Support Vector Machine Acceleration for Intel Xeon Phi Manycore Processors

Renzo Massobrio[1,2], Sergio Nesmachnow[1], and Bernabé Dorronsoro[2]

[1] Universidad de la República, Uruguay
{renzom,sergion}@fing.edu.uy
[2] Universidad de Cádiz, España
bernabe.dorronsoro@uca.es

Abstract. Support vector machines are widely used for classification and regression tasks. However, sequential implementations for support vector machines are usually unable to deal with the increasing size of current real-world learning problems. In this context, Intel®Xeon Phi™processors allow easily incorporating high performance computing strategies to improve execution times. This article proposes a parallel implementation of the popular LIBSVM library, specially adapted to the Intel®Xeon Phi™architecture. The proposed implementation is evaluated using publicly available datasets corresponding to classification and regression tasks. Results show that the proposed parallel version computes the same results than the original LIBSVM while reducing the time needed for training by up to a factor of 4.81.

1 Introduction

Support vector machines (SVMs) are supervised learning models which are used for classification and regression analysis [2]. SVMs have been widely applied to solve various real world problems, e.g., text categorization, image classification, hand-written text recognition, protein classification.

In their simplest form, SVMs are non-probabilistic binary linear classifiers. SVMs build a model given a set of training samples, each marked as members of one of two possible classes. This model is a representation of the training samples as points in space that aims at separating samples from different classes by the widest possible gap. Fig. 1 shows an example of a linear binary SVM classifier, where H1 does not separate the two classes, H2 separates them, but H3 separates the classes creating the widest gap between both groups. Once the model is built, new, unknown samples can be mapped into that same space in order to predict the class to which they belong.

More generally, a SVM builds a hyperplane (or set of hyperplanes) in a high-dimensional space that has the largest distance to the nearest training-data point of any class (known as the functional margin). In most cases, the points to be classified are not linearly separable in the input dimensional space. Therefore, the input points can be mapped into a higher-dimensional space, to make the separation task easier. For this purpose, several kernel functions have

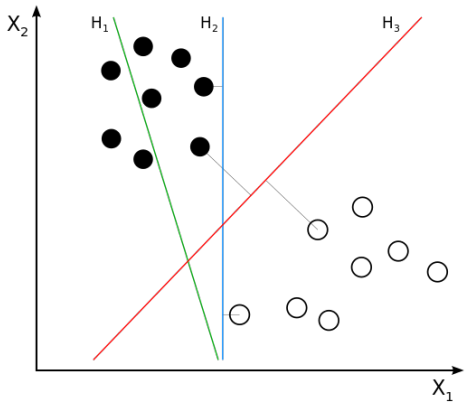


Fig. 1: Example of a linear binary SVM classifier

been proposed, which project the original points into higher dimensional spaces. Then, the SVM finds a linear hyperplane that separates the points with the maximal margin in this higher dimensional space.

Classification and regression tasks are currently demanding increasing computational resources due to the growing size of datasets and input vectors. Therefore, sequential SVM implementations are not able to efficiently cope with large learning tasks. Parallel implementations are necessary to be able to handle large learning problems with SVMs. In this context, manycore processors, such as Intel®Xeon Phi™, are an interesting option to exploit parallelism using a standard CPU form factor, without the need for large and expensive HPC infrastructure.

This article proposes a parallel implementation of the popular LIBSVM library [3], specifically adapted to the latest Intel®Xeon Phi™architecture. The proposed implementation is evaluated using datasets of classification and regression problems. Results show that the proposed implementation is able to significantly reduce training times, which are the most demanding in most learning problems, while computing exactly the same results than the original LIBSVM. For the evaluated datasets, the best results indicate that the proposed implementation accelerates LIBSVM in up to 4.81x.

The remainder of the manuscript is organized as follows. Section 2 introduces manycore processors, the Intel®Xeon Phi™architecture, and the software used in the parallel implementation of LIBSVM. A brief review of related works is presented in Section 3. Then, Section 4 presents in detail the proposed LIBSVM implementation and Section 5 presents the experimental results. Finally, Section 6 presents the conclusions and main lines of future work.

2 Hardware and Software Platform

This section introduces manycore processors and presents the main characteristics of the Intel®Xeon Phi™architecture. Then, a brief introduction to the software used for the parallel implementation is presented.

2.1 Manycore Processors and Intel®Xeon Phi™

Manycore processors are multi-core processors specially designed for a high degree of parallelism, consisting of tens or thousands of simpler independent cores. The use of manycore processors has been increasing in the past years, with extensive applications in embedded systems and high-performance computing platforms. Sunway TaihuLight is the fastest supercomputer as of June 2017 according to the TOP500 ranking [15]. This chinese supercomputer, installed at the National Supercomputing Center in Wuxi, consists of 40,960 manycore processors with 260 cores each, totalling 10,649,600 cores [5].

Xeon Phi™is a brand name given to a series of manycore processors commercialized by Intel®. This family of processors was initially designed as an add-on PCIe card which could be connected to a standard CPU and used for computing intensive tasks. A second generation of Xeon Phi™products, with codename Knights Landing (KNL), was announced on June 2013. The main difference with its prior generation is that KNL are stand-alone processors that can boot an off-the-shelf operating system. Therefore, KNL avoids the bottlenecks in PCIe communications —which are inherent in coprocessors— and provides a powerful HPC platform in a standard CPU form factor. Sodani et al. [14] presents an interesting overview of the KNL architecture. A brief description of the main characteristics of the KNL architecture are described next.

The KNL CPU consists of 38 physical tiles: at most 36 are active and the remaining two are used for recovery purposes. Each tile has two cores, two vector processing units (VPUs) per core, and a shared 1MB L2 cache. The processing cores derive from the Intel®Atom™core microarchitecture, but incorporate several modifications specially design to suit HPC workloads, e.g., support for four threads per core, larger and faster caches, and larger translation look-aside buffers (TLBs). Each KNL core supports up to four hardware contexts or threads by means of hyperthreading techniques. Additionally, KNL incorporates a new instruction set named AVX-512, which supports 512 bit vector instructions and a 2D mesh that interconnects the tiles with other components of the chip such as memory and I/O controllers. KNL introduces an innovative memory architecture comprising two types of memory: Multichannel DRAM (MCDRAM), which provides high bandwidth, and double data rate (DDR) memory for larger capacity. These two types of memory can be used in three different memory modes: i) *cache mode*, where MCDRAM acts as cache for DDR; ii) *flat mode*, where MCDRAM is used as standard memory sharing the same address space as DDR; and *hybrid mode*, where a portion of MCDRAM is in cache mode and the remainder is in flat mode. All these features make the KNL architecture a

good candidate for HPC tasks, without requiring any special way of programming other than the standard CPU programming model, and even having decent support for serial legacy code.

2.2 Intel®C++ Compiler

The Intel®C++ compiler is part of the Intel®Parallel Studio XE suite. The compiler incorporates many optimizations to take advantage of specific processor features, such as the number of available cores and wider vector registers, to speed up computations. Intel®C++ compiler has broad support for current and previous C and C++ standards, including full support for C++11 and C99. Furthermore, it supports integration with OpenMP for parallel implementations.

2.3 Intel®Math Kernel Library

Intel®Math Kernel Library (MKL) supports a series of optimized and threaded math functions to take advantage of the architecture of Intel®processors to solve large computational problems. MKL performs a hardware check on runtime and selects suitable functions to improve execution time by means of instruction-level and register-level SIMD parallelism [16]. Intel®MKL also incorporates threadsafe functions to speedup computations using OpenMP. Although Intel®MKL is optimized for the latest Intel®processors, including processors with multiple cores, it can also be used in non-Intel®CPUs. The library provides Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) routines, fast Fourier transforms, vectorized math functions, random number generation functions, and many other features. In this work, we use specific functions in the BLAS Level 1 suite, which includes routines and functions that perform vector to vector math operations.

3 Related Work

There are multiple libraries and frameworks that implement SVMs in a variety of programming languages. Ivanciuc (2007) provides a thorough list and comparison of different SVM implementations [11].

LIBSVM (Library for Support Vector Machines) was developed by Chang and Lin [3] and supports vector classification, regression, and distribution estimation problems. It was developed with the goal of helping users from other fields to easily use SVM as a research tool. LIBSVM provides a simple interface for users to link with their own programs. The main features of this framework include: several SVM formulations, efficient multi-class classification, cross validation, probability estimates, various kernels (e.g., linear, polynomial, radial basis function (RBF), and sigmoid), and weighted SVM for unbalanced data. It is implemented in both C++ and Java, and also has interfaces in the following languages: Python, R, MATLAB, Perl, Ruby, Weka, Common LISP, CLISP, Haskell, OCaml, LabVIEW, and PHP.

GPU-accelerated LIBSVM is a modified version of the original LIBSVM code that takes advantage of the CUDA framework to significantly reduce processing time while producing identical results to the original framework [1]. This GPU version does not currently supports all the features of the original LIBSVM and is only compatible with C-SVC classification mode using the RBF kernel. From the users perspective, the functionality and interface of LIBSVM remains the same. However, the kernel computation is performed using a GPU. The authors propose a combined CPU+GPU where the computation of the kernel matrix elements is offloaded to the GPU, to decrease the processing time for the training phase.The experimental evaluation was performed using a training dataset consisting of high level features in video shots. Unfortunately, the datasets used are not publicly available for comparison. In small datasets, the difference between using only CPU and the combined CPU+GPU approach is hardly noticeable. However, for larger instances, the execution time when exploiting the GPU is up to one order of magnitude lower in comparison to using the CPU alone. The authors mention that it is important to notice the memory issues that can arise in very large instances due to the limited memory resources in GPU cards.

You et al. (2014) presented a custom implementation of binary classification SVM for multi-core and many-core architectures [17]. The proposed framework is evaluated using Intel®Ivy Bridge CPUs and Intel®Xeon Phi™co-processor (MIC) from the previous generation to the one used in our proposal (KNL). Several strategies are presented to improve the parallelism and optimize the implementation to the underlying architectures, including: parallel kernel evaluation, affinity models for thread-to-core assignment, vectorization, and memory alignment. The authors suggest using both "sparse" and "dense" formats to represent training vectors, depending on the characteristics of the training dataset, as it will be discussed in Section 4.3. An interesting approach is suggested to choose the format automatically. The authors propose training different datasets using both "sparse" and "dense" formats and measuring execution times. Afterwards, they propose building a classifier with features of each trained dataset (e.g., number of vectors, size of vectors, density) and labels indicating whether it was faster to use the "sparse" or the "dense" format. With this trained classifier, a user could predict whether their dataset would benefit from using a specific representation format. This is an interesting approach that could also be applied to our proposal. However, the authors do not provide specific details about the accuracy of the implemented format predictor. The proposed SVM implementation is able to improve LIBSVM by 4.4x–84x on MIC and 18x–47x on Ivy Bridge CPU. The training results are not exactly the same as those computed by LIBSVM due to implementations nuances. However, the authors show that the differences in accuracy for the studied datasets are minimal. Unfortunately, the proposed library does not have an official release and the only code available online has not been updated for two years as of 2017.

The analysis of related works shows that there is an interest in the research community for optimizing SVMs due to the growing size of learning problems. However, the most popular SVM libraries are serial and do not exploit the par-

allelism offered by modern manycore and multi-core architectures. Some efforts have been made either to adapt existing libraries or to implement new ones, that take advantage of highly parallel computing architectures. There is still room to contribute in adapting popular SVM implementations, such as LIBSVM, to modern manycore architectures like Intel®Xeon Phi™.

4 LIBSVM Implementation for Intel®Xeon Phi™

This section presents the specific modifications performed to the LIBSVM code to adapt it to the Intel®Xeon Phi™architecture. The proposed changes were implemented over version 3.22 of the LIBSVM code.

4.1 Coarse-grain Parallelism Using OpenMP

The original LIBSVM code is sequential. Therefore, it does not take advantage of the multiple cores present in Intel®Xeon Phi™machines. A simple way to exploit the availability of multiple cores is by using OpenMP to parallelize the loop that processes each training vector, as suggested in the LIBSVM FAQ [4]. This approach involves two simple modifications to the original LIBSVM code [3]: i) adding two OpenMP pragmas: one before the for loop in function `get_Q` of class `SVC_Q` and one before the for loop in function `svm_predict_values`; ii) adding the `-fopenmp` flag to the LIBSVM Makefile. The pragma line added to the LIBSVM code is the following, where `j` is the loop variable:

```
#pragma omp parallel for private(j) schedule(guided).
```

The `guided` option uses a work queue to give each thread a chunk-sized block of loop iterations. When a thread finishes processing the assigned block, it retrieves the next block of loop iterations from the top of the work queue. The size of the chunk is initially large and decreases over time to better handle load imbalance between iterations. By default, the starting chunk size is approximately equal to the ratio between the number of iterations of the loop and the number of threads.

The number of threads to be used can be set on runtime by setting the appropriate value in the environment variable `OMP_NUM_THREADS`.

4.2 Compiling with Intel®C++ compiler

The previous modification works with any C++ compiler that supports OpenMP, without the need of compiling using the Intel®C++ compiler. However, in order to exploit the specific characteristics of the Intel®Xeon Phi™architecture, and to be able to use the Intel®MKL, we propose changing the compiler used in the original LIBSVM code (g++) for the Intel®C++ compiler.

[3] This modification applies to classification tasks, the approach is similar for regression tasks

The compiler options were set according to the recommendations of the Intel®Math Kernel Library Link Line Advisor [10]. This allows to optimize the compiled code to the specific hardware architecture where the program will execute. In addition to the recommended optimization flags, it was necessary to include the `-fp-model precise` option. This flag instructs the compiler to avoid performing several optimizations targeted to floating point arithmetic. While this prevents from further optimization, it guarantees returning the same results as the original LIBSVM library. If the user allows some float precision differences with the original LIBSVM, this flag can be removed and achieve an extra improvement in performance.

4.3 Integration with Intel®MKL

The previous modifications apply to all learning tasks, independently of the chosen kernel function. However, in the rest of the work we focus on accelerating the training time for classification and regression tasks using the RBF kernel function. The rationale behind this decision is that the RBF kernel is, in general, a reasonable choice for many learning tasks, as recommended by the LIBSVM Practical Guide [8]. The RBF kernel function maps training samples into a higher dimensional space, so it can handle non-linear relations between attributes and class labels. Additionally, it has fewer hyperparameters than other kernels (e.g., polynomial kernel) and presents fewer numerical difficulties than other kernels. The RBF kernel on two samples x and x', represented as feature vectors in some input space, is defined by Equation 1, where $\gamma > 0$ is a free parameter.

$$K(x, x') = e^{-\gamma||x-x'||^2} \tag{1}$$

An initial profiling experiment of the `svm-train` program was performed to identify bottleneck functions. The profiling was performed using *gprof* [6] and training on the *connect-4* dataset [13] available at the LIBSVM dataset repository [3]. This dataset is not used in the experimental evaluation of the proposed solution in order to avoid bias and to demonstrate that the implementation scales well on different datasets.

The profiling of the `svm-train` program revealed that 87.2% of the total execution time was spent in the `dot` function of class `Kernel`. This function performs the dot product of two vectors. The dot product is used when computing the RBF kernel, since the LIBSVM implementation expands Equation 1 into the form presented in Equation 2.

$$K(x, x') = e^{-\gamma\left((x \cdot x)+\left(x' \cdot x'\right)-2\left(x \cdot x'\right)\right)} \tag{2}$$

These results suggest that an improvement in the dot product calculation would greatly impact the overall training time. To improve the `dot` function we propose using the `cblas_ddot` routine available in the BLAS Level 1 group of functions and routines of Intel®MKL, which has the following header:

```
double cblas_ddot (const MKL_INT n, const double *x, const MKL_
INT incx, const double *y, const MKL_INT incy);
```

where the parameters are:

- `n`, the number of elements in vectors `x` and `y`.
- `x`, an array with size at least $(1 + (n - 1) \times abs(\texttt{incx}))$.
- `incx`, the increment for the elements of x.
- `y`, an array with size at least $(1 + (n - 1) \times abs(\texttt{incy}))$.
- `incy`, the increment for the elements of `y`.

Using the `cblas_ddot` function instead of the original code in LIBSVM is not as straightforward as interchanging only that portion of the code, due to the format in which LIBSVM stores the training vectors. LIBSVM uses a "sparse" format, in which zero values are not stored. Instead, training vectors are stored as `<index:value>` pairs. For instance, the training vector `<0,1,0,3>` is internally represented as `(2:1 4:3)`. This format does not allow using the `cblas_ddot` function directly. Therefore, it was necessary to modify other sections of the LIBSVM code to implement a "dense" format, in which vectors are stored directly as arrays, including zero values. As it will be presented in the experimental analysis discussion in Section 5, this design decision may achieve better or worse performance depending on the specific characteristics of the training set used. Therefore, we implemented this modification in a way that the user can decide at compiling time whether to use the original "sparse" format or the proposed "dense" format, depending on the specific characteristics of the training dataset.

The `cblas_ddot` implementation is threaded. The number of threads to be used can be set on runtime by setting the environment variable `MKL_NUM_THREADS`.

5 Experimental Analysis

This section presents and discusses the experimental results of the proposed parallel implementation of LIBSVM for Intel®Xeon Phi™KNL systems.

5.1 Execution Platform

The experimental evaluation was performed on an Intel®Xeon Phi™7250 processor, with 68 cores, and 64GB of RAM. The server ran Linux Ubuntu 16.04 and had version 17.0.1 of the Intel®C++ compiler. The server was not shared with other users or performed any other intensive tasks during the experiments, in order to accurately measure the execution times.

5.2 Problem Instances

Three learning datasets were used for the experimental evaluation of the proposed implementation. These datasets were obtained from the LIBSVM dataset repository [3]. The datasets, which correspond to classification and regression problems, are:

– *gisette*, a dataset corresponding to a handwritten digit recognition problem with the goal of separating the digits '4' and '9'. This dataset was one of five datasets of the NIPS 2003 feature selection challenge [7].
– *E2006*, a dataset with reports from thousands of publicly traded U.S. companies, published in 1996–2006, and stock return volatility measurements in the twelve-month period before and the twelve-month period after each report [12].
– *usps*, a database for handwritten text recognition research, consisting of digitalized images at 300 pixels/in in 8-bit gray scale, corresponding to U.S. post codes scanned from real mail [9].

The datasets *gisette* and *E2006* were sub-sampled using the `subset.py` script included in LIBSVM, which allows making a stratified selection of training samples, keeping the rate of appearance of each label in the dataset. We used a subset of 1000 samples of each dataset. Therefore, we will refer to them as *gisette_1000* and *E2006_1000* for the remainder of the manuscript. Table 1 shows the main characteristics of each of the datasets used in the experimental evaluation, including the type of problem (classification or regression), the number of training vectors (# samples), the length of each training vector (# features) and, for classification problems, the number of classes (# labels).

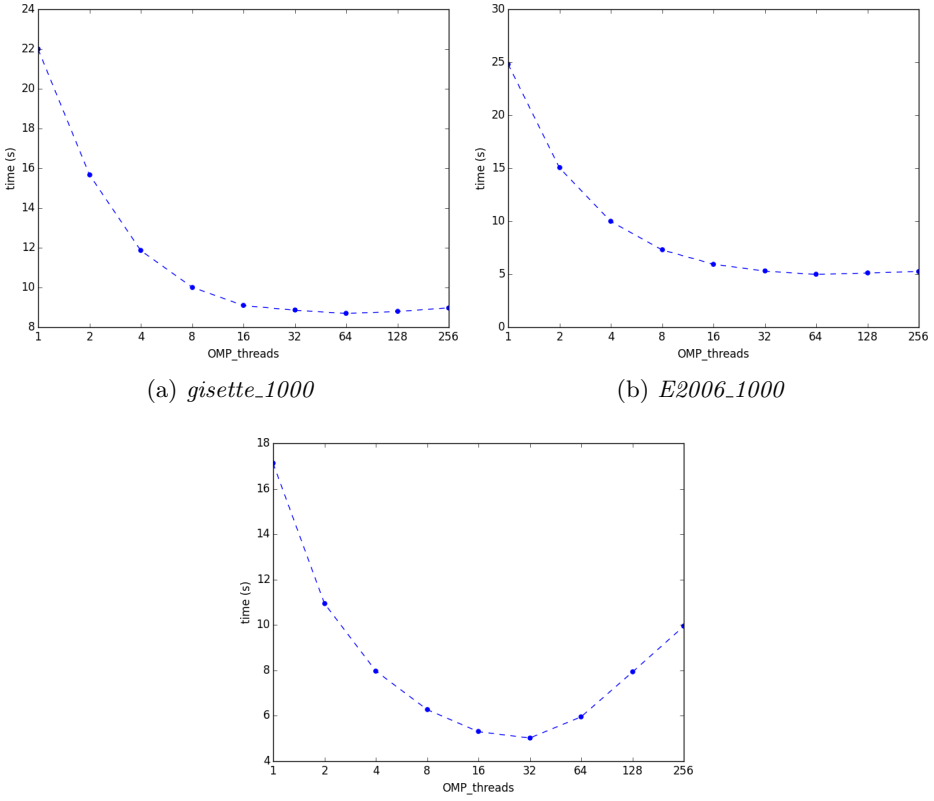
	problem	# *samples*	# *features*	# *labels*
gisette_1000	classification	1000	5000	2
E2006_1000	regression	1000	150360	-
usps	classification	7291	256	10

Table 1: Datasets used for the experimental evaluation

5.3 Coarse-grain Parallelization

Initially, we present the results achieved when using the Intel®C++ Compiler and OpenMP for coarse-grain parallelism of the outer loop that performs the kernel evaluations. These results correspond to the modifications described in Sections 4.1 and 4.2, i.e., without changing the vector representation format and without using Intel®MKL. Fig. 2 shows the average execution time in seconds for the three studied instances, when varying the number of threads (`OMP_NUM_THREADS` environment variable) assigned to the loop. The results correspond to 30 independent executions of each instance with each studied number of threads.

Results show that good execution time improvements are achieved when using more than one core on all studied instances. However, execution times stop improving when using more than 64 cores for both *gisette_1000* and *E2006_1000* datasets, and there is even a significant negative impact when using large number of cores with the *usps* dataset. This could be explained due to the fact



(a) *gisette_1000* (b) *E2006_1000*

(c) *usps*

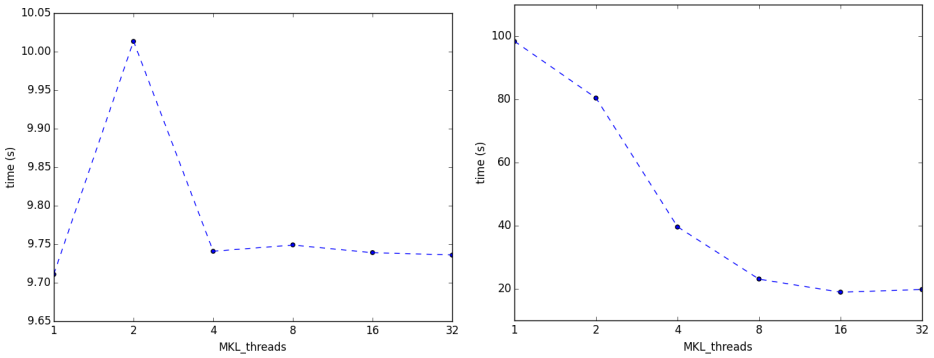
Fig. 2: Mean execution time with different number of OMP threads

that the Intel®Xeon Phi™processor used has 68 physical cores. Therefore, when using more threads, some of the CPU resources are shared among the threads, incurring in a noticeable overhead.

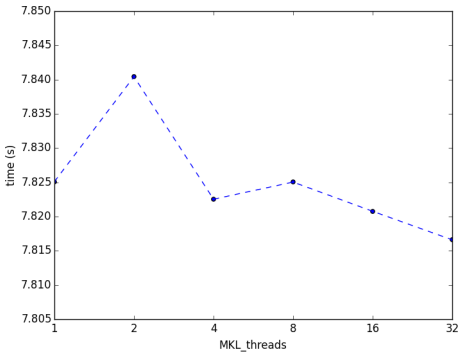
5.4 Vectorized Dot Product Computation

Later, we evaluated the results of implementing the modifications described in Section 4.3, i.e., changing from a "sparse" to a "dense" vector representation and including Intel®MKL for the dot product calculation. Fig. 3 shows the average execution time in seconds for the three studied instances, when varying the number of threads (`MKL_NUM_THREADS` environment variable) assigned to the dot product calculation. There is no coarse-grain parallelization in these executions (i.e., `OMP_NUM_THREADS` = 1). The results correspond to 30 independent executions of each instance with each studied number of threads.

These results give information on two aspects. Firstly, on the convenience (or not) of using the "dense" format and including Intel®MKL for the dot product



(a) *gisette_1000* (b) *E2006_1000*



(c) *usps*

Fig. 3: Mean execution time with different number of MKL threads

calculation. Secondly, to discuss the usefulness of adding parallelism at the vector level when computing the dot product.

To discuss the first aspect, we should compare the execution times when using only one OMP thread in Fig. 2 vs using one MKL thread in Fig. 3. It can be seen that execution times significantly improve when running a sequential version with the "dense" representation format and the MKL dot product calculation for both *gisette_1000* and *usps* instances. However, for *E2006_1000* instance, since the training vectors are much larger, using the "dense" format and only one MKL thread negatively impacts the execution time. In this case, the effects of the "dense" representation are only mitigated when adding more MKL threads to reduce the execution times.

The results show that when using the "dense" format, the improvements achieved by using a larger number of threads for the dot product computation are only noticeable for very large vectors. For *usps* instance, with vectors of size 256, the improvements when using more than one thread are marginal. For *gisette_1000*, with vectors of size 5000, there is even some minor performance decline

when using more than one thread. Additionally, there is a strange behaviour when using exactly two threads, possibly due to the overhead of creating the pool of threads. However, for instance *E2006_1000*, with training vectors of size 150360, there is a noticeable improvement when using more threads for the dot product calculation.

In conclusion, dense vectors benefit from changing the original LIBSVM representation and using Intel®MKL for the dot product calculation, but only dense and large vectors benefit from using multiple threads when computing each dot product. The proposed implementation allows the user to control both the vector representation and the number of outer (OMP) and inner (MKL) threads, thus, enabling the user to tune the library to the specific needs or their learning task. A tool like the one suggested by You et al. (2014) [17] would be interesting to develop, in order to suggest users the values for these parameters that best fit their dataset.

5.5 Two-level Parallelization Approach

Taking into account the results discussed in the previous sections, we performed 30 independent executions of each training dataset, using the configuration of threads that achieved best results. The selected configurations are reported in Table 2, where `OMP_NUM_THREADS` indicates the number of threads used for the outer loop of kernel evaluations (i.e., coarse-grain parallelism) and `MKL_NUM_THREADS` indicates the number of threads assigned to compute each vector dot product, when the "dense" format is used. Additionally, 30 independent executions of the original LIBSVM library were performed over each dataset.

	format	`OMP_NUM_THREADS`	`MKL_NUM_THREADS`
gisette_1000	dense	64	1
E2006_1000	sparse	64	-
usps	dense	32	32

Table 2: Thread configuration used for each problem instance

Table 3 presents the execution times achieved by the original LIBSVM code and the proposed implementation using the best configuration for each problem instance. For each instance the minimum (best), average, and standard deviation are presented with the following format: mean±std (min). All times are expressed in seconds. Additionally, the average acceleration achieved is presented for each instance. The average acceleration is computed as the ratio between the average execution time of LIBSVM and the average execution time of the proposed implementation.

Results show that the proposed implementation is able to efficiently improve the training time while computing exactly the same results than the original

	LIBSVM	best configuration	$\overline{\text{acc}}$
gisette_1000	22.66 ± 0.06 (22.59)	7.07 ± 0.02 (7.03)	3.21x
E2006_1000	20.59 ± 0.03 (20.56)	4.98 ± 0.02 (4.96)	4.13x
usps	18.46 ± 0.06 (18.24)	3.84 ± 0.01 (3.81)	4.81x

Table 3: Execution time in seconds (mean ± std (min)) and average acceleration of the proposed approach against the original LIBSVM

LIBSVM. The proposed implementation achieves an acceleration of up to 4.81x on average on the *usps* instance. These training time improvements are significant, specially when considering larger training datasets that would otherwise be intractable for sequential SVM implementations.

6 Conclusions and Future Work

This article presented a parallel implementation of the popular LIBSVM software, specifically adapted to the Intel®Xeon Phi™architecture. The proposed implementation allow reducing the training time while computing the exact same results than the original LIBSVM. The modifications proposed include: coarse-grain parallelization of kernel evaluations, a new format for representing training vectors, the integration with Intel®MKL for kernel computation, and optimizations at compiling time to exploit the underlying architecture. The experimental evaluation was performed using three publicly available datasets, corresponding to different classification and regression problems, and with different characteristics in terms of number, size, and density of the training vectors.

The main lines of future work include exploring more techniques to further improve the optimization of LIBSVM code. Some of the possible techniques to explore include: aligning vectors in memory to make a better use of caches in Intel®Xeon Phi™cores, improve affinity in the thread-to-core assignment, and optimize other parts of the LIBSVM code that could benefit from the manycore architecture. Additionally, it will be interesting to generalize the proposed approach to other kernels and evaluate the results over larger learning problems, with bigger and denser training vectors. Finally, a tool to help users decide the best vector representation format and number of threads to assign to each level of parallelism should be developed.

Acknowledgement

The work of R. Massobrio and S. Nesmachnow was partly supported by PEDECIBA and ANII, Uruguay. R. Massobrio thanks the support of ANII, Uruguay and Fundación Carolina, Spain. B. Dorronsoro thanks MINECO (TIN2014-60844-R and RYC-2013-13355).

References

1. Athanasopoulos, A., Dimou, A., Mezaris, V., Kompatsiaris, I.: GPU acceleration for support vector machines. In: Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS) (2011)
2. Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. In: Proc. of the Fifth Annual Workshop on Computational Learning Theory. pp. 144–152. COLT '92, ACM, New York, NY, USA (1992)
3. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Trans. on Intelligent Systems and Technology 2, 27:1–27:27 (2011), software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
4. Chang, C.C., Lin, C.J.: LIBSVM FAQ. `http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html#f432` (2015), Accessed: 2017-07-14
5. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The Sunway TaihuLight supercomputer: system and applications. Science China Information Sciences 59(7), 072001 (2016)
6. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. 17(6), 120–126 (1982)
7. Guyon, I., Gunn, S., Hur, A.B., Dror, G.: Result analysis of the nips 2003 feature selection challenge. In: Proc. of the 17th Int. Conf. on Neural Information Processing Systems. pp. 545–552. NIPS'04, MIT Press, Cambridge, MA, USA (2004)
8. Hsu, C.W., Chang, C.C., Lin, C.J.: A Practical Guide to Support Vector Classification. `https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf` (2003), Accessed: 2017-07-14
9. Hull, J.J.: A database for handwritten text recognition research. IEEE Trans. on Pattern Analysis and Machine Intelligence 16(5), 550–554 (1994)
10. Intel®Software: Intel®Math Kernel Library Link Line Advisor. `https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor` (2017), Accessed: 2017-07-14
11. Ivanciuc, O.: Applications of Support Vector Machines in Chemistry, pp. 291–400. John Wiley & Sons, Inc. (2007)
12. Kogan, S., Levin, D., Routledge, B.R., Sagi, J.S., Smith, N.A.: Predicting risk from financial reports with regression. In: Proc. of Human Language Technologies: The 2009 Annual Conf. of the North American Chapter of the Association for Computational Linguistics. pp. 272–280. NAACL '09, Association for Computational Linguistics, Stroudsburg, PA, USA (2009)
13. Lichman, M.: UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml` (2013), Accessed: 2017-07-14
14. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights Landing: Second-Generation Intel Xeon Phi Product. IEEE Micro 36(2), 34–46 (2016)
15. TOP500.org: Top500 List - June 2017. `https://www.top500.org/list/2017/06/` (2017), Accessed: 2017-07-14
16. Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel Math Kernel Library, pp. 167–188. Springer International Publishing, Cham (2014)
17. You, Y., Song, S.L., Fu, H., Marquez, A., Dehnavi, M.M., Barker, K., Cameron, K.W., Randles, A.P., Yang, G.: MIC-SVM: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures. In: 2014 IEEE 28th Int. Parallel and Distributed Processing Symp. pp. 809–818 (2014)