CUSVM: A CUDA IMPLEMENTATION OF SUPPORT VECTOR CLASSIFICATION AND REGRESSION

AUSTIN CARPENTER

Abstract. This paper presents cuSVM, a software package for high-speed Support Vector Machine (SVM) training and prediction that exploits the massively parallel processing power of Graphics Processors (GPUs). cuSVM is written in NVIDIA's CUDA C-language GPU programming environment, includes implementations of both classification and regression, and performs SVM training (prediction) at 13-73 (22-172) times the rate of state of the art CPU software.

1. Introduction

Due to its impressive generalization performance on a wide range of statistical prediction problems, the Support Vector Machine (Cortes and Vapnik, 1995) has been, for several years now, the most widely used kernel learning algorithm; however, the computationally expensive nature of training SVMs and, to a lesser extent, using them to predict new datapoints has limited the amount of training data that researchers have been able to employ. Moreover, the common practice of using cross-validation to select the SVM nuisance parameters (C, λ in classification and C, λ, ϵ in regression) most suitable to a particular prediction problem compounds the SVM's computational cost and puts further constraints on the feasible size of training sets. cuSVM brings the massively parallel processing power of GPUs to bear upon this problem, and as a result, its training times on a set of benchmarks were 12-73x faster than those of arguably the most popular CPU solver, LIBSVM (Chang and Lin, 2001). Its performance gains in prediction, at 22-172x, were even greater. In these tests, cuSVM was run on a NVIDIA GTX 260 GPU, which cost only $250 at the time of writing.

Section 3 of this paper introduces the SVM, the associated Quadratic Programming problem, and the modified Sequential Minimal Optimization (SMO) algorithm used by cuSVM. Section 4 gives a brief introduction to GPUs and CUDA. Section 5 describes cuSVM's implementation details, and finally, in section 6, cuSVM's performance is compared to that of LIBSVM.

2. Related Work

The two SVM training implementations most similar to cuSVM are (Cao et al., 2006) and, especially, (Catanzaro et al., 2008). The former also improved the SMO algorithm's performance by exploiting its inherent parallelism but used a cluster of CPUs synchronized with MPI (message passing interface) rather than a GPU, employed a less sophisticated working set selection method, did not include regression,

Date: January 17, 2009.

and was not published with accompanying software. Moreover, MPI is a very different programming model from CUDA, and it offers much less parallel processing power per dollar. Catanzaro et al. (2008) proved the effectiveness of implementing the same modified SMO algorithm as is used by LIBSVM and cuSVM in CUDA but did not implement regression. Also, their implementation[1] of prediction used solely single precision floating point arithmetic[2] rather than the slightly slower yet, in some problems, greatly more accurate mixed precision arithmetic used by cuSVM.

3. The Support Vector Machine

Given a set of training data vectors $\mathbf{x}_i \in \mathbb{R}^n, i = 1, ..., m$, of two classes and a label vector $\mathbf{y}$ such that $y_i \in \{1,\text{-}1\}$, $i = 1, ..., m$, training a SVM for use in classification (C-SVC) is equivalent to solving the following primal problem:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & \tfrac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, i = 1, ..., m. \end{aligned}$$

The dual of which is:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \tfrac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \mathbf{1}^T\boldsymbol{\alpha} \\ \text{subject to} \quad & \boldsymbol{y}^T\boldsymbol{\alpha} = 0 \\ & 0 \leq \alpha_i \leq C, i = 1, ..., m \end{aligned} \tag{3.1}$$

where $\mathbf{1}$ is a vector of ones; Q is the m by m positive semidefinite kernel matrix, $Q_{ij} \equiv y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$; and $K(\mathbf{x}_i, \mathbf{x}_j) \equiv \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$ is the kernel function, the most popular of which is the Gaussian:

$$e^{-\lambda\|\mathbf{x}_i - \mathbf{x}_j\|^2}. \tag{3.2}$$

Once the SVM has been trained, the following function is used to classify a new datapoint:

$$sign\left(\sum_{i=1}^{m} y_i\alpha_i K(\mathbf{x}_i, \mathbf{x}) + b\right). \tag{3.3}$$

In the case of regression (ϵ-SVR) (Vapnik, 1998), given are a set training data vectors $\mathbf{x}_i \in \mathbb{R}^n, i = 1, ..., m$ and a corresponding vector of target outputs $\mathbf{z} \in \mathbb{R}^m$, $z_i \in \mathbb{R}^1$. The primal is:

$$\begin{aligned} \min_{\mathbf{w},b,\xi,\xi^*} \quad & \tfrac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i + C\sum_{i=1}^{m}\xi_i^* \\ \text{subject to} \quad & \mathbf{w}^T\phi(\mathbf{x}_i) + b - z_i \leq \epsilon + \xi_i, \\ & z_i - (\mathbf{w}^T\phi(\mathbf{x}_i) + b) \leq \epsilon + \xi_i^*, \\ & \xi_i, \xi_i^* \geq 0, i = 1, ..., m. \end{aligned}$$

[1] Catanzaro et al. titled their C-SVC implementation GPUSVM. It is available for download at `http://www.cs.berkeley.edu/~catanzar/GPUSVM/`.

[2] Double precision floating point arithmetic is not supported by their test GPU, the NVIDIA 8800 GTX.

And, the dual is:

$$\begin{array}{ll} \min\limits_{\boldsymbol{\alpha},\boldsymbol{\alpha}^*} & \frac{1}{2}(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*)^T Q(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*) + \epsilon\sum_{i=1}^{m}(\alpha_i+\alpha_i^*) + \sum_{i=1}^{m} z_i(\alpha_i-\alpha_i^*) \\ \text{subject to} & \sum_{i=1}^{m}(\alpha_i-\alpha_i^*) = 0, \quad 0 \le \alpha_i, \alpha_i^* \le C, i=1,...,m, \end{array} \tag{3.4}$$

where $Q_{ij} \equiv K(\mathbf{x}_i, \mathbf{x}_j) \equiv \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$, and a new datapoint's output is predicted with the function:

$$\sum_{i=1}^{m}(\alpha_i^* - \alpha_i)K(\mathbf{x}_i, \mathbf{x}) + b. \tag{3.5}$$

3.1. **Decomposing and Solving the Quadratic Program.** Specialized decomposition methods (Fan et al., 2005; Joachims, 1999; Osuna et al., 1997; Platt, 1999) that carefully exploit the specific nature of the SVM training problem, primarily the sparsity of its solution, have been shown to be much more efficient than general purpose quadratic programming solvers, which require the precomputation of the full kernel matrix. Moreover, decomposition becomes a necessity as the number of training examples grows: the full kernel matrix quickly grows too large to fit in the memory of a reasonably priced computer.

cuSVM employs a modified version of the SMO (Platt, 1999) algorithm. SMO takes decomposition to its extreme by sequentially solving a series of quadratic programming subproblems of only two elements. The modification is the use of the second-order working set selection heuristic introduced in (Fan et al., 2005) and subsequently incorporated into LIBSVM's solver. Compared to its less sophisticated first-order predecessor (Keerthi et al., 2001), the second-order heuristic generally significantly reduces and almost never significantly increases (Catanzaro et al., 2008; Fan et al., 2005) the number of iterations necessary to solve (3.1), and while its computation is slightly more expensive per iteration, the significance of this added cost declines precipitously as the number of training examples grows. As the portion of the kernel matrix that can be cached decreases, the calculation of the rows of the kernel matrix corresponding to the two elements in the working set increasingly dominates the per-iteration cost.

Briefly, the modified SMO algorithm implemented in cuSVM is the following:

Algorithm 1 (A Modified SMO Decomposition Method)

1. Initialize $\boldsymbol{\alpha}^1 = \mathbf{0}$, the iteration counter $r = 1$, and the Karush-Kuhn-Tucker (KKT) optimality conditions vector:

$$f_i^1 = \sum_{j=1}^{m} y_j \alpha_j^1 K(\mathbf{x}_i, \boldsymbol{x}_j) - y_i = -y_i.$$

2. End algorithm if the stopping criterion (3.7) has been met, otherwise use the second-order heuristic (3.8, 3.9) to find the two working set indices i_{high} and i_{low}[3].
3. Update $\alpha_{i_{low}}^r$ and $\alpha_{i_{high}}^r$ to $\alpha_{i_{low}}^{r+1}$ and $\alpha_{i_{high}}^{r+1}$ using the computationally insignificant yet rather involved process described in (Chang and Lin, 2001).

[3] The iteration counter r is omitted from i_{high}, i_{low}, I_{up}, I_{down}, $b_{highstop}$, and $b_{lowstop}$ for simplicity of notation.

4. Update the KKT conditions vector (the algorithm's most expensive step):

$$f_i^{r+1} = f_i^r + (\alpha_{i_{low}}^{r+1} - \alpha_{i_{low}}^r) y_{i_{low}} K(\mathbf{x}_{i_{low}}, \mathbf{x}_i) + (\alpha_{i_{high}}^{r+1} - \alpha_{i_{high}}^r) y_{i_{high}} K(\mathbf{x}_{i_{high}}, \mathbf{x}_i) \tag{3.6}$$

5. Set $r \leftarrow r+1$. Go to Step 2.

In order to solve the regression dual (3.4) with Algorithm 1, cuSVM performs the reformulation found in (Chang and Lin, 2001).

3.1.1. *Stopping Criterion.* First define two index sets:

$$I_{up} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = 0\} \cup \{i : y_i < 0, \alpha_i = C\}$$
$$I_{down} = \{i : 0 < \alpha_i < C\} \cup \{i : y_i > 0, \alpha_i = C\} \cup \{i : y_i < 0, \alpha_i = 0\}$$

Algorithm 1 is deemed to have converged when

$$b_{highstop} - b_{lowstop} \leq \tau \tag{3.7}$$

where

$$b_{highstop} = \max_{i \in I_{up}} -y_i f_i^r$$
$$b_{lowstop} = \min_{i \in I_{down}} -y_i f_i^r.$$

The offset b in (3.3) and (3.5) is then calculated as

$$b = \frac{b_{highstop} + b_{lowstop}}{2}$$

3.1.2. *Second Order Working Set Selection (Fan et al., 2005).*

1. Select:

$$i_{high} \in \arg \max_{i \in I_{up}} \quad -y_i f_i^r \tag{3.8}$$

2. Define $q_i \equiv -y_{i_{high}} f_{i_{high}}^r + y_i f_i^r > 0$. Select:

$$i_{low} \in \arg \min_{i \in I_{down}} \{\frac{-q_i^2}{\psi_i} \quad | \quad -y_i f_i^r < -y_{i_{high}} f_{i_{high}}^r\} \tag{3.9}$$

where

$$\psi_i = \begin{cases} K(\mathbf{x}_i, \mathbf{x}_i) + K(\mathbf{x}_{i_{high}}, \mathbf{x}_{i_{high}}) - 2K(\mathbf{x}_i, \mathbf{x}_{i_{high}}), & x_i \neq x_{i_{high}} \\ \tau, & \text{otherwise.} \end{cases} \tag{3.10}$$

4. Graphics Processors and CUDA

Graphics Processor architectures are optimized for rendering real-time graphics, a highly compute and memory intensive problem domain with enormous inherent parallelism; thus, relative to a CPU, a much larger portion of a GPU's resources is devoted to data processing than to caching or control flow.

A consequence of GPU architectures' deemphasis of caching and control flow is the near impossibility, in most applications, of achieving their peak floating point operation rate; however, in order to realize large performance gains versus a CPU, it is often sufficient to reach only a fraction of this rate. This size of this fraction is determined primarily by the degree of parallelism that can be uncovered in the desired application.

CUDA, which stands for Compute Unified Device Architecture, is NVIDIA's GPU programming environment. The CUDA programming model consists of both *host* and *device* functions. The former are written in nearly standard C/C++, executed on the CPU, and used to call the latter, which are written in annotated C[4] and executed on the GPU, in order to accelerate highly parallel and computationally intensive tasks.

5. Implementation Details

5.1. **Kernel Evaluation.** The root of cuSVM's training performance advantage over CPU-based solvers such as LIBSVM is the parallelism that can be exposed in the most computationally intensive part of the modified SMO training algorithm, the calculation of the two rows of the kernel matrix corresponding to the indices in the working set. The row corresponding to i_{high} is used by the second order working set selection heuristic (3.9), and both are necessary for the update of the KKT conditions (3.6). As is noted in (Catanzaro et al., 2008), $-\lambda \left\|\mathbf{x}-\mathbf{y}\right\|^2$ from the Gaussian kernel (3.2) can be expressed in terms of matrix-vector multiplication as $\lambda(2\mathbf{x}{\cdot}\mathbf{y}-\mathbf{x}{\cdot}\mathbf{x}-\mathbf{y}{\cdot}\mathbf{y})$. Before initiating Algorithm 1, cuSVM computes on the CPU $\mathbf{x}_i \cdot \mathbf{x}_i, i = 1, ..., m$, and transfers the result to GPU memory. Then, during training iterations, cuSVM uses NVIDIA's CUDA basic linear algebra library, CUBLAS, to compute the necessary $\mathbf{x} \cdot \mathbf{y}$ and a GPU helper function to subtract $(\mathbf{x} \cdot \mathbf{x} + \mathbf{y} \cdot \mathbf{y})$, multiply by λ, and exponentiate.

cuSVM's algorithm for batch computation of the kernel matrix needed for the prediction of new datapoints is the result of modifying Volkov's highly optimized CUDA matrix multiplication algorithm (Volkov and Demmel, 2008) so that each operation computes $2x_iy_i - x_i^2 - y_i^2$ rather than x_iy_i and, in each entry of the output matrix, $\sum_{i=1}^{n} 2x_iy_i - x_i^2 - y_i^2$, where n is the number of features, is multiplied by λ and exponentiated.

5.2. **Mixed Precision Floating Point Arithmetic.** In prediction, the batch-computed kernel matrix must be multiplied by either the vector $\boldsymbol{y\alpha}$ (element by element multiplication), in C-SVC (3.3), or the vector $(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*)$, in ϵ-SVR (3.5). The author found, contrary to what is reported in (Catanzaro et al., 2008), that when there are many support vectors and only single precision floating point arithmetic is used, rounding error can cause the accumulation of these dot products to go catastrophically wrong; thus, cuSVM uses a mixed precision approach where the scalar dot product accumulator is stored in double precision. This limited use of double precision arithmetic has a small performance penalty and means that cuSVM is only compatible with relatively newly released NVIDIA GPUs; however, these costs seems merited when one considers that in the Forest prediction benchmark the accuracy rate obtained with a strictly single precision version of the cuSVM prediction algorithm, 56.87%[5], was drastically lower than the 80.05% rate (Table 6.5) obtained with either the same single precision cuSVM-trained SVM coefficients and the mixed precision cuSVM prediction algorithm or double precision LIBSVM-trained SVM coefficients and the double precision LIBSVM prediction algorithm. Finally, cuSVM uses a mixed precision rather than a fully double precision approach because the author has found no evidence that there are significant risks to using strictly single precision arithmetic in either training or the batch computation of

[4] Some C++ features are available

[5] The single precision version's accuracy rates on all other benchmarks in Tables 6.5 and 6.6 were very similar to those obtained by mixed precision cuSVM or LIBSVM

the predictive kernel matrix and thus the performance penalty that would come with the switch to full double precision seems unmerited.

5.3. **Kernel Caching.** The probability that an index will be selected for the working set given that it has recently been selected is much higher than the unconditional probability (Zhao et al., 2007); thus, many expensive kernel evaluations can be avoided by caching (Joachims, 1999) recently used rows of the kernel matrix. cuSVM caches the kernel rows of recently selected indices in GPU memory, and during each iteration, the CPU checks to see whether the needed kernel rows are in the cache. If not, the CPU instructs the GPU to compute the necessary row(s). If the cache is full, the least recently used row(s) is (are) overwritten.

5.4. **Parallel Reduction.** CUDA's strict constraints on communication and synchronization among blocks of processing threads make the implementation of reduction operations such as finding the maximum or sum of a vector nontrivial. Harris (2008) presents several effective techniques for parallelizing and optimizing reductions in CUDA, all of which cuSVM employs in its algorithms for finding $b_{highstop}$, $b_{lowstop}$, i_{low}, and i_{low}. The cuSVM GPU reduction functions check for membership in the appropriate index set (3.7) and either calculate $-y_i f_i^r$ in the cases of $b_{highstop}$, $b_{lowstop}$, and i_{high} or $\frac{-q_i^2}{\psi i}$ in the case of i_{low}. Their output is a vector of 64 candidate values–and in working set selection, accompanying indices–which are copied to CPU memory where the CPU performs a final sequential reduction. This transfer is motivated by the CPU's large efficiency advantage over the GPU in performing very small reductions.

6. Results

6.1. **Training.** cuSVM's performance is compared to that of LIBSVM. cuSVM only supports the widely-used Gaussian kernel function, so this kernel was employed in all tests. The C-SVC (ϵ-SVR) tests were run on the datasets detailed in Table 6.1 (6.2). These tables also contain the hyperparameter values used.

Table 6.1. C-SVC Datasets and Parameter Values

Name	C	γ	# Points	# Features	% Non-Zero
Adult (Asuncion and Newman, 2007)	100	0.5	32,561	123	11.3
Web (Platt, 1999)	64	7.8125	49,749	300	3.9
MNIST (LeCun et al., 1998)	10	0.125	60,000	784	19.1
Forest (Asuncion and Newman, 2007)	10	0.125	561,012	54	22.1

Table 6.2. ϵ-SVR Datasets and Parameter Values

Name	C	γ	ϵ	# Points	# Features	% Non-Zero
Adult (Asuncion and Newman, 2007)	100	0.5	0.5	32,561	123	11.3
Web (Platt, 1999)	64	7.8125	0.5	49,749	300	3.9
MNIST (LeCun et al., 1998)	10	0.125	0.5	60,000	784	19.1
KDDCup98 (Hettich and Bay, 2007)	2^{-7}	13.6436	0.01	95,241	403	82.6

Adult, Web, and MNIST were used in both the C-SVC and ϵ-SVR tests. In the latter case, the learning goal is simply that the continuously valued prediction of a datapoint be as close as possible to its label (either 1 or -1). KDDCup98 is a traditional regression problem.

LIBSVM was run on an Intel Core 2 Duo 2.66 GHz processor and given a cache size of 700MB, which is slightly larger than that allowed cuSVM. The same PC, which features a NVIDIA GTX260 GPU, was also used for cuSVM's benchmarks. The stopping criterion, τ, was set to 0.001 for both solvers. Lastly, neither solver's times include file I/O, and cuSVM's times include all data transfers between CPU and GPU memory.

Tables 6.3 and 6.4 show that, on these benchmarks, cuSVM converged to solutions that were practically the same, in terms of either b or the number of support vectors, as those obtained by LIBSVM in dramatically less time than was needed by the CPU solver. In training on the same dataset, cuSVM's speedup ratios were very similar regardless of whether the objective was classification or regression. Also, as expected given that cuSVM's matrix storage format is dense while LIBSVM's is sparse, cuSVM performed particularly well relative to LIBSVM on the densest of the benchmark datasets, KDDCup98.

Table 6.3. C-SVC Training Results

Dataset	# Support Vectors		Abs Difference in b	Training Time (s)		Speedup (x)
	cuSVM	LIBSVM		cuSVM	LIBSVM	
Adult	18,676	19,059	2.8×10^{-6}	31.6	541.2	17.1
Web	35,220	35,231	2.6×10^{-4}	228.3	2,906.8	12.7
MNIST	43,751	43,754	2.0×10^{-7}	498.9	17,267.0	34.6
Forest	270,305	270,304	8.0×10^{-3}	2,016.4	29,494.3	14.1

Table 6.4. ϵ-SVR Training Results

Dataset	# Support Vectors		Abs Difference in b	Training Time (s)		Speedup (x)
	cuSVM	LIBSVM		cuSVM	LIBSVM	
Adult	18,670	19,079	8.0×10^{-7}	31.6	548.8	17.4
Web	35,220	35,307	3.8×10^{-4}	230.8	3,280.9	14.2
MNIST	43,729	43,732	8.6×10^{-5}	465.9	16,499.0	35.4
KDDCup98	42,284	42,104	4.2×10^{-4}	254.9	18,519.2	72.6

6.2. **Prediction.** cuSVM's speed and accuracy in prediction are compared to LIBSVM's in Tables 6.5 and 6.6. cuSVM's predictions were made using cuSVM-trained SVMs and LIBSVM's were made using LIBSVM-trained SVMs. 10,000 datapoints were randomly selected from each training dataset for use in these tests. The fact that these datapoints were in-sample is irrelevant given that the purpose is simply to assess cuSVM's numerical accuracy.

In no test, whether C-SVC or ϵ-SVR, was cuSVM's predictive accuracy significantly less than LIBSVM's, and here cuSVM's speedups, at 22-172x, were even greater than in training. However, it is important to note that LIBSVM is not an especially optimized CPU classifier and thus not all of these performance gains are attributable to the GPU. Catanzaro et al. (2008) built a CPU classifier based on Intel Math Kernel Library BLAS routines that was 3.4-28.3x as fast as LIBSVM.

Table 6.5. *C*-SVC Prediction Results

Dataset	Accuracy (%)		Prediction Time (s) 10,000 Points		Speedup (x)
	cuSVM	LIBSVM	cuSVM	LIBSVM	
Adult	95.59	95.59	0.6	31.7	55.9
Web	97.43	97.43	2.7	58.8	22.2
MNIST	100.00	100.00	8.4	764.5	91.5
Forest	80.05	80.05	5.5	278.4	50.7

Table 6.6. ϵ-SVR Prediction Results

Dataset	Accuracy (MSE)		Prediction Time (s) 10,000 Points		Speedup (x)
	cuSVM	LIBSVM	cuSVM	LIBSVM	
Adult	0.2943	0.2943	0.6	31.7	55.9
Web	0.3015	0.3013	2.7	59.0	22.2
MNIST	0.2208	0.2208	8.4	706.4	84.3
KDDCup98	0.0004	0.0004	4.3	732.3	172.0

7. Conclusion

With cuSVM, it is now practical, on a retail desktop, to apply SVM classification and regression on a whole new range of problem sizes. All that is required is an inexpensive GPU. Moreover, cuSVM features a Matlab MEX wrapper that allows users to access the GPU's massively parallel processing power without having to perform any "real" programming.

References

A. Asuncion and D.J. Newman. UCI machine learning repository, 2007. URL `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

L.J. Cao, S.S. Keerthi, Chong-Jin Ong, J.Q. Zhang, U. Periyathamby, Xiu Ju Fu, and H.P. Lee. Parallel sequential minimal optimization for the training of support vector machines. *Neural Networks, IEEE Transactions on*, 17(4):1039–1049, July 2006.

Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 104–111, New York, NY, USA, 2008. ACM. Software available at `http://www.cs.berkeley.edu/~catanzar/GPUSVM/`.

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

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning (Historical Archive)*, 20(3):273–297, 1995.

Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, 6:1889–1918, 2005. ISSN 1533-7928.

Mark Harris. Optimizing parallel reduction in CUDA, 2008. `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/reduction/doc/reduction.pdf`.

S. Hettich and S. D. Bay. The UCI KDD archive, 2007. URL `http://kdd.ics.uci.edu`.

T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11, pages 169–184. MIT Press, Cambridge, MA, 1999.

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.*, 13 (3):637–649, 2001.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

E. Osuna, R. Freund, and F. Girosi. Improved training algorithm for support vector machines, 1997.

John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-19416-3.

Vladimir Vapnik. *Statistical Learning Theory*. Wiley, New York, 1998.

Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

Zhen-Dong Zhao, Lei Yuan, Yu-Xuan Wang, Forrest Sheng Bao, Shun-Yi Zhang, and Yan-Fei Sun. A novel model of working set selection for -+ decomposition methods. In *ICTAI '07: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence - Vol.2 (ICTAI 2007)*, pages 283–290, Washington, DC, USA, 2007. IEEE Computer Society.

austincarp@gmail.com, PatternsOnaScreen.net, Apt 18 345 E 12th St., New York, NY 10003