Runtime Data Layout Scheduling for Machine Learning Dataset

Yang You
Computer Science Division, UC Berkeley, CA, USA
youyang@cs.berkeley.edu

James Demmel
Computer Science Division, UC Berkeley, CA, USA
demmel@cs.berkeley.edu

***Abstract*—Machine Learning (ML) approaches are widely-used classification/regression methods for data mining applications. However, the time-consuming training process greatly limits the efficiency of ML approaches. We use the example of SVM (traditional ML algorithm) and DNN (state-of-the-art ML algorithm) to illustrate the idea in this paper. For SVM, a major performance bottleneck of current tools is that they use a unified data storage format because the data formats can have a significant influence on the complexity of storage and computation, memory bandwidth, and the efficiency of parallel processing. To address the problem above, we study the factors influencing the algorithm's performance and conduct auto-tuning to speed up SVM training. DNN training is even slower than SVM. For example, using a 8-core CPUs to train AlexNet model by CIFAR-10 dataset costs 8.2 hours. CIFAR-10 is only 170 MB, which is not efficient for distributed processing. Moreover, due to the algorithm limitation, only a small batch of data can be processed at each iteration. We focus on finding the right algorithmic parameters and using auto-tuning techniques to make the algorithm run faster. For SVM training, our implementation achieves $1.7-16.3\times$ speedup ($6.8\times$ on average) against the non-adaptive case (using the worst data format) for various datasets. For DNN training on CIFAR-10 dataset, we reduce the time from 8.2 hours to only roughly 1 minute. We use the benchmark of dollars per speedup to help the users to select the right deep learning hardware.**

***Keywords*-parallel auto-tuning; machine learning**

I. Introduction

Machine Learning (ML) approaches are widely-used classification/regression methods for data mining applications. In recent years, ML approaches were adapted to the field of High Performance Computing for power/performance prediction, auto-tuning, and runtime scheduling. However, the time-consuming training process greatly limits the efficiency of ML approaches. For example, it takes days or weeks to train a Deep Neural Network (DNN) or a Support Vector Machines (SVM) model. 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 many-core architectures, whose massive parallelism and complex memory hierarchies form a barrier to efficient parallel ML design due to the fact that a typical ML application often is a data-intensive, memory-bound, and irregular-access application. While previous ML designers could depend on the ready-made performance improvement that comes from a faster clock rate ([1], [2], [3]), now they have to face the challenges of scaling the performance over tens or even hundreds of cores within a single node. For example, the latest Intel Knights Landing architecture has 72 cores on a single chip. We use the example of SVM (traditional ML algorithm) and DNN (state-of-the-art ML algorithm) to illustrate the ideas in this paper.

Previous SVM tools only support one type of data format. For example, LIBSVM (state-of-the-art implementation on CPUs [4]) employs the CSR (Compressed Sparse Row) format for all datasets. GPUSVM (state-of-the-art implementation on GPUs [5]) uses the DEN (Dense) format for all the datasets to achieve high efficiency for parallel processing. However, using a unified data storage format could be a major bottleneck for performance enhancement because the data formats can have a significant influence on the complexity of storage and computation, memory bandwidth, and the efficiency of parallel processing. Figure 1 shows the performance comparison among different data formats processed by SVM. It clearly shows that the most suitable formats for different datasets vary significantly. To address the problem above, we study the factors influencing the algorithm's performance and conduct auto-tuning to speed up SVM training.

DNN often performs very well for the complicated applications like computer vision. However, the training part of DNN is very slow. For example, using a 8-core CPUs to train AlexNet model by CIFAR-10 dataset [6] costs 8.2 hours. CIFAR-10 is only 170 MB, which is not efficient for distributed processing. Moreover, due to the algorithm limitation, only a small batch of data can be processed at each iteration. The algorithm requires many iterations. We focus on finding the right algorithmic parameters and using auto-tuning techniques to make the algorithm run faster.

Our experiments show that our SVM implementation can achieve $1.7 - 16.3\times$ speedup ($6.8\times$ on average) against the non-adaptive case (using the worst data format) for various real-world social and scientific datasets. If we use the parallel LIBSVM (state-of-the-art SVM software on CPUs using CSR format) as the baseline, the speedups are 1.2 - 16.5$\times$ (4 $\times$ on average). If we use our own CSR implementation as the baseline, the average speedup of our version over fixed-CSR version is $1.3\times$. For DNN training on CIFAR-10 dataset, we successfully reduce the time from 8.2 hours to only roughly 1 minute. Our auto-tuning strategy

can be used in similar applications. We use the benchmark of dollars per speedup to help the users to select the right deep learning hardware.

II. Background

A. Traditional Machine Learning

Machine Learning approaches usually include two kinds of applications: classification and regression. One of the state-of-the-art traditional ML methods is Support Vector Machines (SVM) [7]. We use SVM to illustrate the computational bottleneck of traditional ML approaches.

The ML algorithm generally includes two parts: (1) Training and (2) Prediction. In the training part, we use the optimization algorithms to generate a model M based on the observed data (the information we know). We refer to the observed data as the training dataset. The training dataset includes a $n \times d$ matrix X and a n dimensional vector y. n is the number of observations or the number of samples. d is the number of features. The label of X_i (i-th row of X) is y_i (i-th element of y). If the training dataset has k classes, i.e. the samples can be classified into k categories, then $y_i \in \{1, 2, ..., k\}$. A concrete example is that we can use 100 people's blood type, weight and height information to classify them into two parts ($n = 100$, $d = 3$, and $k = 2$). The data structure of the regression problem is identical to that of the classification problem. The only difference is that $y_i \in R$ for the regression problem. Because the training part costs much more time than the prediction part, previous work focused on speeding up the training part ([5], [8], [9], [10], [11], [12]). Like them, this paper is also focused on accelerating the training part.

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 trained in parallel once the binary-class SVMs are available. SVM training can be presented as a linear-constraint convex Quadratic Programming (QP) problem (Equations (1) and (2)), where C represents the regularization constant that balances the generality and accuracy, α_i is the Lagrange multiplier, and $K_{i,j}$ denotes the Kernel value of X_i and X_j (Table I). α_i is related to training sample X_i. C can be set by users and each α_i is related to a specific training sample X_i. The standard Kernel functions in SVM are shown in Table I. The objective of SVM training part is to get the model α. Then we use α and the training dataset to predict the label for a given sample from the test dataset.

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

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

Table I
Standard Kernel Functions

Linear	$Kernel(X_i, X_j) = {X_i}^T X_j$
Polynomial	$Kernel(X_i, X_j) = (a{X_i}^T X_j + r)^d$
Gaussian	$Kernel(X_i, X_j) = exp\{-\gamma \|\|X_i - X_j\|\|^2\}$
Sigmoid	$Kernel(X_i, X_j) = tanh(a{X_i}^T X_j + r)$

B. Deep Learning (DL)

Deep Learning means Deep Neural Networks (DNN) training and prediction. Like traditional ML, the training part of DL is to use the data matrix X and label vector y to refine the model or weight W. At each step, the algorithm randomly picks B rows from X and computes the weight's gradient ΔW based on these data. Then the algorithm updates the weight by $W = W - \eta \Delta W$. η is the step size (or learning rate) and B is the batch size. The algorithm repeats this step iteratively until we get a good model. Then we use W to predict the labels $\hat{y}$ of unknown data $\hat{X}$. If the algorithm finishes n/B iterations, we call this as 1 epoch. 1 epoch means the algorithm touches the whole training dataset once. The information of DNN model is stored in W.

III. Memory-Efficient SVM

For machine learning dataset (n-by-d matrix), n is usually much larger than d because the number of observations should be larger than the number of features in each observation. Because of this, the major bottleneck of SVM is the n-by-n kernel matrix. For example, even on a relative small dataset (357 megabytes for a 520,000-by-90 matrix [13]), SVM needs to form a 2 terabytes dense kernel matrix. Also, solving the QP problem by direct method requires $\Theta(n^3)$ floating point operations, which is even more expensive. Due to these concerns, people designed decomposition and iterative methods to approach the SVM Kernel.

A. SMO and its Bottleneck

The idea of decomposition method [1] is to divide the big QP problem into some small QP problems. Then the algorithm solves these small QP problems one-by-one. Solving each small sub-QP problem does not require a huge amount of memory. The extreme case is to divide the original problem into $\Theta(n)$ steps. The algorithm only solves a 2-variable problem at each step. This algorithm is Sequential Minimal Optimization (SMO) [3]. In fact, SMO can get the analytical solution (rather than the numerical solution) of the 2-variable problem. A brief outline of the SMO algorithm is presented in Algorithm 1, which refers to equations (3) through (6). We use $high$ and low to denote the indices of two selected points each step. X_{high} and X_{low} are two rows of data matrix X. Then SMO needs compute two rows of the kernel matrix (K_{high} and K_{low}). The algoritm gets K_{high} by conducting a sparse-matrix sparse-vector

multiplication (SMSV, not SpMV) between X and X_{high}. Then the algorithm gets X_{low} in the same way. Thus, the bottlenecks of each SMO step are two SMSVs.

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

$$\hat{f}_i = f_i + \Delta\alpha_{high} y_{high} K_{high,i} + \Delta\alpha_{low} y_{low} K_{low,i} \tag{4}$$

$$\Delta\alpha_{low} = \frac{y_{low}(b_{high} - b_{low})}{K_{high,high} + K_{low,low} - 2K_{high,low}} \tag{5}$$

$$\Delta\alpha_{high} = -y_{low} y_{high} \Delta\alpha_{low} \tag{6}$$

Algorithm 1 The Original SMO Algorithm

1: Input the samples X_i and labels y_i, $\forall i \in \{1, 2, ..., n\}$.
2: Initialize, $\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1, 2, ..., n\}$.
3: Initialize, $b_{high} = -1$, $high = \min\{i : y_i = -1\}$, $b_{low} = 1$, $low = \max\{i : y_i = 1\}$.
4: Update α_{high} and α_{low} according to Equations (5) and (6).
5: Update f_i according to Equation (4), $\forall i \in \{1, 2, ..., n\}$
6: $I_{high} = \{i : 0 < \alpha_i < C \cup y_i > 0, \alpha_i = 0 \cup y_i < 0, \alpha_i = C\}$
7: $I_{low} = \{i : 0 < \alpha_i < C \cup y_i > 0, \alpha_i = C \cup y_i < 0, \alpha_i = 0\}$
8: $high = \arg\min\{f_i : i \in I_{high}\}$
9: $low = \arg\max\{f_i : i \in I_{low}\}$
10: $b_{high} = \min\{f_i : i \in I_{high}\}, b_{low} = \max\{f_i : i \in I_{low}\}$
11: Update α_{high} and α_{low} according to Equations (5) and (6).
12: If $b_{low} > b_{high} + 2 \times tolerance$, then go to Step 5.

In our experiments, we observe that the data layout of matrix X significantly affects the performance of SMO. For sparse matrix, the data layout is dependent on the data format. In general, there are five basic matrix storage formats that can be used to process the training samples of ML: DEN (Dense), CSR (Compressed Sparse Row) [14], ELL (ELLPACK/ITPACK) [15], COO (Coordinate) [16], and DIA (Diagonal) [17]. Most of the other storage formats can be derived from these basic formats. For instance, the CSC (compressed sparse column) is similar to the CSR format. The only difference is that the columns are used instead of the rows. On the other hand, the block variants like BCSR (Block Compressed Sparse Row) are often used when there are many dense sub-blocks in a sparse matrix. Figure 1 shows the performance comparison among different data formats used by SMO algorithm. Let M and N be the # rows and # columns of the matrix. To store the ML datasets into different data formats, we have $M = n$ and $N = d$. In many real-world scientific computing applications, the matrices are still often sparse. For the extremely sparse datasets, DEN has to store $M \cdot N$ elements while COO and CSR only need to store $O(nnz)$ elements where nnz is the number of nonzeros. DIA needs to store at least max(nnz,min(M,N)) elements since there is at least one non-zero element that requires one diagonal. On the other hand, there are many dense big datasets in the machine-learning field. For these datasets, DEN is the most efficient format. For CSR, both

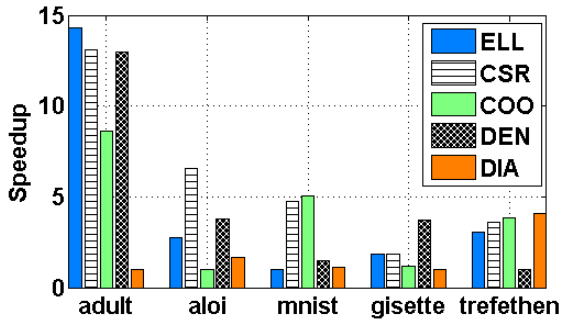

Figure 1. adult [3], aloi [18], minist [19], gisette [20], and trefethen [21] are five read-world datasets. ELL, CSR, COO, DEN, DIA are five widely-used data formats for sparse matrices. This figure shows the performance comparison among different data formats processed by SVM (normalized to the slowest format for each dataset). We can observe the best format (the highest speedup) and the worst format (the lowest speedup) for different dataset are variable. Trefethen is originally from a matrix dataset, we convert it to an SVM test dataset.

the $data$ array and the $indices$ array have $M \cdot N$ elements, and ptr has M elements. In total, CSR needs to store $O(2 \cdot M \cdot N + M)$ elements for (nearly) dense matrices. Similarly, COO stores $O(3 \cdot M \cdot N)$ elements and ELL stores $O(2 \cdot M \cdot N)$ elements for dense datasets. For DIA, since there are $O(M+N-1)$ diagonals, the $data$ array has to store $O(min(M, N) \cdot (M + N - 1))$ elements and the $offsets$ array needs to store $O(M + N - 1)$ elements. DIA in total needs to store $(min(M, N)+1) \cdot (M + N - 1)$. We sum up the maximum and minimum storage of each format in Table II. The complexity of computation in SVM (two SMSVs) is proportional to the complexity of storage.

B. Selecting the right data format

Existing SVM tools (e.g. LIBSVM and GPUSVM) use fixed data formats for all datasets. Nevertheless, Table III shows that the most suitable formats for different datasets vary significantly and the speedup of using the best format over the the worst format can be $3.73-14.3\times$ for processing the same dataset.

Additionally, the performance of the parallel SVM is also bounded by memory bandwidth. Our profiling results show that the bandwidth also varies when using different formats to process the same dataset. For instance, the bandwidth of processing gisette dataset [20] is 25.3 GB/s, 63.9 GB/s, 63.5 GB/s, 53.1 GB/s, and 37.7 GB/s for ELL, CSR, COO, DEN, and DIA, respectively, on Ivy Bridge CPUs. Together with Equation (7), the bandwidth results further confirm and explain the performance gaps between various storage formats shown in Table III. Therefore, current state-of-art tools such as LIBSVM and GPUSVM can only perform well for a limited number of real-world datasets.

$$Execution\_Time \gtrapprox \frac{Transferred\_Memory}{Memory\_Bandwidth} \tag{7}$$

Table II
STORAGE SPACE COMPARISON FOR VARIOUS FORMATS

Format	DEN	CSR	COO	ELL	DIA
Min	$M \cdot N$	$O(M+2)$	$O(1)$	$O(2M)$	$O(M+1)$
Max	$M \cdot N$	$2M \cdot N + M$	$3M \cdot N$	$2M \cdot N$	$O((min(M,N)+1) \cdot (M+N-1))$

Table III
PERFORMANCE COMPARISON AMONG DIFFERENT FORMATS

Dataset	ELL	CSR	COO	DEN	DIA
adult [3]	**14**×	13×	8.6×	13×	1.0
aloi [18]	2.8×	**6.6**×	1.0	3.8×	1.7×
mnist [19]	1.0	4.8×	**5.1**×	1.5×	1.1×
gisette [20]	1.9×	1.9×	1.2×	**3.7**×	1.0
trefethen [21]	3.1×	3.6×	3.9×	1.0	**4.1**×

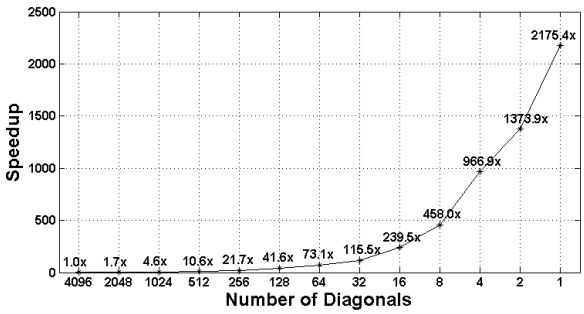

Figure 2. The matrices that have the same M, N, and nnz but different number of diagonals. They are stored in DIA format. The baseline of the speedup is the case with 4096 diagonals (the worst case). The experiments are based on Ivy Bridge CPUs.

To select the right data format for a ML dataset, we need to use the right parameters of the data matrix to tune the code. The two basic parameters to represent a dataset are M (the number of samples) and N (the number of features). The data structures ELL, CSR and COO do not explicitly depend on the number of columns N. However, N affects the performance of DEN significantly since the amount of computation and storage of DEN is growing as N is increasing. Another basic feature is nnz (# non-zero elements) since the value of nnz can lead to the discrepancy of storage space for different formats (Table II). We also add $density$ (the ratio of the number of the non-zero elements to the number of all the elements) since it is positively correlated with the performance of DEN format.

For DIA format, the number of diagonals has a significant influence on the performance of the matrices that have the same M, N, and nnz. In our experiment, we generate a series of matrices that have the same M, N, and nnz ($M = 4096$, $N = 4096$, and $nnz = 4096$) but different number of diagonals $(2, 4, 8, ..., 2048, 4096)$. We use DIA format to store and process all these matrices. Fig. 2 shows that the more diagonals a matrix has, the worse its performance will be. The major reason is the discrepancy in the number of elements per diagonal. For example, each diagonal of the one-diagonal matrix has 4096 elements while each diagonal of the 4096-diagonal matrix only has one element. Each of these diagonals will be padded with 4095 zeros, which increases unnecessary storage and computations. Therefore, we add $ndig$ (# diagonals) and $dnnz$ (# non-zero elements per diagonal) to our influencing-parameter space. $ndig$ and $dnnz$ can be used to evaluate the fitness of DIA format.

We refer to the number of non-zero elements in the i-th ($i \in 1, 2, 3, ..., M$) row as dim_i. For ELL format, the performance is closely related to the maximum dim_i, which is referred to as $mdim$ (Table IV). To illustrate this, we make a series of matrices that have the same M, N, and nnz ($M = 4096$, $N = 4096$, and $nnz = 8192$) but different $mdim$ $(1, 2, 4, ..., 2048, 4096)$. We use ELL format to store and process all these matrices. Fig. 3 shows that the higher $mdim$, the worse its performance will be. We refer to the matrix with $mdim = 2$ as mat_2 and the matrix with $mdim = 4096$ as mat_{4096}. The major reason for the performance difference between mat_2 and mat_{4096} is that the size of mat_2 is 4096×2 while the size of mat_{4096} is 4096×4096. Each row of mat_{4096} will be padded with 4095 zeros, which increases unnecessary storage and computations. Therefore, we add $mdim$ and $adim$ (# non-zero elements per row) to our influencing-parameter space. From Fig. 3 we can also observe that the performance is decreasing as $vdim$ (the variation of the number of non-zero elements in each row) increases. This further confirms that another reason for the performance difference between mat_2 and mat_{4096} is that the distribution of non-zero elements in mat_2 is much more balanced than that of mat_{4096}. Therefore, we add $vdim$ to our influencing-parameter space, shown in Table IV. Finally, we use $mdim$, $adim$ and $vdim$ to evaluate the fitness of ELL format.

COO is similar to CSR only with the difference that COO has to store the row indices for all the non-zero elements. When dim changes significantly among different rows, it could potentially have negative effects on the performance of CSR format on certain architectures such as Intel MIC due to the inefficient usage of the fixed-width SIMD. However, this has little influence on the performance of COO format because all the non-zero elements in $data$ array can be processed in parallel. Fig. 4 shows that the speedup of COO over CSR is increasing as $vdim$ is growing, which is in line with our analysis. Since $vdim$ can reflect whether dim

Table IV
INFLUENCING PARAMETERS OF THE DATA MATRIX, + MEANS POSITIVE CORRELATION BETWEEN THE VALUE AND SOFTWARE EFFICIENCY, − MEANS NEGATIVE CORRELATION BETWEEN THE VALUE AND SOFTWARE EFFICIENCY, ± MEANS POSITIVE OR NEGATIVE CORRELATION BETWEEN THE VALUE AND SOFTWARE EFFICIENCY, AND × IS UNCORRELATED

Parameter	Description	Formula	ELL	CSR	COO	DEN	DIA
M	number of rows	$number\ of\ samples$	±	±	±	±	±
N	number of columns	$maximum\ feature\ index\ of\ all\ samples$	×	×	×	−	×
nnz	number of non-zero elements	$\sum_{i=1}^{M} dim\_i$	±	±	±	+	±
$ndig$	number of diagonals	$number\ of\ diagonals$	×	×	×	×	−
$dnnz$	number of nnz per diagonal	$nnz/ndig$	×	×	×	+	+
$mdim$	maximum nnz in a row	$MAX(dim\_i),\ i \in 1, 2, 3, ..., M$	−	×	×	×	×
$adim$	average number of nnz in a row	nnz/M	+	×	×	+	×
$vdim$	variance of dim	$\sum_{i=1}^{M}(dim_i - adim)^2/M$	−	−	+	×	×
$density$	the ratio of nnz to all the elements	$nnz/(M * n)$	±	±	±	+	±

Table V
EVALUATED REAL-WORLD DATASETS

Dataset	Application	M	N	nnz	$ndig$	$dnnz$	$mdim$	$adim$	$vdim$	$density$
adult	economy	2,265	119	31,404	2,347	13.38	14	13.87	0.059	0.119
breast cancer	clinical	38	7,129	270,902	7,166	37.80	7,129	7,129	0.0	1.000
aloi	vision	1,000	128	32,142	1,125	28.57	74	32.14	85.22	0.251
gisette	selection	6,000	5,000	30,000,000	10,999	2,728	5,000	5,000	0.0	1.000
mnist	recognition	450	772	66,825	1,050	63.64	291	148.5	1,594	0.192
sector	industry	1,500	55,188	238,790	33,770	7.07	1,819	159.19	17,634	0.003
epsilon	AI	390,000	2,000	780,000,000	391,999	1,990	2,000	2,000	0.0	1.000
leukemia	biology	38	7,129	270,902	7,166	37.8	7,129	7,129	0.0	1.000
connect-4	game	1,800	125	75,600	1,922	39.33	42	42	0.0	0.336
trefethen	numerical	2,000	2,000	21,953	12	1,829	12	10.98	1.25	0.006
dna	genomics	3,600,000	200	720,000,000	3,600,199	200.0	200	200	0.0	1.000

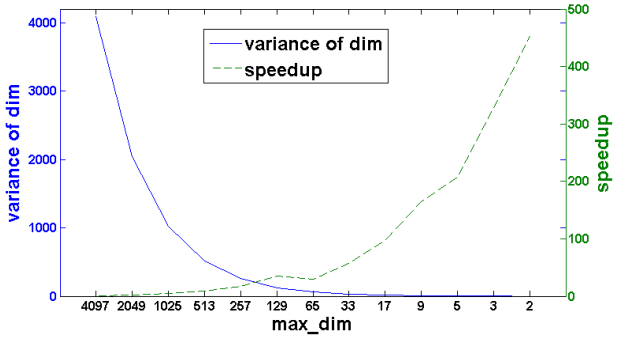

Figure 3. The matrices that have the same M, N, and nnz but different $mdim$. All of them are stored in ELL format. The baseline of the speedups is the worst case. The experiments are based on Ivy Bridge CPUs.

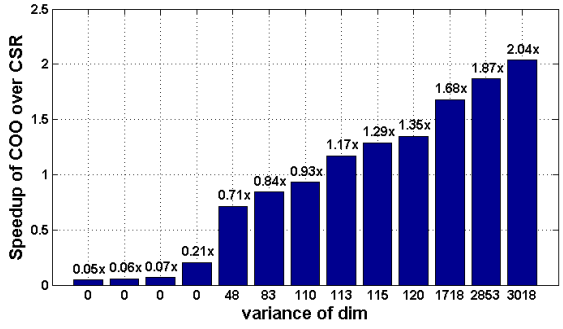

Figure 4. The speedups of COO over CSR for the datasets with different $vdim$. The experiments are based on Ivy Bridge CPUs.

changes significantly among different rows, we select it as the influencing parameter to help the system make the choice between CSR and COO. In total, we extract nine parameters from each given dataset, which are detailed in Table IV.

These parameters helped us to select the right data format for each data matrix. The average speedup in Table VI is the average speedup of using our selection over the other four data formats. The maximum speedup is the speedup of using our selection over the worst data format. Compared with the traditional method, our decision system achieves $1.7-16.2\times$ speedup (average: $6.8\times$).

IV. DEEP LEARNING

For speeding up DNN training, our ultimate objective is to get the target accuracy in a shorter time. In this paper, our target systems include a 8-core CPU, a 68-core Intel KNL, a 32-core Intel Haswell CPU, a P100 GPU and a Nvidia GTX station. Our target application is to get 0.8 testing accuracy for CIFAR-10 dataset [6], which is a standard benchmark of

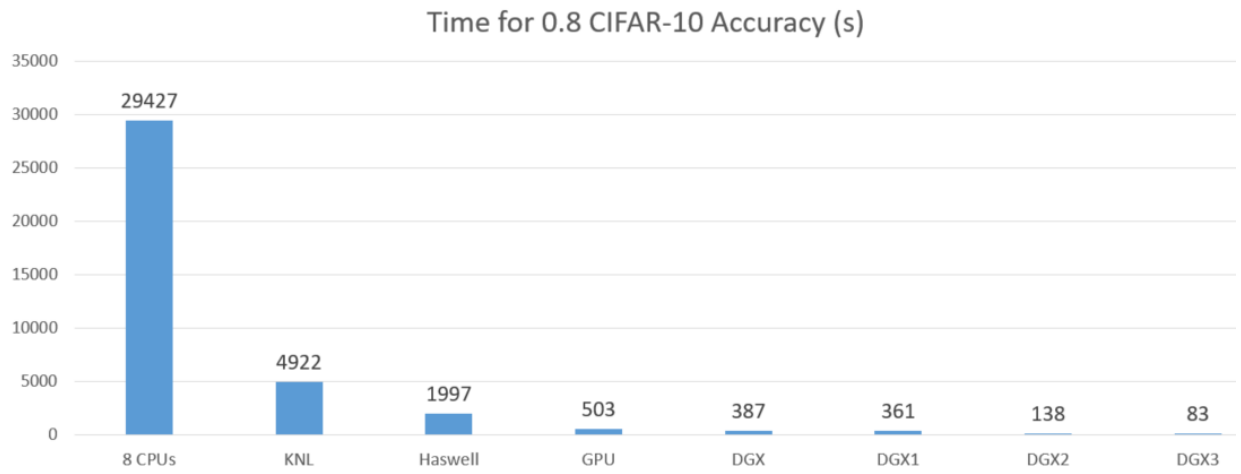

Figure 5. The figure shows the time for 0.8 CIFAR-10 accuracy by different methods. 8 CPUs means Intel Caffe on 8-core CPU, KNL means Intel Caffe on KNL, Haswell means Intel Caffe on Haswell, GPU means NVIDIA Caffe on 1 Tesla P100 GPU, DGX means NVIDIA Caffe on DGX station, DGX1 means our version with tuned batch size, DGX2 means our version with tuned batch size and learning rate, DGX3 means our version with tuned batch size, learning rate, and momentum. Detailed information can be found in Table VII.

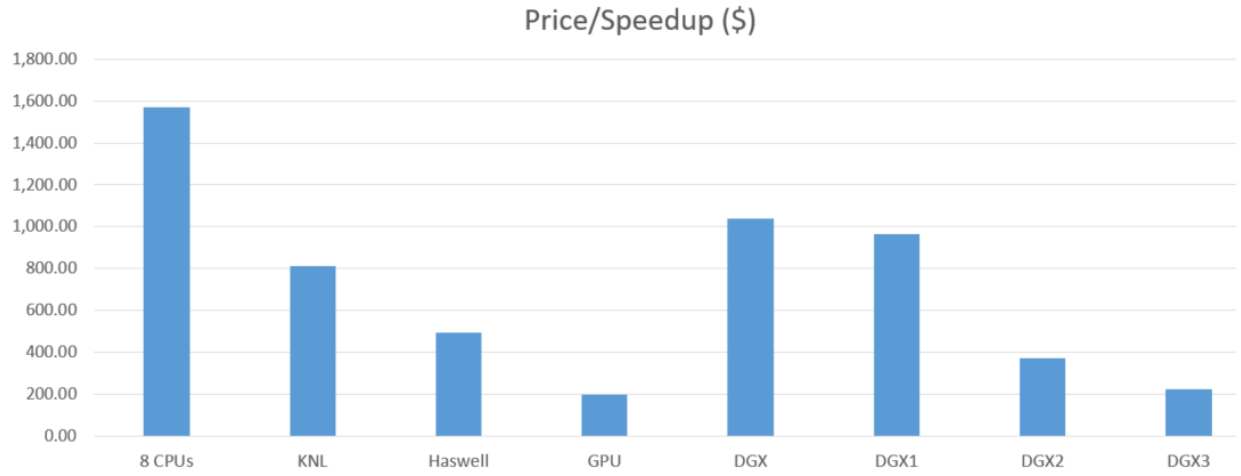

Figure 6. The figure shows the price per speedup for 0.8 CIFAR-10 accuracy by different methods. 8 CPUs means Intel Caffe on 8-core CPU, KNL means Intel Caffe on KNL, Haswell means Intel Caffe on Haswell, GPU means NVIDIA Caffe on 1 Tesla P100 GPU, DGX means NVIDIA Caffe on DGX station, DGX1 means our version with tuned batch size, DGX2 means our version with tuned batch size and learning rate, DGX3 means our version with tuned batch size, learning rate, and momentum. 8 CPUs is the slowest case, which is the baseline and 1.0× speedup. Detailed information can be found in Table VII.

DL research. Our baseline is Caffe's cifar10_full model [22]. The CIFAR-10 dataset includes 60,000 32×32×3 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. Each image is a sample. Obviously, the initial point (random guess) for CIFAR-10 testing accuracy is 1/10. The accuracy becomes 8/10 after the algorithm finishes.

A. *Challenges of Speeding up CIFAR-10 Training*

For speeding up deep learning applications, one bottleneck is that the datasets and weights are often not large enough to use distributed systems. For example, CIFAR-10 is only a 162 MB dataset. However, the training part is very slow because the algorithm has to pass a small batch of data through the deep neural networks for many iterations. Moreover, it is not helpful to load a lot of data to memory at each iteration because using large batch may slow down the algorithm's convergence rate [23]. Our experimental results show that using Caffe [22] (state-of-the-art DL framework) to process CIFAR-10 dataset on a 8-core CPUs (Intel Xeon E5-1660 v4 @ 3.20GHz) for 0.8 testing accuracy will cost 8.2 hours and 120 epochs.

Table VI
EFFECTS OF ADAPTIVE SYSTEM

Dataset	Worst	Selection	Average & Max Speedup
adult	DIA	ELL	3.8× & 14.3×
breast cancer	ELL	CSR	16.2× & 35.7×
aloi	COO	CSR	3.1× & 6.6×
gisette	DIA	DEN	2.4× & 3.7×
mnist	ELL	COO	3.0× & 5.1×
sector	DEN	COO	14.3× & 39.6×
leukemia	ELL	DEN	13.3× & 29.0×
connect-4	COO	DEN	3.3× & 6.4×
trefethen	DEN	DIA	1.7× & 4.1×

B. Choose the Right Hardware

One reason why CIFAR-10 training is so slow is that the computational power of a 8-core CPU is limited. Thus, we need to choose the right architecture to speed up it. Our first choice is the Intel Knights Landing (KNL) archiecture, which is the 2nd generation Intel Xeon Phi platform. KNL has been used in some High Performance Computing data centers since it was released recently. For example, Lawrence Berkeley National Laboratory has a cluster with 9,304 KNLs, and Texas Advanced Computing Center has a cluster with 4,200 KNLs. Our KNL platform is Intel Xeon Phi Processor 7250 processor with 68 cores per node @ 1.4 GHz. The measured peak double-precision peak performance is 3 Tflops.

The major distinct features of KNL that can benefit deep learning applications include the following: **(1) Self-hosted Platform.** The traditional accelerators (e.g. FPGA and GPUs) rely on CPU for control and I/O management. For some machine learning applications, the transfer path like PCIE may become a bottleneck at runtime because the memory on accelerator is limited (e.g. 12 GB GDDR5 on Nvidia K80 GPU). KNL does not need a CPU host. It is self-hosted by an operating system like CentOS 7. **(2) Better Memory.** KNL's measured bandwidth is much higher than that of a 24-core Haswell CPU (450 GB/s vs 100 GB/s). KNL's 384 GB DDR memory size is large enough to handle a typical deep learning dataset. Moreover, KNL is equipped with Multi-Channel DRAM (MCDRAM). MCDRAM's measured bandwidth is 475 GB/s. MCDRAM has three modes: a) Cache Mode. KNL uses it as the last level cache; b) Flat Mode. KNL treats it as the regular DDR; c) Hybrid Mode. Part of it is used as cache, the other is used as the regular DDR memory. **(3) Configurable NUMA.** The basic idea is that the users can partition the on-chip processors and cache into different groups for better memory efficiency and less communication overhead. This maybe useful for some complicated memory-access applications like DNN training.

We then use Intel Haswell CPUs to speed up DNN training. Our Haswell platform is a double-socket Intel Xeon E5-2698 v3 @ 2.3 GHz. The measured peak double-precision peak performance is 1.2 Tflops. Since KNL has a much higher peak floating-point performance than Haswell, the readers maybe surprised to see that KNL runs much slower than Haswell (Figure 7). We tune the code based on Intel's Caffe implementation and report the best performance. We do not claim this is the best implementation on Haswell or KNL. The setting of Haswell is to use 32 threads, OMP_PLACES = threads, and OMP_PROC_BIND = spread. The setting of KNL is to use Cache Mode for MCDRAM, use Quad partition for NUMA, use 68 threads, OMP_PLACES = threads, and OMP_PROC_BIND = spread. The BLAS we use is Intel MKL 2017.

We also use GPUs to process CIFAR-10. Our GPU code is based on NVIDIA Caffe. We observe that Tesla P100 GPU achieves 4× speedup over Haswell CPUs. To furthermore explore the opportunity of improving training speed, we use multiple GPUs. Our parallel strategy is divide-and-conquer for the data and replication for the weights. Let us assume we have P workers. At each iteration, we partition a batch of B samples and each worker gets B/P samples. Each worker gets one copy of the weights W. Different workers will generate different gradients for W because they have different data. Let us refer to ΔW_i as the gradients of W that computed by i-th worker. After a global sum reduce operation, each worker will get $\sum_1^P \Delta W_i$. Then each worker can update their local weights by $W = W - \eta \sum_1^P \Delta W_i / P$.

Our target system is a NVIDIA DGX station, which includes one Intel Xeon E5-1660 v4 @ 3.20GHz CPUs and 4 Tesla P100 GPUs. We use NVIDIA NCCL to connect different GPUs and CUDNN to conduct the neural networks computations. The DGX station is a personal AI supercomputer. However, the straightforward porting from one P100 GPU to one DGX station only brings 1.3× speedup. We expect to at least achieve roughly 4× speedup. Thus, we need to optimize the code to make full use of DGX station's computational power.

C. Tune the Batch Size

There is a tradeoff for tuning the batch size. On one hand, a larger batch size means the BLAS functions can process a larger matrix. Since the computational kernels of deep learning are mainly matrix-matrix multiply and FFT, a larger matrix often can improve the processors' throughput. On the other hand, a large batch size may lead to a sharp optimization problem, which requires more epochs to get the target accuracy [23]. This means that the computational cost per iteration increases at the speed of $\Theta(B)$ while number of iterations (convergence rate) decreases at the speed lower than $\Theta(B)$. To a get a high performance, we have to tune the batch size. After our comprehensive tuning, we observe $B = 512$ is the best choice for CIFAR-10 training on a DGX station. Our tuning space is {64, 100, 128, 256, 512, 1024, 2048, 4096, 8192}.

Table VII
TIME AND SPEEDUP FOR GETTING 0.8 CIFAR-10 ACCURACY. B: BATCH SIZE, η: LEARNING ATE, μ: MOMENTUM.

Methods	B	η	μ	Iterations	Epochs	Time (s)	Price ($)	Speedup	Price/Speedup ($)
Intel Caffe on 8-core CPUs	100	0.001	0.90	60,000	120	29427	1,571	1×	1,571
Intel Caffe on KNL	100	0.001	0.90	60,000	120	4922	4,876	6×	813
Intel Caffe on Haswell	100	0.001	0.90	60,000	120	1997	7,400	15×	493
Nvidia Caffe on Tesla P100 GPU	100	0.001	0.90	60,000	120	503	11,571	59×	196
Nvidia Caffe on DGX station	100	0.001	0.90	60,000	120	387	79,000	76×	1,039
Tune B on DGX station	512	0.001	0.90	30,000	387	361	79,000	82×	963
Tune η on DGX station	512	0.003	0.90	12,000	123	138	79,000	213×	371
Tune M on DGX station	512	0.003	0.95	7,000	72	83	79,000	355×	223

D. Tune the Learning Rate

A large learning rate may help to speed up the algorithm to converge to global minimum. However, due to the non-convex property of deep neural networks, a large learning rate may easily make the algorithm miss the global minimum and get into local minimum. Different batch sizes generally have different optimal learning rate. We tune the learning rate based on the batch size and find the optimal learning rate for $B = 512$ is 0.03. Our tuning space is $\{0.001, 0.002, 0.003, ..., 0.015, 0.016\}$. We achieve a 2.6× speedup by tuning the learning rate.

E. Tune the Momentum

Due to complicated optimization path of non-convex applications (e.g. DL), the algorithm often falls into the local minimum. The momentum technique [24] often can help the algorithm to get out of the local minimum and achieve the global minimum faster. The updating rule of momentum technique is defined in Equations (8) and (9). μ is the momentum parameter. V_t is the momentum vector at iteration t and V_t has the same dimension with W_t. The updating rule becomes the original version if $\mu = 0$. Based on our exprience, μ should be set close to 1 because we want the algorithm to have a good short-term memory to reschedule the optimization path. Our tuning space is $\{0.90, 0.91, 0.92, ..., 0.99\}$. After tuning the momentum, we get an additional 1.7× speedup.

$$V_{t+1} = \mu V_t - \alpha \Delta W_t \tag{8}$$

$$W_{t+1} = W_t + V_{t+1} \tag{9}$$

V. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experimental Setup

The experimental platforms of deep learning are described in Section IV-B. The platform for SVM has one 24-core Intel Ivy Bridge CPU and three 61-core Intel Xeon Phi Knights Corner coprocessors (Intel MIC). We use OpenMP to provide multi-threading. The vectorization of our implementation is based on Xeon Phi Vector Instruction Set and Intel Cilk array notation [25]. The 9 test datasets are listed in Table V.

B. SVM Performance

Comparison with State-of-the-Art SVM Implementation. LIBSVM [26] is the state-of-the-art SVM implementation on CPUs and the most widely used SVM software. To provide a fair comparison, we run parallel LIBSVM (state-of-the-art SVM software on CPUs using CSR format) on our Ivy Bridge CPUs (without using Xeon Phi coprocessor). We achieve $1.2 - 16.5\times$ speedups ($4\times$ on average) over parallel LIBSVM for a series of real-world datasets (Fig. 7). If we use our own fixed-CSR version as the baseline, the average speedup of adaptive system over fixed-CSR is $1.3\times$ because our CSR implementation is more efficient than LIBSVM's CSR implementation.

C. Deep Learning Results

After efficient optimization, we manage to reduce the time of CIFAR-10 training from 8.2 hours (8-core CPU) to roughly 1 minute (DGX station). We achieve a $355\times$ speedup. On the other hand, this comparison maybe unfair because the 8-core CPU is much cheaper than the DGX station. To give a fair comparison, we define the comparison benchmark as price (U.S. Dollars) per speedup. A lower value means a higher efficiency. We make the overall comparisons among all the methods. We observe that the Tesla P100 GPU is the most efficient platform and the 8-core CPU is the least efficient platform (Table VII and Figure 6).

VI. RELATED WORK

In terms of runtime scheduling, there is some related work on auto-tuning **sparse-matrix-dense-vector** (SpMV) multiplication such as OSKI [27]. The computation pattern of each SMO loop is like **sparse-matrix-sparse-vector** multiplication (double vectors). Compared to the traditional sparse-matrix-dense-vector multiplication, there are three major differences: 1) The two vectors (X_{high} and X_{low}) are randomly selected from the rows of the matrix, which makes the design space fundamentally different from the sparse-matrix-dense-vector multiplication. 2) The choice of the format for these two vectors is highly related to the performance. 3) OSKI only targets SpMV, while for machine learning applications, some matrices are dense [28].

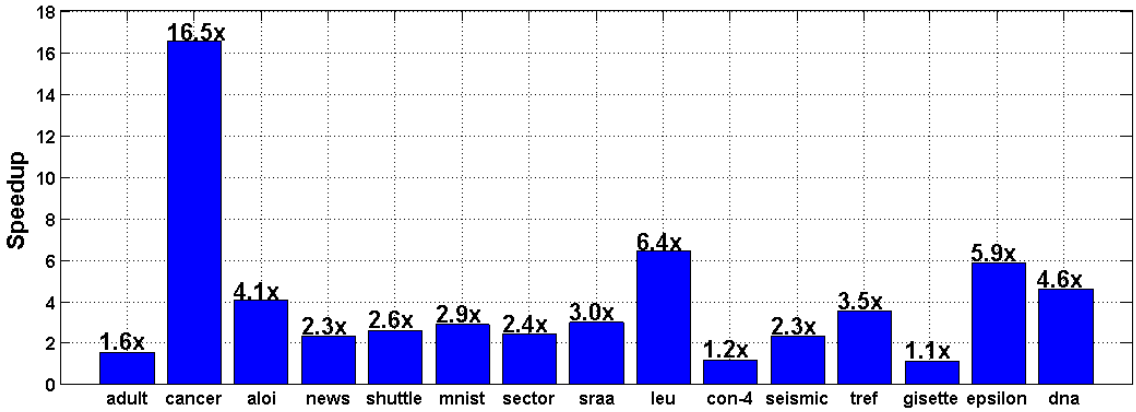

Figure 7. Speedups of HPC-SVM over Parallel Libsvm on the same Ivy Bridge CPUs for different real-world applications

In terms of fast SVM design, continuous efforts have been made to speedup SVM training in the last twenty years. Some pioneers proposed strategies for faster serial algorithms such as dataset decomposition technique [1], points shrinking, caching [2], minimal working set [3], and second order working set selection [29]. Most of these serial techniques have already been adapted in LIBSVM [26]. Others have tried to design parallel SVM on distributed memory systems (e.g. [8]). 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 [3] algorithm. Catanzaro [5] first proposed the GPUSVM for the binary classification problem. Herrero-Lepez then [30] improved Catanzaro's work by adding the support for Multiclass classification. Tsung-Kai Lin [4] used a CSR sparse format for data processing on GPUs. However, all of them lack dynamic adaptive support for input data patterns, which can dramatically reduce performance and practicality of the implementation. We designed and implemented MIC-SVM [31], which is a fast software for x86 platform. Our previous work CA-SVM [7] is a general divide-and-conquer approach for distributed systems. The techniques of this paper can be added to CA-SVM for better performance.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC0010200; by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008700 and AC02-05CH11231; by DARPA Award Number HR0011-12-2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle and S Samsung. Other industrial sponsors include Mathworks and Cray. Yang You also thanks his previous funding mentioned in [32] [33] [34] [35].

REFERENCES

[1] 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.

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

[3] J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods*, pp. 185–208, 1999.

[4] 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.

[5] 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.

[6] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[7] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc, "Ca-svm: Communication-avoiding support vector machines on distributed systems," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 847–859.

[8] 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.

[9] Y. You, S. L. Song, H. Fu, A. Marquez, M. M. Dehnavi, K. Barker, K. W. Cameron, A. P. Randles, and G. Yang, "Mic-svm: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 809–818.

[10] Y. You, H. Fu, S. L. Song, A. Randles, D. Kerbyson, A. Marquez, G. Yang, and A. Hoisie, "Scaling support vector machines on modern hpc platforms," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 16–31, 2015.

[11] Y. You, X. Lian, J. Liu, H.-F. Yu, I. S. Dhillon, J. Demmel, and C.-J. Hsieh, "Asynchronous parallel greedy coordinate descent," in *Advances In Neural Information Processing Systems*, 2016, pp. 4682–4690.

[12] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc, "Design and implementation of a communication-optimal classifier for distributed kernel support vector machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 974–988, 2017.

[13] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *ISMIR 2011: Proceedings of the 12th International Society for Music Information Retrieval Conference, October 24-28, 2011, Miami, Florida*. University of Miami, 2011, pp. 591–596.

[14] L. group. (2015) Compressed row storage (crs). [Online]. Available: http://netlib.org/linalg/html_templates/node91.html

[15] R. G. Grimes, D. R. Kincaid, W. I. MacGregor, and D. M. Young, "Itpack report: adaptive iterative algorithms using symmetric sparse storage," *CNA-139, Center for Numerical Analysis, University of Texas, Austin, Texas*, vol. 78712, 1978.

[16] Y. Saad, *SPARSKIT: A basic toolkit for sparse matrix computations*. Research Institute for Advanced Computer Science, NASA Ames Research Center Moffet Field, California, 1990.

[17] *Iterative methods for sparse linear systems*. Siam, 2003.

[18] A. Rocha and S. Goldenstein, "Multiclass from binary: Expanding one-vs-all, one-vs-one and ecoc-based approaches."

[19] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[20] 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.

[21] T. Davis and Y. Hu, "The university of florida sparse matrix collection," 2014. [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices/

[22] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[23] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[24] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, "On the importance of initialization and momentum in deep learning." *ICML (3)*, vol. 28, pp. 1139–1147, 2013.

[25] 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.

[26] 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.

[27] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[28] 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.

[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] 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.

[31] Y. You, S. Song, H. Fu, A. Marquez, M. Dehnavi, K. Barker, K. W. Cameron, A. Randles, and G. Yang, "Mic-svm: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures," in *Parallel & Distributed Processing (IPDPS), 2014 IEEE 28th International Symposium on*. IEEE, 2014.

[32] Y. You, H. Fu, S. L. Song, M. M. Dehnavi, L. Gan, X. Huang, and G. Yang, "Evaluating multi-core and many-core architectures through accelerating the three-dimensional lax–wendroff correction stencil," *The International Journal of High Performance Computing Applications*, vol. 28, no. 3, pp. 301–318, 2014.

[33] Y. You, H. Fu, X. Huang, G. Song, L. Gan, W. Yu, and G. Yang, "Accelerating the 3d elastic wave forward modeling on gpu and mic," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1088–1096.

[34] Y. You, D. Bader, and M. M. Dehnavi, "Designing a heuristic cross-architecture combination for breadth-first search," in *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 2014, pp. 70–79.

[35] Y. You, H. Fu, D. Bader, and G. Yang, "Designing and implementing a heuristic cross-architecture combination for graph traversal," *Journal of Parallel and Distributed Computing*, 2016.