High-Performance Incremental SVM Learning on Intel® Xeon Phi™ Processors

Dipanjan Sengupta(✉), Yida Wang, Narayanan Sundaram, and Theodore L. Willke

Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95054, USA
{dipanjan.sengupta,yida.wang,narayanan.sundaram,ted.willke}@intel.com

Abstract. Support vector machines (SVMs) are conventionally batch trained. Such implementations can be very inefficient for online streaming applications demanding real-time guarantees, as the inclusion of each new data point requires retraining of the model from scratch. This paper focuses on the high-performance implementation of an accurate incremental SVM algorithm on Intel® Xeon Phi™ processors that efficiently updates the trained SVM model with streaming data. We propose a novel *cycle break* heuristic to fix an inherent drawback of the algorithm that leads to a deadlock scenario which is not acceptable in real-world applications. We further employ intelligent caching of dynamically changing data as well as other programming optimization ideas to speed up the incremental SVM algorithm. Experiments on a number of real-world datasets show that our implementation achieves high performance on Intel® Xeon Phi™ processors (1.1 – 2.1× faster than Intel® Xeon® processors) and is up to 2.1× faster than existing high-performance incremental algorithms while achieving comparable accuracy.

Keywords: High-performance · Incremental SVM · Intel Xeon Phi processor

1 Introduction

Support Vector Machine (SVM) [24] has established itself as one of the most popular and successful methods in example-based learning as an effective pattern classification tool where after training on a series of examples, the resulting model can generalize well on new input samples. Conventionally, SVMs are trained in batch mode, which can be formulated as a quadratic optimization problem. Several special-purpose optimization algorithms have been proposed for batch SVM learning, among which Sequential minimal optimization (SMO) [15] is one of the most commonly used. SVMs are widely applied to many application areas from scientific computing such as neuroscience and bioinformatics to Internet-based information retrieval like text classification, etc.

Intel, Xeon and Intel Xeon Phi are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

J.M. Kunkel et al. (Eds.): ISC High Performance 2017, LNCS 10266, pp. 120–138, 2017.
DOI: 10.1007/978-3-319-58667-0_7

The batch training algorithms assume that the training set is fixed. If there is any sample addition/deletion/modification on the training set, the batch algorithms have to retrain the model on the entire data set from scratch to produce a new model, which is inefficient and expensive. However, in many cases datasets are under change. For example, when training on a data stream, the model should be updated incrementally after getting a new sample (incremental training). Also, an efficient way to do cross validation on a dataset is to train a model on all data and update it by removing different sets of left-out samples (decremental training).

Standardized packages for batch SVM training such as $LibSVM$ [4] and SVM^{light} [11] have been around for years. There are also highly optimized implementations on top of them targeted towards many-core architectures like GPU [3] and the first generation of Intel Xeon Phi products [25]. However, incremental SVM algorithms are not widely popular in machine learning community because there are no efficient implementations of the proposed algorithms readily available for use.

An accurate incremental and decremental SVM learning algorithm has been previously proposed by Cauwenberghs and Poggio [16] and their approach was adapted to other variants of kernel machines [12,13]. When a single sample is added (or removed) this algorithm updates the exact optimal solution recursively without retraining it from scratch. The key idea is to retain the Karush-Kuhn-Tucker (KKT) conditions [24] on all previously trained samples, while adding (or removing) a new sample to the solution. In principle, this is better than other incremental algorithms such as [2,21] which tweak the model without globally optimal solution guarantee. A detailed comparison to related work is in Sect. 6.

Although theoretically possible, there are several challenges to implement a practical, especially high-performance version of incremental SVM algorithm that can be used by potential practitioners in real-world applications on modern many-core architectures. First, an incremental SVM algorithm is usually a multi-stage solution with varying compute and memory requirements. Therefore, a per stage detailed analysis of compute efficiency and memory access pattern is required to design well-tailored data structures and intelligent computation techniques to achieve maximal performance. Moreover, the algorithm involves multiple branching cases (Sect. 3.3) which makes it quite challenging to parallelize. Second, the incremental algorithm may have inherent limitations for convergence in particular scenarios where it fails to making progress as trapping into an infinite loop [12]. In real-world applications such behavior is not acceptable and requires to be handled intelligently. Third, as the algorithm dynamically updates the model, i.e. the support vector set, with each insertion (or deletion) of a sample, in order to get desired running performance, it needs an efficient data caching mechanism to deal with the dynamic change of the support vector set and other corresponding data structures.

In this paper, we propose and implement a high-performance incremental SVM algorithm that runs efficiently on Intel Xeon Phi processors based on Intel Many Integrated Core architecture (referred to hereinafter as *MIC processors*).

We are in the progress of open-sourcing the code. In brief, this paper makes following key contributions:

1. We propose *cycle-breaking*, a practical heuristic to avoid scenarios where the incremental algorithm stops making progress (happening on average once every 27 samples - Sect. 5.3).
2. We conduct several programming optimizations. Caching dynamically changing data results in up to 3.3× speedup for the overall application. Intelligently using efficient data structures and memory access patterns tailored for each stage of the incremental algorithm towards the MIC processors further gives an overall speedup of up to 1.5×.
3. Compared to existing incremental SVM algorithms such as warm-start SMO (described and discussed in Sects. 5.1 and 6), our algorithm is up to 2.1× faster. Our implementation is faster than warm-start SMO for over 90% of samples.
4. Our incremental SVM training algorithm optimized for the MIC processors is up to 1.3× faster than running on the Intel Xeon processors. For performing Leave-One-Out cross validation using the decremental variation of our incremental algorithm, running on the MIC processors is up to 2.1× faster than Intel Xeon processors.

The rest of the paper is organized as follows: Sect. 2 gives a brief overview of the MIC processors. Section 3 explains the incremental SVM training algorithm and our algorithmic contributions. Section 4 explains our optimization ideas for improving the performance on many-core architectures. Section 5 discusses the results and shows how we outperform other batch and incremental SVM algorithms without sacrificing accuracy. We compare our work to the related work in Sect. 6 and conclude in Sect. 7.

2 Intel® Xeon Phi™ Processors

The Intel® Xeon Phi™ processor is based on the Intel® Many Integrated Core (MIC) architecture. Unlike the graphic processing units (GPUs), this many-core processor provides a general-purpose programming environment similar to that of a regular Intel Xeon processor.

We describe the high-level architecture of Intel Xeon Phi processor 7250 (formerly codenamed Knights Landing or KNL) used in this paper. This processor has 68 cores, each of which runs at a processor base frequency of 1.40 GHz and supports up to 4 hardware threads. The cores are tiled in pairs, with each core having 32 KB L1 data cache, 32 KB L1 instruction cache and 1 MB unified L2 cache shared within the tile. The tiles are interconnected via 2D mesh. Cache coherence across cores/tiles is maintained via a global-distributed tag directory provided by caching/home agent (CHA). In this paper, the tiles are clustered in *quadrant mode* [20] for better performance and productivity trade-off.

Each core has two 512-bit vector processing units (VPUs), which allows 16 single precision or 8 double precision floating point numbers to be processed in a single CPU cycle. This makes vectorization challenging and critical to the performance. The theoretical peak floating point performance of the processor is 6.10 TFLOPS for single precision and 3.05 TFLOPS for double precision.

This processor has 16 GB multi-channel DRAM (MCDRAM) with bandwidth larger than 400 GB/s, and supports up to 384 GB DDR4 memory. In order to have the full control of the MCDRAM usage, in this paper we configured the memory subsystem in *flat mode* [20], working as a NUMA system in which MCDRAM serves the local memory to all cores.

3 Algorithm

In this section, we start from the SVM batch training to describe the incremental SVM algorithm we implemented in detail. We also highlight the changes we made to scale up the incremental algorithm to practical problem sizes on the MIC processors.

3.1 Support Vector Machine and KKT Conditions

Suppose we have a set of training data and their labels given by $\mathrm{T} = \{(x_i, y_i), i = 1 \ldots m\}$, where $x_i \in \mathbb{X} \subseteq \mathbb{R}^d$ is the input, $y_i \in \{-1, +1\}$ is the corresponding output label. We can write out the classification function as:

$$f(x) = w^T \Phi(x) + b \tag{1}$$

where $\Phi(x)$ is a fixed feature space transformation mapping the input x to a vector in feature space F. The model parameters can be obtained by solving the optimization problem:

$$\begin{aligned} \max_{\alpha} \ \mathbb{W}(\alpha) &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} Q_{ij} \alpha_i \alpha_j \\ s.t. \sum \alpha_i y_i &= 0 \\ 0 \leq &\alpha_i \leq C, \ i = 1, \ldots, m \end{aligned} \tag{2}$$

where $C \in \mathbb{R}^+$ is the regularization parameter that controls the relative weighting between maximizing the margin and minimizing the error rate, and $Q_{ij} = y_i y_j K(x_i, x_j)$ is the kernel matrix where $K(x_i, x_j) = \Phi(x_i)^T \Phi(x_j)$ is a kernel function [19].

Given the solution to (2), the optimal classification function $f : \mathbb{X} \to \mathbb{R}$ in formula (1) can be written as $f(x) = \sum_{i=1}^{m} \alpha_i y_i K(x_i, x_j) + b$.

From the Karush-Kuhn-Tucker (KKT) conditions, the margin function $g(x_i) = y_i f(x_i) - 1$ and the corresponding α_i must satisfy the following relationship at the optimal solution:

$$\begin{aligned} g(x_i) &\geq 0; \ \alpha_i = 0 \\ g(x_i) &= 0; \ 0 < \alpha_i < C \\ g(x_i) &\leq 0; \ \alpha_i = C \end{aligned} \tag{3}$$

This partitions the samples in training set T into three categories. Let us define the following partitioned index sets:

$$\begin{aligned} S &:= \{i : y_i f(x_i) = 1, 0 < \alpha_i < C\} \text{ (On-margin support vectors)} \\ E &:= \{i : y_i f(x_i) \leq 1, \alpha_i = C\} \text{ (Error support vectors)} \\ R &:= \{i : y_i f(x_i) \geq 1, \alpha_i = 0\} \text{ (Within-margin vectors)} \end{aligned} \tag{4}$$

The incremental SVM algorithm essentially moves the samples across these three sets to reach an optimal solution.

3.2 Incremental SVM Algorithm

The incremental SVM algorithm updates the previously trained SVM model with the inclusion of a new sample point (x_c, y_c) to the training set T instead of batch training the entire training set plus the new sample point. Figure 1 shows the complete software workflow of incremental SVM including the optimizations proposed in this paper. The key idea of the algorithm is to change the coefficient α_c (initialized to 0) corresponding to the new sample x_c in discrete steps with largest possible increments under the constraint that the change is small enough to keep other elements in training set T satisfying the KKT optimality conditions, i.e. no old training samples move across S, E or R sets. The update ends when the new sample satisfies the KKT optimality conditions.

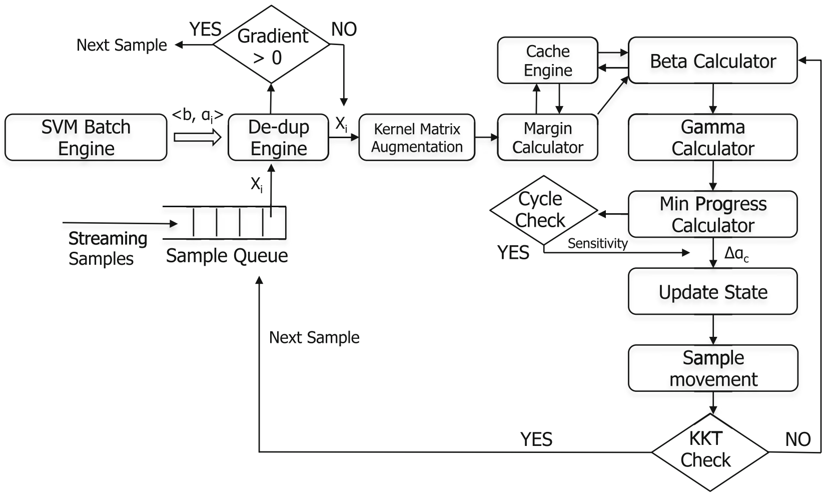

Fig. 1. Software workflow of incremental SVM.

Let us define the indices of the samples in the margin set S as $\{s_1, s_2, \ldots, s_{l_{\mathbb{S}}}\}$. As derived in [16], we need to solve:

$$\mathbb{Q}\begin{bmatrix}\Delta b\\ \Delta\alpha_{s_1}\\ \vdots\\ \Delta\alpha_{s_{l_{\mathbb{S}}}}\end{bmatrix} = -QSc\cdot\Delta\alpha_c \tag{5}$$

$$\text{where } \mathbb{Q} = \begin{bmatrix}0 & y_{s_1} & \cdots & y_{s_{l_{\mathbb{S}}}}\\ y_{s1} & Q_{s_1s_1} & \cdots & Q_{s_1s_{l_{\mathbb{S}}}}\\ \vdots & \vdots & \ddots & \vdots\\ y_{s_{l_{\mathbb{S}}}} & Q_{s_{l_{\mathbb{S}}}s_1} & \cdots & Q_{s_{l_{\mathbb{S}}}s_{l_{\mathbb{S}}}}\end{bmatrix} \text{ and } \; QSc = \begin{bmatrix}y_c\\ Q_{s_1c}\\ \vdots\\ Q_{s_{l_{\mathbb{S}}}}\end{bmatrix} \tag{6}$$

If we define $\beta = -R\cdot QSc$ with $R = \mathbb{Q}^{-1}$ then the bias and coefficients can be expressed in terms of $\Delta\alpha_c$ as:

$$\Delta b = \beta_0\Delta\alpha_c \tag{7}$$

$$\Delta\alpha_j = \beta_j\Delta\alpha_c, \quad \forall j\in S \tag{8}$$

The margin for all the sample points change according to:

$$\begin{aligned}\Delta g(x_i) &= \gamma_i\Delta\alpha_c\ \forall \in T\ \cup\{c\}\\ where,\ \gamma_i &= Q_{ic} + \sum_{j\in S} Q_{ij}\beta_j + y_i\beta_0\\ and\ \gamma_i &= 0\ \forall i\ \in S\end{aligned} \tag{9}$$

As γ_i is non-zero if $i \notin S$ set, we define $N = \{E\ \cup R\} = \{n_1, n_2, \ldots, n_{l_{\mathbb{N}}}\}$ and rewrite γ in matrix form:

$$\begin{aligned}\gamma &= \begin{bmatrix}Q_{n_1c}\\ Q_{n_2c}\\ \vdots\\ Q_{n_{l_{\mathbb{N}}}c}\end{bmatrix} + QNS\cdot\beta + \begin{bmatrix}y_{n_1}\\ y_{n_2}\\ \vdots\\ y_{n_{l_{\mathbb{N}}}}\end{bmatrix}\beta_0\\ where,\ QNS &= \begin{bmatrix}Q_{n_1s_1} & \cdots & Q_{n_1s_{l_{\mathbb{N}}}}\\ \vdots & \ddots & \vdots\\ Q_{n_{l_{\mathbb{N}}}s_1} & \cdots & Q_{n_{l_{\mathbb{N}}}s_{l_{\mathbb{N}}}}\end{bmatrix}\end{aligned} \tag{10}$$

To summarize, given $\Delta\alpha_c$ we can update α_i for $i\in S$ and bias b using the Eqs. (7) and (8), and update $\Delta g(x_i)$ for $i\in\{E\ \cup R\}$ using Eq. (9).

3.3 Accounting: Largest Increment $\Delta\alpha_c$

Equations (5) and (9) hold only when there are no changes to the S set. Once the newly included sample affects the S set, the updated SVM state cannot be directly computed from these equations. Using KKT conditions (3) and Eq. (7), we can also observe that $\varDelta\alpha_i$, $\forall i \in S$ and $\varDelta\alpha_c$ affect the composition of S. Therefore, in the process of incrementing α_c towards the optimal solution, the SVM parameters are required to be updated in discrete steps with the largest possible $\varDelta\alpha_c$ such that KKT optimality conditions are not violated for any of the existing samples. The following bookkeeping is required to ensure that KKT conditions are not violated in each of the discrete steps:

1. If $g(x_c)$ changes from $g(x_c) < 0$ to $g(x_c) = 0$, the new sample x_c is moved to set S and the algorithm terminates. The proposed $\varDelta\alpha_c^S = \frac{-g_c}{\gamma_c}$.
2. If α_c changes from $\alpha_c < C$ to $\alpha = C$, the new sample x_c is moved to set E and the algorithm terminates. The proposed $\varDelta\alpha_c^E = C - \alpha_c$.
3. $\forall i \in$ set S with $0 < \alpha_i < C$:
   - If $\beta_i < 0$ and α_i changes to $\alpha_i = 0$, sample x_i is moved from S to R set and the proposed $\varDelta\alpha_c^{SR} = \underset{i \in S}{min} \frac{-\alpha_i}{\beta_i}$.
   - If $\beta_i > 0$ and α_i changes to $\alpha_i = C$, sample x_i is moved from S to E set and the proposed $\varDelta\alpha_c^{SE} = \underset{i \in S}{min} \frac{C-\alpha_i}{\beta_i}$.
4. $\forall i \in$ set E, if $\gamma_i > 0$ and $g(x_i)$ changes from $g(x_i) < 0$ to $g(x_i) = 0$, sample x_i is moved from E to S set and the proposed $\varDelta\alpha_c^{LE} = \underset{i \in E}{min} \frac{-g_i}{\gamma_i}$.
5. $\forall i \in$ set R, if $\gamma_i < 0$ and $g(x_i)$ changes from $g(x_i) > 0$ to $g(x_i) = 0$, sample x_i is moved from R to S set and the proposed $\varDelta\alpha_c^{LR} = \underset{i \in R}{min} \frac{-g_i}{\gamma_i}$.

The above five cases are used to determine the allowed values of $\varDelta\alpha_c$, among which the minimum value is chosen to ensure that the KKT conditions hold for all samples in the training set.

$$\varDelta\alpha_c = min(\varDelta\alpha_c^S,\ \varDelta\alpha_c^E,\ \varDelta\alpha_c^{SR},\ \varDelta\alpha_c^{SE},\ \varDelta\alpha_c^{LE},\ \varDelta\alpha_c^{LR}) \tag{11}$$

Finally, the algorithm terminates on either $\alpha_c = C$ or $g_c = 0$.

3.4 Incremental Update of R Matrix

R matrix must be updated whenever the S set changes but it is impractical to invert the matrix $\mathbb{Q}$ every time this happens. We apply the Sherman-Morrison-Woodbury formula for block matrix inversion [7] that recursively updates R matrix to avoid the explicit computation of matrix inverse [16]. To add a sample x_c to the S set, R is expanded as:

$$R = \begin{bmatrix} & & 0 \\ & R & \vdots \\ & & 0 \\ 0 & \cdots 0 & 0 \end{bmatrix} + \frac{1}{\gamma_c} \begin{bmatrix} \beta_0 \\ \beta_{s_1} \\ \vdots \\ \beta_{s_{l_\mathbb{S}}} \\ 1 \end{bmatrix} \begin{bmatrix} \beta_0 \ \beta_{s_1} \ \ldots \ \beta_{s_{l_\mathbb{S}}} \ 1 \end{bmatrix} \tag{12}$$

To remove the k^{th} support vector x_{s_k} from set S, R matrix can be contracted as follows:

$$R_{ij} = R_{ij} - \frac{1}{R_{kk}} R_{ik} R_{kj} \forall i,j \in S \cup \{0\}; i,j \neq k \tag{13}$$

3.5 Convergence and Breaking Immediate Cycling

One of the key characteristics of the incremental algorithm proposed in [16] is that it converges in a finite number of steps only if in each of these steps a non-zero update $\Delta\alpha_c$ is found. This brings us to 'zero-progress' scenarios where $\Delta\alpha_c = 0$ is encountered. We next discuss in detail two such scenarios, namely *empty support vector set* and *immediate cycling*.

Empty Support Vector Set: Equation (5) requires that the support vector set S to be non-empty. Otherwise the matrix $\mathbb{Q}$, which is to be inverted to get R, is an empty matrix. To handle this special case, we first try to find a sample point $x_i :\ i \in \{E\ \cup R\}$ with $g(x_i) = 0$. If this sample exists, we can move it to the S set which does not violate the KKT conditions and continue with the algorithm.

Otherwise, the expression for the margin update (formula (9)) reduces to:

$$\Delta g(x_i) = y_i \Delta b,\ \forall i \in \{E\ \cup R\} \cup \{c\} \tag{14}$$

Using this we can find the maximum change in Δb such that $g(x_i)$ for one of the samples becomes 0 and hence can be moved to the S set. With the S set being non-empty, we can continue with the algorithm.

Immediate Cycling and Cycle Breaking: There is another scenario in which $\Delta\alpha_c = 0$ and the algorithm fails to making progress. In this scenario, a sample migrating from one set to another is immediately removed from that set in the very next iteration without making any progress towards convergence. For example, suppose a sample x_i moves from R to S set in an iteration, $\alpha_i = 0$ because x_i was in the R set. In the next iteration, $\Delta\alpha_i = 0$ if $\beta_i < 0$, hence x_i becomes a potential candidate to be selected as the sample with the minimum $\Delta\alpha_c$. This results in a transition from S set back to R set and because the algorithm does not make any progress in the previous iteration it falls into this infinite loop of transitioning x_i back and forth between S and R sets. This is called *immediate cycling*. Laskov et al. [12] showed that it is theoretically impossible to encounter an immediate cycle if the symmetric augmented kernel matrix $\mathbb{Q}$ is positive semi-definite. But in the real world setup $\mathbb{Q}$ matrix might not always be positive semi-definite so that the immediate cycle is inevitable. One simple solution to this issue is to fall back to retrain the model from scratch which is usually not acceptable in streaming applications. Hence there is a need for solutions that can handle such scenarios incrementally without retraining from scratch.

One of the reasons for the $\mathbb{Q}$ matrix becoming non semi-positive is the existence of duplicate sample points in the training set. Hence, as shown in Fig. 1, in the *De-duplication* stage, the framework ensures that a newly arrived sample is ignored if its duplicate already exists in the training set.

Sample deduplication only fixes one kind of immediate cycling. We use a heuristic when other immediate cycling occurs. The heuristic involves two steps:

(1) *cycle detection* and (2) *cycle break*. The cycle detection step identifies the zero-progress scenario and the corresponding sample responsible for it. In particular, going from $(t-1)^{th}$ to t^{th} iteration, the cycle detection conditions are as follows:

$$\begin{aligned} \beta_i > 0 \; and \; \alpha_i = C; \; i^{t-1} \in E \wedge \; i^t \in S \\ \beta_i < 0 \; and \; \alpha_i = 0; \; i^{t-1} \in R \; \wedge \; i^t \in S \end{aligned} \tag{15}$$

After the sample x_i, which is responsible for the cycle, has been detected, the heuristic artificially adds a small positive perturbation in the cycle break step, using a user defined *sensitivity* parameter, ensuring $\Delta\alpha_c > 0$. This is achieved by modifying the coefficient α_i of x_i using the following rule:

$$\begin{aligned} \alpha_i = C \cdot sensitivity; \; i^{t-1} \in R \wedge \; i^t \in S \\ \alpha_i = C \cdot (1 - sensitivity); \; i^{t-1} \in E \; \wedge \; i^t \in S \\ where \; sensitivity \in [0,1] \end{aligned} \tag{16}$$

Note that *sensitivity* controls the progress rate of convergence. Choosing very small values for sensitivity might require a lot of iterations for a new sample x_c, that encountered a cycle, to reach the optimal solution. Choosing a large value ($\approx$1) might deteriorate the accuracy of the SVM solution. In all our experiments we have chosen a sensitivity of 0.01.

3.6 Algorithm Summary and Runtime Analysis

We summarize the incremental SVM algorithms in Algorithm 1. As revealed in the pseudo-code as well as in the flow chart of Fig. 1, the incremental algorithm can be viewed as a multi-stage pipeline, involving arithmetic (matrix-vector and matrix-matrix multiplications) and memory operations, which is iterated until convergence. These iterations also involve sample migration between S, R and E sets before the algorithm converges to an optimal solution. We list the stages as follows, assuming that an existing SVM model is available.

- **Deduplication:** This is the preprocessing stage to check for duplication. Only unseen samples to the model is allowed to enter the pipeline since duplicated ones do not carry additional information to the model.
- **Kernel Matrix Augmentation:** A new row and a new column corresponding to the new sample x_c are computed using $Q_{ij} = y_i y_j K(x_i, x_j)$ to augment the kernel matrix Q.
- **Gradient/Margin Calculation:** The margin or gradient g_i for each of the sample x_i is calculated. This stage is computationally intensive involving a matrix-vector multiplication ($GEMV$) between the kernel matrix Q and the sample coefficient vector α.
- γ **Calculation:** This stage involves a matrix-vector multiplication of matrix QNS and vector β (Eq. 10). Note that QNS is a data structure that depends on the S set which dynamically changes between iterations. Because S set

Algorithm 1. Incremental SVM algorithm

```
1: Read sample $x_c$, compute gradient $g_c$, $\alpha_c \leftarrow 0$.
2: if $CheckDuplicate(x_c)$ = true or $g_c > 0$ then
3:     return
4: end if
5: while $g_c > 0$ and $\alpha_c < C$ do
6:     if Margin $g$ not in $Cache$ then
7:         $g \leftarrow CalculateMargin()$
8:     end if
9:     $\beta \leftarrow CalculateBeta()$
10:    $\gamma \leftarrow CalculateGamma()$
11:    $\Delta\alpha_c \leftarrow FindMinProgress()$
12:    if $\Delta\alpha_c = 0$ then
13:        $\Delta\alpha_c \leftarrow BreakCycle(sensitivity)$
14:    end if
15:    /* Update SVM solution state*/
16:    $\alpha_c \leftarrow \alpha_c + \Delta\alpha_c$
17:    $\alpha_s \leftarrow \beta\Delta\alpha_c, \forall s \in S$
18:    $g_n \leftarrow \gamma\Delta\alpha_c, \forall n \in \{R \cup E\}$
19:
20:    $MoveVector()$ {See Sect. 3.3}
21:    Incrementally update R matrix. {see Sect. 3.4}
22: end while
```

has substantial temporal and spacial locality between consecutive iterations, efficient storage and caching of QNS can avoid expensive recomputation and irregular memory accesses.

- **Minimum Progress and Cycle Check:** Using the accounting rules of Sect. 3.3, all possible $\Delta\alpha_c$ values are calculated from five different cases and the minimum is used as the maximum progress towards optimal solution without violating KKT conditions for other samples. If zero-progress scenarios is encountered i.e. $\Delta\alpha_c = 0$: empty support vector set is handled using Eq. (14) and immediate cycling using *CycleBreak* heuristic (see Sect. 3.5).
- **Update SVM State:** Using (8), (7) and (9) we update the coefficient α_i, bias b and gradient g_i for every sample $x_i \in T$.
- **Sample Movement:** After updating the coefficient α_i, some samples might need to be migrated to different sample sets due to the change of memberships. Using the migration rules described in Sect. 3.3 and the case that was responsible for the minimum $\Delta\alpha_c$, a particular sample is migrated to an appropriate destination set. If in this process S set changes (grows or shrinks), then matrix R is updated using the incremental update trick described in Sect. 3.4.
- **KKT Condition Check:** Finally, the KKT optimality conditions for x_c are checked to terminate the iteration if $\alpha_c = C$ or $g_c = 0$.

3.7 Decremental Algorithm

The decremental algorithm is very similar to the incremental version with minor changes. When a sample x_c is removed from the training set T, the algorithm gradually decreases the value of the coefficient α_c to zero, while ensuring that all other samples satisfy the KKT conditions. In other words, it finds the maximum decrement in the value of α_c instead of maximum increment (as in case of incremental algorithm) following the rules described in Sect. 3.3. Based on the previously discussed incremental version, the decremental algorithm makes the following specific changes:

– If sample $x_c \in R$ set, then remove it from the training set without making any change to the SVM solution.
– There is no case 1 in Sect. 3.3 as the removed sample x_c will never be moved to S set.
– Case 2 is modified to: if α_c changes from $\alpha_c > 0$ to $\alpha_c = 0$, remove the sample x_c and the algorithm terminates.

4 Optimization

In this section, we describe how we optimize the incremental and decremental SVM algorithms described in Sect. 3. The codebase we used to implement the incremental SVM algorithms is originally from a well optimized high-performance GPU-based batch SVM implementation [3], which was also successfully adapted to run efficiently on the previous version of Intel Xeon Phi products [25].

4.1 Caching Dynamic Data Structures

Caching S Set Related Buffer: As mentioned previously, S, R and E sets dynamically change due to sample migrations as the algorithm progresses towards the optimal solution. Because most of the data structures are either dependent on the cardinality of T ($|T| = m$) or S set, we focus on dynamic data structures related to $|S|$. We make two key observations about S: (1) $|S| << m$ (from bounds on error expectation for SVMs [23]), and (2) between two consecutive iterations, S set either remains the same or, grows or shrinks by one sample.

The dynamically changing buffers dependent on S set are β, R, QSc, $\mathbb{Q}$ and QNS. Since $|S| << m$, the buffer with the most significant memory footprint among all is QNS. QNS is required in the gamma calculation stage and with a changing S set this buffer also dynamically changes every iteration. Recomputing the entire QNS matrix every time S set changes is very computationally intensive. On the other hand, copying all the corresponding elements from the cached kernel matrix Q to create QNS will cause a lot of cache misses because of the large memory footprint of both Q and QNS. Therefore, we employ an efficient caching mechanism for QNS and dynamically grow and shrink it as the algorithm progresses. To track the dynamics of the buffer between iterations we maintain two copies of S set indices: S^{t-1} and S^t, corresponding to index set for support vectors in iteration t-1 and t respectively. Hence, if there are no changes to the S set from the previous iteration, we use QNS from previous iteration. Otherwise, we compute the difference between S^{t-1} and S^t set: $\delta_1 = S^t - S^{t-1}$ (grow), $\delta_2 = S^{t-1} - S^t$ (shrink). If δ_1 is not empty, we grow QNS by copying only the corresponding elements from the row in Q matrix indexed by the newly migrated support vector. If δ_2 is not empty, we erase the corresponding row in QNS indexed by the sample removed from support vector set from previous iteration. It is worth noting that all buffers including Q matrix are stored in row-major fashion except for QNS which is stored in column major fashion. This is to make sure that both 'grow' and 'shrink' operations in QNS can be indexed using support vector indices.

Caching Gradient Vector: The gradient information of samples is used in the calculation of minimum progress $\Delta\alpha_c$ as described in the accounting rules of Sect. 3.3 and hence is required to be updated every single iteration. Note that gradient calculation is a computationally expensive operation involving a matrix-vector multiplication between huge kernel matrix Q of size $m \times m$ and sample coefficient vector α. Hence whenever possible gradient information is cached from previous iterations and is reused to avoid this expensive recomputations. Using formula (9) we update the gradient for each sample and reuse it in the next iteration. Note that because the gradients are cached as SVM state, they can be reused across samples as well i.e. we also avoid the gradient recomputation whenever a new sample is inserted.

4.2 Memory Access Pattern

In the incremental SVM training process, a lot of operations (e.g. $GEMV$) are memory-bound. Therefore, it makes sense to place data to MCDRAM which has larger memory bandwidth when running it on the MIC processors. However, since the capacity of MCDRAM is limited (typically 16 GB), we cannot fit the entire working set in. Instead, we explicitly allocated the frequently retrieved data (e.g. QNS matrix, R matrix) to be in MCDRAM using the memkind library. As described above, QNS is a matrix of $m \times |S|$, and R is a $|S| \times |S|$ square matrix. The fact that $|S| << m$ makes it possible to fit QNS and R in MCDRAM.

As an $m \times |S|$ matrix, QNS is maintained in column-major fashion to facilitate the memory access of an entire column corresponding to the kernel values of a specific support vector. And both QNS and R are aligned to 64 bytes so that the vectorization can easily take the entire cache line in for achieving the high performance.

4.3 Parallelization and Vectorization-Friendly Workflow

The workflow of incremental SVM contains multiple stages with various branches. For example, in the bookkeeping process of sample movement, there are five different conditions and within each condition the samples in the same set are handled differently according to their own situations. This characteristics largely prevents the thread-level parallelization and vectorization within a loop.

To facilitate thread-level parallelization, we carefully define stages of the workflow (Fig. 1), so that within each stage the thread-level parallelization can be conducted via OpenMP easily. The philosophy is to make the parallel granularity as large as possible to reduce the OpenMP thread launching overhead while making sure all available cores are utilized. For enabling the vectorization within a loop, we simplify the control flow logic by reducing the number of branches.

5 Evaluation

5.1 Experimental Setup

We tested our implementation of incremental SVM on both the MIC processors and Intel Xeon processors (referred to hereinafter as processors). The configuration of the MIC processors we used was described in Sect. 2. In the experiments we launched 67 threads, one per core, leaving the last core for OS usage. Regarding the processors, we used Intel Xeon E5-2699 v4 (codenamed Broadwell) with 22 cores running at 2.2 GHz. In the experiments we launched 22 threads, one per core.

Table 1 summarizes the datasets we used for evaluation. They are all real datasets in various domains. We z-scored the datasets to bring values of every dimension to the same scale. `covtype` is originally with multiple classes, we converted it into binary classification for our usage.

Table 1. Summary of datasets

Dataset	#samples	#dimensions
`covtype` [5]	50,000	54
`cod-rna` [22]	49,466	7
`ijcnn` [18]	49,990	22
`susy` [1]	60,000	18

In addition to our incremental algorithm, for comparison purpose we also ran another incremental SVM algorithm named warm-start SMO which is used in [26]. The warm-start SMO shares the same high-performance SVM training codebase as our incremental SVM code. Therefore, it is a decent baseline which is supposed to largely outperform the off-the-shelf SVM packages, let alone most of them do not have incremental training component built in. In the warm-start SMO, when a new sample is added, the training takes place from the state of the current model until it gets converged again. Both our incremental algorithm and the warm-start SMO are based on a batch trained model with $C = 1$ and Gaussian kernel with $\gamma = 1/\#dimensions$.

5.2 Overall Performance

We first thoroughly test out the performance of the incremental SVM algorithms by starting from very small models that contain only tens of samples, and incrementally training all the way to the entire dataset, except for 500 samples kept for accuracy testing. Table 2 shows the running time of both our incremental algorithm and the warm-start SMO on the MIC processors. From the table we can see that our incremental algorithm runs $1.1 - 2.1\times$ faster than warm-start SMO.

Table 2. Performance of incremental SVM and warm-start SMO on MIC processors

Dataset	Incremental (s)	Warm-start (s)
`covtype`	238.5	397.6
`cod-rna`	240.2	454.9
`ijcnn`	192.5	409.7
`susy`	3069.5	3344.4

We compared the models we obtained using our incremental algorithm, the warm-start SMO and batch training to verify the correctness of the methods in Table 3. We counted the number of support vectors (the samples in S set plus E set) as well as applied the models to the left-out 500 samples of each dataset. The results show that the models obtained from our incremental SVM is close to the batch training and warm-start SMO training, which verifies the correctness of our implementation.

Table 3. SVM model comparison

Dataset	Incremental		Warm-start		Batch	
	#SV	Accuracy	#SV	Accuracy	#SV	Accuracy
`covtype`	7593	77.8%	7552	77.0%	7704	77.2%
`cod-rna`	7807	94.2%	7764	95.6%	7805	95.8%
`ijcnn`	4849	96.8%	4772	98.0%	4843	98.0%
`susy`	27742	80.6%	27825	80.4%	27881	80.6%

In practice, the incremental training may take place on top of a model trained from a large number of samples in batch. For example, in a real-time data analysis application, one may want to tweak a well-trained model using the incoming data stream. For this scenario, we evaluated the running time of adding 100 samples to a model trained via tens of thousands of samples on both the MIC processors and the processors in Table 4. The results show that running on the MIC processors outperforms the processors by $1.1 - 1.3\times$, which is limited by the unavoidable sequential code spread in the workflow.

Table 4. Streaming in 100 samples on different processors

Dataset	MIC processors (s)	Processors (s)
`covtype`	4.61	5.59
`cod-rna`	0.84	1.04
`ijcnn`	0.65	0.73
`susy`	12.75	15.94

We also investigated the time it takes to incorporate each new sample using both our incremental algorithm and the warm-start SMO on the MIC processors. Similar to what we have done in Table 4, we streamed in 100 samples of each dataset and record the time it took to process them one by one. Figure 2 depicts the results sorted by processing times, from which we see that in most of the cases (90% of the samples) our incremental algorithm processes the samples faster. Especially, for the samples that do not affect the distinguishing hyperplane, the processing rate using our incremental algorithms is more than 10× faster than the warm-start SMO (e.g. 0.3 ms/sample vs. 4 ms/sample). Even for the samples that requires longer training (hundreds of milliseconds), the incremental training is still much faster than batch which typically takes seconds to tens of seconds.

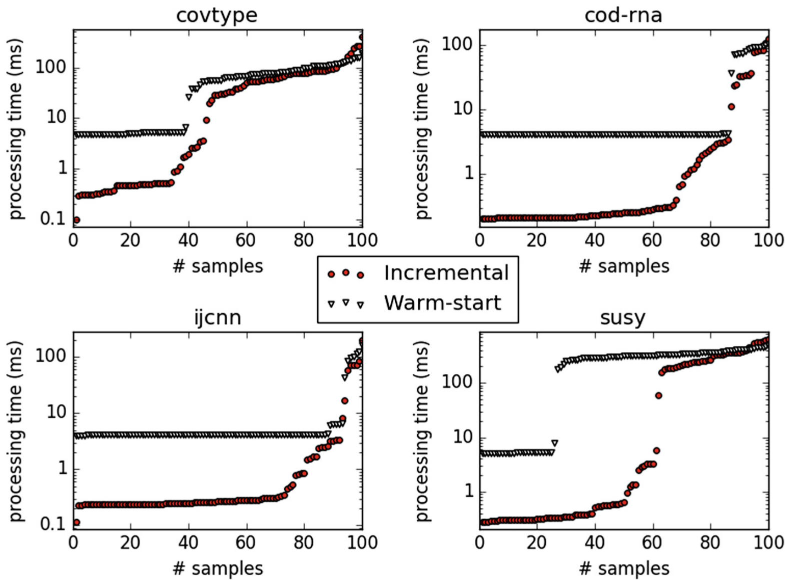

Fig. 2. The time duration of processing single samples in logarithmic scale.

5.3 Performance Gain of Optimization

This subsection shows the performance gain of our proposed optimizations to the incremental SVM algorithm. We summarized the optimization speedup in Table 5 by streaming in 1000 samples to each dataset. Overall, our optimization speed up the algorithm by 4.2−4.8×. Table 5 also listed the number of immediate cycles (Sect. 3.5) encountered during the incremental training. We observed a considerable amount of immediate cycles (in average 1 cycle per 27 samples across all datasets), which suggests that our cycle breaking heuristic is critical in achieving high-performance incremental SVM training.

Table 5. Speedup of optimization and the immediate cycles broken whiling training

Dataset	Caching dynamic data	Memory and vectorization optimization	#cycles
`covtype`	2.8×	1.5×	28
`cod-rna`	3.2×	1.5×	6
`ijcnn`	3.3×	1.4×	7
`susy`	3.0×	1.5×	108

5.4 Leave-One-Out Cross-Validation

Cross-validation (CV) [10] is a popular method to assess the generalization ability of a machine learning model by dividing the dataset into disjoint training and validation sets for training in rotation. Leave-one-out cross-validation (LOOCV) which maintains a one-sample validation set is useful especially when the dataset size is small. In batch processing of SVM, LOOCV can be expensive as retraining of the entire dataset but one sample is required for each sample. LOOCV can be implemented much more efficiently using the decremental variation of our incremental SVM algorithm as follows:

1. Learn the SVM parameter for the entire dataset T in batch;
2. For each sample $x_i \in T$, using the decremental algorithm described in Sect. 3.7, remove x_i to learn the model of $T - \{x_i\}$ to apply to x_i;
3. Summarize the overall classification accuracy on all samples.

We randomly chose 100 samples from each dataset shown in Table 1 to simulate small datasets and ran LOOCV using both batch training and decremental training. From Table 6, we can observe that the decremental algorithm achieves substantial performance benefit over the batch training on the MIC processors. Furthermore, our decremental algorithm on the MIC processors outperforms the processors by $1.1 - 2.1\times$.

Table 6. LOOCV speedup using decremental training

Dataset	Incremental on MIC processors vs. batch on MIC processors	Incremental on MIC processors vs. incremental on processors
`covtype`	121.3×	1.6×
`cod-rna`	11.7×	1.1×
`ijcnn`	108.0×	2.2×
`susy`	15.4×	2.1×

6 Related Work

Incremental training is a popular technique for many machine learning classifiers. For algorithms built by techniques like stochastic gradient descent etc., it is quite simple to train incrementally. For example, incremental learning of neural networks [17] already exist.

Support vector machine training, on the other hand, relies on the convexity of the data space. Therefore, the batch training algorithms are much more efficient than techniques like gradient descent. Techniques such as Sequential Minimal Optimization (SMO), decomposition based SVM^{light} [11] etc. are only slightly worse than linear time in practice [15]. Efficient implementations of SVM training exist in many software packages optimized for different hardware platforms including GPU [3] and the first generation of Intel Xeon Phi products [25].

A good incremental SVM training algorithm in practice must both perform precisely to produce similar results, if not identical, to the batch training, and run in high-performance. Incremental SVM training algorithms such as [2,21] do not solve the problem to full optimality. They are only approximate approaches by applying updates to the set of support vectors instead of the full dataset. Errors can accumulate if these algorithms run for a large number of samples. Incremental SVM training algorithms based on SMO (called warm-start SMO which trains from the previously converged states) [6,8,9,27] have been proposed earlier, but perform slowly especially for the non-support vector new samples.

Our work uses the incremental algorithm presented in [16] which can be used to update models when adding or removing samples. This algorithm was not practical to use due to two reasons - (1) there were no good methods to break out of cycles (Sect. 3.5) and (2) naive implementation of this algorithm would not be faster than warm-start SMO. [12] characterizes the cycling phenomenon but does not propose a feasible solution beyond doing batch retraining. Incremental SVM training is particularly challenging to implement on many-core architectures because of its highly irregular workflow (Sect. 3) and branch-heavy code. As a consequence, there is no existing implementation optimized for highly parallel platforms. We largely relieve these limitations to present a usable and efficient implementation of the algorithm optimized for the MIC processors in this paper. Given that other work [13,14] has extended [16] to support vector regression, our improvements should apply to those techniques as well.

7 Conclusions and Future Work

This paper discussed how to construct an efficient implementation of the incremental SVM training algorithm that runs well on many-core architectures such as Intel Xeon Phi processors. We started from a known algorithm [16] and fixed several issues (immediate cycle, data recompuation, irregular memory access pattern, lack of parallelization and vectorization) to improve the performance. We have shown that our implementation is up to 2.1× faster than warm-start SMO, another high-performance incremental SVM algorithm, on average. Our algorithm is better than warm-start SMO for 90% of samples. The code is planned to be released as open source and we hope it will benefit SVM training in real-time and other streaming-oriented applications in various domains.

Our further work focuses on scaling up the current incremental SVM implementation for very large datasets. The major limitation of current implementation is that the support vectors are required to be maintained in the memory for the entire learning process. Two largest data structures in play are $\mathbb{Q}$ and QNS, with memory complexities $O(|S|^2)$ and $O(m * |S|)$ respectively, where $|S|$ is the size of the support vector set, and m is the number of all training samples. From [24], we know that for an SVM solution to generalize well on test set, $|S| << m$ holds. Hence the size of $\mathbb{Q}$ matrix in all practical scenarios should not become the bottleneck. However QNS can be too large to fit in the main memory. Therefore, in order to scale the algorithm to a very large m ($m \rightarrow \infty$), we would need an intelligent way to handle the QNS matrix efficiently. For example, we may extend the algorithm implementation to more than one MIC processor.

References

1. Baldi, P., Sadowski, P., Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning. Nat. commun. **5** (2014)
2. Bordes, A., Ertekin, S., Weston, J., Bottou, L.: Fast kernel classifiers with online and active learning. J. Mach. Learn. Res. **6**, 1579–1619 (2005)
3. Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: Proceedings of the 25th International Conference on Machine Learning, pp. 104–111 (2008)
4. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines. ACM Trans. Intell. Syst. Technol. **2**(3), 1–27 (2011)
5. Collobert, R., Bengio, S., Bengio, Y.: A parallel mixture of SVMs for very large scale problems. Neural Comput. **14**(5), 1105–1114 (2002)
6. Fliege, J., Heseler, A.: Constructing Approximations to the Efficient Set of Convex Quadratic Multiobjective Problems. Citeseer, Princeton (2002)
7. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
8. Gondzio, J.: Warm start of the primal-dual method applied in the cutting-plane scheme. Math. Program. **83**(1–3), 125–143 (1998)
9. Gondzio, J., Grothey, A.: Reoptimization with the primal-dual interior point method. SIAM J. Optim. **13**(3), 842–864 (2002)

10. Hastie, T.J., Tibshirani, R.J., Friedman, J.H.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. Springer, New York (2009)
11. Joachims, T.: Svmlight: support vector machine. SVM-Light Support Vector Mach. **19**(4) (1999). University of Dortmund, http://svmlight.joachims.org/
12. Laskov, P., Gehl, C., Krüger, S., Müller, K.-R.: Incremental support vector learning: analysis, implementation and applications. J. Mach. Learn. Res. **7**(Sep), 1909–1936 (2006)
13. Ma, J., Theiler, J., Perkins, S.: Accurate on-line support vector regression. Neural Comput. **15**(11), 2683–2703 (2003)
14. Martìn Muñoz, M.: On-line support vector machines for function approximation. Technical report, Universitat Politecnica de Catalunya, Departament de Llengatges i Sistemes Informatics (2002)
15. Platt, J.: Sequential minimal optimization: a fast algorithm for training support vector machines. Technical report MSR-TR-98-14, Microsoft Research, April 1998
16. Poggio, T., Cauwenberghs, G.: Incremental and decremental support vector machine learning. Adv. Neural Inf. Process. Syst. **13**, 409 (2001)
17. Polikar, R., Byorick, J., Krause, S., Marino, A., Moreton, M.: Learn++: a classifier independent incremental learning algorithm for supervised neural networks. In: Proceedings of the 2002 International Joint Conference on Neural Networks, IJCNN 2002, vol. 2, pp. 1742–1747 (2002)
18. Prokhorov, D.: IJCNN 2001 neural network competition. Slide Present. IJCNN **1**, 97 (2001)
19. Smola, A.J., Schölkopf, B.: A tutorial on support vector regression. Stat. Comput. **14**(3), 199–222 (2004)
20. Sodani, A.: Knights landing (KNL): 2nd generation Intel® Xeon Phi$^{\text{TM}}$ processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS), pp. 1–24. IEEE (2015)
21. Syed, N.A., Huan, S., Kah, L., Sung, K.: Incremental learning with support vector machines (1999)
22. Uzilov, A.V., Keegan, J.M., Mathews, D.H.: Detection of non-coding RNAs on the basis of predicted secondary structure formation free energy change. BMC Bioinf. **7**(1), 1 (2006)
23. Vapnik, V., Chapelle, O.: Bounds on error expectation for support vector machines. Neural Comput. **12**(9), 2013–2036 (2000)
24. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer-Verlag New York Inc., New York (1995)
25. Wang, Y., Anderson, M.J., Cohen, J.D., Heinecke, A., Li, K., Satish, N., Sundaram, N., Turk-Browne, N.B., Willke, T.L.: Full correlation matrix analsis of fMRI data on Intel® Xeon Phi$^{\text{TM}}$ coprocessors. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, November 2015
26. Wang, Y., Keller, B., Capotă, M., Anderson, M.J., Sundaram, N., Cohen, J.D., Li, K., Turk-Browne, N.B., Willke, T.L.: Real-time full correlation matrix analysis of fmri data. In: 2016 IEEE International Conference on Big Data (Big Data). IEEE (2016)
27. Yildirim, E.A., Wright, S.J.: Warm-start strategies in interior-point methods for linear programming. SIAM J. Optim. **12**(3), 782–810 (2002)