Leveraging Data Density and Sparsity for Efficient SVM Training on GPUs

Borui Xu
School of Software
Shandong University
boruixu@mail.sdu.edu.cn

Zeyi Wen*
Hong Kong University of Science and Technology (Guangzhou)
Hong Kong University of Science and Technology
wenzeyi@ust.hk

Lifeng Yan
School of Software
Shandong University
lifeng.yan@mail.sdu.edu.cn

Zhan Zhao
School of Software
Shandong University
zhaozhan@mail.sdu.edu.cn

Zekun Yin
School of Software
Shandong University
zekun.yin@sdu.edu.cn

Weiguo Liu*
School of Software
Shandong University
weiguo.liu@sdu.edu.cn

Bingsheng He
School of Computing
National University of Singapore
hebs@comp.nus.edu.sg

***Abstract*—Support Vector Machines (SVMs) are a widely adopted data mining algorithm for binary and multi-class classification due to their ability to handle high-dimensional and non-linearly separable problems. However, SVM training is computationally expensive because of the heavy kernel matrix computation on large training datasets. Although much effort has been made to accelerate the training of SVMs, we find that existing libraries still suffer from inappropriate matrix multiplication methods and inefficient memory access patterns. In this paper, we propose a series of optimization approaches to address these limitations, including (i) matrix partitioning based on column density to achieve efficient kernel matrix computation; (ii) optimizing high latency memory access patterns; and (iii) dynamically selecting more suitable matrix multiplication methods based on the training dataset characteristics. Our proposed methods demonstrate significant improvements in SVM training performance without sacrificing accuracy, achieving a maximum speedup of 52x over the state-of-the-art SVMs on GPUs. These results highlight the effectiveness of our optimization in improving SVM training efficiency.**

***Index Terms*—machine learning, SVM, kernel matrix, graphics processing units**

I. Introduction

Traditional machine learning methods such as Support Vector Machines (SVMs) still perform well in many application scenarios, along with the tremendous success of deep learning techniques [1], [2] in data mining. SVMs are one of the most widely used data mining algorithms in many fields [3], [4]. They are particularly good at tackling classification and regression problems on high-dimensional datasets. Many deep learning studies also consider incorporating SVMs as part of their models [5] or using them as benchmark models [6], [7]. Fig. 1 depicts the number of repositories using SVMs on GitHub has exceeded thirty thousand. As one of the well-known SVM libraries, LibSVM has also been attracting increasing attention. However, training SVMs on large-

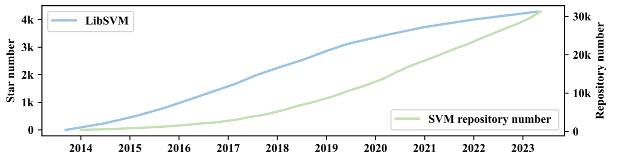


Fig. 1. Growth of repo. using SVMs and that of stars of LibSVM on GitHub.

scale datasets is highly time-consuming. Training large-scale datasets using LibSVM may require more than one or even two days (e.g., *epsilon* and *url_combined* dataset). Therefore, the efficient training of SVMs on large-scale datasets remains a topic of interest.

There are numerous libraries for efficient SVM training. SVMLight [8] is the first SVM library based on the sequential minimal optimization (SMO) algorithm. And LibSVM [9] is the most famous one, which implemented parallel training utilizing OpenMP. LibLINEAR [10] presented a fast dual method to accelerate the training of linear SVMs. ThunderSVM [11] further optimized SVM training by improving SMO on graphics processing units (GPUs). PSVM [12] considered matrix approximation. TensorSVM [13] developed this thought on GPUs. PLSSVM [14] applied the least square training method on GPUs. Despite the considerable progress made by these libraries, most of them only focused on algorithmic-level optimizations. And some libraries trade off accuracy [12]–[14]. Regarding the aspect of kernel matrix multiplication, there are still two challenges.

Firstly, it is inefficient in data representation and computation. The aforementioned software libraries only utilize compressed sparse row (CSR) [15] format or dense format to store and compute various matrices, which sometimes results in extra memory overhead and inefficient matrix multiplication operations. The CSR format is suitable for sparse matrices and stores the non-zero elements of the matrix in a compressed form, which saves memory and reduces the number of computation operations. However, it is not suitable for storing and

This work is in part supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-002).

The source code URL: https://github.com/Xtra-Computing/thundersvm

* Corresponding authors

computing dense matrices. For instance, a dense matrix of size 3 GB requires over 6 GB of space using sparse format. The performance of sparse matrix multiplication is generally significantly lower than dense matrix multiplication [16]. On the other hand, the dense matrix format is also not suitable for sparse matrices.

Secondly, there is low efficiency in memory access during sparse matrix computation. The memory throughput of sparse matrix multiplication in ThunderSVM on NVIDIA RTX 3090 is only around 60 GB/s, which is much lower than the GPU bandwidth. Memory access efficiency is an important factor affecting the performance of matrix multiplication. However, many studies have not considered optimizing memory access patterns. With the rapid growth in dataset size in the field of data mining, there is a need for efficient implementations of SVM training.

To address the above issues, we propose a series of methods to optimize the time-consuming kernel function computation in SVM training from both computational and memory access aspects. We dynamically select the kernel matrix computation method according to the distribution of non-zero elements and dataset sizes. Furthermore, we specifically optimize memory access in sparse matrix computation, thereby obtaining higher computational performance. We conduct an analysis of performance metrics such as FLOPS, memory throughput, and others to demonstrate the effect of memory access optimization. Overall, our key contributions in this paper are as follows.

- We propose a novel column density-based matrix partitioning method to improve the efficiency of matrix multiplication during training. Depending on the size and sparsity of each dataset, we dynamically choose a more suitable matrix multiplication method to maximize training efficiency.
- We optimize memory access patterns in sparse matrix multiplication. We improve the continuity of memory access in matrix multiplication by reordering the matrix columns. Besides, we increase the effective bandwidth during multiplication through matrix transposition and other operations.
- We conduct extensive experiments on datasets of varying sizes and different sparsity to investigate the effectiveness of our proposed methods called "BoostSVM". The experimental results demonstrate that BoostSVM achieves high performance without sacrificing accuracy on datasets of different sizes and sparsity levels. Compared with the CPU-based SVM, our methods are faster by one to two orders of magnitude. Moreover, BoostSVM surpasses the state-of-the-art SMO-based SVM on GPUs by up to 52 times. In terms of kernel matrix multiplication, our implementation achieves more than twice the FLOPS of its counterpart and over four times the memory throughput.

II. Background and Related Work

In this section, we first introduce the fundamental knowledge of SVMs and sequential minimal optimization algorithm. Then, we discuss some related work.

A. Support Vector Machines

SVMs were originally introduced by Vapnik et al. [17] in the 1990s. The fundamental concept in Support Vector Machines (SVMs) involves the identification of a hyperplane within a high-dimensional space that effectively separates data points belonging to distinct classes. The selection of this hyperplane is based on the objective of maximizing the margin, defined as the spatial gap between the hyperplane and the closest data points, referred to as support vectors, within each class. Formally, given a set of training data $x_1, x_2, \ldots, x_m$ where each x_i is an input vector and $y_1, y_2, \ldots, y_m$ are the corresponding output labels (either -1 or 1), the training process is equivalent to solve the following optimization problem:

$$\begin{aligned} \underset{w,\xi,b}{\arg\min} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y_i(w \cdot x_i + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, \forall i \in \{1, \ldots, m\} \end{aligned} \tag{1}$$

where w represents the normal vector of the hyperplane, C denotes the penalty parameter, b signifies the bias associated with the hyperplane, and ξ corresponds to the slack variables that permit certain samples to be misclassified.

A notable advantage of SVMs lies in their capacity to effectively handle non-linearly separable data by employing various kernel functions. The kernel function transforms the initial feature space into a higher dimensional feature space, where it is possible to separate the data points linearly. In general, we solve SVMs with kernel functions optimization problems through dual problems as below:

$$\begin{aligned} \underset{\alpha}{\arg\min} \quad & \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j K(x_i, x_j) - \sum_{i=1}^{m}\alpha_i \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, i \in \{1, \ldots, m\}, \quad \sum_{i=1}^{m} y_i a_i = 0 \end{aligned} \tag{2}$$

where a_i is the Lagrange multipliers of x_i, $K(x_i, x_j)$ is a kernel function that is used to represent the similarity between sample x_i and x_j. In a compact way, $y_i y_j K(x_i, x_j)$ can be represented as $Q_{i,j}$, where Q stands for the $m \times m$ similarity kernel matrix. And the selection of a kernel function will influence the SVM's performance. There are several popular choices, such as the polynomial kernel, the Gaussian function kernel, and the sigmoid kernel.

B. Sequential Minimal Optimization Algorithm

Although the SVM is a powerful algorithm, its training process can be computationally intensive, especially in processing large datasets. To address this problem, various techniques have been proposed to optimize the SVM training process. The SMO algorithm [18] is a popular method for solving the SVM optimization problem, particularly for large-scale datasets. SMO decomposes the whole quadratic optimization problem into a sequence of easier sub-problems that can be solved by optimizing two Lagrange multipliers and fixing the rest at a time. This approach allows SVMs to converge faster than traditional optimization methods.

Formally, in SMO, each training sample x_i has a corresponding optimality indicator, reflecting the optimal condition:

$$g_i = \sum_{j=1}^{m} \alpha_j y_j K(x_i, x_j) - y_i \tag{3}$$

where the first half of the formula on the right is the predicted value of x_i, and y_i is the ground truth of x_i. Now we will introduce the process of using SMO to train a binary SVM.

The **first** step is to select a working set of two training samples whose variable α need to be optimized. We employ the working set selection method proposed by Fan et al. [19] to find two training samples x_i and x_j. The selection formula is as follows:

$$\begin{aligned} i &= \arg\min_t \{g_t | t \in I_{up}\} \\ j &= \arg\max_t \left\{ \frac{(g_i - g_t)^2}{\eta_t} | g_i < g_t, t \in I_{low} \right\} \end{aligned} \tag{4}$$

where kernel similarity variable $\eta_t = K(x_i, x_i) + K(x_t, x_t) - 2K(x_i, x_t)$, i is the sample that most violates the KKT condition, I_{up} and I_{low} are defined as follows:

$$\begin{aligned} I_{up} &= \{t | \alpha_t < C, y_t = 1 \quad or \quad \alpha_t > 0, y_t = -1\} \\ I_{low} &= \{t | \alpha_t < C, y_t = -1 \quad or \quad \alpha_t > 0, y_t = 1\} \end{aligned}$$

The **second** step is to update the weights α_i and α_j corresponding to x_i and x_j. The update formula is as follows:

$$\begin{aligned} \alpha_j^{new} &= \alpha_j + \frac{y_j(g_i - g_j)}{\eta}, \\ \alpha_i^{new} &= \alpha_i - \frac{y_i(g_i - g_j)}{\eta} \end{aligned} \tag{5}$$

where $\eta = K(x_i, x_i) + K(x_j, x_j) - 2K(x_i, x_j)$. It should be noted that the update of α needs to be constrained in $[0, C]$.

The **third** step is to update all optimality indicators. The update formula is as follows:

$$g_t^{new} = g_t + \frac{(g_i - g_j)}{\eta}(K(x_j, x_t) - K(x_i, x_t)) \tag{6}$$

SMO loops the three steps until $g_i \geq \max\{g_t | g_t \in I_{low}\}$.

Although the SMO algorithm is effective in training SVMs, the process of decomposing the problem into extremely small sub-problems does not fit well with the structure of the GPU. In order to fully utilize the effectiveness of SMO on GPUs, Wen et al. [11] modified the SMO algorithm. The modified process is as follows:

- Select a working set consisting of n samples from the training dataset based on certain conditions.
- Compute the multiplication of the training dataset matrix and the working set matrix to get a local kernel matrix.
- Use the SMO algorithm on this working set kernel matrix to obtain a locally optimal solution.
- Repeat the above steps until the conditions are met.

C. Related Work

Mainstream training algorithms for SVMs include SMO [18], least squares [20] and kernel approximation [21].

1) Training SVMs Using SMO on CPUs: SMO algorithm is one of the most widely used SVM training algorithms. In the early stages, the implementation and optimization of SMO are mainly focused on CPU platforms. Many SVM libraries, such as SVMLight [8], LibSVM [9], and WEKA [22], are based on this algorithm on the CPU platform. As an extension of LibSVM, LibLINEAR [10] optimizes the performance of linear SVM. With the development of computer systems and architectures, many studies chose to use distributed systems for SVM training, such as MPI SVMs [23] and MapReduce SVMs [24], but the inherent communication overhead also makes the training efficiency of SVMs low.

2) Training SVMs Using SMO on GPUs: With the development of GPUs and the inherent suitability of graphics processors for parallel computing, how to use GPUs to accelerate SVM training has become a hot topic. Catanzaro et al. [25] are the first to introduce GPUs into the SVM training process. Herrero-Lopez et al. [26] implemented a multi-class SVM on GPUs. Athanasopoulos et al. [27] used GPUs to accelerate the computation of the kernel function during training. These studies require computing the entire kernel matrix, and since GPU memory is much smaller than the host memory, they can only handle datasets that are not very large. In order to enable GPUs to process larger sparse datasets, Cotter et al. [28] utilized a sparse format for training data storage and proposed a clustering method to take advantage of data sparsity. Another work [29] compared various implementations of SVMs on GPUs, provided benchmark results, and proposed a training algorithm specifically designed for binary classification tasks. Wen et al. [11] proposed ThunderSVM, which solves the problem of high memory latency on GPUs through batch processing and kernel value reuse.

3) Least Squares SVM Training: Suykens et al. [20] proposed the least squares support vector machine (LS-SVM), which trains SVMs using the least squares method to simplify the optimization problem. To train the LS-SVM, a suitable kernel function is chosen to describe the similarity between data points. By constructing a kernel matrix, the classification problem is transformed into a linear regression problem. Thus, a linear regression model can be used to train the LS-SVM and obtain the coefficients and intercept of the model. And they extended the LS-SVM by introducing weights [30] and implementing support for multi-class [31] and sparse structures [32]. Do et al. [33] exploited GPUs to accelerate this approach, but their implementation was limited to datasets no more than 200 data points. Craen et al. [14] implemented LS-SVM on GPUs and relaxed the requirement for dataset size.

4) Training SVMs with Kernel Approximation: Kernel approximation methods are able to approximate the kernel function in SVMs to reduce the computational cost and memory usage during training. These methods aim to approximate the kernel matrix by using a low-dimensional feature space, which reduces computational complexity. Fine et al. [21]

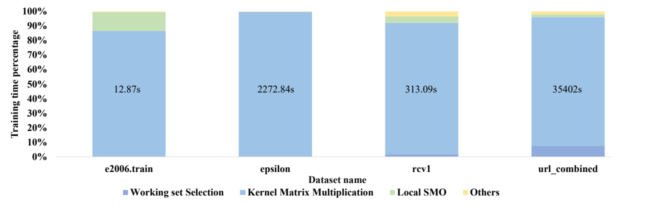


Fig. 2. The percentage of each part in SVM training. Kernel matrix multiplication accounts for the largest proportion.

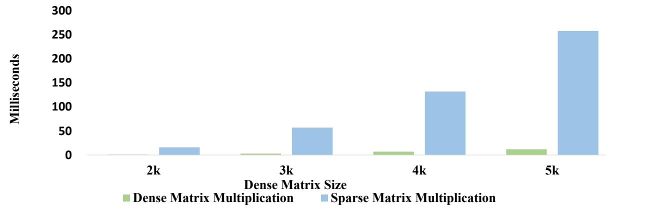


Fig. 3. Efficiency comparison of different multiplication. Dense matrix multiplication is far better on dense matrices.

proposed an SVM training algorithm based on low-rank Gram matrix approximation. Nystrom's method [34] and LLSVM [35] further extended this thought. PSVM [12] considered using MPI parallel implementation for these methods. Zhang et al. [13] developed this thought on GPUs. They utilized TensorCore to accelerate the training process.

5) Comparison of SVM Training Algorithms: Although using algorithms such as least squares and kernel approximation can speed up the training of SVMs compared with SMO, there are also inevitable drawbacks. For the LS-SVM, training results are more prone to overfitting, its solution is not sparse, and the model accuracy is difficult to compare with the SVM trained by SMO. Also, constructing the kernel matrix during LS-SVM training can be computationally expensive and memory-intensive for large-scale data. For kernel approximation, this approach also reduces the accuracy of the model results, and different datasets require the selection of different approximation parameters, which requires a deep understanding and rich experience of the data. In general, the SMO algorithm is still the best choice in terms of universality and model accuracy. Besides that, we can observe that due to the inherent advantage of GPUs in multi-core parallelism, an increasing number of works are considering implementation on GPUs. Therefore, our methods choose to optimize the SMO algorithm on GPUs.

III. METHODOLOGY

In this section, we first analyze the characteristics of kernel matrix multiplication in the SVM and identify the associated challenges. Then we describe the approaches to optimize matrix multiplication: column density-based matrix partitioning, memory access optimization, and dynamic strategy selection.

A. Kernel Matrix Multiplication Analysis

In the process of utilizing SVMs, datasets are commonly represented in matrix form, with each row representing a training sample and each column representing a feature dimension. In the SMO algorithm, the multiplication between the training set matrix and the working set matrix in the kernel function has always been the bottleneck in the training process of SVM. This is because the kernel matrix is usually too large to be directly stored in memory. Assuming there are 10^5 samples in the training dataset, the size of the kernel matrix will be $10^5 \times 10^5$, which will occupy about 40GB of memory footprint. This is not feasible for most GPU devices. Therefore, we need to compute the required parts of the kernel matrix in real time during the training iteration. Fig. 2 illustrates the percentage of training time in different parts using ThunderSVM, indicating that the time spent on kernel matrix multiplication almost exceeds 90% for the whole training process.

When it comes to the implementation of matrix multiplication in existing SVM libraries such as LibSVM and ThunderSVM, only sparse matrix representations (like CSR format) and sparse matrix computations are considered. Compared with the dense matrix multiplication that calculates each element through a traversal loop, sparse matrix multiplication only computes the product of non-zero elements. Taking the CSR format sparse matrix as an example, the non-zero elements are first found through the row pointer array and the column index array and then multiplied accordingly. Although sparse matrix multiplication can reduce the number of operations when handling sparse matrices, it is not suitable for storing and computing dense matrices. Under equivalent computational loads, dense matrix multiplication usually outperforms sparse matrix multiplication. Fig. 3 depicts the performance differences between dense matrix multiplication and sparse matrix multiplication on dense square matrices. As the matrix size increases, the time required for sparse matrix multiplication increases exponentially. On NVIDIA RTX 3090, the computational performance of dense matrix multiplication is around 20 TFLOPS, while the performance of sparse matrix computation is around 1 TFLOPS. This is because, in many cases, sparse matrix multiplication is memory-bound. Using row pointer arrays and column index arrays to locate elements incurs many additional high-latency memory accesses. Obtaining the corresponding element in a dense matrix only requires one memory access. However, in a sparse format matrix, it requires three.

In data mining, there is a great difference in the sparsity of matrices, which is due to the infrequent features that matrices express in different fields. Therefore, leveraging the advantages of both dense matrix multiplication and sparse matrix multiplication will be crucial in optimizing matrix multiplication in SVMs.

B. Column Density-based Matrix Partitioning

In order to achieve more efficient memory access in matrix multiplication for SVMs, we partition the matrix into sparse and dense two sub-matrices based on column density and perform matrix multiplication on each separately.

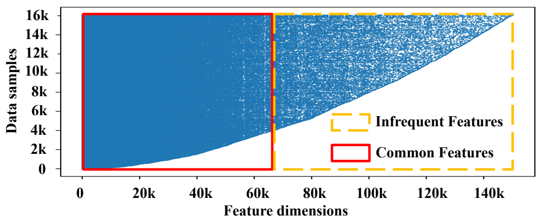


Fig. 4. Non-zero element distribution in *e2006.train*. The blue dot indicates the feature value of a sample on the that feature is non-zero.

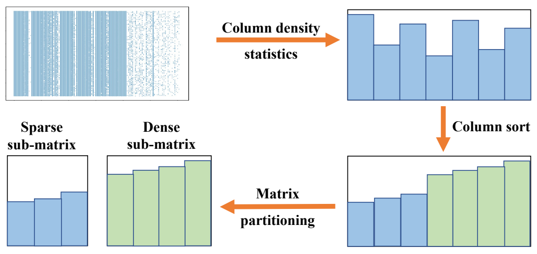


Fig. 5. Column density-based matrix partitioning overview

For a certain dataset, The frequencies of different features are non-uniform. In this paper, we refer to the features that are present in more than 50% of the samples as common features, while those that appear in less than 50% are referred to as infrequent features. Taking the *e2006.train* dataset as an example, we present the non-zero element distribution of this dataset in Fig. 4, where the horizontal axis represents different features, the vertical axis represents different samples, and the blue data points indicate the distribution of non-zero element. As we can see, the density of non-zero elements in different feature columns varies greatly. Although the density of non-zero elements in this dataset is only 0.82%, there are dense sub-matrices within the sparse matrix due to the common features shared among most samples. We can represent and compute this dense sub-matrix using the dense matrix format. Compared with a large amount of random memory access in sparse matrix multiplication, the sequential memory access pattern in dense matrix multiplication will effectively improve the utilization of bandwidth and improve the overall efficiency of matrix multiplication. Therefore, we propose a partitioning method that divides the matrix into two parts: dense sub-matrix and sparse sub-matrix, based on column density. After that, we employ sparse matrix multiplication to handle the sparse sub-matrix and dense matrix multiplication to handle the dense sub-matrix. The partitioning process is visually summarized in Fig. 5. Algorithm 1 presents the pseudocode for the proposed approach. We can obtain the column grouping array $dense\_flag$, indicating the category to which each column feature belongs.

In Algorithm 1, we take the training dataset X and related row-column information as input. X is a two-dimensional array, where the first dimension represents the number of training samples and the second dimension represents the number of features in each sample. During the execution of Algorithm 1, we first initialize the required variables. col_num is used to count the number of non-zero elements in each column of the training matrix, col_index stores the initial position of each column, and $dense\_flag$ is used to indicate whether the current column belongs to the sparse sub-matrix or dense sub-matrix. After counting non-zero elements in each column, we sort the column indices in ascending order using the *sort* function in line 8 based on the number of non-zero elements and store the results in col_index. By traversing col_index, we partition each column into either the sparse or dense sub-matrix based on thresholds. Specifically, we set the threshold to make the number of non-zero elements in the sparse sub-matrix will not exceed 5% of the total elements in the sparse sub-matrix. This is because, in the implementation of sparse matrix multiplication, we use NVIDIA's cuSPARSE library [36] as the backend, which is designed to handle sparse matrices with a sparsity level of no more than 5%. And compared with dense matrix multiplication, the sparser the matrix, the better the performance of sparse matrix multiplication. In addition, we also use column density as another criterion for matrix partitioning. Assuming that there are z non-zero elements in a column of the training set, the CSR format matrix representation will store additional $2z$ elements (element values and column indices). If the size of z exceeds half the number of rows m in the training set, the memory consumption of CSR format storage will be greater than that of dense format ($2z > m$). As the GPU memory is often much smaller than the host memory, to save the GPU memory consumption in matrix storage, we also partition the feature columns with a density greater than 50% into the dense sub-matrix. After that, we use the $dense\_flag$ array to construct sparse and dense sub-matrices.

Algorithm 1: Column Density Matrix Partitioning

```
Input: training dataset $X$, row $m$, column $n$
Output: $dense\_col$, $dense\_flag$, $col\_index$
1  $dense\_col \leftarrow n$,   $sparse\_sum \leftarrow 0$;
2  for $i \leftarrow 1$ to $n$ do                         // parallel
3      $col\_num[i] \leftarrow 0$,   $col\_index[i] \leftarrow i$;
4      $dense\_flag[i] \leftarrow true$;
5  for $i \leftarrow 1$ to $m$ do                         // parallel
6      for $j \leftarrow 1$ to $X[i].size()$ do
7          $col\_num[X[i][j].col]$ += 1;            // atomic
8  sort($col\_index, col\_num$);            // sort $col\_index$
9  for $i \leftarrow 1$ to $n$ do
10     $current\_col \leftarrow col\_index[i]$;
11     $total\_sparse$ += $col\_num[current\_col]$;
12     $sparse\_sum$ += $m$;
13     $ratio \leftarrow total\_sparse \div sparse\_sum$;
14     if $ratio > 0.05$ or
         $col\_num[current\_col] \div m > 0.5$ then
15         break;
16     $dense\_col$ −= 1;
17     $dense\_flag[current\_col] \leftarrow false$;
```

Algorithm 2: Construction of Sparse Sub-matrix

```
Input: training dataset $X$, column map array
       $sparse\_map$, tag array $dense\_flag$
Output: $sparse\_val$, $sparse\_ptr$, $sparse\_index$
1  for $i \leftarrow 1$ to $m$ do                    // parallel
2      for $j \leftarrow 1$ to $X[i].size()$ do
3          if $dense\_flag[X[i][j].col] = false$ then
4              $sparse\_ptr[i+1] += 1$;
5  for $i \leftarrow 1$ to $m$ do
6      $sparse\_ptr[i+1] += sparse\_ptr[i]$;
7  for $i \leftarrow 1$ to $m$ do                    // parallel
8      $begin\_pos \leftarrow sparse\_ptr[i]$;
9      $offset \leftarrow 0$;
10     for $j \leftarrow 1$ to $X[i].size()$ do
11         if $dense\_flag[X[i][j].col] = false$ then
12             $new\_col = sparse\_map[X[i][j].col]$;
13             $sparse\_val[begin\_pos + offset] =$
                 $X[i][j].val$;
14             $sparse\_index[begin\_pos + offset] =$
                 $new\_col$;
15             $offset += 1$;
```

Although the matrix partitioning only needs to be done once at the beginning of SVM training and incurs minor overhead, we parallelize the matrix partitioning algorithm implementation to minimize unnecessary overhead. For obtaining col_num array in algorithm 1, we use the OpenMP library for parallelization and atomic operations for possible write conflicts. For the sub-matrix construction process, we optimize it with the GPU asynchronous execution. We use the CPU to construct the dense sub-matrix and the GPU to construct the sparse sub-matrix. The dense sub-matrix construction can be easily parallelized using the OpenMP library, but due to the dependency of the sparse matrix elements' positions in the CSR format, simple loop traversal cannot be effectively parallelized. Therefore, we adopt a multiple traversal method to construct the sparse sub-matrix and achieve high parallelization. The relevant pseudocode is shown in Algorithm 2. First of all, we parallelize the process of counting the number of non-zero elements in each row of the matrix. Then, we assign values to $sparse\_ptr$, which has dependencies between its values. Next, we parallelize the assignment of values to $sparse\_val$ and $sparse\_index$. This stepwise approach increases the degree of parallelism in the code implementation.

After completing the matrix partitioning, we need to perform matrix multiplication to construct the kernel matrix used in the SVM training. Fig. 6 illustrates the process of the original matrix multiplication in the existing work and the matrix multiplication after partitioning. In the original matrix multiplication, it first needs to extract a working set matrix of size $n \times k$ from the $m \times k$ training set matrix (the working set matrix used here is stored in dense matrix format). The working set matrix consists of several training samples in the training set matrix. And then, we multiply the two matrices

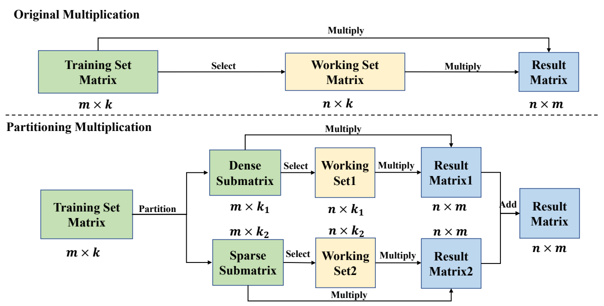


Fig. 6. Comparison of kernel matrix multiplication

to get a result matrix of size $n \times m$. In this process, only values in the same column are multiplied. Therefore, after the column partitioning of the training set matrix, we only need to construct the working set matrix from each of the two sub-matrices separately. Once the two sub-matrices and their corresponding working set matrices have been computed, two $n \times m$ result matrices are obtained. Due to the absence of computation dependencies between elements in two sub-matrices, we only need to add the two $n \times m$ sub-matrices together to restore the required result.

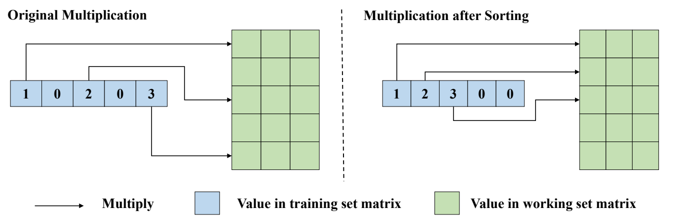
C. Memory Access Optimization

During SVM training, Sparse matrix multiplication (SpMM) is a fundamental operation for calculating the kernel matrix. In this process, the sparse matrix is multiplied by another dense matrix, and the result is stored in a dense matrix. The efficiency of this operation depends heavily on the memory access pattern, which determines the number of times the processor accesses the memory. Therefore, we consider optimizing the memory access in SpMM.

Algorithm 3: A Row-wise Implementation of SpMM

```
Input: sparse matrix $A[M][K]$, dense matrix $B[K][N]$
Output: dense matrix $C[M][N]$
1  for $i \leftarrow 1$ to $A.rows$ do
2      for $j \leftarrow A.row\_ptr[i]$ to $A.row\_ptr[i+1] - 1$ do
3          for $n \leftarrow 1$ to $N$ do
4              $C[i][n] +=$
                 $A.value[j] \times B[A.col\_index[j]][n]$;
```

Algorithm 3 demonstrates a general implementation of sparse matrix multiplication (CSR format). The algorithm iterates through the rows of A, and for each non-zero element in the row, it performs a dot product operation between the non-zero element and the corresponding row of B, then accumulates the result in the corresponding entry of C. This algorithm leverages the sparsity structure of matrix A by only performing computations on non-zero elements, reducing the overall computational complexity of the operation. Although it is difficult to change the memory access pattern of the CSR format sparse matrix during the multiplication process, the order of column indices in the sparse matrix can be changed to



Original Multiplication
Multiplication after Sorting
1 0 2 0 3
1 2 3 0 0
Multiply
Value in training set matrix
Value in working set matrix

Fig. 7. Comparison of memory access pattern. Continuous non-zero elements can reduce memory span.

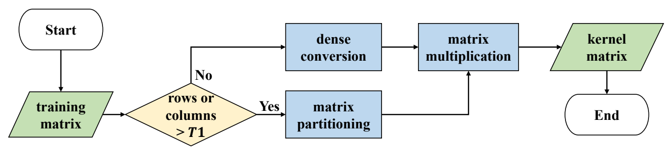


Fig. 8. Workflow of strategy selection

TABLE I
DATASETS FOR PERFORMANCE EVALUATION

Name	Cardinality	Dimension	Data density
url_combined	2396130	3231961	0.0036%
news20.binary	19996	1355191	0.034%
rcv1_test.binary	677399	47236	0.15%
real-sim	72309	20958	0.24%
e2006.train	16087	150360	0.82%
mnist.scale	60000	780	19.22%
ijcnn1.t	91701	22	56.52%
gisette	6000	5000	99.08%
epsilon	400000	2000	100%

affect the memory access continuity of the dense matrix. Fig. 7 illustrates the difference in accessing the working set matrix during multiplication for sparse matrices with different orders of non-zero elements, namely non-contiguous and contiguous orders. Assuming that the matrix is row-major stored in memory, the placement of non-zero elements in the sparse matrix has a significant impact on the access pattern of the working set matrix during multiplication. Specifically, if the majority of non-zero elements are stored contiguously in adjacent columns in a data sample, it greatly reduces the access span of the dense matrix during multiplication and enhances cache continuity. In the case of sequential access, where data is stored in order, memory can utilize prefetching techniques to read and cache the next block of data in advance, thereby speeding up access. Sequential access is usually faster than random access in a large number of consecutive read operations. Therefore, in the matrix partitioning process, we construct the sparse sub-matrix according to the column density order, arranging columns with similar densities together. This makes the local non-zero elements denser and improves the memory access continuity of the working set matrix.

As for the implementation of sparse matrix multiplication on the GPU, we also optimize memory access. Coalesced memory access is a technique used to optimize memory access patterns. It refers to the process of accessing contiguous blocks of memory by threads within a warp. When obtaining the working set matrix, we coalesce the accesses using column-major storage to obtain a dense $n \times k$ matrix. We then multiply this matrix with the $m \times k$ training set matrix to obtain an $m \times n$ dense result matrix, which is then transposed to obtain an $n \times m$ dense target matrix. Compared with obtaining the target matrix directly through multiplication, obtaining the transpose of the target matrix first and then performing the conversion can remarkably reduce data transfer from global memory. In our test, this can reduce the amount of data transfer by approximately 50%.

D. Dynamic Strategy Selection

As mentioned earlier, the size and sparsity of the training matrix vary greatly, and a single kernel matrix computation method is difficult to perform well on all datasets. We adopt a strategy of dynamically selecting matrix computation methods to minimize the SVM training time as much as possible. Based on the dataset size, we dynamically choose between column density-based matrix partitioning multiplication and dense matrix multiplication. Fig. 8 illustrates the selection process for different strategies.

Once the training data matrix is obtained, we first check the size of the matrix samples and features. If it exceeds threshold $T1$, we choose the matrix partitioning computation method. Otherwise, we select the dense matrix multiplication. Although employing a dense matrix representation for sparse matrices leads to an increase in multiplication operations, the peak performance of dense matrix computations surpasses that of sparse matrix multiplication. Besides that, it can be observed that, in general, the matrices of small-scale datasets are not highly sparse. Therefore, when dealing with small-scale datasets, dense matrix multiplication generally requires less time. As the size of the datasets increases, we apply matrix partitioning multiplication to balance the advantages between dense matrix multiplication and sparse matrix multiplication.

IV. EXPERIMENTAL STUDY

In this section, we have empirically evaluated the performance of the proposed optimization methods. We compare the training time and training results with existing relevant works and perform separate comparative analyses on various optimization techniques. We use **BoostSVM** to refer to the proposed optimization methods.

A. Experimental Setup

1) Platform: All experiments have been conducted on a workstation running Ubuntu 20.04 with an Intel(R) Core(TM) i9-10900K 10-core CPU, 128GB of memory, and an NVIDIA RTX 3090 GPU with 24GB global memory. The g++ compiler version is 9.4.0, the NVIDIA driver version is 530.30.02, and the CUDA version is 11.7. Our optimization methods are implemented using CUDA C++. In the implementation of matrix multiplication and matrix transposition, we use NVIDIA's cuSPARSE library for sparse matrix multiplication and the cuBLAS library [37] for dense matrix multiplication and matrix transposition.

TABLE II
EFFICIENCY COMPARISON AMONG LIBSVM, THUNDERSVM, AND OUR BOOSTSVM

Dataset	Training time (sec)				Speedup over		
	LibSVM	ThunderSVM	TensorSVM	BoostSVM	LibSVM	ThunderSVM	TensorSVM
url_combined	8.91×10^4	4.16×10^4	OOM	6.92×10^3	12.81	6.01	-
news20.binary	117.89	6.50	OOM	3.09	38.15	2.10	-
rcv1_test.binary	1.23×10^4	348.58	OOM	93.76	107.33	3.72	-
real-sim	97.97	3.80	21.71	1.97	49.73	1.93	11.02
e2006.train	451.96	13.86	-	3.10	145.79	4.48	-
mnist.scale	1.49×10^3	53.69	-	25.40	58.71	2.11	-
ijcnn1.t	3.98	1.17	2.72	0.84	4.74	1.39	3.24
gisette	80.67	2.75	1.02	0.58	139.09	4.74	1.76
epsilon	$1.72 \times 10^5+$	2.28×10^3	66.98	43.67	3956+	52.25	1.53
Average					500+	8.75	4.39

2) Datasets: We select nine publicly available datasets based on their data size and sparsity from the LibSVM dataset repository[1]. Because existing SVM implementations are already fast on small-scale datasets, we primarily focus on their performance on large-scale sparse datasets. The dataset information is sorted by sparsity in Table I. In this section, *url_combined* is the abbreviation for *url_combined_normalized*, while *epsilon* is the abbreviation for epsilon_normalized. *mnist.scale* is a multi-class classification dataset consisting of ten classes. And *e2006.train* is a regression dataset. Others are all binary classification datasets with significant differences in size and sparsity. Except for the *epsilon_normalized* and *e2006.train*, which have parameters of ($C = 0.01, \gamma = 1$) and ($C = 256, \gamma = 0.125$) respectively, the parameters for all other datasets are set to ($C = 100, \gamma = 0.5$).

3) Baselines: We have compared BoostSVM with LibSVM [9] (OpenMP version), ThunderSVM [11] (GPU version), and TensorSVM [13] to provide a comprehensive evaluation. Both ThunderSVM and LibSVM also use the SMO algorithm to train SVM, which enables us to demonstrate the effectiveness of our proposed methods directly. TensorSVM is an SVM implementation based on kernel approximation on GPUs. Since TensorSVM only offers binary classification SVM, we tested it on binary classification datasets. We used the default approximation rank ($rank = 32$) and enabled the tensor-core in TensorSVM. The threshold $T1$ in BoostSVM is 5×10^4, which is selected empirically. All other parameters are kept the same during training, and the Gaussian kernel function is used in all experiments.

B. Efficiency Comparison

Table II compares the time spent and speedup on training SVMs over different implementations on nine datasets. The experimental results are represented in seconds and rounded to two decimal places. OOM is the abbreviation for "out of memory". The results show that our BoostSVM achieves the best performance on all test datasets. The training time required for BoostSVM is generally one to two orders of magnitude less than that of LibSVM. Compared with ThunderSVM and TensorSVM, it also significantly reduces the training time.

1) Comparison with ThunderSVM: From the table II, it can be seen that BoostSVM is 8.75 times faster on average than ThunderSVM. On the *epsilon* dataset, the speedup of BoostSVM exceeds 52 times over ThunderSVM. This is because the training set is a very dense dataset, and the matrix multiplication method provided in ThunderSVM is not suitable for this dataset. After the optimization of matrix partitioning in BoostSVM, the whole dataset is computed using dense matrix multiplication. This can increase the memory throughput from 46 GB/s to 111 GB/s and the computational performance from 462 GFLOPS to 23 TFLOPS. In addition to a significant improvement on dense datasets, BoostSVM can also bring multiple improvements on sparse datasets. For example, on the *url_combined* dataset, although the data density of this dataset only reaches 0.0036%, there are still dense common feature columns in this dataset. After partitioning, we can obtain a dense matrix with a non-zero elements density of up to 74% and a sparser sparse matrix, then let the two sub-matrices perform multiplication separately. With the support of memory access optimization for sparse matrix operations, the final speedup can exceed 6 times over ThunderSVM.

2) Comparison with TensorSVM: BoostSVM is 4.39 times faster on average than ThunderSVM. TensorSVM is an SVM implementation based on kernel approximation on GPUs. Despite using different training algorithms from BoostSVM, we also compared our methods with TensorSVM to provide a more comprehensive analysis and comparison. Theoretically, kernel approximation needs less training time than the SMO algorithm. However, from the table II, we can observe that BoostSVM shows better training efficiency because of high hardware-based computational optimizations. This further demonstrates the effectiveness of our optimization. As for the result accuracy, typically, the results of kernel approximation are inferior to those of the SMO algorithm. For instance, on the *ijcnn1.t* dataset, BoostSVM achieved an accuracy of 95.96% on the test dataset, while TensorSVM only reached 64.37%. Besides that, BoostSVM also exhibits stronger robustness. TensorSVM fails to execute on large datasets such

[1]http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets

TABLE III
COMPARISON BEFORE AND AFTER MEMORY OPTIMIZATION

Dataset	ThunderSVM(GPU)	
	w/o memory optimization	w/ memory optimization
real-sim	3.80 s	2.69 s
rcv1_test.binary	348.58 s	100.27 s
e2006.train	13.86 s	7.02 s

as *url_combined* due to exceeding the available GPU memory, and it only supports binary classification tasks. But BoostSVM successfully trains under the same circumstances and supports multi-class tasks and regression tasks. Overall, the training results on the nine datasets demonstrate the effectiveness of the optimization in BoostSVM.

C. Impact of Individual Optimizations

In this section, we first illustrate the strategy selection in experiments, then analyze the effect of matrix partitioning and its overhead. Finally, we assess the memory access optimization.

1) Strategy Selection: Regarding the results in Table II, BoostSVM selects dense matrix multiplication on small-scale datasets such as *mnist.scale*, *gisette* and *ijcnn1.t*, and matrix partitioning multiplication on the other datasets. When the training set matrix is not large, regardless of the sparsity level of the matrix, using dense matrix multiplication consistently achieves better performance. This is because matrix multiplication is typically memory-bound. On small matrices, the computational savings achieved by sparse matrix multiplication cannot cover the impact of non-contiguous memory accesses. On the *ijcnn1.t* dataset, the training time is 1.081s using matrix partitioning, which is slower than direct dense matrix multiplication. Therefore, we directly apply the dense matrix multiplication for small-scale training matrices. As for the overhead of strategy selection, it is negligible because it evaluates the dataset only once before the start of training.

2) Analysis of Matrix Partitioning: Even on highly sparse datasets, matrix partitioning multiplication achieves significant acceleration by balancing dense matrix multiplication and sparse matrix multiplication. On the *e2006.train* dataset, the time can be optimized to 5.96 s only using matrix partitioning multiplication. This is twice as fast as ThunderSVM. On the *url_combined* dataset, the time is 1.31×10^4 s, which is four times faster than ThunderSVM. The results in Table II have already included the time of matrix partitioning in BoostSVM. Under parallel computing optimization, the overhead of this portion is negligible. For example, on the url_combined dataset with the longest training time, the matrix partitioning overhead is 2.72s, accounting for approximately 0.04% of the total training time. On the rcv1_test.binary dataset, the matrix partitioning overhead is 0.42s, accounting for approximately 0.44% of the total.

3) Memory Access Optimization: Table III illustrates the effect of memory access optimization. We optimize the GPU version of ThunderSVM to demonstrate the importance of memory access patterns. We are able to reduce the training time on these two sparse training sets only through memory access optimization. Among all memory access optimization methods, the optimization approach that gets the transpose matrix of the result first in the sparse matrix multiplication process, followed by transposing the result, has achieved the greatest improvement. To further investigate changes in relevant performance metrics, we used NVIDIA's Nsight Compute profiling tool to obtain detailed performance information, and the relevant results are shown in Table IV. We can see that both FLOPs, memory throughput, and compute utilization have been greatly improved. Taking the *real-sim* dataset as an example, the FLOPs metric increases from 419.75 GB to 1123.8 GB, and memory throughput increases from 61.11 GB/s to 221.40 GB/s. The data transfer from the GPU global memory has also significantly decreased from 2.31 GB to 1.26 GB. As for *rcv1_test.binary*, the data transfer decreased from 29.04 GB to 15.75 GB.

TABLE IV
DETAILED MATRIX MULTIPLICATION COMPARISON

real-sim	GFLOPs	Memory throughput (GB/s)	Compute Utilization
ThunderSVM	419.75	61.11	0.42
ours	1123.8	221.4	0.48
rcv1_test.binary			
ThunderSVM	425.65	55.41	0.41
ours	1187.18	206.93	0.51

TABLE V
ACCURACY COMPARISON WITH BOOSTSVM

Dataset	Accuracy		
	LibSVM	ThunderSVM	BoostSVM
gisette_scale	50%	50%	50%
real-sim	86.0%	86.0%	86.0%
ijcnn1.t	99.1%	99.1%	99.1%
epsilon	81.7%	81.7%	81.7%
rcv1_test.binary	97.8%	97.8%	97.8%
url_combined	98.6%	98.6%	98.6%

D. Accuracy Comparison

We compare the training results with LibSVM and ThunderSVM to ensure the correctness of our BoostSVM training results. Table V shows the accuracy of 2-fold cross-validation. We can see that the accuracy results are completely consistent. Since the change in the computation order during floating-point operations, the objective function values may have slight differences, which are generally less than 0.3%. But this does not affect the final accuracy. Therefore, LibSVM, ThunderSVM, and BoostSVM produce the same SVMs overall. This verifies the correctness of our optimization methods.

V. CONCLUSION

In this paper, we have optimized the training speed of GPU-based SVMs on large-scale datasets. The main bottleneck of SVM training lies in the computation of kernel functions, where the heavy random memory access causes performance degradation. Based on the analysis of non-zero element distribution in the SVM training dataset, we have proposed column density-based matrix partitioning and memory access optimization to reduce the number of random memory accesses and improve multiplication efficiency. Experimental results show that our method has a significant improvement compared with existing works. Especially on large-scale dense datasets, our methods surpass the state-of-the-art SMO-based SVM on GPUs by up to 52 times, while retaining the same predictive accuracy. Meanwhile, our matrix partitioning method is applicable to optimizing matrix multiplication in other scenarios.

References

[1] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 7, pp. 3523–3542, 2021.

[2] G. H. de Rosa and J. P. Papa, "A survey on text generation using generative adversarial networks," *Pattern Recognition*, vol. 119, p. 108098, 2021.

[3] S.-J. Sammut, M. Crispin-Ortuzar, S.-F. Chin, E. Provenzano, H. A. Bardwell, W. Ma, W. Cope, A. Dariush, S.-J. Dawson, J. E. Abraham *et al.*, "Multi-omic machine learning predictor of breast cancer therapy response," *Nature*, vol. 601, no. 7894, pp. 623–629, 2022.

[4] K. Vos, Z. Peng, C. Jenkins, M. R. Shahriar, P. Borghesani, and W. Wang, "Vibration-based anomaly detection using lstm/svm approaches," *Mechanical Systems and Signal Processing*, vol. 169, p. 108752, 2022.

[5] Z. Hou, X. Liu, Y. Cen, Y. Dong, H. Yang, C. Wang, and J. Tang, "Graphmae: Self-supervised masked graph autoencoders," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 594–604.

[6] D. Wu, Y. He, X. Luo, and M. Zhou, "A latent factor analysis-based approach to online sparse streaming feature selection," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 11, pp. 6744–6758, 2022.

[7] C. Morris, G. Rattan, S. Kiefer, and S. Ravanbakhsh, "Speqnets: Sparsity-aware permutation-equivariant graph networks," in *International Conference on Machine Learning*. PMLR, 2022, pp. 16 017–16 042.

[8] T. Joachims, "Svmlight: Support vector machine," *SVM-Light Support Vector Machine http://svmlight. joachims. org/, University of Dortmund*, vol. 19, no. 4, p. 25, 1999.

[9] 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, pp. 1–27, 2011.

[10] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *the Journal of machine Learning research*, vol. 9, pp. 1871–1874, 2008.

[11] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 797–801, 2018.

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

[13] S. Zhang, R. Shah, and P. Wu, "TensorSVM: accelerating kernel machines with tensor engine," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–11.

[14] A. Van Craen, M. Breyer, and D. Pflüger, "PLSSVM: A (multi-) gpgpu-accelerated least squares support vector machine," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 818–827.

[15] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[16] S. Shi, Q. Wang, and X. Chu, "Efficient sparse-dense matrix-matrix multiplication on gpus using the customized sparse storage format," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, 2020, pp. 19–26.

[17] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, pp. 273–297, 1995.

[18] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," 1998.

[19] R.-E. Fan, P.-H. Chen, C.-J. Lin, and T. Joachims, "Working set selection using second order information for training support vector machines." *Journal of machine learning research*, vol. 6, no. 12, 2005.

[20] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, pp. 293–300, 1999.

[21] S. Fine and K. Scheinberg, "Efficient SVM training using low-rank kernel representations," *Journal of Machine Learning Research*, vol. 2, no. Dec, pp. 243–264, 2001.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

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

[24] F. O. Catak and M. E. Balaban, "CloudSVM: training an SVM classifier in cloud computing systems," in *Pervasive Computing and the Networked World: Joint International Conference, ICPCA/SWS 2012, Istanbul, Turkey, November 28-30, 2012, Revised Selected Papers*. Springer, 2013, pp. 57–68.

[25] 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*, 2008, pp. 104–111.

[26] 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*, 2010, pp. 2–11.

[27] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for support vector machines," in *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*, vol. 164, 2011.

[28] A. Cotter, N. Srebro, and J. Keshet, "A GPU-tailored approach for training kernelized svms," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 805–813.

[29] J. Vaněk, J. Michálek, and J. Psutka, "A GPU-architecture optimized hierarchical decomposition algorithm for support vector machine training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3330–3343, 2017.

[30] J. A. Suykens, J. De Brabanter, L. Lukas, and J. Vandewalle, "Weighted least squares support vector machines: robustness and sparse approximation," *Neurocomputing*, vol. 48, no. 1-4, pp. 85–105, 2002.

[31] J. A. Suykens and J. Vandewalle, "Multiclass least squares support vector machines," in *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339)*, vol. 2. IEEE, 1999, pp. 900–903.

[32] J. A. Suykens, L. Lukas, and J. Vandewalle, "Sparse approximation using least squares support vector machines," in *2000 IEEE international symposium on circuits and systems (ISCAS)*, vol. 2. IEEE, 2000, pp. 757–760.

[33] T.-N. Do and V.-H. Nguyen, "A novel speed-up SVM algorithm for massive classification tasks," in *2008 IEEE International Conference on Research, Innovation and Vision for the Future in Computing and Communication Technologies*. IEEE, 2008, pp. 215–220.

[34] P. Drineas and M. W. Mahoney, "Approximating a gram matrix for improved kernel-based learning," in *Learning Theory: 18th Annual Conference on Learning Theory, COLT 2005, Bertinoro, Italy, June 27-30, 2005. Proceedings 18*. Springer, 2005, pp. 323–337.

[35] K. Zhang, L. Lan, Z. Wang, and F. Moerchen, "Scaling up kernel SVM on limited resources: A low-rank linearization approach," in *Artificial intelligence and statistics*. PMLR, 2012, pp. 1425–1434.

[36] NVIDIA, "cuSPARSE Library," 2022. [Online]. Available: https://developer.nvidia.com/cusparse

[37] ——, "cuBLAS Library," 2022. [Online]. Available: https://developer.nvidia.com/cublas