Multiclass Classification using Support Vector Machines on GPUs

Sergio Herrero-Lopez
sherrero@mit.edu

Abstract

The scaling of serial algorithms cannot rely on the improvement of CPUs anymore. The performance of classical Support Vector Machine (SVM) implementations has reached its limit and the arrival of the multi core era requires these algorithms to adapt to a new parallel scenario. Graphics Processing Units (GPU) have arisen as high performance platforms to implement data parallel algorithms. In this paper, it is described how a naïve implementation of a multiclass classifier based on SVMs can map its inherent degrees of parallelism to the GPU programming model and efficiently use its computational throughput. Empirical results show that the training time of the classifier can be reduced an order of magnitude compared to a classical solver, LIBSVM, while guaranteeing the same accuracy.

1. Introduction

Since the semiconductor industry revealed that physical constraints were imposing an unbeatable upper frequency limit for processors, the computing market has been continuously supplied with multi-core, multi-processor, and most recently GPU and multi-GPU architectures. This shift to parallel architectures was initially ignored, but the demand for performance and scalability eventually has required adapting not only new algorithms, but also old methods to the new existing panorama. Unfortunately, not all the algorithms see their performance enhanced by parallel architectures, but those that do, often called *data parallel* algorithms [1], can be greatly benefited by their adaptation to this new scenario.

Support Vector Machines [2] (SVM) is a learning algorithm that can be conveniently adapted to parallel architectures. Its success solving classification tasks in a wide variety of fields such as text or image processing and medical informatics, have stimulated research not only on their generalization performance, but also in their execution performance and scalability. Nevertheless, the training phase of a SVM is a computationally expensive task. The training time of a binary tasks composed of 100.000 points with tens of dimensions can often take on the order of hours of serial execution.

The efforts to reduce training time have been numerous and effective. Osuna et al. presented a decomposition approach that enabled tackling larger problems by solving sub-problems iteratively [3]. Joachims' introduced additional techniques such as shrinking and kernel caching that are common practice today [4]. Platt presented the Sequential Minimal Optimization (SMO) algorithm, which decomposed the QP problem into a series of smaller QP sub problems that were solvable analytically without the need of the time consuming QP optimization [5]. More recently, Fan et al. developed a series of working set selection heuristics that lead the SMO algorithm to a faster convergence [6]. The combination of these distributions has enabled fast *serial* SVM implementations.

Efforts have not only been focused on the development of techniques to accelerate "serial" SVMs; there are several initiatives that seek to achieve performance gains by either *parallelizing* the algorithm to fit in new architectures (PSVMs), *distributing* the classification problem across nodes in a cluster (DSVMs), or a hybrid of both. Cao et al. [7] present a parallel version of the SMO algorithm that divides the large dataset into smaller subsets where Platt's SMO is applied. Considerable performance gains are reported when executing this approach on multiprocessors machines that use MPI for communication. Similarly, Zanni et al. explored an iterative decomposition technique that would use both storage and computing resources available in multiprocessor systems [8]. Graf et al. designed the Cascade SVM, an algorithm that distributed the classification task into a cascade top-down network topology [9]. This was the first initiative to include network topologies in DSVM problems. Finally, Lu et al. [10] present their DPSVM approach, which follows the basic principles of the Cascade SVM but generalizes the distributed classification problem for general strongly connected network topologies and provides a convergence proof for them.

There is little work on how SVM algorithms map to GPU architectures. Their computational, memory and host - device or device - device communication possibilities seem to provide a convenient platform for the execution of the entire range of SVM alternatives, from simple individual SVMs to complex DPSVM network configurations. Catanzaro et al. [11] presented a pioneer implementation of a binary SVM on a Graphics Processing Unit (GPU), and reported promising speed ups (5-32x) over LIBSVM [12].

In this paper, we continue the research direction opened by Catanzaro et al. and implement a multi-class classifier on a GPU. This work shows that the resources and degrees of parallelism provided by a GPU can be conveniently exploited to train large scale multiclass classification tasks. Although there are a variety of methods to solve the SVM training problem, SMO algorithm was chosen for its popularity and more importantly, because it defines data reusability patterns that can be efficiently exploited by the GPU.

The organization of this paper is as follows. Section 2, briefly introduces the SVM training and classification problem for binary tasks and problems with multiple classes. Section 3 gives an overview of the GPU architecture and programming model. Section 4 describes the Parallel SMO algorithm, the motivation to execute binary tasks concurrently, along with the details of our implementation. The performance results of our multiclass classifier are compared with LIBSVM in Section 5. Section 6 gives the conclusions of this project and section 7 discusses future work.

2. Multiclass Classification

This section succinctly reviews the basic principles of soft-margin binary SVM classification and its combination to solve multiclass classification problems.

2.1. Binary SVM

The binary classification problem is defined as finding the classification function that solves the following regularized learning problem: Given l examples $(\bar{x}_1, y_1), \dots, (\bar{x}_l, y_l)$ with $\bar{x}_i \in R^n$ and $y_i \in \{-1,1\}\ \forall i$.

$$\min_{f \in H} C \sum_{i=1}^{l} V(y_i, f(\bar{x}_i)) + \frac{1}{2}\|f\|_K^2 \qquad (1)$$

where the regularization is controlled via C. Classical SVM arises by considering a specific loss function:

$$V(y_i, f(\bar{x}_i)) = (1 - yf(\bar{x}))_+ \qquad (2)$$

where $(k)_+ = \max(k, 0)$. Then slack variables ξ_i are introduced to overcome the problem introduced by its non-differentiability:

$$\min_{f \in H} C \sum_{i=1}^{l} \xi_i + \frac{1}{2}\|f\|_K^2 \qquad (3)$$

subject to: $y_i f(x_i) \geq 1 - \xi_i$ and $\xi_i \geq 0\ i = 1, \dots, l$.

The dual form of this problem is given by:

$$\max_{\alpha \in R^l} \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\alpha^T K \alpha \qquad (4)$$

subject to: $\sum_{i=1}^{l} y_i \alpha_i = 0$ and $0 \leq \alpha_i \leq C\ i = 1, \dots, l$. where $K_{ij} = y_i y_j \mathrm{k}(\bar{x}_i, \bar{x}_j)$ is a kernel function. (4) is a quadratic programming optimization problem. Common kernel functions are shown in Table 1.

Linear	$\mathrm{k}(\bar{x}_i, \bar{x}_j) = \bar{x}_i \bar{x}_j$
Polynomial	$\mathrm{k}(\bar{x}_i, \bar{x}_j) = (a\bar{x}_i \bar{x}_j + b)^d$
Radial Basis	$\mathrm{k}(\bar{x}_i, \bar{x}_j) = e^{-\beta \|\bar{x}_i - \bar{x}_j\|^2}$
Sigmoid	$\mathrm{k}(\bar{x}_i, \bar{x}_j) = tanh(a\bar{x}_i \bar{x}_j + b)$

Table 1: Kernel functions

Solving (4) defines the classification function:

$$f(x) = \sum_{i=1}^{l} y_i \alpha_i \mathrm{k}(\bar{x}, \bar{x}_i) + b \qquad (5)$$

where b is an unregularized bias term.

2.2. Multiclass SVM

In multiclass classification given l examples $(\bar{x}_1, y_1), \dots, (\bar{x}_l, y_l)$ with $\bar{x}_i \in R^n$, $y_i \in Y\ \forall i$ and $Y = \{1, .., M\}$ the goal is to design a classifier that predicts the label of new unseen samples. Classical approaches construct the multiclass classifier as the combination of N independent binary classification tasks. Binary tasks are defined in the output code matrix R of size M x N, where M is the number of classes and N is the number of tasks, and $R_{ij} \in \{-1,0,1\}$. Each column in R represents how original labels are translated into binary labels for each specific binary task. Then each $f^k(\bar{x})$ is trained separately with $(\bar{x}_1, R_{y_1 k}), \dots, (\bar{x}_l, R_{y_l k})$ where $k = 1..N$. The outputs of trained binary classifiers $\hat{f}^k(\bar{x})$ are used to predict the class label that best agrees with the binary predictions:

$$\hat{y} = \operatorname*{argmax}_{y \in Y} \left\{ \sum_{k=1}^{N} R_{yk} \hat{f}^k(\bar{x}) \right\} \qquad (6)$$

In general, predictions can be derived from output codes by specifying a loss function:

$$\hat{y} = \underset{y \in Y}{\operatorname{argmin}} \left\{ \sum_{k=1}^{N} Loss(R_{yk}, \hat{f}^{k}(\bar{x})) \right\} \qquad (7)$$

The common types of output codes are:

- One-vs-All (OVA): M classifiers are needed. For the $f^{i}(\bar{x})$ classifier, the positive examples are all the points in class i, and the negative samples all the points not in class i.
- All-vs-All (AVA): $\binom{M}{2}$ classifiers are needed, one classifier to distinguish each pair of classes i and j. $f^{ij}(\bar{x})$ is the classifier where class i has positive samples and class j negative.
- Error correcting codes: Often error correcting codes are applied to reconstruct labels from noisy predicted binary labels.

3. General Purpose GPU

The popularity and relatively low price of graphics processors have motivated many programmers to use GPUs for scientific computing. Even though they were designed specifically for triangle rasterization, today they have evolved to serve general purpose computation needs. Since NVIDIA released Compute Unified Device Architecture (CUDA) [13] in 2007, a variety of parallel programs have been developed for a variety of different applications, including fluid dynamics, finance or imaging. The key to CUDA's success are three abstractions that can be integrated using extensions to conventional C code [14]: (1) The hierarchy of thread groups, (2) shared memory, and (3) barrier synchronization. The combination of multiple levels of threads, a memory hierarchy and synchronization mechanisms enable achieving fine-grained data parallelism which can be conveniently interleaved with coarse-grained data parallelism and task parallelism.

Nevertheless, GPUs do not speed up all possible applications. The algorithms need to explicitly express parallelism by the execution of thousands of threads, so that available resources in the GPU are efficiently occupied. Fortunately, machine learning algorithms are typically composed by primitives that are highly parallelizable [15]: (1) Inner products, (2) outer products, (3) linear algebra, (4) the application of non-linearities to vectors or matrices, and (5) matrix transposes.

In November 2006, NVIDIA presented the Tesla architecture, a massively multithreaded processor array capable of concurrently executing tens of thousands of threads [16]. This architecture was designed for computation rather than control flow and caching, and one of its state-of-the art devices reported almost a Teraflop of processing power which is over an order of magnitude larger than the latest CPUs existing today. Furthermore, this high throughput in floating point computation, along with abundant memory at each layer the hierarchy model and communication through fast memory bandwidth have yield to promising acceleration results.

Next the CUDA programming model and the memory model are briefly introduced.

3.1. CUDA Programming Model

The CUDA programming model is based on a logical representation composed by three elements: grids, blocks and threads. This logical representation is generated by the user, and CUDA maps this representation to the real hardware representation underneath. This separation between the logical and physical representations allows algorithms scaling as the newest GPUs increase their capabilities.

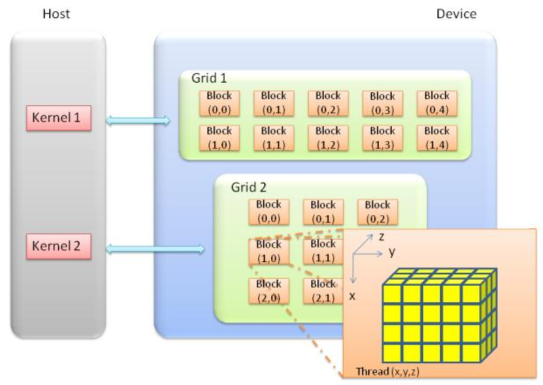

Figure 1: Logical Representation

As a first step, it is programmer's task to adapt the algorithm to a 2D grid structure. Grid executions, known as kernel calls, are sequentially invoked by the host. Grids are composed by blocks, which are groups of threads that share local memory and can be synchronized using barriers. Similarly, threads in a block are organized in 3D. The logical representation is illustrated in Figure 1.

The maximum size of each dimension of a block is (512, 512, 64), but the maximum number of threads in a block cannot exceed 512. The maximum size of each dimension of a grid is (65535, 65535). The blocks in a grid are launched in parallel, which allows a large number of threads being executed in parallel. The number of threads that run simultaneously on a block, which is called *warp*, and the number of blocks that run simultaneously on a grid are hardware implementation specific and depend on the number of Stream Multiprocessors (SMs) and Stream Processors (SPs) available in the device. The number of SMs and SPs increases every generation.

Consequently, developers need to find an appropriate partitioning of the data that occupies the maximum number of blocks possible, and hence utilize hardware resources uninterruptedly, in order to get maximum acceleration of the algorithm.

3.2. CUDA Memory Model

CUDA provides a hierarchy of memories that differ on their accessibility, operability and speed. These memories are illustrated in Figure 2.

- Registers: The smallest but fastest memory available. It is only accessible at the thread level with Read/Write operations.
- Shared Memory: Slightly slower than registers. It is shared by all the threads in a block, and allows Read/Write operations.
- Global Memory: The largest but slowest memory available. Accessible by all the threads executed in the grid, and allows Read/Write operations.
- Constant Memory: It is faster than global memory, and similarly is accessible at the grid level, but is Read only.

Latest generations of GPUs provide 102GB/s memory bandwidth on the GPU and 8GB/s for communication with the CPU across the PCI-express bus.

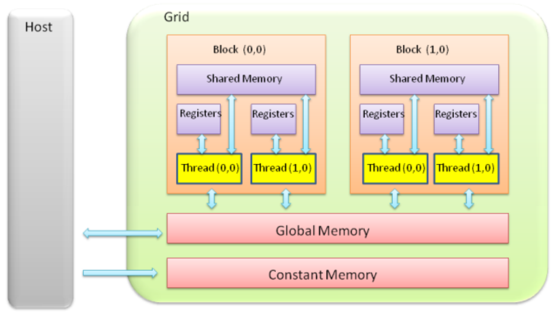

Figure 2: Memory Model

4. Multiclass SVM Implementation

Each of the N binary tasks is trained using the Sequential Minimal Optimization (SMO) algorithm, which solves equation (4). SMO solves very large quadratic programming (QP) optimization problems by breaking it into a series of smaller QP sub problems. These QP sub problems can be solved analytically without the need of numerical optimization. In order to adapt the algorithm to the CUDA programming model, the Parallel SMO (PSMO) version of the algorithm was implemented [7].

In this section, the PSMO algorithm is described. Next, its mapping to grids, blocks and threads is explained. Finally, the implications of the execution of N PSMO instances in parallel are analyzed.

4.1. Parallel SMO Algorithm

Cao et al. designed the PSMO algorithm aiming to accelerate the binary SVM training time by partitioning the algorithm across multiple processors. Their experiments reported considerable speedups while maintaining the accuracy of the sequential SMO. Next, this algorithm is explained:

Since, N binary tasks need to be executed, the correspondence between an instance of the PSMO algorithm and a binary task is represented by the superscript k, where $k = 1..N$. Given P processing units per binary task, the sample dataset l is partitioned in P subsets and one subset is given to each processing unit. The subsets are represented by $\{l^p\}\ \ p = 1\ldots P$, where $\bigcup_{p=1}^{P} l^p = l$.

We follow the following notation:

$$I_0^k = \{i: y_i^k = 1, 0 < \alpha_i^k < C\} \cup \{i: y_i^k = -1, 0 < \alpha_i^k < C\} \quad (8)$$

$$I_1^k = \{i: y_i^k = 1, \alpha_i^k = 0\} \quad (9)$$

$$I_2^k = \{i: y_i^k = -1, \alpha_i^k = C\} \quad (10)$$

$$I_3^k = \{i: y_i^k = 1, \alpha_i^k = C\} \quad (11)$$

$$I_4^k = \{i: y_i^k = -1, \alpha_i^k = 0\} \quad (12)$$

Each of the p subsets of a task k will have its own set of variables:

$$f_i^{p,k} = \sum_{j=1}^{l} \alpha_j^k y_j^k k(\bar{x}_j, \bar{x}_i) - y_i^k \quad (13)$$

$$b_{up}^{p,k} = \min\{f_i^{p,k}: i \in I_0^k \cup I_1^k \cup I_2^k \cup\ l^p\} \quad (14)$$

$$I_{up}^{p,k} = \underset{i}{\operatorname{argmin}} f_i^{p,k} \tag{15}$$

$$b_{low}^{p,k} = \max\{f_i^{p,k} : i \in I_0^k \cup I_3^k \cup I_4^k \cup l^p\} \tag{16}$$

$$I_{low}^{p,k} = \underset{i}{\operatorname{argmax}} f_i^{p,k} \tag{17}$$

Global variables representing the entire dataset can be obtained from the subset variables:

$$b_{up}^{k} = \min\{b_{up}^{p,k}\} \tag{18}$$

$$I_{up}^{k} = \underset{I_{up}^{p,k}}{\arg} b_{up}^{k} \tag{19}$$

$$b_{low}^{k} = \max\{b_{low}^{p,k}\} \tag{20}$$

$$I_{low}^{k} = \underset{I_{low}^{p,k}}{\arg} b_{low}^{k} \tag{21}$$

I_{up}^k and I_{low}^k are the indices of the two weights α_i^k of the smallest QP sub problem and they can be solved analytically:

$$\alpha_{I_{up}}^{new,k} = \alpha_{I_{up}}^{old,k} - \frac{y_{I_{up}}^k (f_{I_{low}}^{old,k} - f_{I_{up}}^{old,k})}{\eta} \tag{22}$$

$$\alpha_{I_{low}}^{new,k} = \alpha_{I_{low}}^{old,k} + s(\alpha_{I_{up}}^{old,k} - \alpha_{I_{up}}^{new,k}) \tag{23}$$

where

$$s = y_{I_{up}}^k y_{I_{low}}^k \tag{24}$$

$$\eta = 2k\left(\bar{x}_{I_{low}}, \bar{x}_{I_{up}}\right) - k\left(\bar{x}_{I_{low}}, \bar{x}_{I_{low}}\right) - k\left(\bar{x}_{I_{up}}, \bar{x}_{I_{up}}\right) \tag{25}$$

$\alpha_{I_{up}}^{new,k}$ and $\alpha_{I_{low}}^{new,k}$ need to be clipped to $[0, C]$.

After optimizing the weights the error on the i^{th} data pattern, $f_i^{p,k}$ needs to be updated:

$$f_i^{p,new,k} = f_i^{p,old,k} + \left(\alpha_{I_{low}}^{new,k} - \alpha_{I_{low}}^{old,k}\right) y_{I_{low}}^k k\left(\bar{x}_{I_{low}}, \bar{x}_i\right) + \left(\alpha_{I_{up}}^{new,k} - \alpha_{I_{up}}^{old,k}\right) y_{I_{up}}^k k\left(\bar{x}_{I_{up}}, \bar{x}_i\right) \tag{26}$$

The iterative algorithm is summarizes as follows:

```
Initialize:
        $\alpha_i^k = 0$, $f_i^{p,k} = -y_i^k, i \in l^p$,
        $p = 1 \dots P$, $k = 1 \dots N$
Calculate:
        $b_{up}^{p,k}$, $I_{up}^{p,k}$, $b_{low}^{p,k}$, $I_{low}^{p,k}$, $p = 1 \dots P$, $k = 1 \dots N$
Obtain:
        $b_{up}^{k}$, $I_{up}^{k}$, $b_{low}^{k}$, $I_{low}^{k}$, $k = 1 \dots N$

Iterate task $k$ until $b_{low}^k > b_{up}^k + 2\tau$
        Optimize $\alpha_{I_{up}}^k, \alpha_{I_{low}}^k$
        Update $f_i^{p,k}$ , $p = 1 \dots P$
        Calculate $b_{up}^{p,k}$, $I_{up}^{p,k}$, $b_{low}^{p,k}$, $I_{low}^{p,k}$, $p = 1 \dots P$
        Obtain $b_{up}^{k}$, $I_{up}^{k}$, $b_{low}^{k}$, $I_{low}^{k}$
Repeat
```

Figure 3: PSMO algorithm

Finally the offset for each task k is calculated:

$$b^k = \frac{b_{up}^k + b_{low}^k}{2} \tag{27}$$

4.2. PSMO Implementation on GPU

For a GPU implementation, there is a natural mapping between the execution of N PSMO algorithm instances and the grid structure defined by the CUDA programming model. Given P processing units and N binary tasks, a grid composed by PxN blocks can be used to execute the most computationally expensive steps of the training phase.

The horizontal dimension of the grid is partitioned into P blocks, and each block p will process a subset l^p of the training samples. Each sample i within a block is handled by a single thread. Threads within a block are organized in a single dimension. The vertical dimension of the grid indicates the task k that is processed by the block.

Blocks in the same column of the grid share the same training samples, but since they belong to different tasks they will have different labels.

Rows of the grid represent an instance of the PSMO algorithm. A single instance of the PSMO algorithm is illustrated in Figure 4.

The computation of the subset variables is carried out by executing P blocks in parallel in the GPU. Given the fact that the number of subset variables is reduced, calculating global variables in the host results in better performance that its calculation on the GPU. Global variables are then used to compute the weights $\alpha_{I_{up}}^k, \alpha_{I_{low}}^k$. Since this step involves kernel evaluations, it needs to be carried out in the GPU. After the weights have been updated, P blocks are executed to calculate the new $f_i^{p,k}$ values. Finally, the stop criterion is checked to determine whether the PSMO instance has converged or needs to proceed to a new iteration.

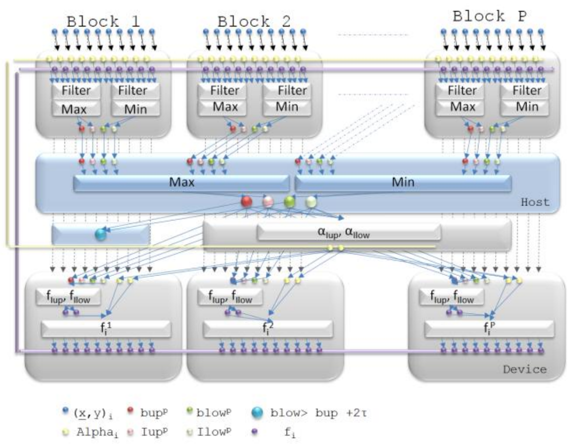

Figure 4: Parallel SMO for a single binary task

4.3. Task Parallelization Implications

Although binary tasks are independently trained, there are two direct implications associated to their parallel execution:

4.3.1. Cross-Task Kernel Caching: As the dimensionality of the samples increases, kernel evaluations become the most computationally expensive step of SVM training. Since SMO algorithm focuses on finding and optimizing non-zero weights, the algorithm tends to demand the same rows of the Gram matrix K several times as it approaches convergence. For large datasets, it is not feasible to store the entire matrix K on memory; hence it is a common practice to implement kernel caching mechanisms that exploit the reusability of rows in matrix K. SVM$_{\text{Light}}$ uses an LRU caching strategy [4].

The concurrent execution of multiple binary tasks that share the same memory allows different tasks sharing kernel evaluations. If a training sample is found to be a support vector in more than one task, a single kernel evaluation will be shared among those tasks and the kernel cache hit rate will increase. A representation of support vectors being shared among tasks is illustrated in Figure 6. Our implementation exploits this beneficial property. Similarly, if multiple tasks need to evaluate the same non-cached row of matrix K in the same iteration, the row is evaluated once and shared with the others avoiding multiple evaluations. Empirical results are presented in Section V.

In cache miss situations, new rows of the Gram matrix are calculated using CUDA Basic Linear Algebra Subroutines (CUBLAS), which provide optimized functions for matrix-vector multiplications [17]. The optimization of these routines has been confirmed by [18]

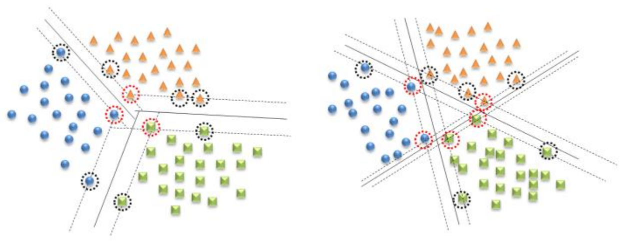

Figure 6: Shared SVs (red). AVA (Left), OVA (Right)

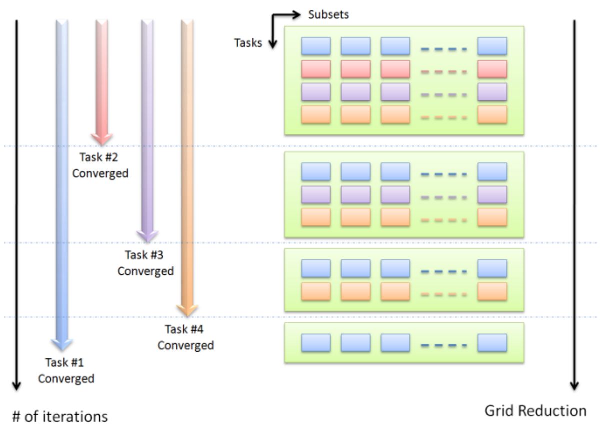

Figure 5: Dynamic Grid Reduction

4.3.2. Progressive Grid Reduction: Each of the N binary tasks has different convergence rates. If any of the tasks has already converged, a fixed PxN grid would require launching idle rows of blocks. Even if passive blocks do not need to run, they would require to be assigned to the underlying hardware like the rest of blocks. Consequently they would hold GPU resources and delay the execution of non-converged blocks. Hence, it is recommended dynamically reducing the vertical dimension of the grid as binary tasks converge.

5. Performance Results

This section presents the performance results of executing this GPU implementation of the multiclass classifier, compared with LIBSVM. For both cases, the same kernel type, regularization parameter C, and stopping criteria is used. LIBSVM is also based on the SMO algorithm. It uses the AVA output code for multiclass classification and executes binary tasks sequentially [19]. Both the LIBSVM cache size and the GPU implementation kernel cache size were set to be of equal size, 2GB.

In this section prediction performance was not considered. The author believes that prediction performance depends directly on the optimization of the kernel evaluation, which can be efficiently done using CUBLAS Library.

5.1. Host and Device

The specifications of the hardware used for the experiments in this section are presented in Table 2.

Host	Device
Ubuntu 8.10 64bit	Tesla C1060
CPU: Intel Core i7 920 @ 2.67 GHz	# Stream Processors: 240
Memory 6GB (3x2 DDR2)	Frequency of Processors: 1.3GHz
	933 Gflops
	Memory: 4GB DDR3
	Memory Bandwidth: 102GB/s
Host <-> Device	
PCIe x16 (8GB/s)	

Table 2: Host and Device Specifications

5.2. Datasets

The GPU implementation was tested on well known datasets. Initially, Adult dataset [20] was used to test the correctness of single binary classifications tasks. Then, MNIST dataset [21] was used to analyze the performance in multiclass problems. The sizes of these datasets and the parameters used for training are indicated in Table 3.

Dataset	# Training Points	# Testing Points	# Features	# Classes	C	β
Adult	32,561	16,281	123	2	100	0.5
MNIST	60,000	10,000	780	10	10	0.125

Table 3: Tested Datasets

5.3. Classifier Accuracy

Binary tasks are the smallest classification units in which accuracy can be evaluated. The classification performance of the multiclass classifier directly depends on the accuracy of the binary tasks. Latest GPUs provide IEEE 754 capabilities with both single precision and double precision support [22]. Unfortunately, it is reported that performance in double precision floating point performance drops more than an order of magnitude. For these experiments, single precision was used. In this subsection the accuracy of the GPU implementation training phase using well known binary tasks is compared to the results provided by LIBSVM. Table 4 shows the accuracy results comparison for binary classification. For the case of the MNIST dataset, the 10 class problem was converted to a 2 class problem by doing even-vs-odd classification.

Results show that classification accuracy in the GPU does as well as the LIBSVM solver. Even if both optimization algorithms run with a tolerance value $\tau = 0.001$, there is some variation on the number of support vectors and the value of the offset. It is speculated that this difference might be due to the application of second order heuristics [23] or *shrinking* techniques [4] in LIBSVM.

Dataset	SVM	Accuracy (%)	# SVs	Difference in b (%)	# Iterations
Adult	GPU	82.697624	18668	0.01	115565
	LIBSVM	82.697624	19058		43735
MNIST	GPU	96	43730	0.04	69535
	LIBSVM	96	43756		76385

Table 4: Binary Classification Accuracy

5.4. Cross-Task Kernel Caching Performance

Figure 7 and Figure 8 show the measurements of kernel cache performance as the number of parallel binary tasks executed is increased. In subsection 4.3, the possibility that shared support vectors across tasks would improve the cache hit rate was explained. Empirical results on the MNIST dataset, both for OVA and AVA output codes, confirmed this behavior, and showed that kernel evaluations can be avoided by sharing previously computed Gram matrix rows among binary tasks.

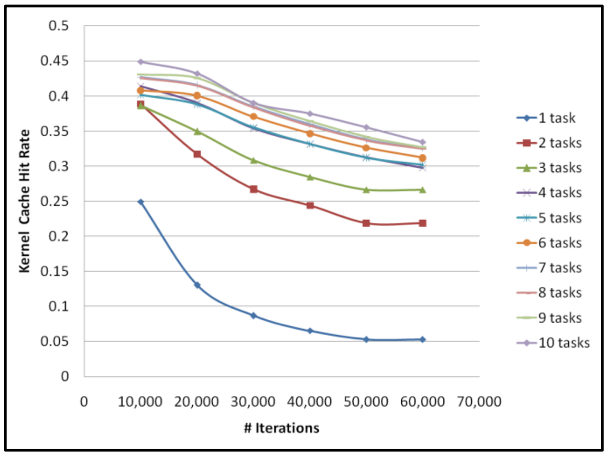

Figure 7: MNIST (OVA) Kernel Cache Hit Rate

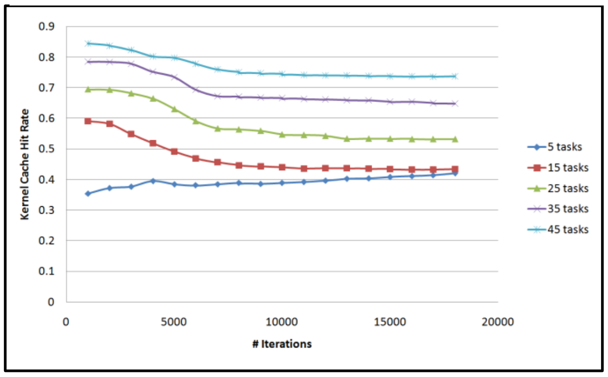

Figure 8: MNIST (AVA) Kernel Cache Hit Rate

5.5. Training Time

The GPU implementation based on the principles described in previous sections yield to a promising acceleration of the training phase of the multiclass classification problem. The results are shown in Table 5. The performance increase for a single binary task is noticeable, while the acceleration of the multiclass problem reduced the overall time from almost 8 hours to approximately 20 minutes.

Dataset	GPU (sec)		LIBSVM (sec)	Speedup
Adult	38.05		479	12.58 x
	OVA (10 tasks)	AVA (45 tasks)	AVA (45 tasks)	
MNIST	2272.71	1217.33	27833	22.86 x

Table 5: Performance Results

Figure 9 and Figure 10 show the evolution of the training time of the MNIST dataset (OVA and AVA) as more tasks are executed concurrently. In both cases, a linear behavior is presented. This linear behavior neither does represent purely serial execution of the tasks nor a purely parallel execution of them.

In the OVA case, 1172 blocks are required to be executed in parallel by the GPU per iteration. In AVA, the number of blocks required is 5274. Unfortunately, the number of processors in the GPU is not capable of executing this number of blocks simultaneously. Hence, they are executed in sequential batches according to the number of processors available in the GPU. For this reason, the slope of the line is an intermediate value between the purely sequential case (concatenation of the duration of the tasks) and the purely parallel case (duration of the longest task).

The key benefit is that using the CUDA programming model, the training time will approach the purely parallel case as the number of processors available increases in the future generations of GPUs.

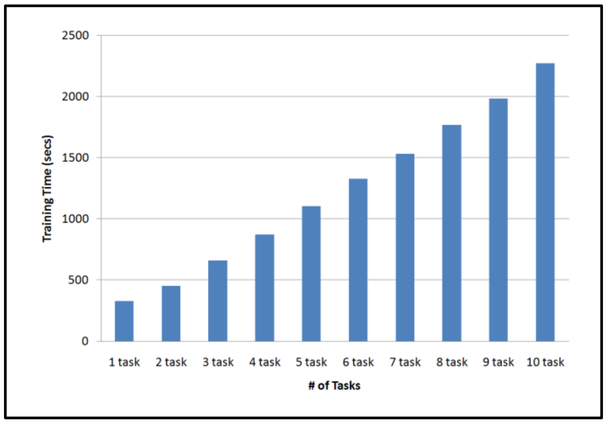

Figure 9: MNIST (OVA) Training Time

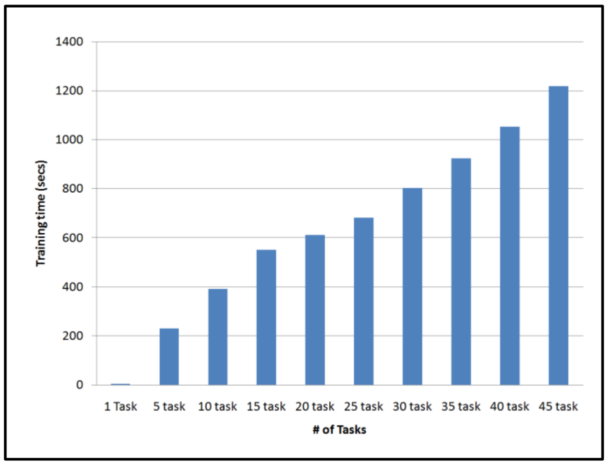

Figure 10: MNIST (AVA) Training Time

6. Conclusions

The raise of GPUs as massive parallel processors opens a wide range of opportunities for the acceleration and scaling of learning algorithms. The data parallel nature of many learning algorithms fits conveniently the set of problems that modern GPUs are meant to solve. Besides, previous research in accelerating SVMs in multiprocessor systems or scaling SVMs in computer clusters can be ported to smaller and cheaper GPU or multi GPU configurations where memory systems are aggressive and communications are considerably faster than networked environments.

It has been shown in this paper that a naïve implementation of the SMO algorithm on a single GPU can lead to speedups in the range of 13-23x, which reduced the training time more than an order of magnitude while maintaining the accuracy of the classification tasks. This multiclass SVM classifier implementation leaves room for improvement and better results could potentially be achieved by using more involved SVM training techniques [23] [24]. Nevertheless, this work showed that the GPU programming model conveniently allowed executing multiple binary tasks in parallel over the same global memory. This fact benefited the training time not only because of the parallel execution, but also due to the reusability of data across binary tasks as it was confirmed by the empirical results.

7. Future Work

Not only classic algorithms can be adapted to state-of-the-art programming models, the latest research on statistical learning algorithms can benefit from them as well. New techniques for large scale learning should be built taking into account this new era of multi core and GPU systems, in order to make training of large size problems practical or allow real-time training of smaller size problems.

The latest research on large scale SVMs uses network topologies to partition the data [8] [9]. A priori, these algorithms may find convenient the use of multi GPU configurations due to the availability of large amounts of memory, and the data transfer speed between devices.

It is a natural continuation of the multiclass classification work to explore the implementation of distributed classification approaches, such as Cascade SVM or DPSVM, by creating a network topology composed by multiple GPU devices that work on partitions of data concurrently.

8. References

[1] Hillis, W. D. and Steele, G. L. 1986. Data parallel algorithms. Commun. ACM 29, 12 (Dec. 1986), 1170-1183.

[2] V. N. Vapnik The Nature of Statistical Learning Theory New York: Springer-Verlag, 1995.

[3] Osuna, E., Freund, R., & Girosi, F. (1997). An improved training algorithm for support vector machines. *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, 276–285.

[4] T. Joachims, Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning, B. Schölkopf and C. Burges and A. Smola (ed.), MIT-Press, 1999.

[5] John C. Platt, Fast training of support vector machines using sequential minimal optimization, Advances in kernel methods: support vector learning, MIT Press, Cambridge, MA, 1999.

[6] R.-E. Fan, P.-H. Chen, and C.-J. Lin. Working set selection using the second order information for training SVM. *Journal of Machine Learning Research* 6, 1889-1918, 2005.

[7] Cao, L.J.; Keerthi, S.S.; Chong-Jin Ong; Zhang, J.Q.; Periyathamby, U.; Xiu Ju Fu; Lee, H.P., "Parallel sequential minimal optimization for the training of support vector machines," *Neural Networks, IEEE Transactions on* , vol.17, no.4, pp. 1039-1049, July 2006.

[8] Zanni, L., Serafini, T., and Zanghirati, G. 2006. Parallel Software for Training Large Scale Support Vector Machines on Multiprocessor Systems. *J. Mach. Learn. Res.* 7 (Dec. 2006), 1467-1492.

[9] Graf, H. P., Cosatto, E., Bottou, L., Dourdanovic, I., & Vapnik, V. (2005). Parallel support vector machines: The cascade svm. In L. K. Saul, Y. Weiss and L. Bottou (Eds.), *Advances in neural information processing systems 17*, 521--528. Cambridge, MA: MIT Press.

[10] Yumao Lu; Roychowdhury, V.; Vandenberghe, L., "Distributed Parallel Support Vector Machines in Strongly Connected Networks," *Neural Networks, IEEE Transactions on* , vol.19, no.7, pp.1167-1178, July 2008.

[11] Catanzaro, B., Sundaram, N., and Keutzer, K. 2008. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international Conference on Machine Learning* (Helsinki, Finland, July 05 - 09, 2008). ICML '08, vol. 307. ACM, New York, NY, 104-111.

[12] Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[13] NVIDIA CUDA Compute Unified Device Architecture. Programming Guide NVIDIA Corporation. June 2007.

[14] Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable Parallel Programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40-53.

[15] Steinkrau, D., Simard, P. Y., and Buck, I. 2005. Using GPUs for Machine Learning Algorithms. *In Proceedings of the Eighth international Conference on Document Analysis and Recognition* (August 31 - September 01, 2005). ICDAR. IEEE Computer Society, Washington, DC, 1115-1119.

[16] Lindholm, E.; Nickolls, J.; Oberman, S.; Montrym, J., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE* , vol.28, no.2, pp.39-55, March-April 2008.

[17] CUDA. CUBLAS Library. NVIDIA Corporation. June 2007.

[18] Barrachina, S.; Castillo, M.; Igual, F.D.; Mayo, R.; Quintana-Orti, E.S., "Evaluation and tuning of the Level 3 CUBLAS for graphics processors," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* , vol., no., pp.1-8, 14-18 April 2008.

[19] C.-W. Hsu and C.-J. Lin. A comparison of methods for multi-class support vector machines , *IEEE Transactions on Neural Networks*, 13(2002), 415-425.

[20] Asuncion, A., & Newman, D. (2007). UCI machine learning repository.

[21] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P.(1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86, 2278–2324.

[22] IEEE 754-2008 Standard for Floating-Point Arithmetic.

[23] S. S. Keerthi , S. K. Shevade , C. Bhattacharyya , K. R. K. Murthy, Improvements to Platt's SMO Algorithm for SVM Classifier Design, *Neural Computation*, v.13 n.3, p.637-649, March 2001.

[24] Joachims, T. 2006. Training linear SVMs in linear time. In Proceedings of the 12th *ACM SIGKDD international Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA, August 20 - 23, 2006). KDD '06. ACM, New York, NY, 217-226.