Scalable and Fast SVM Regression using Modern Hardware

Zeyi Wen*, Rui Zhang*, Kotagiri Ramamohanarao*, Li Yang#[1]
wenzeyi@gmail.com, {rui.zhang, kotagiri}@unimelb.edu.au,
hbyangli@hue.edu.cn

**Department of Computing and Information Systems, The University of Melbourne, VIC, Australia*

#Department of Computer Science, HuBei University of Education, Wuhan, P.R. China

---

Abstract

Support Vector Machine (SVM) regression is an important technique in data mining. The SVM training is expensive and its cost is dominated by: (i) the kernel value computation, and (ii) a search operation which finds extreme training data points for adjusting the regression function in every training iteration. Existing training algorithms for SVM regression are not scalable to large datasets because: (i) each training iteration repeatedly performs expensive kernel value computations, which is inefficient and requires holding the whole training dataset in memory; (ii) the search operation used in each training iteration considers the whole search space which is very expensive. In this article, we significantly improve the scalability and efficiency of SVM regression by exploiting the high performance of Graphics Processing Units (GPUs) and solid state drives (SSDs). Our key ideas are as follows. (i) To reduce the cost of repeated kernel value computations and avoid holding the whole training dataset in the GPU memory, we precompute all the kernel values and store them in the CPU memory extended by the SSD; together with an efficient strategy to read the precomputed kernel values, reusing precomputed kernel values with an efficient retrieval is much faster than computing them on-the-fly. This also alleviates the restriction that the training dataset has to fit into the GPU memory, and hence makes our algorithm scalable to large datasets, especially for large datasets with very high dimensional-

[1]Li Yang and Zeyi Wen are corresponding authors.

ity. (ii) To enhance the performance of the frequently used search operation, we design an algorithm that minimizes the search space and the number of accesses to the GPU global memory; this optimized search algorithm also avoids branch divergence (one of the causes for poor performance) among GPU threads to achieve high utilization of the GPU resources. Our proposed techniques together form a scalable solution to the SVM regression which we call SIGMA. Our extensive experimental results show that SIGMA is highly efficient and can handle very large datasets which the state-of-the-art GPU-based algorithm cannot handle. On the datasets of size that the state-of-the-art GPU-based algorithm can handle, SIGMA consistently outperforms the state-of-the-art GPU-based algorithm by an order of magnitude and achieves up to 86 times speedup.

Keywords: Regression, Support Vector Machines, GPUs, SSDs

1. Introduction

Support Vector Machine (SVM) regression [28] is a widely used machine learning model in many real world applications, such as financial time series forecasting [20], face recognition [21] and outlier detection [18]. Training the SVM regression model requires finding an optimal function that is consistent with as many of the training data points as possible and meanwhile, that is as smooth as possible, such that the function can predict the unseen data more accurately. To find a non-linear regression function, SVMs use a *kernel function* [26] to map the training data points from the original data space to a higher dimensional data space where an optimal function may exist. In each training iteration of the regression[2], the kernel function is computed to obtain kernel values which are used to check whether the currently found function is optimal. If the optimal condition is not met, the search operations are performed to obtain extreme training data points which are used to adjust the currently found regression function.

The SVM training is expensive and its cost is dominated by: (i) the kernel value computation, and (ii) a search operation which finds extreme training data points for adjusting the regression function in every training iteration. Due to a large number of expensive kernel value computations

[2]When the context is clear, we omit “SVM” in the rest of this article, similarly for the SVM training

(time complexity of kernel value computations in each training iteration is $\mathcal{O}(nd)$, where n is the number of training data points; d is the number of dimensions.) and the search operations (time complexity of $\mathcal{O}(n)$), CPU-based training algorithms do not scale to large datasets. As we will discuss in Section 2, MapReduce is not suited to the SVM training (reasons are given in Section 2.2). Hence, recent studies [5, 2] attempt to use Graphics Processing Units (GPUs) to accelerate the training to handle large datasets. However, existing GPU-based training algorithms are not scalable because: (i) each training iteration needs to perform expensive kernel value computations, which is inefficient and also requires holding the whole training dataset in the GPU memory; the high computation cost and memory size constraint make the algorithms unsuitable to large datasets; (ii) the search operation used in each training iteration considers the *whole* search space, which is expensive. Even using a high-end NVIDA GPU, GTX 780, which has 3GB GPU memory and assuming 4 bytes to store a value, existing GPU-based algorithms can slowly process a dataset of only 40,000 data points with 20,000 dimensions (each dimension of a data point 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 data points of 1,000,000 dimensions), the Webspam dataset (350,000 data points of 16,609,143 dimensions) and the Gas Sensor dataset (18,000 data points of 1,950,000 dimensions)[3]. Note that it is inefficient for those GPU-based algorithms to store the datasets in the CPU memory, because transferring the whole dataset from the CPU memory[4] to the GPU memory for computing kernel values in each training iteration is too expensive.

As mining large datasets is becoming more common, the demand for an efficient and scalable training algorithm for the SVM regression is compelling. In this article, we present an efficient solution by exploiting the high performance of modern hardware, particularly the GPU as in the case of the state-of-the-art method and solid state drives (SSDs). We call our solution **S**vm regress**I**on usin**G** **M**odern h**A**rdware (SIGMA). This article is an extension of our earlier conference paper [33]. There we proposed the MASCOT scheme for SVM cross-validation in the SVM *classification* settings and made

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

[4]To distinguish from the GPU memory, we use "the CPU memory" instead of "main memory" in this article.

the following contributions.

- To reduce the cost of repeated kernel value computations and *avoid storing the whole training dataset in the memory*, we precomputed the kernel values, store them to main memory extended by SSDs, and reuse them; together with smart reuse strategies, reusing the kernel values is much faster than computing them on-the-fly. This also alleviates the restriction that the training dataset has to fit into the GPU memory, and hence makes our algorithm scalable.

- To further improve the efficiency of the training, we proposed efficient approaches to reading kernel values in parallel, a caching strategy well-suited to kernel values' access pattern, and an efficient GPU-based search algorithm.

- We conducted extensive experiments to evaluate the performance of MASCOT.

In this article, we extend our work by making the following additional contributions.

- We design a highly optimized search algorithm called "Tight Search" to improve the search operation; Tight Search minimizes the search space and the number of accesses to the GPU global memory which has high-latency, and avoids branch divergence among GPU threads to achieve high utilization of the GPU resources (Section 4.2.2).

- We extend our MASCOT scheme to support the SVM regression, and make use of the properties of the SVM regression to efficiently compute and store a compact kernel matrix (Section 4).

- We conduct extensive experiments to evaluate the performance of SIGMA. Our experimental results show that SIGMA is efficient and scalable to large datasets which the state-of-the-art GPU-based training algorithm cannot handle. On the datasets of size that the state-of-the-art algorithm can handle, SIGMA consistently outperforms the state-of-the-art algorithm by an order of magnitude. Furthermore, we have conducted experiments using LibSVM, the CPU-based algorithm. We have found that LibSVM is extremely slow, so we integrate our kernel value precomputation technique into LibSVM and compare LibSVM with kernel value precomputation with SIGMA. The results show that SIGMA significantly outperforms LibSVM with kernel value precomputation by up to 82 times (Section 5).

The remainder of this article is organized as follows. We review the related literature in Section 2 and present the SVM regression and describe the necessary preliminaries in Section 3. Then, we elaborate our solution SIGMA in Section 4, and report our experimental study in Section 5. Finally, we conclude this article in Section 6.

2. Related work

In this section, we review the related work in the SVM regression. Smola and Scholkopf provided an excellent tutorial on the SVM regression in [28]. Here, we first focus on the training for the SVM regression and categorize the existing work into three groups: (i) SVM training using CPUs; (ii) SVM training using MapReduce and (iii) SVM training using GPUs. Then, we present the existing work in kernel value caching.

2.1. SVM training using CPUs

2.1.1. CPU-based sequential SVM training

Osuna et al. [24] proposed a decomposition based SVM training algorithm. The algorithm divides the training data points into chunks (whose size is typically determined by the memory size). In each iteration, one of the chunks is used to update the optimality indicators. The process continues until convergence. SVM^{light} [16] is an implementation of SVMs based on Osuna et al.'s algorithm.

Platt [25] proposed the *Sequential Minimal Optimization* (SMO) algorithm to train SVMs for classification problems. The algorithm only uses two training data points to update the optimality indicators for all the training data points in each iteration. Flake and Lawrence [11] generalized the SMO algorithm to handle the SVM regression problems. Fan et al. [10] proposed the second order heuristic to select the two training data points in SMO, which achieves an even faster convergence in the SVM training. LibSVM [6] is an implementation of SVMs based on SMO using the second order heuristic.

Another SVM training algorithm [17] uses the cutting-plane approach to improve the training efficiency, but that algorithm only applies to linear

SVMs. Shai et al. [27] proposed a training algorithm called "Pegasos" which demonstrates advantage on training linear SVMs. These studies differ from ours, as our goal is to propose a scalable and efficient training algorithm that can handle linear or non-linear kernels.

2.1.2. CPU-based parallel SVM training

We use GPUs instead of a multi-core CPU because the SVM training is expensive and to achieve a highly efficient SVM training algorithm requires a very high level of parallelism. The SVM training is expensive for the following three reasons (n and d are the cardinality and the dimensionality of the training dataset, respectively). First, the SVM training requires computing the kernel values which is expensive. More specifically, the time complexity of computing the kernel values on-the-fly in each training iteration is $\mathcal{O}(nd)$; the time complexity of the kernel value precomputation is $\mathcal{O}(n^2d)$. Second, the search for the minimum/maximum value in the SVM training has the time complexity of $\mathcal{O}(n)$. Third, the SVM training requires updating all the optimality indicators which has the time complexity of $\mathcal{O}(n)$. The SVM training is very expensive, especially when the data dimension is high or the data cardinality is large. To achieve a highly efficient SVM training algorithm requires a very high level of parallelism. Therefore, GPUs with a large number of cores are more favorable than the CPU with only a few cores in the SVM training.

Some studies [37, 15] attempt to use a multi-core CPU to accelerate the SVM training. As reported in the studies, the multi-core CPU-based algorithms achieve upto six times speedup compared with LibSVM. The speedup of the GPU-based algorithms [2, 8] is more than ten times, which indicates that GPUs are better suited for the SVM training.

2.2. SVM training using MapReduce

MapReduce (both CPU-based and GPU-based [13]) is not suited for the SVM training. The reason is that the SVM training is a global optimization process. Partitioning training data points causes that the support vectors obtained by each mapper are only based on a partition of training data points. 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 SVM training process to meet the optimal condition (i.e., the combined support vectors obtained in the current round of MapReduce training job are the same as those in the previous round), which results in a large amount of repeated computation. Previous studies attempt using MapReduce to accelerate the training either by sacrificing the accuracy or by only handling some special cases. The recent studies [3, 12] attempt to improve the training efficiency by producing approximated results. Catak and Balaban [4] used multiple rounds of the MapReduce training to gradually make the approximation more accurate until the globally optimal SVM is obtained, but this method cannot train SVM with large number of support vectors and does not guarantee convergence. Our study aims at achieving a scalable and efficient training algorithm without approximation.

2.3. SVM training using GPUs

GPUs are a good platform for training SVMs because (i) GPUs have high computation capability to carry out the expensive computation tasks (e.g., kernel value computation and search operation), and (ii) the communication cost in a machine with GPUs is much lower compared with the communication cost among different machines through networks.

Catanzaro et al. [5] proposed a GPU-based SVM training algorithm for classification using SMO. Carpenter [2] extended Catanzaro et al.'s algorithm for handling SVM regression problems. Since Carpenter's algorithm is the state-of-the-art GPU-based training algorithm for the SVM regression, we use the algorithm as our baseline algorithm which will be detailed in the next section.

Other studies have different goals or settings from ours. For example, Cotter et al. [8] proposed a GPU-tailored approach to train SVMs for classification using a clustering technique. Their approach is designed for datasets that are compressible and can be stored entirely in the GPU memory. Athanasopoulos et al. [1] used the GPU to precompute the kernel matrix to improve the efficiency of the SVM cross-validation. Their algorithm can be improved in the following two aspects: (i) the level of parallelism is low, since only the kernel value computation is parallelized; (ii) it is not scalable to large datasets due to the assumption that the whole kernel matrix can be stored in the CPU memory. Codreanu et al. [7] developed a GPU-based SVM training algorithm that requires holding the training dataset in the GPU memory and uses approximation. In this article, we are interested in improving the efficiency and scalability of the SVM training for regression without approximation.

2.4. 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} [16] and LibSVM adopt the Least Recently Used (LRU) replacement strategy for caching kernel values between different iterations. Our later analysis shows that LRU is not efficient for the training because it does not match the kernel values' access pattern. Caching strategy proposed by Yang et al. [35] replaces the kernel values of the data points with weights of 0 in the current iteration. This replacement strategy, however, requires a linear search to find the data point whose weight equals to 0, which is even lesser efficient than LRU.

3. Preliminaries

In this section, we first describe the SVM regression problem and provide an overview of hardware characteristics of SSDs and GPUs. Then, we present the details of a GPU-based parallel reduction process which is an important operation in the GPU-based training algorithm. Finally, we discuss the state-of-the-art GPU-based training algorithm for the SVM regression.

3.1. SVM regression

Given a set $\mathcal{X}$ of training data points, $\mathcal{X} = \{\boldsymbol{x}|\boldsymbol{x}_i \in \mathbb{R}^d, i = 1, ..., n\}$, where n denotes the dataset cardinality and d denotes the data dimensionality. Each training data point is associated with a target value $z \in \mathbb{R}$. The SVM regression is a function estimation process that finds an optimal function $g(\boldsymbol{x})$ which minimizes the difference of the target value of each training data point $\boldsymbol{x}'$ and the function value $g(\boldsymbol{x}')$. Meanwhile, the function $g(\boldsymbol{x})$ is as smooth as possible for achieving high accuracy in predicting unseen data. The training process of the SVM regression is equivalent to solving the following Quadratic Programming (**QP**) problem [22].

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

here $F(\boldsymbol{\alpha})$ is the objective function, and $\boldsymbol{\alpha} = \langle \alpha_1, \alpha_2, ..., \alpha_{2n} \rangle$ is a weight vector for the training data points. The value α_i denotes the contribution of a training data point $\boldsymbol{x}_t$ to the estimated function, where $t = i$ if $i \leq n$, otherwise $t = i - n$. C is a penalty parameter which trades the generality for the accuracy on the training set $\mathcal{X}$. $\mathbf{Q}$ is a positive semi-definite matrix, where $\mathbf{Q} = [Q_{ij}]$, $Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a kernel value computed from a kernel function (e.g., Gaussian kernel, $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = exp\{-\gamma ||\boldsymbol{x}_i - \boldsymbol{x}_j||^2\}$). Here,

$$\begin{cases} y_i = +1, \ s_i = \varepsilon - z_i & \text{if } i \leq n \\ y_i = -1, \ s_i = \varepsilon + z_i & \text{if } n < i \leq 2n \end{cases}$$

and ε is the error tolerant parameter.

The QP problem can be considered as implicitly extending n training data points with target values in $\mathcal{X}$ to $2n$ training data points with ± 1 labels. All the kernel values together form a $2n \times 2n$ matrix shown as follows.

$$\begin{pmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & K(\boldsymbol{x}_1, \boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_1, \boldsymbol{x}_{2n}) \\ K(\boldsymbol{x}_2, \boldsymbol{x}_1) & K(\boldsymbol{x}_2, \boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_2, \boldsymbol{x}_{2n}) \\ \vdots & \vdots & \vdots & \vdots \\ K(\boldsymbol{x}_i, \boldsymbol{x}_1) & K(\boldsymbol{x}_i, \boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_i, \boldsymbol{x}_{2n}) \\ \vdots & \vdots & \vdots & \vdots \\ K(\boldsymbol{x}_{2n}, \boldsymbol{x}_1) & K(\boldsymbol{x}_{2n}, \boldsymbol{x}_2) & \dots & K(\boldsymbol{x}_{2n}, \boldsymbol{x}_{2n}) \end{pmatrix}$$

After the training, we obtain the regression function:

$$g(\boldsymbol{x}) = \sum_{i=1}^{2n} \alpha_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b \tag{2}$$

where b denotes a constant (called bias) that is obtained from the solution of the QP problem.

3.1.1. The SMO algorithm

The SVM training is to find a weight vector $\boldsymbol{\alpha}$ that maximizes the objective function $F(\boldsymbol{\alpha})$ in the QP problem. There are many solvers for the SVM training problem. Here, we focus on one of the most popular SVM training algorithms, the Sequential Minimal Optimization (SMO) algorithm, with the second order heuristic on choosing two training data points in each training iteration [10]. The key idea in the SMO algorithm is to repeatedly update the weight vector by the following three steps until convergence.

Step 1: To find the two training data points $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$ that violate the *Karush-Kuhn-Tucker* conditions maximally, the elements in the *optimality* *indicator vector* $\boldsymbol{f} = \langle f_1, f_2, ..., f_{2n} \rangle$ is divided into three subsets:

$\mathcal{P}_{+1} = \{f_i : (\alpha_i = 0 \wedge y_i = +1) \vee (\alpha_i = C \wedge y_i = -1)\}$

$\mathcal{P}_0 \ \ = \{f_i : 0 < \alpha_i < C\}$

$\mathcal{P}_{-1} = \{f_i : (\alpha_i = 0 \wedge y_i = -1) \vee (\alpha_i = C \wedge y_i = +1)\}$

An optimality indicator f_i is computed using:

$$f_i = \sum_{j=1}^{2n} \alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) + s_i$$

Let $\mathcal{P}_{up} = \mathcal{P}_{+1} \cup \mathcal{P}_0$ and $\mathcal{P}_{low} = \mathcal{P}_0 \cup \mathcal{P}_{-1}$. It has been proven [24] that the indexes of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by u and l respectively, can be computed as follows.

$$u = \underset{i}{\text{argmin}}\{f_i | f_i \in \mathcal{P}_{up}\} \tag{3}$$

$$l = \underset{i}{\text{argmax}}\{\frac{(f_u - f_i)^2}{\eta_i} | f_u < f_i, f_i \in \mathcal{P}_{low}\} \tag{4}$$

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

Step 2: The weights of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by α_u and α_l, are updated using the following formulas.

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

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

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. Specifically, if α'_u (or α'_l) is smaller than 0, we set α'_u (or α'_l) to 0 and adjust α'_l (or α'_u) to ensure the condition $\sum_{i=1}^{2n} y_i \alpha_i = 0$ is satisfied; if α'_u (or α'_l) is larger than C, we set α'_u (or α'_l) to C and adjust α'_l (or α'_u) to ensure the condition $\sum_{i=1}^{2n} y_i \alpha_i = 0$ is satisfied.

Step 3: All the optimality indicators are updated. The optimality indicator f_i of $\boldsymbol{x}_i$ is updated to the new value f'_i using the following formula:

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

The SMO algorithm repeats the above steps until convergence, i.e., $f_u \geq f_{max}$ where,

$$f_{max} = max\{f_i | f_i \in \mathcal{P}_{low}\} \tag{8}$$

Algorithm 1 summarizes the training process of SMO. 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 for SVM regression

```
   Input:  A training set with $2n$ data points
   Output: An optimal weight vector $\boldsymbol{\alpha}$
 1 for $i \leftarrow 1$ to $2n$ do                         /* initialize $\boldsymbol{\alpha}$ and $\boldsymbol{f}$ */
 2     $\alpha_i \leftarrow 0$, $f_i \leftarrow y_i \cdot s_i$
 3 repeat
 4     search for $f_u$ and find $u$                       /* Equation 3 */
 5     compute kernel values $\boldsymbol{\mathcal{K}}_u$
 6     search for $f_l$ and find $l$                       /* Equation 4 */
 7     compute kernel values $\boldsymbol{\mathcal{K}}_l$
 8     update $\alpha_u$ and $\alpha_l$                    /* Equations 5 and 6 */
 9     update $\boldsymbol{f}$                             /* Equation 7 */
10     search for $f_{max}$                                /* Equation 8 */
11 until $f_u < f_{max}$;
```

3.2. SSDs and GPUs

The solid state drive (SSD) is an emerging high-performance storage device. The major difference between the SSD and the hard disk drive (HDD) is that the SSD has no moving mechanical components. The random access and sequential access of the SSD are typically more than 10 times and 3 times faster than the HDD [36], respectively. Another feature of the SSD is that its write speed is usually slower than its read speed. In addition, the SSD consists of a number of blocks, each of which is 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 [34]. Other types of NVRAM [19] are not as widely available as SSDs, and also not as large as SSDs. Therefore, we consider main memory extended by SSDs in this article, such that our algorithms can be easier to be used by others and also can handle bigger datasets.

The Graphics Processing Unit (GPU) has played an important role in personal computers for image rendering tasks. In recent years, as the GPU has become more programmable, many general purpose computing applications [31, 32, 29] have benefited from its high memory bandwidth and high parallel computation capability. Different types of memories have very different access latencies. For example, accessing shared memory is about 7 times faster compared with accessing global memory, while accessing shared memory is about 6 times slower than the registers [30]. The size of the shared memory is much smaller compared with the size of the global memory on a GPU (typically, KB vs MB or GB). Therefore, making efficient use of the shared memory has a significant impact on the performance of GPU-based algorithms. Additionally, the GPU has many cores which have greater potential to enhance the speed of algorithms. For example, a commodity GPU (e.g., GTX 460) which has hundreds of cores allows thousands of light weight threads to execute simultaneously; a high-end GPU (e.g., GTX 780) which has over a thousand cores can run even more threads. Threads running on the GPU cores are organized in a hierarchically grouped manner. In the NVIDIA *Compute Unified Device Architecture* (CUDA) [23], the GPU threads are grouped into blocks which further form a grid. In a block, threads are executed in groups of 32 parallel threads called *warps*. A warp executes one common instruction at a time, so the best efficiency is achieved when all 32 threads of a warp are on the same execution path. If threads of a warp diverge to different paths, the warp serially executes each branch path. A warp diverging to different paths is called *branch divergence* which degrades the performance of GPU-based algorithms.

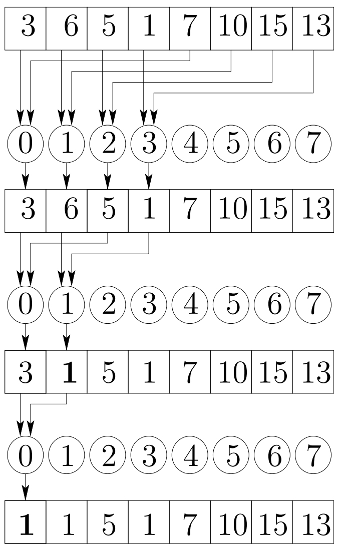
3.3. The standard search algorithm

The standard search algorithm (denoted by Loose Search) has wide use in finding the minimum/maximum value from an array. At each round of the reduction, a GPU thread compares two values and discards the larger (smaller) one to find the minimum (maximum) value. The reduction continues until only one value remains, which is the minimum (maximum) value.

Figure 1a shows the process of finding the minimum value from an array which contains 8 elements, where circles represent GPU threads and rectangles represent elements in the shared memory of the GPU. At the first round, threads 0 to 3 are active and 4 to 7 are idle. Each active thread reads two values, compares them and writes the smaller one back to the shared memory. Then the number of active threads reduces by half, and the active threads continue the reduction process until only one active thread left which obtains the minimum value.

3.4. The state-of-the-art GPU-based algorithm

As discussed in Section 2, Carpenter's algorithm [2] is the state-of-the-art GPU-based training algorithm for the SVM regression using any kernel. We refer to it as "GSVR" and use it as our baseline algorithm in the experimental study. *Please note that GSVR is a GPU-based SMO implementation.* The


(a) Loose Search

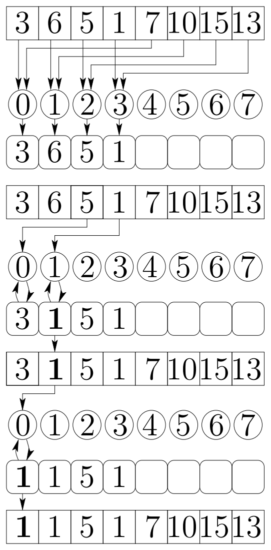

(b) aLoose Search

Figure 1: Two GPU-based search algorithms

main idea of GSVR is to use GPUs to improve two expensive operations, the kernel value computation and the search for two extreme optimality indicators, which are described as follows.

Obtaining the kernel values: In each training iteration, two rows of the kernel matrix—$\boldsymbol{\mathcal{K}}_u = \langle K(\boldsymbol{x}_u, \boldsymbol{x}_1),\ K(\boldsymbol{x}_u, \boldsymbol{x}_2),\ ...,\ K(\boldsymbol{x}_u, \boldsymbol{x}_{2n})\rangle$ and $\boldsymbol{\mathcal{K}}_l = \langle K(\boldsymbol{x}_l, \boldsymbol{x}_1),\ K(\boldsymbol{x}_l, \boldsymbol{x}_2),\ ...,\ K(\boldsymbol{x}_l, \boldsymbol{x}_{2n})\rangle$—are computed on-the-fly. This computation requires holding the whole training dataset in the GPU memory. Furthermore, the same row of kernel matrix may be used for multiple times during the training, and hence computing the rows on-the-fly may result in a large number of repeated computations. Although GSVR attempts to cache some kernel values in the GPU memory using LRU to mitigate the repeated computations, the LRU replacement strategy is ill-suited to the SVM training as we will show in the next section.

Search for extreme optimality indicators: GSVR searches the whole optimality indicator vector $\boldsymbol{f}$ to find the extreme optimality indicators (i.e., to find u, l and f_{max}) using Loose Search (shown in Figure 1a). This is inefficient because it has large search space which causes the following problem. As not all the elements[5] in $\boldsymbol{f}$ can be the candidates of f_u, f_l or f_{max}, GSVR has to

[5]Without confusion, we use "an element" and "an optimality indicator" in the opti-

8		10		5				2		$\mathcal{P}_{+1}$
	3						9			$\mathcal{P}_0$
			7		1	6			4	$\mathcal{P}_{-1}$

(a) Optimality indicators in $\boldsymbol{f}$

8	3	10	$+\infty$	5	$+\infty$	$+\infty$	9	2	$+\infty$

(b) Searching for f_u

$-\infty$	3	$-\infty$	7	$-\infty$	1	6	9	$-\infty$	4

(c) Searching for f_{max}

Figure 2: Search two extreme indicators

identify and mark the non-candidate elements during the search. This results in the following problems: (i) a large number of accesses to the GPU global memory to read the values of $\boldsymbol{y}$ and $\boldsymbol{\alpha}$ to identify non-candidate elements; (ii) branch divergence in identifying candidate and non-candidate elements and (iii) using extra shared memory to store non-candidate elements during the search process.

Let us consider an example shown in Figure 2. There are ten elements in $\boldsymbol{f}$ (in Figure 2a), where each element is contained inside the solid line rectangle. The dashed lines indicate which subset an optimality indicator belongs to. An element appearing in the top, middle or bottom part of a rectangle indicates that the element belongs to $\mathcal{P}_{+1}$, $\mathcal{P}_0$ or $\mathcal{P}_{-1}$. In this example, $\mathcal{P}_{+1} = \{8, 10, 5, 2\}$, $\mathcal{P}_0 = \{3, 9\}$, $\mathcal{P}_{-1} = \{7, 1, 6, 4\}$. To find f_u, GSVR sets all the elements in $\mathcal{P}_{-1}$ to $+\infty$ (cf. Figure 2b) and then uses Loose Search to obtain the minimum value. Similarly, to find f_{max} in $\boldsymbol{f}$, GSVR sets all the elements in $\mathcal{P}_{+1}$ to $-\infty$ (cf. Figure 2c) and then uses Loose Search to obtain the maximum value.

Loose Search for finding extreme optimality indicators is not efficient because: (i) checking whether an optimality indicator f_i is a candidate of f_u, f_l or f_{max} results in branch divergence among GPU threads and also requires accesses to the high-latency GPU global memory for obtaining the values of y_i and α_i; (ii) the shared memory stores $+\infty$ or $-\infty$ to represent non-candidate elements, consuming more shared memory. To summarize, GSVR

mality indicator vector interchangeably

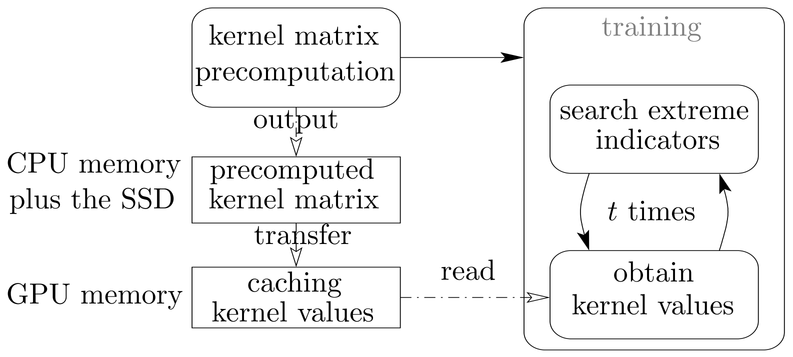


Figure 3: The SIGMA solution

has three major drawbacks.

- **Drawback 1**: GSVR has many repeated computations and requires holding the whole dataset in the GPU memory during the training. As a result, it cannot process datasets larger than the GPU memory.
- **Drawback 2**: GSVR uses a general purpose caching strategy not exploiting kernel values' access pattern.
- **Drawback 3**: GSVR uses Loose Search which searches a space more than necessary and results in a large number of accesses to the GPU global memory, branch divergence among GPU threads and consuming more shared memory.

4. Our solution

In this section, we elaborate our scalable and efficient solution to the SVM regression. To achieve high scalability and efficiency, we precompute the kernel matrix to avoid repeated kernel value computations and *avoid holding the whole training dataset in the GPU memory* during the training. To further improve the efficiency, we organize the optimality indicator vector in a way that a consecutive part of the vector *only* contains all the candidates of f_u, f_l or f_{max}, such that the search does not need to identify non-candidates and hence has a tighter search space. We call our solution Svm regressIon usinG Modern hAreware (SIGMA). Figure 3 shows the overview of SIGMA. As we see from the figure, SIGMA mainly consists of (i) precomputing and storing the kernel matrix (shown in the left part of the figure), and (ii) obtaining kernel values and search extreme optimality indicators in the training (shown in the right part of the figure).

4.1. Precomputing, storing & obtaining kernel values

Here, we explain the properties of the kernel matrix in the SVM regression, and the details of precomputing the kernel matrix, storing the kernel matrix to the SSD and obtaining a row of the kernel matrix.

4.1.1. A compact kernel matrix for the SVM regression

As we have shown in Section 3.1, the whole kernel matrix in the SVM regression is a $2n \times 2n$ matrix and the n training data points are implicitly doubled in the SMO algorithm. Hence, the $2n \times 2n$ kernel matrix can be viewed as a matrix that has four identical blocks. The following matrix gives an overview of the $2n \times 2n$ matrix.

$$\left(\begin{array}{ccc|ccc} K(\boldsymbol{x}_1,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_1,\boldsymbol{x}_n) & K(\boldsymbol{x}_1,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_1,\boldsymbol{x}_n) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ K(\boldsymbol{x}_n,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_n,\boldsymbol{x}_n) & K(\boldsymbol{x}_n,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_n,\boldsymbol{x}_n) \\ \hline K(\boldsymbol{x}_1,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_1,\boldsymbol{x}_n) & K(\boldsymbol{x}_1,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_1,\boldsymbol{x}_n) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ K(\boldsymbol{x}_n,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_n,\boldsymbol{x}_n) & K(\boldsymbol{x}_n,\boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_n,\boldsymbol{x}_n) \end{array}\right)$$

To compute and store the $2n \times 2n$ kernel matrix, we only need to compute and store an $n \times n$ matrix. Therefore, we only need to compute and store a quarter of the kernel matrix to represent the whole kernel matrix. The main operation in precomputing the kernel matrix is essentially matrix multiplication [33]. Hence, the highly optimized cublasSgemm procedure for matrix multiplication in the CUDA library [9] can be used for the kernel matrix precomputation.

We also notice that the $n \times n$ matrix is symmetric. Technically speaking, we can further compress the $n \times n$ matrix, and store an upper triangular matrix to save more space. However, we choose to store an $n \times n$ matrix for the following two reasons. First, recovering a row of the $n \times n$ matrix from the upper triangular matrix requires more read operations to SSDs; those read operations to recover the row is difficult to benefit from the multi-channel functionality of SSDs (cf. Section 3.2). Second, the recovering process introduces much implementation complexity to the algorithm. To make use of the multi-channel functionality and reduce the implementation complexity of the algorithm, we store the $n \times n$ matrix to represent the whole kernel matrix.

4.1.2. Precomputing the kernel matrix

To avoid holding the whole training dataset and avoid repeated kernel value computations, we precompute the kernel matrix for all the training data points using the GPU. If the training dataset is too large to be held in the GPU memory, we partition the dataset into subsets and compute submatrices using the subsets to obtain the whole kernel matrix. As we discussed in Section 3.1, the training data points are used as the input to compute kernel values and are not used for other purposes during the training. Therefore, we do not need to hold it in the GPU memory after the kernel matrix precomputation, and hence SIGMA can handle datasets that are much larger than the size of the GPU memory.

4.1.3. Storing the precomputed kernel matrix

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, 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.

To enable fast reading 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" [23]. 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 helps reduce the time of data transfer between the CPU and the GPU.

Second, we store the remaining portion of the precomputed kernel matrix to the SSD in a way that the kernel values can be read in parallel to make use of SSDs' multi-channel feature. Specifically, we divide a row of the kernel matrix into partitions, such that each partition can be stored in an SSD page. The partitions are stored to a number of SSD pages in different SSD blocks, such that different partitions of the row can be read from the SSD blocks simultaneously.

4.1.4. Obtaining kernel values

1) Obtaining a row from the CPU memory or the SSD

During the training, 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. When the needed rows are in the CPU memory, the GPU directly reads the kernel values. If the needed rows are in the SSD, 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 be 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. After that, we create a number of CPU threads each of which is assigned to read a number of SSD pages that contain a partition of the row. Lastly, the read results are put together and transferred to the GPU memory.

Addressing the scalability issue of GSVR: As discussed above, obtaining the needed kernel values does not require any kernel value computation. Hence, our algorithm avoids repeated kernel value computations and does not need to hold the whole training dataset during the training. Consequently, **we overcome Drawback 1 of GSVR**. This optimization is extremely effective in large datasets, especially datasets of very high dimensionality.

As each training iteration only requires two rows ($2n$ kernel values) of the kernel matrix to be held in the GPU memory, the space complexity of SIGMA 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 with GSVR which has a space complexity of $\mathcal{O}(nd)$, we show the largest datasets that GSVR and SIGMA can handle in Table 1, where the size of datasets is measured by the number of instances. Here we assume that GSVR uses a high-end GPU, GTX 780, which has 3GB GPU memory and SIGMA uses a 4TB SSD. We assume 4 bytes to store a value. As we can see from the table, SIGMA can handle much larger datasets than GSVR, especially when the dimensionality is high. The reason for SIGMA'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

Table 1: Scalability comparison

dimensionality	size of datasets that can be handled	
	SIGMA	GSVR
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

datasets while the GPU memory is not extendable.

2) ***Obtaining a row from the cache***

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 for kernel value caching. As we analyzed in our conference paper [33], 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 favors 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 matches the kernel values' access pattern and well-suited to our storage framework, **we overcome Drawback 2 of GSVR**.

4.2. Search for extreme optimality indicators

In addition to obtaining the needed kernel values, the other expensive operation in each training iteration is the search for extreme optimality indicators. We first propose an improved aLoose Search technique, which requires less shared memory and fewer accesses to shared memory and is used in our MASCOT scheme [33]. Then we elaborate our newly proposed Tight Search technique.

4.2.1. An improved Loose Search technique

In Step 1 of SMO, the search for two extreme optimality indicators is essentially the minimal/maximal value search to obtain f_u, f_l and f_{max}. We use the following two techniques for improving the efficiency of the search.

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 a 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 Loose Search which requires storing n values in the shared memory, this method only needs to store $\lceil n/a \rceil$ 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 3.3, 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 [30] 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.

We call this search technique "Advanced Loose Search" and denote it by *aLoose Search*. Figure 1b gives an overview of aLoose Search. The number of registers used in our aLoose Search algorithm is exactly the same as that in Loose Search. In Loose Search for the minimum value (cf. Figure 1a), each active thread needs to use two registers (for temporarily storing the

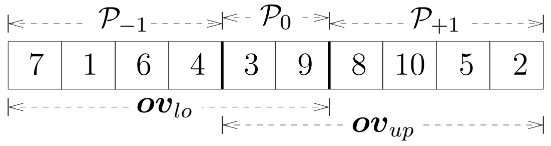


Figure 4: Organized optimality indicator vector

two values read from the shared memory), and writes the smaller value back to the shared memory. In aLoose Search, each active thread also uses two registers, one for storing the newly read value from the shared memory and one for storing the minimum value discovered in the previous round. Then the smaller one among the two values is written back to the register which stores the minimum value discovered. We conduct experiments in Section 5 to investigate the efficiency of aLoose Search.

4.2.2. The Tight Search technique

Our goal here is to reduce the number of accesses to the GPU global memory, avoid branch divergence among the GPU threads, and search as few elements as possible. These factors are vital to the efficiency of the search algorithm. Our key idea is to organize the optimality indicator vector in a way that a consecutive part of the vector *only* contains all the candidates of f_u, f_l or f_{max}, such that the search does not need to identify non-candidates and hence has a tighter search space.

In what follows, we explain the three components of our search algorithm: organizing the optimality indicator vector, searching for the extreme indicators, and maintaining the optimality indicator vector.

1) ***Organizing the optimality indicator vector***

According to the definitions of f_u, f_l and f_{max}, the search for f_u is to find the minimum optimality indicator in $\mathcal{P}_{up}$ (cf. Equation 3), and the search for f_l or f_{max} is to find the maximum value using the optimality indicators in $\mathcal{P}_{low}$ (cf. Equation 4 and 8).

The standard way to search the optimality indicator (e.g., f_u) of interest is to search from all the elements (i.e., $\mathcal{P}_{up} \cup \mathcal{P}_{low}$) in the optimality indicator vector $\boldsymbol{f}$ to find the optimality indicator of interest. Figure 2 gives an example of using the standard way to find f_u and f_{max}.

Instead of searching all the elements (i.e., $\mathcal{P}_{up} \cup \mathcal{P}_{low}$) in $\boldsymbol{f}$ to find the optimality indicators of interest, we organize $\boldsymbol{f}$ in a way that the search only needs to consider either $\mathcal{P}_{up}$ or $\mathcal{P}_{low}$. More specifically, since $\mathcal{P}_{up} = \mathcal{P}_{+1} \cup \mathcal{P}_0$

and $\mathcal{P}_{low} = \mathcal{P}_0 \cup \mathcal{P}_{-1}$, we have $\mathcal{P}_{up} \cap \mathcal{P}_{low} = \mathcal{P}_0$. This property can be exploited to organize the elements of $\boldsymbol{f}$ in three parts: $\mathcal{P}_0$ in the middle part of an array, $\mathcal{P}_{+1}$ at the right side of $\mathcal{P}_0$, and $\mathcal{P}_{-1}$ at the left side of $\mathcal{P}_0$. Figure 4 gives an example of organizing $\boldsymbol{f}$ into three parts. We refer to the **o**rganized optimality indicator **v**ector as $\boldsymbol{ov}$.

Note that $\boldsymbol{ov}$ should be kept in the GPU global memory because of two reasons. First, the search for an extreme optimality indicator is to find an optimality indicator with the globally minimum or maximum value, which requires the GPU threads to be aware of all the optimality indicators. Second, the optimality indicators are frequently accessed, since each training iteration (e.g., Steps 1 and 3 in SMO) requires accesses to all the optimality indicators.

2) ***Searching for the extreme indicators***

When we have the organized optimality indicator vector $\boldsymbol{ov}$ (i.e., organizing the optimality indicator vector $\boldsymbol{f}$ into three parts), we do not need to search in the whole optimality indicator vector $\boldsymbol{f}$ for f_u, f_l and f_{max}. Instead, the search for f_u is to find the minimum optimality indicator in $\boldsymbol{ov}_{up}$ (i.e., $\mathcal{P}_{up}$); the search for f_l and f_{max} is to find the maximum value using $\boldsymbol{ov}_{lo}$ (i.e., $\mathcal{P}_{low}$). Note that $\boldsymbol{ov}_{up}$ equals to $\boldsymbol{f} \setminus \mathcal{P}_{-1}$, and $\boldsymbol{ov}_{lo}$ equals to $\boldsymbol{f} \setminus \mathcal{P}_{+1}$. So, the search space for f_u, f_l and f_{max} is reduced from $\boldsymbol{f}$ to $\boldsymbol{ov}_{up}$ or $\boldsymbol{ov}_{lo}$.

This search for f_u, f_l and f_{max} can be done by our aLoose Search technique and we call this search algorithm "Tight Search", since it has a tighter search space compared with Loose Search and aLoose Search.

3) ***Maintaining the optimality indicator vector***

According to the definitions of $\mathcal{P}_{-1}$, $\mathcal{P}_0$ and $\mathcal{P}_{+1}$ (cf. Section 3.1.1), which subset an optimality indicator f_i belongs to is determined by the label y_i and the weight α_i of the training data point. The labels of the training data points are unchangeable, while the weights may change during the training. The change of weights potentially forces elements to move among $\mathcal{P}_{-1}$, $\mathcal{P}_0$ and $\mathcal{P}_{+1}$. Therefore, we need to move the elements to the correct subsets such that $\boldsymbol{ov}$ *maintains* the property that $\boldsymbol{ov}_{up}$ and $\boldsymbol{ov}_{lo}$ correspond to $\mathcal{P}_{up}$ and $\mathcal{P}_{low}$, respectively. Since each training iteration only updates two weights (i.e., α_u and α_l), there are at most two optimality indicators (i.e., f_u and f_l) in $\boldsymbol{ov}$ which move from one subset to another. The element movement from one subset to another requires accesses (probably not coalesced accesses [23]) to the high-latency GPU global memory, and hence the number of accesses to the GPU global memory should be minimized. We know that the size of $\boldsymbol{ov}$

is equal to the number of the training data points and is a constant during the training. The movement of the elements in the $\boldsymbol{ov}$ vector is an activity inside the vector. Hence, maintaining $\boldsymbol{ov}$ can be considered as swapping optimality indicators among $\mathcal{P}_{-1}$, $\mathcal{P}_0$ and $\mathcal{P}_{+1}$.

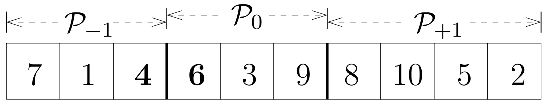
In what follows, we describe a frequently used operation (called *swap operation* that swaps two elements in $\boldsymbol{ov}$) to maintain $\boldsymbol{ov}$'s property and then present an efficient algorithm to maintain $\boldsymbol{ov}$.

The swap operation: The swap operation swaps an element f_i (e.g., f_u) with a boundary element of $\mathcal{P}_{-1}$, $\mathcal{P}_0$ or $\mathcal{P}_{+1}$, such that f_i is moved to the right position in $\boldsymbol{ov}$. For instance, a training iteration forces f_u, the element 6 in Figure 4, to move from $\mathcal{P}_{-1}$ to $\mathcal{P}_0$. This is equivalent to swapping 6 with 4 in $\mathcal{P}_{-1}$. Here, *boundaries* are represented by the bold vertical bars in the figure. The updated $\boldsymbol{ov}$ vector is shown in Figure 5a. This update requires one swap operation. In comparison, moving elements between $\mathcal{P}_{+1}$ and $\mathcal{P}_{-1}$ requires more swap operations. For instance, moving f_l, element 10 in Figure 5a, from $\mathcal{P}_{+1}$ to $\mathcal{P}_{-1}$ requires moving element(s) of $\mathcal{P}_0$. This is because $\mathcal{P}_{+1}$ shrinks while $\mathcal{P}_{-1}$ grows, which forces $\mathcal{P}_0$ to shift. In this example, three elements (i.e., 6, 8 and 10) are moved, which requires two swap operations in total. Figure 5b shows the updated $\boldsymbol{ov}$.

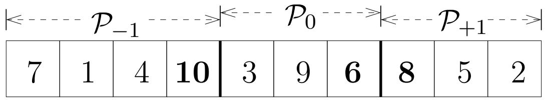
Before we apply the swap operation, we need to compute the new positions of f_u and f_l. This preparation for the swap operation is not parallelizable and is more suitable to be performed on the CPU. As we discussed before, $\boldsymbol{ov}$ is stored in the GPU global memory. Hence, a swap operation requires not only accesses to the high-latency GPU global memory but also communication between the CPU and the GPU. Note that the swap operation happens in each training iteration. Therefore, the number of swap operations has a significant effect on the efficiency of the training. In what follows, we show how we may maintain $\boldsymbol{ov}$ by moving individual elements and how we may maintain $\boldsymbol{ov}$ more efficiently through direct swapping of elements.

Moving individual elements: In each training iteration, at most two elements, i.e., f_u and f_l , in $\boldsymbol{ov}$ are moved. One way to maintain $\boldsymbol{ov}$ is to move the two elements separately, and we call this approach "Single Movement". Figure 6 shows all possible cases of moving an element from one subset to another. In the figure, cases 1 to 4 can be handled by one swap operation and cases 5 and 6 can be handled by two swap operations.

This approach is inefficient, because each iteration may force both f_u and f_l to move and hence the number of swap operations is up to four. Next,



(a) moving 6 from $\mathcal{P}_{-1}$ to $\mathcal{P}_0$



(b) moving 10 from $\mathcal{P}_{+1}$ to $\mathcal{P}_{-1}$

Figure 5: Moving elements in $\boldsymbol{ov}$

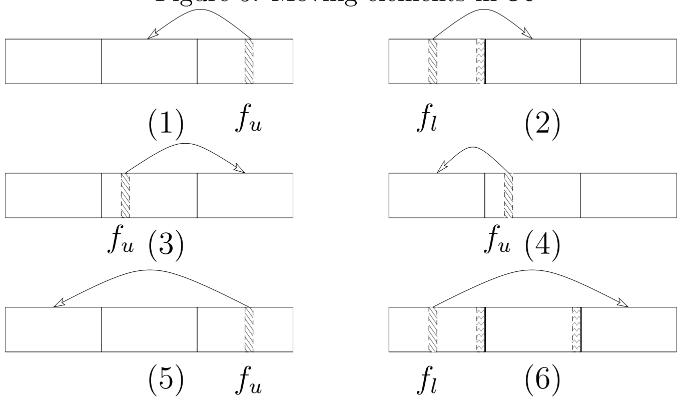


Figure 6: Possible movements of an element

we explore the properties of the training to minimize the number of swap operations.

Direct swapping: As both f_u and f_l may be moved in a training iteration, considering both elements together may be more efficient than considering them separately. Our key idea is to apply the swap operation directly (which we call *direct swapping*) on f_u and f_l when direct swapping can apply. Figure 7 shows possible movements of two elements in $\boldsymbol{ov}$. Direct swapping can apply on f_u and f_l in cases 1 to 3 in Figure 7, and only one swap operation is needed in these cases. Note that moving f_u and f_l separately requires four swap operations for case 3, while direct swapping only needs one swap operation which is much more efficient.

Below we prove that direct swapping can always apply to elements moving between $\mathcal{P}_{+1}$ and $\mathcal{P}_{-1}$ (i.e., cases 5 and 6 in Figure 6).

Lemma 1. *An element moving from $\mathcal{P}_{+1}$ to $\mathcal{P}_{-1}$ is a sufficient and necessary condition for the other element moving from $\mathcal{P}_{-1}$ to $\mathcal{P}_{+1}$.*

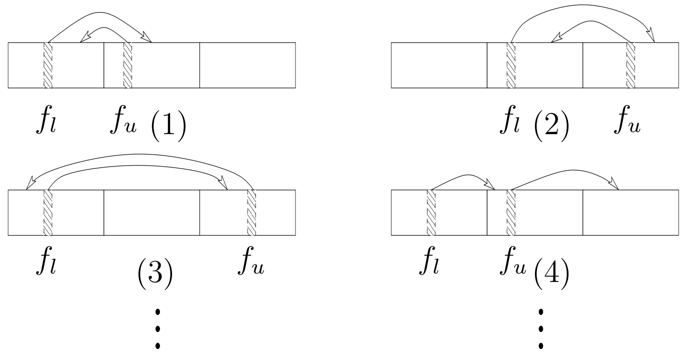


Figure 7: Possible movements of two elements

Proof 1. *To prove its sufficiency, from the constraint $\sum_{i=1}^{2n} y_i\alpha_i = 0$ in the QP problem, we can get the following equation $y_i\alpha_i + y_j\alpha_j = y_i\alpha_i' + y_j\alpha_j'$. Without loss of generality, suppose an element f_i moves from $\mathcal{P}_{+1}$ to $\mathcal{P}_{-1}$. The value of α_i changes from 0 to C if $y_i = +1$, or changes from C to 0 if $y_i = -1$. Therefore, $y_i\alpha_i' - y_i\alpha_i = y_j\alpha_j - y_j\alpha_j' = C$. This indicates that the value of $y_j\alpha_j$ decreases by C. Thus, the element f_j must move from $\mathcal{P}_{-1}$ to $\mathcal{P}_{+1}$. Similarly, we can prove the necessity.*

When direct swapping does not apply, we move the two elements separately. We can guarantee that the maintenance of $\boldsymbol{ov}$ can be done by at most two swap operations. This is because: (i) when direct swapping can apply, the maintenance can be done by one swap operation; (ii) when we need to swap f_u and f_l separately, one swap operation for each element is enough. The above lemma simplifies the maintenance process, as we do not need to handle individual movement in cases 5 and 6.

4) ***Analysis of Tight Search***

The Tight Search algorithm has the following two advantages over the Loose Search algorithm. First, the search space for f_u, f_l and f_{max} is minimized, since only candidates (i.e., elements in $\boldsymbol{ov}_{up}$ or $\boldsymbol{ov}_{lo}$) are taken into consideration; no shared memory is used for storing non-candidates. Second, there is no branch divergence, because Tight Search does not need to access to $\boldsymbol{y}$ and $\boldsymbol{\alpha}$ to check whether an element is a candidate of the extreme optimality or not. Note that whether an optimality indicator f_i is in $\boldsymbol{ov}_{up}$ or $\boldsymbol{ov}_{lo}$ is decided by the label y_i and the weight α_i according to the definition of $\mathcal{P}_{+1}$, $\mathcal{P}_0$ and $\mathcal{P}_{-1}$ in Step 1 of SMO (cf. Section 3.1.1). Hence, **we overcome Drawback 3 of GSVR**.

Additionally, Tight Search is the key to achieve more benefits when the number of support vectors is small. The reasoning is as follows. From the definition of $\boldsymbol{ov}_{up}$ and $\boldsymbol{ov}_{lo}$, we know that $\boldsymbol{ov}_{up}$ equals to $\boldsymbol{f} \setminus \mathcal{P}_{-1}$, and $\boldsymbol{ov}_{lo}$ equals to $\boldsymbol{f} \setminus \mathcal{P}_{+1}$. According to the definition of $\mathcal{P}_0$ in Section 3.1.1, we can find out that $\mathcal{P}_0$ only contains the optimality indicators of the support vectors (i.e., the optimality indicators corresponding to $\alpha \in (0, C)$) of the currently found hyperplane. Similarly, we can find out that $\mathcal{P}_{+1}$ and $\mathcal{P}_{-1}$ contain the optimality indicators of the non-support vectors. $\mathcal{P}_0$ is contained in both $\boldsymbol{ov}_{up}$ and $\boldsymbol{ov}_{lo}$, while $\mathcal{P}_{+1}$ and $\mathcal{P}_{-1}$ are contained in either $\boldsymbol{ov}_{up}$ or $\boldsymbol{ov}_{lo}$. In each iteration of the training, $\boldsymbol{ov}_{up}$ is searched once to find f_u, and $\boldsymbol{ov}_{lo}$ is searched twice to find f_l and f_{max}. So, $\mathcal{P}_0$ is searched three times, $\mathcal{P}_{+1}$ is searched once, and $\mathcal{P}_{-1}$ is searched twice. Therefore, the smaller the size of $\mathcal{P}_0$, the more benefits our Tight Search algorithm has. More intuitively, the conclusion here is that the larger the number of non-support vectors the problem has, the more benefit our Tight Search algorithm has. We will study the effectiveness of Tight Search by experiments in Section 5.

4.2.3. Difference of Tight Search and Loose Search

Tight Search and Loose Search may obtain a different optimality indicator when two optimality indicators have the equal value. Suppose we want to find the minimum value from elements in $\mathcal{P}_{up} = \langle 8, 2, 10, 5, 2 \rangle$. Loose Search gives the first "2", while Tight Search may give the second "2" in $\mathcal{P}_{up}$ due to the organization and maintenance of the optimality indicator vector. Note that choosing a different optimality indicator in a training iteration only affects the convergence rate, but has little effect on the final regression function.

4.3. Distributing tasks on GPUs and CPUs

During the training, some operations (e.g., search extreme indicators) can be parallelized while others are not parallizable. Apart from using the GPU to perform the parallizable operations, we have two techniques for making efficient use of the CPU. First, we use the CPU to update the two weights α_u and α_l according to Equations 5 and 6 (cf. Step 2 of SMO), since the updates are not parallelizable. Second, we use the CPU to compute the two unchanged terms, $(\alpha'_u - \alpha_u)y_u$ and $(\alpha'_l - \alpha_l)y_l$, in each iteration. Then the two terms are passed to the GPU, instead of computing them in every GPU thread.

Algorithm 2: The training algorithm in SIGMA

```
Input:  A training set with $2n$ data points
        the precomputed kernel matrix $\boldsymbol{\mathcal{K}}$
Output: An optimal weight vector $\boldsymbol{\alpha}$
1  $d_{-1} \leftarrow 0$, $d_{+1} \leftarrow 2n$ ;                    /* boundaries in $\boldsymbol{ov}$ */
2  for $i \leftarrow 1$ to $2n$ do                             /* initialize $\boldsymbol{\alpha}$ and $\boldsymbol{ov}$ */
3      $\alpha_i \leftarrow 0$, $f_i \leftarrow y_i \cdot s_i$;
4      if $y_i = +1$ then
5          $\boldsymbol{ov}[d_{+1}] \leftarrow f_i$, $d_{+1} \leftarrow d_{+1} - 1$;
6      else
7          $\boldsymbol{ov}[d_{-1}] \leftarrow f_i$, $d_{-1} \leftarrow d_{-1} + 1$;
8  repeat
9      $u \leftarrow \text{argmin}\{\boldsymbol{ov}[i] : d_{+1} < i \leq 2n\}$, $f_u \leftarrow \boldsymbol{ov}[u]$;
10     $\boldsymbol{\mathcal{K}}_u \leftarrow$ ParallelRead($\boldsymbol{\mathcal{K}}$, $u$)
11     search for $l$ using $\boldsymbol{\mathcal{K}}_u$ and $\{\boldsymbol{ov}[i] : 1 \leq i < d_{-1}\}$
12     $\boldsymbol{\mathcal{K}}_l \leftarrow$ ParallelRead($\boldsymbol{\mathcal{K}}$, $l$)
13     update $\alpha_u$ and $\alpha_l$                          /* Equations 5 and 6 */
14     Maintain($\boldsymbol{ov}$)
15     update $\boldsymbol{ov}$ using $\boldsymbol{\mathcal{K}}_u$, $\boldsymbol{\mathcal{K}}_l$                       /* Equation 7 */
16     $f_{max} \leftarrow \max\{\boldsymbol{ov}[i] : 1 \leq i < d_{-1}\}$
17 until $f_u < f_{max}$;
```

4.4. The full training algorithm

Our full training algorithm is outlined in Algorithm 2. First, the weight vector and organized optimal indicator vector $\boldsymbol{ov}$ are initialized (lines 1 to 7). Then, the index of the first extreme optimality indicator, u, is found in $\mathcal{P}_{up}$, and the corresponding row $\boldsymbol{\mathcal{K}}_u$ is obtained from the precomputed kernel matrix $\boldsymbol{\mathcal{K}}$ (lines 9 and 10). Inside the procedure "ParallelRead", we check if the needed row is in the cache; if a cache miss occurs, we update the cache using the cache replacement strategy discussed in Section 4.1.4, and read the row from the CPU memory or the SSD. Based on the row $\boldsymbol{\mathcal{K}}_u$, the index of the second extreme optimality indicator, l, is found according to Equation 4, and the corresponding row $\boldsymbol{\mathcal{K}}_l$ is obtained from $\boldsymbol{\mathcal{K}}$ (lines 11 and 12). Next, the weights (i.e., α_u and α_l) of the two training data points with extreme optimality indicators are updated (line 13) using Equations 5 and 6. After the update on the weights, the two extreme optimality indicators f_u and f_l may be moved from one subset to another and hence the maintenance process for $\boldsymbol{ov}$ should be performed (line 14). After that, all the optimal indicators in $\boldsymbol{ov}$ are updated using the updated α_u and α_l according to Equation 7 (line 15). Lastly, we search the maximal optimal indicator in $\mathcal{P}_{low}$ and check if the optimal condition $f_u \geq f_{max}$ is met (lines 16 and 17). The above process is repeated until the optimal condition is met.

4.5. Discussion

SIGMA can easily scale to a number (denoted by m) of machines with GPUs, since the kernel matrix precomputation and the training 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 evenly divide the optimality indicator vector $\boldsymbol{f}$ into m parts $\boldsymbol{q}_1$, $\boldsymbol{q}_2$, ..., $\boldsymbol{q}_m$ where $\boldsymbol{q}_i = \langle f_{(i-1)\cdot z+1}, f_{(i-1)\cdot z+2}, \ldots f_{i\cdot z}\rangle$ and z is the number of the optimality indicators in each part. The i^{th} machine stores $\boldsymbol{q}_i$ and organizes its own $\boldsymbol{ov}^i$ using $\boldsymbol{q}_i$. 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

Table 2: Datasets and kernel parameters

Dataset	Cardinality	Dim.	C	γ
Amazon	30,000	20,000	32	0.0625
CT-slices	53,500	386	64	0.25
E2006-tfidf	16,087	150,360	256	0.125
KDD-CUP'98	191,779	479	64	0.25

and independently updates $\boldsymbol{q}_i$ according to Equation 7; the maintenance of the $\boldsymbol{ov}^i$ vector may occur depending on whether f_u or f_l is in $\boldsymbol{q}_i$. Furthermore, the i^{th} machine only needs to store z columns—the $(i \cdot z - z + 1)^{th}$ column to the $i \cdot z^{th}$ column—of the kernel matrix. This is because only the $(i \cdot z - z + 1)^{th}$ to $i \cdot z^{th}$ kernel values of the u^{th} row and the l^{th} row are used to update $\boldsymbol{q}_i$ at each training iteration. As each machine only needs to store z columns of the kernel matrix, SIGMA can handle problems with an even larger kernel matrix using multiple machines with GPUs and SSDs.

5. Experimental study

In this section, we empirically evaluate the performance of our solution SIGMA for the SVM regression. SIGMA and GSVR were implemented in CUDA-C. We did not use any libraries except the libraries (e.g., cublasSgemm) provided by CUDA. The experiments were conducted on a desktop computer running Linux with a quad-core Intel Xeon E5-2643 CPU, 32GB main memory, a 240GB SSD and a GTX460 GPU with 768MB memory. Following a common practice, we set the SSD block size and the SSD page size to 512KB and 4KB, respectively, and the GPU block size to 128B. The source code of SIGMA will be released to GitHub after this work is accepted for publication.

We use four real world datasets from the UCI Machine Learning Repository[6] and the LibSVM site[7]. Specifically, the datasets are: (i) Amazon, which contains an anony-mized sample of access provisioned within the Amazon company and person business title serves as the target value of the regression; (ii) CT-slices, which contains features extracted from CT images, and the relative location of the image on the axial axis serves as the target value; (iii) E2006-tfidf, which contains features of corpus and tf-idf of unigrams,

[6]archive.ics.uci.edu/ml/datasets.html

[7]www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets/

and (iv) KDD-CUP'98, which is the dataset used for the KDD Cup 1998 Data Mining Tools Competition, and their first features serve as the target values. We use the Gaussian kernel in our experiments, and the parameters C and γ of the kernel are chosen using grid search [14]. Dataset details are shown in Table 2.

We conducted experiments on the KDD-CUP'98 dataset using LibSVM (i.e., the CPU-based training algorithm). LibSVM took *over 50 hours* which is two orders of magnitude more time than SIGMA. As LibSVM is too slow and uncompetitive with SIGMA, we do not compare it with SIGMA in experiments using the whole datasets. We postpone the experiments on comparing LibSVM with SIGMA until Section 5.2 when we study the overall efficiency of different algorithms on sub-datasets which GSVR and LibSVM can handle. As we discussed in Section 2, the MapReduce based SVM training algorithms produce approximated results and/or can only handle some special cases, and hence we will not compare them with our solution SIGMA either. Therefore, we compare SIGMA with GSVR, the state-of-the-art training algorithm. Because SIGMA does not need to hold the training data points during the training, it uses the GPU memory to cache kernel values.

For the experimental results on the effectiveness of our caching strategy and Parallel Kernel Value Read, and the efficiency comparison between the CPU-based training algorithm and our GPU-based training algorithm, please refer to our conference version of this article [33]. Here, we focus on showing the experimental results on our newly designed Tight Search algorithm and on the overall scalability and efficiency of SIGMA for the SVM regression.

5.1. Efficiency of Tight Search

We used the whole KDD-CUP'98 dataset to investigate the efficiency of Tight Search. The experimental results on the other datasets are similar to the results on KDD-CUP'98, and hence are omitted in this set of experiments. To demonstrate the efficiency of Tight Search, we used Loose Search and aLoose Search (i.e., advanced Loose Search) discussed in Section 4 as our baselines. We created about 1,500 GPU threads to search f_u (i.e., the first extreme optimality indicator), and measured the total elapsed time in searching for f_u during the training. Figure 8a shows that Tight Search dramatically outperforms Loose Search by *over 4 times*, and aLoose Search by over 3 times. This is because Tight Search minimizes the search space, reduces the number of global memory accesses, and avoids branch divergence as discussed in Section 4. In our experiment, we noticed that the average

Table 3: Datasets the algorithms can handle

Dataset	GSVR	SIGMA	
		elapsed time	read : comp
Amazon	—	420 (sec)	0.39:1
CT-slices	2,731 (sec)	226 (sec)	0.53:1
E2006-tfidf	—	1,789 (sec)	0.23:1
KDD-CUP'98	—	2,628 (sec)	1.60:1

length of $\mathcal{P}_{up}$ is about 69% of the length of $\boldsymbol{ov}$, and the length of $\mathcal{P}_{low}$ is about 62% of that of $\boldsymbol{ov}$. This indicates the significance of the search space reduction. We also noticed that the total elapsed time on the maintenance of $\boldsymbol{ov}$ was 6.5 seconds. It is a small amount of time compared with the time taken by the search operations. Note that not only the search for f_u benefits from the maintenance, but also the search for f_l and f_{max}.

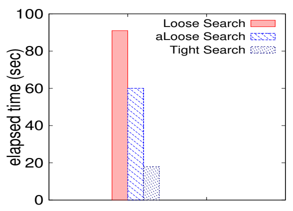
5.2. Overall scalability and efficiency

Please note that the elapsed time of SIGMA reported here includes the time of precomputing and storing kernel values, and the time of training.

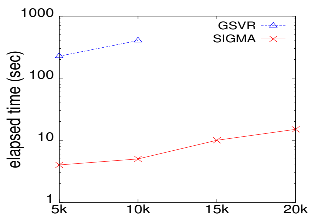
Scalability: As can be seen from the second and the third columns of Table 3, SIGMA can easily process all the four datasets with a reasonably small amount of time, while GSVR can only process the CT-slices dataset taking an order of magnitude more time than SIGMA and fails to process the other three datasets. This is because GSVR assumes all the training data points can be stored in the GPU memory, while SIGMA does not have such constraints.

To examine the time required for the GPU kernel execution and the time required for obtaining the precomputed kernel values, we measured the total time for the GPU kernel execution during the training and measured the total time of obtaining the precomputed kernel values during the training. The results are provided in the last column of Table 3, where "read : comp" denotes the ratio of the time required for obtaining the precomputed kernel values and that for the GPU kernel execution. According to the results, the time required for the GPU kernel execution is longer than that for obtaining the precomputed kernel values in most of the datasets tested. For the KDD-CUP'98 dataset, the time required for the GPU kernel execution is comparable to that for obtaining the precomputed kernel values.

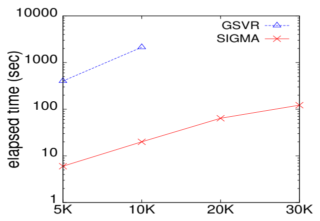
To demonstrate the scalability of SIGMA over the cardinality and dimensionality, we constructed two groups of datasets from the Amazon dataset



(a) Tight Search



(b) Dimensionality



(c) Cardinality

Figure 8: Search and the scalability

as follows. One group has datasets with dimensionality set to 5,000 and cardinality varying from 5,000 to 30,000. The second group has datasets with cardinality set to 5,000 and dimensionality varying from 5,000 to 20,000. Figure 8b shows the results on the scalability in data dimensionality. SIGMA can efficiently process datasets with high dimensions. In contrast, GSVR fails to process the datasets with data dimensionality over 10,000, although it can handle the datasets of smaller dimensionality with much more time than SIGMA. On the scalability in dataset cardinality, Figure 8c shows the results. As the dataset cardinality increases from 5,000 to 10,000, the elapsed time of SIGMA slightly increases from 6 seconds to 20 seconds. In comparison, the elapsed time of GSVR grows dramatically from hundreds of seconds to thousands of seconds. GSVR runs out of memory with datasets of larger cardinality while SIGMA can handle them efficiently.

Efficiency: To accelerate LibSVM with our techniques, we implemented a CPU version of our kernel value precomputation and computation reuse techniques, and integrated the techniques to LibSVM. Note that even though LibSVM has an interface for using precomputed kernel values, LibSVM does not have functions for precomputation. To compare the efficiency of SIGMA with GSVR and LibSVM with precomputation denoted by *CSVR*, we constructed three smaller datasets (both in terms of cardinality and dimensionality) so that GSVR and CSVR can handle the four datasets without running out of GPU memory or running out of our time limit. The details of the subsets are as follows: (i) Amazon has 15,000 data points with 3,500 dimensions, (ii) E2006-tfidf has 16,087 data points with 3,500 dimensions, and (iii) KDD-CUP'98 has 50,000 data points with 479 dimensions. The CT-slices dataset remains unchange as GSVR could process it.

Table 4 gives the elapsed time of GSVR and CSVR, and the total elapsed time—including *the time of precomputing the kernel matrix*—of SIGMA. The

Table 4: Efficiency comparison (sec)

Sub-dataset	GSVR	CSVR	SIGMA	Speedup	
				GSVR	CSVR
Amazon	1,077	2,201	27	39.9	81.5
CT-slices	2,731	4,341	226	12.1	19.2
E2006-tfidf	7,053	2,782	82	86.0	33.9
KDD-CUP'98	639	3,456	54	11.8	64.0

Table 5: Efficiency comparison using Tesla K40 (sec)

Sub-dataset	GSVR	SIGMA	Speedup
Amazon	551	25	22
CT-slices	1,479	127	11.6
E2006-tfidf	3,488	81	43
KDD-CUP'98	336	32	10.5

elapsed time was measured in seconds. As we can see from the table, SIGMA consistently outperforms GSVR and CSVR by an order of magnitude over all the datasets tested.

To investigate the performance gain of our solution in more recent GPUs, we conducted the experiments on overall efficiency using a Tesla K40 and the results are shown in Table 5. As we can see from the results, SIGMA consistently outperforms GSVR by one order of magnitude. These results demonstrate that our solution is more favorable than GSVR for various GPUs. The results of Table 4 and Table 5 indicate the effectiveness of our computation reuse techniques and of Tight Search discussed in Section 4. Please note that the improvement on kernel value computation contributes more than Tight Search to the speedup, because the kernel value computation is time complexity of $\mathcal{O}(nd)$ while the time complexity of the search is $\mathcal{O}(n)$.

5.3. Training result comparison

To validate that the final training results of GSVR and SIGMA are the same, we measured two more indicators in our experiments: the number of training iterations and the value of the bias b of Equation 2. Table 6 shows the number of training iterations and the difference of the bias of GSVR and SIGMA on each dataset tested. The number of training iterations is slightly different (i.e., SIGMA has less than 3% fewer or more training iterations than GSVR). The slight difference is because two extreme optimality indicators

Table 6: Training Result Comparison between GSVR and SIGMA

Dataset	# training iterations		diff. of bias
	GSVR	SIGMA	
Amazon (subset)	106,847	107,508	0
CT-slices	424,681	411,105	0.001
E2006-tfidf (subset)	401,356	402,453	0
KDD-CUP'98 (subset)	82,094	80,414	0

chosen by GSVR and SIGMA may be different in a training iteration as discussed in Section 4.2.3. Please note that the difference on the chosen of two optimality indicators only affects the convergence rate. As can be seen from the table, the bias values are the same in the three datasets, and are almost the same (i.e., the difference is only 0.001) in the CT-slices dataset. In summary, SIGMA is scalable and is more efficient than GSVR, and SIGMA produces the same (or almost the same) result as GSVR.

6. Conclusion

In this article, we proposed a highly scalable and efficient solution (called SIGMA) for the SVM regression, which significantly outperforms the state-of-the-art GPU-based training algorithm. Our key ideas are as follows. (i) To avoid the repeated kernel value computations and avoid holding the whole training dataset in the GPU memory, we precomputed the kernel values and stored them in the CPU memory extended by the SSD. (ii) To obtain kernel values from the CPU memory or the SSD more efficiently, we proposed a caching strategy and a Parallel Kernel Value Read technique; to improve the search operation, we designed a highly efficient algorithm called Tight Search. Our experimental results show that SIGMA is efficient and scalable to very large datasets even with very high dimensionality (e.g., E2006-tfidf with 150,360 dimensions) which the state-of-the-art GPU-based algorithm simply could not handle. For datasets of size which the state-of-the-art GPU-based algorithm can handle, SIGMA consistently outperforms the algorithm by an order of magnitude.

Acknowledgement

Rui Zhang is supported by ARC Future Fellow project FT120100832. This work is partially supported by the National Natural Science Foundation of China (No 61402155). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

References

[1] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. GPU acceleration for support vector machines. In *International Workshop on Image Analysis for Multimedia Interactive Services*, 2011.

[2] Austin Carpenter. Cusvm: A cuda implementation of support vector classification and regression. *patternsonscreen. net/cuSVMDesc. pdf*, 2009.

[3] Godwin Caruana, Maozhen Li, and Man Qi. A MapReduce based parallel SVM for large scale spam filtering. In *The International Conference on Fuzzy Systems and Knowledge Discovery*, volume 4, pages 2659–2662, 2011.

[4] F Ozgur Catak and M Erdal Balaban. CloudSVM: training an SVM classifier in cloud computing systems. In *Pervasive Computing and the Networked World*, pages 57–68. Springer-Verlag, 2013.

[5] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML*, pages 104–111. ACM, 2008.

[6] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[7] Valeriu Codreanu, Bob Dröge, David Williams, Burhan Yasar, Po Yang, Baoquan Liu, Feng Dong, Olarik Surinta, Lambert Schomaker, Jos Roerdink, et al. Evaluating automatically parallelized versions of the SVM. *Concurrency and Computation: Practice and Experience*, 2014.

[8] Andrew Cotter, Nathan Srebro, and Joseph Keshet. A GPU-tailored approach for training kernelized SVMs. In *KDD*, pages 805–813, 2011.

[9] Nvidia CUDA. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008.

[10] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *JMLR*, 6:1889–1918, 2005.

[11] Gary William Flake and Steve Lawrence. Efficient SVM regression training with SMO. *Machine Learning*, 46(1-3):271–290, 2002.

[12] Pedro A Forero, Alfonso Cano, and Georgios B Giannakis. Consensus-based distributed support vector machines. *The Journal of Machine Learning Research*, 99:1663–1707, 2010.

[13] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[14] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification, 2003.

[15] Mingsheng Hu and Weixu Hao. A parallel approach for SVM with multi-core CPU. In *International Conference on Computer Application and System Modeling*, volume 15, pages V15–373. IEEE, 2010.

[16] Thorsten Joachims. Making large-scale SVM learning practical. In *Advances in kernel methods*, pages 169–184. MIT Press, 1999.

[17] Thorsten Joachims. Training linear SVMs in linear time. In *KDD*, pages 217–226, 2006.

[18] Elsa M Jordaan and Guido F Smits. Robust outlier detection using SVM regression. In *IEEE International Joint Conference on Neural Networks*, volume 3, pages 2017–2022. IEEE, 2004.

[19] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jae-hyuk Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *IEEE Transactions on Computers*, 58(6):744–758, 2009.

[20] Kyoung-jae Kim. Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1):307–319, 2003.

[21] Yongmin Li, Shaogang Gong, and Heather Liddell. Support vector regression and classification based multi-view face detection and recognition. In *International Conference on Automatic Face and Gesture Recognition*, pages 300–305. IEEE, 2000.

[22] Jorge Nocedal and S Wright. Numerical optimization, series in operations research and financial engineering. *Springer*, 2006.

[23] CUDA Nvidia. NVIDIA CUDA programming guide, 2011.

[24] Edgar Osuna, Robert Freund, and Federico Girosi. An improved training algorithm for support vector machines. In *IEEE Workshop on Neural Networks for Signal Processing*, pages 276–285. IEEE, 1997.

[25] John C. Platt. Fast training of SVMs using sequential minimal optimization. In *Advances in kernel methods*, pages 185–208. MIT Press, 1999.

[26] Bernhard Scholkopf and Alex Smola. Learning with kernels, 2002.

[27] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical Programming*, 127(1):3–30, 2011.

[28] Alex J Smola and Bernhard Schölkopf. A tutorial on SVM regression. *Statistics and computing*, 14(3):199–222, 2004.

[29] Yu Sun, Nicholas Jing Yuan, Yingzi Wang, Xing Xie, Kieran McDonald, and Rui Zhang. Contextual intent tracking for personal assistants. 2016.

[30] Vasily Volkov. Better performance at lower occupancy. In *The GPU Technology Conference*, volume 10, 2010.

[31] Phillip GD Ward, Zhen He, Rui Zhang, and Jianzhong Qi. Real-time continuous intersection joins over large sets of moving objects using graphic processing units. *The VLDB Journal*, 23(6):965–985, 2014.

[32] Zeyi Wen, Rui Zhang, and Kotagiri Ramamohanarao. Enabling precision/recall preferences for semi-supervised svm training. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 421–430. ACM, 2014.

[33] Zeyi Wen, Rui Zhang, Kotagiri Ramamohanarao, Jianzhong Qi, and Kerry Taylor. Mascot: Fast and highly scalable svm cross-validation using GPUs and SSDs. In *ICDM*. IEEE, 2014.

[34] Balgeun Yoo Youjip Won, Seokhei Cho Sooyong Kang, Jongmoo Choi, and Sungroh Yoon. SSD characterization: From energy consumptions perspective. *Proceedings of HotStorage*, 2011.

[35] Liang Yang, Fen Zhou, and Yanwu Xia. An improved caching strategy for training SVMs. *International Conference on Intelligent Systems and Knowledge Engineering*, pages 1397–1401, 2007.

[36] Qing Yang and Jin Ren. I-cash: Intelligently coupled array of ssd and hdd. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 278–289. IEEE, 2011.

[37] HX Zhao and Frédéric Magoules. Parallel support vector machines on multi-core and multiprocessor systems. In *International Conference on Artificial Intelligence and Applications*. IASTED, 2011.