Scaling Support Vector Machines on modern HPC platforms

Yang You\textsuperscript{a,b}, Haohuan Fu\textsuperscript{a,*}, Shuaiken Leon Song\textsuperscript{c}, Amanda Randles\textsuperscript{d}, Darren Kerbyson\textsuperscript{c}, Andres Marquez\textsuperscript{c}, Guangwen Yang\textsuperscript{b}, Adolfy Hoisie\textsuperscript{c}

\textsuperscript{a} Ministry of Education Key Laboratory for Earth System Modeling, and Center for Earth System Science, Tsinghua University, Beijing, China
\textsuperscript{b} Department of Computer Science and Technology, Tsinghua University, Beijing, China
\textsuperscript{c} Performance and Architecture Lab, Pacific Northwest National Lab, Richland, WA, United States
\textsuperscript{d} Lawrence Livermore National Lab, Livermore, CA, United States

Highlights

- An efficient parallel support vector machine for x86 based multi-core platforms.
- The novel optimization techniques to fully utilize the multi-level parallelism.
- The improvement for the deficiencies of the current SVM tools.
- Select the best architectures for input data patterns to achieve best performance.
- The large-scale distributed algorithm and power-efficient approach.

Abstract

Support Vector Machines (SVM) have been widely used in data-mining and Big Data applications as modern commercial databases start to attach an increasing importance to the analytic capabilities. In recent years, SVM was adapted to the field of High Performance Computing for power/performance prediction, auto-tuning, and runtime scheduling. However, even at the risk of losing prediction accuracy due to insufficient runtime information, researchers can only afford to apply offline model training to avoid significant runtime training overhead. Advanced multi- and many-core architectures offer massive parallelism with complex memory hierarchies which can make runtime training possible, but form a barrier to efficient parallel SVM design.

To address the challenges above, we 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 and serve as general optimization methods for other machine learning tools.

MIC-SVM achieves 4.4–84× and 18–47× speedups against the popular LIBSVM, on MIC and Ivy Bridge CPUs respectively, for several real-world data-mining datasets. Even compared with GPUSVM, running on the NVIDIA k20x GPU, the performance of our MIC-SVM is competitive. We also conduct a cross-platform performance comparison analysis, focusing on Ivy Bridge CPUs, MIC and GPUs, and provide insights on how to select the most suitable advanced architectures for specific algorithms and input data patterns.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

Support Vector Machine (SVM) \cite{10} (Figs. 1–6) is a classification method that has been widely used in text categorization \cite{22}. financial analysis \cite{42}, bioinformatics \cite{25} and many other fields. SVM has been employed by the advanced databases like Oracle (10g, 11g, 12c) \cite{30} as a major data mining technique as companies increasingly rely on their analytic capabilities. However, the time-consuming training process greatly limits the efficiency of SVM. This concern is likely to be magnified by the increasing volume of data in the Big Data era. Meanwhile, this issue is also exacerbated by the loss of increase in clock frequency and rise of multi- and many-core architectures, whose massive parallelism
and complex memory hierarchies form a barrier to efficient parallel SVM design (due to its memory bound and irregular memory access bottlenecks). While programmers in the past could depend on the ready-made performance improvement that comes from a faster clock frequency, now they have to face the challenge of scaling the performance over tens or even hundreds of cores within a single chip often operating at a lower frequency [38]. One of the most representative multicore architectures is Intel Xeon Phi Co-processor (MIC) [7].

In the field of High Performance Computing (HPC), SVMs have recently been adapted at runtime for performance and power prediction at system and compiler level for design space exploration [41] [45]. The common approach of the current work is to train the SVM model offline and then apply the static model for online prediction in order 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 current runtime behavior information. In order to use SVM at runtime for dynamic modeling and scheduling in HPC, a method is needed to accelerate its training phase on modern advanced multi- and many-core architectures.

Previous work either focused on designing SVM tools for CPUs with relatively few cores and simple memory hierarchies, such as the serial LIBSVM [6] or [19], or creating techniques to accelerate SVM on GPUs such as GPUSVM [4]. However, there has been no efficient SVM tool designed for advanced x86 based multi- and many-core architectures, even though they have already gained popularity on recent TOP500 list [12]. Meanwhile, 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 [13]. Our goal is to design and implement an open-source SVM Library that is not only highly efficient but also can be easily adopted to the existing runtime systems or software. We also want to create methods and techniques that are general enough to be applied to optimize other similar machine learning models.
The contributions of this work include:

- The design and implementation of MIC-SVM (open-source library), a highly efficient parallel support vector machine for x86 based multi- and many-core architectures such as MIC.
- The development of 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.
- The exploration and improvement of the deficiencies of the current SVM tools such as LIBSVM and GPUSVM.
- The method 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 4.4–84× and 18–47× 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.

2. Background

Support Vector Machines (SVMs) (Figs. 1–6) consist of two major phases: training and classification. The user obtains the model file (which contains the best decision boundary) through training process and uses it to make prediction in the classification process. Based on our performance profiling, the majority of SVM execution time has been spent on the training phase and this makes it the targeted performance bottleneck for SVM.

2.1. 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: \(X_i, i \in 1, 2, \ldots, n\) and \(y_i, i \in 1, 2, \ldots, n\). 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).

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

Subject to : \(\sum_{i=1}^{n} \alpha_i y_i = 0\) and \(0 \leq \alpha_i \leq C, \forall i \in 1, 2, \ldots, n. \) (1)

SVM training can be presented as a linear-constraint convex Quadratic Programming (QP) problem (Eq. (1)), where \(C\) represents the regularization constant that balances the generality and accuracy, \(\alpha_i\) is the Lagrange multiplier, and \(\text{Kernel}\) denotes the Kernel function. \(C\) can be set by users and each \(\alpha_i\) is related to a specific training sample. The standard Kernel functions in SVM are shown in Table 1.

2.2. Sequential minimal optimization (SMO)

To make large scale SVM training problems practical, several algorithms have been proposed, including chunking [44], decomposition [34], and caching/shrinking [23]. 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.

| Table 1 Standard Kernel functions, the related parameters are explained in [39]. |
|-----------------|------------------|
| Linear: \(\text{Kernel}(X_i, X_j) = X_i X_j\) | Polynomial: \(\text{Kernel}(X_i, X_j) = (\alpha X_i + r)^d\) |
| Gaussian: \(\text{Kernel}(X_i, X_j) = -\gamma \|X_i - X_j\|^2\) | Sigmoid: \(\text{Kernel}(X_i, X_j) = \tanh(\alpha X_i + r)\) |

To minimize the size of the working set in the training phase, Sequential Minimal Optimization (SMO) algorithm was proposed [35, 24] and implemented in several popular SVM tools including LIBSVM [6]. 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 6).

A brief outline of the commonly used SMO (serial code) algorithm is presented in Algorithm 1. All the mathematical derivation of the major parameters in Algorithm 1 are shown in Eqs. (2)–(6) given in Box I. Specifically, \(f(i \in \{1, 2, \ldots, n\})\) represents the discrepancy between the calculated objective value and the real objective value (Eq. (2)) for each point. The update of all \(f(i)\) (Eq. (2)) at each step requires to access all the training samples, which costs more than 90% of time in the entire SMO algorithm. Therefore, accelerating the update of \(f(i)\) is our top priority for optimization. Algorithm parallelization and detailed optimization methodologies will be shown in Section 4.

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 (e.g., \(\text{Kernel}(X_i, X_j)\)) in the remaining memory through the least-recent-use (LRU) strategy in order 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 \(\alpha_i\) 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 cause negative performance impact. For instance, if the algorithm is memory bound, the caching strategy may not bring any performance benefit for these architectures due to higher memory access cost, memory contention from multi-threading and limited bandwidth. In Section 4, we will show analysis on how these traditional optimizations may not be suitable for our paralleled MIC-SVM tool designed based on the features of modern multi- and many-core architectures.

Algorithm 1 The Original SMO Algorithm

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

3. Evaluated architectures

We have designed our parallel SMO algorithm and accelerated it on the most advanced Xeon-family Multi-Core and Many-Core architectures: Intel Ivy Bridge CPUs and Intel Xeon Phi.
coprocessor. We also transplanted the GPUSVM to the new Tesla-family platforms: Nvidia Fermi C2070 GPU and Nvidia Kepler 20x GPU. The important parameters of these architectures are listed in Table 2.

### 3.1. Memory hierarchy

#### 3.1.1. x86 cores

The Ivy Bridge architecture used for the comparison analysis in this paper is shown in Fig. 7. It is composed of two Xeon E5-2692 v2 CPUs and twenty-four x86 cores (twelve cores per CPU socket) in total. Each core has a 64 KB private L1 cache and a 512 KB private L2 cache. Additionally, each CPU is equipped with an extra 30 MB coherent L3 cache which is shared by all the 12 cores. These two 30 MB L3 caches are connected by the 8GT/s Intel Quickpath Interconnect, providing real-time data exchanges between different cores. In addition to the limited high-speed three-level cache, each CPU possesses four memory channels, providing a 128 GB/s theoretical bandwidth for the whole system.

Similar to the Ivy Bridge architecture, our Intel Knights Corner MIC (shown in Fig. 7) also consists of many x86 cores, Quickpath Interconnect (3.5 GT/s) and several memory controllers. Nevertheless, there are several distinct differences between these two architectures: (1) the KNC MIC provides a significantly higher number of cores (61 in total, 60 for computation and one for OS management) than the Ivy Bridge CPU; (2) the MIC core is based on the Intel P54 (the first generation of Pentium) micro-architecture, which is much simpler compared with a Ivy Bridge core; (3) for memory bandwidth, KNC MIC can provide a 352 GB/s theoretical bandwidth (5.5GTransfers/s × 16 channels × 4B/Transfer) and 159 GB/s practical bandwidth (STREAM benchmark), which are almost three times of that of Ivy Bridge; (4) MIC does not have a L3 cache but has a 31 MB coherent L2 cache (512 KB for each core).

### 3.1.2. GPU

Each Fermi GPU card is equipped with 14 streaming multiprocessors (SM). In each SM (Fig. 7), there is a 64 KB high-speed on-chip buffer that can be configured as either 48 KB shared memory + 16 KB L1 cache (default), or the other way round. Each SM also has a 12 KB read-only texture cache. Additionally, there is a unified 768 KB L2 cache that is shared among all the SMs, and a 6 GB on-board GDDR5 memory with a bandwidth of 192 GB/s.

There are some notable architectural improvements in terms of memory from Kepler over Fermi: (1) users have an additional

---

**Table 2**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Name</th>
<th>Ivy Bridge Xeon E5-2697</th>
<th>MIC Xeon Phi 7110P</th>
<th>Fermi TESLA M2090</th>
<th>Kepler Tesla GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (GHz)</td>
<td></td>
<td>2.20</td>
<td>1.09</td>
<td>1.50</td>
<td>0.73</td>
</tr>
<tr>
<td>Double Precision Peak Performance (Gflops)</td>
<td></td>
<td>422</td>
<td>1010</td>
<td>515</td>
<td>1320</td>
</tr>
<tr>
<td>Single Precision Peak Performance (Gflops)</td>
<td></td>
<td>844</td>
<td>2020</td>
<td>1030</td>
<td>3950</td>
</tr>
<tr>
<td>L1 cache (KB)</td>
<td></td>
<td>32/core</td>
<td>32/core</td>
<td>64/SM</td>
<td>64/SM</td>
</tr>
<tr>
<td>L2 cache (KB)</td>
<td></td>
<td>512/core</td>
<td>768/card</td>
<td>1536/card</td>
<td></td>
</tr>
<tr>
<td>L3 cache (MB)</td>
<td></td>
<td>30/socket</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Coherent cache</td>
<td>L3</td>
<td></td>
<td>L2</td>
<td>L2</td>
<td></td>
</tr>
<tr>
<td>Memory type</td>
<td>DDR3</td>
<td></td>
<td>GDDR5</td>
<td>GDDR5</td>
<td>GDDR5</td>
</tr>
<tr>
<td>Memory size (GB)</td>
<td></td>
<td>64</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Theoretical Memory bandwidth (GB/s)</td>
<td></td>
<td>128</td>
<td>352</td>
<td>192</td>
<td>250</td>
</tr>
<tr>
<td>Measured Memory bandwidth (GB/s)</td>
<td></td>
<td>68</td>
<td>159</td>
<td>97</td>
<td>188</td>
</tr>
<tr>
<td>Single Precision RCMB (Gflops/GB)</td>
<td></td>
<td>6.59</td>
<td>5.74</td>
<td>5.36</td>
<td>15.8</td>
</tr>
<tr>
<td>Double Precision RCMB (Gflops/GB)</td>
<td></td>
<td>3.30</td>
<td>2.87</td>
<td>2.68</td>
<td>5.28</td>
</tr>
</tbody>
</table>

**Box 1.**

\[
f_i = \sum_{j=1}^{n} \alpha y_i \text{Kernel}(X_i, X_j) - y_i, \quad f_i = f_i + (\hat{\alpha}_{\text{high}} - \alpha_{\text{high}}) y_{\text{high}} \text{Kernel}(X_{\text{high}}, X_i) + (\alpha_{\text{low}} - \alpha_{\text{low}}) y_{\text{low}} \text{Kernel}(X_{\text{low}}, X_i)
\]  

\[
\hat{\alpha}_{\text{low}} = \alpha_{\text{low}} + \frac{y_{\text{low}} (b_{\text{high}} - b_{\text{low}})}{\text{Kernel}(X_{\text{high}}, X_{\text{high}}) + \text{Kernel}(X_{\text{low}}, X_{\text{low}}) - 2\text{Kernel}(X_{\text{high}}, X_{\text{low}})}
\]

\[
\hat{\alpha}_{\text{high}} = \alpha_{\text{high}} + y_{\text{low}} y_{\text{high}} (\alpha_{\text{low}} - \hat{\alpha}_{\text{low}})
\]

\[
i_{\text{high}} = \arg \min \{ f_i : i \in I_{\text{high}} \}, \quad i_{\text{low}} = \arg \max \{ f_i : i \in I_{\text{low}} \}, \quad b_{\text{high}} = \min \{ f_i : i \in I_{\text{high}} \}, \quad b_{\text{low}} = \max \{ f_i : i \in I_{\text{low}} \}
\]

\[
i_{\text{high}} = \{ i : 0 < \alpha_i < C \} \cup \{ i : y_i > 0, \alpha_i = 0 \} \cup \{ i : y_i < 0, \alpha_i = C \}
\]

\[
i_{\text{low}} = \{ i : 0 < \alpha_i < C \} \cup \{ i : y_i > 0, \alpha_i = 0 \} \cup \{ i : y_i < 0, \alpha_i = 0 \}
\]
option to divide 64 KB on-chip buffer to 32 KB + 32 KB; (2) the size of the read-only texture cache increases from 12 KB to 48 KB; (3) the size of the L2 cache doubles; and (4) the bandwidth of global memory increases to 250 GB/s.

3.2. Processing power

The growing concern of power dissipation [31] has moved the focus of modern processors from increasing clock rate to increasing parallelism to improve peak performance without drastic power increment. Our experimental architectures employ two-level parallelism: task and data parallelism.

4. 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. Fig. 8 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 [29].

4.1. Analysis for algorithm and architecture

As mentioned in Section 2.2, the updates of \( f_i \) (\( i \in \{ 1, 2, \ldots, n \} \)) at each step is the most time consuming phase for the SMO algorithm because it requires to access all the training samples. Fortunately, for any pair of \( i, j \in \{ 1, 2, \ldots, n \} \), the updates of \( f_i \) and \( f_j \) are independent of each other. Therefore, we can convert the serial form (shown in Eq. (2)) to the parallel form (shown in Eq. (7)):

\[
\hat{F} = F + (\alpha_{\text{high}} - \alpha_{\text{low}}) y_{\text{high}} \text{HighKernel} \\
+ (\alpha_{\text{low}} - \alpha_{\text{low}}) y_{\text{low}} \text{LowKernel}
\]

where \( F \), HighKernel, and LowKernel are vectors.
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.

### 4.2. Adaptive heuristic support for input datasets

In MIC-SVM, we provide a 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 formats 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 always applies dense format for efficient data processing. This may make GPUSVM difficult for handling some sparse datasets as it can significantly increase the memory requirements. 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 [28] for example, it only requires 27 MB in sparse format. While in dense format, 11.6 GB is required. This large data size in dense format would make the training process impossible to run on a single GPU card (Table 8).

However, the inability of Fermi or Kepler GPUs to run the sraa dataset is not due to hardware restrictions, but due to the implementation used by GPUSVM. If GPUSVM applies sparse format, it may perform better for the sparse dataset like sraa. However, the sparse format may decrease the efficiency in parallel processing, especially for the dense dataset. Therefore, the best approach is the heuristic method that being able to support different kinds of data formats and select the best format according to the information of the incoming dataset. Once GPUSVM is equipped with this kind of function, it will become highly efficient for all kinds of real-world applications.

To address the issues above, we apply adaptive support for handling different types of datasets in our MIC-SVM. Fig. 10 illustrates the process flow. We construct a classification model (or classifier) from the information of training N (N ≥ 100) different real-world datasets, which could cover a variety of data patterns. Then any given dataset will be classified as +1 (dense-like) or −1 (sparse-like) by the pre-generated classifier according to its features such as n, nDim, and density. After that, our MIC-SVM Library will select the corresponding heuristic that is optimal for the specific dataset. 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 while also reducing memory requirement for extremely sparse datasets.

To evaluate our method, we remove the adaptive method from our implementation, and we call this version as No-Adaptive method. No-Adaptive method only provides one type of data format (random selection), which often use the wrong format for a given dataset. We run the No-Adaptive and Adaptive method on the same architectures for the same test datasets. The results are summarized in Table 3. Compared with No-Adaptive method, Adaptive method normally achieve $4.0 \times -6.1 \times$ speedups over No-Adaptive method. At the worst case, Adaptive method only has $2.3 \times$ speedup over No-Adaptive method. In the best situation, however, Adaptive method can achieve $89 \times$ speedup over No-Adaptive method. Overall, Adaptive method has a average of $18.3 \times$ speedup over No-Adaptive method, which proves that the adaptive support is essential for a better performance.

### 4.3. Two-level parallelism

All our experimental architectures (i.e. MIC and Ivy Bridge abstract architectures are shown in Fig. 7) employ two-level parallelism: task and data parallelism.

#### 4.3.1. Task parallelism

As mentioned in Section 4.1, the evaluation of a given element in HighKernel (the same with LowKernel) is independent from any other element (Fig. 9). Each kernel-evaluation (e.g. HighKernel(i)) needs to access one specific training sample (e.g. $X_i$) and $X_{\text{high}}$ (shared by all). In other words, each kernel-evaluation has to access $2 \times nDim$ floating points, which can be categorized as a coarse-granularity operation. The independent computation, independent memory requirement, and coarse parallel granularity will make the kernel evaluation a good candidate for task parallelism. 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,
Table 3
Speedup of adaptive method over no-adaptive method, more information about the dataset can be found in Section 6.2.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Gisette</th>
<th>Epsilon</th>
<th>Sraa</th>
<th>Adult</th>
<th>Forest</th>
<th>Usps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute time of adaptive method</td>
<td>5.7 (s)</td>
<td>64.2 (s)</td>
<td>243 (s)</td>
<td>2.1 (s)</td>
<td>2.1 (s)</td>
<td>884.2 (s)</td>
</tr>
<tr>
<td>Execute time of no-adaptive method</td>
<td>23.7 (s)</td>
<td>257.9 (s)</td>
<td>2159.3 (s)</td>
<td>8.4 (s)</td>
<td>12.9 (s)</td>
<td>2013.6 (s)</td>
</tr>
<tr>
<td>Speedup</td>
<td>4.2×</td>
<td>4.0×</td>
<td>89.0×</td>
<td>4.0×</td>
<td>6.1×</td>
<td>2.3×</td>
</tr>
</tbody>
</table>

Fig. 11. Results for strong scaling test of MIC-SVM on Intel Xeon Phi using different number of threads and cores. Threads/cores means number of threads and number of cores.

Fig. 12. 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. 13. Performance speedup of using balanced mode over compact mode when running MIC-SVM on Intel Xeon Phi with various types of datasets.

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 Fig. 11, 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.** We use three different affinity models (Fig. 12) to facilitate load balancing. The affinity models provide different ways to allocate the virtual threads to the cores for better performance and resource utilization. There are three affinity models for load balancing. The first one is compact mode: the new thread will be firstly allocated to one core until the core reaches its maximum load; another one is scatter mode: the new thread will be firstly allocated to the core that has the lightest load; the third one is 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. The affinity model can have a significant impact on overall performance when using less than maximum number of threads. For instance, Fig. 13 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 so that tasks can be decomposed into small portions and distributed across all the physical devices evenly.

4.3.2. Data parallelism

After accomplishing efficient task parallelism, data parallelism within each hardware thread still needs to be achieved 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 [2] 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 4.2), 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 6.6.1 will use sraa dataset as an example to illustrate that this functionality is critical at selecting the most suitable parallel granularity for performance.
4.4. Removing caching and reduce shrinking frequency

As discussed in Section 2.2, in theory, caching strategy can effectively reduce the number of kernels being evaluated. However, it also requires more memory access because the algorithmic caching size is normally much larger than the first level cache size of the computer hardware. 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 multi-threading, and limited bandwidth. Fig. 14 confirms our assumption that memory activities rise significantly when increasing the caching size for MIC. Fig. 15 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 2.2). 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 overhead for data movement. 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 \((2.7 \times \text{speedup for each iteration})\).

4.5. 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 4.2), which may further limit the improvement in performance. Therefore, together with load balancing, we employ the OpenMP [11] 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 pre-fetching (we tried both compiler guidance and manual tuning), 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.

4.6. 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 2 MB to obtain significantly more memory coverage when the datasets require more than 16 MB 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, $i$ is the thread ID, and $F_i$ and $Y_i$ are the vector forms of $f_i$ and $y_i$ ($\forall i \in \{1, 2, \ldots, n\}$) respectively. It is only a skeleton algorithm without addressing every individual optimization method.

**Algorithm 2** Parallel SMO Algorithm

1. Input the training samples $X$, and pattern labels $y_i$, $\forall i \in \{1, 2, \ldots, n\}$.
   Apply 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 $\alpha_i$ ($\forall i \in \{1, 2, \ldots, n\}$) to all the hardware threads evenly through static SCHEDULE.
3. Initializations, for all hardware threads concurrently. $\alpha^T_0 = 0$, $\vec{F}_i = -Y_i$, $\forall t \in \{1, 2, \ldots, T\}$.
4. Initializations, $b_{high} = -1$, $b_{low} = \min[i : y_i = -1]$, $b_{high} = 1$, $b_{low} = \max[i : y_i = 1]$.
5. Check the thread Affinity to keep load balancing.
6. Update $a_{high}$ and $a_{low}$ according to Equation (3).
7. If kernel is vectorized, then do each kernel evaluation on each hardware thread through vectorization. Update $\vec{F}_i$ according to Equation (2) through task parallelism, $\forall t \in \{1, 2, \ldots, T\}$.
8. If kernel is un-vectorized, then do each kernel evaluation in each hardware serially. Update $\vec{F}_i$ according to Equation (2) through the combination of task parallelism and data parallelism, $\forall t \in \{1, 2, \ldots, T\}$.
9. Local reduce to obtain $b_{high}$, $b_{low}$, $b_{high}$, and $b_{low}$ according to Equation (4) in each hardware thread.
10. Global reduce to obtain $b_{high}$, $b_{low}$, $b_{high}$, and $b_{low}$ according to Equation (4).
11. Update $a_{high}$ and $a_{low}$ according to Equation (4).
12. If $b_{low} > b_{high} + 2 \times$ tolerance, then go to Step 5.

5. Scaling on multiple accelerators and cluster

For extremely large datasets that have more than 30 million non-zero elements (e.g. gisette, epsilon-full, and dna in Table 6), the limited processing power, memory bandwidth, and on-chip buffer of single architecture restrict the further performance improvement. Meanwhile, some of the most advanced supercomputers today have more than one co-processors on each node (e.g. Tianhe-2[12]). To further scale up our algorithm and fully utilize heterogeneous platforms, we redesign our SVM algorithm for HPC systems with multiple co-processors (using Intel MICs as an example). Our design is a general approach and can be applied to similar multi-co-processor based platforms.

5.1. Multi MIC cards optimization

**Removing the data transfer conflict.** To reduce the memory requirements and achieve load balance among the MIC cards, we divide the training samples dataset into multiple parts and distribute them evenly to the MIC cards. Step 5 of Algorithm 2 shows that all the MIC cards require to use $X_{high}$ and $X_{low}$ for the update of $f_i$ (Eq. (2)). However, after the data distribution, only one MIC can have $X_{high}$ or $X_{low}$. Therefore, we need to transfer $X_{high}$ and $X_{low}$ to the cards that do not have them. Meanwhile, CPU also needs to use $X_{high}$ and $X_{low}$ to update $a_{high}$ and $a_{low}$ (Step 7 in Algorithm 1). Since $X_{high}$ and $X_{low}$ keep changing at different iterations, the best solution is to put them on CPU at the start of each iteration and then send them to the MIC cards in need. The commonly used approach for transferring the data to multiple co-processors in parallel and launching these co-processors concurrently is to use multi-threading on the host. Each thread handles one co-processor and all the threads share the same copy of the transferring data. However, because of the fierce data competition among different transferring threads, the simultaneous transfer of data suffers from significant data-conlict overheads in our experiments (Fig. 18).

To remove the overheads, we transfer the data to all the MIC cards one by one. Although the transferring is processed serially, we observe a significant performance enhancement. The reason is that the transfer conflict will prevent all the MIC cards from working normally. For the serial transferring, there is at least one MIC card is working with full capability. This can be justified by Figs. 16 and 17, which show that the no-conflict version (serialized transfer) has much better memory bandwidth and cache usages than the conflict version (multi-threading launching). The memory bandwidth and cache usages are very important factors for optimizing the data-intensive applications [46,27]. In a 3-MIC card setup, we achieve 432 $\times$, 18 $\times$, and 29 $\times$ speedup for gisette, dna, and epsilon, respectively through removing the data conflict overheads (Fig. 18).

The reason for the significant discrepancy in speedups is that the size of gisette is only 112 MB while the size of dna and epsilon are more than 2.6 GB. The single iteration time of the optimal one-MIC version is 1.7 and 52.5 ms for gisette and dna respectively. However, the overhead of data conflict itself is about 300 ms for gisette on a single MIC. To further improve the performance and avoid data conflicts, we reproduce the data to several copies and distribute them to different cores. Each involved core only handles the data transfer of one MIC card. In this way, data transfer can be processed in parallel by different cores without data conflict overhead, which achieves 1.9–2.7 $\times$ speedups for the three-MIC version over only No-Conflict scenario (Fig. 18).

**Effects of 3 MIC cards.** For epsilon (2.91 GB) and dna (2.68 GB) datasets, the speedup enjoys nearly linear growth as the number
MIC cards and cores increase (Fig. 19), which indicates our implementation has decent strong scaling. Since the data size of gisette (112 MB) is not large enough, the performance enhancement cannot cover the overheads from 3-MIC scheduling and communication. The linear scaling growth stops at two MIC cards (122 cores). Although epsilon and dna have similar data size, the speedup of one MIC card over host CPUs is $2.5 \times$ for epsilon while it is only $1.1 \times$ for dna. In our experiments, the bandwidth of processing epsilon dataset on one MIC card is 127.2 GB/s while it is only 60.7 GB/s for dna dataset. Our profiling results indicate that the reason for this discrepancy is that we design the fully-vectorization kernel functions (Table 1) in our implementation in order to utilize the vector processing units (VPU) on MIC. The actual parallelism of the kernel function (Kernel($X_i$, $X_j$)) is decided by the dim of $X_i$ and $X_j$. The average dim of epsilon (2000) is $10 \times$ that of dna (200), which will achieve better vectorization improvement by using MIC (512-bit SIMD and independent VPU) over CPU (256-bit SIMD). After efficient optimizations, we manage to improve the memory bandwidth to 161.1 GB/s and 151.2 GB/s on 3 MIC cards for epsilon and dna respectively (Fig. 16).

Power consumption is one of the most important concerns for modern HPC systems. The power consumptions of our system (in Watt) are listed in Table 4. To get the actual power of our implementation, we subtract the system idle power (502 W) from the whole system power. From Table 4, we can observe that the power of the 3-MICs is less than the power of the CPUs for epsilon and dna, which means our 3-MIC implementation is not only time-efficient but also energy-efficient. The major reason is that we employ Xeon Phi Vector Instruction Set [36] to fully utilize the VPUs on MIC. VPUs are highly power efficient for HPC systems. A single operation can encode a great deal of work and does not incur energy costs associated with fetching, decoding, and retiring many instructions [8].

To evaluate the overall feature of our design, the proper performance metric should take both speedup and power into consideration. We define our overall performance in Eq. (10) where 1000 is used to make the values readable. The overall performances and variation tendencies of our three big datasets are listed in Fig. 20. For epsilon, which is the biggest dataset (2.91 GB) among the three, the overall performance nearly achieves a linear growth with the increasing cores. This means both the processing capacity and the efficient energy mechanism have been utilized by our design on MICs. However, for epsilon, which is only 112 MB, the enhanced performance is not enough to cover the contention overheads by using large number of cores.

$$\text{Overall Performance} = \frac{\text{Speedup} \times 1000}{\text{Power}}$$

(10)

### 5.2. Scaling on cluster

SVM is considered as a data-intensive application, which is often communication bound on the large scale clusters. In our situation, the major overhead of original SMO comes from Step 6–8 in Algorithm 1, which corresponds to two parts: (1) a global gather to get all the $f_i$ to the root node; and (2) the updating of alpha (Eq. (3)), $i_{High}$, $i_{Low}$, $b_{High}$, and $b_{Low}$ needs to be processed on a
single node. Suppose the number of training samples is \( n_{\text{samples}} \), then overhead of global gather is \( O(n_{\text{samples}}) \). The overhead of updating \( i_{\text{high}}, i_{\text{low}}, b_{\text{high}}, b_{\text{low}} \) is also \( O(n_{\text{samples}}) \). \( n_{\text{samples}} \) is often extremely large for some real-world datasets. For example, \( n_{\text{samples}} \) of dna dataset is 3,600,000. These overheads can become extremely large as the data size and number of processors increase. In our experiment, Step 6–8 can cost up to 98.7% of the total execution time in original SMO algorithm for processing the dna dataset on 1536 processors.

To remove the overheads above, we redesign the algorithm to make it support multi-level reduction. The number of nodes of our cluster is referred as \( n_{\text{nodes}} \). For Step 7, we first conduct a local reduce on each node to get the \( b_{\text{high}}, b_{\text{low}}, i_{\text{high}}, \) and \( i_{\text{low}}, \forall j \in \{1, 2, \ldots, n_{\text{nodes}}\} \). Then we do a global gather to send all these local values to the root node. On the root node, \( b_{\text{high}}, b_{\text{low}}, i_{\text{high}}, \) and \( i_{\text{low}} \) will be obtained among these local values through global reduce. In this way, we manage to reduce the overheads of global communication from \( O(n_{\text{samples}}) \) to \( O(n_{\text{nodes}}) \). The overheads of single node computation is also reduced from \( O(n_{\text{samples}}) \) to \( O(n_{\text{nodes}}) \). \( n_{\text{samples}} \) is orders of magnitude larger than \( n_{\text{nodes}} \). For example, we use 128 nodes to process the 3,600,000 samples of dna dataset, which means \( n_{\text{samples}} = 28, 125 \times n_{\text{nodes}} \). The results in Table 5 show that we can use the optimized algorithm to improve the strong scaling from 2.29% to 112% and the weak scaling from 1.30% to 74.7% on 1536 processors.

6. Experimental results and analysis

6.1. Experimental setup

The architecture details of the four evaluated architectures in this paper can be found in Table 2. For the purpose of cross-platform performance evaluation, we further optimized the traditional GPUSVM tool and ported it to the new NVIDIA Kepler k20x GPU.

6.2. 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 6, where \( n \) is the number of training samples, \( n_{\text{Dim}} \) 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 (\( \text{Cost} \) and \( \text{Gamma} \)) are also shown in Table 6. Cost represents the regularization constant that balances the generality and accuracy. Gamma is shown in Table 1. In order to make \( n \) and \( n_{\text{Dim}} \) more representative, we reduce the number of training samples for two datasets (epilson and forest) without modifying the contents of the corresponding training samples.

6.3. Accuracy validation for MIC-SVM

We select LIBSVM, which is a widely-used standard SVM library, as baseline to validate the correctness of our implementations. Table 7 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 defined 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 7, we can observe that only the result based on Aeronautics 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 completely 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, which is a normal case in SVM software. It is worth nothing that the training process for the sraa dataset cannot be converged through GPUSVM because it would require 11.6 GB storage for dense format, which is beyond the memory capacity of a single GPU card.

<table>
<thead>
<tr>
<th>Table 5: Strong scaling and weak scaling on Tianhe-1A for dna dataset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes/Cores</td>
</tr>
<tr>
<td>Un-optimized strong scaling Speedup/Efficiency</td>
</tr>
<tr>
<td>Optimized strong scaling Speedup/Efficiency</td>
</tr>
<tr>
<td>Un-optimized weak scaling 10⁴ iterations time (s)/Efficiency</td>
</tr>
<tr>
<td>Optimized weak scaling 10⁴ iterations time (s)/Efficiency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 6: The test datasets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset and reference</td>
</tr>
<tr>
<td>Epsilon [40]</td>
</tr>
<tr>
<td>Epsilon-full [40]</td>
</tr>
<tr>
<td>Giacette [16]</td>
</tr>
<tr>
<td>Forest [9]</td>
</tr>
<tr>
<td>Usps [20]</td>
</tr>
<tr>
<td>Adult [35]</td>
</tr>
<tr>
<td>Sraa [28]</td>
</tr>
<tr>
<td>Dna [40]</td>
</tr>
</tbody>
</table>
6.4. Theoretical analysis for peak performance

As mentioned in Section 4.1, SVM is based on a data-intensive algorithm and its peak performance is limited by the memory bandwidth. Fig. 16 shows that our algorithm achieves a 161 GB/s memory bandwidth, which reaches the peak bandwidth of Intel MIC architecture (Table 2). This indicates that our HPC-SVM can nearly achieve the theoretical peak performance on a system with multiple MICs. Take the single precision as an example, the peak performance of Intel MIC architecture is 2020 GFlops while the RCMA of our algorithm is 1.5 (Section 4.1). According to Eq. (11), our application will need a 1347 GB/s bandwidth to achieve the architectural peak performance, which is much higher than the actual architectural bandwidth. Therefore, we have achieved the theoretical peak performance of SVM on the state-of-the-art x86 (CPUs and MICs) architectures.

\[
\text{Required Bandwidth} = \frac{\text{Architecture Peak Performance}}{\text{Algorithm RCMA}}.
\]  

(11)

6.5. Strong and weak scalings

We use dna dataset to evaluate the scaling of our implementation because it has a large number of samples. For the weak scaling experiment, we put 20,000 samples on each node. From Table 5 we can observe that our optimized algorithm achieve much better strong scaling and weak scaling compared with the state-of-art approach. The reason for the decreasing weak scaling efficiency when increasing number of nodes is that we only have limited data (15 MB) allocated on each node and the performance benefits gained from HPC-SVM cannot totally cover the increasing overheads of communication and single-node computation. This indicates that a even larger input dataset is required to improve the weak scaling efficiency. The definition of strong and weak scaling efficiencies are in Eqs. (12) and (13).

\[
\text{Strong Scaling Efficiency} = \frac{\text{speedup over one node}}{\text{number of nodes}}.
\]

(12)

\[
\text{Weak Scaling Efficiency} = \frac{\text{one node execution time}}{\text{execution time}}.
\]

(13)

6.6. 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 8 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× and 18–47× speedups over the serial LIBSVM through Intel Knights Corner MIC and Ivy Bridge CPUs respectively. Fig. 21 shows the speedups of MIC-SVM and GPUSVM over LIBSVM on single iteration (instead of the total execution time) for various input patterns (shown in Table 6). 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 8 and Fig. 21, we conduct the following analysis to help us understand how to choose suitable implementation and architecture for a specific input data pattern.

6.6.1. MIC-SVM on CPUs vs. MIC-SVM on MIC

MIC is suitable for dense high-dimension datasets. From Fig. 21, we observe that the speedup achieved using MIC-SVM on MIC is about twice (1.8 × and 2.1 ×) that on Ivy Bridge CPUs for the high dimension datasets (Table 6) 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 ×) 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 × 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. Fig. 21 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 4.2), 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 because the dimension (123d × 0.11 = 135d) is too low to fully take advantage of the SIMD instruction (discussed in Section 4.3.2). 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 (256/32) kernel evaluations while each MIC hardware thread corresponds to 16 (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 (240/12) that of CPUs. However, MIC-SVM on CPUs achieves a 7 × speedup (Fig. 21) over that on MIC, which means each serial kernel evaluation on MIC is 140 × 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 2); (2) each instruction of MIC cannot 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 2).

As discussed in Section 4.3.2, 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 (2095d × 0.0024 = 54d). Even though the data parallelism still cannot meet the requirement of the wide MIC SIMD, it does help reduce the performance discrepancy (from 7 × to 3 ×) between MIC and CPUs (Fig. 21). This example clearly proves the importance of the adaptive support for data parallelism.

### 6.6.2. MIC-SVM on MIC vs. GPUSVM on GPUs

In Fig. 21, 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 9 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 instruction 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 6), 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

---

**Table 9**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Gisette</th>
<th>Epsilon</th>
<th>Usps</th>
<th>Forest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension</td>
<td>5000</td>
<td>2000</td>
<td>123</td>
<td>54</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.5</td>
<td>0.9</td>
<td>3.4</td>
<td>8.7</td>
</tr>
</tbody>
</table>
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.

6.7. Comparisons between Fermi and Kepler GPUs

6.7.1. Processing power and memory bandwidth

From the comparison between Kepler GPUs and Fermi GPUs in Fig. 21, we can observe that the biggest performance improvement from Fermi to Kepler are from the forest (2.5 ×) and sraa (2.35 ×) datasets. The forest dataset (2.5 MB) and the sraa dataset (15 MB) are the smallest ones among all the test datasets.

From Fermi to Kepler, there is a 3.8 × (from 1.30 Tflops to 3.95 Tflops) performance improvement for the single precision peak performance. While for the memory bandwidth, there is merely a 1.30 times (from 192 to 250 GB/s) improvement, which results in a three times discrepancy in the RCMB (Eq. (9)). Additionally, there is no size improvement (64 vs. 64 KB) between the high-speed on-chip buffers (L1 cache and shared memory). Hence, as the size of dataset increases, there will be more aggressive performance degradation on Kepler than that on Fermi since the algorithm is memory bound Section 4.1.

6.7.2. The increasing parallelism

Although the memory requirement further increases (from 205 to 685 MB) from epsilon to usps (Fig. 21), the performance gap between Fermi and Kepler is still increasing (from 1.2 × to 1.8 ×). This may appear to be against our analysis above Section 6.7.1. However, several important architectural improvements from Fermi to Kepler [32,33] may account for such improvement:

(1) there is no difference between Fermi C2070 and Kepler 20x for the number of SMs (both are 14). For each SM, however, Kepler has 192 single precision cores and 64 double precision units while Fermi only has 32 cores. Additionally, from Fermi to Kepler, the number of SFU (special function unit) increases from 4 to 32 and the number of LDST (load/store unit) increases from 16 to 32.

(2) from Fermi to Kepler, the number of warps available for scheduling concurrently on each SM increases from 2 to 4. Moreover, each Kepler warp scheduler controls double instruction dispatch units, which means two independent instructions could be dispatched to each warp per cycle (shown in Fig. 22). Combined with the improvement (1), the parallelism of Kepler is four times higher than Fermi.

Combining the experimental results with the analysis on RCMB and parallelism, we conclude that the dataset with large number (high parallelism) of low-dimension (low memory requirement) training samples will be more likely to benefit from the architectural improvements from Fermi to Kepler.

7. 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 [34], points shrinking, caching [23], minimal working set [35], and second order working set selection [14]. Most of these techniques have already been adapted in the widely used LIB-SVM [6], which is designed for serial processing. Others have tried to parallelize SVM on distributed memory systems [15,3,5] 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 CPUs. Almost all of them have been focusing on using GPUs to accelerate the SMO [35] algorithm. Catanzaro [4] first proposed the GPUSVM for binary classification problem and achieved speedup of 9–35 × over LIBSVM. Herrera-Lepez then [18] improved Catanzaro’s work by adding the support for Multiclass classification.

Tsung-Kai Lin [26] 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 it to our implementation with our MIC-SVM. Tyree et al. [43] did an parallel SVM implementation through converting the computation to high-level linear algebra operations, which could make full use of the efficient library like Intel MKL [21]. However, their paper is orthogonal to our work. Our work is to try to get the maximum performance as possible while their objective is to minimize the programming and tuning efforts.

Since the emerging of the Intel MIC architecture, there has been limited research applying MIC to solve data-intensive applications such as sorting [37], data mining [17], and ray tracing [1]. To the best of the author’s knowledge, our proposed MIC-SVM is the first trial 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 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.

Our previous work [47] only address the problems of single node acceleration. In this paper, we design and implement the scalable approaches on multiple co-processors system and clusters, which is highly necessary for processing the large-scale applications. We also do the performance and power consumption analysis, both of which are the concerns of modern HPC systems. Besides, this paper greatly modifies the adaptive heuristic method and adds the latest experimental results, which can justify the effects of our novel method. This paper also supplements the comparison between Nvidia Fermi GPUs and Kepler GPUs, which hopefully can help the reader to get a better understanding of the many-core
architectural development trend and select the most suitable accelerators for their applications.

8. 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 work, 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.

Acknowledgments

The authors would like to thank the reviewers for their valuable comments that lead to the improved quality of the article. The financial support from National Natural Science Foundation of China (Grant Nos. 61303003 and 41374113) and National High-tech R&D (863) Program of China (Grant No. 2013AA01A208) is also gratefully acknowledged. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported by DOE OSCAR Beyond Standard Model Project #62853.

References


Yang You is a Research-Oriented Master Candidate in the Department of Computer Science and Technology at Tsinghua University. His major research interest is Parallel/Distributed Computing. Specifically, his research is focused on designing the high-performance algorithms to optimize the real-world applications such as Massive Graph Traversal (e.g., BFS), Scientific Computing (e.g., Stencils), Machine Learning (e.g., classification), and bioinformatics (e.g., Genome) on the Many-core and Multi-core architectures like CPUs, GPU and Intel Xeon Phi Coprocessor. He is also interested in solving the large-scale problems on the Multi-Node cluster. He is a student member of IEEE.

Haohuan Fu is an associate professor in the Ministry of Education Key Laboratory for Earth System Modeling, and the Center of Earth System Science, at Tsinghua University. His research interests mainly focus on the high-performance computing applications in earth and environmental sciences. Dr. Fu has a Ph.D. in computing from Imperial College London. He is a member of IEEE.

Shuaiwen Leon Song is currently a research staff scientist for PAL (Performance Analysis Lab) at Pacific Northwest National Lab (PNNL). He graduated with a Ph.D. from Computer Science department of Virginia Tech in May 2013. Before he joined PNNL, he was a member of Scalable Performance Lab (SCAPE) directed by Dr. Kirk W. Cameron at Virginia Tech. In the past, he was an intern for Center for Advanced Computing (CASC) at Lawrence Livermore National Lab (LLNL), Performance Analysis Lab (PAL) at Pacific Northwest National Lab (PNNL), and also a R&D intern for the Architecture Research Division at NEC Research American in Princeton, New Jersey. He was a 2011 ISCR scholar and recipient of 2011 Paul E. Torgersen Excellent research award. His research interests lie in several areas of High Performance Computing (HPC).

Amanda Randles is a Lawrence Fellow at Lawrence Livermore National Laboratory. In general, her work focuses on the design of large-scale parallel applications targeting problems in physics. Her research goals are to both investigate fundamental questions related to fluid dynamics as well as extend the multiscale models developed in her thesis to study cancer metastasis. For this collaborative work, she continues as a Visiting Scientist at the Dana–Farber Cancer Institute working in Franziska Michor’s lab.

Darren Kerbyson is a computer scientist at the Department of Energy’s Pacific Northwest National Laboratory. Dr. Kerbyson is a co-investigator on the new “Performance Evaluation and Analysis Consortium (PEAC) End Station” project that was awarded a total of $5 million core hours, divided between the Mira and Intrepid supercomputers at the Argonne Leadership Computing Facility and the Titan supercomputer at the Oak Ridge Leadership Computing Facility.

Andres Marquez is a computer and compiler architect who has worked on the design and development of the German Supercomputers Supremum and MANN, on design studies for the Hybrid Technology and MultiThreaded (HTMT) technology funded by the National Aeronautics and Space Administration (NASA), Defense Advanced Research Projects Agency (DARPA), National Security Agency (NSA), and Jet Propulsion Laboratory (JPL) and on academic multithreaded projects such as the Efficient Architecture for Running Threads (EARTH) and the Compiler Aided Recursive Engine (CARE).

Guangwen Yang is Professor of Computer Science and Technology, Tsinghua University, Beijing, China. He received the Ph.D. degree of the Department of Computer Science, Harbin Institute of Technology University, China in 1996. He is mainly engaged in the research of grid computing and parallel and distributed computing.

Adolfy Hoisie is a Laboratory Fellow and Director of the Center for Advanced Architectures at the Pacific Northwest National Laboratory. Prior PNNL, he was at the Los Alamos National Laboratory, where he served in a variety of scientific and leadership positions, including as the Director of the Center for Advanced Architectures and Usable Supercomputing and the Leader of the Computer Science for High-Performance Computing Group which included the Performance and Architecture Lab.