Cent. Eur. J. Comp. Sci. • 1-22
Author version

Central European Journal of Computer Science

GPUSVM: A Comprehensive CUDA Based Support Vector Machine Package

Research Article

Qi Li*, **Raied Salman, Erik Test, Robert Strack, Vojislav Kecman**

Virginia Commonwealth University
Richmond, VA 23284, US

Abstract: GPUSVM (Graphic Processing Unit Support Vector Machine) is a Computing Unified Device Architecture (CUDA) based Support Vector Machine (SVM) package. It is designed to offer an end-user a fully functional and user friendly SVM tool which utilizes the power of GPUs. The core package includes an efficient cross validation tool, a fast training tool and a predicting tool. In this article, we first introduce the background theory of how we build our parallel SVM solver using CUDA programming model. Then we compare our GPUSVM package with the popular state of the art Libsvm package on several well known datasets. The preliminary results have shown one to two orders of magnitude speed improvement in both training and predicting phases compared to Libsvm using our Tesla server.

Keywords: Parallel Support Vector Machine • CUDA • GPU• Cross Validation

1. Introduction

The recent developments of Graphic Processing Unit (GPU) have shown superb computational performance on floating point operations compared to the current mainstream multi-core Central Processing Unit (CPU). Certain high performance GPUs are now designed to solve general purpose computing problems instead of graphic rendering, which was considered as the sole purpose of using GPUs previously. More and more research work have shown promising performance results gained by using GPUs on a wide area of topics. Two recent applications of using GPU for fast distance computations [1] and SVM training [2] on large datasets have shown great speed improvements. Although not all algorithms can be parallelized, many applications can benefit from the massive parallel processing capability of GPU by doing some very simple tweaks. For example, using GPU Basic Linear Algebra Subroutines (CUBLAS) [3] instead of conventional Intel Math Kernel Library (MKL) [4] can speed up the application three to five times. Those algorithms which are benefited the most from GPUs are often called data parallel algorithms [5]. They usually show very good data level parallelism compared to other algorithms.

* *E-mail: liq@vcu.edu*

Task level parallelism is not the primary advantage of GPU compared to multi-core CPU, however possibilities of using both task and data level parallelism should be explored and analyzed in order to achieve the maximum performance of GPU.

Although there are plenty of researches done using GPU to improve speed performance of complex algorithms, many applications are still theory oriented and lack of practical usage. For example, Catanzaro et al. developed a SVM using CUDA in [6]. However, there is no cross validation support. Cross validation is considered as the most important procedure for finding the best parameter in order to build an accurate model using SVM. This paper not only introduces the parallel SVM algorithm designed for GPU programming, but it also implements it using CUDA framework and makes it practical for processing real world datasets. The CUDA implementation of parallel SVM developed in this paper has achieved great performance using Fermi series of GPUs, which are the second generation hardware platform for CUDA. The software utilizes the parallelism in both data level and task level to maximize the performance of GPU or GPUs. It also leverages the computation load between CPU and GPU. This helps improving the efficiency of the GPUSVM algorithm. The current implementation of the GPUSVM outperforms the state of the art LIBSVM tool in speed performance for both training phase and predicting phase. It also has as good accuracy performance as LIBSVM. Besides, the software is compatible with previous generation of GPUs and it is practical in solving real world problems. It supports multi-GPU system to enable even further speed improvement on cross-validation training, which is a slow procedure in classic sequential machines.

This paper is organized as follows. Section 2 reviews the history and current development of SVM using sequential and parallel methods. It presents the basic principles of SVM and introduces the GPU hardware and its software platform CUDA for general purpose computing. Section 3 explains the Parallel Sequential Minimal Optimization (PSMO) algorithm implemented on GPU using CUDA. This is the core SVM solver. It also explains the cache design in GPUSVM which helps on accelerating the SVM training. The last part of this section introduces the GUI of GPUSVM and its three-layer structure. Performance results of GPUSVM package is shown in Section 4. Section 5 summarizes the conclusions and points out the potential extensions for the project.

2. Background and Related Work

Support Vector Machine [7] is a learning algorithm which has become popular due to its high accuracy performance in solving both regression and classification tasks. Nevertheless, the training phase of an SVM is a computationally expensive task because the core part of the training stage is solving a Quadratic Programming (QP) problem [8]. There are countless efforts and research which have been done on how to reduce the training time of SVM. After Vapnik invented SVM, he described a method known as “chunking” to break down the large QP problem into a series of smaller QP problems. This method significantly reduced the size of the matrix but it still could not solve large problem due to the computer memory limitations at that time. Osuna et al. presented a decomposition

approach using iterative methods in [9]. Joachims introduced practical techniques such as shrinking and kernel caching in [10], which are common implementation in many modern SVM software today. He also published his own SVM software called SVMLight [10] that uses these techniques. Platt invented Sequential Minimal Optimization (SMO) [8] to solve the standard QP problem by iteratively solving a QP problem with only two unknowns using analytic methods. This method requires a small amount of computer memory. Therefore, it addresses the memory limitation issue brought by large training datasets. Later on, Keerthi et al. developed an improved SMO in [11], which resolved the slow convergence issue in Platt's method. More recently Fan et al. introduced a series of working set selection [12], which further improved the speed of convergence. The method has been implemented in the state of the art Libsvm software [13]. Huang et al. developed the ISDA algorithm [14] to solve the SVM QP problem without the bias. The above major contributions summarize the work of how to implement a fast classic SVM in sequential programming.

Some earlier works using parallel techniques in SVM can be found in [15], [16], [17] and [18]. Cao et al. presented a very practical Parallel SMO [19] implemented with Message Passing Interface on a cluster system. The performance gain of training SVM using clusters shows the beauty of parallel processing. This method is also the foundation of our GPUSVM package. Graf et al. introduced the Cascade SVM [20] which decomposes the training dataset to multiple chunks and trains them separately. Then the support vectors from different individual classifiers are combined and fed back to the system again. They proved that the global optimal solution can be achieved by using this method. This parallelism is in the task level compared to the data level parallelism in Cao et al. 's work [19]. Cascade SVM offers a way to handle ultra-large datasets training. Catanzaro et al. proposed a method to train a binary SVM classifier using GPU in [6]. They use a map-reduce technique. Significant speed improvement is reported compared to the Libsvm software. The latest GPU version of SVM is from Herrero-Lopez et al. [21]. They enable the possibility of training a multi-class classification problem on a GPU.

Although several scientific work and research have shown the possibilities of using GPU to accelerate the SVM applications, there is not one complete package which integrates enough functionalities and provides an user friendly interface. Our aim is to build a comprehensive SVM package which can be easily used and integrated into the real world SVM applications. The current build of GPUSVM package offers an efficient cross validation tool for finding the best training parameters, a training tool for generating the SVM model from the training datset with specific input parameters and kernel type and a predicting tool for doing predictions on query datasets. The cross validation tool supports multi-GPU system.

2.1. Support Vector Machine

This section briefly reviews the basic principles of designing an SVM for solving two-class classification problems. More details of SVM formulation and examples can be found in [22].

2.1.1. Hard-Margin SVM

Hard-margin SVM forms a hyperplane that separates a set of data points with label "+1" from a set of data points with label "-1" using the maximum margin. The graphic presentation is shown in Figure 1. The output of a training data point $\vec{x}_i$ can be computed by

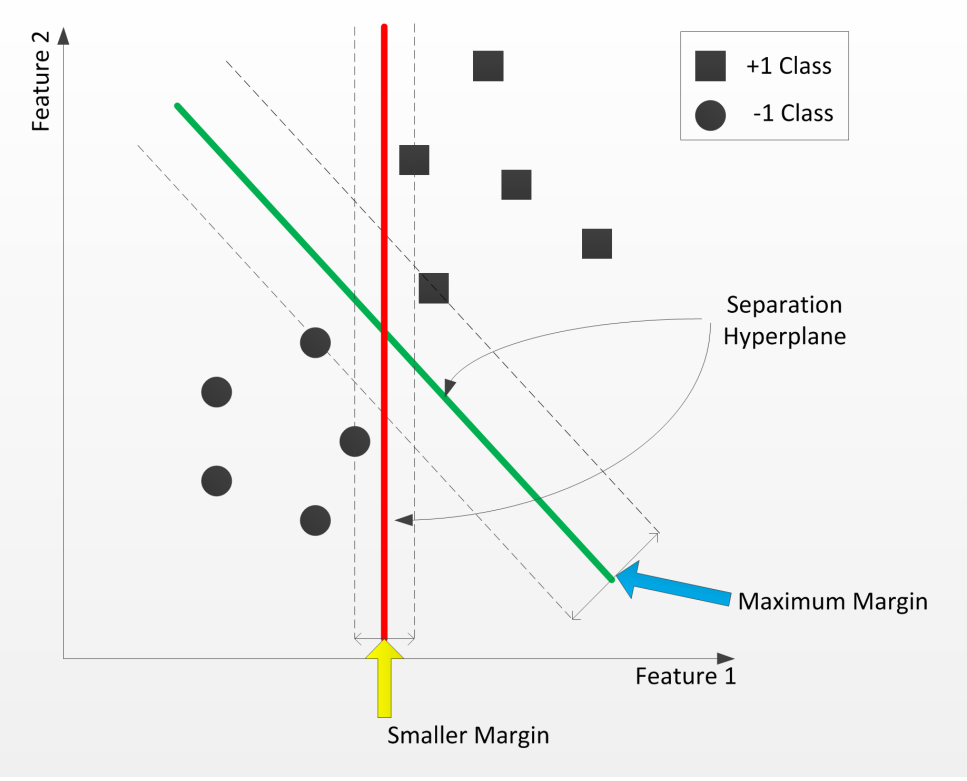

Figure 1. **The graphic representation of a linear SVM.**

$$o(\vec{x}_i) = \vec{w}^T \vec{x}_i + b, \tag{1}$$

where $\vec{w}$ is the normal vector to the separation hyperplane and $\vec{x}_i$ is a training data point. The term b is an added bias. In the case of linearly separable dataset, no training data point satisfies $o(\vec{x}_i) = 0$. Thus, to control separability, the following inequalities

$$\vec{w}^T \vec{x}_i + b = \begin{cases} \geq 1 & \text{for} \quad y_i = 1; \\ \leq -1 & \text{for} \quad y_i = -1 \end{cases}$$

are used. They are equivalent to

$$y_i(\vec{w}^T \vec{x}_i + b) \geq 1. \tag{2}$$

All hyperplanes which satisfy

$$o(\vec{x}_i) = \vec{w}^T \vec{x}_i + b = c \quad \forall c: \ -1 < c < 1 \tag{3}$$

can separate the dataset correctly. They are called feasible solutions. When $c = 0$, the hyperplane is in the middle of two hyperplanes with $c = -1$ and $c = 1$. The margin is defined as the distance, multiplied by 2, from a hyperplane to its nearest positive and negative points. The margin m can be calculated by

$$m = \frac{2}{||\vec{w}||}. \tag{4}$$

A hyperplane is considered as optimal if it is a feasible solution and it has the maximum margin. Equation 4 shows that the maximum margin can be found when the Euclidean norm of $\vec{w}$, which satisfies Equation 2, is minimized. The following optimization problem

$$\begin{aligned} &\min_{\vec{w},b} \frac{1}{2}||\vec{w}||, \\ &\text{s.t.} \quad \forall i: \ y_i(\vec{w}^T \vec{x}_i + b) \geq 1 \end{aligned} \tag{5}$$

can be formulated to find the optimal hyperplane. $\vec{x}_i$ is the ith training data point, and y_i is the corresponding label of $\vec{x}_i$. The value of y_i is either $+1$ or -1. This optimization problem can be transformed to a dual form

$$\begin{aligned} &\min_{\vec{\alpha}} L_d(\vec{\alpha}) = \min_{\vec{\alpha}} (\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j (\vec{x}_i^T \vec{x}_j) \alpha_i \alpha_j - \sum_{i=1}^{n} \alpha_i), \\ &\text{s.t.} \quad \forall i: \ \alpha_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{n} y_i \alpha_i = 0 \end{aligned} \tag{6}$$

using Lagrangian multipliers. It is a quadratic programming problem where the objective function L_d solely depends on Lagrangian multipliers $\vec{\alpha}$. The value of n is the total number of training data points. There is an one-to-one relationship between each Lagrangian multiplier and each training data point. For those training data points whose α is bigger than zero, they are referred as support vectors. S describes the set which contains all support vectors. Once the QP problem is solved and the $\vec{\alpha}$ is found, the normal vector $\vec{w}$ and the bias b can be computed by

$$\vec{w} = \sum_{i:\vec{x}_i \in S} y_i \alpha_i \vec{x}_i \tag{7}$$

and

$$b = \frac{1}{|S|} \sum_{i:\vec{x}_i \in S} (y_i - \vec{w}^T \vec{x}_i). \tag{8}$$

The classification function uses the following *sgn* function

$$d(\vec{x}) = sgn(o(\vec{x})) = sgn(\vec{w}^T \vec{x} + b), \tag{9}$$

$$d(\vec{x}) = \begin{cases} +1 & \Rightarrow \vec{x} \in \text{Positive Class}; \\ -1 & \Rightarrow \vec{x} \in \text{Negative Class}, \end{cases}$$

which assigns the correct label to an input query.

2.1.2. *L1* Soft-Margin Linear SVM

The SVM was originally proposed as an algorithm searching for the maximal margin in classifying two linearly separable classes. It is also referred to as hard-margin SVM and it cannot classify datasets with overlapping. In order to utilize the merits of SVM on overlapped datasets, Cortes and Vapnik [23] proposed an extension by adding a set of slack variables ζ_i, which allows certain degree of misclassifications. This formulation is called soft-margin SVM and it is shown in

$$\min_{\vec{w},b,\vec{\zeta}} \frac{1}{2}||\vec{w}|| + C\sum_{i=1}^{n}\zeta_i, \tag{10}$$
$$\text{s.t.} \quad \forall i : y_i(\vec{w}^T\vec{x}_i + b) \geq 1 - \zeta_i$$

where ζ_i permits a few misclassifications and the penalty value C determines the trade off between the maximization of the margin and the minimization of the errors. This optimization problem has a dual form as shown in

$$\min_{\vec{\alpha}} L_d(\vec{\alpha}) = \min_{\vec{\alpha}} \left(\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j(\vec{x}_i^T\vec{x}_j)\alpha_i\alpha_j - \sum_{i=1}^{n}\alpha_i\right), \tag{11}$$
$$\text{s.t.} \quad \forall i : \ 0 \leq \alpha_i \leq C \text{ and } \sum_{i=1}^{n} y_i\alpha_i = 0.$$

The Lagrangian multiplier α_i has a box constraint and an equality constraint shown in Equation 11. And each data point $\vec{x}_i$ is associated with one α_i. In order to guarantee the existence of an optimal point for a positive definite QP problem, the Karush-Kuhn-Tucker (KKT) conditions should be satisfied. The KKT conditions for the QP problem described in Equation 11 are met when

$$\begin{aligned} \forall i : \ \alpha_i(y_i(\vec{w}^T\vec{x}_i + b) - 1 + \zeta_i) &= 0, \\ (C - \alpha_i)\zeta_i &= 0, \\ \alpha_i \geq 0 \text{ and } \zeta_i &\geq 0. \end{aligned} \tag{12}$$

There are three different cases for α_i:

1. $\alpha_i = 0$. Then $\zeta_i = 0$. Thus $\vec{x}_i$ is correctly classified.
2. $0 < \alpha_i < C$. Then $y_i(\vec{w}^T\vec{x}_i + b) - 1 + \zeta_i = 0$ and $\zeta_i = 0$. Therefore, $y_i(\vec{w}^T\vec{x}_i + b) = 1$ and $\vec{x}_i$ is called unbounded support vector and correctly classified. Denote set U containing all unbounded support vectors.
3. $\alpha_i = C$. Then $y_i(\vec{w}^T\vec{x}_i + b) - 1 + \zeta_i = 0$ and $\zeta_i \geq 0$. Thus, $\vec{x}_i$ is called bounded support vector. If $0 \leq \zeta_i < 1$, $\vec{x}_i$ is correctly classified. If $\zeta_i \geq 1$, $\vec{x}_i$ is misclassified. Denote set B containing all bounded support vectors. Thus $U \cup B = S$, where set S contains all support vectors.

After the dual variables are calculated, the bias b is averaged over all unbounded support vectors by

$$b = \frac{1}{|U|} \sum_{i:\vec{x}_i \in U} (y_i - \sum_{j:\vec{x}_j \in S} \alpha_j y_j (\vec{x}_j^T \vec{x}_i)). \tag{13}$$

The query point can be classified using

$$d(\vec{x}) = \sum_{i:\vec{x}_i \in S} \alpha_i y_i (\vec{x}_i^T \vec{x}) + b \tag{14}$$

and

$$o(\vec{x}) = sgn(d(\vec{x})), \tag{15}$$

where $sgn()$ is a sign function.

$$o(\vec{x}) = \begin{cases} +1 & \Rightarrow \vec{x} \in \text{Positive Class;} \\ -1 & \Rightarrow \vec{x} \in \text{Negative Class.} \end{cases}$$

2.1.3. $L1$ Soft-Margin Nonlinear SVM

The dual form of an $L1$ soft-margin nonlinear SVM is very similar to Equation 11.

$$\min_{\vec{\alpha}} L_d(\vec{\alpha}) = \min_{\vec{\alpha}} \left(\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j K(\vec{x}_i, \vec{x}_j) \alpha_i \alpha_j - \sum_{i=1}^{n} \alpha_i \right), \tag{16}$$

$$\text{s.t.} \quad \forall i: \ 0 \leq \alpha_i \leq C \text{ and } \sum_{i=1}^{n} y_i \alpha_i = 0.$$

The kernel function $K(\vec{x}_i, \vec{x}_j)$ introduced in Equation 16 maps the original input space to a higher dimensional dot-product feature space. Several popular kernel functions are listed in Table 1. The bias term b can be computed

Table 1. **List of popular kernel functions.**

Type of Classifier	Kernel Function
Linear Kernel	$K(\vec{x}_i, \vec{x}_j) = \vec{x}_i^T \vec{x}_j$
Polynomial Kernel	$K(\vec{x}_i, \vec{x}_j) = (\vec{x}_i^T \vec{x}_j + 1)^d$
Radial Basis Kernel	$K(\vec{x}_i, \vec{x}_j) = e^{-\gamma \lvert\lvert \vec{x}_i - \vec{x}_j \rvert\rvert^2}$, $\gamma = \frac{1}{2\sigma^2}$

by

$$b = \frac{1}{|U|} \sum_{i:\vec{x}_i \in U} (y_i - \sum_{j:\vec{x}_j \in S} \alpha_j y_j K(\vec{x}_j, \vec{x}_i)), \tag{17}$$

and the decision function is

$$d(\vec{x}) = \sum_{i:\vec{x}_i \in S} \alpha_i y_i K(\vec{x}_i, \vec{x}) + b. \tag{18}$$

Computing the outputs of kernel functions are referred to as kernel computations, which are expensive in terms of time cost due to its nature of intensive computation. These computations drastically slow down the training procedure of SVM. Kernel computations can be accelerated by using GPUs, but it is more important to minimize the total amount of kernel computations. This can be done through caching which is explained in Section 3.2.

2.2. General Purpose Computing Using Graphic Processing Unit

This section briefly introduces the GPU hardware and the CUDA development platform.

2.2.1. Graphic Processing Unit

GPUs are micro processors commonly seen on video cards. The main function of GPU is offloading and accelerating the graphic rendering jobs from the CPU. Rendering is a process of generating an image from a model by a set of computer programs and it usually involves floating point intensive computations based on various mathematical equations. Thus, before 2006, most of these GPUs were designed in a way that computing resources were partitioned into vertexes and pixel shaders. Even though the hardware of GPUs have matured for intensive floating point computations, there is no other way but using OpenGL or DirectX to access the features in GPUs. Smart programmers disguised their general computations to graphic problems in order to utilize the hardware capability of GPU. They were the first who started to use GPUs to solve general purpose computing problems. In order to overcome this inflexibility, NVIDIA introduced the GeForce 8800 GTX in 2006, which maps the separated programmable graphics stages to an array of unified processors. Figure 2 shows the shader pipeline of GeForce 8800 GTX GPU. It is organized into an array of highly threaded streaming processors (SMs). In Figure 2, two SMs form a building block; however, the number of SMs in a building block can vary between different generations of CUDA GPUs. Each SM in Figure 2 has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 6GB (e.g. Tesla C2070) GDDR DRAM, referred to as global memory. These global memory are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images and texture information for rendering, but for computing purpose they function as high bandwidth off-chip memory. All later GPU products from NVIDIA follow this design philosophy thus they are capable of general purpose computing and referred to as CUDA capable devices.

The latest Tesla GPU has the shader processors (cores) fully programmable with large instruction memory, instruction cache and instruction sequencing control logic. In order to reduce the total hardware cost, several shader processors will share the same instruction cache and instruction sequencing control logic. The Tesla architecture introduced a more generic parallel programming model with a hierarchy of parallel threads, barrier synchronization and atomic operations to dispatch and manage highly parallel computing work. Combined with C/C++ compiler, libraries, runtime software and other useful components, CUDA Software Development Kit is offered to developers who do not posses the programming knowledge of graphic applications. With a minimal learning curve of some extended C/C++ syntax and some basic parallel computing techniques, developers can start migrating existing projects using CUDA with NVIDIA GPUs.

2.2.2. Computing Unified Device Architecture

CUDA is a software platform developed by NVIDIA to support their general purpose computing GPUs for easy programming and porting existing applications to GPUs. It primarily uses C/C++ syntax and a few new keywords as an extension, which offers a very low learning curve for an application designer. The latest CUDA version

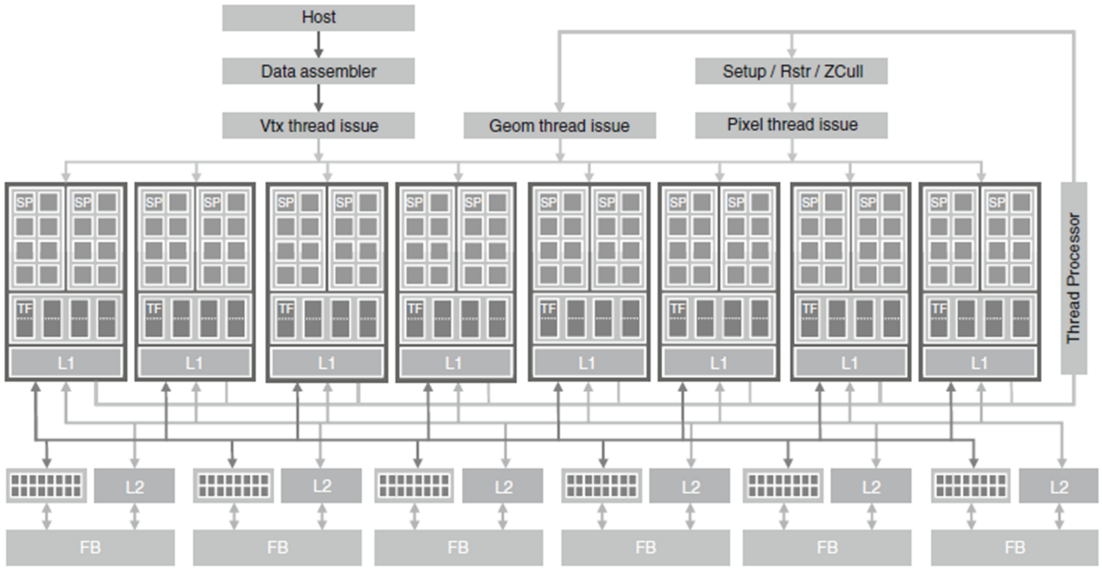

Figure 2. **The architecture of a CUDA-capable GPU.**

has been supported by various third parties. Many toolboxes and plug-ins can be found to help increase the productivity. CUDA memory model and thread organization is introduced in this part.

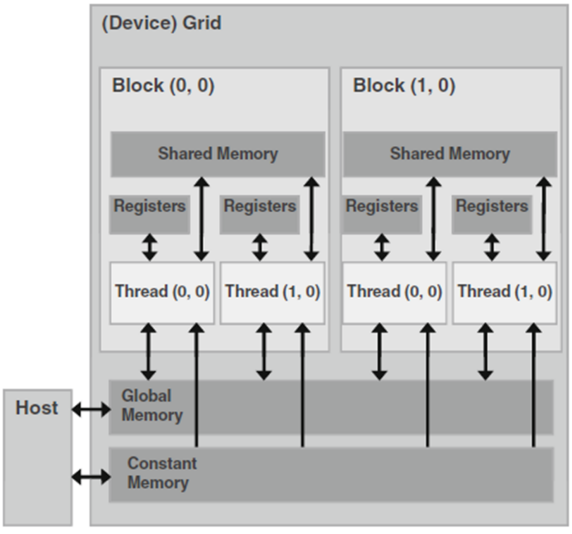

Figure 3. **CUDA device memory model.**

Figure 3 shows the memory model of the CUDA device. The device codes can read/write per-thread registers;

read/write per-block shared memory; read/write per-grid global memory; read only per-grid constant memory. The host codes can transfer data to/from per-grid global and constant memory. Constant memory offers faster memory access to CUDA threads compared to global memory. The threads are organized in a hierarchical structure. The top level is a grid which contains blocks of threads. Each grid can contain at most 65535 blocks in either x- or y-dimension or both in total. Each block can contain at most 1024 (Fermi series) or 512 threads in either x- or y- dimension, or maximally 64 in z-dimension. The total number of threads in all three dimensions must be less than or equal to 1024 or 512 depending on the hardware specification. The organization of threads is shown in Figure 4. The host (CPU) launches the kernel function on the device (GPU) in the form of grid structure. Once the computation is done, the device becomes available again then the host can launch another kernel function. If multiple devices are available at the same time, every kernel function can be managed through one CPU thread. It is fairly easy to launch a grid structure containing thousands of threads. The optimum number of thread and block configuration varies among different applications. To achieve better performance, there should be at least thousands or tens of thousands of threads with in one grid. It would not make much sense to use too few threads to extract maximal performance from hardware. However, too many threads whose number exceeds the number of data would also increase the thread overhead and bring down the efficiency. The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Thus a multiple of 32 could be a good candidate value for the optimal number of threads per block. Threads within the same block have limited shared memory and they are able to communicate with each other by using these shared memory. All threads have their own registers and access to the global memory as well as the constant memory. The size of the global memory can be as large as up to 6GB (depending on the GPU hardware). Similar to Message Passing Interface (MPI), there is no shared memory between host and device thus the data must be transferred from the host memory to device memory in the first place. The result must also be transferred back for future processing or storage.

3. GPUSVM

This section introduces the proposed GPUSVM algorithm, its unique cache design and its user friendly Graphic User Interface.

3.1. CUDA Based Parallel SMO

Cao et al. [19] developed the PSMO algorithm to accelerate binary SVM training by partitioning the input dataset and splitting it across multiple computing nodes. Herrero-Lopez et al. [21] further improved it and developed the P2SMO algorithm for multi-class SVM. GPUSVM borrows the merits from these two algorithms including parallel scan, parallel updating error vectors.

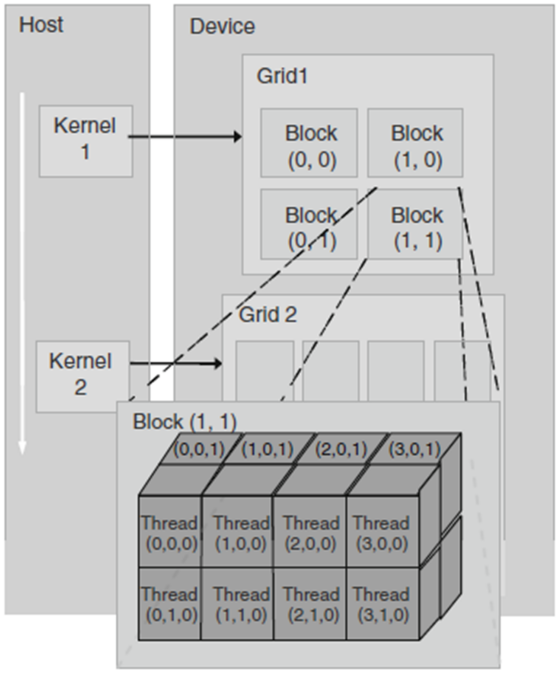

Figure 4. **CUDA thread organization.**

Define the following index sets at a given α and y:

$$\begin{aligned} I_0 &= \{i:\ 0 < \alpha_i < C\}, \\ I_1 &= \{i:\ y_i = 1,\ \alpha_i = 0\}, \\ I_2 &= \{i:\ y_i = -1,\ \alpha_i = C\}, \\ I_3 &= \{i:\ y_i = 1,\ \alpha_i = C\}, \\ I_4 &= \{i:\ y_i = -1,\ \alpha_i = 0\}, \\ I_{up} &= I_0 \cup I_1 \cup I_2, \\ I_{lo} &= I_0 \cup I_3 \cup I_4. \end{aligned}$$

The KKT conditions can be rewritten as

$$\begin{aligned} \forall i \in I_{up}:\ b \leq e_i, \\ \forall i \in I_{lo}:\ b \geq e_i, \end{aligned}$$

where

$$e_i = \sum_{j:\vec{x}_j \in S} \alpha_j y_j K(\vec{x}_j, \vec{x}_i) - y_i, \tag{19}$$

thus the KKT conditions will hold if and only if

$$\begin{aligned} b_{up} &= \min\{e_i : i \in I_{up}\}, \\ b_{lo} &= \max\{e_i : i \in I_{lo}\}, \\ b_{lo} &\leq b_{up}. \end{aligned} \tag{20}$$

An index pair (i, j) violates the KKT condition if

$$i \in I_{lo},\ j \in I_{up} \quad \text{and} \quad e_i > e_j, \tag{21}$$

thus the objective is eliminating all (i, j) pairs which violate the KKT condition. However, it is usually not possible to achieve the exact optimality conditions. Thus, it is necessary to define the approximate optimality conditions. This is shown in the following equation:

$$b_{lo} \leq b_{up} + 2\tau, \tag{22}$$

where τ is a positive tolerance parameter. It is usually set to 0.001 for general applications recommended in [8]. The bias value can be computed by

$$b = \frac{b_{lo} + b_{up}}{2}. \tag{23}$$

In each iteration of the training phase, the α values are updated by

$$s = y_{up} y_{lo}, \tag{24}$$

$$\begin{aligned} \eta &= K(\vec{x}_{lo}, \vec{x}_{lo}) + K(\vec{x}_{up}, \vec{x}_{up}) \\ &\quad -2K(\vec{x}_{lo}, \vec{x}_{up}), \end{aligned} \tag{25}$$

$$\alpha_{up}^{new} = \alpha_{up} + \frac{y_{up}(e_{lo} - e_{up})}{\eta}, \tag{26}$$

$$\alpha_{lo}^{new} = \alpha_{lo} + s(\alpha_{up} - \alpha_{up}^{new}). \tag{27}$$

After new α values are computed, the error vector for all training data must be updated by

$$\begin{aligned} e_i^{new} &= e_i + (\alpha_{lo}^{new} - \alpha_{lo}) y_{lo} K(\vec{x}_{lo}, \vec{x}_i) \\ &\quad +(\alpha_{up}^{new} - \alpha_{up}) y_{up} K(\vec{x}_{up}, \vec{x}_i). \end{aligned} \tag{28}$$

The complete algorithm is shown below:

Algorithm 1 Parallel Sequential Minimal Optimization on CUDA.

$\alpha_i = 0, f_i^p = -y_i$ (device)
compute $b_{up}^p, b_{lo}^p, I_{up}^p, I_{lo}^p$ (device)
compute $b_{up}, b_{lo}, I_{up}, I_{lo}$ (host)
while $b_{lo} > b_{up} + 2\tau$ **do**
obtain $k_{I_{lo},I_{lo}}, k_{I_{up},I_{up}}, k_{I_{up},I_{lo}}$ (device)
update $\alpha_{I_{up}}, \alpha_{I_{lo}}$ (device)
compute $b_{up}^p, b_{lo}^p, I_{up}^p, I_{lo}^p$ (device)
compute $b_{up}, b_{lo}, I_{up}, I_{lo}$ (host)
end while
return α_i

3.2. Cache Design

In Section 2.1, we have mentioned that the kernel computations are the most expensive operations in SVM training phase. Thus, it is very important and critical to minimize the duplicated kernel computations. However, it is not possible to store the kernel values for all support vectors due to the memory limitation. This issue can be resolved by designing an efficient cache. The cache is used for storing the kernel values, which might be reused in future iteration.

There are two layers in the cache shown in Figure 5. They are the abstract layer and the physical layer. The abstract layer is used as a programming interface to maintain the Least Recently Used (LRU) list which is on the CPU side. The physical layer is the GPU device memory layout. A 2D array referred to as cache array on the GPU device is used as the storage of kernel matrix. Each row stores a kernel vector containing kernel values from one support vector to all data points. Thus the number of columns is fixed to the number of all data points and the number of rows is the size of the cache, which depends upon the available memory on the GPU device. The abstract layer contains a vector of nodes and a LRU list. Each nodes includes information about *status*, *location* and *lrulistpos*.

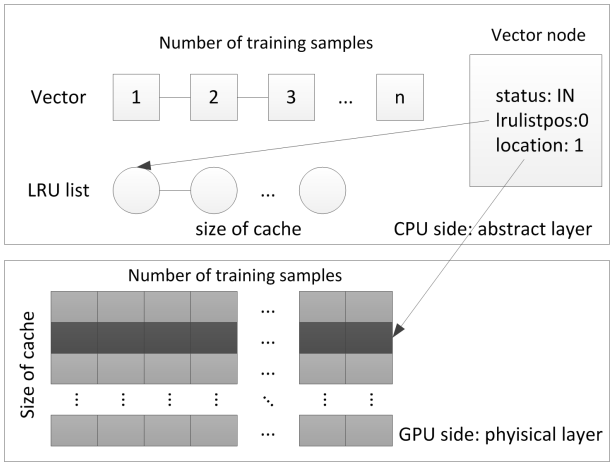

Figure 5. **The design structure for cache.**

Each node represents a data point, *status* indicates whether the node is in the LRU list; *location* stores the row number of cache array on the GPU device; *lrulistpos* stores the position of the node in the LRU list. The LRU list has the same size as the cache. There are two different scenarios of doing operations on the cache:

1. The new support vector is in the cache. If *lrulistpos* points at the head of LRU list, do nothing and return its *location*. If not, remove it from the LRU list and append it back to the head of LRU list. Update the its *lrulistpos* and return its *location*. The GPU fetches the kernel vector from the *location* directly.

2. The new support vector is not in the cache.

   (a) If the cache is not full, append the new support vector at the head of the LRU list and update its *lrulistpos*. Increase the size of the LRU list by 1 and set the new support vector's *location* to the value of LRU list's size after the increment. Set its *status* to *IN*, return its *location* and ask for kernel computation. The GPU computes the kernel vector and stores it in the *location* on the GPU device memory. This operation overwrites a blank space.

   (b) If the cache is full, retrieve the support vector from the end of the LRU list and assign its *location* value to the new support vector's *location*. Set the expired support vector's *status* to *OUT*. Remove the expired support vector from the LRU list and append the new support vector at the head of the LRU list. Update the *lrulistpos* of the new support vector and set its *status* to *IN*. Return its *location* and ask for the kernel computations. The GPU computes the kernel vector and stores it in the *location* on the GPU device memory. This operation overwrites the memory space used by the expired support vector.

By carrying out the above operations, the most recently used support vector will always appear at the head of the LRU list. Whenever the cache is full, the erased point is always the least recently used support vector. This cache design minimizes the unnecessary kernel computations within one single binary task as well as multitask cross validation. If the cache size is large enough, kernel vectors of all support vectors appeared during training are only computed once.

3.3. A Glance of GPUSVM

GPUSVM has three layers. The top layer is a Graphic User Interface (GUI) written in Java. The GUI offers the user easy access to the tools and parameter setting. It has three main tabs which are for cross validation, training and predicting purposes. They are shown in Figure 6, Figure 7 and Figure 8. The cross validation interface allows the user to choose the training file and configure various parameters such as kernel type and scaling method. User can enter the specific GPU device id in a list to run the cross validation procedure on multiple GPUs if available. Our tesla system has two Tesla C2070s and six Tesla C2050s, thus the user can use all eight GPUs to accelerate the cross validation at the same time. The results of the cross validation will be returned as a table showing

both number of support vectors and number of misclassifications for all different combinations of the training parameters. It draws a 2D surface for Gaussian/polynomial kernel and a curve for linear kernel.

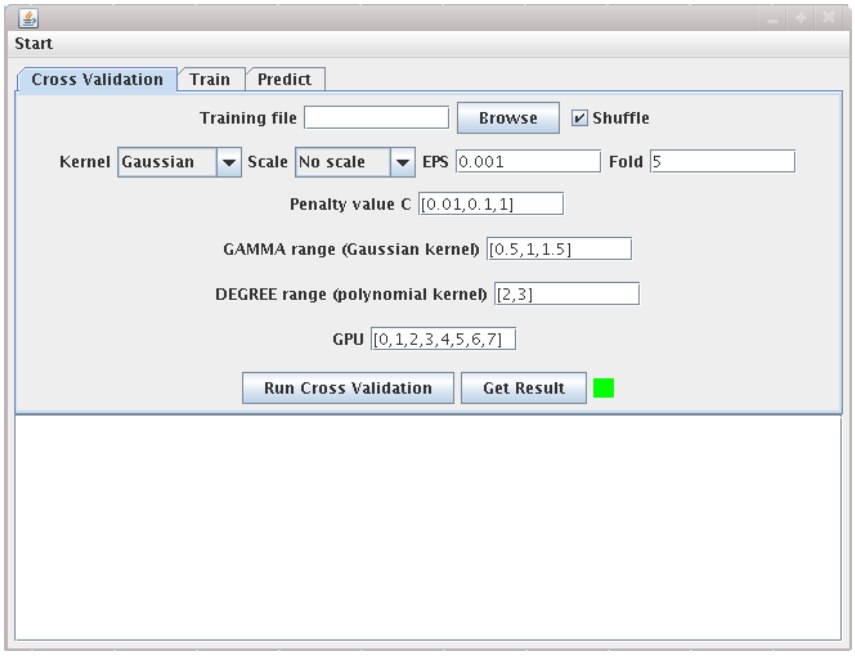

Figure 6. **GPUSVM: cross validation interface.**

The training interface lets the user choose the training file and enter the specific parameters for a particular kernel. The user is also able to choose one GPU device to run the training task. After the training procedure is done, a model file will be generated which contains all the necessary information for making predictions on the query datasets. The training file could also be scaled before it is used. Three scaling methods are offered which are $[0, 1]$, $[-1, 1]$ and zero mean. The scaling information are stored in a separated file which is used to scale the query datasets to align the input feature space. The predicting interface requires the user to choose the query datasets and the model file as well as the scaling file if scaling is done before training phase. The user can specify the GPU device id for predicting phase. The result shows the number of misclassifications and the actual predicting accuracy.

The middle layer of GPUSVM package are several script files written in Python. Their major tasks are doing file manipulations. The shuffling tool rearranges the input data sequences randomly which is critical in cross validation. In each fold of cross validation, if the training part does not have equal distributions from all different classes, it could lead to some bad results. The scaling tool scales the input dataset according to a particular scaling method or an existing scaling file. The cross validation tool slices the input data file to multiple folds and prepare the training dataset and verification dataset for the SVM solver and predictor. The SVM solver tool and predictor tool calls the actual routine in the bottom layer for training and predicting purposes.

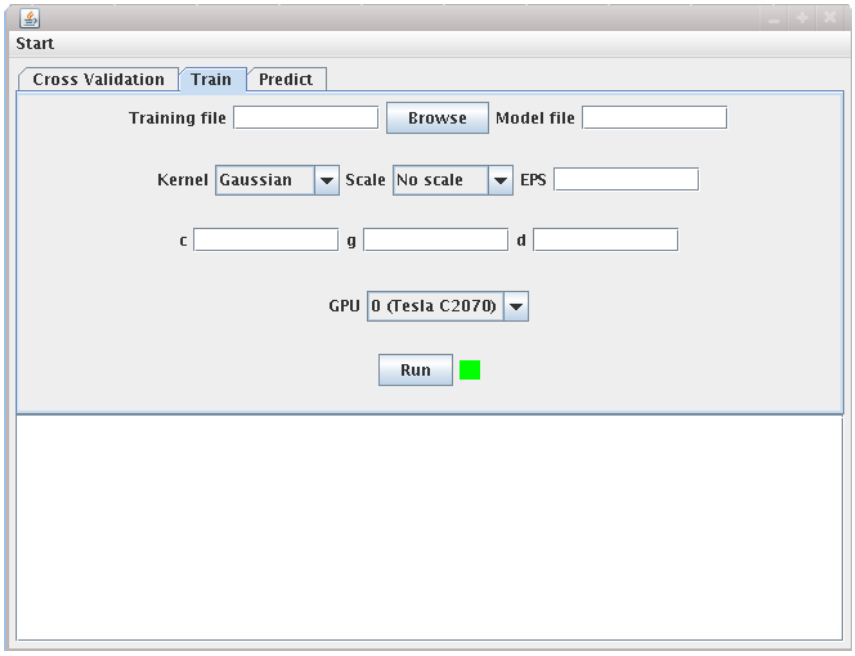

Figure 7. **GPUSVM: training interface.**

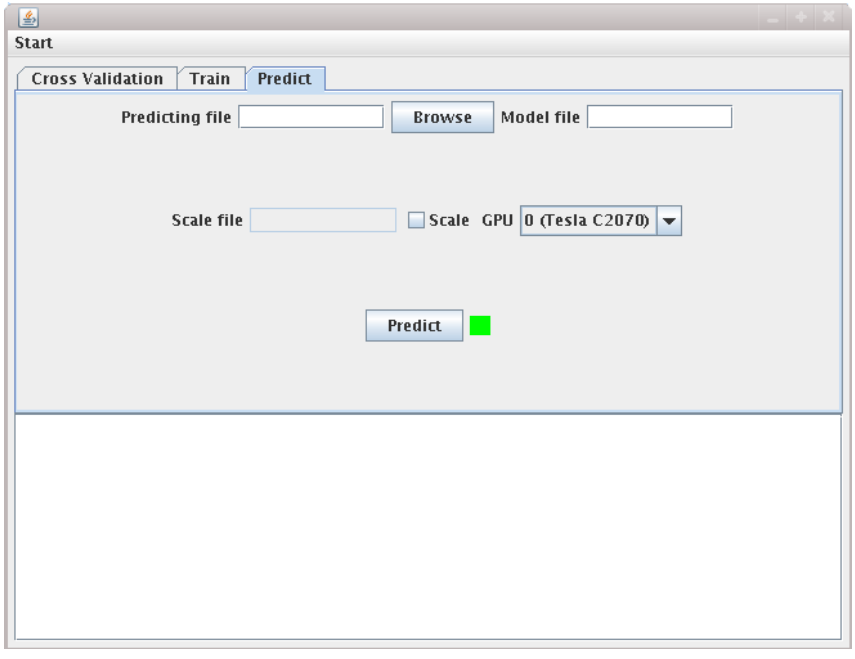

Figure 8. **GPUSVM: predicting interface.**

The bottom layer of GPUSVM package contains only two executable files written in C on top of CUDA. One is an SVM solver which solves the QP problem in SVM and generate the model file using GPU. The other one is a

predictor which takes in the query file and a model file generated by the SVM solver to make predictions.

4. Performance Results

This section presents the performance results obtained by GPUSVM. The following measurements are carried out by our latest tesla system equipped with two Intel Xeon X5680 3.3GHz six-core CPUs, 96GB ECC DDR3 1333MHz main memory, six Tesla C2050s with 3GB GDDR5 memory each and two Tesla C2070s with 6GB GDDR5 memory each. The storage device is a 128GB SSD with Fedora Core Linux 14 x64 installed. The CUDA driver and runtime version are both 3.2.

4.1. Datasets

All datasets used in the experiments are downloaded from official Libsvm website with pre-scaled value. The characteristics of the datasets and the training parameters for the Gaussian RBF kernel are listed in Table 2. Several of them are binary class datasets and the others are multi-class datasets. *Glass, iris, wine, sonar, breast-cancer, adult* datasets are from UCI [24] and *heart, letter, shuttle* datasets are from Statlog [25]. *Usps* is a hand written dataset [26] for text recognition. *Web* is web pages text categorization used in [8]. *Mnist* is another hand written text recognition dataset used in [27]. The training parameter C is the penalty value and γ is the shape value of RBF kernel. The best C and γ are found out by using 5-fold cross validation on $C \in \{2^i, i \in [-10, 10]\}$, $\gamma \in \{2^i, i \in [-5, 5]\}$.

Table 2. The experimental datasets and their training parameters for the Gaussian RBF kernel.

Dataset	# of training sample	# of testing sample	# of feature	# of class	C	γ
glass	214	N/A	9	6	512	2
iris	150	N/A	4	3	16	0.5
wine	178	N/A	13	3	1	0.25
heart	270	N/A	13	2	0.5	0.0625
sonar	208	N/A	60	2	4	0.125
breast-cancer	683	N/A	10	2	0.25	0.125
adult	32561	16281	123	2	1	0.0625
usps	7291	2007	256	10	128	0.015625
letter	15000	5000	16	26	16	8
shuttle	43500	14500	9	7	1	1
web	49749	14951	300	2	64	8
mnist	60000	10000	780	10	16	0.00390625

4.2. Accuracy Performance Comparison Test

Table 3 shows the accuracy performance between GPUSVM and Libsvm. In this test, both methods have very close performance compared to each other on all datasets. The final accuracies are slightly different due to the different number of support vectors used. GPUSVM uses a little more support vectors compared to Libsvm. This is because Libsvm uses a working set method which solves a QP problem with its size larger than two. GPUSVM uses analytic method to iteratively solve QP problems with its size fixed at two. Also, Libsvm implements OVO and GPUSVM uses OVA approach for multi-class datasets.

Table 3. **The accuracy comparison between GPUSVM and Libsvm on the experimental datasets.**

Dataset	SVM	Training Accuracy	Predicting Accuracy	# of support vector
glass	GPUSVM	98.1308%	N/A	144
	Libsvm	**98.5981**%		133
iris	GPUSVM	98%	N/A	27
	Libsvm	98%		25
wine	GPUSVM	99.4382%	N/A	75
	Libsvm	99.4382%		68
heart	GPUSVM	85.1852%	N/A	146
	Libsvm	85.1852%		146
sonar	GPUSVM	100%	N/A	150
	Libsvm	100%		150
breast-cancer	GPUSVM	97.2182%	N/A	91
	Libsvm	97.2182%		91
adult	GPUSVM	85.7928%	**85.0193**%	11587
	Libsvm	85.7928%	85.0132%	11647
usps	GPUSVM	99.9863%	**95.715**%	1923
	Libsvm	99.9863%	95.6153%	1785
letter	GPUSVM	99.8467%	**97.38**%	11936
	Libsvm	**100**%	96.82%	10726
shuttle	GPUSVM	99.4736%	99.5655%	3667
	Libsvm	**99.5149**%	**99.6069**%	3109
web	GPUSVM	99.4553%	99.4515%	35220
	Libsvm	99.4553%	99.4515%	35231
mnist	GPUSVM	99.4617%	**98.27**%	12919
	Libsvm	**99.5917**%	98.03%	9738

4.3. Speed Performance Comparison Test

The following tests shown in Table 4 and Table 5 are the speed performance between Libsvm and GPUSVM in both training and predicting phases. Datasets which do not have testing set use training set for predicting purpose. The performance of Libsvm using one core of Xeon processor is set as the base line. It is compared with Libsvm using all 12 cores from two Xeon CPUs with Libsvm's built in OpenMP feature enabled. The total

number of threads is set at 12 to extract the maximum performance of multi-core CPU. GPUSVM using one Tesla C2050/C2070 is also listed as the comparison reference. All GPU devices used for tests have the Error Correction Code (ECC) function disabled. This will free more device memory to applications. It is easy to see that whether using GPU or multi-core CPU does not bring any performance gain for solving small SVM classification problem due to the overhead of using OpenMP and CUDA. On the other hand, GPUSVM shows much better performance on medium to large datasets and it achieves a speedup of 2.27x - 77x. Tesla C2070 is generally faster than Tesla C2050 because of the doubled device memory.

Table 4. **The speed performance comparison between Libsvm and GPUSVM on small datasets.**

Dataset	SVM	Processor	Training Time	Speedup	Predicting Time	Speedup
glass	Libsvm	Xeon 1-core	0.008s	1x	0.004s	1x
		Xeon 12-core	0.010s	0.8x	0.005s	0.8x
	GPUSVM	Tesla C2050	3.759s	0.0021x	0.009s	0.4444x
		Tesla C2070	2.32s	0.0035x	0.008s	0.5x
iris	Libsvm	Xeon 1-core	0.002s	1x	0.002s	1x
		Xeon 12-core	0.003s	0.6667x	0.004s	0.5x
	GPUSVM	Tesla C2050	1.305s	0.0015x	0.006s	0.3333x
		Tesla C2070	1.284s	0.0016x	0.006s	0.3333x
wine	Libsvm	Xeon 1-core	0.003s	1x	0.003s	1x
		Xeon 12-core	0.004s	0.75x	0.005s	0.6x
	GPUSVM	Tesla C2050	1.567s	0.0019x	0.008s	0.375x
		Tesla C2070	1.055s	0.0028x	0.007s	0.4286x
heart	Libsvm	Xeon 1-core	0.006s	1x	0.005s	1x
		Xeon 12-core	0.005s	1.2x	0.006s	0.8333x
	GPUSVM	Tesla C2050	1.03s	0.0058x	0.01s	0.5x
		Tesla C2070	1.048s	0.0057x	0.01s	0.5x
sonar	Libsvm	Xeon 1-core	0.014s	1x	0.011s	1x
		Xeon 12-core	0.011s	1.2727x	0.011s	1x
	GPUSVM	Tesla C2050	1.383s	0.0101x	0.009s	1.2222x
		Tesla C2070	1.645s	0.0085x	0.009s	1.2222x
breast-cancer	Libsvm	Xeon 1-core	0.008s	1x	0.006s	1x
		Xeon 12-core	0.006s	1.3333x	0.012s	0.5x
	GPUSVM	Tesla C2050	1.352s	0.0059x	0.025s	0.24x
		Tesla C2070	1.395s	0.0057x	0.022s	0.2727x

The speed performance of OpenMP enabled Libsvm is quite well too. This is due to the shared memory system setting. All threads resided in the CPU can access the large main memory. However, this performance is strictly limited by the number of CPUs and the cores of each CPU on the motherboard. That means the maximum performance of CPU in this workstation is achieved. Using any number of threads other than 12 will not gain any benefit. On the other hand, the potential of using GPU is huge since there is only one GPU device involved in the current testing.

Table 5. **The speed performance comparison between Libsvm and GPUSVM on medium and large datasets.**

Dataset	SVM	Processor	Training Time	Speedup	Testing Time	Speedup
adult	Libsvm	Xeon 1-core	60.634s	1x	20.273s	1x
		Xeon 12-core	8.998s	6.7386x	2.216s	9.1485x
	GPUSVM	Tesla C2050	8.644s	7.0145x	0.697s	29.0861x
		Tesla C2070	7.636s	7.9405x	0.649s	31.2373x
usps	Libsvm	Xeon 1-core	4.901s	1x	2.113s	1x
		Xeon 12-core	1.331s	3.6822x	0.446s	4.7377x
	GPUSVM	Tesla C2050	3.005s	1.6309x	0.088s	24.0114x
		Tesla C2070	2.158s	2.2711x	0.081s	26.0864x
letter	Libsvm	Xeon 1-core	37.768s	1x	4.666s	1x
		Xeon 12-core	11.902s	3.1712x	1.88s	2.4819x
	GPUSVM	Tesla C2050	11.318s	3.3370x	0.465s	10.0344x
		Tesla C2070	10.554s	3.5785x	0.445s	10.4854x
shuttle	Libsvm	Xeon 1-core	9.379s	1x	2.402s	1x
		Xeon 12-core	2.047s	4.5818x	0.642s	3.7414x
	GPUSVM	Tesla C2050	3.267s	2.8708x	0.573s	4.192x
		Tesla C2070	2.238s	4.1908x	0.526s	4.5665x
web	Libsvm	Xeon 1-core	1450.933s	1x	59.278s	1x
		Xeon 12-core	199.784s	7.2625x	6.819s	8.6931x
	GPUSVM	Tesla C2050	94.317s	15.3836x	1.267s	46.7861x
		Tesla C2070	71.291s	20.3523x	1.217s	48.7083x
mnist	Libsvm	Xeon 1-core	256.579s	1x	86.559s	1x
		Xeon 12-core	64.04s	4.0065x	10.183s	8.5003x
	GPUSVM	Tesla C2050	58.308s	4.4004x	1.154s	75.0078x
		Tesla C2070	39.552s	6.4871x	1.124s	77.0098x

4.4. Cross Validation Comparison

The speed performance of cross validation is not done due to the difficulties of setting a base line. Using more GPU devices roughly gives a linear speed improvement in the cross validation procedure. This is also true in Libsvm. When OpenMP is enabled in Libsvm, more cores bring similar linear improvement. If only one CPU core is used in Libsvm and one GPU device is used in GPUSVM, the speed performance of cross validation is close to the result shown in the previous one.

5. Conclusions and Future Work

In sum, GPUSVM shows excellent speed improvement compared to the state of the art Libsvm. It has as good performance as LIBSVM in terms of classification accuracy. GPUSVM also offers a nicer GUI for end-user. It includes cross validation tool, training tool and predicting tool and forms a complete package for solving real world SVM problems. It also supports multi-GPU hardware environment.The three-layer structure of GPUSVM makes it easier to add more components and functionalities in the future. However, GPUSVM still lacks certain

features compared to Libsvm such as support for $L2$ SVM and support for solving regression problems. The top and middle layer of GPUSVM relies on Java and Python environment which could be a problem in certain system.

GPUSVM is a starting point of building efficient SVM package for GPUs. We plan to integrate more functionalities in the future work such as support for solving regression problems which is the number one task. Other things such as different methods for multi-class classification, more kernel functions and more adaptable support for multi-GPU system will be continuously added to the package.

References

[1] Q. Li, V. Kecman, and R. Salman, "A chunking method for euclidean distance matrix calculation on large dataset using multi-gpu," in *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pp. 208–213, December 2010.

[2] Q. Li, R. Salman, and V. Kecman, "An intelligent system for accelerating parallel svm classification problems on large datasets using gpu," in *Intelligent Systems Design and Applications (ISDA), 2010 10th International Conference on*, pp. 1131–1135, Nov 2010.

[3] NVIDIA, *CUDA CUBLAS Library*, June 2007.

[4] INTEL, *Math Kernel Library 10.3*, 2011.

[5] W. D. Hillis and G. L. Steele Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, pp. 1170–1183, December 1986.

[6] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th international conference on Machine learning*, ICML '08, (New York, NY, USA), pp. 104–111, ACM, 2008.

[7] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Springer, 2000.

[8] J. C. Platt, *Fast training of support vector machines using sequential minimal optimization*, pp. 185–208. Cambridge, MA, USA: MIT Press, 1999.

[9] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," in *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, pp. 276–285, September 1997.

[10] T. Joachims, *Making large-scale support vector machine learning practical*, pp. 169–184. Cambridge, MA, USA: MIT Press, 1999.

[11] C. B. S. S. Keerthi, S. K. Shevade and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural Computation*, vol. 13, pp. 637–649, March 2001.

[12] P.-H. C. R.-E. Fan and C.-J. Lin, "Working set selection using second order information for training support vector machines," *Journal of Machine Learning Research*, vol. 6, pp. 1889–1918, December 2005.

[13] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[14] T.-M. Huang, V. Kecman, and I. Kopriva, "Iterative single data algorithm for kernel machines from huge data sets: Theory and performance," in *Kernel Based Algorithms for Mining Huge Data Sets*, vol. 17 of *Studies in Computational Intelligence*, pp. 61–95, Springer Berlin / Heidelberg, 2006.

[15] S. B. R. Collobert and Y. Bengio, "A parallel mixture of svms for very large scale problems," *Neural Computation*, vol. 14, no. 5, pp. 1105–1114, 2002.

[16] A. K. J.-X. Dong and C.-Y. Suen, "A fast parallel optimization for training support vector machine," in *Machine Learning and Data Mining in Pattern Recognition*, vol. 2734 of *Lecture Notes in Computer Science*, pp. 96–105, Springer Berlin / Heidelberg, 2003.

[17] G. Zanghirati and L. Zanni, "A parallel solver for large quadratic programs in training support vector machines," *Parallel Computing*, vol. 29, pp. 535–551, April 2003.

[18] C.-K. S. G.-B. Huang, K. Z. Mao and D.-S. Huang, "Fast modular network implementation for support vector machines," *Neural Networks, IEEE Transactions on*, vol. 16, pp. 1651–1663, November 2005.

[19] L. J. Cao, S. S. Keerthi, C.-J. Ong, J. Q. Zhang, U. Periyathamby, X. J. Fu, and H. P. Lee, "Parallel sequential minimal optimization for the training of support vector machines," *Neural Networks, IEEE Transactions on*, vol. 17, pp. 1039–1049, July 2006.

[20] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," in *In Advances in Neural Information Processing Systems*, pp. 521–528, MIT Press, 2005.

[21] S. Herrero-Lopez, J. R. Williams, and A. Sanchez, "Parallel multiclass classification using svms on gpus," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, (New York, NY, USA), pp. 2–11, ACM, 2010.

[22] S. Abe, *Support Vector Machines for Pattern Classification (Advances in Pattern Recognition)*. London: Springer-Verlag, 2005.

[23] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, September 1995.

[24] A. Frank and A. Asuncion, "UCI machine learning repository," 2010.

[25] LIACC, "Statlog datasets," 2010.

[26] J. Hull, "A database for handwritten text recognition research," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, pp. 550–554, may 1994.

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, nov 1998.