LibSVM: A Parallelized Implementation Using GPUs

Daniel Rivera Ruiz
Department of Computer Science
New York University
daniel.rivera@nyu.edu

Abstract

Support Vector Machines (SVMs) are a widely used algorithm for supervised machine learning tasks such as classification and linear regression. They are characterized by strong theoretical guarantees and a simple model that scales seamlessly to higher dimensional spaces. However, the optimization problem associated with an SVM can be computationally expensive, rendering the algorithm less attractive for big datasets. In this project we explore LibSVM, a well-known open-source library designed to train SVM models, and the possibility of parallelizing some of the tasks in the pipeline of the SVM problem setting (scaling, training and prediction) utilizing Graphics Processing Units (GPUs).

1 Introduction

In the context of supervised learning, support vector machines are a very popular algorithm for both classification and regression due to their flexibility and high predictive power. However, one major obstacle in their application is execution time, which scales approximately cubically with the number of observations in the training set. Combined with the fact that not one but multiple model fits have to be performed, for hyperparameter tuning or for vote aggregation in multi-class classification problems, their runtime often becomes prohibitively large beyond 100k or 1 million observations.
Mayer et al. [1] propse a simple yet elegant solution to this problem: The Cascade SVM. A cascade SVM is nothing but a group of SVMs working in parallel with disjoint subsets of the input data, generating several models that are brought together until a unified model is produced. The implementation of the original Cascade SVM was designed for a multicore system.
In this project we take the idea of the Cascade SVM and try to implement it in a different setting: a GPU. While a multicore system has the advantage of more powerful cores, GPUs are known to excel at parallelized algorithms such as the Cascade SVM. Additionaly, we work on top of the LibSVM library, which is one of the most commonly used open-pieces of software when it comes to SVM modeling. The library offers three main tasks for the SVM modeling pipeline - parameter scaling, training and prediction - all of them implemented in a sequential manner. Through this project we explore the possibility of bringing these tasks to the parallel world using GPUs.
The rest of the document is organized as follows: in section 2 we present a brief theoretical background on support vector machines and kernel methods. In section 3 we introduce the LibSVM library, its main features and the way it is conceived.Section 4 deals with related work, in particular the quadratic optimization algorithm underlying LibSVM and the Cascade SVM algorithm. In section 5 we present our implementation of Cascade SVM on top of LibSVM designed to run in a GPU environment. The results we obtained with this implementation are presented in section 6, and finally sections 7and 8 present some guidelines for future work and conclusions.

2 Support Vector Machines

Support Vector Machines (SVM) were originally conceived to address the problem of binary linear classification. Given an input space $\mathcal{X} \in \mathbb{R}^N$ with $N \geq 1$, an output space $\mathcal{Y} = \{-1, 1\}$ and a mapping function $f : \mathcal{X} \longrightarrow \mathcal{Y}$, the binary linear classification problem consists of finding a hypothesis h, i.e. a binary classifier, with small generalization error [2]:

$$R_D(h) = \Pr_{x\sim D}[h(x) \neq f(x)]$$

The SVM algorithm approach is to select h from the hypothesis set H of linear classifiers or hyperplanes, which can be defined as follows:

$$H = \{\mathbf{x} \mapsto \text{sign}(\mathbf{w}\cdot\mathbf{x} + b) : \mathbf{w} \in \mathbb{R}^N, b \in \mathbb{R}\}$$

A hypothesis of the form $\mathbf{x} \mapsto \text{sign}(\mathbf{w}\cdot\mathbf{x} + b)$ thus labels positively all points falling on one side of the hyperplane $\mathbf{w}\cdot\mathbf{x} + b = 0$ and negatively all others. The solution returned by the SVM algorithm is the hyperplane with the maximum margin, or distance to the closest points, and is thus known as the *maximum-margin hyperplane*.

If the training points in the input space are linearly separable, the SVM algorithm will return a hyperplane such that $y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1$ for all values of i. In practice, however, this is rarely the case, and so slack variables ξ_i are introduced to relax this constraint:

$$y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \xi_i$$

Here, a slack variable ξ_i measures the distance by which vector $\mathbf{x}_i$ violates the desired inequality $y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1$. Thus, for a hyperplane $\mathbf{w}\cdot\mathbf{x} + b = 0$, a vector $\mathbf{x}_i$ with $\xi_i > 0$ can be viewed as an outlier. Figure 1 illustrates this situation. These outliers, along with the vectors that lie on the marginal hyperplanes ($y_i(\mathbf{w}\cdot\mathbf{x}_i + b) = 1$) are called *support vectors* and thus the name of the algorithm Support Vector Machines.

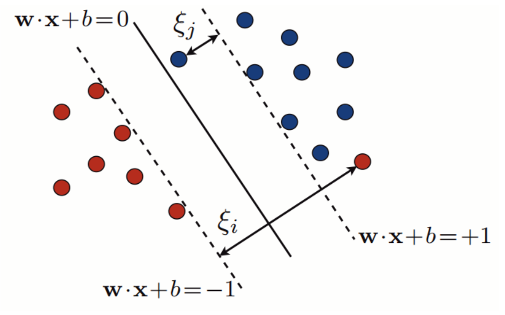

$\mathbf{w}\cdot\mathbf{x}+b=0$
ξ_j
ξ_i
$\mathbf{w}\cdot\mathbf{x}+b=+1$
$\mathbf{w}\cdot\mathbf{x}+b=-1$

Figure 1: A separating hyperplane with point x_i classified incorrectly and point x_j correctly classified, but with margin less than 1.

2.1 Optimization Problem

Under the conditions described in the previous section, the primal optimization problem for SVMs can be formulated as follows

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \quad \frac{1}{2}||\mathbf{w}||^2 \;+\; C\sum_{i=1}^{m}\xi_i \tag{1}$$

$$\text{subject to} \quad y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \xi_i \;\wedge\; \xi_i > 0,\; i \in [1,m]$$

Where the parameter $C \geq 0$ determines the trade-off between margin maximization (minimization of $||\mathbf{w}||^2$) and the minimization of the penalty introduced by the slack variables. This parameter is typically determined via n-fold cross validation.

Since the problem defined by equation 1 and the affine constraints associated to it are convex and differentiable, a dual version can be derived by applying the method of Lagrange multipliers:

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{m}\alpha_i \;-\; \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j(\mathbf{x}_i\cdot\mathbf{x}_j) \tag{2}$$

$$\text{subject to} \quad 0 \leq \alpha_i \leq C \;\wedge\; \sum_{i=1}^{m}\alpha_i y_i = 0,\; i \in [1,m]$$

The solution $\boldsymbol{\alpha}$ of the dual problem in equation 2 can be used directly to determine the hypothesis returned by SVMs:

$$h(\mathbf{x}) = \text{sgn}(\mathbf{w}\cdot\mathbf{x} + b) = \text{sgn}(\alpha_i y_i(\mathbf{x}_i\cdot\mathbf{x}) + b)$$

2.2 Kernel Methods

Kernel methods are widely used in machine learning. They are flexible techniques that can be used to extend algorithms such as SVMs to define non-linear decision boundaries [2]. The main idea behind these methods is based on or kernel functions, which implicitly define an inner product in a high-dimensional space. Replacing the original inner product in the input space (like the one that appears in the dual optimization problem for SVMs in equation 2) with kernels immediately extends the algorithm to a linear separation in the high-dimensional space, or, equivalently, to a non-linear separation in the input space.
Formaly, a kernel function K is defined such that for any two points $x, x' \in \mathcal{X}$, $K(x, x')$ is equal to an inner product of vectors $\Phi(x)$ and $\Phi(y)$:

$$\forall x, x' \in \mathcal{X}, \quad K(x, x') = \langle \Phi(x), \Phi(x') \rangle$$

for some mapping $\Phi : \mathcal{X} \mapsto \mathbb{H}$ to a Hilbert Space $\mathbb{H}$ called the *feature space*. Since an inner product is a measure of the similarity of two vectors, K is often interpreted as a similarity measure between elements of the input space $\mathcal{X}$. An important advantage of using kernels is efficiency: K is often significantly more efficient to compute than Φ and an inner product in $\mathbb{H}$. The kernels implemented in the LibSVM library will be presented in section 3.
With the introduction of kernels, the dual optimization problem for the SVM algorithm can be rewritten as follows:

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{3}$$

$$\text{subject to} \quad 0 \le \alpha_i \le C \, \wedge \, \sum_{i=1}^{m} \alpha_i y_i = 0, \; i \in [1, m]$$

and so the the hypothesis solution h can be written as:

$$h(\mathbf{x}) = \text{sgn}(\alpha_i y_i K(\mathbf{x_i}, \mathbf{x}) + b) \tag{4}$$

This form of the optimization problem for SVMs is the most common and the one that will be referenced in further sections.

3 The LibSVM Library

LibSVM is a widely used integrated software for solving classification (binary and multi-class), regression and distribution estimation problems using SVMs [3]. Although the previous section focused only on the binary classification problem, the theoretical results presented are easily extrapolated to the multi-class, regression and distribution estimation settings.
The library is fully developed in C/C++ (some interfaces to other popular languages have also been implemented) and offers functionalities for the three main steps of an SVM algorithm pipeline: parameter scaling, model training and prediction.

3.1 Parameter Scaling

Although several ML algorithms can benefit from parameter scaling, it is crucial to perform it for SVMs in order to obtain reasonable results and accuracy. The intuition behind parameter scaling is to keep attributes in greater numerical ranges from dominating those in smaller numerical ranges. Since SVMs try to maximize the distance between the separating plane and the support vectors, they are less sensitive to changes in features with smaller numerical ranges which directly impacts their accuracy. Furthermore, scaling helps to avoid numerical difficulties during training: given that kernel values usually depend on the inner products of the feature vectors, large attribute values might cause numerical instability.
The most common techniques implemented for parameter scaling are the following:

$$x_i = \frac{x_i - \mu}{\sigma}$$

$$x_i = l + (u - l) \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{5}$$

In the first case, μ and σ are the mean and standard deviation of the data, respectively. In the second case, l and u are parameters provided by the user to map the input space from $[x_{min}, x_{max}]$ to $[l, u]$. In LibSVM, the second approach is implemented.

3.2 Model Training

This is the main step of the process and it is where the actual optimization problem from equation 3 is solved. Details on the algorithm used to solve it are presented in section 4. It is important to notice, however, that regardless of the algorithm several kernel functions K can be plugged in equation 3. For the particular case of LibSVM, linear, polynomial, radial basis function (RBF) and sigmoid kernels are implemented:

$$\begin{aligned} K_L(u,v) &= u^T v \\ K_P(u,v) &= (\gamma u^T v + c_0)^d \\ K_{RBF}(u,v) &= e^{-\gamma|u-v|^2} \\ K_S(u,v) &= \tanh(\gamma u^T v + c_0) \end{aligned}$$

The output from the training phase is a model structure that includes all the information necessary to perform predictions and that fully characterizes the problem solved.

3.3 Prediction

During the prediction phase, the main objective is to test the model to see how it performs on previously unseen data. Usually, a set of labeled data (which is disjoint from the training data) is reserved for testing, By doing this, the accuracy of the model can easily be measured:

$$acc = \frac{1}{T}\sum_{i=1}^{T} \mathbb{1}(h(x_i) = y_i)$$

where T is the number of data points in the test set and $\mathbb{1}$ is the indicator function. To calculate the predictions $h(x_i)$, the parameters available in the model structure from the previous step are used along with equation 4.

4 Related Work

In [4], Fan et al. propose a very ingenious algorithm to solve the SVM dual optimization problem from equation 3 and is the one that Chang et al. implemented in [3]. First they rewrite equation 3 as follows:

$$\min_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \mathbf{e}^T \boldsymbol{\alpha} \tag{6}$$

where Q is an $m \times m$ symmetric matrix with $Q_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$. The matrix Q is usually fully dense and may be too large to be stored. Decomposition methods are designed to handle such difficulties by modifying only a subset of $\boldsymbol{\alpha}$ per iteration. This subset, denoted as the working set B, leads to a small sub-problem to be minimized in each iteration. An extreme case is the Sequential Minimal Optimization, which restricts B to have only two elements. Then in each iteration one does not require any optimization software in order to solve a simple two-variable problem. Under these conditions, the algorithm can be described as follows:

1. Find $\boldsymbol{\alpha}^1$ as the initial feasible solution. Set $k = 1$.
2. If $\boldsymbol{\alpha}^k$ is an optimal solution of 6, stop. Otherwise, find a two-element working set $B = \{i, j\} \in \{1, \ldots, m\}$. Define $N \equiv \{1, \ldots m\} \setminus B$ and let $\boldsymbol{\alpha}_B^k$, $\boldsymbol{\alpha}_N^k$ be sub-vectors of $\boldsymbol{\alpha}^k$ corresponding to B and N, respectively.
3. Solve the optimization problem for $\boldsymbol{\alpha}_B^k$ (which only has two elements).
4. Set $\boldsymbol{\alpha}_B^{k+1}$ to be the optimal solution from the previous step and $\boldsymbol{\alpha}_N^{k+1} \equiv \boldsymbol{\alpha}_N^k$. Set $k \leftarrow k + 1$ and go to step 2.

It is easy to see from the nature of the algorithm that it is not easily parallelizable, since the value at iteration k for the optimal solution $\boldsymbol{\alpha}_B^k$ depends explicitly on the value of the previous iteration $k-1$. This condition holds for most algorithms proposed to solve SVMs, and together with the high computational cost of the runtime complexity, applying SVMs to larger datasets becomes problematic. To mitigate this, Mayer et al. [1] proposed the Cascade SVM. The Cascade SVM is a step-wise procedure that combines the results of multiple regular support vector machines to create one final model. The main idea is to iteratively reduce a data set to its crucial data points before the last step. This is done by locating potential support vectors and removing all other samples from the data. Not only does this approach yield a significant speedup, but it can easily be parallelized because a number of independent models have to be fitted during each stage of the cascade. The basic steps of the method, which is depicted in figure 2, are as follows:

1. Partition the data into k disjoint subsets of preferably equal size.
2. Independently train an SVM on each of the data subsets.
3. Combine the SVs of, e. g., pairs or triples of SVMs to create new subsets.
4. Repeat steps 2 and 3 for some time.
5. Train an SVM on all SVs that were finally obtained in step 4.

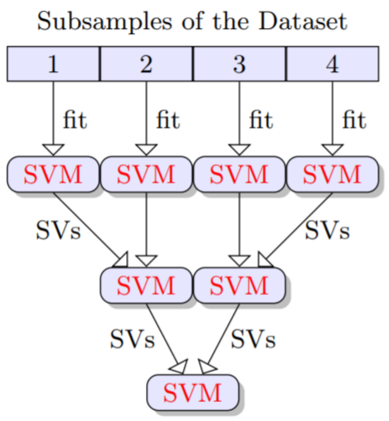


Figure 2: Scheme for the Cascade SVM Model proposed by Meyer et al.

5 Implementation

The main objective of this project was to come up with parallelized implementations for GPUs of the scaling, training and predicting routines available in the LibSVM library. The following subsections explain the methodology applied to transform each sequential routine into its parallel counterpart, as well as the problems encountered along the process.

5.1 Parameter Scaling

Looking at equation 5, it is easy to see that the parameter scaling process can be easily parallelized: the scaled version of each element depends only on itself, the values l and u, which are constant and provided as input, and the values x_{min} and x_{max}. The parallelized implementation of the scaling routine can be conceptually described as follows:

1. Read in the input file and store it as an `svm_problem` structure. This structure basically contains m nodes of the form $(\mathbf{x}_i, y_i)$, i.e. the feature vector and its associated label. The `svm_problem` structure is read into unified memory using `cudaMallocManaged`.
2. Asynchronously launch two kernels using streams, one to calculate the maximum value of each feature and one to calculate the minimum. Each kernel consists of a single block of $1\,024$ threads, which means that each thread will be responsible of updating $m/1\,024$ min and max values.

3. To accelerate the updating process, an array of size n equals the dimension of the feature space is allocated in shared memory and all threads write to it via atomic operations.
4. Once x_{min} and x_{max} are available for all features, launch a third kernel with m threads. Each thread is responsible to update the values of all features for only one node in the input dataset.
5. Write the updated `svm_problem` with the scaled values back to the file system.

Since the parameter scaling process lends itself to be parallelized, there were not many difficulties encountered. The results obtained with this routine and the comparison to the sequential counterpart are presented in section 6.

5.2 Model Training

This subroutine was the most problematic in terms of implementation and the least fruitful in terms of results. The basic idea was to implement the Cascade SVM model presented in [1], partitioning the input dataset and assigning one SVM subproblem to each thread in the device. This worked fine for small size problems, where delegating the task to the GPU is not even profitable due to the high overhead associated to data allocation, transfer and kernel launch.
For larger datasets, the parallelized implementation on the GPU wasn't even able to terminate properly due to the high requirements of dynamically allocated memory in the optimization algorithm. In contrast to this implementation, Mayer et al. used multiple CPUs to train the Cascade SVM. This probably makes more sense because memory allocation is not really an issue and because the threads of the CPU are much more robust, which makes them more suitable to handle the large amounts of work required to solve a whole SVM problem by themselves.
That being said, the parallelized version of the training routine performs quite poorly when compared to its sequential counterpart: it only works for small datasets and it takes much longer to terminate. The accuracy achieved, however, is comparable for both implementations.

5.3 Prediction

Similarly to parameter scaling, the prediction routine is also easy to parallelize. Since for each prediction the only information necessary are the model parameters (which are the same for all nodes) and all nodes are independent from one another, the predictions could theoretically be performed all at once. The parallelized implementation of the prediction routine can be conceptually described as follows:

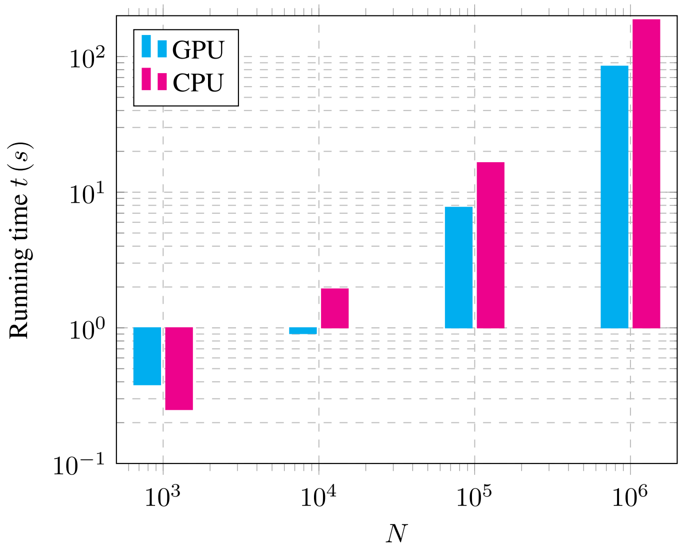
1. Read the test data file and store it as an `svm_problem` structure in unified memory using `cudaMallocManaged`.
2. Read the model information file (generated by the training routine) into an `svm_model` structure in unified memory using `cudaMallocManaged`.
3. Launch a kernel with m threads, each thread being responsible to calculate the prediction for one data point.
4. In the host, calculate accuracy values and write the predictions to the file system.

Even though conceptually it is a simpler task than scaling, the steps under the hood to actually calculate the predictions are computationally intensive and memory demanding. In particular it was very important to calculate the memory required by the whole kernel beforehand, in order to allocate it in unified memory and avoid dynamically allocations within the device itself. Not doing this would very likely result in issues similar to the ones encountered for the parallelized training routine.

6 Analysis and Results

To test and compare the sequential and parallelized implementations, we used a copy of the MNIST dataset in LibSVM format. This dataset consists of 8 million handwritten digits, each one represented as a vector of 780 features classified into one of 10 classes (i.e. the decimal digits 0-9). To evaluate performance for different data sizes, we generated 4 subsets with 1k, 10k, 100k and 1M elements each. We ran both versions of the scaling routine with each one of the subsets as input. We measured

Figure 3: Execution time for the scaling task for different input size. Both axes are shown in logarithmic scale.



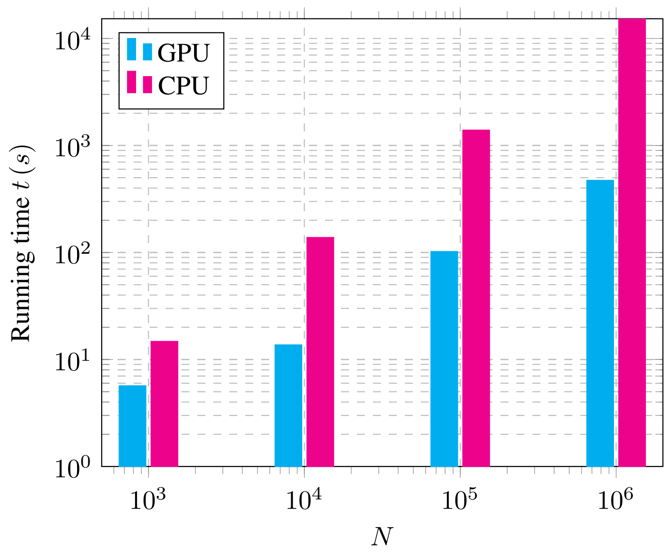
the execution time for each experiment using the linux `time` command. The results are depicted in figure 6.
Since the parallelized version of the training routine was not working properlly for big datasets, and the focus of the project was not to achieve high accuracy, a simple model was trained using the sequential routine and 60k elements. This model was used as the input for the prediction task.
We used the scaled versions of the 4 data subsets generated with the scaling routines and the pre-trained model to compare the performance of the two prediction routines. Again, we measured the execution time for each experiment using the linux `time` command. The results are depicted in figure 6.
Table 1 summarizes the results for both tasks (scaling and prediction) and includes a speedup factor S calculated as the running time of the sequential task, divided by the running time of the parallelized task. Two results are worth remarking from this table:

1. The speedup factor doesn't seem to be increasing a lot with the size of the problem for the scaling task. This means that the part of the algorithm that is being parallelized doesn't account for much of the total execution time. This makes sense, because there is a lot of data being read and written from external files, which can only be executed sequentially.
2. In contrast, the prediction task seems to benefit a lot more fro the parallelization as the problem size increases. Although there's also some data being written at the end of the task, it isn't nearly as much as for scaling (in this case it is only labels and an accuracy value). Furthermore, as it had been previously explained, the actual task being parallelized (prediction) is much more computationally intensive.

Enclosed with this paper, the file *nvprof.out* provides additional information retrieved using NVDIA profiler *nvprof* for the parallelized tasks . Two results stand out from this report and are worth noticing:

1. For the scaling routine, most of the time is spent in finding the maximum and the minimum values of the features. This makes sense because the block size for these tasks is fixed to $1\,024$, which means that for bigger problem sizes each thread has to update more values. However, this is a small price to pay compared to a scenario with more threads, in which case the serialization factor introduced by the atomic operations would be much larger.
2. For the prediction routine, memory allocation and device synchronization account for a significant amount of time. This can be explained by the fact we mentioned before that this task has high memory demands which are fulfilled by means of unified memory. Considering

Figure 4: Execution time for the prediction task for different input size. Both axes are shown in logarithmic scale.



Task	Input size	$t_{\text{CPU}}(s)$	$t_{\text{GPU}}(s)$	S
Scale	1k	0.250	0.381	0.656
	10k	1.934	0.908	2.130
	100k	16.488	7.731	2.133
	1M	186.561	84.834	2.199
Predict	1k	14.733	5.672	2.597
	10k	137.913	5.672	10.085
	100k	1390.427	13.675	13.640
	1M	15218.03	101.934	**32.271**

Table 1: Comparison of the execution times for the Scaling and Prediction tasks with inputs of different sizes, taken from the MNIST dataset for LibSVM.

that this memory has to be available both at the host and the device, the synchronization overhead is understandable.

7 Future Work

Based on the good results obtained for the parallelized prediction task using a pre-allocated amount of unified memory, we believe that applying a similar approach to the training task could also yield favorable results and eliminate the problems encountered due to memory limitations within the device. If enough space can be allocated in unified memory for the first level of the Cascade, the training time is likely to be considerably reduced for large datasets. This follows from the fact that in deeper levels of the Cascade only support vectors are considered instead of all the data points, and in practice the number of SVs is usually much smaller than the size of the training set.
Modifying the parallelized implementations for the scaling and prediction routine to support multi-GPUs systems would also be an interesting task. Doing this would help to answer the question if these routines can further benefit from more parallelization and more computational power.
Finally, once the three parallelized tasks have been properly tuned, it would be ideal to implement a unified LibSVM library with only one executable per task. Each executable would be smart enough to decide, based on the nature and size of the input data, whether to run it in the GPU or the CPU.

8 Conclusions

In this project we attempted to implement a Cascade SVM algorithm on top of the LibSVM library to run in a GPU. Although the results at this point are not as expected, we have reason to believe that by means of a careful memory allocation process, the algorithm should be able to run in a GPU and outperform the sequential version. Additionally, we parallelized the parameter scaling and prediction tasks of the LibSVM library, which are much more GPU friendly in nature than the training task, and thus we obtained very favorable results in terms of speedup in comparison to the sequential counterparts.

References

[1] O. Meyer, B. Bischl, and C. Weihs, "Support vector machines on large data sets: Simple parallel approaches," in *GfKl*, 2012.

[2] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. The MIT Press, 2012.

[3] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[4] R.-E. Fan, P.-H. Chen, and C.-J. Lin, "Working set selection using second order information for training support vector machines," *J. Mach. Learn. Res.*, vol. 6, pp. 1889–1918, Dec. 2005.