MASCOT: Fast and Highly Scalable SVM Cross-validation using GPUs and SSDs

Zeyi Wen [#], Rui Zhang [#], Kotagiri Ramamohanarao [#], Jianzhong Qi [#], Kerry Taylor [*]

[#] *University of Melbourne, Australia*

[#] {zeyi.wen, rui.zhang, kotagiri, jianzhong.qi}@unimelb.edu.au

[*] *Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia*

[*] kerry.taylor@csiro.au

***Abstract*—Cross-validation is a commonly used method for evaluating the effectiveness of Support Vector Machines (SVMs). However, existing SVM cross-validation algorithms are not scalable to large datasets because they have to (i) hold the whole dataset in memory and/or (ii) perform a very large number of kernel value computation. In this paper, we propose a scheme to dramatically improve the scalability and efficiency of SVM cross-validation through the following key ideas. (i) To avoid holding the whole dataset in the memory and avoid performing repeated kernel value computation, we precompute the kernel values and reuse them. (ii) We store the precomputed kernel values to a high-speed storage framework, consisting of CPU memory extended by solid state drives (SSDs) and GPU memory as a cache, so that reusing (i.e., reading) kernel values takes much lesser time than computing them on-the-fly. (iii) To further improve the efficiency of the SVM training, we apply a number of techniques for the extreme example search algorithm, design a parallel kernel value read algorithm, propose a caching strategy well-suited to the characteristics of the storage framework, and parallelize the tasks on the GPU and the CPU. For datasets of sizes that existing algorithms can handle, our scheme achieves several orders of magnitude of speedup. More importantly, our scheme enables SVM cross-validation on datasets of very large scale that existing algorithms are unable to handle.**

I. Introduction

Support Vector Machines (SVMs) are a popular supervised learning model for classification problems. To handle non-linearly separable data, SVMs use a kernel function (e.g., the Gaussian kernel function [1]) to map the data from their original space to a higher dimensional space where they may become linearly separable. Cross-validation is a commonly used method for evaluating the effectiveness of SVMs so as to identify suitable hyper-parameters such as kernel parameters. Figure 1 shows the overview of k-fold SVM cross-validation that repeats the training and classification for k times. During the training[1] using popular algorithms such as Sequential Minimum Optimization, the search for extreme training examples and the kernel value computation are repeated a number (denoted by t in Figure 1) of times until convergence. The kernel value computation with the time complexity of $\mathcal{O}(nd)$ in each iteration is the most time consuming operation, where n is the dataset cardinality and d is the data dimensionality. The k-fold cross-validation is an even more expensive operation with the time complexity of $\mathcal{O}(ktnd)$ and requires reading the dataset kt times. To avoid reading the whole dataset from the external memory in each training iteration, existing CPU-based cross-validation algorithms have to hold the whole dataset in memory and repeatedly perform the kernel value computation. Consequently, prohibitive computation is the main impediment to the scalability of CPU-based algorithms.

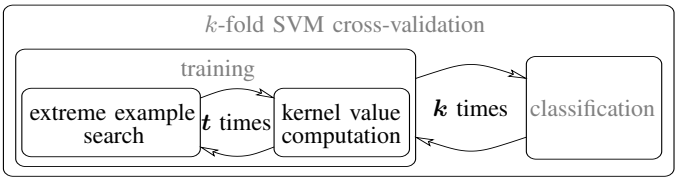


Fig. 1: k-fold SVM cross-validation

Some studies attempt to use Graphics Processing Units (GPUs) to improve the efficiency of the training [2]. However, the scalability of those algorithms is limited by the GPU memory size as they require to hold the whole dataset in the GPU memory, which means a space complexity of $\mathcal{O}(nd)$. Even using a high-end NVIDIA GPU, GTX 780, which has 3GB GPU memory and assuming 4 bytes to store a value, existing GPU-based algorithms can handle a dataset of only 40,000 examples with 20,000 dimensions (each dimension of an example is a value). They are far from being able to handle large datasets emerging in many new applications such as the YouTube Multiview dataset (120,000 examples of 1,000,000 dimensions), the Webspam dataset (350,000 examples of 16,609,143 dimensions) and the Gas Sensor dataset (18,000 examples of 1,950,000 dimensions)[2]. Note that it is inefficient for those GPU-based algorithms to store the datasets in the CPU memory, because frequent data transfers (kt times of the whole dataset) from the CPU memory to the GPU memory is too costly. In addition, some studies tries to improve the efficiency and scalability of the training using MapReduce [3] by producing approximated results. We will discuss in Section II that the MapReduce framework is not suited to the training. To summarize, the scalability of existing cross-validation algorithms has been seriously limited by expensive computation and the requirement of storing the whole dataset in memory.

With the advent of big data, performing cross-validation becomes prohibitive. However, there has been no study aiming at improving the scalability of the cross-validation and this is the first work dedicated to addressing this problem. To

[1]Without confusion, we omit "SVM" in the rest of this paper, similarly for SVM classification and SVM cross-validation.

[2]The datasets are found in LibSVM site and UCI repository.

approach this problem, we design a scheme that exploits modern hardware's high computation power (GPUs) and fast access (SSDs), so we call our scheme Modern hArdware enabled Svm CrOss-validaTion (MASCOT). Specifically, we make the following contributions:

- To avoid holding the whole dataset in the memory and avoid performing repeated kernel value computation, we precompute the kernel values and reuse them. We store the precomputed kernel values to a high-speed storage framework, consisting of CPU memory extended by SSDs and the GPU memory as a cache, so that reusing (i.e., reading) kernel values takes much lesser time than computing them on-the-fly.
- To further improve the efficiency of the training, we apply a number of techniques for the extreme example search algorithm, design a parallel kernel value read algorithm, propose a caching strategy well-suited to the characteristics of the storage framework, and design better distribution of the tasks on the GPU and the CPU.
- We conduct extensive experiments and the results show that MASCOT is scalable to large datasets while the existing algorithms either become extremely slow, or totally fail due to the limited memory size. For datasets of sizes that existing algorithms can handle, MASCOT achieves an order of magnitude of speedup over the state-of-the-art GPU-based algorithm and two orders of magnitude of speedup over the CPU-based algorithm. More importantly, MASCOT enables the cross-validation on datasets of very large scale that existing algorithms are unable to handle.

The rest of this paper is organized as follows. We discuss the related work in Section II, and present preliminaries in Section III. We describe MASCOT in Section IV. In Section V, we show detailed experimental results. We conclude the paper in Section VI.

II. Related Work

A. SVM Cross-validation

Athanasopoulos et al. [4] proposed the first GPU-based algorithm for accelerating the cross-validation. Their method is difficult to scale due to two reasons. First, the algorithm requires all the kernel values to be stored in the GPU memory, which is not feasible for large datasets. Second, their method does not make full use of the computational capability of the GPU since it is only used for kernel value precomputation while the CPU handles the rest of the work. A more recent study [5] uses the GPU to run multiple cross-validation tasks at the same time for improving the efficiency, but it does not improve the scalability of the cross-validation because the multiple cross-validation tasks require even more memory. Other studies [6], [7] use a technique called "alpha seeding" to improve the efficiency of the leave-one-out cross-validation which is a special case of the k-fold cross-validation when k equals to the dataset cardinality. Our study focuses on improving the scalability and efficiency using modern hardware.

B. SVM Training

Accelerating each training iteration: The techniques used in CPU-based algorithms for accelerating each training iteration can be adapted to GPU-based algorithms, so we focus our discussion on algorithms specially designed for GPUs. Catanzaro et al. [2] used the GPU to compute kernel values and update the weights of the training examples in each iteration of the training. A recent work [8] adopted the algorithm for semi-supervised SVM training. Catanzaro et al. also proposed an algorithm for the classification based on the GPU, which can be adapted for the classification phase of the cross-validation. Catanzaro et al.'s algorithms serve as our GPU-based baseline algorithm and we describe the details in Section III.

Other studies have different goals or settings from ours. For example, Cotter et al. [9] proposed a GPU-tailored approach for the training using a clustering technique. This approach is designed for sparse datasets which can be stored entirely in the GPU memory.

Accelerating convergence: Joachims [10] proposed a training algorithm using the cutting-plane approach to improve the efficiency, but this algorithm only applies to linear SVMs. "Pegasos" [11] is another training algorithm for linear SVMs. Osuna et al. [12] proposed a training algorithm based on decomposition, and Platt [13] improved Osuna et al.'s algorithm by proposing the *Sequential Minimal Optimization* (SMO) training algorithm. SMO is generic and widely used in implementations such as LibSVM [14], WEKA [15] and Catanzaro's GPU-based SVMs, so we use the SMO algorithm in our scheme.

Attempts to accelerate the training by MapReduce: MapReduce is not suited to the training because SVM training is a global optimization process. If we partition training examples, then each mapper can only obtain support vectors locally based on the examples in one partition. After the reduce phase, the locally obtained support vectors put together are unlikely to correspond to those of the globally optimal SVM. Hence, it is difficult for a single round of MapReduce based training process to meet the termination condition, leading to many rounds of MapReduce jobs and no guarantee of convergence. Previous work attempts using MapReduce to accelerate the training by sacrificing accuracy or by only handling some special cases. Specifically, Some studies [16], [17] attempts to improve the training efficiency by producing approximated results. Catak and Balaban [3] proposed algorithms used multiple rounds of the MapReduce training to gradually make the approximation more accurate until the globally optimal SVM is obtained, but this method does not guarantee convergence.

C. Kernel Value Caching

In the training, the same kernel values may be used in different iterations, so we may cache some kernel values and avoid reading them from the CPU memory or the SSD to the GPU memory. Popular SVM libraries such as SVM^{light} [18] and LibSVM adopt the Least Recently Used (LRU) replacement strategy for caching kernel values between different iterations. Our later analysis will show that LRU is not efficient for the training because it does not suit the kernel values' access pattern. Another caching strategy [19] replaces the kernel values of the examples with weights of 0 in the current

iteration. This replacement strategy requires a linear search to find the examples whose weights equal to 0, which is even less efficient than LRU.

III. Preliminaries

For ease of presentation, we focus our discussion on k-fold cross-validation for *binary* classification, although our techniques can be applied to multi-class classification and SVM regression [20] straightforwardly. The k-fold cross-validation evenly divides the dataset into k subsets. One subset is used as the test set $\mathcal{X}_v$ while the rest $(k-1)$ subsets together as the training set $\mathcal{X}_t$. The SVM is first trained using $\mathcal{X}_t$. Then the trained SVM is used to classify the test examples in $\mathcal{X}_v$ by predicting their labels. To obtain more reliable results, the above training-classification process is repeated for k times, where every subset is used as the test set in turn. The predicted labels are compared with their true labels to evaluate the effectiveness of the SVM. Next, we describe the training-classification process.

Training: A training example $\boldsymbol{x}_i$ is attached with an integer $y_i \in \{+1, -1\}$ as its label. A positive (negative) example is a training example with the label of $+1$ (-1). Given a set $\mathcal{X}$ of n training examples, the goal of training SVMs is to find a hyperplane that separates the positive and the negative examples in $\mathcal{X}$ with the maximum margin and meanwhile, with the minimum misclassification error on the training examples. The training is equivalent to solving the following optimization problem:

$$\begin{aligned} \underset{\boldsymbol{w},\boldsymbol{\xi},b}{\text{argmin}} \quad & \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i - b) \geq 1-\xi_i \\ & \xi_i \geq 0,\ \forall i \in \{1,...,n\} \end{aligned} \tag{1}$$

where $\boldsymbol{w}$ is the normal vector of the hyperplane, C is the penalty parameter, $\boldsymbol{\xi}$ is the slack variables to tolerant some training examples falling in the wrong side of the hyperplane, and b is the bias of the hyperplane.

To handle the non-linearly separable data, SVMs use a mapping function to map the training examples from the original data space to a higher dimensional data space where the data may become linearly separable. The optimization problem 1 can be rewritten to a dual form [21] where mapping functions can be replaced by kernel functions [1] which make the mapping easier. The optimization problem in the dual form is shown as follows.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & F(\boldsymbol{\alpha}) = \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\boldsymbol{\alpha}^{\boldsymbol{T}}\boldsymbol{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \forall i \in \{1,...,n\} \\ & \sum_{i=1}^{n} y_i\alpha_i = 0 \end{aligned} \tag{2}$$

where $F(\boldsymbol{\alpha})$ is the objective function; $\boldsymbol{\alpha} \in \mathbb{R}^n$ is a weight vector, where α_i denotes the *weight* of $\boldsymbol{x}_i$; C is the penalty parameter; $\boldsymbol{Q}$ is a positive semi-definite matrix, where $\boldsymbol{Q} = [Q_{ij}]$, $Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a kernel value computed from a kernel function (e.g., Gaussian kernel, $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = exp\{-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2\}$). All the kernel values together form an $n \times n$ kernel matrix shown as follows.

$$\begin{pmatrix} K(\boldsymbol{x}_1,\boldsymbol{x}_1) & K(\boldsymbol{x}_1,\boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_1,\boldsymbol{x}_n) \\ K(\boldsymbol{x}_2,\boldsymbol{x}_1) & K(\boldsymbol{x}_2,\boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_2,\boldsymbol{x}_n) \\ \vdots & \vdots & \vdots & \vdots \\ K(\boldsymbol{x}_i,\boldsymbol{x}_1) & K(\boldsymbol{x}_i,\boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_i,\boldsymbol{x}_n) \\ \vdots & \vdots & \vdots & \vdots \\ K(\boldsymbol{x}_n,\boldsymbol{x}_1) & K(\boldsymbol{x}_n,\boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_n,\boldsymbol{x}_n) \end{pmatrix}$$

The goal of the training translates to finding a weight vector $\boldsymbol{\alpha}$ that maximizes the value of the objective function $F(\boldsymbol{\alpha})$. Here, we describe a popular training algorithm, the Sequential Minimal Optimization (SMO) algorithm [13]. It iteratively improves the weight vector until the optimal condition of the SVM is met. The optimal condition is reflected by an *optimality indicator vector* $\boldsymbol{f} = \langle f_1, f_2, ..., f_n\rangle$ where f_i is the optimality indicator for the i^{th} example $\boldsymbol{x}_i$ and f_i can be obtained using the following equation: $f_i = \sum_{j=1}^{n}\alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) - y_i$. In each iteration, the SMO algorithm has the following three steps:

Step 1: Search two extreme examples, denoted by $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, which have the maximum and minimum optimality indicators, respectively. It has been proven [22], [23] that the indexes of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by u and l respectively, can be computed by the following equations.

$$u = \underset{i}{\text{argmin}}\{f_i | \boldsymbol{x}_i \in \mathcal{X}_{upper}\} \tag{3}$$

$$l = \underset{i}{\text{argmax}}\{\frac{(f_u - f_i)^2}{\eta_i} | f_u < f_i, \boldsymbol{x}_i \in \mathcal{X}_{lower}\} \tag{4}$$

where

$$\begin{aligned} \mathcal{X}_{upper} &= \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3, \\ \mathcal{X}_{lower} &= \mathcal{X}_1 \cup \mathcal{X}_4 \cup \mathcal{X}_5; \end{aligned}$$

and

$$\begin{aligned} \mathcal{X}_1 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, 0 < \alpha_i < C\}, \\ \mathcal{X}_2 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = 0\}, \\ \mathcal{X}_3 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = C\}, \\ \mathcal{X}_4 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = C\}, \\ \mathcal{X}_5 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = 0\}; \end{aligned}$$

and $\eta_i = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_i)$; f_u and f_l denote the optimality indicators of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, respectively.

Step 2: Improve the weights of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by α_u and α_l, by updating them using Equations 5 and 6.

$$\alpha_l' = \alpha_l + \frac{y_l(f_u - f_l)}{\eta} \tag{5}$$

$$\alpha_u' = \alpha_u + y_l y_u(\alpha_l - \alpha_l') \tag{6}$$

where $\eta = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_l, \boldsymbol{x}_l) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_l)$. To guarantee the update is valid, when α_u' or α_l' exceeds the domain of $[0, C]$, α_u' and α_l' are adjusted into the domain.

Step 3: Update the optimality indicators of all examples. The optimality indicator f_i of the example $\boldsymbol{x}_i$ is updated to f_i'

using the following formula:

$$f_i' = f_i + (\alpha_u' - \alpha_u) y_u K(\boldsymbol{x}_u, \boldsymbol{x}_i) + (\alpha_l' - \alpha_l) y_l K(\boldsymbol{x}_l, \boldsymbol{x}_i) \quad (7)$$

SMO repeats the above steps until the optimal condition is met, i.e., $f_u \geq f_{max}$, where

$$f_{max} = max\{f_i | \boldsymbol{x}_i \in \mathcal{X}_{lower}\} \quad (8)$$

After the optimal condition is met, we obtain the $\boldsymbol{\alpha}$ values which corresponding to the optimal hyperplane and the SVM with these $\boldsymbol{\alpha}$ values is considered *trained*. Algorithm 1 summarizes the whole training process. In Algorithm 1, $\boldsymbol{\mathcal{K}}_u$ and $\boldsymbol{\mathcal{K}}_l$ correspond to the u^{th} and the l^{th} rows of the kernel matrix, respectively.

Algorithm 1: The SMO algorithm

Input: a training set $\mathcal{X}$ of n instances with labels $\boldsymbol{y}$
Output: a weight vector $\boldsymbol{\alpha}$

```
for i ← 1 to n do        /* initialize α and f */
    α_i ← 0, f_i ← −y_i
repeat
    search for f_u and u using Equation 3;
    compute kernel values K_u ;          /* u^th row */
    search for f_l and l using Equation 4;
    compute kernel values K_l ;          /* l^th row */
    update α_u and α_l using Equations 5 and 6;
    update f using Equation 7;
    search for f_max using Equation 8;
until f_u ≥ f_max
```

Classification: After the training, the trained SVM is used in the classification phase to predict the label of every test example. The label of a test example $\boldsymbol{x}_j$, denoted by y_j, is predicted by the following formula:

$$y_j = \text{sgn}\left(\sum_{i=1}^{n} y_i \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_j) + b\right) \quad (9)$$

where b is the bias of the hyperplane of the trained SVM. The training examples with their weights greater than zero are called *support vectors*, which are used to predict the labels of test examples.

A. GPUs and SSDs

In recent years, as Graphics Processing Units (GPUs) have become more programmable, many general purpose computing applications [24], [25] have benefited from their high-performance. In the NVIDIA *Compute Unified Device Architecture* (CUDA) [26], GPU threads are grouped into blocks which further form grids as shown in Figure 2. In a block, every 32 threads are grouped into a warp and executed in parallel. Due to the hierarchy of the threads, how well an algorithm is parallelized determines the performance of the algorithm. As we can see from Figure 2, there are registers, shared memories and a global memory on a GPU. Different types of memories have vastly different access latency. For example, access to the global memory is over 7 times slower than to the shared memory, while access to the shared memory is about 6 times slower than to the registers [27].

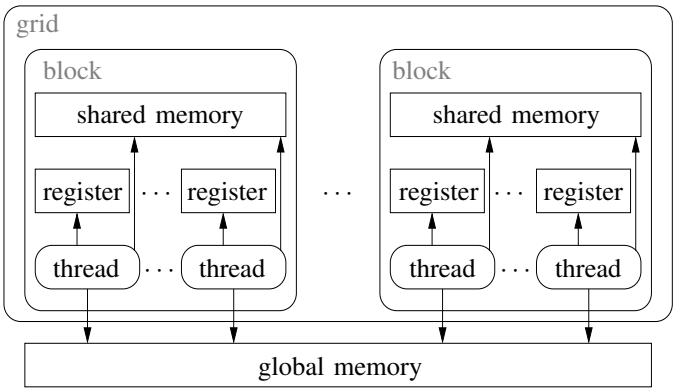


Fig. 2: Threads and memory on a GPU

Solid state drives (SSDs) are emerging high-performance storage devices. They have no moving mechanical components, which distinguish them from hard disk drives (HDDs). SSDs' random access and sequential access are typically more than 10 times and 3 times faster than those of HDDs, respectively. In addition, the SSD consists of a number of blocks which are further divided into pages. *Different blocks can be read simultaneously*, which significantly accelerates the read speed. For example, 20 pages can be read in parallel on the Intel X25M SSD [28].

B. The State-of-the-art GPU-based SVM Cross-validation Algorithm

Catanzaro et al. [2] proposed a GPU-based training and classification algorithm, which we refer to as **gSVM** in the rest of the paper. In gSVM the SMO training algorithm is parallelized as follows. In Step 1, the cost of the search for extreme examples is dominated by searching minimal/maximal values to obtain u, l and f_{max} according to Equations 3, 4 and 8. This search operation is done by a technique called *GPU reduction operation*. Specifically, to search the minimal (maximal) value from a vector of n numbers, the GPU reduction operation starts with creating n threads. Each thread loads a number from the global memory and writes it to the shared memory. Then half of the threads (i.e., $n/2$ threads) are used and each thread reads two of the n loaded numbers from the shared memory, compares them, and writes the smaller (larger) number back to the shared memory, while the larger (smaller) number is discarded. After this round, $n/2$ numbers remain in the shared memory. In the next round, $n/4$ of the threads will read the $n/2$ numbers, compare them and write back $n/4$ numbers. This process repeats until there is only one thread left, which produces the minimal (maximal) value. Note that this process requires storing n values in the shared memory. This limits the number of active threads especially when n is large, since the size of shared memory is small in the GPU. In Step 2, one GPU thread is used to compute the two updated weights α_u' and α_l' according to Equations 5 and 6. In Step 3, n threads are used to update the optimality indicators of the n training examples, one for each optimality indicator. The three steps are repeated until the optimal condition is met. In each iteration, n optimality indicators are updated using Equation 7. When updating each optimality indicator, gSVM computes the two kernel values in the right hand side of Equation 7 on-the-fly and hence needs to compute $2n$ kernel values in each iteration.

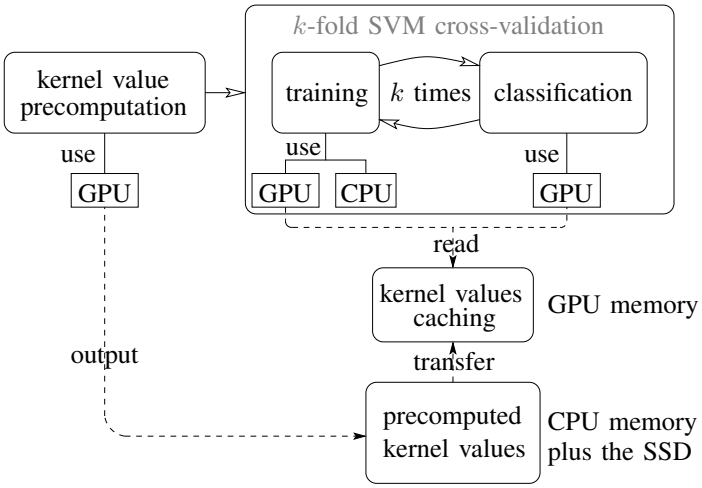


Fig. 3: The MASCOT scheme

To reduce the cost of the kernel value computation, gSVM uses the LRU replacement strategy for kernel value caching.

In the classification phase, the algorithm computes Equation 9 for every example to be classified in parallel. The computation of the equation for one example is further parallelized into n threads, where each thread computes one term of the summation. It has been reported that gSVM outperforms the CPU-based cross-validation by an order of magnitude [2], and gSVM is the state-of-the-art SVM cross-validation algorithm. Therefore, gSVM is used as our baseline for GPU-based algorithm. However, it has the following drawbacks.

Drawback 1: gSVM repeatedly computes kernel values on-the-fly, which requires the whole dataset to be stored in the GPU memory. As a result, it cannot process datasets larger than the GPU memory.

Drawback 2: gSVM uses a straightforward algorithm for searching extreme examples in Step 1 of the training phase, which consumes more shared memory and limits the number of active GPU threads.

Drawback 3: gSVM uses a general purpose cache strategy which does not exploit the kernel values' access pattern.

IV. The MASCOT Scheme

Towards highly scalable and efficient SVM cross-validation, we propose a scheme called Modern hArdware enabled Svm CrOss-validaTion (MASCOT). Figure 3 gives an overview of MASCOT which consists of kernel value precomputation, training and classification. The GPU performs the kernel value precomputation and stores the precomputed kernel values to the CPU memory extended by the SSD. The GPU memory is used to cache kernel values that are reused in different iterations of the training. Next, we explain the three phrases of MASCOT in detail.

A. Precomputing, Storing and Obtaining Kernel Values

As discussed in Section III, the kernel matrix consists of the kernel values of all the examples in $\mathcal{X}$. The i^{th} row $\boldsymbol{\mathcal{K}}_i = \langle K(\boldsymbol{x}_i, \boldsymbol{x}_1),\ K(\boldsymbol{x}_i, \boldsymbol{x}_2),\ ...,\ K(\boldsymbol{x}_i, \boldsymbol{x}_n)\rangle$ of the matrix corresponds to all the n kernel values of the example $\boldsymbol{x}_i$. Since $\mathcal{X}$ contains n examples $\langle \boldsymbol{x}_1,\ \boldsymbol{x}_2,\ ...,\ \boldsymbol{x}_n\rangle^T$, the kernel matrix is an $n \times n$ matrix.

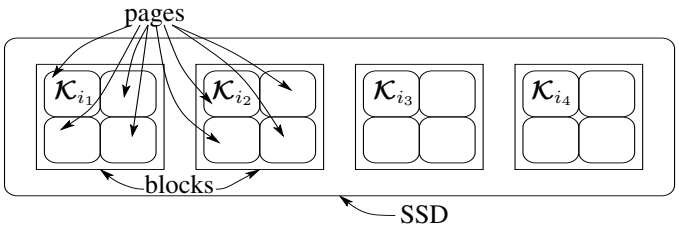


Fig. 4: Storing the row $\boldsymbol{\mathcal{K}}_i$ to an SSD

1) Computing Kernel Values: We use the GPU to precompute the kernel matrix. If the dataset is too large to be held in the GPU memory, we partition the dataset into subsets and compute sub-matrices using those subsets. The sub-matrices together form the kernel matrix.

2) Storing Kernel Values: The precomputed kernel matrix is stored row after row sequentially in the CPU memory extended by the SSD. When we need to use the kernel values in the training and the classification, we simply read the needed ones from the CPU memory or the SSD. Thus, we avoid repeated computation of the kernel values. Note that in each iteration of the training, two rows (i.e., the u^{th} and the l^{th} rows in the kernel matrix) are needed according to Equation 7; in the classification for an example $\boldsymbol{x}_j$, one row (i.e., the j^{th} row in the kernel matrix) is needed according to Equation 9.

To enable fast read of the kernel values from the precomputed kernel matrix, we use the following two techniques. First, the portion of the precomputed kernel matrix stored in the CPU memory can be directly read by the GPU using a technique called "Zero Copy memory" [26]. Zero Copy is a hardware technique that allows the GPU to use the CPU memory as if the CPU memory was the GPU memory. This way the communication cost between the CPU and the GPU is reduced.

Second, the portion of the precomputed kernel matrix beyond the CPU memory size is stored in the SSD; the kernel values are stored in a way that they can be read in parallel to make use of SSDs' multi-channel feature (cf. Section III-A). Specifically, we divide a row of the kernel matrix into partitions, so that each partition can be stored in an SSD page. The partitions are stored to a number of SSD pages, each in a different SSD block. Hence, different partitions of the row can be read from the SSD blocks simultaneously. Figure 4 shows an example of storing the i^{th} row $\boldsymbol{\mathcal{K}}_i$ to an SSD; the row $\boldsymbol{\mathcal{K}}_i$ is divided into four partitions which are stored to four SSD pages in different SSD blocks. The pseudo-code of storing the kernel matrix is shown in Algorithm 2. Initially, the number of pages to store a row of the kernel matrix is calculated using the number of kernel values in a row and the number of kernel values that an SSD page can hold (line 1). Then, given a row index i of the kernel matrix, a number of SSD pages in different SSD blocks are identified to store that row (lines 3 and 4). Finally, the row is stored to the identified SSD pages in the corresponding SSD blocks (lines 5 and 6).

3) Obtaining Kernel Values: Each training iteration requires two rows from the kernel matrix. As we have precomputed the kernel matrix, we just need to read two rows from it. Based on a row index, we know whether the row is stored in the CPU memory. When the needed rows are in the CPU memory, the GPU directly reads the kernel values using the Zero-copy technique. When the needed rows are in the SSD,

Algorithm 2: Storing the precomputed kernel matrix

Input: An $n \times n$ kernel matrix,
p_s: the number of kernel values in an SSD page,
b_s: the number of pages in an SSD block.
Output: A parallelly accessible kernel matrix $\mathcal{K}$

```
n_p ← ⌈n / p_s⌉                          /* # of pages for a row */
for i ← 1 to n do                        /* each row of K */
    b_id ← (i / b_s) · n_p               /* 1st block for row i */
    p_id ← i mod b_s                     /* page id in a block */
    for j ← 1 to n_p do                  /* store to blocks */
        store p_s values to page p_id of the (b_id + j)th block
```

we first read the kernel values from the SSD in parallel and then pass them together to the GPU.

We call the process of reading kernel values from the SSD "parallel kernel value read" which works as follows. Similar to the process of storing the kernel matrix, initially the parallel kernel value read algorithm calculates the number of SSD pages to read for obtaining a row of the kernel matrix. Then based on the index of the row, we can identify a set of SSD page indexes. Next, we create a number of CPU threads and each CPU thread is assigned to read a number of SSD pages that contain a partition of the row. Lastly, all the read results are put together and transferred to the GPU memory.

Addressing Scalability and Drawback 1 of gSVM: After the precomputation, the dataset $\mathcal{X}$ is no longer used in the cross-validation and we do not need any GPU memory to hold it, and hence we overcome *Drawback 1* of gSVM. As each training iteration only requires two rows ($2n$ kernel values) of the kernel matrix and the classification requires one row (n kernel values) of the kernel matrix to be held in the GPU memory, the space complexity of MASCOT is $\mathcal{O}(n)$ in terms of the GPU memory usage. Note that the space complexity is irrelevant to the data dimensionality. This is an advantage of our method since the big datasets we have these days tend to have thousands to millions of dimensions. As a comparison to gSVM which has a space complexity of $\mathcal{O}(nd)$, we show the largest datasets that gSVM and MASCOT can handle in Table I. Here we assume that gSVM uses a high-end GPU, GTX 780, which has 3GB GPU memory and MASCOT uses a 4TB SSD. We assume 4 bytes to store a value. As we see, MASCOT can handle much larger datasets than gSVM, especially when the dimensionality is high. The reason for MASCOT's high scalability is that we shift the space constraint from the size of the GPU memory to the size of the SSD which is much larger. Note that we can install more SSDs to a computer to handle even larger datasets while the GPU memory is not extendable.

B. Training

For improving the efficiency of the training, we describe a number of techniques for the extreme example search algorithm in Section IV-B1, propose a kernel value caching strategy which better suits the kernel values' access pattern in Section IV-B2 and present techniques better distribute the tasks between the GPU and the CPU in Section IV-B3.

1) Improving Extreme Example Search: In Step 1 of SMO, the search for two extreme examples is essentially the minimal/maximal value search to obtain u, l and f_{max}. We use the following three techniques for improving the efficiency of the search.

TABLE I: Scalability comparison

dimensionality	size of datasets that can be handled	
	MASCOT	gSVM
1,000	1,000,000	805,300
10,000	1,000,000	80,530
100,000	1,000,000	8,053
1,000,000	1,000,000	805

Reducing the shared memory consumption. When loading values from the global memory to the shared memory at the beginning of the search, we let each thread load a number m of elements instead of one. Each thread computes and maintains its local minimal (maximal) value in one of its registers. Then each thread writes its local minimal (maximal) value to the shared memory. Compared with the straightforward reduction operation which requires storing n values in the shared memory, our method only needs to store $\lceil n/m \rceil$ values.

Reducing accesses to the shared memory. After the values are loaded from the global memory to the shared memory, the GPU reduction operation starts. As we discussed in Section III-B, in each round of the reduction, an active GPU thread requires reading two values from the shared memory and writing back the smaller (larger) value to the shared memory. We can reduce read and write operations to the shared memory using registers which are about 6 times faster [27] than the shared memory. Each thread maintains its local minimal (maximal) value in a register in each round of the reduction. In the next round, a thread only needs to read one value from the shared memory, compare the value with its local minimal (maximal) value and write the smaller value back to the register as its new local minimal (maximal) value. Thus, the number of read operations to the shared memory is reduced by half. The write operation to the shared memory only happens when a thread becomes inactive. Inactive threads write their local minimal (maximal) values back to the shared memory so that active threads in the next round can read the value to search the global minimal (maximal) value. Active threads do not need to perform the write operations and hence the number of write operations is significantly reduced.

Early thread termination. After each round of the reduction, half of the active threads will no longer be used. We terminate them to save the cost of rescheduling them.

These techniques reduce the shared memory consumption, reduce accesses to the shared memory and improve the utilization of threads. Hence, **we overcome Drawback 2 of gSVM**.

2) Caching Strategy: The same row of kernel values is often used multiple times during the training and may be cached for reuse. Previous studies have simply used LRU replacement strategy for kernel value caching. Here we analyze the kernel values' access pattern during the training and propose a caching strategy that better suits the access pattern.

First, we show that the optimality indicators of the examples in $\mathcal{X}_{upper}$ increases while those of the examples in $\mathcal{X}_{lower}$ decreases every time they are updated (using Equation 7). As discussed in Section III, in each iteration of the training two

training examples $\boldsymbol{x}_u \in \mathcal{X}_{upper}$ and $\boldsymbol{x}_l \in \mathcal{X}_{lower}$ are used. The kernel values of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$ corresponding to the u^{th} row $\langle K(\boldsymbol{x}_u, \boldsymbol{x}_1)$, $K(\boldsymbol{x}_u, \boldsymbol{x}_2)$, ..., $K(\boldsymbol{x}_u, \boldsymbol{x}_n)\rangle$ and the l^{th} row $\langle K(\boldsymbol{x}_l, \boldsymbol{x}_1)$, $K(\boldsymbol{x}_l, \boldsymbol{x}_2)$, ..., $K(\boldsymbol{x}_l, \boldsymbol{x}_n)\rangle$ in the kernel matrix, respectively, are used to update all the optimality indicators. The optimality indicator f_i of the example $\boldsymbol{x}_i$ is updated to f'_i, which can be rewritten as Equation 10 using Equations 5 and 6.

$$f'_i = f_i + \frac{f_u - f_l}{\eta}\delta, \quad \delta = K(\boldsymbol{x}_l, \boldsymbol{x}_i) - K(\boldsymbol{x}_u, \boldsymbol{x}_i) \tag{10}$$

Based on the definition of η, it can be proven that η is always greater than 0 [23]. According to Equation 4, f_u is smaller than f_l, so we have $f_u - f_l < 0$. Therefore, the value $(f_u - f_l)/\eta$ is a negative constant for all the optimality indicators in the same iteration. How much f'_i changes over f_i is negatively proportional to δ. As the kernel function approximates the similarity between two examples [29], δ indicates whether $\boldsymbol{x}_i$ is more similar to $\boldsymbol{x}_l$ or to $\boldsymbol{x}_u$. Specifically, if the example $\boldsymbol{x}_i$ is in $\mathcal{X}_{lower}$ ("lower" side of the current hyperplane), then $\boldsymbol{x}_i$ tends to be more similar to $\boldsymbol{x}_l$ than to $\boldsymbol{x}_u$; similarly, if $\boldsymbol{x}_i$ is in $\mathcal{X}_{upper}$ ("upper" side of the current hyperplane), then $\boldsymbol{x}_i$ tends to be more similar to $\boldsymbol{x}_u$ than to $\boldsymbol{x}_l$. In summary, the following inequalities are often true:

$$\delta = \begin{cases} K(\boldsymbol{x}_l, \boldsymbol{x}_i) - K(\boldsymbol{x}_u, \boldsymbol{x}_i) > 0, \text{ if } \boldsymbol{x}_i \in \mathcal{X}_{lower} \\ K(\boldsymbol{x}_l, \boldsymbol{x}_i) - K(\boldsymbol{x}_u, \boldsymbol{x}_i) < 0, \text{ if } \boldsymbol{x}_i \in \mathcal{X}_{upper} \end{cases} \tag{11}$$

According to Equation 10 and Inequality 11, the values of the optimality indicators of the examples that are in the "upper" side of the current hyperplane increase every time they are updated. On the contrary, those of the examples in the "lower" side of the current hyperplane decrease every time they are updated.

Second, we show that the optimality indicator of $\boldsymbol{x}_u$ increases almost the most among all the optimality indicators of the examples in $\mathcal{X}_{upper}$, and the optimality indicator of $\boldsymbol{x}_l$ decreases almost the most among all the optimality indicators of the examples in $\mathcal{X}_{lower}$ in the current iteration. Given an example $\boldsymbol{x}_i \in \mathcal{X}_{upper}$ the more similar $\boldsymbol{x}_i$ is to $\boldsymbol{x}_u$, the larger $K(\boldsymbol{x}_u, \boldsymbol{x}_i)$ is because kernel functions approximate the similarity between two examples. Among all the examples in $\mathcal{X}_{upper}$, when $\boldsymbol{x}_i$ equals $\boldsymbol{x}_u$, $K(\boldsymbol{x}_l, \boldsymbol{x}_i) - K(\boldsymbol{x}_u, \boldsymbol{x}_i)$ has nearly the smallest value because $K(\boldsymbol{x}_l, \boldsymbol{x}_i)$ has nearly the smallest value when $\boldsymbol{x}_i = \boldsymbol{x}_u$ and $K(\boldsymbol{x}_u, \boldsymbol{x}_i)$ has almost the largest value when $\boldsymbol{x}_i = \boldsymbol{x}_u$ (recall that kernel value approximates the similarity between two examples and $\boldsymbol{x}_l$ is nearly the most dissimilar one to $\boldsymbol{x}_u$). Therefore, when $\boldsymbol{x}_i$ equals $\boldsymbol{x}_u$, δ reaches its smallest value approximately according to Inequality 11. As a result, f'_u increases mostly upon update among all the optimality indicators of the examples in $\mathcal{X}_{upper}$ in the same iteration. Similarly, we can derive that f'_l decreases mostly among all the optimality indicators of the examples in $\mathcal{X}_{lower}$.

Third, we show that the examples are used in a quasi-round-robin manner. The changes on the optimality indicators of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$ are the most significant. As we will show below, $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$ are unlikely to be used as the extreme examples in the near future of the training. As f_u increases the most among all the optimality indicators of the examples in $\mathcal{X}_{upper}$, it could possibly have the largest value of the optimality indicator in the next iteration. As a result, $\boldsymbol{x}_u$ may not be used again until all the other example in $\mathcal{X}_{upper}$ are used. This analysis applies to the update of f_l. Also, the analysis applies to each iteration of the training. Therefore, the examples are used in a quasi-round-robin manner and hence the kernel values are accessed in a quasi-round-robin manner.

The kernel value's access pattern, i.e., the quasi-round-robin access, is similar to sequentially scan all the kernel values for multiple times. For such access pattern, no caching strategy will increase the probability of hits compared with caching a fixed part of the kernel values. LRU is ill-suited to this access pattern since LRU caches recently used kernel values, which are actually the least possible ones to be accessed again in the near future. Since two rows of the kernel values in the kernel matrix are used together in each iteration as discussed in Section 4.1, we have used a simple caching strategy of replacing the row with the minimum row index in the kernel matrix when the cache is full. We call our caching replacement strategy **LAT** since our strategy replaces the row with **l**ongest **a**ccess **t**ime. LAT effectively caches the last part of the kernel matrix. It guarantees that a fixed part of the kernel matrix is cached and reused. As mentioned in Section 4.1 we store the rows with large row indexes in the kernel matrix in the SSD, so LAT favourites to cache the kernel values that are stored in the SSD. Compared with caching kernel values stored in the CPU memory, caching kernel values stored in the SSD saves more data transfer time. As our caching strategy is congruent with the kernel values' access pattern and well-suited to our storage framework, hence **we overcome Drawback 3 of gSVM**.

3) Distributing tasks on GPUs and CPUs: During the training, some operations (e.g., search extreme examples) can be parallelized while others are not parallizable. Hence, making full use of GPUs' high parallel computing capability and CPUs' high serial computing capability are important for achieving an efficient training algorithm. Apart from using the GPU to perform computation, we have two techniques for making full used of the CPU.

Parallelizing tasks on the CPU and the GPU: Step 2 updates the two weights α_u and α_l according to Equations 5 and 6. Updating on α_u and α_l are not parallelizable, as updating α_u requires the updated value of α_l. It is not helpful to run them on the GPU. Therefore, we run Step 2 on the CPU. Meanwhile, we prefetch the kernel values which are required by Step 3 from the CPU memory to the GPU memory to reduce the latency.

Using the CPU to compute values shared to GPU threads: Step 3 updates all the optimality indicators using Equation 7. This Equation contains five terms: $(\alpha'_u - \alpha_u)y_u$, $(\alpha'_l - \alpha_l)y_l$, f_i, $K(\boldsymbol{x}_u, \boldsymbol{x}_i)$ and $K(\boldsymbol{x}_l, \boldsymbol{x}_i)$. The first two terms remain the same when computing all the optimality indicators in the current iteration. Hence, we compute the two terms by the CPU and pass them to the GPU, instead of computing the two terms in every GPU thread.

C. Classification in the Cross-validation

The classification also benefits from the precomputed kernel matrix. According to Equation 9, we need kernel values (e.g., $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$) to predict the labels of the test examples. As $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ equals $K(\boldsymbol{x}_j, \boldsymbol{x}_i)$, we have two ways to read those kernel values: (i) reading all the rows of the kernel matrix corresponding to the support vectors (e.g., read rows

TABLE II: Datasets and kernel parameters

dataset	cardinality	dimension	C	γ
Adult	32,561	123	100	0.5
Epsilon	120,000	2,000	0.01	1
Gisette	6,000	5,000	100	0.382
MNIST	60,000	780	10	0.125
RCV1	20,242	47,236	100	0.125
Webdata	49,749	300	64	7.8125

with index respected to i), and (ii) reading all the rows of the kernel matrix corresponding to the test examples (e.g., read rows with index respected to j). If the number of support vectors is smaller (greater) than the number of test examples, then we read all rows of the kernel matrix corresponding to the support vectors (test examples). Thus, we can read less rows of the kernel matrix and hence read less kernel values.

D. Discussion

MASCOT can easily scale to a number (denoted by m) of machines with GPUs, since the kernel matrix precomputation, the training and the classification can be parallelized.

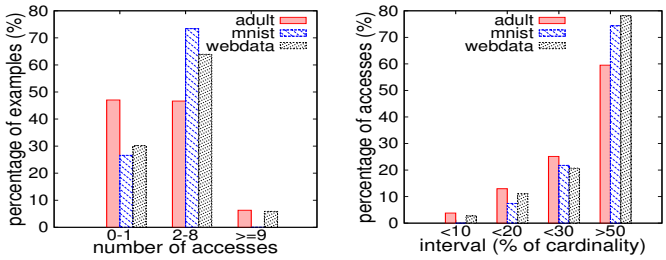
For the kernel matrix precomputation, we can decompose the whole kernel matrix into sub-matrices, and each machine computes all the kernel values of a sub-matrix independently.

The training can be parallelized as follows. We divide the optimality indicator $\boldsymbol{f}$ into m parts $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, ..., $\boldsymbol{p}_m$ where $\boldsymbol{p}_i = \langle f_{(i-1)z+1}, f_{(i-1)z+2}, ...f_{iz} \rangle$ and z is the number of the optimality indicators in each part. The i^{th} machine stores $\boldsymbol{p}_i$. For Step 1 (i.e., the search for minimum or maximum value), each machine obtains its local extreme indicator and reports it to a "Master" machine. The Master machine then can obtain the global extreme indicator. For Step 2 (i.e., improving the weights) which is not computationally expensive, the Master machine calculates the updated weights α'_u and α'_l. For Step 3 (i.e., updating $\boldsymbol{f}$), the i^{th} machine obtains the weights α'_u and α'_l from the Master machine and independently updates $\boldsymbol{p}_i$ according to Equation 7. Furthermore, the i^{th} machine only needs to store z columns, i.e., the $(iz-z+1)^{th}$ to iz^{th} columns of the kernel matrix. This is because only the $(iz-z+1)^{th}$ to iz^{th} kernel values of the u^{th} row and the l^{th} row are used to update $\boldsymbol{p}_i$ at each training iteration. As each machine only needs to store z columns of the kernel matrix, MASCOT can handle problems with an even larger kernel matrix using multiple machines.

The classification essentially computes a sum of n values according to Equation 9. Each machine sums n/m values, and reports the local sum to the Master machine. Then, the Master computes the predicted label y_j.

V. Experimental Study

We performed extensive experiments to evaluate the performance of MASCOT. The caching strategy, the parallel kernel value read technique and the improved extreme example search algorithm can be evaluated independently, so we first study the effects of these three techniques. The other techniques are inseparable from the MASCOT scheme. Therefore, we evaluate them as a whole in Section V-D when studying



(a) # accesses to examples (b) Two accesses to an example

Fig. 5: Kernel matrix access pattern

the scalability and efficiency of MASCOT. We conducted the experiments on a computer running Linux with a Xeon E5-2643 CPU, 16GB CPU memory, a 240GB SSD and a GTX460 GPU with 768MB memory. We used six datasets from the UCI Machine Learning Repository [3] and LibSVM site [4]. Table II gives the details of the datasets and parameters of Gaussian kernel. Following common practice, we set k to 10 and hence perform 10-fold cross-validation and set the page size of the SSD to 4KB. MASCOT and gSVM are implemented in CUDA. We also compare the efficiency of MASCOT with a popular CPU-based SVM cross-validation program, LibSVM, which is from LibSVM site and is implemented in C++.

A. Performance of the Caching Strategy

Due to the limited space, we only present and discuss the results on Adult, Webdata and MNIST datasets. The experiments on other datasets showed similar results which are omitted. Here we first show the access pattern of the kernel matrix and then verify the effectiveness of our kernel value caching strategy LAT.

Access pattern of the kernel matrix: Figure 5a shows the access frequency of the training examples. We can see that a major portion of the dataset is accessed for more than once. As accessing a training example indicates accessing to its corresponding kernel values, therefore an effective kernel value caching algorithm could improve the efficiency of the training. Figure 5b shows the number of iterations between two consecutive accesses to a training example. For most of the training examples (60% to 80%), the access interval between two consecutive accesses to them is more than half of the dataset cardinality. Therefore, recently used examples are unlikely to be used in the near further and hence LRU is not a suitable caching strategy for kernel value caching in the training. This also indicates that the kernel matrix is accessed in a quasi-round robin manner and confirms our analysis in Section IV-B.

We also conducted the experiments to count the number of accesses to different partitions of a dataset by evenly dividing the dataset into 10 partitions. The result shows that the numbers of accesses to different partitions are very similar. Hence, caching any partition will produce similar hit ratio.

Hit ratio comparison: Figure 6 shows the hit ratio comparison between LRU and our caching strategy LAT. On all the datasets, LAT significantly outperforms LRU.

[3] archive.ics.uci.edu/ml/datasets.html
[4] www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets/

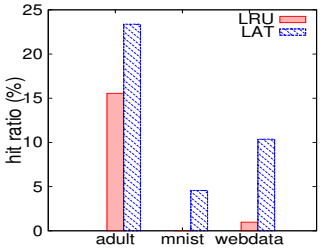


Fig. 6: Caching strategies

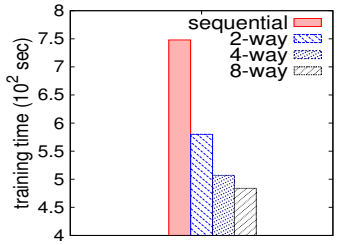


Fig. 7: Parallel kernel value read

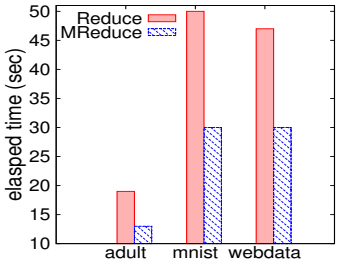
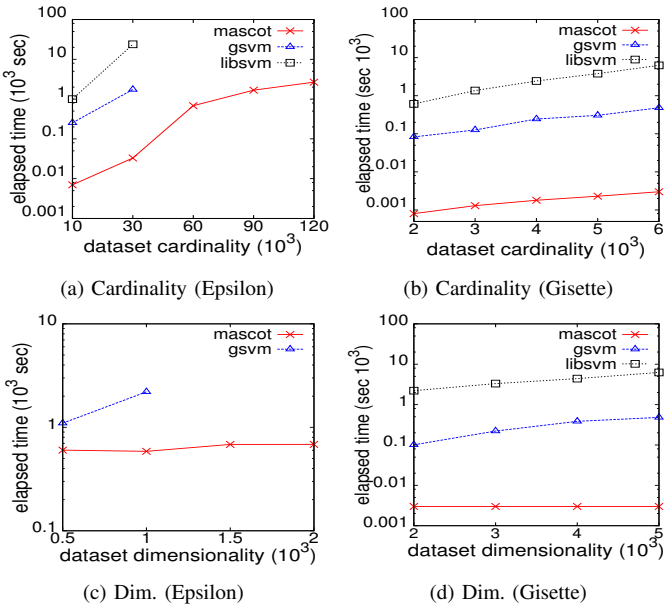


Fig. 8: Efficiency of search algorithms



(a) Cardinality (Epsilon)
(b) Cardinality (Gisette)
(c) Dim. (Epsilon)
(d) Dim. (Gisette)

Fig. 9: Scalability

B. Parallel Kernel Value Read

To show the efficiency of the parallel kernel value read technique, we conducted experiments using 1 thread (i.e., sequential read) and 2, 4 and 8 threads to read a row of the precomputed kernel matrix. The dataset used here is the Epsilon dataset with 120,000 training examples. We measured the total elapsed time on reading kernel values from the SSD in the cross-validation. Figure 7 shows the results. As can be seen from the figure, the parallel kernel value read technique using 8 threads achieves the best performance and significantly outperforms the sequential read. Another observation from the figure is that increasing the number of threads decreases the improvement ratio. This is because of the overhead of creating more threads at each training iteration.

C. Performance of the Extreme Example Search

Here, we compare our improved search algorithm (denoted by "MReduce" where "M" stands for MASCOT) with the search algorithm used in gSVM (denoted by "Reduce"). We measured the total time of searching u in the 10-fold cross-validation. As shown in Figure 8, MReduce considerably outperforms Reduce on all the datasets. This demonstrates the effectiveness of our techniques used in MReduce. We notice that the improvement on the MNIST dataset is the most significant. This is because the search operation is more expensive due to the larger cardinality compared with the other two datasets. The experimental results of searching l and f_{max} are similar to the above and hence we omit them here.

D. Overall Scalability and Efficiency

Scalability: To demonstrate the scalability of MASCOT over dataset cardinality and dimensionality on large datasets and small datasets, we generated two groups of datasets from the Epsilon dataset and the Gisette dataset, respectively. (i) The first group contains the subsets of the Epsilon dataset with cardinality varying from 10,000 to 120,000 and dimensionality varying from 500 to 2,000. (ii) The second group contains the subsets of the Gisette dataset with cardinality varying from 2,000 to 6,000 and dimensionality varying from 2,000 to 5,000. When varying cardinality (dimensionality), the default dimensionality (cardinality) of the subsets of the Epsilon dataset is fixed at 2,000 (60,000). Similarly, when varying cardinality (dimensionality), the default dimensionality (cardinality) of the Gisette dataset is fixed at 5,000 (6,000).

As can be seen from Figures 9a and 9b, MASCOT scales very well as the dataset cardinality increases. LibSVM is extremely slow on the subsets of the Epsilon dataset with cardinality larger than 30,000 and the results are omitted. In comparison with MASCOT, gSVM is much slower on small datasets and it cannot handle datasets of over 30,000 examples because the datasets are too large to be held in the GPU memory. As we can see from Figure 9c, gSVM cannot handle the subsets of the Epsilon dataset with more than 1,000 dimensions. We omit the experimental results on LibSVM, since LibSVM is extremely slow on the subsets of the Epsilon dataset due to the large cardinality (i.e., 60,000). Figures 9d shows that the elapsed time of gSVM and LibSVM increases much faster than that of MASCOT when the data dimensionality increases on the Gisette dataset. We observe that the elapsed time of MASCOT is almost stable as the dimensionality increases. Since after the kernel matrix precomputation, dimensionality is irreverent to the cross-validation.

Efficiency: In this set of experiments, we limit the cardinality and dimensionality of the datasets so that most of them can be handled by gSVM and LibSVM. As can be seen from Figure 10, gSVM cannot process the rcv1 dataset due to its limitation on memory usage and MASCOT is three orders of magnitude faster than LibSVM on this dataset. For the others, MASCOT is an order of magnitude faster than gSVM and two orders of magnitude faster than LibSVM.

Note that the total time shown in Figure 10 of MASCOT includes the time for the kernel matrix precomputation. In the experiments, we noticed that the cost of the kernel matrix precomputation is low. For example, the kernel value precom-

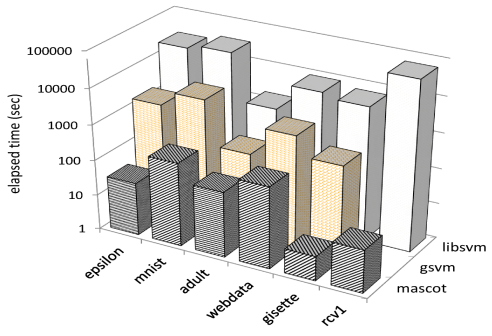


Fig. 10: Efficiency

putation took less than 20% of the total time of a 10-fold cross-validation on Adult, MNIST and Webdata datasets. The kernel matrix precomputation is only performed once and can be used for many cross-validations.

Our experimental results also showed that the classification accuracy of the trained classifiers using MASCOT is the same as that of the trained classifiers using gSVM and LibSVM on all the datasets tested.

VI. Conclusions

In this paper, we proposed MASCOT, a scheme for scalable and fast SVM cross-validation by exploiting the high computation power of GPUs and fast access of SSDs. Our key ideas are as follows. (i) To avoid holding the whole dataset in the memory and avoid performing repeated kernel value computation, we precompute the kernel values and reuse them. (ii) We store the precomputed kernel values to a high-speed storage framework, consisting of CPU memory extended by solid state drives and the GPU memory as a cache, so that reusing (i.e., reading) kernel values takes much lesser time than computing them on-the-fly. (iii) To further improve the efficiency of the SVM training, we apply a number of techniques for the extreme example search algorithm, design a parallel kernel value read algorithm, propose a caching strategy well-suited to the characteristics of the storage framework, and parallelize the tasks on the GPU and the CPU. Our experimental results show that for datasets of sizes that existing algorithms can handle, MASCOT achieves orders of magnitude speedup. More importantly, MASCOT enables SVM cross-validation on datasets of very large scale that existing algorithms are unable to handle.

Acknowledgements

This work is supported in part by the Australian Research Council (ARC) Discovery Project DP130104587. Dr. Rui Zhang is supported by the ARC *Future Fellowships Project* FT120100832. Zeyi Wen is supported by the Commonwealth Scientific and Industrial Research Organisation (CSIRO).

References

[1] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Workshop on Computational Learning Theory*, 1992, pp. 144–152.

[2] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast SVM training and classification on graphics processors," in *ICML*, 2008, pp. 104–111.

[3] F. O. Catak and M. E. Balaban, "Cloudsvm: training an svm classifier in cloud computing systems," in *Pervasive Computing and the Networked World*, 2013, pp. 57–68.

[4] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for support vector machines," in *International Workshop on Image Analysis for Multimedia Interactive Services*, 2011.

[5] Q. Li, R. Salman, E. Test, R. Strack, and V. Kecman, "Parallel multitask cross validation for support vector machine using GPU," *JPDC*, vol. 73, no. 3, pp. 293–302, 2013.

[6] D. DeCoste and K. Wagstaff, "Alpha seeding for support vector machines," in *KDD*, 2000.

[7] M. M. Lee, S. S. Keerthi, C. J. Ong, and D. DeCoste, "An efficient method for computing leave-one-out error in SVMs with gaussian kernels," *IEEE Transactions on Neural Networks*, 2004.

[8] Z. Wen, R. Zhang, and K. Ramamohanarao, "Enabling precision/recall preferences for semi-supervised svm training," in *CIKM*, 2014.

[9] A. Cotter, N. Srebro, and J. Keshet, "A GPU-tailored approach for training kernelized SVMs," in *KDD*, 2011, pp. 805–813.

[10] T. Joachims, "Training linear SVMs in linear time," in *KDD*, 2006, pp. 217–226.

[11] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: Primal estimated sub-gradient solver for svm," *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, 2011.

[12] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for SVMs," in *Neural Networks for Signal Processing*, 1997, pp. 276–285.

[13] J. C. Platt, "Fast training of SVMs using sequential minimal optimization," in *Advances in kernel methods*. MIT Press, 1999, pp. 185–208.

[14] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, pp. 1–27, 2011.

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

[16] G. Caruana, M. Li, and M. Qi, "A mapreduce based parallel svm for large scale spam filtering," in *Fuzzy Systems and Knowledge Discovery*, vol. 4, 2011, pp. 2659–2662.

[17] P. A. Forero, A. Cano, and G. B. Giannakis, "Consensus-based distributed support vector machines," *The Journal of Machine Learning Research*, vol. 99, pp. 1663–1707, 2010.

[18] T. Joachims, "Making large-scale SVM learning practical," in *Advances in kernel methods*. MIT Press, 1999, pp. 169–184.

[19] L. Yang, F. Zhou, and Y. Xia, "An improved caching strategy for training SVMs," *International Conference on Intelligent Systems and Knowledge Engineering*, pp. 1397–1401, 2007.

[20] G. W. Flake and S. Lawrence, "Efficient SVM regression training with SMO," *Machine Learning*, vol. 46, no. 1-3, pp. 271–290, 2002.

[21] K. P. Bennett and E. J. Bredensteiner, "Duality and geometry in svm classifiers," in *ICML*, 2000, pp. 57–64.

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

[23] 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.

[24] P. G. Ward, Z. He, R. Zhang, and J. Qi, "Real-time continuous intersection joins over large sets of moving objects using graphic processing units," *The VLDB Journal*, to appear.

[25] H. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B+-tree based indexing method for nearest neighbor search," *TODS*, vol. 30, no. 2, pp. 364–397, 2005.

[26] C. NVIDIA, "NVIDIA CUDA programming guide," 2011.

[27] V. Volkov, "Better performance at lower occupancy," in *the GPU Technology Conference*, vol. 10, 2010.

[28] B. Y. Y. Won, S. C. S. Kang, J. Choi, and S. Yoon, "SSD characterization: From energy consumptions perspective," *Proceedings of HotStorage*, 2011.

[29] M.-F. Balcan and A. Blum, "On a theory of learning with similarity functions," in *ICML*, 2006, pp. 73–80.