A Portable OpenCL-based Approach for SVMs in GPU

Henry E.L. Cagnini[*], Ana T. Winck[†], Rodrigo C. Barros[*]
[*]Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, Brazil
Email: henry.cagnini@acad.pucrs.br, rodrigo.barros@pucrs.br
[†]Universidade Federal de Santa Maria, Santa Maria, RS, Brazil
Email: ana@inf.ufsm.br

***Abstract*—Support Vector Machines (SVMs) is one of the most efficient methods for data classification in machine learning. Several efforts were dedicated towards improving its performance through source-code parallelization, particularly within the Graphics Processor Unit (GPU). Those studies make use of the well-known CUDA framework, which is provided by NVIDIA for its graphics cards. Nevertheless, the main disadvantage of CUDA-based solutions is that they are specific to NVIDIA cards, reducing the applicability of such solutions in heterogeneous environments. In this work, we propose the parallelization of SVMs through the OpenCL framework, which allows the generated solution to be portable to a wide range of GPU manufacturers. The proposed approach parallelizes the most costly steps that are performed when training SVMs. We show that the proposed solution achieves a significant speedup regarding the algorithm's original version, and also that it outperforms the state-of-the-art CUDA-based approach in terms of computational performance in 11 out of the 12 datasets that were tested in this work.**

I. Introduction

One of the most studied tasks within the machine learning literature is data classification, which can be roughly seen as the task of labeling objects. Given a set $\mathbf{X}$ of N objects (examples, instances), a classification algorithm seeks to create a model that properly represents the relationship among d predictive attributes (features) of such objects (i.e., $\{\mathbf{x}_i\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^d$) and their corresponding labels ($\{y_c\}_{c=1}^{k} \in Y$). Such a model should be capable of automatically predicting the label of unseen objects based solely on their known attribute values.

One can see classification as an optimization problem, where the goal is to find a function $\hat{f}$ that reasonably approximates the unknown true function f responsible for mapping attribute values into labels, $f : \mathbf{X} \to Y$. Nowadays, one of the most widely-used machine learning algorithms for classification are the Support Vector Machines (SVMs) [1], especially due to their effectiveness in solving complex high-dimensional classification problems.

Notwithstanding, SVMs are computationally expensive, especially due to the fact that they must compute mathematical operations between every pair of training objects. Therefore, as the size of the dataset increases, the computational resources required to run SVMs over the given problem also increase, eventually becoming unfeasible. Considering that most current interesting problems can demand several hours/days for the sequential SVMs to be executed, one must look for novel computational approaches to run these complex machine learning algorithms, one of them being their parallelization within the Graphics Processor Unit (GPU).

A number of studies parallelize SVMs in GPU through the well-known CUDA framework [2], achieving considerably better computational performance than the original sequential version of the algorithm. However, CUDA-based approaches lack the capability of running in a wider range of GPUs, being strictly restricted to NVIDIA cards. One framework that does not suffer from this lack of portability is OpenCL [3], which is capable of running in several hardware architectures, such as GPUs, CPUs, APUs (a GPU that shares area in the processor chip along the CPU cores) and even mobile microprocessors [4]. Recent surveys conducted by specialized consultants [5] and online videogames distributors [6] indicate that the number of GPUs from other manufacturers than NVIDIA – and hence only capable of running OpenCL-based approaches – is significantly larger than the number of NVIDIA GPUs. In a world that currently relies on cloud platforms that usually provide heterogeneous environments, there is a clear necessity of developing portable approaches for scaling machine learning algorithms.

In this work, we propose to improve the efficiency of binary classification tasks through the parallelization of SVM in GPU, with the final goal of improving the computational efficiency of SVMs in a portable fashion. For such, we parallelize the well-known LIBSVM library [7] using the previously-mentioned OpenCL framework. As an additional contribution, the source code of our approach is made fully available[1]. For comparison purposes, we evaluate the performance of our proposed approach with regard to the original sequential version of the SVMs and also to the state-of-the-art CUDA-based parallelization algorithm [8].

This paper is organized as follows. Section II presents a brief background on SVMs, whereas Section III discusses related work. Section IV describes our proposed approach for parallelizing SVMs within the GPU, and Section V details the empirical analysis we conducted to validate its performance. Finally, we end this paper with our conclusions on the matter and we point to some interesting future work directions in Section VI.

[1]Omitted due to blind review.

978-1-5090-0016-6/15 $31.00 © 2015 IEEE
DOI 10.1109/BRACIS.2015.27

II. Background

In this section, we briefly explain how SVMs work while also providing our motivation for the choice of this particular machine learning algorithm for its further parallelization in GPUs.

A. Support Vector Machines

Support Vector Machines (SVMs) is a machine learning algorithm introduced by Vapnik and Cortes [9] based on the Statistical Learning Theory [1]. SVMs belong to the class of algorithms that generate the so-called geometric models, which assume the objects of a given dataset are located in a d-dimensional cartesian space, and the ultimate goal is to find a large-margin separating hyperplane that is capable of labeling novel objects with respect to predefined categories.

SVMs are presented as follows. The algorithm assumes each attribute to be a cartesian axis in the feature space, and hence each object can be regarded as a point in this $\mathbb{R}^d$ multi-dimensional space. SVMs then try to linearly separate the data with $g(\mathbf{x})$:

$$g(\mathbf{x}) = \begin{cases} +1 \text{ if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ -1 \text{ if } \mathbf{w} \cdot \mathbf{x} + b < 0 \end{cases} \tag{1}$$

where $\mathbf{w}$ is the hyperplane's normal vector, $\mathbf{x}$ is a training set instance, and $b/||\mathbf{w}||$ the distance to the origin, with $b \in \mathbb{R}$. There are, however, an infinite number of hyperplanes that can linearly separate the classes by multiplying $\mathbf{w}$ and b by the same constant. The canonical hyperplane is the one in which $\mathbf{w}$ and b are chosen to satisfy:

$$|\mathbf{w} \cdot \mathbf{x}_i + b| = 1 \tag{2}$$

this constraints the hyperplane to be one such as

$$\begin{cases} \mathbf{w} \cdot \mathbf{x} + b \geq +1 \text{ if } y_i = +1 \\ \mathbf{w} \cdot \mathbf{x} + b \leq -1 \text{ if } y_i = -1 \end{cases} \tag{3}$$

Note that this effectively leaves a margin between the outermost objects of the positive class (that is, the objects closest to the negative class objects), and vice versa. Objects that define these margins are called support vectors, and the hyperplane is situated between these support vectors. Figure 1 shows a hyperplane between objects from two classes, as well as their corresponding support vectors.

If the problem is not linearly-separable in the original feature space, SVMs are capable of casting the data into higher dimensional spaces where the problem becomes linearly-separable [1], by making use of a function called *kernel*. In this enhanced feature space, the optimization problem can then be seen as the one of maximizing the distance of the hyperplane to the boundary objects from each class, which is equivalent to minimizing half of the squared norm of $\mathbf{w}$:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 \tag{4}$$

with constraints: $y_i(\mathbf{w} \cdot \mathbf{x} + b) - 1 \geq 0, \forall i = 1, ..., N$.

The constraints are defined in order to prevent training instances from locating within the margins. These constraints can be relaxed by adding slack variables ξ to Equation 4:

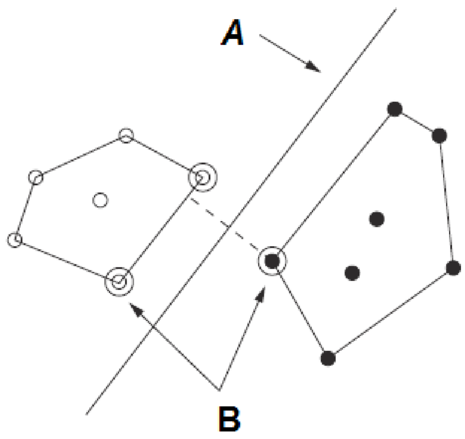

Fig. 1. A hyperplane (A) that maximizes the distance among objects from two different classes (B). The circled objects are the corresponding support vectors. Adapted from [10].

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C \left(\sum_{i=1}^{N} \xi_i \right) \tag{5}$$

where $\sum_{i=1}^{N} \xi_i$ denotes the penalty for misclassifying instances from the training set, and C is a regularization term that seeks for a tradeoff between minimizing the misclassification error and maximizing the hyperplane's margins – the well-known tradeoff between predictive performance and model complexity.

SVMs are also capable of solving multi-class classification problems, where the class attribute $\{\mathbf{C}_j\}_{j=1}^{k}, k > 2$. There are several approaches to deal with multi-class problems, but the most straightforward strategies are the decompositional ones: i) either train $k(k-1)/2$ predictive models, one for each pair of classes; or ii) train k models, considering that any instance that does not belong to the positive class C_j belongs to a hypothetical negative class (formed by objects from the remaining classes) [11]. Following the training of multiple models, it is necessary to merge the results in order to correctly predict the class of the test instances.

B. Motivation for Using SVMs

SVMs have gained space among other machine learning algorithms due to its powerful generalization performance even in high-dimensional problems, making it a suitable method for tasks such as image recognition, bioinformatics, and text mining, just to name a few [12]. Moreover, considering the availability of stable open-source code libraries such as the LIBSVM [7], it has attracted considerable interest from researchers and practitioners alike. We present in Figure 2 the amount of SVM-based scientific publications indexed by Scopus between the years of 1998 and 2013, as a means to illustrate the growing interest on such a technique.

SVMs are available in the widely-used Weka toolkit [10], which is a well-known machine learning and data mining API. When comparing SVMs to other Weka-available classification algorithms, we can notice the growing attention it has received

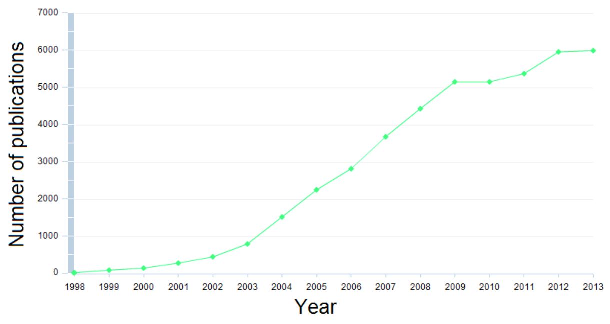

Fig. 2. Number of publications containing the term “Support Vector Machines” between 1998 and 2013 according to the Scopus knowledge database.

in recent years. Figure 3 shows the amount of returned papers when searching for different machine learning algorithms followed by the term ”classification”.

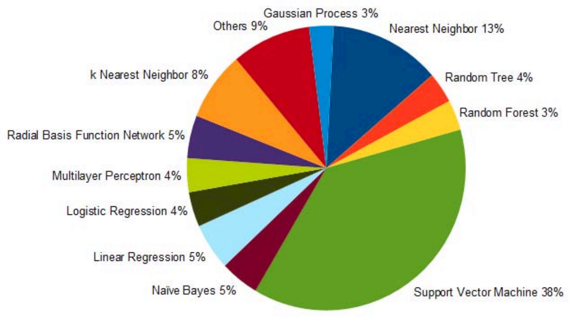

Fig. 3. Classification algorithms available at Weka which were searched within Scopus alongside the term ”classification”. The search was performed in July 2014 and returned 58,279 results. Algorithms with more than 1,000 results are shown separately.

III. Related Work

Many approaches have been proposed in the literature for speeding up the computational performance of SVMs. These approaches vary in aspects such as the source-code library that implements the SVMs and the hardware in which they are exploited. The parallelization strategy used in the work of Lu et al. [13] is to swap support vectors between machines executing the SVM^{light} [14] source code in a strongly connected network of computers. The authors swap support vectors using several network layouts, pointing out that strongly connected ones perform better than others, whilst also speeding up the performance of the sequential SVMs algorithm.

With regard to employing GPUs for speeding up SVMs, Cantarazo and Sundaran [12] propose several approaches to improve LIBSVM [7] performance in terms of efficiency, such as modifying the Sequential Minimal Optimization (SMO) strategy to port it to the GPU. They also perform memory mapping using third party source code to reduce useless computation. Athanasopoulos et al. [8] also make use of CUDA as the GPU framework to speed up SVMs implemented through the LIBSVM library. They propose to pre-compute the kernel matrix to avoid recomputing these values during the cross-validation step. They use the RBF kernel of SVMs to detect high level features of video shots within several hours of videos. Since both the studies of Cantarazo and Sundaran [12] and of Athanasopoulos et al. [8] make use of the CUDA framework [2] to port the source code to the GPU, their solutions are restricted to hardware environments with NVIDIA graphics cards.

IV. Proposed Approach

In this section, we describe how we identified the most computationally expensive functions in LIBSVM’s source code, as well as the viability analysis that was performed to port it to the GPU. We also detail how our proposed approach was actually implemented.

A. *Profiling*

As previously mentioned, we make use of the OpenCL framework to run the source code within the GPU, widening the range of compatible computers. Microsoft Visual Studio 2012 [15] was used used as an IDE for code developing and also as a profiler – i.e., a program that counts the number of function calls. Finally, we used the following computer configuration in all the performed tests: Intel Core i7 4770 processor, 12 GB DDR3 RAM, NVIDIA GTX 750Ti GPU, 128GB SSD secondary storage (in which the datasets were stored) and 1TB HD secondary storage.

The first step of this work aims at identifying the most processor-consuming functions within the SVM source code. We employed the Radial Basis Function (RBF) kernel as the function responsible for creating the novel linearly-separable feature space of the SVMs. We identified that its inner dot product between pairs of dataset objects is the bottleneck in terms of computational resources. The RBF kernel has to compute the dot product between all training instances, and as the size of a given dataset grows, so does the CPU’s share of processing regarding the dot product.

The dot product is perfectly suitable for parallelization, since there is no data dependency between iterations. LIBSVM’s original source code computes the dot product as a double loop over the dataset objects, hence being of O(N^2) complexity, where N is the number of dataset objects.

B. *Implementation*

The proposed computation of the kernel’s dot product is similar to the one available in the original LIBSVM’s source code – to compute it only when requested. The difference between the sequential and parallel implementations is that the GPU computes dot products much faster than a CPU due to its stream processors. The CPU used in this work has 8 cores capable of processing all kinds of instructions. The GPU, in turn, has 640 stream processors, each one being capable of processing only a limited range of instructions, which conveniently include the dot product’s arithmetics.

The implementation makes use of an image, allocated at the GPU memory, to store values of the objects’ attributes. Images are optimized for use in GPUs, since their main purpose is graphics’ processing. Even though GPUs have lately

TABLE I. ARTIFICIAL AND REAL DATASETS USED IN THE PROFILING FOR BOTH ORIGINAL AND PARALLEL VERSIONS OF THE SVMS.

	Dataset	# Attributes	# Objects	Instance / Attribute ratio	Size (bytes)
UCI	adult	123	32,561	264.72	16,020,012
	letter	16	20,000	1.25	1,280,000
	census-income-full	42	299,295	7.13	50,279,880
Artificial	RDG6k10k	10,000	6,000	0.60	240,000,000
	RDG7k10k	10,000	7,000	0.70	280,000,000
	RDG8k10k	10,000	8,000	0.80	320,000,000
	RDG9k10k	10,000	9,000	0.90	360,000,000
	RDG10k10k	10,000	10,000	1.00	400,000,000
	RDG10k6k	6,000	10,000	1.67	240,000,000
	RDG10k7K	7,000	10,000	1.43	280,000,000
	RDG10k8k	8,000	10,000	1.25	320,000,000
	RDG10k9k	9,000	10,000	1.00	360,000,000

evolved into a more generic programming hardware, capable of handling buffers of memory, the choice for coding objects as images is merely due to convenience: OpenCL provides a built-in function that calculates dot products between regions of images. Regardless of the choice for representing objects within the GPU memory, OpenCL limits the size of available memory to approximately half of all physical memory. For the NVIDIA GTX 750Ti GPU, for instance, each image can have up to 16,384 floating-point values wide by 16,384 floating-point values high, totalizing 1GB of memory. We sort out the image space to store the dataset from left to right (regarding the attributes) and up to down (with respect to the objects). The four cases of dataset storage that may arise are shown in Figure 4.

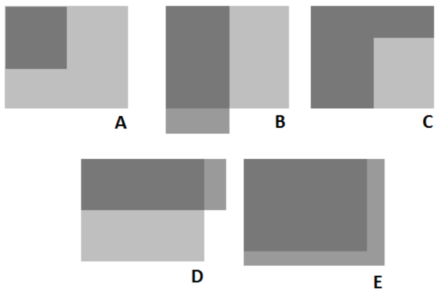

Fig. 4. Four possible situations (A, B, D, and E) for storing a dataset into a GPU's image. In (A), the dataset is smaller than the boundaries of the image, dispensing any treatment. In (B), the dataset has more instances than the image, but as the image has more horizontal space to fit another column of attributes, the overflowed instances are stored in the second column, as shown in (C). Cases (D) – the dataset has more attributes than the GPU has columns to store it – and (E) – dataset has both more attributes and instances than GPU's storage capability – are not supported by the current implementation, and will result in failure.

Due to the fact that the data are encoded as an image, the number of attributes must be a factor of 4, since this is the number of channels of a pixel (red, green, blue, and alpha). If a dataset does not have a compatible number of attributes, we insert dummy attributes with zeros for all objects at the end of the image matrix. It is important to notice that this modification is exclusive to the dataset encoding within the GPU memory. The objects stored in host memory remain unaltered. Additionally, the results of the dot products do not change, which means the augmented feature space does not affect the effectiveness of the SVMs. For instance, the dot product between objects $[2\ 3\ 5]^T$ and $[7\ 11\ 13]^T$ will be the same as the dot product of $[2\ 3\ 5\ 0]^T$ and $[7\ 11\ 13\ 0]^T$, $(2 \times 7) + (3 \times 11) + (5 \times 13) + (0 \times 0) = 112$.

The parallelism is exploited object-wise: several dot products can be calculated at the same time – it depends on the number of threads that the GPU hardware can handle. The dot product is now calculated within a single pass over the dataset. A simplified explanation of the algorithm is presented in Figure 5.

```
1: procedure GPUdotProduct(objIndex, pivotIndex)
2:     sum ← 0
3:     //one pixel is 4 attributes
4:     for i ← 0 until attributeCount do
5:         pixel1 ← image[pivotIndex][i]
6:         pixel2 ← image[objIndex][i]
7:         sum ← sum + dotProduct(pixel1, pixel2)
8:     return sum
```

Fig. 5. Dot product calculation at the GPU. *pivotIndex* is passed to the GPU's parameters to let it know which object is having its dot product currently calculated. *objIndex* may be any number ranging from 0 to the number of the dataset's objects, and is automatically assigned by the GPU.

V. EXPERIMENTAL ANALYSIS

A. Methodology

To assess the performance of the proposed approach over different scenarios, we make use of artificial datasets generated by RDG, a Weka data generator [10], as well as real-world data from the UCI Machine Learning Repository[2] [16]. All datasets used in this analysis represent binary classification problems. The artificial and real-world datasets are presented in Table I.

We executed three versions of the LIBSVM source code: i) the original sequential version; ii) the official LIBSVM's CUDA version, which was detailed in [8] and is available for downloading at the LIBSVM's website[3]; and iii) our proposed approach version[4]. Since OpenCL is a cross-platform framework and we are using an NVIDIA GPU, it is possible to run both CUDA and OpenCL versions in our computer. Each dataset is executed ten times, and the final results are averages computed from these different runs. The results of the tests are presented in the next section.

[2]Available at http://archive.ics.uci.edu/ml/ and http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[3]http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[4]Omitted due to blind review.

B. *Results*

Our proposed approach achieves, as one should expect, the same accuracy (predictive performance) than both the CUDA-based strategy and the original sequential version, considering that we are not modifying the optimization algorithm that is performed by the SVMs.

In terms of computational performance, our approach outperforms the original sequential version and the CUDA-based method in 11 out of the 12 datasets considered in this work. The single case in which our version was outperformed by CUDA has a straightforward explanation: OpenCL presents an overhead not verified in CUDA, which is the fact that the GPU kernel must be compiled in every execution of the program, whereas CUDA compiles it only once. Thus, OpenCL spends some time compiling the kernel on-the-fly, which ends up being the total execution time of the CUDA and sequential versions for the *letter* dataset. Regardless of this overhead, the OpenCL is capable of compensating for larger datasets, eventually performing better than the sequential and CUDA-based implementations. The average of ten runs for each dataset, as well as the standard deviations and relative speedups with regards to the sequential code are presented in Table II.

Although OpenCL requires the kernel to be compiled on-the-fly, it is clear by analyzing the data in Table II, specially for the real-world datasets, that GPU implementations may be outperformed by the sequential version if the dataset is not sufficiently large in bytes. This is due to the cost of transferring the dataset to GPU memory (to compute the dot products) and from it (to get the computed results).

In order to explain why our version also outperforms the Athanasopoulos' CUDA-based version [8], it is necessary to carefully examine the latter's source code. We noticed that the differences are mainly due to the way some optimization strategies were implemented. The key differences are:

1) We calculate the sum of squares of each object (the dot product of an object with itself) also in GPU, instead of CPU;
2) The dataset is loaded to GPU only once, during the reading process. Athanasopoulos's version [8] loads it whenever it is necessary.

We also noticed that, for the largest datasets that were used in this work, the speedup of the CUDA-based version reaches a plateau when it is about 4 times faster than the sequential version, whilst OpenCL reaches a similar plateau only when it is about 36 times faster. We believe this difference is due to the aforementioned differences between versions.

In order to provide more insights on the data presented in Table II, we illustrate the execution time of the artificial datasets when varying the number of objects in Figure 6, whilst in Figure 7 we show the same for the case of varying the number of attributes. Finally, Figure 8 depicts the speedup achieved by both OpenCL and CUDA versions in relation to the sequential original version.

VI. Conclusions and Future Work

In this work, we proposed to improve the efficiency of binary classification tasks through the parallelization of the

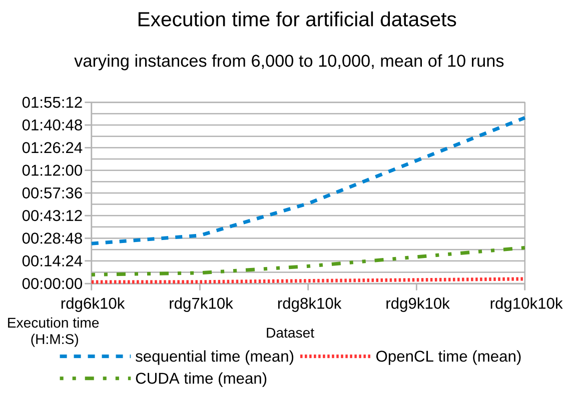

Fig. 6. Execution time for the artificial datasets, when increasing the number of objects. The datasets were ordered in ascending order regarding the number of objects.

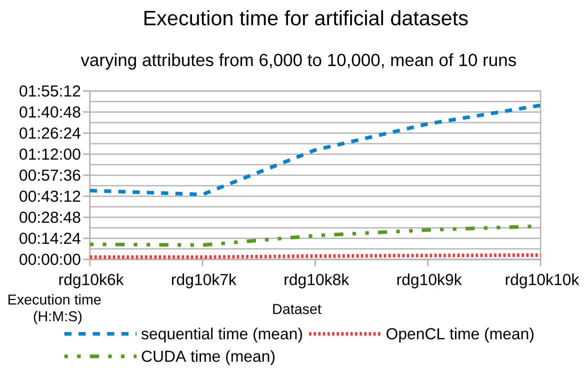

Fig. 7. Execution time for the artificial datasets, when increasing the number of attributes. The datasets were ordered in ascending order regarding the number of attributes.

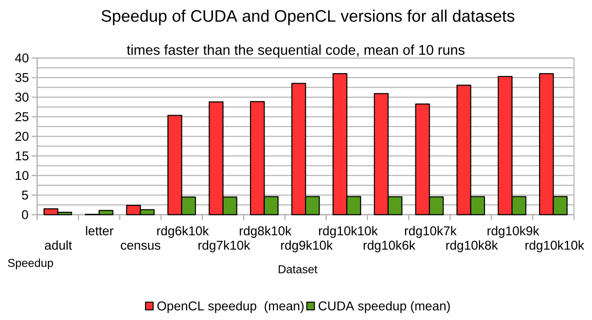

Fig. 8. Speedup of both CUDA and OpenCL versions in relation to the sequential original version.

TABLE II. MEAN EXECUTION TIME μ_e (H:M:S,MS), STANDARD DEVIATION σ (M:S,MS), AND SPEEDUP μ_s OF 10 EXECUTIONS OF OPENCL AND CUDA-BASED APPROACHES IN RELATION TO THE SEQUENTIAL VERSION.

	Dataset	Sequential		OpenCL			CUDA		
		μ_e Sequential	σ Sequential	μ_e OpenCL	σ OpenCL	μ_s OpenCL	μ_e CUDA	σ CUDA	μ_s CUDA
UCI	adult	00:00:37.34	00:00.25	00:00:25.09	00:00.30	**1.49**×	00:01:01.93	00:00.34	0.60×
	letter	00:00:12.49	00:00.06	00:02:17.92	00:00.82	0.09×	00:00:11.58	00:00.01	**1.08**×
	census-income-full	08:37:39.43	05:15.59	03:38:51.90	00:43.65	**2.37**×	06:48:15.29	02:42.85	1.27×
Artificial #obj	rdg6k10k	00:25:25.60	00:01.41	00:01:00.18	00:00.55	**25.35**×	00:05:38.80	00:00.26	4.50×
	rdg7k10k	00:30:33.92	00:00.52	00:01:03.68	00:00.15	**28.80**×	00:06:45.79	00:00.11	4.52×
	rdg8k10k	00:50:54.06	01:04.07	00:01:45.77	00:01.80	**28.87**×	00:11:03.04	00:10.39	4.61×
	rdg9k10k	01:18:11.68	00:01.07	00:02:19.99	00:00.19	**33.52**×	00:16:56.16	00:00.46	4.62×
	rdg10k10k	01:45:24.80	00:05.21	00:02:55.67	00:00.80	**36.00**×	00:22:46.30	00:00.39	4.63×
Artificial #att	rdg10k6k	00:47:04.46	00:02.03	00:01:31.35	00:00.72	**30.92**×	00:10:18.87	00:00.12	4.56×
	rdg10k7k	00:44:22.87	00:01.43	00:01:34.24	00:00.40	**28.26**×	00:09:46.30	00:00.33	4.54×
	rdg10k8k	01:14:51.43	00:03.46	00:02:15.88	00:00.80	**33.06**×	00:16:14.75	00:00.55	4.61×
	rdg10k9k	01:32:42.77	00:03.78	00:02:37.67	00:00.92	**35.28**×	00:20:07.65	00:00.80	4.61×
	rdg10k10k	01:45:24.80	00:05.21	00:02:55.67	00:00.80	**36.00**×	00:22:46.30	00:00.39	4.63×

SVMs algorithm [7] within a GPU, achieving a considerable speedup when compared to its sequential version, and also to a CUDA-based approach [8]. For such, we employed the OpenCL framework [3], which allows the proposed approach to be portable to heterogeneous environments – our approach is capable of running in CPUs, GPUs, APUs, and even in mobile architectures.

The proposed approach is successful in significantly increasing LIBSVM's computational performance up to 36×, while keeping the same predictive performance measured in terms of classification accuracy. The most time-consuming classification problem, which was represented by the artificial dataset rdg10k10k, was reduced from approximately 1 hour and 45 minutes of computation to only 3 minutes. While we are confident that this is quite an advance for solving large problems with SVMs within a wide range of GPUs, we believe there is room for improvement and exciting future work opportunities, such as performing the parallelization of SVMs for other tasks like multi-class and multi-label classification, and also for regression problems.

ACKNOWLEDGMENTS

The authors would like to thank CAPES – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – and CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico – for funding this work.

REFERENCES

[1] V. Vapnik, "An overview of statistical learning theory," *IEEE Transactions on Neural Networks*, vol. 10, pp. 988–999, 5 1999.

[2] NVIDIA, *CUDA Toolkit Documentation*, http://docs.nvidia.com/cuda/, 2014.

[3] Khronos, *The OpenCL 1.2 Specification*, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2012.

[4] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *ArXiv preprint arXiv:1005.2581*, 2010.

[5] Forbes, *Pc gpu market bounces back, with nvidia up and amd down*, http://www.forbes.com/sites/jasonevangelho/2014/02/19/pc-gpu-market-bounces-back-with-nvidia-up-and-amd-down/, 2014.

[6] Valve, *Hardware and software survey*, http://store.steampowered.com/hwsurvey/videocard/, 2014.

[7] C.-C. Chang and C.-J. Lin, "Libsvm: A library for Support Vector Machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, 27:1–27:27, 3 2011.

[8] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "Gpu acceleration for support vector machines," in *Proceedings of 12th International Workshop on Image Analysis for Multimidia Interactive Services*, Delft, Netherlands, 2011.

[9] V. Vapnik and C. Cortes, "Support vector networks," *Machine Learning*, vol. 20, p. 273, 3 1995.

[10] I. Witten, E. Frank, and M. Hall, *Data Mining - Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011, p. 629.

[11] A. C. Lorena, "Investigação de estratégias para a geração de máquinas de vetores de suporte multiclasses," http://www.teses.usp.br/teses/disponiveis/55/55134/tde-26052006-111406/, PhD thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, São Paulo, Brazil, 2006.

[12] B. Catanzaro, N. Sundaram, and K. Keutzer, "Support vector machine training and classification on graphics processors," in *Proceedings of the International Conference on Machine Learning*, Helsink, Finland, 2008, pp. 104–111.

[13] Y. Lu, V. Roychowdhury, and L. Vandenberghe, "Distributed parallel support vector machines in strongly connected networks," *IEEE Transactions on Neural Networks*, pp. 1167–1178, 2008.

[14] T. Joachims, "Making large-scale svm learning practical," in *Advances in Kernel Methods - Support Vector Learning*, B. Schölkopf, C. Burges, and A. Smola, Eds., Cambridge, MA: MIT Press, 1999, ch. 11, pp. 169–184.

[15] Microsoft, *Visual studio documentation*, http://www.visualstudio.com/pt-br/get-started/overview-of-get-started-tasks-vs, 2014.

[16] K. Bache and M. Lichman, *UCI Machine Learning Repository*, http://archive.ics.uci.edu/ml, 2013.