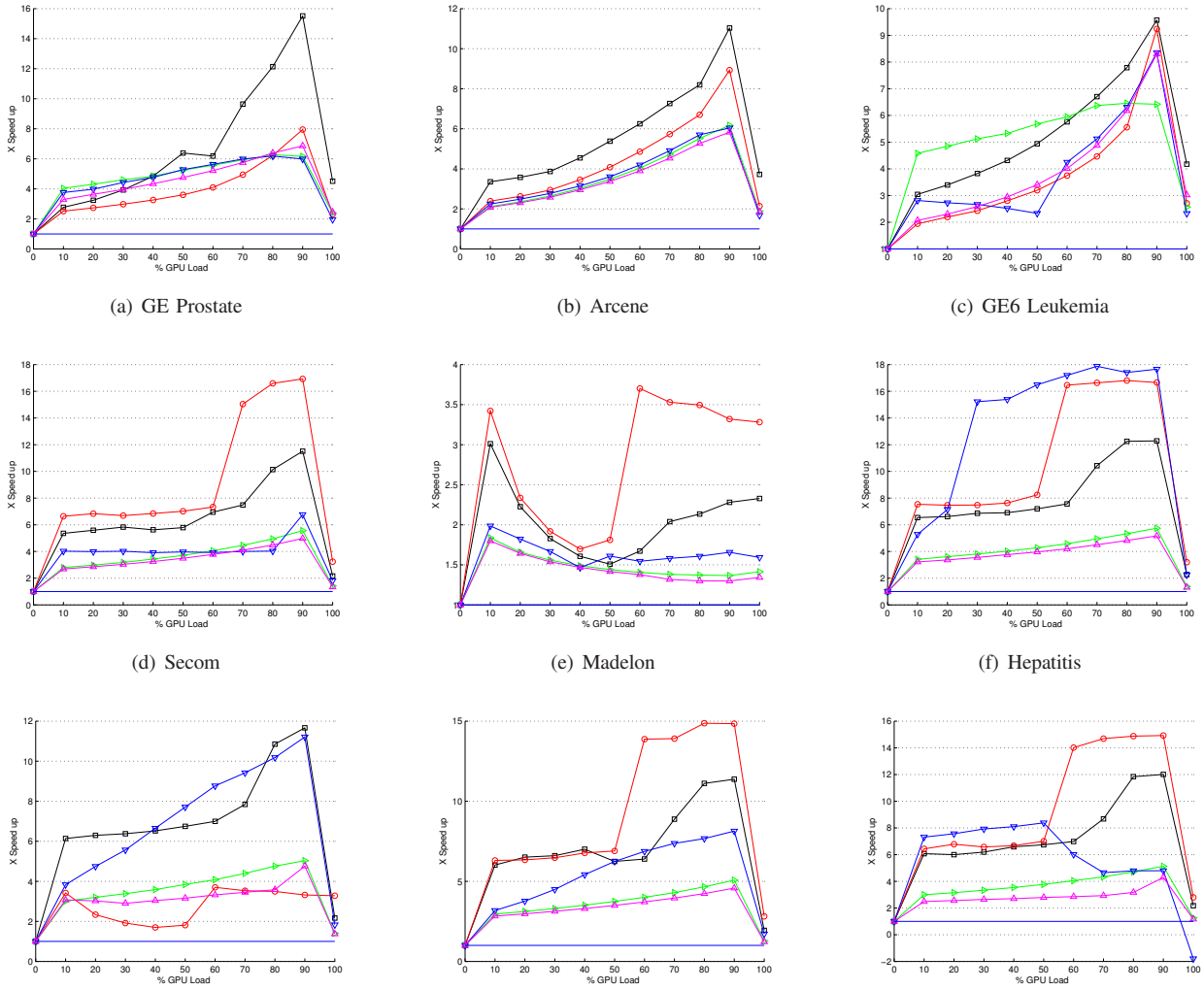
Fig. 5. Experimental results of the gained speed-up by implementing the first (task-oriented) model. The graphs show the increase/decrease in execution time (in speed-up scale) by changing the GPU loads. Each sub-graph presents the results of using one data set for the five different platforms. For example, Figure (a) depicts the experiments of GE prostate cancer (a high-dimensional data set). It shows that the model can get a very low level of speed-up when run on two of the GPU servers, and on other hardware the model performs poorly. Figure (i) demonstrates a consistent and similar behavior of the model over all platforms on the Breast Cancer (a very low-dimensional data set). Legend: see Figure 4.

especially for high-dimensional problems (having a huge number of features). We employed the massively parallel processing capability of a GPGPU to handle a massive number of conditions simultaneously. The XCS execution profiles recorded in this paper reveal that the matching function is a highly demanding computing component of XCS. Therefore, it has a great potential to improve the speed of XCS if we can boost the speed of the matching process. We worked on the original XCSR implementation and modified the GetMatchSet() function to be capable of using both CPU and GPGPU as computing resources. Two different CUDA-based rule matching functions are proposed, which can run part of the matching process on a GPGPU and the rest on the CPU. The second model, based on a data parallelization strategy, has demonstrated outstanding acceleration in the matching process (a speed-up to a factor of 18). This model also depicts the advantage of using GPGPU-enabled models for high-dimensional data sets. It follows the best practice guidelines for CUDA utilization to the word. The first, a task-based approach, on the other hand, does not fully consider these best practices and actually performs much worse.

For future work, we plan to extend our model for data sets with a large number of samples. Using multi-GPU nodes may also improve the speed performance. Nowadays, GPU servers have more than one GPU installed on each node. Therefore, we can easily distribute the GPU loads between two or more GPUs – CUDA supports multi-GPU development.

ACKNOWLEDGEMENTS

We would like to acknowledge VPAC and MASSIVE for supporting this work and granting us the access to their



(a) GE Prostate (b) Arcene (c) GE6 Leukemia

(d) Secom (e) Madelon (f) Hepatitis

(g) Parkinson (h) Pima (i) Breast Cancer

Fig. 6. Experimental results of the gained speed-up by implementing the second (data-oriented) model (the one with the better design). The graphs show the increase/decrease in execution time (in speed-up scale) by changing the GPU loads. Each sub-graph presents the results of using one data set on five different platforms. For example, Figure (a) depicts the experiments of GE prostate cancer (a high-dimensional data set). It shows that the model can speed up the learning process by increasing the GPU load, on all platforms. Figure (i) demonstrates a consistent and similar behavior of the model over all platforms on the Breast Cancer (a very low-dimensional data set). Legend: see Figure 4.

HPC/GPU cluster servers. The resource grant on MASSIVE was dedicated to the project with ID pMelb0103.

REFERENCES

[1] M. Abedini, M. Kirley, R. Chiong, and T. Weise, "GPU Accelerated eXtended Classifier System," in *Proceedings of the 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM'13)*. Singapore: Grand Copthorne Waterfront Hotel: Los Alamitos, CA, USA: IEEE Computer Society Press, Apr. 16–19, 2013, pp. 293–300.

[2] S. W. Wilson, "Classifier Fitness Based on Accuracy," *Evolutionary Computation*, vol. 3, no. 2, pp. 149–175, 1995.

[3] X. Llorà and K. Sastry, "Fast rule matching for learning classifier systems via vector instructions," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '06. New York, NY, USA: ACM, 2006, pp. 1513–1520.

[4] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, April 1992.

[5] P. Lanzi and D. Loiacono, "Speeding up matching in learning classifier systems using cuda," in *Learning Classifier Systems*, ser. Lecture Notes in Computer Science, J. Bacardit, W. Browne, J. Drugowitsch, E. Bernad?-Mansilla, and M. Butz, Eds. Springer Berlin / Heidelberg, 2010, vol. 6471, pp. 1–20.

[6] M. V. Butz and S. W. Wilson, "An algorithmic description of XCS," *Soft Computing*, vol. 6, no. 3–4, pp. 144–153, 2002.

[7] S. W. Wilson, "Get Real! XCS with Continuous-Valued Inputs," in *Learning Classifier Systems, From Foundations to Applications*, ser. Lecture Notes in Computer Science, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds., vol. 1813. Springer, 1999, pp. 209–222.

[8] Nvidia, "Nvidia cuda c programming best practices guide cuda toolkit 2.3," *Optimization*, no. July, 2009.

[9] D. Loiacono, "Fast prediction computation in learning classifier systems using cuda," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (Companion Material)*, ser. GECCO '11, N. Krasnogor and P. L. Lanzi, Eds. New York, NY, USA: ACM, 2011, pp. 169–170.

[10] M. A. Franco, N. Krasnogor, and J. Bacardit, "Speeding up the evaluation of evolutionary learning systems using gpgpus," in *Proceedings of*

the 12th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 1039–1046.
[11] "UCI Machine Learning Repository," http://archive.ics.uci.edu/ml/.
[12] D. Singh, P. G. Febbo, K. Ross, D. G. Jackson, J. Manola, C. Ladd, P. Tamayo, and A. A. Renshaw, "Gene expression correlates of clinical prostate cancer behavior," *Cancer Cell*, vol. 1, pp. 203–209, 2002.
[13] T. R. Golub, D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, M. L. Loh, J. R. Downing, M. A. Caligiuri, and C. D. Bloomfield, "Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring," *Science*, vol. 286, pp. 531–537, 1999.
[14] Y. Zhang and J. C. Rajapakse, *Machine Learning in Bioinformatics*, ser. Wiley Series in Bioinformatics, 2008.

```
@inproceedings{AKCW2013GAECS,
  author    = {Mani Abedini and Michael Kirley and Raymond Chiong and
               Thomas Weise},
  title     = {{GPU Accelerated eXtended Classifier System}},
  booktitle = {Proceedings of the 2013 IEEE Symposium on Computational
               Intelligence and Data Mining (CIDM'13)},
  publisher = {Los Alamitos, CA, USA: IEEE Computer Society Press},
  address   = {Singapore: Grand Copthorne Waterfront Hotel},
  pages     = {293--300},
  year      = {2013},
  month     = apr # {~16--19, },
  doi       = {10.1109/CIDM.2013.6597250},
  eiid      = {20134316873557},
  inspec    = {13768347},
},
```