Accelerating Machine Learning Applications on Graphics Processors

Narayanan Sundaram & Bryan Catanzaro
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA, USA
{narayans,catanzar}@eecs.berkeley.edu

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 Platt's 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 63-133× over LIBSVM.

1 Introduction

Computing industry is facing its most massive challenge in recent years due to the transition from single-processor CPUs to manycore CPUs. This change has been facilitated due to increasing transistor density on silicon dictated by Moore's law and dwindling benefits from increasing uniprocessor complexity. Also, managing power and complexity pushes us into multi-processor-on-chip designs. To build successful, scalable and reliable software in this model is a challenge [1].

Machine Learning algorithms have widespread use in information retrieval, bioinformatics, speech and image recognition systems etc. These tasks are important future workloads as identified by Intel in their categorization[2]. Machine Learning is a dominant field in the Recognition-Mining-Synthesis (RMS) categorization. Machine learning is an area that can benefit from the increased computational resources available due to parallelism.

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. Interestingly, GPUs are massively parallel computers whose use in the past (for general purpose computation) had been hampered by the difficulty in programming them. With the recent trend towards more programmable GPUs, they seem to be on a collision course with manycore CPU architectures. We believe that the successes and failures in porting applications for GPUs will help manycore CPU architects and application developers too.

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 gives an overview of the architectural and programming features of the GPU. Section 3 describes the SVM training and classification problems briefly. Section 4 presents the details of implementation of the parallel SVM training and classification approach on the GPU. Results are presented in Section 5. Section 6 related work in SVM parallelization and using GPUs for general purpose computations. We conclude in Section 7.

2 GPU Architecture

GPU architectures are specialized for compute-intensive, memory-intensive, highly-parallel compu-

tation, 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 [1]. 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 parallelism is important for compute performance in the future. Programming for GPUs is then indicative of the future many-core programming experience. Since the individual processors in the GPU are very simple (no branch prediction, caches, prefetch mechanisms etc.), the only way to get performance is through parallelism.

2.1 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 1. We refer the reader to the Nvidia CUDA reference manual for more details [3].

The GPU has a local store (referred to as shared memory), which can act as a software managed cache. The GPU is essentially a SIMD machine, and executes 32 threads (called a warp) with one instruction issue. Since instruction issue does not happen at less than warp granularity, any conditional statement (like if-then-else) executes both paths if any of the threads involved has to take the path. This leads to code divergence. Loads and stores to the device memory are issued for 16 threads at one time. If the memory addresses involved in this issue are consecutive, then the queries get coalesced into one large request and can make the most efficient use of the memory system. Otherwise, the memory requests get serialized, leading to slowdowns. Extracting the maximum performance from the GPU involves efficient utilization of the shared memory, minimizing non-coalesced memory accesses and avoiding code divergence between threads in a warp.

An interesting design tradeoff in GPU architectures is the lack of a well-defined memory consistency model for device memory. Since the threads are all scheduled by hardware, it is not possible to control their execution sequence. This leads to restricted

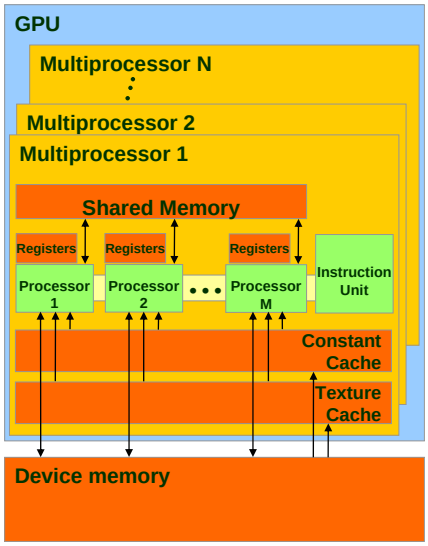

Figure 1: NVIDIA GeForce 8800 GPU Architecture

Table 1: Nvidia GeForce 8800 GTX Parameters

Number of Multiprocessors	16
Multiprocessor Width	8
# 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$\longleftrightarrow$GPU bandwidth	3.2 Gbit/s

synchronization capabilities (Only threads running on a single multiprocessor as a block can synchronize). This tradeoff, however enables easy scaling of the number of multiprocessors.

2.2 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.

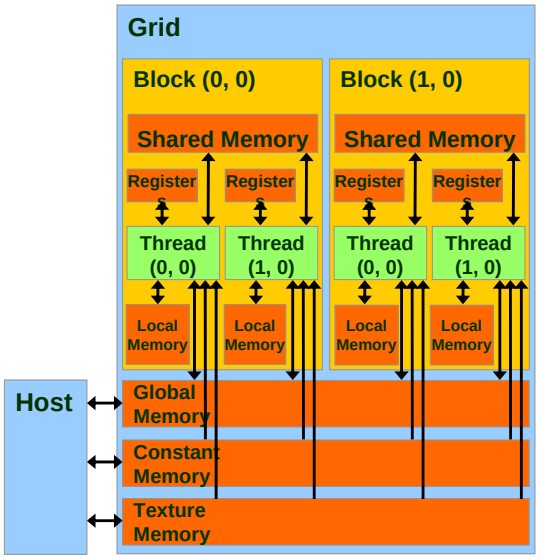

Figure 2: Logical organization of the GeForce 8800

Figure 2 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.

An important ramification of widespread GPU use would be SIMD-ification of data structures and algorithms. SIMD is already important for performance in uniprocessor systems (Intel SSE, for example). The biggest problem with SIMD usage has been programmability, as it is hard to think of vectors and managing corner cases becomes tricky. With the runtime managing the SIMD with CUDA, the programmer is able to think of just one thread (as opposed to a vector), leading to better programming efficiency. We believe that this is a characteristic that will be important for future many-core architectures as well.

3 Support Vector Machines

Support Vector Machines have found use in diverse classification tasks, such as image recognition, bioinformatics, and text processing due to their robust generalization characteristics. 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 $y \in \{-1, 1\}$.

3.1 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:

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

where $x_i \in \mathbb{R}^n$ is training data point i, $y_i \in \{-1, 1\}$ is the label attached to point x_i, and 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 $Q_{ij} = y_i y_j \Phi(x_i, x_j)$, where $\Phi(x_i, x_j)$ is a kernel function. We used the Gaussian kernel $\Phi(x_i, x_j; \gamma) = \exp\left\{-\gamma ||x_i - x_j||^2\right\}$ for our experiments.

3.1.1 SMO Algorithm

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 [4], Joachims' SVM^{light} [5], which introduced shrinking and kernel caching, Platt's Sequential Minimal Optimization (SMO) algorithm [6], and the working set selection heuristics presented in LIBSVM [7]. Despite this research, SVM training time is still significant for larger training sets.

Each of the previous methods have different parallelism implications. We have implemented the Sequential Minimal Optimization algorithm, [6], with a hybrid working set selection heuristic making use of the first order heuristic proposed by [8] as well as the second order heuristic proposed by [7].

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 $b_{low} \leq b_{high} + 2\tau$

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.

We initialize $b_{high} = -1$, $i_{high} = \min\{i : y_i = 1\}$, $b_{low} = 1$, and $i_{low} = \min\{i : y_i = -1\}$.

Updating the two alpha weights is done as follows:

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

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

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 $0 \leq \alpha_i \leq C$.

After the α update, the optimality condition vector f is updated for all points. This is one of the major computational steps of the algorithm, and is done as follows:

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

We define index sets:

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

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

We measure convergence by computing $b_{high} = \min\{f_i : i \in I_{high}\}$, and $b_{low} = \max\{f_i : i \in I_{low}\}$. The final offset for the solver is then $b = (b_{high} + b_{low})/2$.

3.1.2 Working set selection

At this point, we need to choose i_{high} and i_{low}, which correspond to the weights which will be changed in the optimization step. The first order heuristic from [8] chooses them as follows:

$$i_{high} = \arg\min\{f_i : i \in I_{high}\} \quad (7)$$

$$i_{low} = \arg\max\{f_i : i \in I_{low}\} \quad (8)$$

The second order heuristic from [7] chooses i_{high} and i_{low} to optimize the SVM functional without regards to feasibility. More explicitly:

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

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

To do this, after choosing i_{high}, we compute for all $i \in \{1..l\}$

$$\beta_i = f_{i_{high}} - f_i \tag{11}$$

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

$$\Delta F_i(\alpha) = \beta_i^2/\eta_i \tag{13}$$

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

3.1.3 Adaptive heuristic

Computationally, the second order heuristic requires significantly more work than the first order heuristic due to extra kernel function evaluations. In our GPU implementation, the geometric mean of iteration time over our benchmark set increased by 1.9× compared to the first order heuristic. On some benchmarks, the total number of iterations is decreased enough to provide a significant speedup, but on others, the second order heuristic slows computation down considerably.

To overcome this, 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 time using both heuristics, then chooses the more productive heuristic.

3.2 SVM Classification

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

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

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 4.3.

4 Application Mapping onto GPU and optimizations

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' 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.

The SMO algorithm is a good fit for the GPU architecture for the following reasons:

1. From figure 3, it is clear that for small working set sizes, most of the time is spent in updating the KKT conditions. This fits the GPU well because the individual updates are independent and the memory accesses involved are regular.

2. The amount of shared state between the threads is small - only two points that are involved in the QP have to be shared. This is important as the amount of shared memory available is limited on the GPU.

3. By storing the data appropriately, most memory accesses can be coalesced, leading to good performance.

The last optimization leads to the memory reads for the 2 shared data points (for loading them into shared memory) to become non-coalesced. To alleviate this, we store both the row major and column major versions of the data and read from the appropriate version.

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 [9]. This algorithmic pattern is very useful for extracting parallelism, since it is

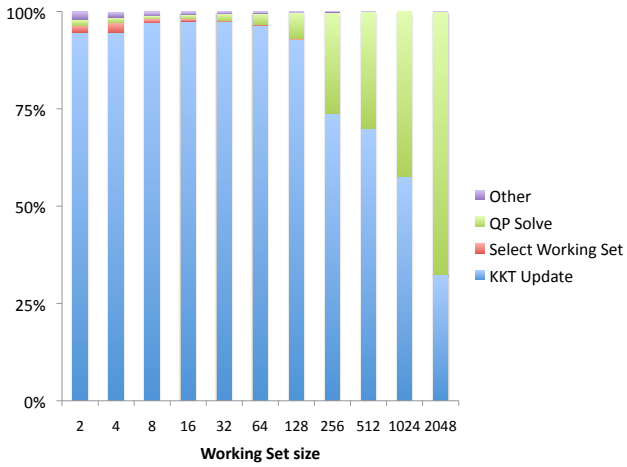

Figure 3: Task distribution Versus Working Set Size for Chunking algorithms on the Faces dataset

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 [10], and is a natural fit for our SVM training algorithm. In many machine learning applications, the Map Reduce pattern usually occurs inside an iterative structure. This makes any overhead inherent in the Map Reduce call significant, as the number of iterations can be huge ($> 1,000,000$ in one of our examples). Thus having an abstraction with a large overhead is usually unacceptable. Runtime systems like Phoenix [11] that perform Map Reduce on multicore report considerable overhead for machine learning algorithms like k-means.

For the first order heuristic, the computation of f' in (4) 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', b_{high} and i_{high}, and another where the map stage computes ΔF_i for all points, and the reduce stage computes b_{low} and i_{low}. In order to extract maximum parallelism, we structure the reductions as a tree, where the number of elements still participating in each reduction halves at each level.

Because the CUDA programming model has strict limitations on synchronization and communication between thread blocks, we have organized the reduction in two phases.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 $K_{ij} = \Phi(x_i, x_j)$ [5]. We compute rows of this matrix on the fly, as needed by the algorithm, and cache them in the available memory on the GPU. This is a software managed cache. The control structure for the cache is dealt with by the CPU, explicitly telling the GPU if it should read or write the line of cache passed to it.

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 SVM Classification Implementation

We approached the SVM classification problem by making use of the Map Reduce computations as well as vendor supplied Basic Linear Algebra Subroutines - specifically, the Matrix Matrix Multiplication routine (SGEMM). 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.

4.3.1 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 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 4.3.

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

The results of these optimizations are discussed in Section 5.

5 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.

5.1 Training

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

Table 2: Dataset Size

DATASET	# POINTS	# DIMENSIONS
ADULT [12]	32,561	123
WEB [6]	49,749	300
MNIST [13]	60,000	784
USPS [14]	7,291	256
FOREST [12]	561,012	54
FACE [15]	6,977	381

The references for the datasets used and their sizes are given in table 3.

We ran LIBSVM on an Intel Core 2 Duo 2.66 GHz processor, and gave LIBSVM a software 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 4 shows results from our solver. File I/O varies from 1.2 seconds for USPS to about 12 seconds for Forest dataset. File I/O has been excluded as it is an artifact of our experiments (It is not needed for most cases where the data is already in main memory). The CPU - GPU data transfer overhead was also very low. The exact times for the (one-time) data transfer are also given in table 5. 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.

Table 5 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

Table 4: SVM Training Results

Dataset	GPU 1st Order		GPU 2nd Order		GPU Adaptive		CPU-GPU data	LIBSVM		Speedup (×)
	Iter.	Time (s)	Iter.	Time (s)	Iter.	Time (s)	Transfer time(s)	Iter.	Time (s)	(Adaptive)
Adult	114,985	30.15	40,044	30.46	64,446	**26.92**	0.157	43,735	550.2	20.4
Web	79,749	174.17	81,498	290.23	70,686	**163.89**	0.209	85,299	2422.46	14.8
MNIST	68,055	**475.42**	67,731	864.46	68,113	483.07	0.323	76,385	16965.79	35.1
USPS	6,949	0.596	3,730	**0.546**	4,734	0.576	0.128	4,614	5.092	8.8
Forest	2,070,867	4571.17	236,601	**1441.08**	450,506	2023.24	0.390	275,516	66523.53	32.9
Face	6,044	**1.30**	4,876	**1.30**	5,535	1.32	0.131	5,342	27.61	20.8

Table 3: 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

shows that the adaptive heuristic performs robustly, surpassing or coming close to the performance of the best static heuristic on all benchmarks.

5.2 Classification

Results for our classifier are presented in table 6. We achieve $63 - 133\times$ 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.

The optimizations for the CPU code mentioned in Section 4.3.1 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 6. It should be noted that the GPU version is better than the optimized CPU versions by a factor of $4.7 - 18.5\times$.

For some insight into these results, we see 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 large dimensions (800), the CPU spends most of the time (70%) in compute bound SGEMM (For 10,000 support vectors and 12,000 data points). For problems with a small number of input space dimensions, the SVM classification process is memory bound, meaning it is limited by memory bandwidth. For low dimensions (50), the time taken on the CPU for memory bound tasks is 80%. The GPU however spends most of its time in the compute bound portion in all cases (96% and 69% spent in SGEMM respectively). Since the GPU has much higher memory bandwidth, as noted in section 2, it is attractive for such problems.

We tested the combined SVM training-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. The results were identical, which shows that our GPU based SVM system is as accurate as traditional CPU based methods.

6 Related Work

6.1 SVM Parallelization

There have been previous attempts to parallelize the SVM training problem. The most similar to ours is [16], 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 sophisti-

Table 5: SVM Classification Results

Dataset	LibSVM	CPU Optimized classifier		GPU Classifier			
	Time (s)	Time (s)	Speedup (×) compared to LIBSVM	CPU - GPU data transfer time(s)	Total Time (s)	Speedup (×) compared to LibSVM	Speedup (×) compared to CPU optimized code
Adult	61.307	7.476	8.2	0.023	0.598	102.5	12.5
Web	106.835	15.733	6.8	0.037	1.100	97.1	14.3
MNIST	269.880	9.522	28.3	0.086	2.037	132.5	4.7
USPS	0.777	0.229	3.4	0.0028	0.01238	62.8	18.5
Face	88.835	5.191	17.1	0.028	0.733	121.2	7.1

cated working set selection heuristics.

Many other approaches for parallelizing SVM training have been presented. The cascade SVM [17] is another proposed method for parallelizing SVM training on clusters. It uses a method of divide and conquer to solve large SVM problems. [18] 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 [19]. [20] proposes a method where the several smaller SVMs are trained in a parallel fashion and their outputs weighted using a Artificial Neural Network. [21] 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. [22] gives 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.

6.2 Using GPUs for general purpose computations

GPUs have been used for accelerating graphics and scientific applications. Sparse matrix solvers, Mesh refinement algorithms etc. are available on GPUs. Efficient scans have also been implemented on GPUs using CUDA recently [23]. Active work is being done in porting different applications on GPUs. [24] gives an extensive overview of the different efforts towards using GPUs for general purpose computations.

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 $63-133\times$ compared to LIBSVM. 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. With SIMD architectures becoming ubiquitous, it is necessary for programmers and computer architects to explore its ramifications. Machine learning algorithms, in particular, benefit a lot from the specific architectural characteristics of the GPUs, as they require high memory bandwidth and have (or can be restructured to have) good memory access properties.

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.

References

[1] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A

View from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technology@Intel Magazine*, February 2005.

[3] Nvidia, "Nvidia CUDA," 2007. `http://nvidia.com/cuda`.

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

[5] T. Joachims, "Making large-scale support vector machine learning practical," in *Advances in kernel methods: support vector learning*, Cambridge, MA, USA: MIT Press, 1999.

[6] J. C. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in kernel methods: support vector learning*, pp. 185–208, Cambridge, MA, USA: MIT Press, 1999.

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

[8] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to Platt's SMO Algorithm for SVM Classifier Design," *Neural Comput.*, vol. 13, no. 3, pp. 637–649, 2001.

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

[10] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *Advances in Neural Information Processing Systems 19* (B. Schölkopf, J. Platt, and T. Hoffman, eds.), pp. 281–288, Cambridge, MA: MIT Press, 2007.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multicore and multiprocessor systems," *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 13–24, 10-14 Feb. 2007.

[12] A. Asuncion and D. Newman, "UCI machine learning repository," 2007.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[14] J. J. Hull, "A database for handwritten text recognition research," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 16, no. 5, pp. 550–554, 1994.

[15] H. A. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 1, pp. 23–38, 1998.

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

[17] H. P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," in *Advances in Neural Information Processing Systems 17* (L. K. Saul, Y. Weiss, and L. Bottou, eds.), pp. 521–528, Cambridge, MA: MIT Press, 2005.

[18] L. Zanni, T. Serafini, and G. Zanghirati, "Parallel software for training large scale support vector machines on multiprocessor systems," *J. Mach. Learn. Res.*, vol. 7, pp. 1467–1492, 2006.

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

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

[21] L. V. Ferreira, E. Kaskurewicz, and A. Bhaya, "Parallel implementation of gradient-based neural networks for svm training," *International Joint Conference on Neural Networks*, Apr 2006.

[22] L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, *Large-Scale Kernel Machines*. The MIT Press, 2007.

[23] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *GH '07: ACM SIGGRAPH Symposium on Graphics Hardware*, pp. 97–106, 2007.

[24] "General-purpose computation using graphics hardware." `http://gpgpu.org`.