Fast Support Vector Machine Training and Classification on Graphics Processors

Bryan Catanzaro CATANZAR@EECS.BERKELEY.EDU
Narayanan Sundaram NARAYANS@EECS.BERKELEY.EDU
Kurt Keutzer KEUTZER@EECS.BERKELEY.EDU
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, USA

Abstract

Recent developments in programmable, highly parallel Graphics Processing Units (GPUs) have enabled high performance implementations of machine learning algorithms. We describe a solver for Support Vector Machine training running on a GPU, using the Sequential Minimal Optimization algorithm and an adaptive first and second order working set selection heuristic, which achieves speedups of 9-35× over LIBSVM running on a traditional processor. We also present a GPU-based system for SVM classification which achieves speedups of 81-138× over LIBSVM (5-24× over our own CPU based SVM classifier).

1. Introduction

Driven by the capabilities and limitations of modern semiconductor manufacturing, the computing industry is currently undergoing a massive shift towards parallel computing (Asanović et al., 2006). This shift brings dramatically enhanced performance to those algorithms which can be adapted to parallel computers.

One set of such algorithms are those used to implement Support Vector Machines (Cortes & Vapnik, 1995). Thanks to their robust generalization performance, SVMs have found use in diverse classification tasks, such as image recognition, bioinformatics, and text processing. Yet, training Support Vector Machines and using them for classification remains very computationally intensive. Much research has been done to accelerate training time, such as Osuna's decomposition approach (Osuna et al., 1997), Platt's Sequential Minimal Optimization (SMO) algorithm (Platt, 1999), Joachims' SVM^{light} (Joachims, 1999), which introduced shrinking and kernel caching, , and the working set selection heuristics used by LIBSVM (Fan et al., 2005). Despite this research, SVM training time is still significant for large training sets.

In this paper, we show how Support Vector Machine training and classification can be adapted to a highly parallel, yet widely available and affordable computing platform: the graphics processor, or more specifically, the Nvidia GeForce 8800 GTX, and detail the performance gains achieved.

The organization of the paper is as follows. Section 2 describes the SVM training and classification problems briefly. Section 3 gives an overview of the architectural and programming features of the GPU. Section 4 presents the details of implementation of the parallel SMO approach on the GPU. Section 5 explains the implementation details of the SVM classification problem. We present our results in Section 6 and conclude in Section 7.

2. Support Vector Machines

We consider the standard two-class soft-margin SVM classification problem (C-SVM), which classifies a given data point $x \in \mathbb{R}^n$ by assigning a label $y \in \{-1, 1\}$.

2.1. SVM Training

Given a labeled training set consisting of a set of data points $x_i, i \in \{1, ..., l\}$ with their accompanying labels $y_i, i \in \{1, ..., l\}$, the SVM training problem can be written as the following Quadratic Program, where α_i is a set of weights, one for each training point, which are being optimized to determine the SVM classifier, C is a parameter which trades classifier generality for accuracy on the training set, and $Q_{ij} = y_i y_j \Phi(x_i, x_j)$,

Appearing in *Proceedings of the 25th International Conference on Machine Learning*, Helsinki, Finland, 2008.

where $\Phi(x_i, x_j)$ is a kernel function.

$$\begin{aligned} \max_{\alpha} \; & F(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\alpha^T Q \alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \; \forall i \in 1 \ldots l \\ & y^T \alpha = 0 \end{aligned} \tag{1}$$

We consider the standard kernel functions shown in table 1.

Table 1. Standard Kernel Functions

LINEAR	$\Phi(x_i, x_j) = x_i \cdot x_j$
POLYNOMIAL	$\Phi(x_i, x_j; a, r, d) = (a x_i \cdot x_j + r)^d$
GAUSSIAN	$\Phi(x_i, x_j; \gamma) = \exp\{-\gamma \|x_i - x_j\|^2\}$
SIGMOID	$\Phi(x_i, x_j; a, r) = \tanh(a x_i \cdot x_j + r)$

2.1.1. SMO Algorithm

The SVM Training problem can be solved by many methods, each with different parallelism implications. We have implemented the Sequential Minimal Optimization algorithm (Platt, 1999), with a hybrid working set selection heuristic making use of the first order heuristic proposed by (Keerthi et al., 2001) as well as the second order heuristic proposed by (Fan et al., 2005).

The SMO algorithm is a specialized optimization approach for the SVM quadratic program. It takes advantage of the sparse nature of the support vector problem and the simple nature of the constraints in the SVM QP to reduce each optimization step to its minimum form: updating two α_i weights. The bulk of the computation is then to update the Karush-Kuhn-Tucker optimality conditions for the remaining set of weights and then find the next two weights to update in the next iteration. This is repeated until convergence. We state this algorithm briefly, for reference purposes.

Algorithm 1 Sequential Minimal Optimization

Input: training data x_i, labels y_i, $\forall i \in \{1..l\}$
Initialize: $\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1..l\}$,
Initialize: b_{high}, b_{low}, i_{high}, i_{low}
Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
repeat
  Update f_i, $\forall i \in \{1..l\}$
  Compute: b_{high}, i_{high}, b_{low}, i_{low}
  Update $\alpha_{i_{high}}$ and $\alpha_{i_{low}}$
until $b_{low} \leq b_{high} + 2\tau$

For the first iteration, we initialize $b_{high} = -1$, $i_{high} = \min\{i : y_i = 1\}$, $b_{low} = 1$, and $i_{low} = \min\{i : y_i = -1\}$.

During each iteration, once we have chosen i_{high} and i_{low}, we take the optimization step:

$$\alpha'_{i_{low}} = \alpha_{i_{low}} + y_{i_{low}}(b_{high} - b_{low})/\eta \tag{2}$$

$$\alpha'_{i_{high}} = \alpha_{i_{high}} + y_{i_{low}} y_{i_{high}} (\alpha_{i_{low}} - \alpha'_{i_{low}}) \tag{3}$$

where $\eta = \Phi(x_{i_{high}}, x_{i_{high}}) + \Phi(x_{i_{low}}, x_{i_{low}}) - 2\Phi(x_{i_{high}}, x_{i_{low}})$. To ensure that this update is feasible, $\alpha'_{i_{low}}$ and $\alpha'_{i_{high}}$ must be clipped to the valid range $0 \leq \alpha_i \leq C$.

The optimality conditions can be tracked through the vector $f_i = \sum_{j=1}^{l} \alpha_j y_j \Phi(x_i, x_j) - y_i$, which is constructed iteratively as the algorithm progresses. After each α update, f is updated for all points. This is one of the major computational steps of the algorithm, and is done as follows:

$$\begin{aligned} f'_i = f_i &+ (\alpha'_{i_{high}} - \alpha_{i_{high}}) y_{i_{high}} \Phi(x_{i_{high}}, x_i) \\ &+ (\alpha'_{i_{low}} - \alpha_{i_{low}}) y_{i_{low}} \Phi(x_{i_{low}}, x_i) \end{aligned} \tag{4}$$

In order to evaluate the optimality conditions, we define index sets:

$$\begin{aligned} I_{high} = &\{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = 0\} \\ &\cup \{i : y_i < 0, \alpha_i = C\} \end{aligned} \tag{5}$$

$$\begin{aligned} I_{low} = &\{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = C\} \\ &\cup \{i : y_i < 0, \alpha_i = 0\} \end{aligned} \tag{6}$$

Because of the approximate nature of the solution process, these index sets are computed to within a tolerance ϵ, e.g. $\{i : \epsilon < \alpha_i < (C - \epsilon)\}$.

We can then measure the optimality of our current solution by checking the optimality gap, which is the difference between $b_{high} = \min\{f_i : i \in I_{high}\}$, and $b_{low} = \max\{f_i : i \in I_{low}\}$. When $b_{low} \leq b_{high} + 2\tau$, we terminate the algorithm.

2.1.2. Working set selection

During each iteration, we need to choose i_{high} and i_{low}, which index the α weights which will be changed in the following optimization step. The first order heuristic from (Keerthi et al., 2001) chooses them as follows:

$$i_{high} = \arg\min\{f_i : i \in I_{high}\} \tag{7}$$

$$i_{low} = \arg\max\{f_i : i \in I_{low}\} \tag{8}$$

The second order heuristic from (Fan et al., 2005) chooses i_{high} and i_{low} to optimize the unconstrained SVM functional. An optimal approach to this problem would require examining $\binom{l}{2}$ candidate pairs, which would be computationally intractable. To simplify the problem, i_{high} is instead chosen as in the first order

heuristic, and then i_{low} is chosen to maximally improve the objective function while still guaranteeing progress towards the constrained optimum from problem (1). More explicitly:

$$i_{high} = \arg\min\{f_i : i \in I_{high}\} \quad (9)$$

$$i_{low} = \arg\max\{\Delta F_i(\alpha) : i \in I_{low}, f_{i_{high}} < f_i\} \quad (10)$$

After choosing i_{high}, we compute for all $i \in \{1..l\}$

$$\beta_i = f_{i_{high}} - f_i \quad (11)$$

$$\eta_i = \Phi(x_{i_{high}}, x_{i_{high}}) + \Phi(x_i, x_i) - 2\Phi(x_{i_{high}}, x_i) \quad (12)$$

$$\Delta F_i(\alpha) = \beta_i^2/\eta_i \quad (13)$$

We then find the maximum ΔF_i over all valid points ($i \in I_{low}$) for which we are guaranteed to progress towards the constrained optimum ($f_{i_{high}} < f_i$).

2.1.3. Adaptive heuristic

The second order heuristic utilizes more information from the SVM training problem, and so it generally reduces the number of iterations necessary during the solution process. However, it is more costly to compute. In our GPU implementation, the geometric mean of iteration time over our benchmark set using the second order heuristic increased by 1.9× compared to the first order heuristic. On some benchmarks, the total number of iterations decreased sufficiently to provide a significant speedup overall, but on others, the second order heuristic is counterproductive for our GPU implementation.

To overcome this problem, we implemented an adaptive heuristic that chooses between the two selection heuristics dynamically, with no input or tuning from the user. The adaptive heuristic periodically samples progress towards convergence as a function of wall-clock time using both heuristics, then chooses the more productive heuristic.

This sampling occurs every $l/10$ iterations, and during each sample, the heuristic under test is executed for two phases of 64 iterations each. The average optimality gap in each of these phases is computed, and then the rate of progress is estimated by dividing the change in the optimality gap over the two phases by the time it has taken to execute them. The same sampling process is then performed with the other heuristic, and the best heuristic is then used until the next sampling period.

2.2. SVM Classification

The SVM classification problem is as follows: for each data point z which should be classified, compute

$$\hat{z} = \operatorname{sgn}\left\{b + \sum_{i=1}^{l} y_i \alpha_i \Phi(x_i, z)\right\} \quad (14)$$

where $z \in \mathbb{R}^n$ is a point which needs to be classified, and all other variables remain as previously defined.

From the classification problem definition, it follows immediately that the decision surface is defined by referencing a subset of the training data, or more specifically, those training data points for which the corresponding $\alpha_i > 0$. Such points are called support vectors.

Generally, we classify not just one point, but a set of points. We exploit this for better performance, as explained in Section 5.

3. Graphics Processors

Graphics processors are currently transitioning from their initial role as specialized accelerators for triangle rasterization to general purpose engines for high throughput floating-point computation. Because they still service the large gaming industry, they are ubiquitous and relatively inexpensive.

GPU architectures are specialized for compute-intensive, memory-intensive, highly parallel computation, and therefore are designed such that more resources are devoted to data processing than caching or control flow. State of the art GPUs provide up to an order of magnitude more peak IEEE single-precision floating-point than their CPU counterparts. Additionally, GPUs have much more aggressive memory subsystems, typically endowed with more than 10x higher memory bandwidth than a CPU. Peak performance is usually impossible to achieve on general purpose applications, yet capturing even a fraction of peak performance yields significant speedup.

GPU performance is dependent on finding high degrees of parallelism: a typical computation running on the GPU must express thousands of threads in order to effectively use the hardware capabilities. As such, we consider it an example of future "many-core" processing (Asanović et al., 2006). Algorithms for machine learning applications will need to consider such parallelism in order to utilize many-core processors. Applications which do not express parallelism will not continue improving their performance when run on newer computing platforms at the rates we have enjoyed in the past. Therefore, finding large scale par-

allelism is important for compute performance in the future. Programming for GPUs is then indicative of the future many-core programming experience.

3.1. Nvidia GeForce 8800 GTX

In this project, we employ the NVIDIA GeForce 8800 GTX GPU, which is an instance of the G80 GPU architecture, and is a standard GPU widely available on the market. Pertinent facts about the GPU platform can be found in table 2. We refer the reader to the Nvidia CUDA reference manual for more details (Nvidia, 2007).

Table 2. Nvidia GeForce 8800 GTX Characteristics

# of stream processors	128
Peak general purpose IEEE SP	346 GFlops
Multiprocessor local store size	16 kB
Clock rate	1.35 GHz
Memory capacity	768 MB
Memory bandwidth	86.4 GB/s
CPU⟷GPU bandwidth	3.2 Gbit/s

3.2. CUDA

Nvidia provides a programming environment for its GPUs called the Compute Unified Device Architecture (CUDA). The user codes in annotated C++, accelerating compute intensive portions of the application by executing them on the GPU.

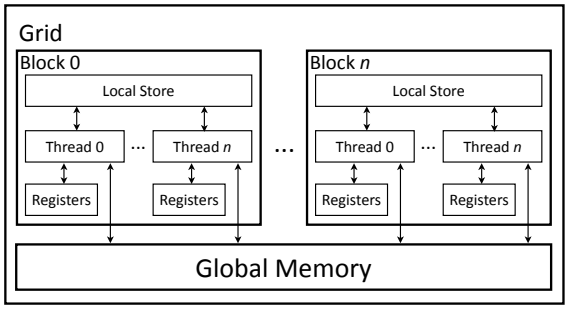

Figure 1. Logical organization of the GeForce 8800

Figure 1 illustrates how the GPU appears to the programmer. The programmer organizes the computation into grids, which are organized as a set of thread blocks. The grids run sequentially on the GPU, meaning that all computation in the grid must finish before another grid is invoked. As mentioned, grids contain thread blocks, which are batches of threads that execute together, sharing local memories and synchronizing at programmer specified barriers. A maximum of 512 threads can comprise a thread block, which puts a limit on the scope of synchronization and communication in the computation. However, enormous numbers of blocks can be launched in parallel in the grid, so that the total number of threads that can be launched in parallel is very high. In practice, we need a large number of thread blocks to ensure that the compute power of the GPU is efficiently utilized.

4. SVM Training Implementation

Since GPUs need a large number of threads to efficiently exploit parallelism, we create one thread for every data point in the training set. For the first phase of the computation, each thread computes f_i' from equation (4). We then apply a working set selection heuristic to select the next points which will be optimized. The details are explained in the following section.

4.1. Map Reduce

At least since the LISP programming language, programmers have been mapping independent computations onto partitioned data sets, using reduce operations to summarize the results. Recently, Google proposed a Map Reduce variant for processing large datasets on compute clusters (Dean & Ghemawat, 2004). This algorithmic pattern is very useful for extracting parallelism, since it is simple to understand, and maps well to parallel hardware, given the inherent parallelism in the map stage of the computation.

The Map Reduce pattern has been shown to be useful for many machine learning applications (Chu et al., 2007), and is a natural fit for our SVM training algorithm. For the first order heuristic, the computation of f_i' for all points is the map function, and the search for b_{low}, b_{high}, i_{low} and i_{high} is the reduction operation. For the second order heuristic, there are two Map Reduce stages: one to compute f_i', b_{high} and i_{high}, and another where the map stage computes ΔF_i for all points, while the reduce stage computes b_{low} and i_{low}.

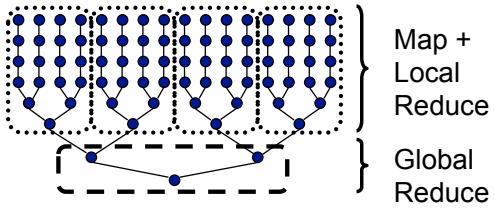

Figure 2. Structuring the Map Reduce

Because the CUDA programming model has strict lim-

itations on synchronization and communication between thread blocks, we organize the reductions in two phases, as shown in figure 2. The first phase does the map computation, as well as a local reduce within a thread block. The second phase finishes the global reduction. Each phase of this process is implemented as a separate call to the GPU.

4.2. Implementation Details

4.2.1. Caching

Since evaluating the kernel function $\Phi(\cdot)$ is the dominant part of the computation, it is useful to cache as much as possible from the matrix of kernel function evaluations $K_{ij} = \Phi(x_i, x_j)$ (Joachims, 1999). We compute rows of this matrix on the fly, as needed by the algorithm, and cache them in the available memory on the GPU.

When updating the vector f, we need access to two rows of K, since we have changed exactly two entries in α. In our system, the CPU checks to see which of these two rows, if any, are present in the cache. If a row is not present, the CPU voids the least recently used row of the cache, and assigns it to the new row which is needed. For the rows which hit in the cache, the GPU avoids doing the kernel evaluations. Otherwise, the GPU writes out the appropriate row or rows after computing the kernel values. When using the second order heuristic, the computation of ΔF references the row of K corresponding to i_{high}, which guarantees that the next update of f will have a cache hit for its access to the same row.

4.2.2. Data Movement

Programming the GPU requires manually copying data from the host computer to the GPU and vice versa, and it also requires manually copying data from the GPU's global memory to the fast local stores. As mentioned previously, if the cache does not contain a particular row of K corresponding to the point x_j, that row will need to be generated, which means that we need to compute $\Phi(x_i, x_j)\ \forall i \in 1..l$. Since the vector x_j is shared between all computations, we load it into the GPU's local store. This is key to performance, since accessing the local store is orders of magnitude faster than accessing the global memory.

4.3. Related Work

There have been previous attempts to parallelize the SVM training problem. The most similar to ours is (Cao et al., 2006), which parallelizes the SMO algorithm on a cluster of computers using MPI. Both our approach and their approach use the concurrency inherent in the KKT condition updates as the major source of parallelism. However, in terms of implementation, GPUs present a completely different model than clusters, and hence the amount of parallelism exploited, such as the number of threads, granularity of computation per thread, memory access patterns, and data partitioning are very different. We also implement more sophisticated working set selection heuristics.

Many other approaches for parallelizing SVM training have been presented. The cascade SVM (Graf et al., 2005) is another proposed method for parallelizing SVM training on clusters. It uses a method of divide and conquer to solve large SVM problems. (Zanni et al., 2006) parallelize the underlying QP solver using Parallel Gradient Projection Technique. Work has been done on using a parallel Interior Point Method for solving the SVM training problem (Wu et al., 2006). (Collobert et al., 2002) proposes a method where the several smaller SVMs are trained in a parallel fashion and their outputs weighted using a Artificial Neural Network. (Ferreira et al., 2006) implement a gradient based solution for SVM training, which relies on data parallelism in computing the gradient of the objective function for an unconstrained QP optimization at its core. Some of these techniques, for example, the training set decomposition approaches like the Cascade SVM are orthogonal to the work we describe, and could be applied to our solver. (Bottou et al., 2007) give an extensive overview of parallel SVM implementations. We implemented the parallel SMO training algorithm because of its relative simplicity, yet high performance and robust convergence characteristics.

5. SVM Classification Implementation

We approached the SVM classification problem by making use of Map Reduce computations as well as vendor supplied Basic Linear Algebra Subroutines - specifically, the Matrix Matrix Multiplication routine (SGEMM), which calculates $C' = \alpha AB + \beta C$, for matrices A, B, and C and scalars α and β. For the Linear, Polynomial, and Sigmoid kernels, calculating the classification value involves finding the dot product between all test points and the support vectors, which is done through SGEMM. For the Gaussian kernel, we use the simple identity $||x - y||^2 = x \cdot x + y \cdot y - 2x \cdot y$ to recast the computation into a Matrix Matrix multiplication, where the SGEMM computes $D_{ij} = -\gamma||z_i - x_j||^2 = 2\gamma(z_i \cdot x_j) - \gamma(z_i \cdot z_i + x_j \cdot x_j)$, for a set of unknown points z and a set of support vectors x. We then apply a map reduce computation to

combine the computed D values to get the final result.

Continuing the Gaussian example, the map function exponentiates D_{ij} element wise, multiplies each column of the resulting matrix by the appropriate $y_j\alpha_j$. The reduce function sums the rows of the matrix and adds b to obtain the final classification for each data point as given by equation (14). Other kernels require similar Map Reduce calculations to finish the classification.

6. Results

The SMO implementation on the GPU is compared with LIBSVM, as LIBSVM uses Sequential Minimal Optimization for SVM training. We used the Gaussian kernel in all of our experiments, since it is widely employed.

6.1. Training

We tested the performance of our GPU implementation versus LIBSVM on the datasets detailed in tables 3 and 4.

Table 3. Datasets - References and training parameters

Dataset	C	γ
Adult (Asuncion & Newman, 2007)	100	0.5
Web (Platt, 1999)	64	7.8125
MNIST (LeCun et al., 1998)	10	0.125
USPS (Hull, 1994)	10	2^{-8}
Forest (Asuncion & Newman, 2007)	10	0.125
Face (Rowley et al., 1998)	10	0.125

Table 4. Dataset Size

Dataset	# Points	# Dimensions
Adult	32,561	123
Web	49,749	300
MNIST	60,000	784
USPS	7,291	256
Forest	561,012	54
Face	6,977	381

The sizes of the datasets are given in table 4. References for the datasets used and the (C, γ) values used for SVM training are provided in table 3.

We ran LIBSVM on an Intel Core 2 Duo 2.66 GHz processor, and gave LIBSVM a cache size of 650 MB, which is larger than our GPU implementation was allowed. CPU-GPU communication overhead was included in the solver runtime, but file I/O time was excluded for both our solver and LIBSVM. Table 5 shows results from our solver. File I/O varies from 1.2 seconds for USPS to about 12 seconds for Forest dataset. The CPU - GPU data transfer overhead was also very low. The time taken to transfer the training data to the GPU and copy the results back was less than 0.6 seconds, even for our largest dataset (Forest).

Since any two solvers give slightly different answers on the same optimization problem, due to the inexact nature of the optimization process, we show the number of support vectors returned by the two solvers as well as how close the final values of b were for the GPU solver and LIBSVM, which were both run with the same tolerance value $\tau = 0.001$. As shown in the table, the deviation in number of support vectors between the two solvers is less than 2%, and the deviation in the offset b is always less than 0.1%. Our solver provides equivalent accuracy to the LIBSVM solver, which will be shown again in the classification results section.

Table 5. SVM Training Convergence Comparison

Dataset	Number of SVs		Difference in b (%)
	GPU Adaptive	LIBSVM	
Adult	18,674	19,058	-0.004
Web	35,220	35,232	-0.01
MNIST	43,730	43,756	-0.04
USPS	684	684	0.07
Forest	270,351	270,311	0.07
Face	3,313	3,322	0.01

Table 6 contains performance results for the two solvers. We see speedups in all cases from 9× to 35×. For reference, we have shown results for the solvers using both heuristics statically. Examining the data shows that the adaptive heuristic performs robustly, surpassing or coming close to the performance of the best static heuristic on all benchmarks.

6.2. Classification

Results for our classifier are presented in table 8. We achieve $81 - 138\times$ speedup over LibSVM on the datasets shown. As with the solver, file I/O times were excluded from overall runtime. File I/O times vary from 0.4 seconds for Adult dataset to about 6 seconds for MNIST dataset.

6.2.1. Optimizations to CPU based classifier

LIBSVM classifies data points serially. This effectively precludes data locality optimizations and produces significant slowdown. It also represents data in a sparse format, which can cause overhead as well.

To optimize the CPU classifier, we performed the following:

1. We changed the data structure used for storing

Table 6. SVM Training Results

Dataset	GPU 1st Order		GPU 2nd Order		GPU Adaptive		LIBSVM		Speedup (×) (Adaptive)
	Iter.	Time (s)	Iter.	Time (s)	Iter.	Time (s)	Iter.	Time (s)	
Adult	114,985	30.15	40,044	30.46	64,446	**26.92**	43,735	550.2	20.4
Web	79,749	174.17	81,498	290.23	70,686	**163.89**	85,299	2422.46	14.8
MNIST	68,055	**475.42**	67,731	864.46	68,113	483.07	76,385	16965.79	35.1
USPS	6,949	0.596	3,730	**0.546**	4,734	0.576	4,614	5.092	8.8
Forest	2,070,867	4571.17	236,601	**1441.08**	450,506	2023.24	275,516	66523.53	32.9
Face	6,044	**1.30**	4,876	**1.30**	5,535	1.32	5,342	27.61	20.8

the support vectors and test vectors from a sparse indexed set to a dense matrix.

2. To maximize performance, we used BLAS routines from the Intel Math Kernel Library to perform operations similar to those mentioned in Section 5.

3. Wherever possible, loops were parallelized (2-way for the dual-core machine) using OpenMP.

These optimizations improved the classification speed on the CPU by a factor of $3.4 - 28.3\times$. The speedup numbers for the different datasets are shown in table 8. It should be noted that the GPU version is better than the optimized CPU versions by a factor of $4.9 - 23.9\times$.

For some insight into these results, we note that the optimized CPU classifier performs best on problems with a large number of input space dimensions, which helps make the SVM classification process compute bound. For problems with a small number of input space dimensions, the SVM classification process is memory bound, meaning it is limited by memory bandwidth. Since the GPU has much higher memory bandwidth, as noted in section 3, it is even more attractive for such problems.

We tested the combined SVM training and classification process for accuracy by using the SVM classifier produced by the GPU solver with the GPU classification routine, and used the SVM classifier provided by LIBSVM's solver to perform classification with LIBSVM. Thus, the accuracy of the classification results presented in table 7 reflect the overall accuracy of the GPU solver and GPU classifier system. The results are identical, which shows that our GPU based SVM system is as accurate as traditional CPU based methods.

Table 7. Accuracy of GPU SVM classification vs. LIBSVM

Dataset	GPU Accuracy	LIBSVM Accuracy
Adult	6619/8000	6619/8000
Web	3920/4000	3920/4000
MNIST	2400/2500	2400/2500
USPS	1948/2007	1948/2007
Face	23665/24045	23665/24045

7. Conclusion

This work has demonstrated the utility of graphics processors for SVM classification and training. Training time is reduced by $9 - 35\times$, and classification time is reduced by $81 - 138\times$ compared to LIBSVM, or $5 - 24\times$ over our own CPU based SVM classifier. These kinds of performance improvements can change the scope of SVM problems which are routinely solved, increasing the applicability of SVMs to difficult classification problems. For example, training a classifier for an input data set with almost 600000 data points and 50 dimensions takes only 34 minutes on the GPU, compared with over 18 hours on the CPU.

The GPU is a very low cost way to achieve such high performance: the GeForce 8800 GTX fits into any modern desktop machine, and currently costs $300. Problems which used to require a compute cluster can now be solved on one's own desktop. New machine learning algorithms that can take advantage of this kind of performance, by expressing parallelism widely, will provide compelling benefits on future many-core platforms.

Acknowledgements

The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Bryan Catanzaro is also supported by a National Science Foundation Graduate Research Fellowship. The authors thank the anonymous reviewers for their com-

Table 8. Performance of GPU SVM classifier compared to LIBSVM and Optimized CPU classifier

Dataset	LibSVM	CPU Optimized classifier		GPU Classifier		
	Time (s)	Time (s)	Speedup (×) compared to LIBSVM	Time (s)	Speedup (×) compared to LibSVM	Speedup (×) compared to CPU optimized code
Adult	61.307	7.476	8.2	0.575	106.6	13.0
Web	106.835	15.733	6.8	1.063	100.5	14.8
MNIST	269.880	9.522	28.3	1.951	138.3	4.9
USPS	0.777	0.229	3.4	0.00958	81.1	23.9
Face	88.835	5.191	17.1	0.705	126.0	7.4

ments and suggestions.

References

Asanović, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., & Yelick, K. A. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley* (Technical Report UCB/EECS-2006-183). EECS Department, University of California, Berkeley.

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

Bottou, L., Chapelle, O., DeCoste, D., & Weston, J. (2007). *Large-scale kernel machines*. The MIT Press.

Cao, L., Keerthi, S., Ong, C.-J., Zhang, J., Periyathamby, U., Fu, X. J., & Lee, H. (2006). Parallel sequential minimal optimization for the training of support vector machines. *IEEE Transactions on Neural Networks*, *17*, 1039–1049.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y., & Olukotun, K. (2007). Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt and T. Hoffman (Eds.), *Advances in neural information processing systems 19*, 281–288. Cambridge, MA: MIT Press.

Collobert, R., Bengio, S., & Bengio, Y. (2002). A parallel mixture of svms for very large scale problems. *Neural Computation*, *14*, 1105–1114.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Mach. Learn.*, *20*, 273–297.

Dean, J., & Ghemawat, S. (2004). Mapreduce: simplified data processing on large clusters. *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association.

Fan, R.-E., Chen, P.-H., & Lin, C.-J. (2005). Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, *6*, 1889–1918.

Ferreira, L. V., Kaskurewicz, E., & Bhaya, A. (2006). Parallel implementation of gradient-based neural networks for svm training. *International Joint Conference on Neural Networks*.

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.

Hull, J. J. (1994). A database for handwritten text recognition research. *IEEE Trans. Pattern Anal. Mach. Intell.*, *16*, 550–554.

Joachims, T. (1999). Making large-scale support vector machine learning practical. In *Advances in kernel methods: support vector learning*. Cambridge, MA, USA: MIT Press.

Keerthi, S. S., Shevade, S. K., Bhattacharyya, C., & Murthy, K. R. K. (2001). Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Comput.*, *13*, 637–649.

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

Nvidia (2007). Nvidia CUDA. `http://nvidia.com/cuda`.

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.

Platt, J. C. (1999). Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, 185–208. Cambridge, MA, USA: MIT Press.

Rowley, H. A., Baluja, S., & Kanade, T. (1998). Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *20*, 23–38.

Wu, G., Chang, E., Chen, Y. K., & Hughes, C. (2006). Incremental approximate matrix factorization for speeding up support vector machines. *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 760–766). New York, NY, USA: ACM Press.

Zanni, L., Serafini, T., & Zanghirati, G. (2006). Parallel software for training large scale support vector machines on multiprocessor systems. *J. Mach. Learn. Res.*, *7*, 1467–1492.