A GPU-Tailored Approach for Training Kernelized SVMs

Andrew Cotter
Toyota Technological Institute at Chicago
6045 S. Kenwood Ave.
Chicago, Illinois 60637
cotter@ttic.edu

Nathan Srebro
Toyota Technological Institute at Chicago
6045 S. Kenwood Ave.
Chicago, Illinois 60637
nati@ttic.edu

Joseph Keshet
Toyota Technological Institute at Chicago
6045 S. Kenwood Ave.
Chicago, Illinois 60637
jkeshet@ttic.edu

ABSTRACT

We present a method for efficiently training binary and multiclass kernelized SVMs on a Graphics Processing Unit (GPU). Our methods apply to a broad range of kernels, including the popular Gaussian kernel, on datasets as large as the amount of available memory on the graphics card. Our approach is distinguished from earlier work in that it cleanly and efficiently handles sparse datasets through the use of a novel clustering technique. Our optimization algorithm is also specifically designed to take advantage of the graphics hardware. This leads to different algorithmic choices then those preferred in serial implementations. Our easy-to-use library is orders of magnitude faster then existing CPU libraries, and several times faster than prior GPU approaches.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Design, Experimentation, Performance

1. INTRODUCTION

Support Vector Machines (SVMs) are among the most popular general purpose learning methods in use today. SVM learning amounts to learning a linear predictor, with regularization (corresponding to a "large margin") ensuring good generalization even in very high dimensions. This predictor need not be linear in the input representation: it is possible to learn a linear predictor in some extremely high dimensional space specified implicitly through a *kernel function*. SVMs were originally suggested in the context of binary classification, but more recently variants following the same principles have also been developed and successfully applied to more complex prediction tasks such as multiclass classification and prediction of structured outputs such as sequences.

Training an SVM amounts to solving a quadratic programming problem (see Section 2). Although general-purpose quadratic programming solvers can only handle fairly small SVM instances, much effort has been made in the past two decades to design special-purpose solvers that can handle large-scale SVM instances. This effort resulted in widely-used packages that can solve both "linear" SVMs (i.e. where the prediction is linear in the input representation) and "kernelized" SVMs (where a non-linear kernel defines the linear prediction space). For linear SVMs, stochastic methods such as PEGASOS [13] and Stochastic Dual Coordinate Ascent [8] have recently been established as being effective at solving extremely large SVM instances, typically in less time than that which is required to read the data into memory. For kernel SVMs, most leading solvers are based on decomposing the dual optimization problem into small subproblems [11, 9, 4, 1, and see also Section 3]. Such approaches can indeed handle fairly large problems, provided that the data fits in memory, but it is not uncommon for training to require many hours or days, even using state-of-the-art optimizers. There is therefore still a strong need for faster training of kernel SVMs.

One attractive possibility for enabling faster SVM training is to leverage the power of Graphical Processing Units (GPUs). GPUs are highly parallel, structured, computational engines and are now available relatively inexpensively and are found in many modern computers. In this paper we discuss how SVM training can be efficiently implemented on a GPU, and present such an implementation for both binary and multiclass SVMs.

Several authors have recently proposed using GPUs for kernelized SVM training [3, 2] and related problems [6]. These previous approaches, however, primarily focused on pointing out the advantages of implementing standard algorithms on graphics hardware, typically using GPU matrix-multiplication libraries, and not on how these algorithms can be modified to better take advantage of the GPU architecture. We study various algorithmic choices for SVM training in the context of GPUs, discuss how the optimal choices and algorithms on a GPU are different than those for a serial implementation, and arrive at an implementation specifically designed for graphics hardware. As with many previous approaches, we assume that the dataset fits in memory, and focus mostly on the Gaussian kernel, although our implementation can handle any kernel function which can be written in the form $K(x, y) = f(\|x\|, \|y\|, \langle x, y \rangle)$ (see Section 5.2), and our ideas apply even more broadly to any kernel which is an aggregation of element-wise operations.

One particularly significant drawback of other GPU SVM solvers is their lack of support for sparse datasets. On the CPU, taking advantage of sparsity is a simple matter, and sparse datasets are encountered frequently enough that many widely-used SVM solvers treat all input vectors as sparse, by default [9, 4, 1]. On the GPU, however, maximum performance is only achieved if memory accesses follow certain fairly-restrictive patterns, which are difficult

KDD'11, August 21–24, 2011, San Diego, California, USA.

to ensure with sparse data. In contrast to other GPU SVM solvers, our implementation does take advantage of sparsity in the training set through a novel "sparsity clustering" approach (Section 5.3).

Overall, our implementation is orders of magnitudes faster then existing CPU implementations, and several times faster on sparse datasets then prior GPU implementations of SVM training.

2. SUPPORT VECTOR MACHINES

We will briefly review the optimization problems that need to be solved in order to train binary and multiclass Support Vector Machine (SVM) classifiers. For a complete description of Support Vector Machines, motivating these optimization problems, we refer the reader to, e.g., Schölkopf and Smola [12].

2.1 Binary classification

We consider training a kernel SVM with an unregularized bias term. Let $(x_1, y_1), \ldots, (x_n, y_n)$, with $x_i \in \mathbb{R}^d$ and $y_i \in \{\pm 1\}$, be a training set of n labeled examples, and let $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be a (positive semi-definite) kernel. Here, we focus mostly on the Gaussian kernel $K(x_1, x_2) = e^{-\gamma \|x_1 - x_2\|_2^2}$, parametrized by a scale parameter $\gamma \in \mathbb{R}$, although the methods we present are applicable to a wide range of kernels (see Section 5.2). Given a regularization trade-off parameter $C \in \mathbb{R}$, training an SVM classifier amounts to solving the following optimization problem:

$$\underset{\alpha \in \mathbb{R}^n,\ b \in \mathbb{R}}{\text{minimize}} : \frac{1}{2}\alpha^T Q \alpha + C \sum_{i=1}^{n} \max\left(0, 1 - y_i\left(b + c_i\right)\right) \qquad (1)$$

where here and throughout we denote $c_i \overset{def}{=} \sum_{j=1}^{n} \alpha_j y_j K(x_i, x_j)$, which we will call the "responses", and $Q \in \mathbb{R}^{n \times n}$ is a matrix with entries $Q_{ij} \overset{def}{=} y_i y_j K(x_i, x_j)$. The first term in (1) is a regularization term (corresponding to a norm in an implied Hilbert space) and the second term is the empirical loss. After training, the label of an input vector $x \in \mathbb{R}^d$ is given by $\text{sign}(b + \sum_{j=1}^{n} \alpha_j y_j K(x, x_j))$.

As is commonly done, we instead solve the dual of (1):

$$\begin{aligned} \underset{\alpha \in \mathbb{R}^n}{\text{maximize}} &: \mathbf{1}^T \alpha - \frac{1}{2}\alpha^T Q \alpha \\ \text{subject to} &: \forall i \left(0 \le \alpha_i \le C\right) \\ &: \sum_{i=1}^{n} y_i \alpha_i = 0 \end{aligned} \qquad (2)$$

An optimum of (2) is also an optimum of (1), with $b = y_i - c_i$ for any i such that $0 < \alpha_i < C$. The goal of this paper is to suggest an efficient method for solving (2) on a GPU.

2.2 Multiclass Classification

For problems where the labels take on more then two possible values, $y_i \in \{1, \ldots, m\}$, we follow the multiclass SVM formulation of Crammer and Singer [5]. In this formulation, each class label y is associated with a coefficient vector $\alpha^{(y)} \in \mathbb{R}^n$, and no unregularized bias is allowed. Training amounts to solving the following optimization problem:

$$\begin{aligned} \underset{\alpha^{(1)}, \ldots, \alpha^{(m)} \in \mathbb{R}^n}{\text{minimize}} &: \frac{1}{2} \sum_{y=1}^{m} \left(\alpha^{(y)}\right)^T K \alpha^{(y)} \\ &+ C \sum_{i=1}^{n} \max_{y \in \{1, \ldots, m\}} \left(1 - \delta_{y, y_i} - c_i^{(y_i)} + c_i^{(y)}\right) \end{aligned} \qquad (3)$$

where we denote $c_i^{(y)} \overset{def}{=} \sum_{j=1}^{n} \alpha_j^{(y)} K(x_i, x_j)$, $K \in \mathbb{R}^{n \times n}$, $K_{ij} = K(x_i, x_j)$ is the Gram matrix and δ is the Kronecker delta function. Again, the first term is a regularization term (this time, the sum of the norms of the predictors for each class) and the second is again the empirical loss (a multiclass hinge loss—see Crammer and Singer for further details). After training, the label of an input vector $x \in \mathbb{R}^d$ is given by $\underset{y}{\text{argmax}} \left(\sum_{j=1}^{n} \alpha_j^{(y)} K(x, x_j)\right)$.

We will again solve the dual, which is given by:

$$\begin{aligned} \underset{\alpha^{(1)}, \ldots, \alpha^{(m)} \in \mathbb{R}^n}{\text{maximize}} &: \sum_{i=1}^{n} \alpha_i^{(y_i)} - \frac{1}{2} \sum_{y=1}^{m} \left(\alpha^{(y)}\right)^T K \alpha^{(y)} \\ \text{subject to} &: \forall i \forall y \left(\alpha_i^{(y)} \le C \delta_{y, y_i}\right) \\ &: \forall i \left(\sum_{y=1}^{m} \alpha_i^{(y)} = 0\right) \end{aligned} \qquad (4)$$

3. OPTIMIZATION IN THE DUAL

Efficient kernel SVM optimizers tend to work on the dual formulation (2). For most datasets, the kernel matrix is much too large to fit in memory. Therefore, optimization is typically done by iteratively choosing a subset of the dual variables, which we will call a *working set*, and updating the coefficients α_i corresponding to this set. While the early "chunking" algorithm [10] relied on choosing a *large* working set, subsequent work has tended to show that performing many computationally inexpensive updates on small working sets leads to faster convergence than performing fewer relatively expensive ones. Variants of the popular SMO algorithm take these working sets to be as small as possible (two with an unregularized bias, one without) [11, 7], while SVM-Light, by default, uses working sets of size 10 [9].

The subproblem of optimizing the coefficients corresponding to a given working set is again a quadratic program (though a smaller one). For a binary classification problem, let $\mathcal{I}$ to be the working set of indices to optimize (while holding the remainder fixed). We assume the current set of dual variables α satisfies the constraints, and consider as optimization variables the values Δ_i to *add* to each $\alpha_i : i \in \mathcal{I}$. The subproblem is then given by:

$$\begin{aligned} \underset{\Delta_i \in \mathbb{R} : i \in \mathcal{I}}{\text{maximize}} &: \mathbf{1}^T \Delta - \sum_{i \in \mathcal{I}} \Delta_i y_i c_i - \frac{1}{2} \sum_{i,j \in \mathcal{I}} \Delta_i \Delta_j Q_{i,j} \\ \text{subject to} &: \forall i \in \mathcal{I} \left(-\alpha_i \le \Delta_i \le C - \alpha_i\right) \\ &: \sum_{i \in \mathcal{I}} y_i \Delta_i = 0 \end{aligned} \qquad (5)$$

The submatrix $Q_{i,j} : i, j \in \mathcal{I}$ may be found by evaluating only $\frac{1}{2}|\mathcal{I}|\left(|\mathcal{I}| + 1\right)$ kernel inner products, so for small working sets it is reasonable to calculate it on-demand. Calculating each of the responses $c_i : i \in \mathcal{I}$, however, requires the evaluation of n kernel inner products.

This cost cannot be entirely avoided. However, instead of calculating the responses at each iteration, we can alternatively keep a complete set of up-to-date responses on-hand, and merely *update* them after each iteration:

$$c_i' = c_i + \sum_{j \in \mathcal{I}} \Delta_j y_j K(x_i, x_j) \qquad (6)$$

The computational cost of such an update is the same as that of calculating the needed responses ($c_i : i \in \mathcal{I}$) afresh at each iteration. However, we may benefit from having all responses available to us, because they are useful in *working set selection*. In recent years, it has been found that when optimizing linear SVMs without an unregularized bias, the cost of choosing a working set in some "smart" way is not justified–doing so randomly leads to far better

performance [13, 8]. For kernel SVMs, however, the cost of each update is sufficiently high that it is often worthwhile to choose the working set intelligently, even if this necessitates examining all values c_i at each iteration [14]. SVM optimizers differ in the heuristics used to choose the working set. We discuss this issue in detail in Section 5.

Training a multiclass SVM in the dual poses similar challenges, except that the situation is complicated by the fact that each example is associated with m dual variables, subject to inequality and equality constraints. This makes choosing the working set a more constrained, and thus more difficult, problem.

4. THE GPU ARCHITECTURE

Optimizing the dual problem (2) on a serial architecture is fairly well understood, and many successful implementations are available. In order to illustrate the different considerations involved in optimizing (2) on a highly parallel Graphics Processing Unit (GPU), we briefly discuss the relevant aspects of this specialized hardware. We make no attempt to provide a full description of the graphics hardware, and only highlight those aspects that most influence the different design decisions in a GPU versus CPU based implementation of kernel SVM training. We implemented our optimizer on a NVIDIA GPU using the Compute Unified Device Architecture (CUDA) tools, and refer here specifically to this hardware, although the high-level design considerations are also relevant for other graphics processors.

While one might simplistically think of a GPU as a massively-parallel execution environment for a large number of independent threads, full utilization of the graphics hardware is only possible if the operations performed by concurrently-executing threads are, in a sense, "compatible". In this regard, a GPU shares many aspects of Single-Instruction-Multiple-Data computers.

On the coarsest level, a CUDA program is executed by running the same code on some number of threads, which are distributed among the GPU's processors. The threads possess a small amount of fast local memory which is shared among blocks of threads (see below), but must copy data back and forth between this small fast local memory and the higher latency main memory.

In the problem which we are considering, the most important efficiency consideration is, overwhelmingly, access to main memory ("global memory"). This memory is relatively high bandwidth, but is also high latency. The graphics card performs automatic "latency hiding", in which a thread waiting on a memory access is swapped out for one which is not. This is helpful, but does not fully solve the problem—waiting for memory accesses can still often dominate the runtime of a GPU program. An important programming technique for improving the memory-access patterns of a GPU program is called "coalescing": whenever aligned blocks of 16 consecutive threads (with consecutive thread identifiers) access aligned blocks of 16 consecutive memory locations, these accesses will be "coalesced" into a single memory access, resulting in a significant speedup.

The threads involved in a computation are organized into "blocks" of some number of threads (256 in our application). Each such block has an associated pool of "shared memory", which is much faster than global memory, but slower then registers, and may be accessed by all of the threads in the block. The fact that shared memory is so much faster than global memory may be exploited through a technique known as "staging", in which blocks of global memory are copied into shared memory using coalesced reads, manipulated (not necessarily in a coalesced fashion) by the threads in a block, and then (possibly) staged back into global memory using coalesced writes.

In applications such as ours, where a moderate amount of computation is performed on a large amount of data, taking full advantage of coalescing (often via staging) is of the highest importance. Hence, we will focus primarily on this issue for the remainder of the paper.

5. OUR IMPLEMENTATION

We are now ready to describe the approach we take in optimizing the SVM training problems (2) and (4) on a GPU. Our optimizer closely follows the sketch of Section 3. Computation is divided between the CPU and GPU, with large parallel computations being performed on the graphics hardware, and lightweight serial tasks on the host processor. The vectors α are stored in both the host machine's main memory, and the graphics card's global memory. These two copies must be kept in sync throughout the optimization. The response vectors c are stored in the graphics card's global memory, with only the working set values (i.e. for $i \in \mathcal{I}$) passed to the CPU at each iteration. The cost of these memory transfers is negligible. We begin by initializing $\alpha = 0$ and $c = 0$, and then proceed iteratively, performing the following steps:

1. On the GPU, choose a working set $\mathcal{I}$
2. On the CPU, calculate the Gram matrix restricted to the working set $\mathcal{I}$, and optimize the subproblem (5). This results in updates to the values α_i for $i \in \mathcal{I}$.
3. On the GPU, update all elements of c in response to the change in α_i for $i \in \mathcal{I}$ as in (6).

For the subproblem size that we use, the cost of step (2), performed on the CPU, is insignificant, relative to the GPU portions of the algorithm. For multiclass problems we follow a similar approach, at each iteration optimizing over $\alpha_i^{(y)}$ for $i \in \mathcal{I}$ and all $y \in \{1, \ldots, m\}$.

While it is possible to use working sets as small as two, we use larger working sets, optimizing larger subproblems. This choice is motivated by the observation (Section 4) that the primary efficiency constraint, on the GPU, is not *computation*, but is rather *memory accesses*. We may calculate the kernel matrix rows corresponding to a sufficiently small working set in only one pass over the training data. Hence, in order to extract the maximal utility from each such pass, one would like to use working sets which are as large as is feasible. Conversely, one will often experience diminishing returns (in terms of the amount of "progress" made per iteration) as the size of the working set grows. We found the use of size 16 working sets to be most convenient.

For a sparse dataset, ensuring that memory accesses are coalesced is more difficult than it is for a dense dataset, since if "adjacent" threads are accessing training vectors with different sparsity patterns, then they will not be accessing adjacent memory addresses. We resolve this by first observing that, on sparse machine learning datasets, certain features may occur more frequently than others, or certain sets of features might tend to co-occur, resulting in distinct training vectors often having similar sparsity patterns. We therefore propose clustering the dataset by sparsity pattern, using a simple greedy heuristic, about which more will be said in Section 5.3. Our implementation is structured in such a way that every element of each thread block is always working on the same cluster, i.e. on vectors with the same sparsity pattern, permitting memory accesses to be coalesced.

In the following sections, we discuss the heuristic procedure used to choose the working set, and its GPU implementation (Step 1, discussed in Section 5.1), updating of the c_is (Step 3, discussed in Section 5.2), how sparsity is handled via clustering (Section 5.3),

and finally, we briefly mention how labels are predicted for new examples following training (Section 5.4).

5.1 Heuristics

The intuition behind the heuristics which we use to select the working set $\mathcal{I}$ is the same for both binary classification and multiclass: we wish to select the working set which will result in the largest increase in the dual objective function value. We use "first-order" heuristics, in the sense that we estimate the quality of a working set by looking only at the first derivatives of the dual objective. Second-order heuristics have been found to outperform first-order heuristics for binary classification problems with an unregularized bias, when the working set is of size 2 [7]. However, it is not clear how to efficiently generalize such heuristics to either the larger working sets which we must use in order to minimize the number of memory accesses, or to the multiclass objective.

For binary classification problems, we use the same heuristic as SVM-Light [9]. The goal of this heuristic is to choose those elements with respect to which the partial derivatives of the dual objective are large, and to do so in such a way that the set of chosen indices allows for large-magnitude changes which satisfy the constraints. The partial derivatives of the dual objective for binary classification (2) are:

$$\frac{\partial g_b}{\partial \alpha_i} = 1 - y_i c_i$$

Due to the equality constraint $\sum_i y_i \alpha_i = 0$, any increase in $y_i \alpha_i$ for one index i must be matched by corresponding decreases for others. As a result, our size-16 working set will be chosen to contain 8 elements maximizing y_i times the corresponding partial derivative, and 8 minimizing it. We must also respect the inequality constraints $0 \leq \alpha_i \leq C$, so cannot increase any $\alpha_i = C$, nor decrease any $\alpha_i = 0$. Hence, indices i with $\alpha_i = C$ are not considered as candidates for increase during working set selection, with the $\alpha_i = 0$ case being handled analogously.

For multiclass problems, our heuristic is taken from Crammer and Singer [5]. The intuition behind this heuristic is the same as for binary classification, although there are additional complications, due to the fact that there are multiple dual variables corresponding to each element of the training set. The partial derivatives of the multiclass dual objective (4) are:

$$\frac{\partial g_m}{\partial \alpha_i^{(y)}} = \delta_{y,y_i} - c_i^{(y)}$$

Here, δ is the Kronecker delta function. Ideally, we would choose those indices i which maximize the magnitude of the "row gradient" $\{\alpha_i^{(y)} : y \in \{1, \ldots, m\}\}$. However, this simple choice ignores the equality constraints. The ideal solution would be to project these row gradients onto the constraints, but doing so is itself a nontrivial optimization problem, and inappropriate as a component of a "simple" heuristic. Instead, we will approximate these projected row gradients by identifying, for each i, the pair of labels y_+ and y_- for which increasing the dual variable corresponding to the first while decreasing that corresponding to the second by the same amount (thus respecting the constraint $\sum_y \alpha_i^{(y)} = 0$) results in the largest first-order improvement in the dual objective:

$$y_+ = \operatorname*{argmax}_{y:\alpha_i^{(y)} < C} \frac{\partial g_m}{\partial \alpha_i^{(y)}}$$

$$y_- = \operatorname*{argmin}_{y} \frac{\partial g_m}{\partial \alpha_i^{(y)}}$$

Defining $v_i = {}^{\partial g_m}/_{\partial \alpha_i^{(y_+)}} - {}^{\partial g_m}/_{\partial \alpha_i^{(y_-)}}$ as the magnitude of the gradient in the direction induced by y_+ and y_- gives the heuristic value for the index i. One can see that this heuristic value lower-bounds the magnitude which would result from properly projecting the row gradient onto the constraints, and furthermore that it must be positive if a nonzero improvement is possible. The working set $\mathcal{I}$ will be composed of the 16 indices i maximizing maximizing v_i.

Both of these heuristics are implemented on the GPU. Calculation of the heuristic values themselves is computationally inexpensive, and may be performed independently for each training example. Hence, distributing this task among the graphics card's threads is straightforward. Finding the maximum (equivalently, minimum) values of these heuristics is a parallel max-reduction, which requires some care to implement efficiently. Our implementation uses a "chunking-and-sorting" procedure, which begins by breaking up the list of heuristic values into chunks, each of which is sorted in parallel using bitonic sort. Then, the maximum values of each chunk are copied into a new list, and we repeat until the total number of elements is below some threshold value. At this point, there are few enough values that parallelization on the GPU is no longer cost-effective, so the remaining heuristic values are copied to the CPU, and the maxima located.

This reduction step is computationally expensive—on our experimental datasets, performing it occupies a significant proportion of the runtime. For smaller working sets (say, size 2, rather than size 16), a similar heuristic, or even a superior second-order heuristic, could be implemented far more efficiently. We have found, however, that the benefits of using large working sets outweigh the extra time spent in the heuristic.

5.2 Updates

The final step in an iteration, and the "heart" of the optimization procedure, is updating the responses c in response to the changes in the dual variables α. Our general approach could theoretically support any kernel which may be written in terms of element-wise operations on the vectors, but our implementation currently only supports kernel functions which can be written in the form $K(x, y) = f(\|x\|, \|y\|, \langle x, y \rangle)$, a representation which supports several popular kernels, including the Gaussian, sigmoid and polynomial kernels. Efficient evaluation of kernel inner products is accomplished by precalculating $\|x\|$ for every training vector, and then calculating $\langle x, y \rangle$ by iterating over those vector elements which are nonzero in both x and y, taking the product of each such pair, and summing the results. The calculation of these inner products amounts to matrix multiplication, which is efficiently implementable on a GPU.

The primary difficulty in the calculation of these kernel elements is in ensuring that memory accesses are coalesced. If the training vectors are dense, then one natural approach would be to make the $16 \cdot i + j$th thread responsible for calculating the kernel inner product of the ith training vector with the jth working set element, by first copying the 16 elements of the working set into a separate buffer (so that they may be read out 16 elements at a time), and then having blocks of 256 threads cooperatively stage in 16×16 chunks of the training and working sets, before calculating their product. This approach would ensure coalesced memory accesses, and is popular in matrix-multiplication implementations.

As is illustrated in the left-hand side of Figure 1, if the training vectors are sparse, then this simple strategy fails, because the differing sparsity patterns of the training vectors cause memory accesses of the working set to be uncoalesced. Our solution, illustrated in the right-hand side of the same figure, is to cluster the training vectors by sparsity pattern. This causes blocks of sequentially-numbered threads to access training vectors with the same sparsity pattern,

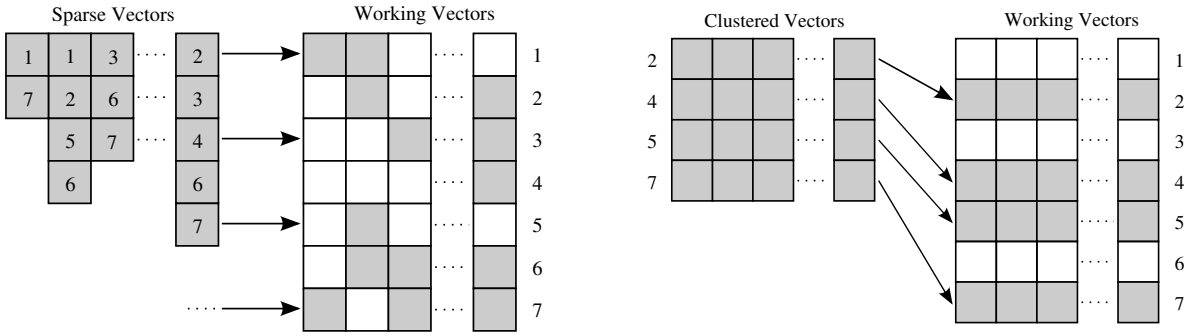

Figure 1: Illustration of memory-access patterns when the training data is sparse (left), or clustered by sparsity pattern (right). The "sparse vectors" are a set of vectors with different sparsity patterns, with the index of each nonzero element being the number contained within the corresponding box. The "clustered vectors" represent a sequence of consecutive training vectors which all have the same sparsity pattern, with the nonzero indices being the numbers to the left. The "working vectors" are dense copies of the 16 elements of the working set $\mathcal{I}$. Note that, in the right-hand diagram, the memory locations which must be accessed lie in the same rows, illustrating that they may be read as aligned 16-element chunks.

ensuring that their accesses to the working set will be coalesced. This comes at the cost of requiring that additional elements occasionally be introduced into the training vectors, in order to ensure that all elements of each cluster have the same sparsity pattern.

Algorithm 1 contains pseudocode for updating the "responses" $c_i = \sum_{j=1}^{n} \alpha_j y_j K(x_i, x_j)$ for all i, in response to a change in 16 αs. As before, the working set vectors are copied out of the training set into a temporary dense matrix, before the call to the update routine. Within the update, the kth training vector is assigned to the kth thread, which calculates the kernel inner products of this vector with all 16 elements of the working set. Each block of 256 threads repeatedly stages in 16×16 chunks of the working set. After each such chunk has been staged in, each thread steps through it, updating the inner products for which it is responsible. Note that in the key inner-loop of the psuedocode, only registers and shared memory are accessed, and these chunks of shared memory are staged in using coalesced reads, thanks to the clustering of the training data.

5.3 Clustering

Clustering by sparsity pattern will result in the greatest improvement in runtime if the clusters S_i are chosen to minimize:

$$\sum_i |\{j : \exists x \in S_i \, (x_j \neq 0)\}|$$

However, it would be counterproductive to perform an extremely high-quality clustering if the resulting savings in optimization runtime were to be less than the amount of time spent finding the clusters. Hence, we wish only to find a coarse clustering quickly. The solution which we propose is to find the clusters greedily, in a single pass over the training data, on the CPU. We define the nonzero elements of a cluster to be the union of the nonzero elements of the vectors assigned to that cluster. If we assign a new training vector x to cluster S, then the sparsity patterns of both S and x will be changed, via the addition of new elements, in order to bring them into correspondence. Algorithm 2 simply iterates over the training set, greedily assigning each x to the cluster which results in the insertion of the fewest new elements.

There is one additional parameter to this algorithm which remains to be explained: the maximum number of "active clusters" k, denoting the number of candidate clusters which will be considered for each training vector. If $k = \infty$, then all clusters are active from the start, resulting in a runtime which is quadratic in n. When k is smaller, a training vector can only be assigned to those clusters which are active at the time it is considered, which improves the runtime from $O(n^2)$ to $O(nk)$. Empirically, we have found that by decreasing k, clustering times can be improved dramatically, with little degradation in quality.

5.4 Classification

It is also important that it be possible to rapidly classify testing vectors using a learned classifier. Recall that we must calculate, for each testing vector x:

$$\text{sign}\left(b + \sum_{i=1}^{n} \alpha_i y_i K\left(x, x_i\right)\right) \tag{7}$$

Our classification routine is extremely similar to the update routine of Algorithm 1. The testing set is first broken up into pseudo-"working sets" of size 16. Kernel inner products are calculated in the same fashion as before, after which the sums in equation 7 are computed using a parallel sum-reduction. The key difference is that, during classification, we need only consider those elements of the training set which have nonzero coefficients α_i in equation 7. As a result, before classification, we "finalize" the learned classifier by discarding all of the training vectors with zero coefficients, and re-clustering the remaining vectors.

5.5 Other Approaches

The design choices which we have thus far described have the side-effect of making it difficult for our algorithm to incorporate certain optimizations which are popular in other implementations.

Most SVM optimizers keep a "cache" of recently-computed rows of the kernel matrix, based on the intuition that the same elements of the training set will be repeatedly chosen by the heuristic, particularly when the algorithm is close to convergence. If the kernel matrix rows corresponding to these elements are on-hand, then significant computational effort may be saved. On the GPU, keeping a cache of kernel matrix rows comes at a high cost. As it stands, our algorithm never explicitly stores individual elements of the kernel matrix, and global memory accesses are sufficiently expensive that the performance impact of doing so would be non-negligible. This cost might be justified in some cases, particularly on those datasets containing a small number of training examples, for which cache elements are likely to be accessed frequently. Our GPU implementation, however, is deliberately targeted towards datasets containing

Algorithm 1 GPU algorithm which, for the binary classification objective, updates the responses $c_i = \sum_{j=1}^{n} \alpha_j y_j K(x_i, x_j)$ based on changes of the dual variables α_i to $\alpha_i + \Delta_i$ for $i \in \mathcal{I}$, where $\mathcal{I}$ is the working set. This pseudocode should be read as if it were executing in lockstep on all threads, although in practice synchronization must be performed in order to guarantee that the threads within each block do not fall out of sync. The training vectors for the working set are stored (as dense vectors) in u, while the nonzeros of the current cluster are stored in x, with $\mathcal{J}$ being the indices of these nonzeros.

```
// Thread t ∈ {0, ..., 255} working on cluster C
input u[i][j]: ith element of jth vector of working set
input y[i]: label of ith vector of working set
input Δ[i]: change in α for ith vector of working set
input J[i]: index of ith nonzero of C
input x[i][j]: ith nonzero of jth vector of C
in/out c[i]: response for ith vector of C
// Calculate inner products a[i] = ⟨x[···][t], u[···][i]⟩
let a[i] := 0 for i ∈ {0, ..., 15}
for i = 0 to |J| step 16
    let i' := min {i + 15, |J|}
    Stage u[J[i], ..., J[i']][0, ..., 15] into shared memory
    for j = i to i'
        Copy x[j][t] into a register
        for k = 0 to 15
            let a[k] := a[k] + x[j][t] · u[J[j]][k]
// Update classification c[t]
let b := 0
for i = 0 to 15
    let k := K(‖x[···][t]‖, ‖u[···][i]‖, a[i])
    let b := b + Δ[i] · y[i] · k
let c[t] := c[t] + b
```

Algorithm 2 Algorithm for greedily clustering a dataset by sparsity pattern. The algorithm passes once through the dataset, assigning each vector to the cluster to which its addition results in the introduction of the smallest number of nonzero elements. The number of "active clusters" is the total number of candidate clusters which are available at a given time. Choosing this number to be smaller than the total number of clusters dramatically improves runtime, at the cost of a very slight penalty to clustering performance.

```
input n: number of training vectors
input k: number of active clusters
input ℓ: maximum cluster size
input x[i][j]: jth element of ith training vector
output S[i]: ith cluster
let S[i] := ∅ for i ∈ {0, ..., ⌈n/ℓ⌉ − 1}
// a[i] will be 1 for "active" clusters, 0 otherwise
let m := max(k, ⌈n/ℓ⌉)
let a[i] := 1 for i ∈ {0, ..., m − 1}
let a[i] := 0 for i ∈ {m, ..., ⌈n/ℓ⌉ − 1}
for i = 1 to n in random order
    // calculate costs c[j] for active clusters j
    let J_x := {k : x[i][k] ≠ 0}
    for all j : a[j] = 1
        let J_S := {k : ∃x' ∈ S[j] (x'[k] ≠ 0)}
        let c[j] := |S_j| |J_x \ J_S| + |J_S \ J_x|
    // insert x[i] into the lowest-cost cluster
    let j' := argmin_{j:a[j]=1} c[j]
    let S[j'] := S[j'] ∪ {x[i]}
    // activate a new cluster, if necessary
    if |S[j']| = ℓ
      let a[j'] := 0
      if m < ⌈n/ℓ⌉
        let a[m] := 1
        let m := m + 1
```

a relatively large number of high dimensional sparse examples, for which it is unlikely that the use of a cache would prove to be cost-effective.

Bordes et al. [1] propose an elegant compromise between the use of a heuristic, and choosing working sets randomly. Their approach is to maintain a relatively large "active set" of training vectors, for which the responses c are kept up-to-date, and from which elements of the working set are selected using a heuristic. Less useful elements of the active set are periodically removed, and replaced with new elements, selected randomly from the complement of the active set. This reduces the number of elements of c which must be kept up-to-date, thus making both the heuristic, and the c-updates, less expensive. Unfortunately, it is difficult to reconcile this optimization with our clustering-based solution to handling sparsity, since it would be necessary to re-cluster the entire active set after every addition or removal of elements.

Another popular optimization is the use of a "shrinking" heuristic. This heuristic is based on the observation that it may be the case that, during optimization, some correctly-classified training examples will stray so far from the decision surface that they are unlikely to ultimately make any contribution to the optimal classifier. Hence, it is safe to remove these vectors, provided that a check is performed, once optimization completes, to ensure that these removed examples are still correctly-classified, and are outside the margin. Because our clustering-based approach makes it difficult to remove individual elements from the training set during optimization, we do not make use of this optimization.

6. PERFORMANCE COMPARISON

We compared the runtime of our GPU-based SVM training implementation to the fastest CPU-based implementations of which we are aware, as well as to a previous GPU-based implementation. We use these experiments to study the efficiency of specific aspects of our implementation. We tested all implementations on the same machine, which contains an Intel Core i7 920 CPU, and 12G of memory, as well as a pair of NVIDIA Tesla C1060 graphics cards with 4G of memory. The tested GPU implementations use only a single graphics card, so the other is left idle.

Our testing datasets are listed in Figure 1. The Adult and MNIST datasets were downloaded from Léon Bottou's LASVM web page[1]. Cov1 is the covertype-1 dataset of Blackard & Dean. TIMIT is a phonetically transcribed corpus of speech spoken by North American speakers. We use MFCC features extracted from every 10ms frame of a subset of this dataset, along with their first two derivatives, for framewise classification of the stop consonants. Both the MNIST and TIMIT datasets are multiclass, but we also used them in testing binary classification experiments by taking the digit 8 and phoneme `/k/` to be the positive classes, respectively, and the union of all other labels to be the negative class. The regularization and Gaussian kernel parameters for the Adult dataset are taken from Platt [11], while those for MNIST and Cov1 are taken from Bordes et al. [1], except that the regularization parameter on MNIST was

[1] `http://leon.bottou.org/projects/lasvm`

DATA SET	TRAINING SIZE	TESTING SIZE	DIMENSION	CLASSES	BINARY C	γ
ADULT	31562	16282	123	2	1	0.05
COV1	522911	58101	54	2	3	1
MNIST	60000	10000	768	10	1	0.02
TIMIT	63881	22257	39	6	1	0.025

Table 1: Dataset properties and parameters. The "binary C" column contains the regularization trade-off parameter used for binary classification on each dataset. For multiclass, we choose C to be half as large.

	CRAMMER & SINGER		OURS	
DATA SET	TIME	SPEEDUP	TIME	SPEEDUP
MNIST	40M	1×	25S	97×
TIMIT	26M	1×	13S	121×

Table 4: Performance comparison of our multiclass GPU implementation to the CPU implementation of Crammer and Singer [5]. As in Table 3, the reported runtimes do not include time spent during initialization (or clustering).

decreased from 1000 to 1, which we have observed to improve the rate of convergence, with nearly no impact on classification performance.

We instrumented all of the tested implementations to periodically output trace points containing the current program state and the total elapsed time. The time spent inside this instrumentation code was subtracted from this running total. We then used these traces to find the first time at which the relative duality gap, $2(p-d)/(p+d)$, where p is the primal objective value and d the dual, dropped below the threshold $\epsilon = 0.01$.

6.1 Clustering

We first investigate the efficiency of our simple clustering algorithm at creating "good" clusters, i.e. where the sparsity patterns of all data points in each cluster are similar. We investigate the effect of the parameter k controlling the number of active clusters under consideration (see Section 5.3) on both the runtime, and on the quality of the resulting clustering. Table 2 displays the time required to cluster each of the sparse datasets into clusters of size 256, using a varying number of active clusters (including keeping all clusters "active"). We see that by choosing a relatively small number of active clusters, we can achieve fast clustering, with nearly the same quality as we would have with all clusters active. Hence, our implementation defaults to using 64 active clusters.

We also see from Table 2 that the clustering is effective at creating sparse clusters, with the number of non-zeros per cluster being well below the overall dimension of the data. However, on the highly sparse Adult and MNIST data sets, the average number of non-zeros per cluster is still well above the average number of non-zeros per data vector, showing that we do access many extra elements, although far fewer then we would if the data were treated as dense, as it is in other GPU implementations. We have found that the exploitation of sparsity has a concrete impact on performance: on the three sparse datasets on which we experimented, it alone was responsible for roughly a $1.5\times$ speedup.

6.2 Training Runtime

In Table 3 we report the actual training runtimes, and speedups relative to the CPU implementation, for training binary classifiers using the various GPU and CPU implementations[2]. We compare to the following:

- LIBSVM [4], a CPU-based implementation using SMO with the second-order heuristic of Fan et al. [7]. This is, for these datasets, the fastest CPU-based implementation of which we are aware.
- GPUSVM [3], which is a GPU-based implementation of essentially the algorithm used by LIBSVM.

We can see that our method efficiently utilized the GPU, taking between one and two orders of magnitude less time then the best CPU implementation (this of course depends on the specific CPU and GPU used). Our method also outperforms the previous GPU implementation on all datasets, even the dense TIMIT dataset (on which our implementation does not enjoy the benefit of sparsity), and Cov1, which contains a very small number of nonzeros per vector, on average, causing constant per-vector costs (such as evaluating the heuristic) to have a greater impact on the performance than those which depend on the dimensionality of the data (such as kernel evaluations). Our implementation is particularly effective on relatively high-dimensional sparse data sets, such as MNIST, where more time is spent performing kernel evaluations. On lower dimensional data sets, the cost of the chunking-and-sorting procedure by which we select our working set (see Section 5.1) becomes more significant, and the gain relative to Catanzaro et al. is smaller. Even on datasets such as TIMIT and Cov1, the benefits of using larger working sets, and the better handling of sparsity, outweigh the cost of chunking-and-sorting, and we still observe performance gains over Catanzaro et al.

In Table 4, we report runtimes for training multiclass SVMs, comparing with Koby Crammer's CPU-based implementation [5]. We are not aware of any previously published GPU implementation for multiclass SVMs. We see a roughly 100-fold speedup over the CPU implementation.

Finally, we measured proportions of training runtime spent within various components of the optimization routine. The results are plotted in Figure 2. On all datasets except for Cov1, the "update" step takes the majority of the runtime (on Cov1 it takes roughly 43%). This update code utilizes the GPU very efficiently, as all memory accesses are coalesced, indicating that we cannot expect to improve our runtime much by more careful coding. The time spent choosing the working set, and in particular performing the

[2]We are aware of two other GPU SVM implementations. Carpenter [2] is algorithmically very similar to GPUSVM, and reports roughly the same speedups on the Adult, Cov1 and MNIST datasets, relative to LIBSVM, as we observed for GPUSVM (albeit with different values of the parameters C and γ). This is a Windows-only implementation, so we could not experiment with it on our test machine. The other, recently made available at `http://mklab.iti.gr/project/GPU-LIBSVM`, will be presented in the upcoming WIAMIS'11 conference, and is based on precomputing the full Gram matrix on the GPU. Such an approach is only applicable when there are few enough training examples that this matrix can be stored in memory.

Data set	Avg. NZ	16 Active		64 Active		Max. Active	
		Avg. NZ	Time	Avg. NZ	Time	Avg. NZ	Time
Adult	13.9	57.5	0.042s	48.6	0.13s	45.7	0.24s
Cov1	12.0	15.9	0.44s	12.8	1.3s	12.2	20s
MNIST	150	406	0.43s	357	1.5s	345	4.6s

Table 2: Time spent clustering each of the datasets of Table 1 into clusters of size 256, and the resulting average number of nonzeros per vector (see Table 1 for the dataset dimensions). TIMIT is a dense dataset, and is therefore not included.

Data set	LIBSVM		GPUSVM		Ours	
	Time	Speedup	Time	Speedup	Time	Speedup
Adult	61s	1×	4.5s	14×	1.2s	52×
MNIST	4.4m	1×	11s	23×	3.9s	68×
TIMIT	4.6m	1×	4.9s	56×	3.5s	78×
Cov1	4.5h	1×	11m	26×	7.2m	38×

Table 3: Performance comparison of our GPU implementation to the CPU implementation LIBSVM [4] and GPU implementation GPUSVM [3]. All three implementations used the termination criterion $2(p-d)/(p+d) < 0.01$, where p is the primal objective, and d the dual. The reported runtimes do not include time spent loading the dataset, during initialization, or (for our implementation) clustering the training set by sparsity pattern.

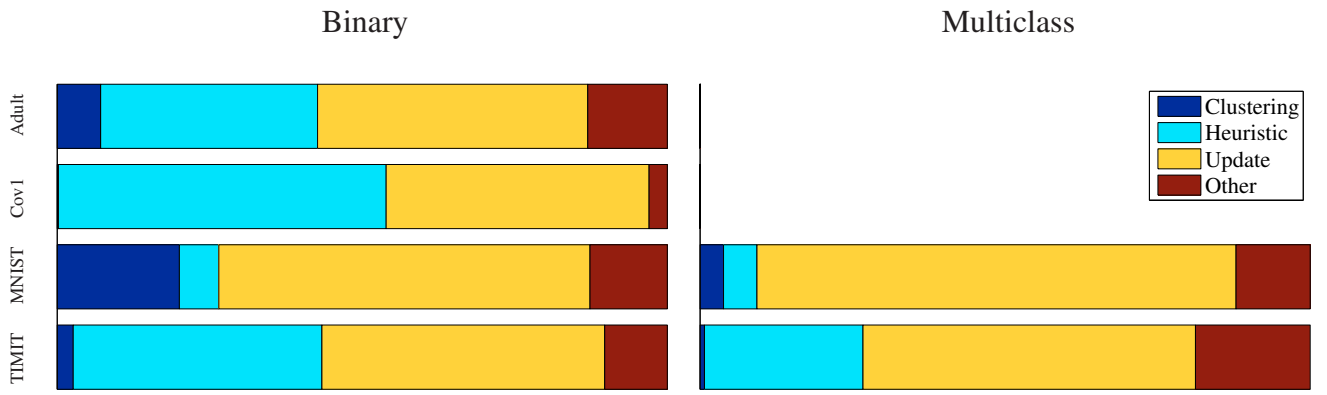

Figure 2: Illustration of the how runtime is distributed among the various parts of the optimization routine. Unlike in Tables 3 and 4, time spent clustering *is* included in these plots. The "heuristic" portion includes the time spent evaluating the heuristic and finding its maximum values. The "other" portion is mostly occupied by copying data between the CPU and GPU, optimizing subproblems on the CPU, and evaluating and testing the termination criterion.

chunking-and-sorting procedure, consumes a significant, but not dominant, portion of the runtime. This seems to be a good trade-off in terms of the time spent choosing the working set, versus the effectiveness of this choice. As discussed in Section 5, neither clustering the data by sparsity pattern, nor optimizing the subproblems (both performed on the CPU), require a significant amount of time.

7. SUMMARY

We presented a method for efficiently training binary and multiclass kernel SVMs using a GPU. We discussed various design considerations, which are often different than they would be for a serial CPU-based implementation, and also presented a novel approach for handling sparse data on the GPU. The result is an implementation of SVM training that is orders of magnitude faster then CPU implementations, and several times faster than previous GPU implementations. It is also the first GPU implementation we are aware of that handles multiclass classification.

Our implementation is freely available[3] and easy to set up and use. It includes not only a command-line interface, which is similar to those of other popular SVM solvers, but also C library and Matlab interfaces, all of which have full support for multiclass classification, sparse datasets, and all implemented kernels.

[3] `http://nagoya.uchicago.edu/~cotter/projects/gtsvm`

References

[1] A. Bordes, S. Ertekin, J. Weston, and L. Bottou. Fast kernel classifiers with online and active learning. *JMLR*, 6:1579–1619, September 2005.

[2] A. Carpenter. CUSVM: A CUDA implementation of support vector classification and regression. `http://patternsonascreen.net/cuSVM.html`, 2009.

[3] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML'08*, pages 104–111, 2008.

[4] 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`.

[5] K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *JMLR*, 2: 265–292, March 2002. ISSN 1532-4435.

[6] T.-N. Do, V.-H. Nguyen, and F. Poulet. Speed up SVM algorithm for massive classification tasks. In *ADMA'08*, pages 147–157, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] R.-E. Fan, P.-S. Chen, and C.-J. Lin. Working set selection using second order information for training support vector machines. *JMLR*, 6:1889–1918, 2005.

[8] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *ICML'08*, pages 408–415, 2008.

[9] T. Joachims. Making large-scale support vector machine learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.

[10] E. Osuna, R. Freund, and F. Girosi. Training support vector machines: an application to face detection. In *CVPR'97*, June 1997.

[11] J. C. Platt. Fast training of support vector machines using Sequential Minimal Optimization. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.

[12] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001. ISBN 0262194759.

[13] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *ICML'07*, pages 807–814, 2007.

[14] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. *Mathematical Programming*, pages 1–34, October 2010.