MIC-SVM: Designing A Highly Efficient Support Vector Machine For Advanced Modern Multi-Core and Many-Core Architectures

Yang You[*||], Shuaiwen Leon Song[†], Haohuan Fu[*], Andres Marquez[†], Maryam Mehri Dehnavi[‡], Kevin Barker[†], Kirk W. Cameron[§], Amanda Peters Randles[¶], Guangwen Yang[*||]
[*]Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth Science, Tsinghua University, Beijing, China
[||]Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing, China
[†]Pacific Northwest National Lab, Richland, WA, USA
[‡] Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA
[§]Virignia Tech, Blacksburg, VA, USA
[¶]Lawrence Livermore National Lab, Livermore, CA, USA

***Abstract*—Support Vector Machine (SVM) are widely used in data-mining and big data applications. In recent years, SVM has been used in High Performance Computing (HPC) for power/performance prediction, auto-tuning, and runtime scheduling. However, to avoid runtime training overhead HPC researchers can only afford to apply offline model training. This often leads to loosing prediction accuracy because no runtime information is available. Advanced multi- and many-core architectures offer massive parallelism with complex memory hierarchies which makes runtime training possible, but efficiently parallelizing the SVM algorithm on these architectures is challenging.**

To address the challenges above, we have designed and implemented MIC-SVM, a highly efficient parallel SVM for x86 based multi-core and many-core architectures, such as the Intel Ivy Bridge CPUs and Intel Xeon Phi co-processor (MIC). We propose various novel analysis methods and optimization techniques to fully utilize the multilevel parallelism provided by these architectures. The proposed techniques serve as general optimization methods for other machine learning tools.

MIC-SVM achieves 4.4-84x and 18-47x speedup against the popular LIBSVM, on MIC and Ivy Bridge CPUs respectively, for several real-world data-mining datasets. Compared to GPUSVM, ran on a modern NVIDIA k20x GPU, the performance of our MIC-SVM is competitive. We also conduct a cross-platform performance comparison, focusing on Ivy Bridge CPUs, MIC, and GPUs. We also provide insights on how to select the most suitable architecture for individual algorithms and input data patterns.

***Index Terms*—Support Vector Machine; Multi- & Many-Core architectures; Parallelization; Optimization Techniques; Performance Analysis**

I. Introduction

Support Vector Machine (SVM) [1] shown in Figure 1 is a classification method that has been widely used in text categorization [2], financial analysis [3], bioinformatics [4] and many other fields. SVM has been used in the advanced databases like Oracle (10g, 11g, 12c) [5] as a major data mining technique as companies increasingly rely on their analytic capabilities. However, time-consuming training processes greatly limit the efficiency of SVM. This concern is likely to increase with the growing volume of data in big data computations. This issue is also exacerbated by the limits in increasing clock frequency and the rise of multi- and many-core architectures, whose massive parallelism and complex memory hierarchies form a barrier to efficient parallel SVM design. While programmers in the past could depend on the ready-made performance improvement that comes from a faster clock frequency, now they are faced with the challenge of scaling performance over tens or even hundreds of cores within a single chip [6]. One of the most representative architectures is Intel Xeon Phi (MIC) [7].

SVMs are recently being used in HPC for performance and power prediction during runtime at the system and compiler level for design space exploration [8] [9]. Common approaches in current work train the SVM model offline and then apply the static model for online prediction to avoid significant performance overheads. However, this dramatically reduces the flexibility of the runtime system; it may also increase the chance of model misprediction due to the lack of runtime behavior information. In order to use SVM at runtime for dynamic modeling and scheduling in HPC, we need to accelerate its training phase on advanced multi- and many-core architectures.

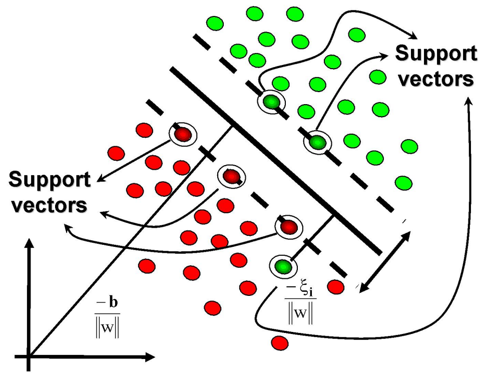

Fig. 1. The figure shows Support Vector Machines, whose decision boundary is decided by Support Vectors (SVs).The left-lower dots represent the -1 class and the right-upper dots represent +1 class. These two classes are classified by the the decision boundary (the dashed lines in this figure). We can observe that the classification result is decided by the six Support Vectors.

Previous work either focused on designing SVM tools for CPUs with relatively few cores and simple memory hierarchies, such as the serial LIBSVM [10] or [11], or creating techniques to accelerate SVM on GPUs such as GPUSVM [12]. However, there has been no efficient SVM tool designed for advanced x86 based multi- and many-core architectures such as Ivy Bridge CPUs and Intel Xeon Phi (MIC), even though they have already gained popularity on recent TOP500 list [13]. Also there are several design deficiencies (i.e. no adaptive runtime support for efficient memory management and data parallelism) within the existing tools such as LIBSVM and GPUSVM, that will ultimately limit the performance improvements for future architectures.

In this paper, we present MIC-SVM, a highly efficient parallel support vector machine designed for x86 based multi-core and many-core architectures such as Ivy Bridge CPUs and Intel Knight Corner(KNC) MIC [14]. We design and implement an open-source SVM Library that is not only highly efficient but can also be easily adopted to the existing runtime systems or software. We want to create methods and techniques general enough to be applied to optimize similar machine learning models.

The contributions of this work include:

- Designing and implementing MIC-SVM (open-source libraries), a highly efficient parallel support vector machine for x86 based

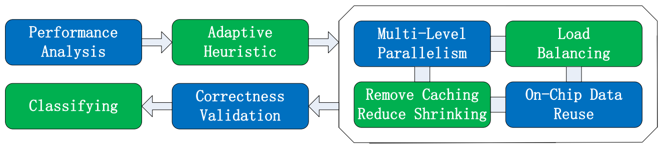

Fig. 2. General flow for parallelizing and optimizing MIC-SVM.

TABLE I
STANDARD KERNEL FUNCTIONS

Linear	$Kernel(X_i, X_j) = X_i\dot{X}_j$
Polynomial	$Kernel(X_i, X_j) = (aX_i\dot{X}_j + r)^d$
Gaussian	$Kernel(X_i, X_j) = -\gamma\|\|X_i - X_j\|\|^2$
Sigmoid	$Kernel(X_i, X_j) = tanh(aX_i\dot{X}_j + r)$

multi- and many-core architectures such as MIC.

- Proposing novel analysis methods and optimization techniques (i.e. adaptive support for input patterns and data parallelism) to fully utilize the multi-level parallelism provided by the studied architectures.
- Exploring and improving the deficiencies of the current SVM tools such as LIBSVM and GPUSVM.
- Providing insights on how to select the most suitable architectures for specific algorithms and input data patterns to achieve best performance.

Our experiments show that our proposed MIC-SVM can achieve 4.4-84x and 18-47x speedups against the widely-used LIBSVM on MIC and Ivy Bridge CPUs respectively, for several real-world data-mining datasets. Even compared with the highly optimized GPUSVM, the performance of our MIC-SVM is still very competitive.

II. BACKGROUND

Support Vector Machines (SVMs) (shown in Fig. 1) consist of two major phases: training and classification. The user obtains the model file through the training process and uses it to make predictions in the classification process. Based on our performance profiling, the majority of SVM execution time is spent on the training phase which makes training the major performance bottleneck in SVM.

A. SVM Training Phase

In this work, we focus on binary-class SVMs since multi-class SVMs are generally implemented as several independent binary-class SVMs. The multi-class SVMs can be easily processed in parallel once the binary-class SVMs are available. The training data of SVMs contains two parts: $X_i, i \in 1, 2, ..., n$ and $y_i, i \in 1, 2, ..., n$. Each X_i is a training sample that contains many features. Each y_i is the sample label that corresponds to one and only one training sample X_i. n denotes the number of the training samples (and the sample labels).

SVM training can be presented as a linear-constraint convex Quadratic Programing (QP) problem (shown in Equation (1) and Equation (2)), where C represents the regularization constant that balances the generality and accuracy, α_i is the Lagrange multiplier, and $Kernel$ denotes the Kernel function. C can be set by users and each α_i is related to a specific training sample. The standard Kernel functions in SVM are shown in Table I.

B. Sequential Minimal Optimization (SMO)

To make a large scale SVM training problem practical, several algorithms have been proposed, including chunking [15], decomposition [16], and caching/shrinking [17]. These algorithms generally decompose a large QP problem into several small QP problems and solve one small problem at each step. Each small QP problem corresponds to one working set and the size of a working set is the number of training samples it contains.

To minimize the size of the working set in the training phase, Sequential Minimal Optimization (SMO) algorithm has been proposed [18] [19] and implemented in several popular SVM tools including LIBSVM [10]. In this work, we focus on providing various optimization strategies to accelerate SMO algorithm (designed for serial processor) in the SVM training phase based on features of modern advanced multi-core and many-core architectures. Eventually, we will conduct cross-platform performance comparison analysis using our proposed MIC-SVM Library (Section IV).

A brief outline of the commonly used SMO (serial code) algorithm is presented in Algorithm 1. All the mathematical derivations of the major parameters in Algorithm 1 are shown in equation (3)-(11). Specifically, f_i ($i \in \{1, 2, ..., n\}$) represents the discrepancy between the calculated objective value and the real objective value (Equation (3)) for each point. The update of all f_i (Equation (6)) at each step requires access to all the training samples, which costs more than 90% of the total time in the entire SMO algorithm. Therefore, accelerating f_i updates is our top priority for optimization. Algorithm parallelization and detailed optimization methodologies will be shown in Section III.

Two popular optimization techniques have been applied to the training phase of previous SVM tools such as LIBSVM: Caching and Shrinking. The Caching technique is proposed to store the computed kernel elements ($Kernel(X_i, X_j)$) in the remaining memory through the least-recent-use (LRU) strategy—to decrease the number of kernel evaluations efficiently by avoiding constantly accessing the training samples. The Shrinking method has been employed to remove the bounded elements such as α_i and C in order to reduce the size of the optimization problem and the amount of calculations. However, when designing a parallel SVM tool for advanced multi- and many-core architectures, these optimization techniques may lead to negative performance. For instance, if the algorithm is memory bound, the caching strategy may not bring any performance benefits for these architectures due to higher memory access cost, memory contention from multi-threading and limited bandwidth. In Section III, we will show how these traditional optimizations may not be suitable for our parallel MIC-SVM tool designed based on the features of modern multi- and many-core architectures.

III. METHODOLOGY FOR MIC-SVM

In this section, we will briefly describe several important design methods and optimization details for our proposed MIC-SVM on parallelizing and accelerating SVM method on Intel Ivy Bridge CPUs and Intel Xeon Phi coprocessor (MIC). Since Ivy Bridge CPUs and MIC share many similarities in architecture, we employ several similar optimization techniques for them. Figure 2 shows the general flow for parallelizing and optimizing our MIC-SVM, which can also be applied to optimize similar machine learning methods. The contents of this section will follow this flow as well. Our MIC-SVM tool can be downloaded at [20].

$$\text{Maximize: } F(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j Kernel(X_i, X_j) \tag{1}$$

$$\text{Subject to: } \sum_{i=1}^{n} \alpha_i y_i = 0 \text{ and } 0 \le \alpha_i \le C, \forall i \in 1, 2, ..., n \tag{2}$$

$$f_i = \sum_{j=1}^{n} \alpha_j y_j Kernel(X_i, X_j) - y_i \tag{3}$$

$$\hat{\alpha}_{low} = \alpha_{low} + \frac{y_{low}(b_{high} - b_{low})}{Kernel(X_{high}, X_{high}) + Kernel(X_{low}, X_{low}) - 2Kernel(X_{high}, X_{low})} \tag{4}$$

$$\hat{\alpha}_{high} = \alpha_{high} + y_{low} y_{high} (\alpha_{low} - \hat{\alpha}_{low}) \tag{5}$$

$$\hat{f}_i = f_i + (\hat{\alpha}_{high} - \alpha_{high}) y_{high} Kernel(X_{high}, X_i) + (\hat{\alpha}_{low} - \alpha_{low}) y_{low} Kernel(X_{low}, X_i) \tag{6}$$

$$I_{high} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = 0\} \cup \{i : y_i < 0, \alpha_i = C\} \tag{7}$$

$$I_{low} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = C\} \cup \{i : y_i < 0, \alpha_i = 0\} \tag{8}$$

$$i_{high} = \text{arg min}\{f_i : i \in I_{high}\}, i_{low} = \text{arg max}\{f_i : i \in I_{low}\} \tag{9}$$

$$b_{high} = \min\{f_i : i \in I_{high}\}, b_{low} = \max\{f_i : i \in I_{low}\} \tag{10}$$

Algorithm 1 The Original SMO Algorithm

```
0: Start
1: Input the training samples $X_i$ and pattern labels $y_i$, $\forall i \in \{1, 2, ..., n\}$.
2: Initializations, $\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1, 2, ..., n\}$.
3: Initializations, $b_{high} = -1$, $i_{high} = \min\{i : y_i = -1\}$, $b_{low} = 1$, $i_{low} = \max\{i : y_i = 1\}$.
4: Update $\alpha_{high}$ and $\alpha_{low}$ according to Equation (4) and (5).
5: If $Kernel(X_{high}, X_i)$ is not in memory, then compute $Kernel(X_{high}, X_i)$ and cache $Kernel(X_{high}, X_i)$ in memory using the LRU strategy.
6: If $Kernel(X_{low}, X_i)$ is not in memory, then compute $Kernel(X_{low}, X_i)$ and cache $Kernel(X_{low}, X_i)$ in memory through LRU strategy.
7: Update $f_i$ according to Equation (6), $\forall i \in \{1, 2, ..., n\}$
8: Compute $i_{high}$, $i_{low}$, $b_{high}$, and $b_{low}$ according to Equation (9) and (10).
9: Update $\alpha_{high}$ and $\alpha_{low}$ according to Equation (4) and (5).
10: If iterations meet certain number, then do shrinking.
11: If $b_{low} > b_{high} + 2 \times tolerance$, then go to Step 5.
12: End
```

A. Analysis of the Algorithm and Architecture

As mentioned in Section II-B, the updates of f_i ($i \in \{1, 2, ..., n\}$) at each step is the most time consuming phase in the SMO algorithm because it needs access to all the training samples. Fortunately, for any pair of $i, j \in \{1, 2, ..., n\}$, the updates of f_i and f_j are independent of each other. Therefore, we can convert the serial form (Equation (6)) to the parallel form (Equation (11)) :

$$\hat{F} = F + (\hat{\alpha}_{high} - \alpha_{high}) y_{high} HighKernel + (\hat{\alpha}_{low} - \alpha_{low}) y_{low} LowKernel \tag{11}$$

where F, $HighKernel$, and $LowKernel$ are vectors.

According to Equation (6)) and (11)), we can easily get $F(i) = f_i$, $HighKernel(i) = Kernel(X_{high}, X_i)$, and $LowKernel(i) = Kernel(X_{low}, X_i)$. Figure 3 illustrates the computation of $HighKernel$ used in Equation (11)). $LowKernel$ can be processed in the same fashion.

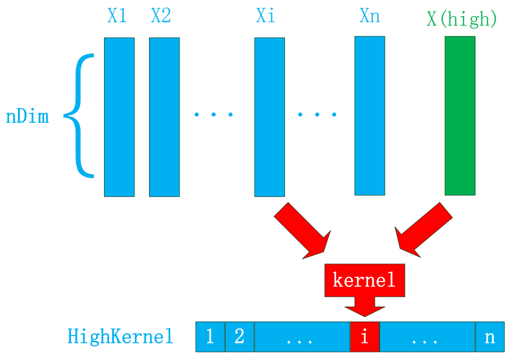

Fig. 3. This figure corresponds to the computation of $HighKernel$ (used in Equation (11)). $HighKernel$ is a vector, and $HighKernel(i) = Kernel(X_{high}, X_i)$. From this figure we can see that the evaluation of $HighKernel$ need to access all the training samples. The computation of $LowKernel$ can be accomplished in the same way.

In order to effectively analyze and optimize the parallel SMO algorithm in MIC-SVM, we need to analyze on the gap between algorithmic features and characteristics of the underlying architectures. Here, we define the Ratio of Computation to Memory Access (RCMA) to describe SMO's algorithmic feature (shown in Equation (12)):

$$RCMA = \frac{number\_of\_comp\_flops}{number\_of\_memory\_access\_bytes} \tag{12}$$

According to Figure (3) and Equation (11), we take the single precision case as an example: the total bytes of memory access of the algorithm can be estimated as $n \times nDim \times 4$ where n is the

TABLE II
RCMB VALUES FOR EVALUATED ARCHITECTURES IN THIS PAPER

Architecture	Ivy Bridge	MIC	Fermi	Kepler
Frequency (GHz)	2.20	1.09	1.50	0.73
Peak Performance Double (Gflops)	422	1010	515	1320
Peak Performance Single (Gflops)	844	2020	1030	3950
Theoretical Memory bandwidth (GB/s)	128	352	192	250
RCMB Single (Gflops/GB)	6.59	5.74	5.36	15.8
RCMB Double (Gflops/GB)	3.30	2.87	2.68	5.28

number of training samples and $nDim$ is the maximal number of features in one training sample. In order to obtain $HighKernel$ and $LowKernel$, we need to conduct $2n$ kernel evaluations (Figure 3). Take the Gaussian kernel for example (shown in Table I), each kernel evaluation requires $nDim$ subtraction operations, $(nDim+1)$ multiplication operations and $(nDim - 1)$ addition operations. In total, the computations of $HighKernel$ and $LowKernel$ require $2 \times n \times 3 \times nDim$ floating point operations. According to equation (12), the RCMA (Ratio of Computation to Memory Access) of SMO in MIC-SVM is about 1.5 for processing single precision Gaussian kernels (theoretical lower bound).

To compare with the algorithmic bound of parallel SMO, we use the Ratio of Peak Computation to Peak Memory Bandwidth (RCMB) to describe the theoretical architectural bound. RCMB can be defined as (13):

$$RCMB = \frac{theoretical\_peak\_performance}{theoretical\_bandwidth} \tag{13}$$

All the RCMB values of the evaluated architectures in this paper are shown in Table II (both single and double precisions). Take the single precision case for example, the RCMA of the SMO algorithm is around 1.5 (discussed previously), which is lower than all the RCMBs of the evaluated architectures. This indicates that the limited memory bandwidth of the evaluated architectures may not match the high processing power required for accelerating the parallel SMO. Therefore, improving data reuse is very necessary to reduce the impact of the bandwidth constraint.

B. Adaptive Heuristic Support for Input Datasets

In MIC-SVM, we provide methodology for more efficiently processing different types of input datasets (sparse and dense), which has not been well addressed in the previous SVM tools such as LIBSVM and GPUSVM. For instance, in order to reduce the memory requirement, LIBSVM applies sparse data format for processing input datasets. However, certain sparse format can limit the parallelism greatly on specific architectures such as MIC. Moreover, the AOS (array of structure) method used in data processing phase of LIBSVM can lead to noncontinuous memory access, which may be very expensive due to high memory access latency and contention.

On the other hand, GPUSVM applies dense format for efficient data processing. However, this may make GPUSVM difficult for handling some sparse datasets. For example, a training phase that can be completed on a single CPU chip by LIBSVM may require several GPU cards. Take one of our test datasets sraa [21] for example, it only

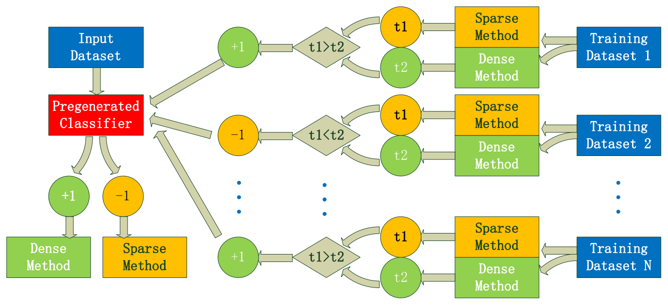

Fig. 4. The figure shows our adaptive support for different types of datasets in our MIC-SVM. The pre-generated classifier is trained through N (N $\geq$ 100) different real-world datasets which cover a variety of patterns.

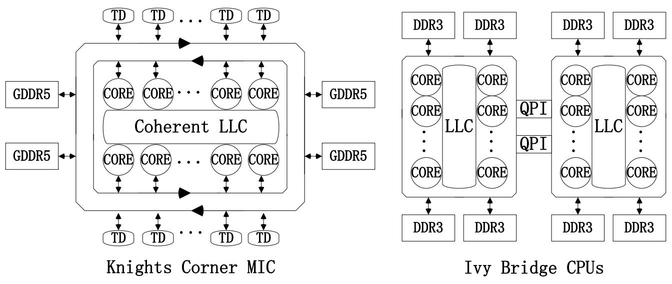

Fig. 5. Abstract Diagrams of two evaluated architectures: MIC and Ivy Bridge CPU.

requires 27 megabytes in sparse format. While in dense format, 11.6 gigabytes are required. This large data size in dense format would make the training process impossible to run on a single GPU card.

To address the issues above, we apply adaptive support for handling different types of datasets in our MIC-SVM. Figure 4 illustrates the process flow. We construct a classification model (or classifier) from the information of training N (N $\geq$100) different real-world datasets, which could cover a variety of data patterns. Then any given dataset will be classified as +1 or -1 by the pre-generated classifier according to its features such as n, $nDim$, and $density$. After that, our MIC-SVM Library will select either dense or sparse method to process the $+1$ and -1 dataset respectively. The classification model and the original training data will also be provided to users. Users have the flexibility to add new training samples to update the classifier. Additionally, unlike LIBSVM, we use SOA (structure of array) instead of AOS (array of structure) for continuous memory access in the data processing phase. The adaptive support for input patterns is necessary for achieving efficient parallelism for dense datasets and reduce memory requirement for extremely sparse datasets.

C. Two-Level Parallelism

The growing concern of power dissipation [22] has moved the focus of modern processors from increasing clock rate to increasing parallelism to improve peak performance without drastic power increment. All our experimental architectures (i.e. MIC and Ivy Bridge abstract architectures are shown in Figure 5) employ two-level parallelism: task and data parallelism.

For Ivy Bridge and MIC, the task parallelism is achieved by utilizing multiple hardware threads. For MIC, the data parallelism benefits from on-core VPU (Vector Processing Unit) and SIMD. For Fermi and Kepler, the task parallelism comes from the independent warps that are executed by different SMs. In each warp, the data-parallelism

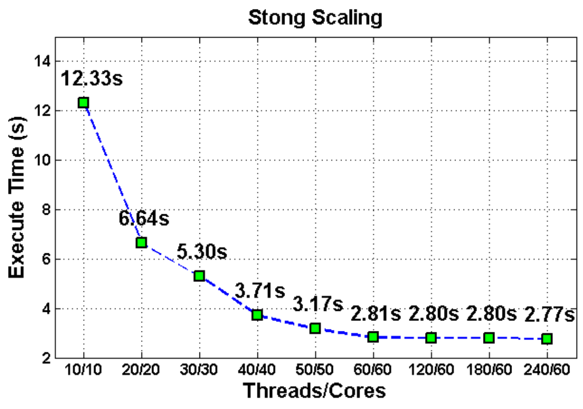

Fig. 6. Results for strong scaling test of MIC-SVM on Intel Xeon Phi using different number of threads and cores. threads/core=number of threads/number of cores.

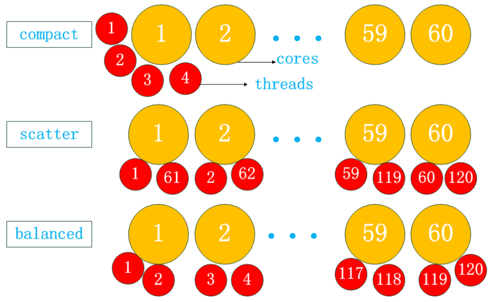

Fig. 7. Three affinity models for load balancing. 1) compact mode: the new thread will be firstly allocated to one core until the core reaches its maximum load; 2) scatter mode: the new thread will be firstly allocated to the core that has the lightest load; 3) balanced mode: it not only assigns the new thread to the core that has the lightest load but also tries to assign the neighboring threads to the same core.

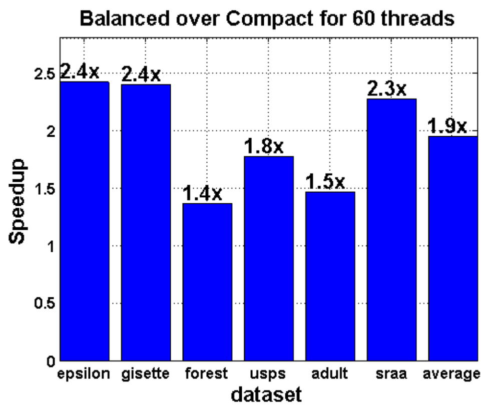

Fig. 8. Performance speedup of using ”balanced mode” over ”compact mode” when running MIC-SVM on Intel Xeon Phi with various types of datasets.

is achieved by the computations performed by the different CUDA cores within the SM. In order to get satisfactory performance, fully utilizing the two-level parallelism and improving the occupancy rate of the computing resources are crucial.

1) Task Parallelism: As mentioned in Section III-A, the evaluation of a given element in $HighKernel$ (the same with $LowKernel$) is independent from any other element (Figure 3). Each kernel-evaluation (e.g. $HighKernel(i)$) needs to access one specific training sample (e.g. X_i) and X_{high} (shared by all). Therefore, each kernel-evaluation needs to access $2 \times nDim$ floating points, which can be categorized as a coarse-granularity operation. In short, the independent computation, independent memory requirement, and coarse parallel granularity will make kernel evaluation a good candidate for task parallelism. For our case, the efficient techniques of task parallelism contain the configuration of proper thread number and balanced utilization of cores and threads.

The proper number of threads: For both CPUs and MIC, each independent hardware device owns a specific amount of physical resources, which provides an inherent task parallelism. Hence, for taking full advantage of this mechanism, a practical way is to set the number of threads according to the number of physical resources in each device. Taking MIC as an example, according to Figure 6, our MIC-SVM implementation shows decent strong scalability even though the performance does not have a significant increment from 60 threads to 240 threads due to physical resource limitation and memory contention from multi-threading.

Load balancing: In our case, we use affinity models (shown in Figure 7) to facilitate load balancing. The affinity models provide different ways to allocating the virtual threads to the proper cores for better performance and resource utilization. The affinity model can have a significant impact on overall performance when using less than maximum number of threads. For instance, Figure 8 shows the performance speedup of using ”balanced” mode over ”compact” mode when running our MIC-SVM on Intel Xeon Phi with various types of datasets. This is because each MIC core supports 4 hardware threads and using compact mode reduces the actual number of physical cores used. Therefore, on MIC, we use ”balanced” mode for our MIC-SVM. In this way, tasks can be decomposed into small portions and distributed across all the physical devices evenly. Balanced mode also works for Ivy Bridge CPUs because only one hardware thread is supported per core.

2) Data Parallelism: After accomplishing efficient task parallelism, another concern is how to achieve data parallelism within each hardware thread for the data-wise operations in individual kernel evaluation. One possible solution to this is applying Single Instruction Multiple Data (SIMD) mechanism, which is supported by both Ivy Bridge CPU and Intel MIC. Additionally, MIC provides VPU (Vector Processing Unit) for more efficient vectorization. MIC also supports 512-bit instruction, which means 16 single-precision or 8 double-precision operations can be executed at one time. We use the Cilk [23] array notation to achieve the data parallelism explicitly rather than applying the implicit compiler model. The explicit data parallelism will help to unleash vectorization by providing context information. To achieve better performance, we apply memory alignment so that vectorization can use aligned load instructions.

Adaptive support for data parallelism: As for some extreme cases, the number of non-zero elements in one training sample (e.g. ≤ 8) is too few to take any advantage of the SIMD instruction. Under such circumstances, both the vectorization and hardware threads are dedicated to task parallelism, which means each kernel evaluation is processed serially. Similar to the adaptive support for data patterns (discussed in Section III-B), our MIC-SVM Library will make the decision on whether employing data parallelism within each kernel evaluation through a trained classifier based on both features of inputs and underlying architectures. This functionality is very useful because it will not discriminate inputs that have already been processed as ”sparse” from being vectorized for best performance. Section IV-D(1)

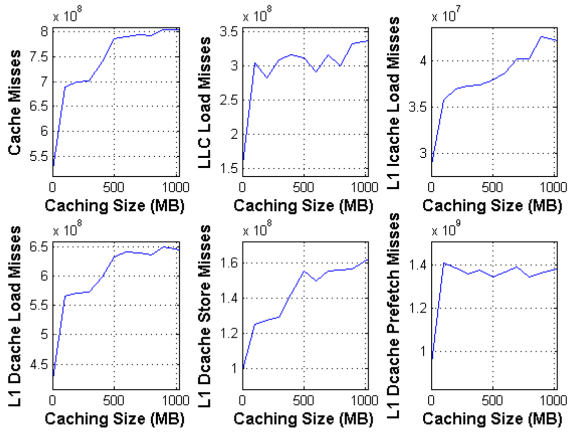

Fig. 9. Memory activities of running MIC-SVM on Intel Xeon Phi architecture while increasing the caching size .

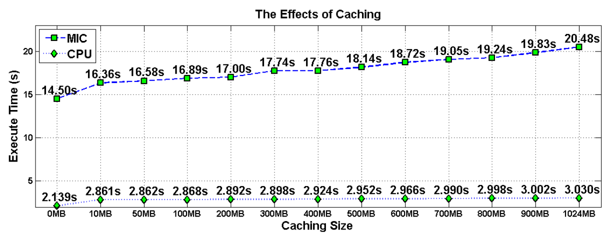

Fig. 10. Caching effects on overall execution time when running MIC-SVM on Intel Xeon Phi (MIC) and Ivy Bridge CPUs under increasing caching size.

will use sraa dataset as an example to illustrate that this functionality is critical at selecting the most suitable parallel granularity for performance.

D. Removing Caching and Reduce Shrinking Frequency

As discussed in Section II-B, in theory, caching strategy can effectively reduce the number of kernels being evaluated. However, it also requires more memory access. Since our parallel SMO is more likely memory bound on our evaluated architectures, applying caching may affect the performance negatively due to high memory access latency, memory contention from multithreading, and limited bandwidth. Figure 9 confirms our assumption that memory activities rise significantly when increasing the caching size for MIC. Figure 10 illustrates the caching effects on overall execution time when running our MIC-SVM on the MIC and Ivy Bridge architectures. It shows that using no additional caching strategy (0 MB) achieves the best overall performance for both cases. Based on similar experiments, we eliminate the caching optimization in our MIC-SVM because it generally brings no performance benefits on the evaluated architectures.

The purpose of shrinking is to get rid of the bounded elements in order to reduce the size of optimization problem (Section II-B). However, the shrinking strategy requires index rearrangement, data marshaling and reconstruction. These operations will incur significant overheads since serial processing, memory allocation and deallocation greatly limits the performance on the low-frequency processors (i.e.MIC cores). Moreover, in order to achieve load balancing, the data has to be scattered and gathered, which increases the overheads for data movements. Therefore, the performance loss may outweigh the gain when shrinking is frequently invoked. We use a trained SVM regression model to predict the best frequency of shrinking based on the information of datasets and the program parameters. Therefore, the general focus has been switched from maximally shrinking the iterations for convergence, which may encounter higher overhead, to reducing the time of each iteration.

E. Reducing the Gap Between RCMA and RCMB

The two-level parallelism requires a large amount of data during a short period of time, which may be beyond the capacity of peak memory bandwidth. This bottleneck indeed is caused by the significant gap between algorithmic RCMA and architectural RCMB (Section III-B), which may further limit the improvement in performance. Therefore, together with load balancing, we employ the OpenMP [24] SCHEDULE technique (static type, default chunk size) to distribute the training samples (one training sample corresponds to one iteration) to all hardware threads evenly. Along with prefetching, this method not only eliminates the overheads of data communication between different hardware threads (through avoiding scatter/gather operation), but also better utilizes abundant caches provided by the CPUs and MIC. In this way, we can effectively reduce the discrepancy between the algorithmic RCMA and architectural RCMB through data reuse.

F. Other Optimization Techniques

In order to further improve the performance, we supplement other useful techniques such as exploring the proper granularity of parallelism by configuring the data sizes for two-level parallelism, minimizing the threads' creation and destroy to reduce synchronization overhead, and maximizing the TLB page size to 2MB to obtain significantly more memory coverage when the datasets require more than 16MB memory. Other optimization details can be found in the SVM-MIC open-source Library.

To sum up, a skeleton parallel SMO algorithm in MIC-SVM is described in Algorithm 2 where T is the number of hardware threads, t is the thread ID, and F, Y are the vector forms of f_i and y_i ($\forall i \in \{1, 2, ..., n/T\}$) respectively. It is only a skeleton algorithm without addressing every individual optimization method.

Algorithm 2 Parallel SMO Algorithm

```
0: Start
1: Input the training samples $X_i$ and pattern labels $y_i$, $\forall i \in \{1, 2, ..., n\}$. Applying adaptive heuristic support to decide how to process input datasets (sparse or dense) and whether to vectorize the kernel or not.
2: Map all the $X_i$, $y_i$, and $\alpha_i$ ($\forall i \in \{1, 2, ..., n\}$) to all the hardware threads evenly through static SCHEDULE.
3: Initializations, for all hardware threads concurrently, $\overrightarrow{\alpha}_t = 0$, $\overrightarrow{F}_t = -\overrightarrow{Y}_t$, $\forall t \in \{1, 2, ..., T\}$.
4: Initializations, $b_{high} = -1$, $i_{high} = \min\{i : y_i = -1\}$, $b_{low} = 1$, $i_{low} = \max\{i : y_i = 1\}$.
5: Check the thread Affinity to keep load balancing.
6: Update $\alpha_{high}$ and $\alpha_{low}$ according to Equation (4) and (5).
7: If kernel is vectorized, then do each kernel evaluation in each hardware thread through vectorization. Update $\overrightarrow{F}_t$ according to Equation (6) through task parallelism, $\forall t \in \{1, 2, ..., T\}$
8: If kernel is un-vectorized, then do each kernel evaluation in each hardware serially. Update $\overrightarrow{F}_t$ according to Equation (6) through the combination of task parallelism and data parallelism, $\forall t \in \{1, 2, ..., T\}$
9: Local reduce to obtain $i_{high}$, $i_{low}$, $b_{high}$, and $b_{low}$ according to Equation (9) and (10) in each hardware thread.
10: Global reduce to obtain $i_{high}$, $i_{low}$, $b_{high}$, and $b_{low}$ according to Equation (9) and (10).
11: Update $\alpha_{high}$ and $\alpha_{low}$ according to Equation (4) and (5).
12: If $b_{low} > b_{high} + 2 \times tolerance$, then go to Step 5.
13: End
```

TABLE III
RELATED PARAMETERS OF ARCHITECTURES

Architecture	Ivy Bridge	KNC	Fermi	Kepler
Cores	24	61	448	2496
L1 cache (KB)	64/core	64/core	64/SM	64/SM
L2 cache (KB)	256/core	512/core	768/card	1536/card
L3 cache (MB)	30/socket	0	0	0
Coherent cache	L3	L2	L2	L2
Memory type	DDR3	GDDR5	GDDR5	GDDR5
Memory size (GB)	64	8	6	6
Measured Memory bandwidth (GB/s)	68	159	97	188

TABLE IV
THE TEST DATASETS

Dataset	n	$nDim$	Density	Cost	Gamma
epsilon [25]	27,000	2,000	1.0000	1.0	0.08000
gisette [26]	6,000	5,000	0.9910	1.0	0.00020
forest [27]	12,000	54	0.2386	10000	0.00010
usps [28]	266,079	675	0.1497	1.0	0.03125
adult [18]	32,561	123	0.1128	0.1	0.06250
sraa [21]	72,309	20,958	0.0024	1.0	0.03125

IV. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experimental Setup

The architecture details of the four evaluated architectures in this paper can be found in Table III and Table II. For the purpose of cross-platform performance evaluation, we further optimized the traditional GPUSVM tool and port it to the new NVIDIA Kepler k20x GPU.

B. Test Datasets

We select several popular real-world datasets (i.e. data mining, digital recognition, aviation,etc) to evaluate MIC-SVM on the evaluated architectures. Their features are shown in Table IV, where n is the number of training samples, $nDim$ is length of each training sample, and density is the ratio of the number of non-zero elements to the total number of elements. We use the Gaussian Kernel for our experiments since it is the most widely-used kernel function. The parameters for Gaussian Kernel (Cost and Gamma) are also shown in Table IV. In order to make n and $nDim$ more representative, we reduce the number of training samples for two datasets (epsilon and forest) without modifying the contents of the corresponding training samples.

C. Correctness Validation for MIC-SVM

We select LIBSVM, which is a widely-used standard SVM library, as baseline to validate the correctness of our implementations. TableV shows the classification accuracy and two impacting factors (b and SVs) of different implementations on the evaluated architectures.

The classification accuracy can be used to describe how close a new implementation to the baseline (e.g.LIBSVM) in terms of prediction accuracy. b is treated as the factor of convergence condition. The support vectors (SVs) can be interrupted as following: Each training sample has its own alpha (Algorithm 2). If a training sample's alpha is not zero after the training process, then this training sample is a support vector. Only support vectors can impact the predication accuracy in the classification process.

From Table V, we can observe that only the result based on gisette dataset has a 0.1% discrepancy with LIBSVM for classification accuracy (comparing the accuracy between the elements in the same row). The accuracy results based on other test datasets are totally identical with LIBSVM. Due to the differences in implementations, there are some small discrepancies in terms of the value of b and the number of Support Vectors. However, they almost have no effect on the classification accuracy. The training process for sraa dataset can not be converged through GPUSVM because it will require 11.6 gigabytes storge for dense format, which is beyond the memory capacity of a single GPU card.

D. Comparisons and Analysis

In this section, we will provide a cross-platform performance comparison analysis between our proposed MIC-SVM and previous tools including LIBSVM (serial) and optimized GPUSVM. Another important objective is to provide insights for users on how to choose the most suitable architecture towards a specific implementation and input data pattern.

Table VI shows the speedups of different implementations on various architectures over LIBSVM (baseline). The results marked bold represent the best speedup achieved by this implementation on such architecture for this specific input dataset. We can observe that our proposed MIC-SVM achieves 4.4 - 84 x and 18 - 47x speedups over the serial LIBSVM through Intel Knights Corner MIC and Ivy Bridge CPUs respectively. Figure 11 shows the speedups of MIC-SVM and GPUSVM over LIBSVM on one single iteration (instead of the total execution time) for various input patterns (shown in Table III). This shows the performance gaps among different scenarios more clearly without the interference of the iterations. We argue that speedup is a combination factor of implementation, architecture and input data pattern. From Table VI and Figure 11, we conduct the following analysis to help us understand how to choose suitable implementation and architecture for a specific input data pattern.

1) MIC-SVM on CPUs vs. MIC-SVM on MIC: **MIC is suitable for dense high-dimension datasets.** From Figure 11, we can observe that the speedup achieved using MIC-SVM on MIC is about twice ($1.8\times$ and $2.1\times$) of that on Ivy Bridge CPUs for the high dimension datasets (Table III) such as epsilon ($2000d$) and gisette ($5000d$). Even for the medium dimension dataset like usps ($675d$), the performance of MIC is still better ($1.1\times$) than that of CPUs. Although sraa has a high dimension (20.958d), it is indeed a low dimension dataset because it is so sparse (density: 0.0024) that our Library automatically process it in sparse method ($20958d \times 0.0024 = 54d$).

Since both epsilon and gisette are dense datasets, MIC-SVM Library automatically applies data parallelism to process each kernel evaluation (involving two training samples) through vectorization (SIMD). Ivy Bridge CPUs are based on Advanced Vector Extensions (AVX) instruction set, which has a 256-bit SIMD register file. However, the SIMD width (512 bits) of Knights Corner MIC is twice of that on CPUs. Therefore, the high-dimension dense dataset is more likely to benefit from the powerful vectorization scheme on MIC than CPUs.

Ivy Bridge CPUs are suitable for coarse-grained parallel processing. Figure 11 shows that MIC-SVM on CPUs outperforms MIC-SVM on MIC by a large margin for adult and sraa datasets. There are several reasons behind this. First, based on our adaptive support for input data patterns (shown in Section III-B), our MIC-SVM Library will automatically process both adult (0.1100) and sraa (0.0024) with sparse method. As for the adult dataset, the MIC-SVM Library does not apply vectorization within each kernel evaluation

TABLE V
THE CLASSIFICATION ACCURACY, THE VALUE OF B AND THE NUMBER OF SUPPORT VECTORS.

Datasets	LIBSVM Accuracy–b–SVs	MIC-SVM on KNC MIC Accuracy–b–SVs	MIC-SVM on Ivy Bridge Accuracy–b–SVs	GPUSVM on Kepler Accuracy–b–SVs	GPUSVM on Fermi Accuracy–b–SVs
epsilon	87.5%–0.0200–18116	87.5%–0.0200–18114	87.5%–0.0200–18113	87.5%–0.0200–18113	87.5%–0.0200–18113
gisette	97.7%–−0.0034–1666	97.6%–−0.0033–1665	97.6%–−0.0033–1665	97.6%–−0.0036–1665	97.6%–−0.0036–1665
forest	82.2%–0.0120–11612	82.2%–0.0120–11609	82.2%–0.0120–11610	82.2%–0.0120–11610	82.2%–0.0120–11610
usps	99.2%–−0.9464–39570	99.2%–−0.9464–38596	99.2%–−0.9464–38589	99.2%–−0.9464–38581	99.2%–−0.9464–38581
adult	84.4%–0.576–12281	84.4%–0.576–12273	84.4%–0.577–12273	84.4%–0.576–12278	84.4%–0.576–12278
sraa	97.0%–−1.33–24430	97.0%–−1.33–24392	97.0%–−1.33–24395	misconvergence	misconvergence

TABLE VI
ITERATIONS, TRAINING TIME AND SPEEDUPS.

Datasets	LIBSVM Iterations–Time(s)–Speedup	MIC-SVM on MIC Iterations–Time(s)–Speedup	MIC-SVM on Ivy Bridge Iterations–Time(s)–Speedup	GPUSVM on Kepler Iterations–Time(s)–Speedup	GPUSVM on Fermi Iterations–Time(s)–Speedup
epsilon	11040–2959–1×	**11686–35.1–84**×	11616–64.2–46×	11537–38.6–76×	11537–46.1–64×
gisette	1938–117.8–1×	**1887–2.75–43**×	1872–5.68–21×	1894–5.46–22×	1894–7.65–15×
forest	23520–77.2–1×	28912–17.6–4.4×	28983–2.14–36×	**29018–2.04–38**×	28946–5.10–15×
usps	34992–21945–1×	47201–806–27×	47173–884–25×	**47195–239–92**×	47195–429–51×
adult	8028–84.1–1×	8092–14.5–5.8×	**8150–2.14–39**×	8097–2.41–35×	8097–5.67–15×
sraa	14802–1128–1×	13687–81.3–14×	**13676–24.9–45**×	misconvergence	misconvergence

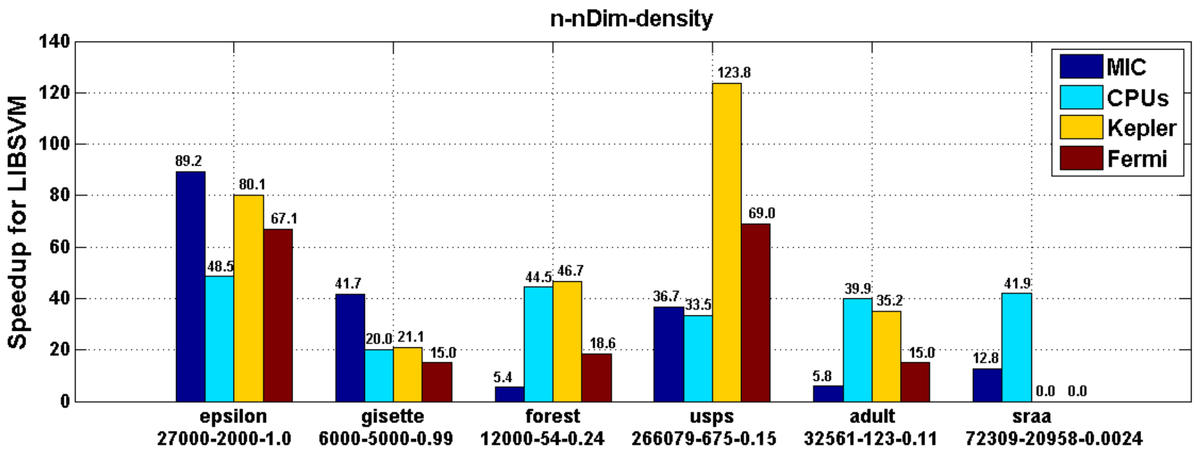

Fig. 11. This figure shows the speedups over LIBSVM based on the time of each iteration (rather than the whole time), n is the number of training samples, $nDim$ is the dimension of each training sample, and density represents the sparseness of a given dataset. All the results data shown in this figure are the average of many runs.

because the dimension ($123d \times 0.11 = 15d$) is too low to fully take advantage of the SIMD instruction (discussed in Section III-C(2)). In this situation, both the vectorization and hardware threads are dedicated to the task parallelism and letting each kernel process serially. Consequently, each CPU hardware thread corresponds to 8 ($\frac{256}{32}$) kernel evaluations while each MIC hardware thread corresponds to 16 ($\frac{512}{32}$) kernel evaluations. Besides, MIC has 240 hardware threads while the Ivy Bridge CPUs in this experiment are only equipped with 24 hardware threads. In other words, the parallelism of MIC is 20 times ($\frac{240 \times 16}{24 \times 8}$) that of CPUs. However, MIC-SVM on CPUs achieves a 7× speedup (Figure 11) over that on MIC, which means each serial kernel evaluation on MIC is $140\times$ slower than that on CPUs.

There are several factors that may lead to the huge performance difference for serial processing between MIC and Ivy Bridge: 1) the difference in clock rate (Table II); 2) each instruction of MIC can not be executed in the consecutive two cycles; 3) MIC core does not support out-of-order execution instruction; 4) CPUs provide additional L3 cache; 5) a MIC core (original Pentium) is composed of much less computational/logical units compared with an Ivy Bridge core.

The coarse-grained parallel processing (often required by high-degree sparse high-dimension datasets) demands many serial processes and abundant local storages, which is in favor of Ivy Bridge CPUs because they are equipped with more functional on-chip buffers and cores with faster clock frequency compared to MIC and GPUs (Table II).

As discussed in Section III-C(2), although the sraa dataset is automatically processed by the sparse method, the heuristic support for data parallelism in our MIC-SVM Library classify sraa as being suitable for vectorization within each kernel evaluation (unlike for adult) because the heuristic still decides its dimension is big enough for some kind of speedup through vectorization

TABLE VII
THE SPEEDUPS OF GPUSVM ON KEPLER OVER MIC-SVM ON MIC

Dataset	gisette	epsilon	usps	forest
Dimension	5,000	2,000	123	54
Speedup	0.5	0.9	3.4	8.7

($20958d \times 0.0024 = 54d$). Even though the data parallelism still can not meet the requirement of the wide MIC SIMD, it does help reduce the performance discrepancy (from $7\times$ to $3\times$) between MIC and CPUs (Figure 11). This example clearly proves the importance of the adaptive support for data parallelism.

2) MIC-SVM on MIC vs. GPUSVM on GPUs: In Figure 11, both GPUSVM and MIC-SVM apply the dense method to process epsilon, gisette, forest, and usps datasets, of which epsilon (205 MB) and usps (685 MB) require the largest memory. For these four datasets, GPUSVM and MIC-SVM have employed similar processing techniques: using one thread to handle one training sample each time. Additionally, Table VII also shows that as the dimension of dataset decreases, the speedup of GPUSVM on Kepler over MIC-SVM on MIC increases. This suggests that the architectural differences in granularity of two-level parallelism could be the major reason causing the performance variation with different sample number, dimension and density.

GPUs employ the SIMT (Single Instruction Multiple Thread) architecture. The multiprocessor creates, schedules, and executes a group of threads concurrently. Similarly, MIC is based on the SIMD architecture, which is the foundation for vectorization scheme on MIC. Although both SIMD and SIMT are useful for enhancing parallelism, there are essential differences between them in terms of the granularity of parallelism: (1) SIMD: one MIC thread executes one 512-bit instruction. Therefore, one MIC thread corresponds to 16 single precision operations and 32 single precision floating points (or 8 double precision operations and 16 double precision floating points); (2) SIMT: 32 threads (a warp) execute the same 32 instructions concurrently. Thus, one thread corresponds to one operation and 2 single precision floating points.

In terms of the core architecture, a MIC core is composed of more complicated units (i.e. code cache, instruction decode, scalar and vector units, and L1 cache) than a GPU core. Therefore, the GPU thread is more lightweight and provides finer-grain parallelization compared to a MIC thread.

For gisette and epsilon datasets, due to their high dimensions ($5000d$ and $2000d$) and density (0.99 and 1.0), the efficient data parallelism on MIC can benefit from the large number of non-zero elements in each training sample. The powerful MIC instruction is well utilized through efficient vectorization. For usps and forest dataset, due to their high sample number, low dimensions and density (Table IV), SVM on GPU can achieve a better speedup over that on MIC because its large number of training samples (i.e. 266,079) can provide enough parallelism for millions of GPU lightweight threads without suffering from low data-level parallelism.

With the additional performance discrepancies (i.e. different thread switching speed, clock frequency, memory bandwidth, and mechanism to control cache), the analysis above is still in line with our experimental results. To sum it up, we have the following conclusions: (1) With powerful SIMD mechanism (512 bits), MIC is a good candidate for the dense high-dimension datasets with modest number of training samples because data parallelism can be achieved efficiently through vectorization; (2) Equipped with sufficient caches and high clock frequency, Ivy Bridge CPUs are suitable for sparse high-dimension datasets since these datasets often require coarse-grained parallel processing; (3) GPUs are proper for the datasets with large number of training samples and low dimension because they are more likely to benefit from millions of fine-grained lightweight threads (under resource limitation though). Additionally, in order to make GPUSVM practical for all cases including the extremely sparse datasets like sraa, the adaptive support for input patterns and data-parallelism is very necessary.

V. RELATED WORK

In the last twenty years, continuous efforts have been made to improve the performance of SVM. Some pioneers proposed strategies for faster serial algorithms such as dataset decomposition technique [16], points shrinking, caching [17], minimal working set [18], and second order working set selection [29]. Most of these techniques have already been adapted in the widely used LIBSVM [10], which is designed for serial processing. Others have tried to parallel SVM on distributed memory systems ([30], [31], [32]) without considering underlying architecture details.

Since 2008, there have been some existing efforts for accelerating the time-consuming training phase in SVM on many-core GPUs. Almost all of them have been focusing on using GPUs to accelerate the SMO [18] algorithm. Catanzaro [12] first proposed the GPUSVM for binary classification problem and achieved speedup of 9-35$\times$ over LIBSVM. Herrero-Lepez then [33] improved Catanzaro's work by adding the support for Multiclass classification. Tsung-Kai Lin [34] applied sparse format for data processing on GPU without providing implementation details, source code or Library. All of the above are based on older generations of GPUs (before Kepler) so many new architectural improvements are not considered for optimization. They all lack dynamic adaptive support for input data patterns and data parallelism, which can dramatically reduce performance and practicality of the implementation. For the purpose of comparison and analysis, we further optimize the existing GPUSVM for Kepler and compare its performance with our MIC-SVM.

Since the emerging of the Intel MIC architecture, there have been a few work on applying MIC to solve data-intensive applications such as sorting [35], data mining [36], and ray tracing [37]. All of these work are based on the older version of the Knights Ferry MIC (KNF) rather than the more advanced Knights Corner MIC (KNC) [14]. To authors' knowledge, our proposed MIC-SVM is the first trail for designing a highly efficient SVM for advanced multi- and many-core architectures such as Ivy Bridge CPUs and Intel KNC MIC. Our approach also provides methodologies and techniques such as adaptive support for data patterns and parallelism that can be generalized to optimize similar machine learning methods. Additionally, we also provide insights on how to select the most suitable architecture for a specific implementation and input data pattern, which has not been addressed in the previous work.

VI. CONCLUSION

In this work, we propose MIC-SVM, a highly efficient parallel support vector machine for x86 based multi-core and many-core architectures such as Intel Ivy Bridge CPUs and Intel KNC MIC. We propose various novel analysis and optimization strategies that are general and can be easily applied to accelerate other machine learning methods. We also explore and improve the deficiencies of the current SVM tools. Finally, we provide insights on how to map the most suitable architectures to specific data patterns in order to achieve the best performance. In future, we plan to extend the current MIC-SVM to distributed memory environment using multiple MICs. We also intend to employ MIC-SVM at runtime for dynamic modeling and scheduling.

ACKNOWLEDGMENT

This work was supported in part by National Natural Science Foundation of China (61303003 and 41374113) and National High-tech R&D (863) Program of China (2013AA01A208), and DOEnOSCAR Beyond Standard Model Project #62855.

REFERENCES

[1] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[2] T. Joachims, *Text categorization with support vector machines: Learning with many relevant features*. Springer, 1998.

[3] F. E. Tay and L. Cao, "Application of support vector machines in financial time series forecasting," *Omega*, vol. 29, no. 4, pp. 309–317, 2001.

[4] C. Leslie, E. Eskin, and W. S. Noble, "The spectrum kernel: A string kernel for svm protein classification," in *Proceedings of the Pacific symposium on biocomputing*, vol. 7. Hawaii, USA., 2002, pp. 566–575.

[5] B. L. Milenova, J. S. Yarmus, and M. M. Campos, "Svm in oracle database 10g: removing the barriers to widespread adoption of support vector machines," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 1152–1163.

[6] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *Proceedings of the 39th International Symposium on Computer Architecture*. IEEE Press, 2012, pp. 440–451.

[7] G. Chrysos, "Knights corner, intels first many integrated core (mic) architecture product," in *Hot Chips*, vol. 24, 2012.

[8] C. Su, D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. d. Supinski, and E. A. Leon, "Model-based, memory-centric performance and power optimization on numa multiprocessors," in *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 164–173. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2012.6402921

[9] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504189

[10] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[11] M. Hu and W. Hao, "A parallel approach for svm with multi-core cpu," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 15, 2010, pp. V15–373–V15–377.

[12] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 104–111.

[13] J. Dongarra. (2012) Top500 june 2013 list. [Online]. Available: http://www.top500.org/lists/2013/06/

[14] A. Duran and M. Klemm, "The intel® many integrated core architecture," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 2012, pp. 365–366.

[15] V. N. Vapnik and S. Kotz, *Estimation of dependences based on empirical data*. Springer-Verlag New York, 1982, vol. 41.

[16] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," in *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*. IEEE, 1997, pp. 276–285.

[17] T. Joachims, "Making large scale svm learning practical," 1999.

[18] J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization," 1999.

[19] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural Computation*, vol. 13, no. 3, pp. 637–649, 2001.

[20] M.-S. group. (2013) Mic-svm v0.1. [Online]. Available: http://github.com/You1991/MICSVM

[21] G. S. Mann and A. McCallum, "Simple, robust, scalable semi-supervised learning via expectation regularization," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 593–600.

[22] T. Mudge, "Power: A first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[23] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

[24] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[25] S. Sonnenburg, V. Franc, E. Yom-Tov, and M. Sebag, "Pascal large scale learning challenge," in *25th International Conference on Machine Learning (ICML2008) Workshop. http://largescale. first. fraunhofer. de. J. Mach. Learn. Res*, vol. 10, 2008, pp. 1937–1953.

[26] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror, "Result analysis of the nips 2003 feature selection challenge," in *Advances in Neural Information Processing Systems*, 2004, pp. 545–552.

[27] R. Collobert, S. Bengio, and Y. Bengio, "A parallel mixture of svms for very large scale problems," *Neural computation*, vol. 14, no. 5, pp. 1105–1114, 2002.

[28] J. J. Hull, "A database for handwritten text recognition research," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, no. 5, pp. 550–554, 1994.

[29] R.-E. Fan, P.-H. Chen, and C.-J. Lin, "Working set selection using second order information for training support vector machines," *The Journal of Machine Learning Research*, vol. 6, pp. 1889–1918, 2005.

[30] H. P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," *Advances in neural information processing systems*, vol. 17, pp. 521–528, 2004.

[31] L. J. Cao, S. Keerthi, C.-J. Ong, J. Zhang, U. Periyathamby, X. J. Fu, and H. Lee, "Parallel sequential minimal optimization for the training of support vector machines," *Neural Networks, IEEE Transactions on*, vol. 17, no. 4, pp. 1039–1049, 2006.

[32] E. Y. Chang, "Psvm: Parallelizing support vector machines on distributed computers," in *Foundations of Large-Scale Multimedia Information Management and Retrieval*. Springer, 2011, pp. 213–230.

[33] S. Herrero-Lopez, J. R. Williams, and A. Sanchez, "Parallel multiclass classification using svms on gpus," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 2–11.

[34] T.-K. Lin and S.-Y. Chien, "Support vector machines on gpu with sparse matrix format," in *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*. IEEE, 2010, pp. 313–318.

[35] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey, "Fast sort on cpus, gpus and intel mic architectures," Technical report, Intel, Tech. Rep., 2010.

[36] A. Heinecke, M. Klemm, D. Pflüger, A. Bode, and H. Bungartz, "Extending a highly parallel data mining algorithm to the intel® many integrated core architecture," in *Euro-Par 2011: Parallel Processing Workshops*. Springer, 2012, pp. 375–384.

[37] C. Benthin, I. Wald, S. Woop, M. Ernst, and W. Mark, "Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 9, pp. 1438–1448, 2012.