# Adaptive Kernel Value Caching for SVM Training

Qinbin Li, Zeyi Wen*, Bingsheng He*

***Abstract*—Support Vector Machines (SVMs) can solve structured multi-output learning problems such as multi-label classification, multiclass classification and vector regression. SVM training is expensive especially for large and high dimensional datasets. The bottleneck of the SVM training often lies in the kernel value computation. In many real-world problems, the same kernel values are used in many iterations during the training, which makes the caching of kernel values potentially useful. The majority of the existing studies simply adopt the LRU (least recently used) replacement strategy for caching kernel values. However, as we analyze in this paper, the LRU strategy generally achieves high hit ratio near the final stage of the training, but does not work well in the whole training process. Therefore, we propose a new caching strategy called EFU (less frequently used) which replaces the less frequently used kernel values that enhances LFU (least frequently used). Our experimental results show that EFU often has 20% higher hit ratio than LRU in the training with the Gaussian kernel. To further optimize the strategy, we propose a caching strategy called HCST (hybrid caching for the SVM training), which has a novel mechanism to automatically adapt the better caching strategy in the different stages of the training. We have integrated the caching strategy into ThunderSVM, a recent SVM library on many-core processors. Our experiments show that HCST adaptively achieves high hit ratios with little runtime overhead among different problems including multi-label classification, multiclass classification and regression problems. Compared with other existing caching strategies, HCST achieves 20% more reduction in training time on average.**

***Index Terms*—SVMs, caching, kernel values, efficiency.**

## I. Introduction

The Support Vector Machine (SVM) [1] is a classic supervised machine learning algorithm, and can solve problems with structured, unstructured and semi-structured data [2]. SVMs can solve structured multi-output learning problems, which include multi-label classification [3], [4], [5], multiclass classification [6], [7] and vector regression [8]. Some examples applications of SVMs include document classification, object detection, and image classification [9]. The underlying idea of training SVMs is to find a hyperplane to separate the two classes of data in their original data space. To handle non-linearly separable data, SVMs use a kernel function [10] to map data to a higher dimensional space, where the data may become linearly separable.

Although SVMs have several intriguing properties, the high training cost for large datasets is a deficiency. Even though many existing studies have been done for accelerating the SVM training [11], [12], [13], the kernel value computation is usually the most time-consuming operation of the training. To reduce the cost of kernel value computation, caching kernel values may be a good solution. Since the same kernel values are often used in different iterations during the training, we can avoid computing the kernel values if they are cached. Many SVM libraries such as LIBSVM and SVM$^{light}$ [14] adopt the LRU replacement strategy for caching kernel values. LRU works well for occasions with good temporal locality, which may not happen in the SVM training. In order to analyze the patterns in the entire SVM training, we divide the training process into several stages evenly according to the number of iterations. As we observed in the experiments, LRU only works well near the final stage of the training.

However, proposing a suitable caching strategy for the SVM training is challenging, because (i) the access pattern is not trivial to identify, and (ii) the caching strategy should be lightweight in terms of runtime overhead. Based on the pattern analysis, we propose a new strategy EFU that enhances LFU. The EFU strategy replaces the less frequently used kernel values when the cache is full, which is suitable for the access pattern of the kernel values. In order to adapt the access pattern of different stages, we propose the HCST replacement strategy, which automatically switches to the better strategy between EFU and LRU using the collected statistics. We collect the exact number of cache hits of the strategy being used and estimate the approximate number of cache hits of the other strategy based on its characteristic. Thus, HCST can automatically switch the strategy between EFU and LRU. To reduce the cost of copying data to cache, we perform the replacement of HCST in parallel. Compared with the existing strategies, the HCST strategy can achieve 20% more reduction in training time on average.

The main contributions of this paper are as follows.

- By splitting the training process to stages, we discover common features for the access patterns of different datasets.
- We propose a new caching strategy, EFU, which enhances LFU to take advantage of the access patterns of the kernel values.
- We design an adaptive caching scheme, HCST, which can fully utilize the characteristics of EFU and LRU to achieve a better performance.
- We conduct experiments on different problems including multi-label classification, multiclass classification and regression problems. The experimental results show that HCST is superior compared with other caching strategies, including LRU, LFU, LAT [15] and EFU.

## II. Preliminaries and related work

In this section, we first present the formal definition of the SVM training problem, and explain a commonly used SVM

Q. Li and B. He are with National University of Singapore. Email: {qinbin, hebs}@comp.nus.edu.sg

Z. Wen is with The Univerisity of Western Australia. Email: zeyi.wen@uwa.edu.au

Z. Wen and B. He are the corresponding authors.

training algorithm called *SMO*. Then, we describe a more recent SVM training algorithm [13] which solves multiple SMO subproblems in each iteration and exploits batch processing. Finally, we discuss the existing SVM libraries and caching strategies.

### A. The SVM training problem

A training instance $x_i$ is attached with an integer $y_i \in \{+1, -1\}$ as its label. A positive (negative) instance is an instance with the label of $+1$ ($-1$). Given a set of $n$ training instances, the goal of the SVM training is to find a hyperplane that separates the positive and the negative training instances in the feature space induced by the kernel function with the maximum margin and meanwhile, with the minimum misclassification error on the training instances. The SVM training is equivalent to solving the following problem:

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

where $\alpha \in \mathbb{R}^n$ is a weight vector, and $\alpha_i$ denotes the *weight* of $x_i$; $C$ is for regularization; $Q$ denotes an $n \times n$ matrix and $Q = [Q_{ij}]$, $Q_{ij} = y_i y_j K(x_i, x_j)$ and $K(x_i, x_j)$ is a kernel value computed from a kernel function.

Kernel functions (e.g., the Gaussian kernel function [10], the ideal kernel function [16]) are used to map the problem from the original data space to a higher dimensional data space. For a training set with $n$ instances, the $i^{th}$ row $\mathcal{K}_i = \langle K(x_i, x_1), K(x_i, x_2), \ldots, K(x_i, x_n) \rangle$ of the kernel matrix corresponds to all the $n$ kernel values of the instance $x_i$.

### B. The SMO algorithm

Problem (1) is a quadratic programming problem, and can be solved by many algorithms. Here, we describe a popular training algorithm, namely the Sequential Minimal Optimization (SMO) algorithm [11], which is adopted in many existing SVM libraries such as LIBSVM [17] and liquidSVM [18]. The SMO algorithm iteratively improves the weight vector $\alpha$ until the optimal condition of the SVM is met. The optimal condition is reflected by an *optimality indicator vector* $f = \langle f_1, f_2, \ldots, f_n \rangle$ where $f_i$ is the optimality indicator for the $i^{th}$ instance $x_i$ and $f_i$ can be obtained using the following equation: $f_i = \sum_{j=1}^{n} \alpha_j y_j K(x_i, x_j) - y_i$. The SMO algorithm has the following three steps:

**Step 1**: Find two extreme training instances, denoted by $x_u$ and $x_l$, which have the minimum and maximum optimality indicators, respectively. The indexes of $x_u$ and $x_l$, denoted by $u$ and $l$ respectively, can be computed by the following equations [19].

$$\begin{aligned} u &= \arg\min_i \{f_i | x_i \in X_{upper}\} \\ l &= \arg\max_i \{\frac{(f_u - f_i)^2}{\eta_i} | f_u < f_i, x_i \in X_{lower}\} \end{aligned} \tag{2}$$

where

$$X_{upper} = X_1 \cup X_2 \cup X_3, \; X_{lower} = X_1 \cup X_4 \cup X_5$$

and

$$\begin{aligned} X_1 &= \{x_i | x_i \in X, 0 < \alpha_i < C\} \\ X_2 &= \{x_i | x_i \in X, y_i = +1, \alpha_i = 0\} \\ X_3 &= \{x_i | x_i \in X, y_i = -1, \alpha_i = C\} \\ X_4 &= \{x_i | x_i \in X, y_i = +1, \alpha_i = C\} \\ X_5 &= \{x_i | x_i \in X, y_i = -1, \alpha_i = 0\} \\ \eta_i &= K(x_u, x_u) + K(x_i, x_i) - 2K(x_u, x_i) \end{aligned}$$

$f_u$ and $f_l$ denote the optimality indicators of $x_u$ and $x_l$, respectively.

**Step 2**: Improve the weights of $x_u$ and $x_l$, denoted by $\alpha_u$ and $\alpha_l$, by updating them as follows.

$$\alpha'_l = \alpha_l + \frac{y_l(f_u - f_l)}{\eta}, \quad \alpha'_u = \alpha_u + y_l y_u(\alpha_l - \alpha'_l)$$

where $\eta = K(x_u, x_u) + K(x_l, x_l) - 2K(x_u, x_l)$. To guarantee the update is valid, when $\alpha'_u$ or $\alpha'_l$ exceeds the domain of $[0, C]$, $\alpha'_u$ and $\alpha'_l$ are adjusted into the domain.

**Step 3**: Update the optimality indicators of all the training instances. The optimality indicator $f_i$ of the instance $x_i$ is updated to $f'_i$ using the following formula:

$$f'_i = f_i + (\alpha'_u - \alpha_u) y_u K(x_u, x_i) + (\alpha'_l - \alpha_l) y_l K(x_l, x_i)$$

SMO repeats the above steps until the following condition is met.

$$f_u \geq f_{max} = \max\{f_i | x_i \in X_{lower}\} \tag{3}$$

### C. A more recent SVM training algorithm based on SMO

As we have mentioned in Section II-B, the SMO algorithm selects two training instances (which together form a working set) to improve the current SVM. Instead of using a working set of size two, a more recent SVM training algorithm uses a bigger working set and solves multiple subproblems of SMO in a batch [13], which is implemented in ThunderSVM[1].

Given the training dataset, ThunderSVM does the following steps to train the SVM. First, a working set is formed with a number of instances that violate the optimality condition the most. Second, the rows of kernel values needed for the subproblems corresponding to the working set are computed. Third, the SMO algorithm is used to solve each of the subproblems. ThunderSVM repeats the above steps until the termination criterion is met.

The second step can be done by matrix multiplication, and the third step can be solved by SMO as discussed in Section II-B. Solving the first step is much more challenging. Here we elaborate the details about the first step. At each iteration when ThunderSVM updates the working set, $q$ instances in the working set will be replaced with $q$ new violating instances (e.g., $q = 512$ by default in ThunderSVM). The intuition for updating the working set is to choose $q$ training instances

[1]For ease of presentation, we use ThunderSVM to refer to "the SVM training algorithm implemented in ThunderSVM".

that violate the optimality condition (cf. Inequality 3) the most, such that the current SVM can be potentially improved the most. ThunderSVM does not update the whole working set to mitigate the local optimization. ThunderSVM sorts the optimality indicators ascendingly. Then, it chooses the top $q$=2 training instances whose $y_i$ $_i$ can be increased, and the bottom $q$=2 training instances whose $y_i$ $_i$ can be decreased. ThunderSVM considers $y_i$ $_i$, because of the constraints 0 $_i$ $C$ and $\sum_{i=1}^{n} y_i$ $_i$ = 0 in Problem (1).

In summary, the training algorithm of ThunderSVM has two phases: selecting a working set from the training dataset, and solving the subproblems using SMO. Since the same row of the kernel values is often used multiple times in different iterations during the training, we can adopt a cache to store the kernel values to reuse them between the iterations.

### D. Existing SVM libraries and caching

One of the key factors for the success of SVMs is that many easy-to-use libraries are available. The popular libraries include SVM$^{light}$ [20], LIBSVM [17] and ThunderSVM. SVM$^{light}$, LIBSVM and many other SVM libraries (e.g., liquidSVM) adopt the LRU strategy for kernel value caching, while ThunderSVM has not applied caching strategy. The LRU caching strategy works well for occasions with good temporal locality. However, no evidence has shown that the time of the reuse interval of kernel values is relatively small, and hence the LRU caching strategy may not be a good option for kernel value caching in the SVM training.

MASCOT [15] adopts a new caching strategy called LAT. When the cache is full, LAT replaces the row with the minimum row index in the kernel matrix. LAT effectively caches the last part of the kernel matrix which is stored in SSDs in MASCOT. This makes LAT prefer to cache kernel values stored in SSDs rather than those in the main memory. LAT shows better performance than LRU on MASCOT.

## III. Motivations

For ease of presentation, we call one row of kernel matrix an *item*, which is the smallest unit for the replacement operation in the training. Our design of HCST is motivated by the following observations of the access pattern of the items.

*Observation 1: Due to the relatively large reuse interval, LRU is not suitable for the overall training.* Suppose the cache can store at most 5,000 items. We define *reuse interval* $R$ to be the number of iterations between two consecutive accesses to an item. For clarity of presentation, we divide the reuse interval $R$ to four different levels: *small* (0 < $R$ < 5,000), *medium* (5,000 $R$ < 10,000) , *large* (10,000 $R$ < 15,000) and *huge* ($R$ 15,000). Figure 1 shows the cumulative percentage of different levels. LRU works well if most reuse intervals are small so that the item is still in cache when accessed again. However, as we can see from the results, except for *rcv1*, the proportion of small reuse interval is very low, which means LRU is not a suitable strategy for the kernel value caching in SVM training. Wen *et al*. [15] has proved that the items are accessed in a quasi-round-robin manner, where the selected

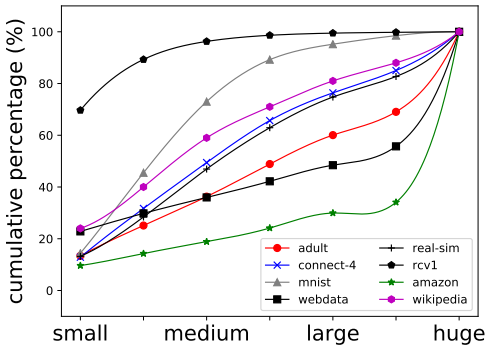



Fig. 1. Interval of two consecutive accesses of the same item

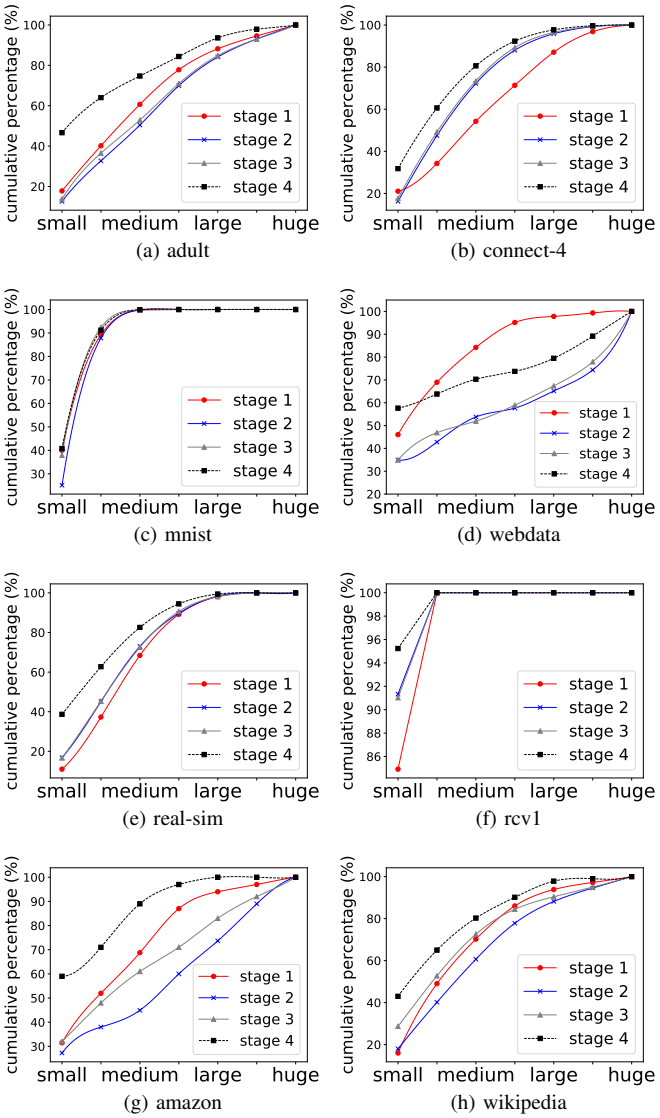



Fig. 2. Cumulative distribution of reuse intervals in different stages

items are unlikely to be used again in the near future of the training, which is consistent with our observation.

*Observation 2: LRU may have a better performance in the late stage than the early stages of the training process, where the proportion of small reuse intervals is relatively large.* To analyze changes in the access patterns throughout the whole SVM training, we divide the training process into

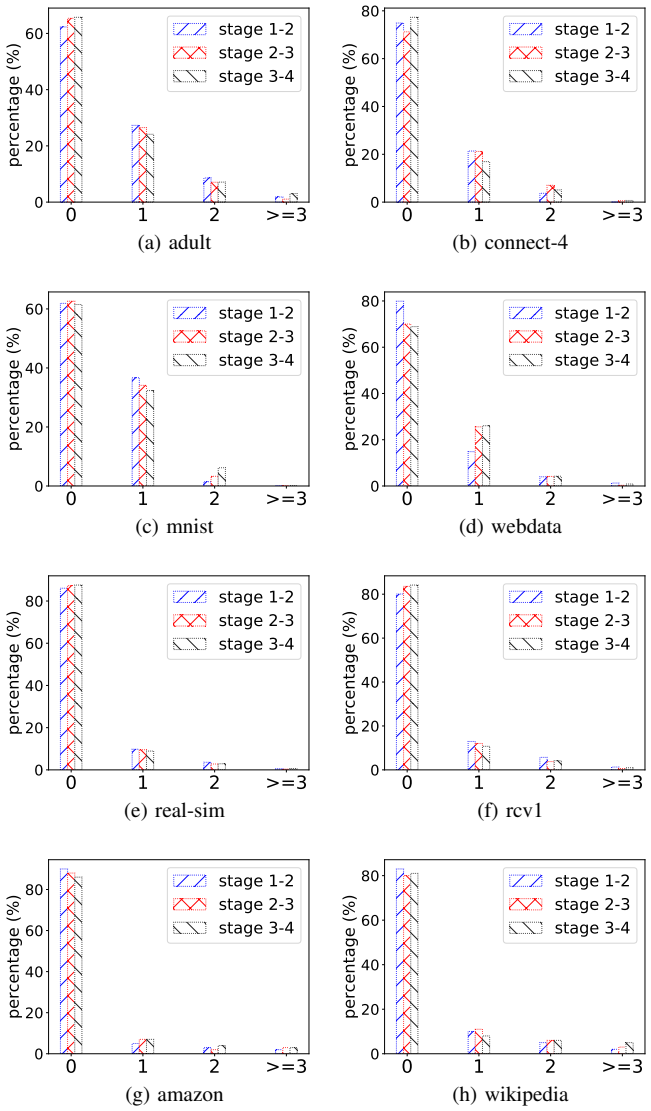



Fig. 3. Distribution of the difference of access frequencies in different stages

four stages evenly according to the number of iterations. We have also tested different number of stages such as two and eight, and observed similar results. Figure 2 shows the cumulative distribution of reuse intervals in different stages. We observe that there are some common features of the access patterns in the training progress. The stage 4 always has the highest proportion of small reuse intervals. Here is a scientific explanation. Compared with the early stages, the SVM training is more likely to choose the support vectors to adjust the hyperplane near the end of the training process [21]. Then, the distribution of the accesses is more concentrated in the late stage. Thus, the proportion of small reuse interval is larger in the late stage, where LRU may have a good performance. So LRU can perform better in the late stage of the training.

*Observation 3: The items tend to have similar access frequency across different stages.* In order to study the uniformity of access frequencies of the items, we calculate the difference of access frequencies of the same item between different stages. The distribution of the difference are shown in Figure 3. We can find the distribution of access frequencies is quite uniform in different stages. More than 60% of the items have no difference in the access frequencies of different stages. This observation results from the property of the SVM training. In many real world problems, most training instances are non-support vectors, which are not actively selected to update the hyperplane in the whole training process [20]. Then, for most items, the access frequencies across different stages tend to be low and close. Based on this characteristic, items with higher access frequency should be cached since they are likely to be accessed with a higher frequency in the subsequent training progress. A classic algorithm for frequency based cache replacement is LFU. However, the LFU strategy always caches the newly generated item and replaces the least frequently used item in cache, even though the item in cache has a higher frequency than the new item, which is not good enough for caching items with higher access frequencies. That is why we propose to enhance LFU with EFU.

## IV. The HCST strategy

### A. An Overview of HCST

The structure of the SVM training with HCST is shown in Figure 4. There are mainly two components of HCST: the candidate strategies and the strategy selection. In the iterative process of the SVM training, HCST selects the better caching strategy from candidate strategies and applies it until the next selection. The main aim of HCST is to improve the hit ratio of items in the entire training process.

*1) Candidate strategies:* According to the observations in Section III, we use EFU and LRU as our candidate strategies. EFU is our proposed caching strategy, which aims to cache the items with higher access frequencies. We will introduce the details of EFU in Section IV-B. LRU is a classic caching strategy, which aims to cache the items with recently time used. By splitting the training process to different stages, as we claimed in Section III, we have two key findings. One is that the items with higher access frequency are more likely to also be accessed more times in the next stage, which is fully utilized by EFU. The other one is that the reuse interval of items is likely to be smaller in the late stage of the training process, which makes LRU a potentially suitable strategy. By making full use of these two features, we propose the EFU strategy and use it and LRU as our candidate strategies.

*2) Strategy selection:* At first we adopt the EFU strategy, which appears to have a good performance in the overall training. During the training, after every fixed number of iterations, we make a selection from the candidate strategies based on collected statistics. Here we use the number of cache hits as the criterion. If the number of cache hits of EFU is bigger than that of LRU in the current stage, we adopt EFU in the next stage, otherwise we adopt LRU. In this way we can switch to LRU timely if LRU already works better than EFU. In addition, due to the jitter of the number of hits using LRU, there are some cases we switch to LRU prematurely. This deficiency is generally fine, since HCST can switch back to EFU quickly in the next comparisons. In general, HCST is pure EFU if no switch happens or piecewise EFU and LRU if any switch happens in the training process.

Fig. 4. The process of the SVM training with HCST

Fig. 5. A running example of EFU

### B. The EFU strategy

The key ideas of EFU are described as follows. We maintain a counter for each item of the kernel matrix to record the access frequency. When a new item is computed and the cache is already full, we first decide whether it will be added into the cache. If all the items in the cache have higher accumulated access frequencies than the new item, EFU will not cache it. Otherwise, EFU replaces the item in the cache with a lower access frequency than this newly computed item. In this way, we can always store the items which have higher frequency of usage currently in the SVM training. Figure 5 shows a running example of EFU. Suppose the cache stores 3 items: $M_1$, $M_2$ and $M_3$. These items have been accessed 4 times, 5 times and 4 times, respectively. When a newly computed item $M_4$ comes, EFU finds the item $M_1$ which has a lower access frequency than $M_4$ and replaces it. When another newly computed item $M_5$ comes, EFU does not do replacement since no item in the cache has a lower access frequency.

The LFU strategy always replaces the least frequently used item, even though it has a higher access frequency than the new item, while the EFU strategy can avoid the problem. EFU is more suitable to store the items with higher access frequencies compared with LFU.

### C. The HCST strategy

As we have shown in Figure 2, the proportion of small reuse distance differs clearly in the late stage of the training process. There may be some cases where the performance of LRU is better than EFU in the late stages as the training processes. To handle it, we implement a dynamic strategy called HCST which allows the caching strategy to switch between EFU and LRU. At the beginning of training, we adopt EFU. In regular stages during the training, we compare the number of cache hits if adopted LRU ($H_{LRU}$) with the number of cache hits if adopted EFU ($H_{EFU}$). If we found $H_{LRU}$ is bigger than $H_{EFU}$, we use LRU for the next stage, otherwise using EFU. As we have discussed in Section III, the performance of LRU strategy can be better in the late stage of the SVM training, so it is reasonable to switch the strategy to LRU if the hit number of LRU is already bigger than EFU. Furthermore, to reduce the bad effect of the premature switching from EFU to LRU, which may be caused by the instability of the number of cache hits using LRU, HCST can switch back to EFU timely in the next comparison.

In practice, it is challenging to know the exact number of cache hits of both strategies since we only can adopt one strategy at one time. However, we can estimate the number of cache hits based on the features of the cache strategy. For ease of presentation, we call the time we compare $H_{LRU}$ with $H_{EFU}$ as a checkpoint. Suppose the cache size is $s$, which means at most $s$ items can be cached.

Assume we are using EFU before a checkpoint $T_1$. We use a counter to record the number of cache hits $H_{hit}$ after $T_1$. So $H_{hit}$ is the exact $H_{EFU}$ when we arrived the next checkpoint $T_2$. We use another counter to record the number of accesses $H_s$ whose reuse interval is smaller than $s$ after $T_1$. Note that these accesses can all yield cache hits if using LRU. So we use $H_s$ as the approximate $H_{LRU}$ when we arrived $T_2$. If $H_s$ is not bigger than $H_{hit}$ in $T_2$, we do not change the strategy and do the same operations for the next stage.

If $H_s$ is bigger than $H_{hit}$, we change the strategy to LRU and save the value $H_{hit}$. When using LRU, we still use the counter to record the number of cache hits $H_{hit'}$ after checkpoint $T_2$, which is the exact $H_{LRU}$. Since the distribution of access frequencies is quite evenly as we describe in Section III, the hit ratio of EFU should be close between different stages. So we use $H_{hit}$ as the approximate $H_{EFU}$. In the next checkpoint $T_3$, we compare $H_{hit'}$ with $H_{hit}$. If $H_{hit'}$ is smaller than $H_{hit}$, we switch the strategy back to EFU. Otherwise we do not change the strategy. Algorithm 1 shows the training process with the HCST strategy.

Overall, we can get the exact number of cache hits with the strategy being used and estimate the approximate number of cache hits if adopted the other strategy. By comparing these two values, the dynamic selection of the strategies can make efficient use of the characteristics we observed for the access patterns of the items.

### D. Optimization for multi-output learning tasks

By adopting the "one-vs-all" scheme, we can transform a multi-output learning task into a set of independent single-output learning tasks [6]. Then. we train a solver for each single-output learning task. Since each solver can utilize the CPU resources well, the solvers are trained in a sequential manner. Although the training of each solver is independent, they use the same training instances. To exploit this property, we can reuse the kernel values between different solvers. Instead of adopting individual cache for each solver, we use one shared cache during the whole training process for the multi-output tasks. Thus, after a solver finishes the training, the kernel values in the cache can still be reused in the next solver.

In summary, there are two levels of reuses of the kernel values in the multi-output learning tasks. One is the reuse

**Algorithm 1:** Pseudo of the training process with HCST

```
Input: The training dataset
Output: The SVM model
Adopt EFU at the beginning;
while the optimality condition of SMO is not met do
    if it is a checkpoint then
        if it is using EFU then
            Get the number of cache hits with EFU (H_hit);
            Estimate the number of cache hits with LRU (H_s);
            if H_hit < H_s then
                Save the value of H_hit;
                Switch to LRU;
        else
            Get the number of cache hits with LRU (H_hit');
            Estimate the number of cache hits with EFU (H_hit);
            if H_hit' < H_hit then
                Switch to EFU;
    Training;
```

of the kernel values between different iterations of a solver (iteration-level reuse). The other one is the reuse of the kernel values between different solvers of the training process (solver-level reuse). Compared with the original HCST, which only exploits the iteration-level reuse, the solver-level reuse can fully utilize the characteristic of the multi-output learning task and has no extra overhead. As we will show in Section V-C2, our technique can reduce the training time significantly.

### E. Parallel implementation of the replacement operations

Since ThunderSVM accesses $q$ items on each iteration, it is time-consuming to do replacement one by one. To reduce the cost of caching, we implement the parallel replacement operation, which is shown in Figure 6. Suppose we have $p$ threads and there are $u$ items that yield cache misses among accessed $q$ items. We evenly divide these $u$ items into $p$ groups, and each group is assigned to one thread. To avoid conflicts in the cache when multiple threads perform the replacement, we also divide the cache into $p$ parts, and each part is assigned to one thread. Each thread traverses its corresponding part to find an eligible item to replace. Since each thread performs replacement operations for the same number of items (i.e., $u/p$) and has the same size of cache space ($s/p$) to traverse, the workload of each thread is balanced. By using multiple threads, we can reduce the cost of the replacement significantly, as we will show in Section V-C.

### F. Efficiency analysis

Here we analyze the theoretical efficiency of the HCST strategy. We suppose the cardinality of the dataset is $n$ and the dimension of the dataset is $d$. ThunderSVM accesses $q$ items on each iteration.

1) Time complexity of the replacement: Suppose the cache size is $s$. We need at most $O(s)$ to traverse on the cache to find an item that meets the requirements of the cache strategy to replace. On each iteration, there are at most $q$ items to be replaced. So the time complexity of our caching strategy in one iteration is $O(qs)$.

Time complexity improvement by parallelism: Suppose the number of threads is $p$. If we perform the replacement in parallel, we only need $O(s/p)$ to find an item to replace. Additionally, each thread only needs to handle at most $q/p$ items. The time complexity of HCST in one iteration is $O(qs/p^2)$, which is reduced by $p^2$ times compared with performing replacement in serial. This property makes HCST appealing in the context of parallel computing.

Overhead of the dynamic selection: Although the HCST strategy needs a selection operation in the training process, the cost of it is very low. To get the hit number of EFU and LRU, we need maintain counters for each item as we have discussed in Section IV-C. We need $O(q)$ to update these counters in each iteration, where $q$ is much smaller than $n$. The comparison between the numbers of cache hits can be done in $O(1)$. The overhead of the dynamic selection is $O(q)$ in each iteration, which can be ignored compared with the time complexity of the training.

2) Memory consumption of the HCST strategy: To enable the switch between EFU and LRU, we need maintain two counters for each item to record the access frequency and the recently used time. Suppose each integer and float use 4 bytes to store. So we need $8n$ bytes for all counters. The extra cost of memory is very small compared with the size of the dataset, which costs $4nd$ bytes to store.

3) Benefit of caching: We can compute the benefit of caching theoretically for the SVM training. Suppose the number of floating point operations per second (i.e., flops) of CPU is $f$. Suppose we use the Gaussian kernel for the SVM training. The Gaussian kernel is defined as follows.

$$K(x_i, x_j) = \exp\{-(\|x_i\|^2 + \|x_j\|^2 - 2x_i^T x_j)\}$$

To compute an item (i.e., a row of the kernel matrix), we need to compute the inner products of all the training

Fig. 6. The parallel replacement operations

instances (i.e., $n$ vectors of $1 \times d$ dimensions), a multiplication of a $1 \times d$ vector and a $d \times n$ matrix (i.e., $x_i$ and the whole training dataset). Hunger [22] showed the ops of the dot product computation is $(2d - 1)n$. Since inner products are computed only once at the beginning of the SVM training, we do not count them in the cost. Ignoring the oating operations to compute the exponential formula, we can get the cost for computing an item as $T_o = (2 \cdot d - 1)n/l$. Suppose the number of cache hits is $h$. The time of kernel value computation saved by the caching strategy $T_s = h(2d - 1)n/l$.

Let the number of replacement operations be $u$. To simplify the model, we assume the sequential copy bandwidth of memory is a constant $b$. Each kernel value is 4 bytes. The size of an item is $4n$ bytes. So we can compute the time of replacement in cache as $T_c = 4un/b$.

Ignoring other costs, we can get the bene t of caching strategy as follows.

$$T_b = T_s - T_c = h(2d - 1)n/l - 4un/b \quad (4)$$

Equation (4) shows the bene t is positively related to the number of hits and negatively related to the number of copies. Furthermore, for datasets with higher cardinality and dimensionality, the bene t is higher.

## V. Experimental study

In this section, we present the empirically results of our proposed caching strategy HCST. We conducted the experiments on a workstation running Linux with two Xeon E5-2640v4 10 core CPUs and 256GB main memory. The number of threads is set to 20 by default to effectively utilize the resources of CPUs. The maximum number of items that cache can store is set to 5,000 by default, which is a relatively small size with little memory cost. We also try different cache sizes (i.e., 10K, 15K and 20K) in our experiments. The number of iterations between two consequent checkpoints is set to 20, which is an appropriate value as we will show in Section V-C1. The cache replacement strategies we used in our experiments include HCST, LRU, EFU, LFU and LAT. To ensure fairness, we used the same data structure to implement all the cache strategies. The kernel functions we used in our experiments include the Gaussian kernel and the sigmoid kernel. We used ThunderSVM as our training library and the stopping criteria are the same for all the experiments. We used 14 public datasets from the LIBSVM website[2] and this link[3] with four different problems including binary, multi-label and multiclass classi cation, and regression. Among 14 datasets, 6 datasets (adult, connect-4, mnist, webdata, real-sim and rcv1) are used in every experiment while the other 8 datasets are used together with the previous datasets to more thoroughly evaluate the training time of SVMs with HCST. For mediamill and rcv1s2, the label dimensionality is 101. For amazon and wikipedia, we randomly choose 10 labels to perform the classi cation task. Table I gives the details of the datasets and parameters of kernel functions used in the experiments. The parameters are the same as the existing study [15], [23], [24] or selected by a grid search on $C$ (ranges from 1 to 10 and $\gamma$ ranges from 0.001 to 0.5).

TABLE I
DATASETS AND KERNEL PARAMETERS

| dataset | task | cardinality | dimension | Gaussian | | sigmoid | |
|---|---|---|---|---|---|---|---|
| | | | | C | $\gamma$ | C | $\gamma$ |
| adult | binary classi cation | 32,561 | 123 | 100 | 0.5 | 10 | 0.01 |
| rcv1 | | 20,242 | 47,236 | 100 | 0.125 | 10 | 0.5 |
| real-sim | | 72,309 | 20,958 | 4 | 0.5 | 10 | 0.5 |
| webdata | | 49,749 | 300 | 10 | 0.5 | 10 | 0.01 |
| covtype | | 581,012 | 54 | 3 | 1 | 10 | 0.5 |
| connect-4 | multiclass classi cation | 67,557 | 126 | 1 | 0.3 | 10 | 0.001 |
| mnist | | 60,000 | 780 | 10 | 0.125 | 10 | 0.001 |
| mnist8m | | 8,100,000 | 784 | 1000 | 0.006 | \ | \ |
| mediamill | multi-label classi cation | 30,993 | 12,914 | 10 | 0.5 | 10 | 0.01 |
| rcv1s2 | | 3,000 | 101 | 10 | 0.5 | 10 | 0.5 |
| amazon | | 1,717,899 | 337,067 | 10 | 0.5 | 10 | 0.5 |
| wikipedia | | 1,813,391 | 2,381,304 | 10 | 0.5 | 10 | 0.5 |
| abalone | regression | 4,177 | 8 | 10 | 0.5 | 10 | 0.01 |
| E2006-t df | | 16,087 | 150,360 | 10 | 0.5 | 10 | 0.01 |

(a) Hit ratio with the Gaussian kernel (b) Hit ratio with the sigmoid kernel

Fig. 7. Hit ratio comparison

(a) adult (b) connect-4

(c) mnist (d) webdata

(e) real-sim (f) rcv1

Fig. 8. Hit ratio with different cache sizes

[2]https://www.csie.ntu.edu.tw/cjlin/libsvm/index.html
[3]http://manikvarma.org/downloads/XC/XMLRepository.html

### A. Hit ratio comparison

*Key finding 1: HCST can always achieve the highest hit ratio among all the caching strategies.*

Figure 7 shows the hit ratio of different caching strategies. We tried two different kernel functions: the Gaussian kernel and the sigmoid kernel. We first compare the four caching strategies except HCST. EFU and LFU have a relatively better performance when using the Gaussian kernel, while LRU outperforms the other strategies when using the sigmoid kernel. Furthermore, the hit ratio of EFU is higher than LFU on some datasets with the Gaussian kernel (e.g., *adult* and *mnist*). The LAT strategy does not show advantage on both two kernel functions. Except for HCST, none of the strategies is optimal across different datasets and different kernel functions. However, due to our specialized design, HCST is comparable to or even better than the best of them in all cases. HCST can adaptively achieve high hit ratios among different cases.

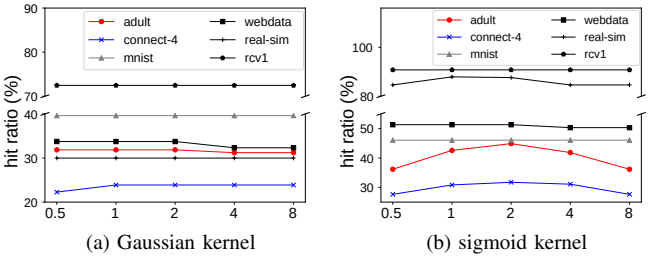
Figure 8 shows the hit ratio of different caching strategies with different cache sizes using the Gaussian kernel, which is the most widely used kernel function in the SVM training. The HCST strategy always has good performance on different settings of cache size in the experiments. Furthermore, all five cache replacement strategies have a similar hit ratio when the cache size is big. This is because when the cache is large enough, the cache can store almost all the items. As a result, the accesses that yield cache misses are mainly first usages of the items, which are inevitable no matter what caching strategy is used. Note that the increase in memory size can not keep up with the speed of data growth. The memory size is always relatively small compared with the size of kernel values of huge datasets. The HCST strategy is superior to the other strategies when the memory for cache is limited.

### B. The overall training with HCST

*Key finding 2: HCST can always clearly reduce the SVM training time.*

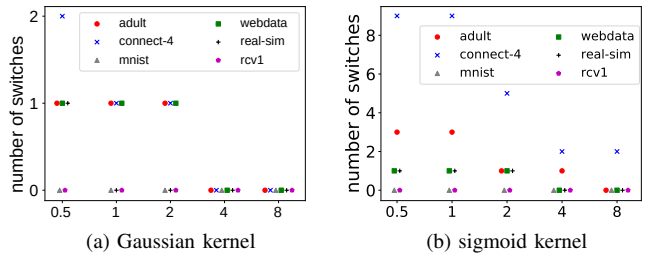
To show the performance of the HCST strategy, we compare it with the cache strategies LRU, LFU, LAT and EFU on all the 14 datasets. In Table II, we show the elapsed training time of no cache and the relative values of the other caching strategies against no cache. Compared with HCST, the other strategies have a smaller improvement, and sometimes even have no benefit due to the cost of caching (e.g., *adult*). The HCST strategy has a stable improvement and helps improve the SVM training without caching by at least 25% in all 14 datasets. For *rcv1*, the speedup is bigger than 4. For the multi-label datasets, HCST can reduce the training time by at least 30%, which is a significant improvement.

Table III shows the calculation time of kernel values and the cost of caching when using different caching strategies with the Gaussian kernel. Compared with LFU, LRU, LAT and EFU, HCST always has lower calculation time of kernel values and lower cost of caching. As we have shown in Figure 7a, the HCST strategy mostly has a higher hit ratio than the other strategies, especially LRU and LAT, which explains the lower calculation time of kernel values. As we have discussed in Section IV-E, the replacement of HCST can be performed in parallel, which indicates the lower cost of updating cache.




(a) Gaussian kernel (b) sigmoid kernel

Fig. 9. Hit ratio with different




(a) Gaussian kernel (b) sigmoid kernel

Fig. 10. The number of switches with different

Moreover, we can estimate the speedup of HCST on a GPU. Specifically, the training time with HCST on a GPU can be approximately computed by the hit ratio, which is independent of the hardware. Suppose the hit ratio of HCST is $h$ and the calculation time of kernel values without cache is $t_k$. Then we can estimate the calculation time of kernel values with HCST as $ht_k$ and ignore the overhead of caching which is relatively small. We have conducted experiments with a Pascal P100 GPU of 12GB memory. Table IV shows the exact training time without cache and the estimated training time with HCST. HCST can still reduce the training time by at least 10%.

### C. Effect of factors

*1) The impact of the length between two checkpoints: By setting the number of iterations between two consequent checkpoints to 2s=q, HCST can achieve the best performance.*

The setting of number of iterations between two consequent checkpoints should be appropriate. Suppose the number of iterations between two consequent checkpoints is $N_c$ and the cache size is $s$. Since ThunderSVM chooses $q$ items in each iteration, there are up to $qN_c$ replacement operations between two consequent checkpoints. We set the parameter $= qN_c=s$ and try different . The results are shown in Figure 9. Here we only show the experiments on six representative datasets, while the behavior of the other datasets is similar. From the results, we can observe that the hit ratio does not change clearly on *mnist* and *rcv1* with different . Also, we can find that the hit ratio changes more dramatically in the sigmoid kernel than the Gaussian kernel on *adult* and *connect-4*. To explain these cases, we have measured the number of switches of the training on different datasets, as shown in Figure 10. For *mnist* and *rcv1*, the number of switches is always zero, which means EFU is always the appropriate strategy for the training. Then, the hit ratio does not change no matter the value of . For *adult* and *connect-4*, compared with the Gaussian kernel, the

TABLE II
COMPARISON AMONG HCST AND THE OTHER EXISTING CACHING STRATEGIES

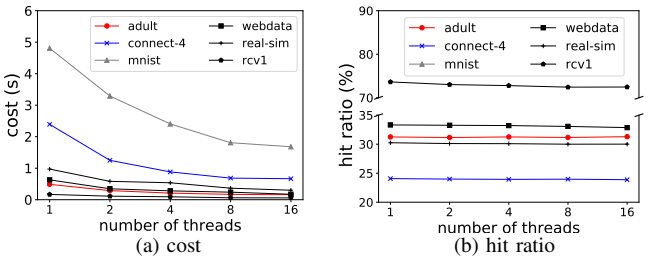
| dataset | Gaussian kernel | | | | | | | | sigmoid kernel | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | elapsed time (sec) / relative value against no cache | | | | | | speedup of HCST | training error /RMSE | elapsed time (sec) / relative value against no cache | | | | | | speedup of HCST | training error /RMSE |
| | no cache | HCST | LRU | LFU | LAT [15] | EFU | | | no cache | HCST | LRU | LFU | LAT | EFU | | |
| adult | 24.65 | -31.4% | +1.1% | -9.5% | -0.6% | -17.6% | **1.46** | 4.4% | 6.86 | -24.2% | -13.3% | -2.6% | +5.0% | -6.9% | **1.32** | 15.2% |
| connect-4 | 91.61 | -26.0% | +2.1% | -11.0% | +3.1% | -18.8% | **1.35** | 4.39% | 58.34 | -20.6% | -18.5% | -8.2% | -2.7% | -10.9% | **1.26** | 24.42% |
| mnist | 346.12 | -36.2% | -9.1% | -26.0% | -29.6% | -33.5% | **1.57** | 0% | 40.02 | -28.8% | -26.3% | -23.5% | -25.1% | -28.3% | **1.40** | 5.57% |
| webdata | 38.21 | -23.9% | -0.4% | -11.2% | 0% | -17.7% | **1.31** | 0.54% | 2.89 | -33.9% | -24.6% | -23.9% | -23.5% | -30.1% | **1.51** | 1.18% |
| real-sim | 75.93 | -34.4% | -11.2% | -28.1% | -21.8% | -33.9% | **1.52** | 0.27% | 239.13 | -79.0% | -76.7% | -72.6% | -42.8% | -75.7% | **4.76** | 0.84% |
| rcv1 | 27.66 | -77.3% | -78.9% | -78.3% | -73.9% | -77.2% | **4.41** | 0.11% | 57.87 | -89.6% | -89.4% | -88.9% | -88.5% | -89.4% | **9.59** | 0.29% |
| mnist8m | 219142 | -45.7% | -34.2% | -39.9% | -7.1% | -40.5% | **1.84** | 0% | > 7 days | | | | | | \ | \ |
| covtype | 2819 | -23.7% | -14.5% | -19.8% | -15.4% | -20.7% | **1.31** | 0% | 2164 | -20.2% | -10.9% | -9.9% | -10.9% | -20.6% | **1.25** | 51.24% |
| mediamill | 448.54 | -33.9% | -22.9% | -20.9% | -16.1% | -21.5% | **2.52** | 2.81% | 468.48 | -44.7% | -30.3% | -27.4% | -18.2% | -31.0% | **1.81** | 3.13% |
| rcv1s2 | 252.48 | -98.3% | -62.0% | -62.2% | -62.2% | -61.9% | **57.4** | 1.97% | 243.74 | -98.5% | -61.3% | -61.4% | -61.7% | -62.2% | **65.0** | 1.98% |
| amazon | 22317 | -33.5% | -9.0% | -21.5% | -13.6% | -29.5% | **1.50** | 0% | 1770 | -42.0% | -21.7% | -21.8% | -21.9% | -24.5% | **1.73** | 0% |
| wikipedia | 75384 | -49.6% | -15.6% | -27.8% | -21.6% | -35.0% | **1.98** | 0% | 10913 | -52.2% | -30.5% | -28.6% | -25.5% | -27.3% | **2.09** | 5.93% |
| abalone | 0.53 | -30.2% | -17.0% | -18.9% | -20.8% | -22.6% | **1.43** | 4.54 | 0.41 | -22.0% | -12.2% | -17.1% | -17.1% | -19.5% | **1.28** | 6.46 |
| E2006-tfidf | 224.43 | -48.0% | -38.4% | -33.1% | -36.1% | -34.0% | **1.92** | 0.12 | 219.54 | -26.6% | -23.0% | -13.8% | -18.5% | -19.2% | **1.36** | 0.50 |

TABLE III
CALCULATION TIME OF KERNEL VALUES AND COST OF CACHING (SEC)

| dataset | calculation time of kernel values | | | | | | overhead of cache replacement | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | no cache | relative value against no cache | | | | | HCST | relative value against HCST | | | |
| | | HCST | LRU | LFU | LAT | EFU | | LRU | LFU | LAT | EFU |
| adult | 18.32 | -42.6% | -9.5% | -26.9% | -14.7% | -35.3% | 0.19 | +1405% | +1463% | +1363% | +1047% |
| connect-4 | 75.76 | -33.6% | -9.4% | -26.9% | -11.0% | -26.6% | 0.84 | +910% | +952% | +988% | +238% |
| mnist | 315.7 | -38.7% | -15.8% | -32.54% | -37.7% | -38.1% | 1.85 | +773% | +823% | +544% | +541% |
| webdata | 29.63 | -33.5% | -16.8% | -30.4% | -17.4% | -32.7% | 0.20 | +2170% | +2220% | +2365% | +1010% |
| real-sim | 67.1 | -40.7% | -20.6% | -39.2% | -30.3% | -41.9% | 0.33 | +1321% | +1227% | +1030% | +315% |
| rcv1 | 26.36 | -82.1% | -83.7% | -83.2% | -78.8% | -77.4% | 0.06 | +66.7% | +133% | +167% | 0% |
| mnist8m | 213595 | -48.2% | -37.0% | -42.5% | -8.3% | -43.0% | 2380 | +27.0% | +14.4% | +92.7% | +5.9% |
| covtype | 2760 | -43.4% | -36.3% | -40.2% | -37.7% | -41.2% | 319 | +25.7% | +12.2% | +28.5% | +1.9% |
| mediamill | 392.36 | -41.7% | -33.5% | -30.4% | -21.1% | -30.5% | 5.32 | +130% | +168% | +338.2% | 0% |
| rcv1s2 | 249.86 | -99.5% | -62.9% | -63.0% | -63.0% | -62.8% | 0.31 | +9.7% | 0% | +3.2% | 0% |
| amazon | 21371 | -35.8% | -11.9% | -24.5% | -16.5% | -30.9% | 29 | +1697% | +1486% | +1603% | -13.8% |
| wikipedia | 74820 | -50.3% | -16.1% | -28.3% | -22.2% | -35.3% | 5.9 | +5137% | +4442% | +5239% | +290% |
| abalone | 0.37 | -45.9% | -40.5% | -43.2% | -43.2% | -40.5% | 0.02 | +200% | +200% | +150% | 0% |
| E2006-tfidf | 222.91 | -48.6% | -38.7% | -33.6% | -36.5% | -34.2% | 0.15 | +227% | +300% | +267% | +120% |

TABLE IV
THE ESTIMATED TRAINING TIME ON A GPU WITH HCST (SEC)

| datasets | calculation time of kernel values (sec) | | elapsed time of the training (sec) | | speedup of HCST |
|---|---|---|---|---|---|
| | no cache | HCST | no cache | HCST | |
| adult | 750 | 516 | 2749 | 2515 | 1.09 |
| connect-4 | 2320 | 1766 | 6550 | 5996 | 1.09 |
| mnist | 15122 | 9134 | 33626 | 27638 | 1.22 |
| webdata | 1013 | 682 | 3421 | 3090 | 1.11 |
| real-sim | 2358 | 1651 | 4105 | 3398 | 1.21 |
| rcv1 | 795 | 219 | 1555 | 979 | 1.59 |



(a) cost (b) hit ratio

Fig. 11. The parallel HCST strategy

range of the number of switches among different is bigger in the sigmoid kernel. Thus, the hit ratio also change more clearly in these cases. Generally, HCST has a well hit ratio on all cases when setting = 2, where $N_c = 2s=q$. Since $s = 5000$ and $q = 512$ in our experimental setting, we set the number of iterations between two consequent checkpoints to 20 (i.e., 2 5000=512) by default.

*2) The impact of the number of threads on HCST: The cost of HCST can be reduced significantly by using multiple threads while retaining the same hit ratio.*

Figure 11a shows the cost of the HCST caching strategy with different numbers of threads. Here, the "cost" of HCST means the extra computation results from the caching strategy, which includes copying the items to the cache and deciding whether the items are in the cache. As we can see from the results, the cost is significantly reduced by using multiple threads. When we use two threads, the cost can be reduced by more than 40% compared with the sequential version of HCST. Parallel HCST can improve the bandwidth usage and can save notable amount of time on copying items to the cache.

Next, we inspect the effect of multi-threading on the hit ratio. Figure 11b shows the hit ratio of HCST with different numbers of threads. The hit ratio is almost unchanged for all the datasets. This confirms that the parallel implementation of HCST has little impact on the hit ratio.

*3) The impact of different computational environments: The HCST strategy is portable across different environments.*

To show the effect of HCST on different computational environments, we conducted the experiments on a supercomputer in National Supercomputing Centre Singapore. The server of the supercomputer has four Xeon E7-4830v3 12 core CPUs and 1TB memory. The number of threads is set to 48 and the cache size is set to 5000. The results are shown in Table V. We can still achieve over 1.2 times speedup by using HCST. Compared with the other strategies, HCST has a superior performance. The effectiveness of our caching strategy is stable regardless of the computational environments.

*4) The impact of the two-level reuses in multi-output tasks: The solver-level reuse of the kernel values can reduce the training time significantly.*

Table VI shows the training time of no cache and the relative time with different kinds of reuses against no cache, where IR denotes the iteration-level reuse and SR denotes the solver-level reuse. On the basis of the original HCST (i.e., only IR), the SR technique can usually further reduce the training time by more than 10%. For *rcv1s2*, our approach can even

TABLE V
TRAINING TIME ON A SERVER OF NSCC SUPERCOMPUTER (SEC)

| dataset | elapsed time (sec) / relative value against no cache | | | | | | speedup of HCST |
|---|---|---|---|---|---|---|---|
| | no cache | HCST | LRU | LFU | LAT | EFU | |
| adult | 26.74 | -22.7% | +0.9% | -11.0% | -2.1% | -20.1% | 1.29 |
| connect-4 | 70.73 | -14.78% | +11.4% | +2.2% | +10.9% | -7.9% | 1.17 |
| mnist | 244.56 | -23.7% | -1.1% | -11.1% | -11.3% | -21.0% | 1.31 |
| webdata | 41.89 | -27.0% | -1.5% | -12.8% | +0.6% | -24.5% | 1.37 |
| real-sim | 60.89 | -38.3% | -16.9% | -31.0% | -27.4% | -35.8% | 1.62 |
| rcv1 | 32.66 | -81.8% | -76.0% | -77.9% | -75.5% | -80.0% | 5.48 |