Fast Training of Support Vector Machines on the Cell Processor

Moreno Marzolla*

Dipartimento di Scienze dell'Informazione, Università di Bologna (Italy)

Abstract

Support Vector Machines (SVMs) are a widely used technique for classification, clustering and data analysis. While efficient algorithms for training SVM are available, dealing with large datasets makes training and classification a computationally challenging problem. In this paper we exploit modern processor architectures to improve the training speed of LIBSVM, a well known implementation of the Sequential Minimal Optimization algorithm. We describe LIBSVM$_{CBE}$, an optimized version of LIBSVM which takes advantage of the peculiar architecture of the Cell Broadband Engine. We assess the performance of LIBSVM$_{CBE}$ on real-world training problems, and we show how this optimization is particularly effective on large, dense datasets.

Keywords: Support Vector Machines, Cell Broadband Engine, Parallel Computing

1. Introduction

SVMs are widely used supervised learning methods which can be employed in many classification tasks (see, e.g., [1] and references therein). In this paper we consider the problem of *binary* classification, where N data points (*training set*) must be classified in two classes.

Formally, let us consider N vectors in m-dimensional space: $\boldsymbol{x}_i \in \mathbf{R}^m, i = 1, \ldots, N$. Vector $\boldsymbol{x}_i$ is associated with label $y_i \in \{-1, 1\}$. The set $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, N\}$ is called the *training set*. The classification problem is to separate the two classes with a m-dimensional surface that maximizes the margin between them. The separating surface is obtained by computing the solution $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_N]^{\mathrm{T}}$ of a Quadratic Programming (QP) problem of the form [2]:

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^{\mathrm{T}} Q \boldsymbol{\alpha} - \sum_{i=1}^{N} \alpha_i \\ \text{subject to} \quad & \sum_{i=1}^{N} y_i \alpha_i = 0 \\ & 0 \le \alpha_j \le C, \quad j = 1, \ldots, N \end{aligned} \tag{1}$$

where the entries Q_{ij} of the symmetric positive semidefinite matrix Q are defined as

$$Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j), \quad i, j = 1, \ldots, N \tag{2}$$

$K : \mathbf{R}^m \times \mathbf{R}^m \to \mathbf{R}$ is a kernel function which depends on the type of the separating surface. Examples are the polynomial kernel:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j; a, r, d) = \left(a \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j + r\right)^d, \quad a, r \in \mathbf{R}, d \in \mathbf{N} \tag{3}$$

and the Radial Basis Function (RBF) kernel:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j; \gamma) = \exp\left(-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2\right), \quad \gamma \in \mathbf{R}^+ \tag{4}$$

A SVM is trained by solving the QP problem (1) using vectors $\boldsymbol{x}_i$ and the corresponding labels y_i. The solution $\boldsymbol{\alpha}$ can then be used to classify any new point $z \in \mathbf{R}^m$ by computing its class $f(z)$ as:

$$f(z) = \operatorname{sgn}\left(b + \sum_{i=1}^{N} y_i \alpha_i K(\boldsymbol{x}_i, z)\right) \tag{5}$$

where the offset b is computed during the training step as well.

The size of real-world datasets makes the solution of (1) using general purpose QP solvers impractical. For this reason, efficient ad-hoc algorithms that take advantage of the special structure of (1) have been developed. The Sequential Minimal Optimization (SMO) algorithm, originally proposed by Platt [3], decomposes the original QP problem into two-dimensional subproblems which can be solved analytically. The idea of SMO is to compute a solution iteratively, by optimizing two coefficients α_i, α_j at each iteration. SMO is efficient because it does not use a costly numerical QP solver in its inner loop.

Unfortunately, training times are still significant for many real-world datasets. The reason is that matrix Q can be very large and can not be kept entirely in memory. Thus, the SMO algorithm (as well as most of the existing SVM training algorithms) needs to recompute the values Q_{ij} many times, which in turn require many evaluations of the kernel function.

In this paper we describe LIBSVM$_{CBE}$, an efficient implementation of the SMO algorithm for asymmetric multi-core architectures. Specifically, LIBSVM$_{CBE}$ employs the peculiar

*Corresponding Author. Address: Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura A. Zamboni 7, I-40127 Bologna, Italy. Phone +39 051 20 94847, Fax +39 051 20 94510

Email address: marzolla@cs.unibo.it (Moreno Marzolla)

features of the Cell Broadband Engine (CBE), an asymmetric multi-core processor architecture originally developed for the consumer market (it is used inside Sony's PlayStation®3 (PS3) gaming console), but which is also used on high-end servers targeted at scientific computations. The Cell processor includes a conventional core based on the PowerPC architecture, together with specialized vector co-processors called Synergistic Processor Elements (SPEs). LIBSVM_{CBE} splits the evaluation of the elements of matrix Q across the SPEs; an optimized, Single-Instruction Multiple-Data (SIMD) algorithm for computing the dot product of two vectors is used inside each SPE, so that the evaluation of kernel functions is very fast.

LIBSVM_{CBE} is based on LIBSVM [4], an efficient and widely used implementing the SMO algorithm. LIBSVM employs several heuristics to reduce the training time, such as the *shrinking heuristic* which dynamically reduces the set of coefficients to be optimized, and a caching strategy to avoid recomputations of recently used entries of matrix Q; still, evaluation of elements Q_{ij} is the bottleneck of LIBSVM. LIBSVM_{CBE} improves that bottleneck by offloading the computation of Q to the vector coprocessors. We test LIBSVM_{CBE} on several real-world datasets and show how this optimization yields significant speedups over the sequential algorithm. Our optimization is very general, because it can be applied to any SVM training and classification package which relies on multiple evaluations of the kernel function.

Organization of this paper. This paper is organized as follows. In Section 2 we revise some of the existing parallelization strategies for training SVMs. In Section 3 we describe the SMO algorithm. In Section 4 we give a brief overview of the architecture of the Cell processor, and then present LIBSVM_{CBE}, a Cell-optimized version of the LIBSVM software package. In Section 5 we evaluate LIBSVM_{CBE} on some training datasets. Finally, conclusions and future works are illustrated in Section 6. We include some implementation details in Appendix A.

2. Related Work

There have been several attempts to optimize the training and classification times of SVMs, by considering parallel approaches to the solution of the QP (1).

In [5] the authors describe an optimized version of SMO which makes use of Graphics Processing Units (GPUs). Modern GPUs can be considered as specialized highly parallel processors, containing a large number (hundreds, or even thousands) of relatively simple processing cores connected to a high bandwidth memory subsystem. This kind of architecture is quite different from the CBE, as the latter includes a limited number of more powerful processing elements, each one having access to a small (but very fast) local store.

A version of SMO for parallel machines using the Message Passing Interface (MPI) library is described in [6]. In [7, 8] the authors propose a parallel training and classification algorithm for large quadratic programs which is based on the Parallel Gradient Projection-based Decomposition Technique (PGPDT). Instead of optimizing two variables at each iteration, PGPDT decomposes the original QP problem in larger subproblems which are be solved on a cluster of workstations using MPI. In [9] the author shows how the CBE can be used to speed up the PGPDT solution technique.

An hybrid algorithm using both MPI and OpenMP compiler extensions is used in [10] to train linear SVMs. The authors use an interior point method for solving the optimization problem; training is performed on a cluster of multiprocessor machines. MPI is used to distribute data amongst the processors, while efficient OpenMP BLAS implementations are used within each node to speed up local computations.

In [11] the authors propose Parallel SVM (PSVM), a parallel approximation technique for SVM training and classification. PSVM is based on an incomplete Cholesky factorization, which greatly reduces both the memory requirement and the computation time on each node. It must be observed that PSVM is based on an approximation technique, so it is slightly less accurate than the other parallel SVM implementations described above. Another recent parallel approximate training and classification algorithm is P-packSVM [12]. P-packSVM uses a stochastic gradient descent method, and employs a packing strategy to reduce multiple iterations on a single one in order to reduce the communication costs.

We observe that the CBE architecture is remarkably different from both conventional clusters of workstations, and from GPUs. A cluster of workstations is made of a large number of nodes, where each node is equipped with a powerful CPU and a large amount of RAM; however, inter-node communications are orders of magnitude slower than computations. A GPU, on the other hand, contains a large number of simple cores connected with a very aggressive memory subsystem. The CBE contains a single general-purpose CPU core connected with a limited number of independent vector coprocessors called SPEs. Therefore, for each of these architectures it is necessary to use ad-hoc algorithmic strategies to achieve good performance and scalability. To the best of our knowledge, no previous attempt has been made to optimize the SMO algorithm for the CBE architecture. The SMO algorithm is attractive because it is widely used in practice and many good open source implementations exist.

3. The SMO Algorithm

Sequential Minimal Optimization is sketched in Algorithm 1 (see [13] for a more detailed description). Algorithm 1 includes the main loop which is used to optimize two coefficients α_i, α_j at each iteration. The selection of the index i, j is a crucial task, as it influences the convergence speed. LIBSVM uses the *second order heuristic* proposed in [13], shown in Algorithm 2.

We observe that Algorithm 1 requires multiple evaluations of rows of matrix Q (we underlined the pseudocode statements where Q is used). For realistic training sets, Q is too large to be stored in memory, so it is necessary to evaluate its elements as they are needed. One optimization is to keep a cache of (partially filled) rows of Q which have been recently computed, so that unnecessary evaluations can be avoided. Despite this

Algorithm 1 Sequential Minimal Optimization

$\tau := 1e-12$ {*small positive constant*}
for all $i := 1, \ldots, N$ **do**
  $G_i := 0$
  $\alpha_i := -1$
loop
  Select i, j using Algorithm 2
  if $i = -1$ **then**
    Stop
  $a := \max\{Q_{ii} + Q_{jj} - 2y_i y_j Q_{ij}, \tau\}$
  $b := y_i G_i + y_j G_j$
  $\alpha_i^{\text{old}} := \alpha_i$
  $\alpha_j^{\text{old}} := \alpha_j$
  $\alpha_i := \alpha_i + y_i b/a$
  $\alpha_j := \alpha_j - y_j b/a$
  $S := y_i \alpha_i^{\text{old}} + y_j \alpha_j^{\text{old}}$
  if $\alpha_i < 0$ **then**
    $\alpha_i := 0$
  else if $\alpha_i > C$ **then**
    $\alpha_i := C$
  $\alpha_j := y_j(S - y_i \alpha_i)$
  if $\alpha_j < 0$ **then**
    $\alpha_j := 0$
  else if $\alpha_j > C$ **then**
    $\alpha_j := C$
  $\alpha_i := y_i(S - y_j \alpha_j)$
  for all $t := 1, \ldots, N$ **do**
    $G_t := Q_{ti}(\alpha_i - \alpha_i^{\text{old}}) + Q_{tj}(\alpha_j - \alpha_j^{\text{old}})$

Algorithm 2 Selection of i, j

$\tau := 1e-12$ {*small positive constant*}
$I_{\text{high}} := \{i : y_i = 1 \wedge \alpha_i < C\} \cup \{i : y_i = -1 \wedge \alpha_i > 0\}$
$i := \arg\max\{-y_i G_i : i \in I_{\text{high}}\}$
$G^+ := \max\{-y_i G_i : i \in I_{\text{high}}\}$
$j := -1$
$I_{\text{low}} := \{i : y_i = 1 \wedge \alpha_i > 0\} \cup \{i : y_i = -1 \wedge \alpha_i < C\}$
$G^- := \min\{-y_i G_i : i \in I_{\text{high}}\}$
$O^- := \infty$
for all $t \in I_{\text{low}}$ **do**
  $b := G^+ + y_t G_t$
  if $b > 0$ **then**
    $a := \max\{Q_{ii} + Q_{tt} - 2y_i y_t Q_{it}, \tau\}$
    if $(-b^2/a) \leq O^-$ **then**
      $j := t$
      $O^- := -b^2/a$
if $(G^+ - G^-) < \epsilon$ **then**
  Return $(-1, -1)$
else
  Return (i, j)

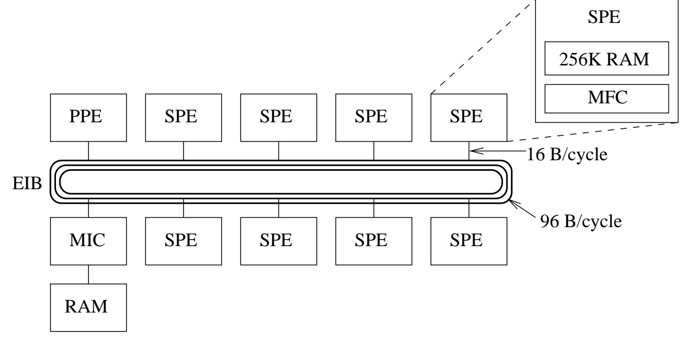


Figure 1: Architecture of the CBE

optimization, code profiling reveal that for some datasets, the evaluation of Q takes up to 90% of the total training time.

4. Fast kernel evaluation on the Cell Processor

The CBE is a heterogeneous multi-core processor, whose internal architecture is shown in Fig. 1. The CBE contains nine processors on a single chip, connected with a high bandwidth circular bus [14].

The Power Processor Element (PPE) is the main processor, and is based on a 64 bit PowerPC architecture with vector and SIMD multimedia extensions. The PPE is responsible for executing the Operating System, allocating resources and distributing the workload to the other computing cores. The PPE has direct access to the main system memory, and includes 32 KB of L1 instruction and data caches, and 512 KB of L2 cache.

The eight SPEs are SIMD processors optimized for data-intensive computations. A SPE contains 128 registers that are 128 bits wide. A single SIMD instruction can operate on sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers or four single-precision floating point numbers, in a single clock cycle. Each SPE has 256 KB of private RAM, called Local Store (LS), which holds data and instructions. A SPE can access the system memory through asynchronous Direct Memory Access (DMA) operations, handled by a dedicate component called Memory Flow Controller (MFC).

The Element Interconnect Bus (EIB) is a 4-ring bus for data, and a tree structure for commands. The EIB internal bandwidth is 96 bytes per cycle, and supports about 100 outstanding DMA transfers between main memory and the SPEs. The Memory Interface Controller (MIC) provides an interface between the EIB and the main storage.

The PowerXCell 8i has an aggregate peak double-precision floating-point performance of 102.4 GFLOPS [14]. However, achieving such a high performance on a given computational problem is challenging, First, the problem must be decomposed so that it can be solved in parallel on the SPEs. Then, optimized SIMD algorithms must be executed on each SPE. Finally, it is necessary to use an appropriate memory layout for the program data in order to allow efficient DMA transfers, which need to be carefully overlapped with computations in order to hide the memory latency.

val	0 12 0 0	3 9 0 0	0 0 0 7	X X X X
index	0	4	12	-1

Figure 2: Encoding of a sparse vector whose nonzero values are $12, 3, 9, 7$ at positions $1, 4, 5, 15$ respectively.

LIBSVM$_{CBE}$ is based on LIBSVM version 2.89 [4]. By profiling LIBSVM, it turns out that the most time-consuming operation (responsible for up to 90% of the training time) is the kernel evaluation, which is done inside the `Qfloat* kernel::get_Q(int i, int l)` method (`Qfloat` being defined as an alias of the C language `float` datatype). This function returns a pointer to an array containing the values $Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for $j = 1, \ldots, l$. The first m elements of row i might have been cached; if $m < l$, then the missing values need to be computed.

LIBSVM can be easily parallelized on symmetric multi-core processors using OpenMP (see the FAQs from LIBSVM web page [4]). This optimization results in the parallel evaluation of the elements Q_{ij} over the CPU cores. In the case of LIBSVM$_{CBE}$, the performance gain over LIBSVM has been obtained by offloading computations to the vector coprocessors of the CBE, and hand-tuning the vector code running on these coprocessors. Current C compilers supporting standard OpenMP can not do that automatically.

In LIBSVM$_{CBE}$ we compute the values Q_{ij}, $j = m + 1, m + 2, \ldots, l$ in parallel across the SPEs. In particular, we partition the range $J = [m + 1, m + 2, \ldots, l]$ into K contiguous, non-overlapping sub-ranges $J_1, J_2, \ldots, J_K$ so that the computation of each Q_{iJ_k} is assigned to one of the available SPE[1]. Each SPE receives the following parameters: (i) the scalar y_i; (ii) the values y_{J_k}; (iii) the vector $\boldsymbol{x}_i$; (iv) the vectors $\boldsymbol{x}_{J_k}$. The value of K and the cardinality of each sub-range J_k are adaptively determined by the PPE so that all input data needed to compute Q_{iJ_k} fit into the SPE limited buffer space.

Another problem which must be addressed is to find an efficient encoding of the training vectors $\boldsymbol{x}_i$. Since vectors $\boldsymbol{x}_i$ are generally sparse, LIBSVM encodes them in a compact form, storing only the index and value of nonzero elements. In this way the memory required for storing the training data is greatly reduced. LIBSVM$_{CBE}$ uses a slightly different representation called 4-element sparse block [9]. Each block holds indexes and values of *four* contiguous elements, of which at least one value is nonzero; a termination block holding a negative index is used as a sentinel (see Fig. 2); the starting index of each block is a multiple of 4. In this way, four contiguous indexes and values fit in a pair of SPE registers, and the dot product required by the kernel evaluation can be done efficiently (see Appendix A for details).

Note that LIBSVM$_{CBE}$ differs from LIBSVM only in the implementation of the method `Qfloat* kernel::get_Q(int i, int l)` described above, and for the data structure used to encode sparse vectors.

[1]With abuse of notation, if $\boldsymbol{w}$ is a vector and A is a set of indices, we denote with $\boldsymbol{w}_A$ the subvector of $\boldsymbol{w}$ consisting of all elements whose positions are in A

Dataset	N	m	**Density**	**Av. nonzero**
`chess8_12K`	12000	2	100%	2.00
`mnist8n8-10k`	10000	779	20.65%	160.86
`uciadu6`	11220	122	11.37%	13.87
`web-a`	49749	300	3.88%	11.64
`rcv1_train_binary`	20242	47236	0.16%	75.58
`realsim-10k`	10000	20958	0.23%	48.20

Table 1: Datasets used in the experiments.

Everything else is exactly the same as in LIBSVM, including the shrinking heuristic and the caching strategy.

5. Experimental Results

In this section we analyze the performance of LIBSVM$_{CBE}$ by measuring the training time on the datasets listed in Table 1: N is the number of training vectors; m is the number of elements of each vector; *Density* is the average fraction of nonzero elements in each training vector (100% denotes fully dense vectors); finally, *Av. nonzero* is the average number of nonzero elements (computed as $m \times$ *Density*).

`chess8_12K` contains 12000 points which are randomly distributed over a 8×8 chessboard; each point is classified according to the color of the square containing it. The `mnist8n8-10k` dataset is a 10000 samples subset of the MNIST handwritten digits database (`http://yann.lecun.com/exdb/mnist`), containing 5000 samples of the digit "8" and 5000 samples of the other digits. The UCI Adult dataset [15] (`uciadu6`) allows to train a SVM to predict whether a household has an income greater than $50000. The Web dataset (`web-a`) [3] is related to the problem of classifying Web pages into topics according to keywords extracted from the pages themselves. `rcv1_train_binary` is a subset of the Reuters Corpus Volume 1 (RCV1) dataset [16], which consists of news stories which are classified according to their main topic. In `rcv1_train_binary` two classes are considered: one including news from the CCAT (Corporate/Industrial) or ECAT (Economics) main classification groups, and the other including news from the GCAT (Government/Social) or MCAT (Markets) groups. News belonging to both classes have been removed. Finally, `realsim-10k` is a subset of 5000 positive and 5000 negative samples from the `real-sim` data, which contains UseNet articles from four discussion groups: simulated auto racing, simulated aviation, real autos, real aviation. `uciadu6`, `web-a`, `rcv1_train_binary` and `real-sim` have been obtained from `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

LIBSVM$_{CBE}$ have been implemented and tested on a Sony PS3 running Yellow Dog Linux version 6.1 (kernel 2.6.23). The PS3 contains a 3.2GHz Cell processor (revision 5.1), with 6 SPEs available to the user. The system has 256MB of XDR RAM; all datasets have been chosen so that they fit entirely into the RAM. LIBSVM employs a caching strategy to store the last computed rows of Q in a cache. All tests on

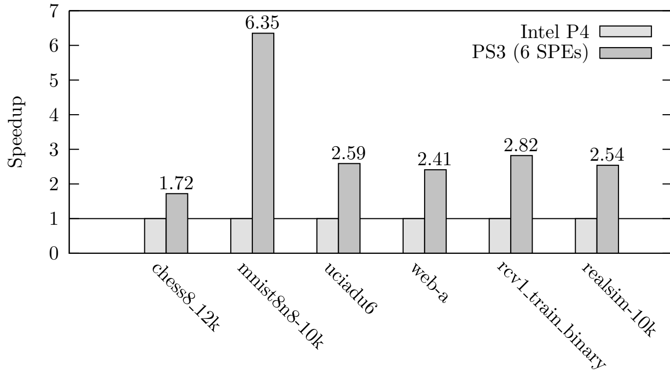


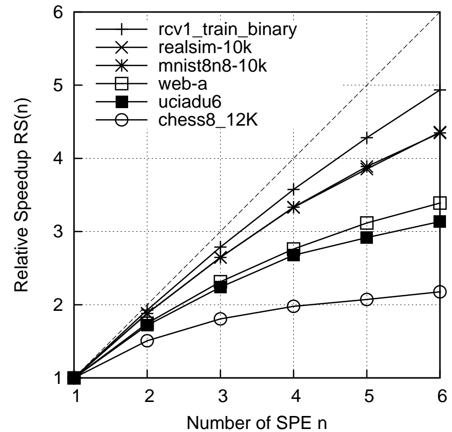
Figure 3: Speedup of LIBSVM$_{CBE}$ on the PS3 with respect to LIBSVM running on an Intel P4 processor ($T_{\text{CPU}}/T_{\text{SPE},6}$)

the PS3 were performed with a cache size of 40MB (command line option `-m 40`). LIBSVM$_{CBE}$ has been compiled with the GNU C compiler version 4.1.1 using the `-O3` flag (both for the PPE and SPE code). For all experiments we used the RBF kernel (Eq. (4)) with the default parameters. Tests of the sequential implementation use LIBSVM version 2.89 on an Intel Pentium 4 processor running at 2.4 GHz with 512KB of L1 cache and 1GB of RAM, under Linux kernel 2.6.28. LIBSVM was compiled with the GNU C compiler version 4.3.3 using the default compilation flags from the LIBSVM source distribution (`-Wall -Wconversion -O3 -fPIC`). For the sequential code we used the default LIBSVM cache size of 100MB. For each test we computed the average execution time of 5 independent runs. Training times have been measured by instrumenting the code with the `clock_gettime(2)` function using the `CLOCK_MONOTONIC` time source; preprocessing and input/output time has been excluded from the measurements.

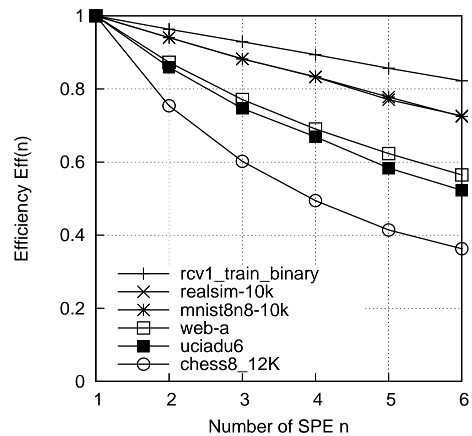
Training times are shown in Table 2. T_{CPU} denotes the training time of LIBSVM on the Intel P4 processor; T_{PPE} is the training time of LIBSVM$_{CBE}$ on the PS3 using the PPE only; $T_{\text{SPE},n}$ is the training time of LIBSVM$_{CBE}$ on the PS3 using n SPE. The *speedup* is computed as the ratio of the execution time on the CPU and the execution time on the Cell with 6 SPEs ($T_{\text{CPU}}/T_{\text{SPE},6}$).

The speedup of LIBSVM$_{CBE}$ with respect to the Intel P4 is shown in Figure 3. The larger speedup (6.35×) is achieved on the `mnist8n8-10k` dataset, which is the one with higher density (number of nonzero elements) and larger vector size. On average, each training vector of `mnist8n8-10k` has $779 \times 0.2065 \approx 160.86$ nonzero elements. Large and dense training vectors allow the CBE to perform larger DMA transfers to the SPEs, which can be handled more efficiently than small data transfers. Furthermore, large training vectors improve the computation/data transfer ratio, better exploiting the computational power of the SPEs. The smaller speedup (1.72×) is achieved on the `chess8_12K` dataset where all training vectors are two-dimensional.

To evaluate the scalability of LIBSVM$_{CBE}$ we consider the relative speedup $RS(n)$ with n SPEs, defined as $RS(n) = T_{\text{SPE},1}/T_{\text{SPE},n}$. It should be observe that $T_{\text{PPE}}/T_{\text{SPE},n}$ is not appropriate to measure the scalability of an application running on



(a) Relative Speedup



(b) Efficiency

Figure 4: Relative speedup 4(a) and efficiency 4(b) versus number of SPEs

the CBE, because the Cell processor has an asymmetric internal architecture. This means that The PPE and SPEs have different computational power, in particular the SPEs are optimized for vector computations for which the PPE is not as efficient.

Another metric we consider to assess the scalability of LIBSVM$_{CBE}$ is the *efficiency* $\mathit{Eff}(n)$, defined as $\mathit{Eff}(n) = RS(n)/n$. By definition, $0 \leq \mathit{Eff}(n) \leq 1$ for all n. The efficiency measures the fraction of time which is used by the n SPEs for actual computation. For example, if the efficiency is 0.5, then half the execution time is used in actual computation, while the rest is devoted to communication and synchronization overhead.

Figure 4(a) and 4(b) show the relative speedup and efficiency as a function of the number n of SPEs. Not surprisingly, the `rcv1_train_binary` dataset achieves the best scalability and efficiency. On the other hand the `chess8_12K` dataset exhibits the worst scalability and efficiency.

From Fig. 4(b) we observe that the communication and synchronization overhead increases with the number n of SPEs, due to contention on the EIB. This means that the speedup obtained by offloading the computational activity to the SPEs is ultimately limited by the memory latency of the data transfers. The number of DMA operations depends on the size and structure of the training dataset $\mathcal{D}$ and thus can not be predicted. In

	T_{CPU}	T_{PPE}	$T_{\text{SPE},1}$	$T_{\text{SPE},2}$	$T_{\text{SPE},3}$	$T_{\text{SPE},4}$	$T_{\text{SPE},5}$	$T_{\text{SPE},6}$	**Speedup** ($T_{\text{CPU}}/T_{\text{SPE},6}$)
`chess8_12K`	32.82	62.53	41.64	27.61	23.05	21.05	20.10	19.13	1.72
`mnist8n8-10k`	253.46	334.33	173.64	92.30	65.62	52.09	44.65	39.93	6.35
`uciadu6`	24.96	52.72	30.20	17.57	13.48	11.28	10.35	9.63	2.59
`web-a`	70.22	149.92	98.74	56.54	42.68	35.74	31.68	29.13	2.41
`rcv1_train_binary`	617.17	1674.64	1079.19	560.00	387.19	301.79	251.96	218.71	2.82
`realsim-10k`	107.76	279.62	185.07	98.38	69.91	55.57	48.02	42.46	2.54

Table 2: Wall-clock training time, in seconds (average of 5 runs, lower is better)

fact, the computations performed by each SPE is the dot product $\boldsymbol{x}_i{}^{\mathrm{T}}\boldsymbol{x}_j$, $j \in J_k$. The size of each J_k is dynamically computed by the PPE as the largest multiple of 16 such that all inputs needed to compute Q_{iJ_k} fit in the LS. Training vectors are transferred to the SPE using DMA List requests [14], because the vectors might not be contiguous in memory. DMA list operations are more inefficient than a single bulk DMA transfer.

However, it is possible to empirically estimate the worst-case scalability of LIBSVM$_{CBE}$ by looking at the `chess8_12K` dataset. This dataset exhibits a worst-case behavior, because training vectors have the smallest possible size: this maximizes the communication overhead, and reduces the computation/-communication ratio at each SPE. Despite that, `chess8_12K` still exhibits a modest (1.72×) speedup with respect to the sequential version, which is quite remarkable. This suggests that the Cell processor is effective in speeding up SVM training even for low-dimensional datasets.

6. Conclusions

In this paper we described LIBSVM$_{CBE}$, an optimized implementation of the SMO algorithm for the Cell processor. LIBSVM$_{CBE}$ is a modified version of LIBSVM which improves the most time-consuming step of the training process, that is the evaluation of the kernel function.

LIBSVM$_{CBE}$ has been tested on some widely used datasets; results show speedups up to 6.35× with respect to the sequential version. High speedups are achieved on datasets with dense, high dimensional training vectors. We remark that these are precisely the cases in which large speedups are desirable. Lower speedups have been observed on datasets with training vectors of low dimension. These datasets exhibit a worst-case behavior with respect to DMA transfer from main memory to LS, requiring the transfer of many small data blocks. In particular, for the `chess8_12K` dataset, which has two-dimensional training vectors and thus represents a worst-case scenario, we observe a speedup of 1.72×. This suggests that the Cell processor is effective in improving SVM training times even for low-dimensional datasets.

Acknowledgments. The author is grateful to Gaetano Zanghirati for suggesting this problem and for many useful discussions.

Software Availability. The source code of LIBSVM$_{CBE}$, including the datasets and scripts used to produce the results shown in this paper, is available at `http://www.cs.unibo.it/pub/marzolla/svmcell/`.

References

[1] V. David Sánchez A., Advanced support vector machines and kernel methods, Neurocomputing 55 (1–2) (2003) 5–20, support Vector Machines. doi:10.1016/S0925-2312(03)00373-4.

[2] V. N. Vapnik, Statistical Learning Theory, John Wiley and Sons, New York, 1998.

[3] J. C. Platt, Fast training of support vector machines using sequential minimal optimization, in: Advances in kernel methods: support vector learning, MIT Press, Cambridge, MA, USA, 1999, pp. 185–208.

[4] C.-C. Chang, C.-J. Lin, LIBSVM: a library for support vector machines, software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm` (2001).

[5] B. Catanzaro, N. Sundaram, K. Keutzer, Fast support vector machine training and classification on graphics processors, in: ICML '08: Proc. 25th international conference on Machine learning, ACM, 2008, pp. 104–111. doi:10.1145/1390156.1390170.

[6] L. Cao, S. Keerthi, C.-J. Ong, J. Zhang, U. Periyathamby, X. J. Fu, H. Lee, Parallel sequential minimal optimization for the training of support vector machines, IEEE Transactions on Neural Networks 17 (4) (2006) 1039–1049. doi:10.1109/TNN.2006.875989.

[7] G. Zanghirati, L. Zanni, A parallel solver for large quadratic programs in training support vector machines, Parallel Computing 29 (4) (2003) 535–551. doi:10.1016/S0167-8191(03)00021-8.

[8] L. Zanni, T. Serafini, G. Zanghirati, Parallel software for training large scale support vector machines on multiprocessor systems, Journal of Machine Learning Research 7 (2006) 1467–1492.

[9] M. Wyganowski, Classification algorithms on the cell processor, Master's thesis, Rochester Institute of Technology (Aug. 2008).

[10] K. Woodsend, J. Gondzio, Hybrid MPI/OpenMP parallel linear support vector machine training, Journal of Machine Learning Research 10 (2009) 1937–1953.

[11] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, H. Cui, PSVM: Parallelizing support vector machines on distributed computers, in: NIPS, 2007, software available at `http://code.google.com/p/psvm`.

[12] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, Z. Chen, P-packSVM: Parallel Primal grAdient desCent Kernel SVM, in: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, ICDM '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 677–686. doi:10.1109/ICDM.2009.29.

[13] R.-E. Fan, P.-H. Chen, C.-J. Lin, Working set selection using second order information for training support vector machines, Journal of Machine Learning Research 6 (2005) 1889–1918.

[14] IBM Corporation, Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor, Version 1.11 (May 12 2008).

[15] A. Asuncion, D. Newman, UCI machine learning repository, university of California, Irvine, School of Information and Computer Sciences, `http://www.ics.uci.edu/~mlearn/MLRepository.html` (2007).

[16] D. D. Lewis, Y. Yang, T. G. Rose, F. Li, RCV1: A New Benchmark Collection for Text Categorization Research, Journal of Machine Learning Research 5 (2004) 361–397.

Appendix A. Implementation details

Parallel computation of Q_{ij}. In this section we give details on how the elements of matrix Q can be computed in parallel across the SPEs. The function `Qfloat* kernel::get_Q(int i, int l)` of LIBSVM is used to compute Q_{ij}, for $j = 1, \ldots, l$. LIBSVM first checks whether some initial portion of row i is already stored in the cache. If only the first $m < l$ elements are available, then the missing values Q_{iJ}, $J = [m+1, m+2, \ldots, l]$ are computed. LIBSVM$_{CBE}$ optimizes the computation of Q_{iJ} by evaluating all elements in parallel across the SPEs.

First, we describe the memory layout of LIBSVM$_{CBE}$ program data, shown in Figure A.5. The value y_i is stored in `y_j[i]`[2]. `sz[i]` contains the number of blocks encoding the sparse vectors $\boldsymbol{x}_i$; the blocks are located at `x[i][0]` through `x[i][sz[i]-1]`. All sparse vectors $\boldsymbol{x}_i$, $i = 1, \ldots, N$ are stored in the array `x_space[]`. The C struct **svm_4node** represents a 4-element sparse block and is defined as:

```
typedef struct {
    vec_int4 index;
    vec_float4 val;
} svm_4node;
```

where the `vec_int4` and `vec_float4` types are vectors of four integer and float elements, respectively. The size of `vec_int4` and `vec_float4` is 128 bits, so they fit inside one SPE register.

Algorithm 3 is executed by the PPE to computes Q_{iJ}, $J = [m+1, m+2, \ldots, l]$. If J has less than 48 elements then the computation is done entirely by the PPE in order to avoid the communication and synchronization overhead (line 2). The threshold of 48 elements has been empirically determined.

If J has at least 48 elements, then the PPE distributes the computation of Q_{iJ} to the SPEs. To guarantee that the data which must be transferred to the SPEs is properly aligned, it is necessary to identify a new range [`jstart`, ..., `jend`] $\subseteq J$ such that the addresses of `y_j[jstart]` and `y_j[jend]` are aligned at 16B (quadword) boundary (lines 6–9). In fact, the Cell processor requires that the starting address of a DMA transfer is quadword-aligned, and also requires that the length of the transferred block is multiple of 16 bytes. Leading and trailing elements of Q_{iJ} are evaluated by the PPE (lines 11–14). Since all vectors are allocated at 16B boundary, then if `y_j[jstart]` is quadword-aligned, also the corresponding elements of all other vectors are aligned.

Now the range [`jstart`, ..., `jend`] is partitioned into K disjoint sub-ranges $J_1, J_2, \ldots, J_K$ (line 16), and the computation of Q_{iJ_k} is delegated to one of the SPEs. The number of elements of each J_k is computed by the PPE such that: (i) the input parameters needed to compute Q_{iJ_k} fit into the LS; (ii) the starting address of the memory block to transfer is quadword-aligned; (iii) the length of each block is a multiple of 16 bytes.

After having determined the partition $J_1, J_2, \ldots, J_K$, the PPE allocates a data structure containing a description of the tasks which will be executed by the SPEs. A *task descriptor* is a C struct containing the following items:

- The scalar value y_i (**signed char** yi);
- Effective Address (EA) of the first element of vector y_{J_k} (**uint64_t** yj_ead);
- The number of elements of J_k (**uint32_t** yj_size);
- EA of the first block of sparse vector $\boldsymbol{x}_i$ (**uint64_t** xi_ead);
- The number of blocks representing vector $\boldsymbol{x}_i$ (**uint32_t** xi_size);
- EA of the array of pointers to vectors $\boldsymbol{x}_{J_k}$ (**uint64_t** xj_ar_ead);
- EA of the array of sizes (number of blocks) of sparse vectors $\boldsymbol{x}_{J_k}$ (**uint64_t** xj_sz_ead);
- EA of the array where the result Q_{iJ_k} must be stored (**uint64_t** result_ead);
- The type of kernel function, and all additional parameters needed to evaluate it.

There are exactly K task descriptors $T_1, T_2, \ldots, T_K$, corresponding to the data needed to compute $Q_{iJ_1}, Q_{iJ_2}, \ldots, Q_{iJ_K}$. The task descriptors are stored contiguously in main memory as an array handled as a FIFO queue (task queue TQ). The address of TQ is sent to each SPE (line 19); at this point, the PPE waits for all SPEs to complete execution (line 21).

Algorithm 3 Computation of Q_{iJ}, PPE code

Require: Row index i
Require: Column indexes $J = [m+1, m+2, \ldots, l]$

```
1:  if (|J| < 48) then
2:     Compute Q_iJ on the PPE using Eq. (2)
3:  else
4:     jstart := m + 1
5:     jend := l
6:     while (y_j[jstart] is not quadword-aligned) do
7:        jstart := jstart + 1
8:     while (y_j[jend] is not quadword-aligned) do
9:        jend := jend − 1
10:    {Evaluate unaligned portion on the PPE}
11:    for j := m + 1, ..., jstart − 1 do
12:       Compute Q_ij on the PPE using Eq. (2)
13:    for j := jend + 1, ..., l do
14:       Compute Q_ij on the PPE using Eq. (2)
15:    {Evaluate aligned portion on the SPE}
16:    Define a partition J_1, J_2, ..., J_K of [jstart, ..., jend]
17:    Setup task queue TQ with K task descriptors T_1, T_2, ..., T_K
18:    for all SPE i do
19:       Send address of TQ to SPE i
20:    for all SPE i do
21:       Wait for SPE i to complete
```

[2] For the sake of simplicity, we ignore the fact that arrays in the C language are indexed from 0

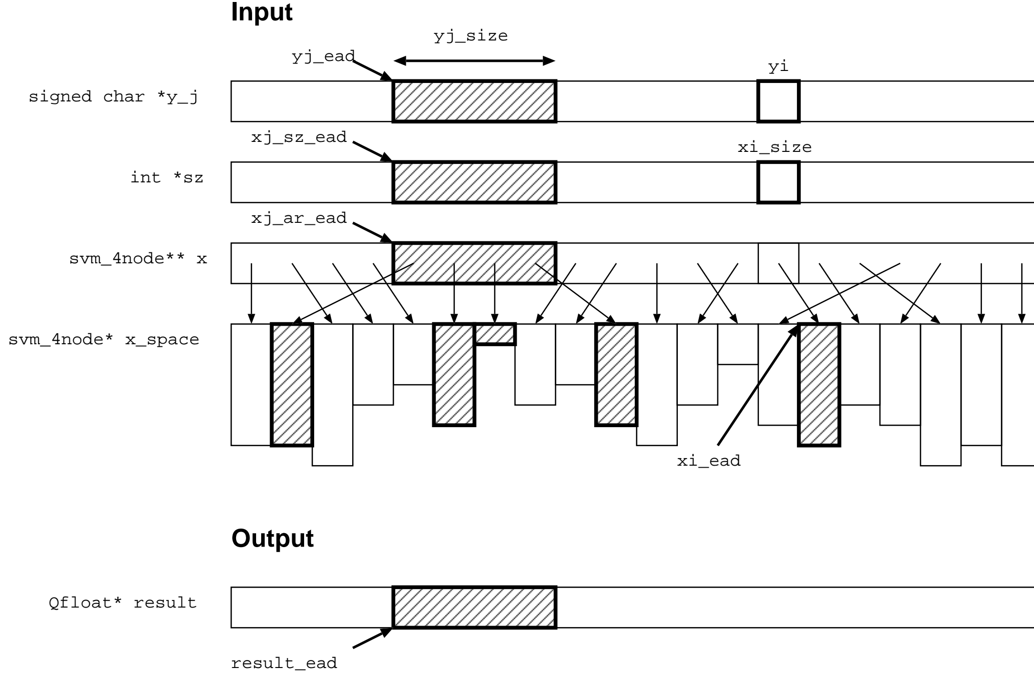


Figure A.5: *Input* represents the data transferred from main memory to LS for kernel evaluation; *Output* shows the results transferred back from LS to main memory.

Each SPE executes the pseudocode shown in Algorithm 4. First, the SPE receives the EA (main memory address) of the task queue TQ (line 2). Then, using atomic operations, one task descriptor T_k is removed from the task queue and copied into the LS (line 4). All data referenced by the task descriptor–the hatched area from Figure A.5–are copied from main memory to LS (line 5) using DMA operations. When the data transfer completes, the result Q_{iJ_k} is computed and put back to main memory (line 8) at the EA indicated by the `result_ead` field of the task descriptor. The steps above are repeated until the task queue becomes empty; at that point, each SPE signals the PPE that the computation is complete.

Algorithm 4 Computation of Q_{iJ}, SPE code

1: **loop**
2: Receive task queue address TQ
3: **while** (TQ not empty) **do**
4: Get task descriptor T_k from TQ
5: Get input data for T_k from main memory
6: **for all** $j \in J_k$ **do**
7: Compute Q_{ij} on the SPE using Eq. (2)
8: Put Q_{iJ_k} to main memory
9: Signal completion to the PPE

Computing vector dot product on the SPE. We now describe how to compute efficiently the dot product of two sparse vectors px and py on the SPE, using SIMD vector instructions. The dot product can be implemented by the SPE by multiplying the values of two blocks with the same index, and accumulating the result with the multiply-and-add SIMD instruction. The C language SPE implementation of the dot product is the following:

```
float spu_dot( const svm_4node* px, const svm_4node* py ) {
  const vec_int4 *pxidx = &(px->index);
  const vec_int4 *pyidx = &(py->index);
  const vec_float4 *pxval = &(px->val);
  const vec_float4 *pyval = &(py->val);
  vec_float4 result = spu_splats((float )0.0);
  int idx_x = spu_extract( *pxidx, 0 );
  int idx_y = spu_extract( *pyidx, 0 );
  while( idx_x != -1 && idx_y != -1 ) {
    if ( idx_x==idx_y ) {
      result = spu_madd(*pxval,*pyval,result);
      pxidx += 2; pxval += 2; pyidx += 2; pyval += 2;
      idx_x = spu_extract( *pxidx, 0 );
      idx_y = spu_extract( *pyidx, 0 );
    } else {
      if ( idx_x < idx_y ) {
        pxidx += 2; pxval += 2; idx_x = spu_extract( *pxidx, 0 );
      } else {
        pyidx += 2; pyval += 2; idx_y = spu_extract( *pyidx, 0 );
      }
    }
  }
  return spu_extract( result ,0) + spu_extract( result ,1) +
         spu_extract( result ,2) + spu_extract( result ,3);
}
```

Note the use of pointers rather than indexing expressions to compute the addresses of vector elements without using multiplications. `spu_splats(v,s)` sets all elements of $\boldsymbol{v}$ to the scalar value s. `spu_madd(v,w,r)` computes $r_i := r_i + v_i \times w_i$ for vectors $\boldsymbol{r} = [r_1, r_2, r_3, r_4]$, $\boldsymbol{v} = [v_1, v_2, v_3, v_4]$ and $\boldsymbol{w} = [w_1, w_2, w_3, w_4]$. `spu_extract(v,i)` returns the i-th element of vector $\boldsymbol{v}$.