Optimized Training of Support Vector Machines on the Cell Processor

Moreno Marzolla

Technical Report UBLCS-2010-02

February 2010

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

Optimized Training of Support Vector Machines on the Cell Processor

Moreno Marzolla[1]

Technical Report UBLCS-2010-02

February 2010

Abstract

Support Vector Machines (SVMs) are a widely used technique for classification, clustering and data analysis. While efficient algorithms for training SVMs 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 software tool which implements the Sequential Minimal Optimization algorithm. We describe* LIBSVM$_{CBE}$*, an optimized version of* LIBSVM *which takes advantage of the peculiar architecture of the Cell Processor. 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.*

1. Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy, Email: marzolla@cs.unibo.it

1 Introduction

SVMs are used in many classification tasks. The general problem they address can be stated as follows: let us consider a set of N data points which belong to two classes. The goal is to classify any new point in the appropriate class. In the case of SVM, each data point is a m-dimensional vector.

Formally, let us consider N vectors in m-dimensional space: $\mathbf{x}_i \in \mathbb{R}^m, i = 1, \dots N$. Vector $\mathbf{x}_i$ is associated with label $y_i \in \{-1, 1\}$. The set $\mathcal{D} = \{(\mathbf{x}_i, y_i) \ : \ i = 1, \dots N\}$ 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 α of a Quadratic Programming (QP) problem of the form:

$$
\begin{aligned}
\min_{\alpha} \quad & f(\alpha) = \frac{1}{2}\alpha^{\mathrm{T}} Q \alpha - \sum_{i=1}^{N} \alpha_i \\
\text{s.t.} \quad & 0 \le \alpha_j \le C, \quad j = 1, \dots N \\
& \mathbf{y}^{\mathrm{T}} \alpha = 0
\end{aligned}
\tag{1}
$$

where $\mathbf{y} = [y_1, y_2, \dots y_N]^{\mathrm{T}}$ and the entries Q_{ij} of the symmetric positive semidefinite matrix Q are defined as

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

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

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

and the Radial Basis Function (RBF) kernel:

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

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

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

where the offset b is computed during the training step.

The size of real-world datasets makes the solution of Eq. (1) using general purpose QP solvers impractical. For this reason, efficient solution strategies have been developed, which take advantage of the special structure of the problem (1). The Sequential Minimal Optimization (SMO) algorithm, originally proposed by Platt [9], decomposes the original QP problem into the smallest possible subproblems which can be solved analytically. The idea is to compute a solution iteratively by optimizing two coefficients α_i, α_j at the time. For this reason, SMO does not need to use a costly numerical QP solvers in its inner loop.

Unfortunately, training times are still significant for large datasets. To alleviate this problem, in this paper we show how the Cell Broadband Engine (CBE) can be used to improve the performance of the SMO algorithm. In particular, we describe and analyze LIBSVM$_{CBE}$, a Cell-optimized version of the LIBSVM software package [5]. LIBSVM is widely used, being very efficient as it employs several heuristics to reduce the training time of SMO. LIBSVM$_{CBE}$ improves the most computationally intensive part of LIBSVM, that is the evaluation of entries of the matrix Q from (2). We show how this optimization yields significant speedups over the sequential version on large, dense datasets.

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

In [4] the authors describe an optimized version of the SMO algorithm which makes use of 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 an 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 [3]. In [12, 13] the authors propose a parallel implementation for large quadratic programs which is based on a different solution technique–the Parallel Gradient Projection-based Decomposition Technique (PGPDT). Instead of optimizing two variable for each iteration, as the SMO algorithm does, the PGPDT decomposes the original QP problem in larger subproblems, and each subproblem is suitable to be solved on a cluster of workstations using MPI. In [11] the author shows how the CBE can be used to speed up the PGPDT solution technique.

To the best of our knowledge, no previous attempt has been made to optimize the SMO algorithm specifically for the CBE architecture. The SMO algorithm is attractive because it is simpler to implement than other SVM training algorithms. Many good open source implementations exist, and we optimized one of those to take advantage of the peculiar architecture of the Cell processor.

Organization of this paper This paper is organized as follows. In Section 2 we briefly describe the SMO algorithm. In Section 3 we illustrate the architecture of the CBE. In Section 4 we present LIBSVM$_{CBE}$, a Cell-optimized version of the LIBSVM software package; in particular, we describe how the evaluation of the kernel matrix Q (the most computationally intensive part of LIBSVM) has been optimized for the CBE. Section 5 evaluates LIBSVM$_{CBE}$ on some training datasets. Finally, conclusions and future works are illustrated in Section 6.

2 The SMO Algorithm

SMO is sketched in Algorithm 1; the reader is referred to [6] 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 to optimize is a crucial task, as it influences the convergence speed. The implementation we consider uses the *second order heuristic* proposed in [6].

We observe that Algorithm 1 requires to evaluate rows of matrix Q (we underlined the pseudocode statements where matrix 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. According to (2), the computation of Q_{ij} requires one kernel evaluation, which in turn requires at least one dot product of sparse vectors. The evaluation of elements of Q is the bottleneck of the SMO algorithm, and thus is the candidate operation to optimize.

One optimization is to avoid recomputing the elements of Q by keeping a cache of (partially filled) rows. When a row is requested, it is computed only if it is not already in the cache. Newly computed rows are inserted in the cache for future use; if the cache is full, an entry is evicted (for example, the least recently used one). Despite this optimization, code profiling of LIBSVM reveal that for some datasets, the evaluation of Q takes up to 95% of the total training time.

3 The Cell Processor

The CBE is an heterogeneous multicore processor, whose architecture is shown in Fig. 1. Specifically, the Cell is made of nine processors on a single chip, connected together with a high bandwidth circular bus [7].

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

Algorithm 1 Sequential Minimal Optimization

Main SMO Algorithm

```
**for all** $i = 1, 2, \dots N$
  $G_i \leftarrow 0$
  $\alpha_i \leftarrow -1$
**while** (true)
  Select $i, j$ (see next column)
  if $i = -1$
    Stop
  $\underline{a \leftarrow \max\{Q_{ii} + Q_{jj} - 2y_i y_j Q_{ij}, \tau\}}$
  $b \leftarrow y_i G_i + y_j G_j$
  $\alpha_i^{\text{old}} \leftarrow \alpha_i$
  $\alpha_j^{\text{old}} \leftarrow \alpha_j$
  $\alpha_i \leftarrow \alpha_i + y_i b/a$
  $\alpha_j \leftarrow \alpha_j - y_j b/a$
  $S \leftarrow y_i \alpha_i^{\text{old}} + y_j \alpha_j^{\text{old}}$
  Clip $\alpha_i$ to $[0, C]$
  $\alpha_j \leftarrow y_j(S - y_i \alpha_i)$
  Clip $\alpha_j$ to $[0, C]$
  $\alpha_i \leftarrow y_i(S - y_j \alpha_j)$
  **for all** $t = 1, 2, \dots N$
    $\underline{G_t \leftarrow Q_{ti}(\alpha_i - \alpha_i) + Q_{tj}(\alpha_j - \alpha_j)}$
```

Selection of i, j

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

the Operating System, allocating resources and distributing the workload to the other computing cores. The PPE includes 32 KB L1 instruction and data caches, and a 512 KB L2 cache. The PPE is a dual issue, in-order execution design, with two way simultaneous multithreading.

The eight Synergistic Processor Elements (SPEs) are SIMD processors optimized for data-intensive computations. Each SPE has two separate pipelines (odd and even), so that two SIMD instructions can be issued per cycle, provided that they can be executed on different pipelines. The SPE does not support instruction reordering nor dynamic branch prediction, but instead relies on user-provided or compiler-generated branch hints. Each SPE has a Local Store (LS) of 256 KB or RAM, which holds data and instructions. An SPE can access the main memory only using asynchronous DMA transfers from main memory to LS and back. DMA is 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.

While the Cell Processor has been initially developed for the consumer market (one of its first applications was inside Sony's PlayStation®3 gaming console), its potential for scientific applications has already been recognized [10]. However, its peculiar architecture requires that applications are specifically optimized to take advantage of the SPE [1]. In particular, compute-intensive tasks should be offloaded to the SPE, possibly taking advantage of the very powerful SIMD instructions provided by these units. Special care must be taken to ensure that data is properly aligned in memory to guarantee efficient DMA transfers. Peak DMA performance is achievable when both the Effective Address (EA) in main memory and the LS address are aligned at 128 bytes boundary, and the transfer size is a multiple of 128 bytes [7]. For this reason, Cell-optimized applications might require to use a carefully chosen memory layout for their internal working data.

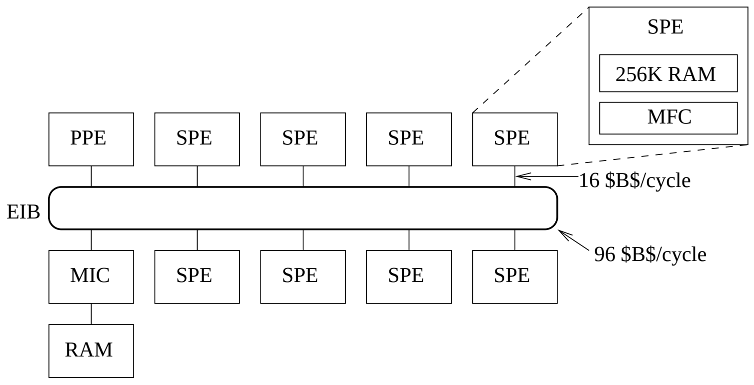


Figure 1. Architecture of the CBE

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

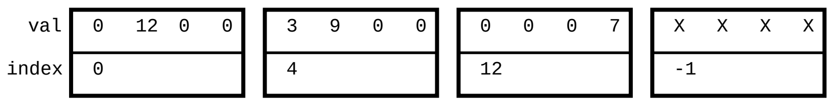


Figure 2. Encoding of a sparse vector whose nonzero elements are $12, 3, 9, 7$ **at positions** $1, 4, 5, 15$ **respectively. Structure svm_4node represents a block within the SPEs**

4 Optimizing LIBSVM for the Cell Processor

While the SMO algorithm can be easily described, writing an efficient implementation can be tricky. For this reason, our starting point is an an existing open source implementation of SMO, namely LIBSVM version 2.89 [5]. By profiling LIBSVM, it turns out that the most time-consuming operation (responsible for up to 90% of the total wall-clock training time for some datasets) is the kernel evaluation executed by the Qfloat* kernel::get_Q(**int** i, **int** l) method. This method returns a pointer to an array containing the values $Q_{ij} = y_i y_j K(x_i, x_j)$ for $j = 1, 2, \dots l$. We optimize this function by computing the elements Q_{ij} in parallel across the available SPEs. We denote with LIBSVM$_{CBE}$ the modified version of LIBSVM. Note that LIBSVM$_{CBE}$ differs from LIBSVM only in the implementation of the method kernel::get_q() and for the data structure used to encode sparse vectors.

The first problem which must be addressed is to find an efficient way to encode the sparse vectors of the training set. Keeping all vectors in memory is in general not possible, because the training set can be large. For this reason, LIBSVM encodes training vectors in sparse form, storing only the index and value of nonzero elements. LIBSVM$_{CBE}$ uses a slightly different representation called 4-element sparse block [11]. Training vectors are stored as sequences of blocks of type `svm_4node` (see Fig. 2), each block holding four consecutive index/value pairs where at least one value is nonzero.

The representation of this structure takes advantage of SPE vector data types: `vec_int4` and `vec_float4` are vectors of four integer and float elements, respectively. The size of `vec_int4` and `vec_float4` is 128 bits. Each sparse vector has a termination block in which all index elements are set to -1 (see Fig. 2 for an example). The dot product of two sparse vectors 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 reported in Appendix A.

Given that each element the matrix Q can be computed independently from the others, the idea is to partition the range $I = [s+1, s+2, \dots l]$ into contiguous, non-overlapping sub-ranges $I_1, I_2, \dots I_k$. Let Q_{iI_t} denote the set of values $\{Q_{ij} \ : \ j \in I_t\}$, for $t = 1, 2, \dots k$[2]. The computation

2. To simplify the notation, if $\mathbf{w}$ is a vector and A is a set of indices, we let $\mathbf{w}_A = \{w_i \ : \ i \in A\}$

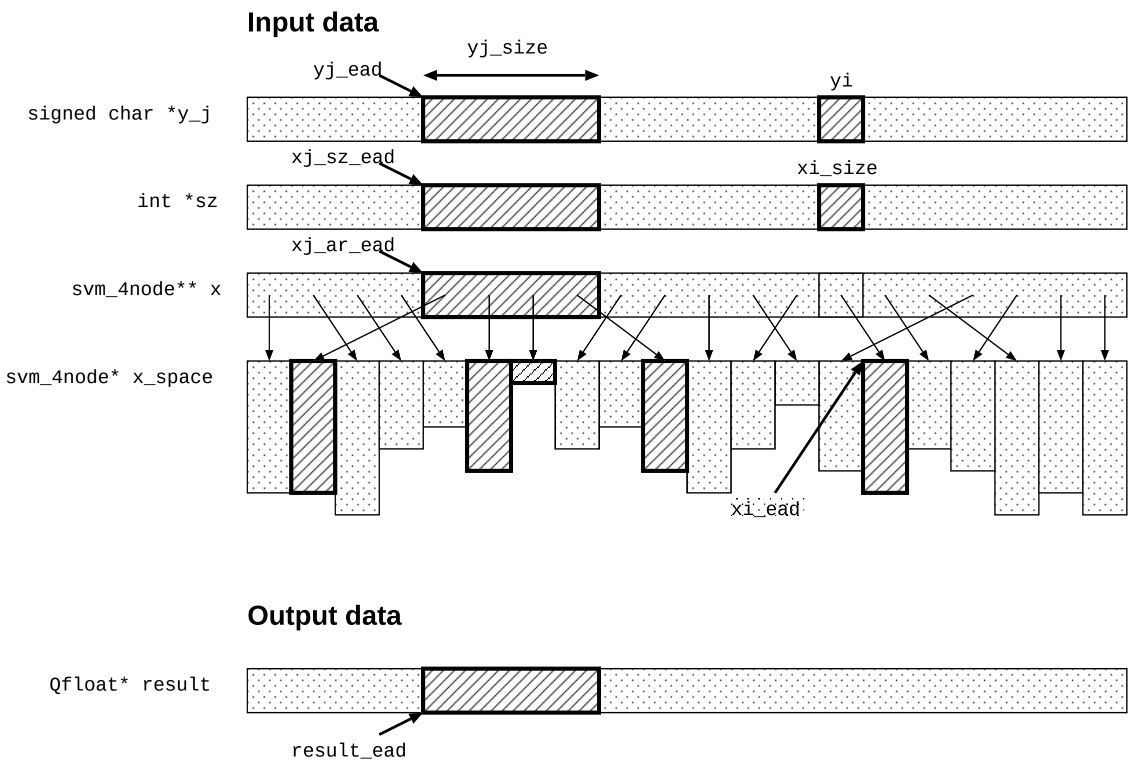


Figure 3. Data transferred from main memory to LS for kernel computation

of each Q_{iI_t} is assigned in round-robin fashion to one of the available SPE. The number of elements in each I_t is determined by the PPE as the largest multiple of 16 such that all input data needed to compute Q_{iI_t} fit into the SPE buffer space.

Each SPE receives the following parameters:

- The scalar value y_i (**signed char** yi);
- EA of vector y_{I_t} (**uint64_t** yj_ead);
- Length $|I_t|$ (number of elements) of vector y_{I_t} (**uint32_t** yj_size);
- EA of sparse vector $\mathbf{x}_i$ (**uint64_t** xi_ead);
- Number of blocks representing vector $\mathbf{x}_i$ (**uint32_t** xi_size);
- EA of the array of pointers to vectors $\mathbf{x}_{I_t}$ (**uint64_t** xj_ar_ead);
- EA of the array of sizes (number of blocks) of sparse vectors $\mathbf{x}_{I_t}$ (**uint64_t** xj_sz_ead);
- EA of the array where the result Q_{iI_t} must be stored (**uint64_t** result_ead);

Fig. 3 shows which data is moved from main memory to LS. The value of yj_size (that is, the number $|I_t|$ of training vectors to transfer for each block) is determined by the PPE as the largest multiple of 16 such that the size of all input data fits into the SPEs local store. Due to the small dimension of the code executed in each SPE, about 160KB of LS are available for data. We experimented with double buffering [1] (managing two buffers on the SPE, such one buffer can be processed while the other is being transferred), but observed that it does not provide any benefit, as the bottleneck is the data transfer rather than the computations performed on the SPE.

Dataset	Size	Dimension	Density	Avg. 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
`reuters`	7770	8315	0.52%	43.24

Table 1. Summary of the datasets used in the experiments. *Size* **is the number N of training vectors;** *Dimension* **is the total number m of elements of each vector;** *Density* **is the average fraction of nonzero elements in each training vector (100% denotes fully dense training vectors);** *Avg. nonzero* **is the average number of nonzero elements.**

Vectors $\mathbf{x}_I$ are in general not contiguous in memory because LIBSVM swaps indexes during the optimization step: so, vectors $\mathbf{x}_{I_t}$ must be transferred using DMA list requests. If y_I is not quadword-aligned, or if its size is a multiple of 16B, the PPE takes care of computing the leading and trailing elements of Q_{iI}.

It is important to observe that, while the SPEs are fully IEEE754 compliant when operating in double precision, they are not when operating in single precision arithmetic [7]; in particular, only *round-towards-zero* is implemented, as opposed to *round-to-even* mode (the PPE fully supports IEEE754 arithmetic). This means that when using the SPEs, LIBSVM$_{CBE}$ produces slightly different results than LIBSVM.

5 Experimental Results

In this section we describe the performance of LIBSVM$_{CBE}$ by measuring the training time on some well known datasets, which are listed in Table 1. `chess8_12K` contains 12000 points which are randomly distributed over a 8×8 chessboard, which are classified according to the color of the square containing it. The `mnist8n8-10k` dataset contains 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 [2] (`uciadu6`) allows to train a SVM to predict whether a household has an income greater than $50000. The Web dataset (`web-a`) [9] is related to the problem of classifying Web pages into topics according to keywords extracted from the pages themselves. The `rcv1_train_binary` dataset is a subset of the Reuters Corpus Volume 1 (RCV1) dataset [8], 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. The `uciadu6`, `web-a` and `rcv1_train_binary` are available from `http://www.csie.ntu.edu.tw/~cjlin/`. Finally, `reuters` is another subset of the Reuters dataset, available from `http://www.cse.ust.hk/~ivor/cvm.html`.

LIBSVM$_{CBE}$ have been executed on a Sony PlayStation®3 (PS3) machine 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; due to the limited size of the available memory, we only considered datasets which fit into the RAM. LIBSVM employs a caching strategy to keep the last computed rows of Q in memory. All tests on the PS3 were performed with the internal cache set to 40MB (command line option `-m 40`); tests of the scalar code used the default value of 100MB. 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 as used by LIBSVM. Tests of the scalar code use LIBSVM version 2.89 on an Intel Pentium 4 processor running at 2.4 GHz with 512KB of L1

	T_{CPU}	T_{PPE}	$T_{\mathrm{SPE},1}$	$T_{\mathrm{SPE},2}$	$T_{\mathrm{SPE},3}$	$T_{\mathrm{SPE},4}$	$T_{\mathrm{SPE},5}$	$T_{\mathrm{SPE},6}$	**Speedup**
`chess8_12K`	35.36	63.57	41.51	27.57	22.40	22.55	21.86	19.07	1.85
`mnist8n8-10k`	255.34	339.53	176.67	95.89	69.35	56.34	48.56	44.02	5.80
`uciadu6`	26.28	53.40	30.78	17.93	13.86	11.45	10.53	9.62	2.73
`web-a`	72.71	150.32	98.49	56.59	42.65	35.84	31.72	29.16	2.49
`rcv1_train_binary`	632.70	1674.72	1088.99	569.33	396.31	310.41	260.71	226.99	2.79
`reuters`	9.15	23.68	17.16	9.44	6.99	5.69	4.93	4.48	2.05

Table 2. Wall-clock training time in seconds (average of 5 measurements, lower is better). The speedup is computed as $T_{\mathrm{CPU}}/T_{\mathrm{SPE},6}$, higher is better.

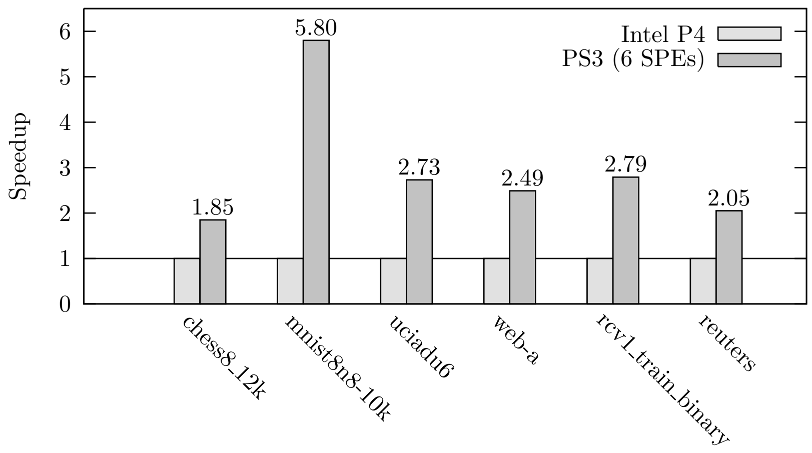


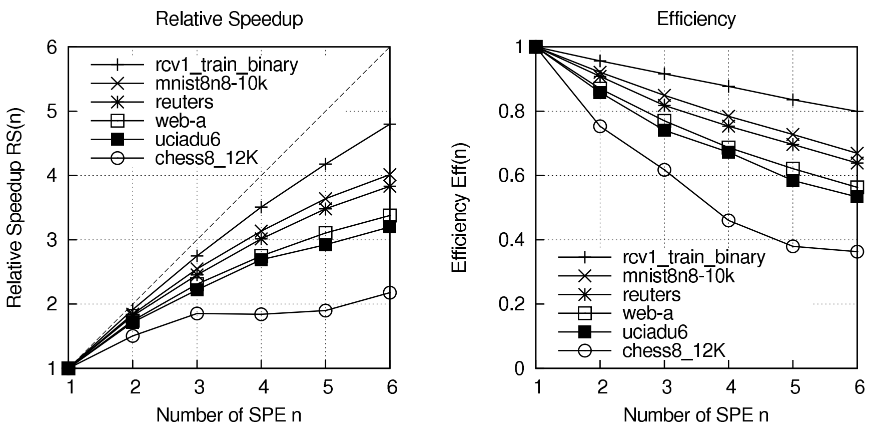
Figure 4. Speedup of LIBSVM$_{CBE}$ on the PS3 with respect to the sequential version running on an Intel P4 processor ($T_{\mathrm{CPU}}/T_{\mathrm{SPE},6}$)

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 specified in the LIBSVM source distribution (`-Wall -Wconversion -O3 -fPIC`). For all tests we consider the average execution time of 5 independent runs, as measured by the `time(1)` Unix command.

Performance results are shown in Table 2. T_{CPU} denotes the execution time of LIBSVM on the Intel P4 processor; T_{PPE} is the execution time of LIBSVM$_{CBE}$ on the PPE only; $T_{\mathrm{SPE},n}$ is the execution time of LIBSVM$_{CBE}$ on the PS3 using n SPE. The *speedup* is computed with respect to the sequential version on the Intel P4 ($T_{\mathrm{CPU}}/T_{\mathrm{SPE},6}$).

Figure 4 shows the speedup achieved by LIBSVM$_{CBE}$ with 6 SPEs with respect to LIBSVM on the Intel P4. The larger speedup (5.80) is obtained on the `mnist8n8-10k` dataset: this dataset is the one with higher density (number of nonzero elements) and larger vector dimension. On average, each training vector of `mnist8n8-10k` has $779 \times 0.2065 \approx 160.85$ 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 transfers. Furthermore, large training vectors improve the computation/data transfer ratio, better exploiting the computational power of the SPEs. The smaller speedup (1.85) is achieved on the `chess8_12K` dataset where all training vectors are very small (2 elements each). The `chess8_12K` dataset has been chosen to demonstrate the limited efficiency of LIBSVM$_{CBE}$ on low dimensional vectors, as `chess8_12K` is probably not representative of any real world training dataset.

To evaluate the scalability of LIBSVM$_{CBE}$ we consider the relative speedup $RS(n)$ with n SPEs, defined as $RS(n) = T_{\mathrm{SPE},1}/T_{\mathrm{SPE},n}$, and the efficiency $\mathit{Eff}(n)$, defined as $\mathit{Eff}(n) = RS(n)/n$. Figure 5(a) and 5(b) show the relative speedup and efficiency as a function of the number n of SPEs used. Not surprisingly, the best efficiency is achieved by the `rcv1_train_binary` dataset. On the other hand the `chess8_12K` dataset exhibits the worst scalability, because as we already pointed out it represents a worst-case scenario for LIBSVM$_{CBE}$, having very short training vec-



(a) (b)

Figure 5. Relative speedup and efficiency versus number of SPEs (curves are labeled from top to bottom).

tors.

It turns out that LIBSVM$_{CBE}$ is communication bound: this means that the speedup obtained by offloading the computational activity to the SPEs is limited by the memory latency which is experienced to transfer the input data to the SPEs, and the results back to main memory. The computations performed by each SPE is the dot product of sparse vectors $\mathbf{x}_i \mathbf{x}_j$, $j \in I_t$. The number of vectors which can be transferred ($|I_t|$) is computed by the PPE as the larger multiple of 16 such that all inputs needed to compute Q_{ij} fit in the LS. All input vectors are transferred using DMA List requests [7], as they might not be contiguous in memory. DMA list transfers are slower than a single DMA transfer of a large contiguous data block; in Fig. 5(b) it is possible to observe the bad impact of memory fragmentation on low dimensional or highly sparse datasets.

6 Conclusions

In this paper we described LIBSVM$_{CBE}$, an optimized implementation of the SMO algorithm for the CBE. We modified LIBSVM to improve 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 between 1.85 and 5.80 with respect to the sequential version running on an Intel P4 processor. 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.

LIBSVM$_{CBE}$ optimizes the computation of the matrix Q_{ij}, which is the primary bottleneck of the LIBSVM program. However, there are other steps of the SMO algorithm which could be parallelized as well. Future extensions of this work will be devoted to explore the effects of parallelizing the whole second order selection heuristic instead of just the computation of Q.

Acknowledgements The author is grateful to Gaetano Zanghirati for suggesting this problem and for many useful discussions. The source code of LIBSVM$_{CBE}$ and all datasets and scripts used to produce the results described in this paper, is available from `http://www.moreno.marzolla.name/software/svmcell`.

References

[1] Abraham Arevalo, Ricardo M. Matinata, Maharaja Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, and Chris Almond. *Programming the Cell Broadband Engine Architecture–Examples and Best Practices*. IBM Redbooks. IBM International Support Organization, August 2008.

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

[3] L.J. Cao, S.S. Keerthi, Chong-Jin Ong, J.Q. Zhang, U. Periyathamby, Xiu Ju Fu, and H.P. Lee. Parallel sequential minimal optimization for the training of support vector machines. *IEEE Transactions on Neural Networks*, 17(4):1039–1049, July 2006.

[4] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML '08: Proc. 25th international conference on Machine learning*, pages 104–111. ACM, 2008.

[5] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[6] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, 6:1889–1918, 2005.

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

[8] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. RCV1: A New Benchmark Collection for Text Categorization Research. *J. of Machine Learning Research*, 5:361–397, 2004.

[9] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[10] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[11] Mateusz Wyganowski. Classification algorithms on the cell processor. Master's thesis, Rochester Institute of Technology, August 2008.

[12] Gaetano Zanghirati and Luca Zanni. A parallel solver for large quadratic programs in training support vector machines. *Parallel Computing*, 29(4):535–551, 2003.

[13] Luca Zanni, Thomas Serafini, and Gaetano Zanghirati. Parallel software for training large scale support vector machines on multiprocessor systems. *Journal of Machine Learning Research*, 7:1467–1492, 2006.

A Sparse Dot Product on the SPE

The following C function computes the dot product of two sparse vectors `px` and `py` using the SPE. 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 v to the scalar value s. spu_madd(v,w,r) computes $r_i \leftarrow r_i + v_i \times w_i$ for vector elements r, v and w. spu_extract(v,i) returns the i-th scalar element of v.

```
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);
}
```