When to Cache Block Sparse Matrix Multiplication: A Statistical Learning Approach

Rajesh Nishtala
CS281A Final Project
December 2004

Abstract

In previous work it was found that cache blocking of sparse matrix vector multiplication yielded significant performance improvements (upto 700% on some matrix and platform combinations) however deciding when to apply the optimization is a non-trivial problem. This paper applies four different statistical learning techniques to explore this classification problem. The statistical techniques used are naive Bayes classifiers, logistic regression, support vector machines with linear kernels, and support vector machines with polynomial kernels. The results show that the support vector machines with polynomial kernels yield the best results. This paper also reasons about the distribution of the data from the differences in accuracy of the various models.

1 Introduction and Overview

We consider the problem of building high-performance implementations of sparse matrix-vector multiply (SpM×V), or $y \leftarrow y + A \cdot x$. We call x the *source vector* and y the *destination vector*. Making SpM×V fast is complicated both by modern hardware architectures and by the overhead of manipulating sparse data structures. It is not unusual to see SpM×V run at under 10% of the peak floating point performance of a single processor [9, Figure 1.1]. Moreover, in contrast to optimizing dense matrix kernels (dense BLAS) [10, 1], performance depends on the nonzero structure of the matrix which may not be known until run-time.

In prior work on the Sparsity system (version 1.0) [3], Im developed an algorithm generator and search strategy for SpM×V that was quite effective in practice. The Sparsity generators employed a variety of performance optimization techniques, including *register blocking*, *cache blocking*, and multiplication by *multiple vectors*. Cache blocking differs from register blocking in that cache blocking reorders memory accesses to increase temporal locality, whereas register blocking compresses the data structure to reduce memory traffic. We assume a reference implementation which stores the matrix in a compressed sparse row (CSR) format [7]. Cache blocking breaks the CSR matrix into a number of smaller r_{cache} x c_{cache} CSR matrices and then stores these sequentially in CSR format in memory. More information on the specifics of the blocking techniques used can be found in previous work [6] on the subject. The trade off that needs to be made is whether the benefit of the added temporal locality outweighs the added overhead.

As shown by Nishtala *et al.*[6], deciding when to apply the optimization is a challenging and non-trivial problem. Previous work created deterministic models and reasoned on the characteristics of the matrix that would signal when cache blocking should be applied; however most of these heuristics had exceptions and the performance models were unable to paint a definitive picture. This work re-approachs the problem of classification using statistical learning methods and analyzes how well these methods determine whether or not to cache block a matrix given the structural properties of the matrix.

2 Statistical Techniques

This section describes four different classification techniques that are used to classify a matrix as amenable to cache blocking. This section only presents a high-level overview of the techniques and thus the reader is referred to the references within each subsection for further reading on the subject. The classification goal of all the learning machines is as follows. Given a set of features (X_i), classify the matrix as being in one of two classes: matrices in which cache blocking helps $(Y = 1)$ and where it does not $(Y = 0)$. In our training set we classify a matrix in class $Y = 1$ if the optimally blocked performance is greater than 10% of the unblocked implementation. The X_i vector represents various structural features of the matrix which is further described in Section 4.

2.1 Gaussian Class Conditional Densities and Generative Models

In order to get a simple and intuitive model for $p(Y_i = 1|X_i)$, we first assume that that each of the different classes ($i.e. Y = 1$) implies a certain set of features X_i. In order to find a probability on Y, we must determine the mean values for the features in each of the classes, μ_0 and μ_1. In addition we find a pooled variance on the features, σ^2. This method assumes that the data are distributed via a Gaussian in each of the different classes. An extension for other distributions is straight forward. The probability, $p(Y_i = 1|X_i)$ is found to be in the form of the logistic function, $\phi(z)$:

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{1}$$

where z is an affine function on the feature vector X_i.

This model works well when the data are easily separable and well clustered. However, when this is not the case, the probability densities for both classes smear into each other resulting in high error rates in the classification. The reader is referred to [5, Section 7.2] for a more detailed treatment.

2.2 Logistic Regression

Another way to view the model is to assume that the set of features X_i imply a certain class Y. The subtle difference between this and the generative models is that we no longer assume an explicit clustering within each class of data. Rather, we try to fit the logistic function (Equation (1)) through our data. Since we don't know the form of the parameters to the logistic function, we need to estimate them with our training data. To find the *maximum likelihood* estimate of the parameters, the error joint probability (under the assumption that each of the individual probabilities are identically and independently distributed Bernoulli trials) is minimized using the parameter θ, given our training set. This produces a gradient descent algorithm that tries to minimize the error between the observed class and the probability that we are in that class.

This algorithm is a lot better about classifying the data since we no longer hold the assumption that the data are clustered well. However, we still try to fit a linear separator through the data points which might lead to suboptimal solutions. The reader is again referred to [5, Section 7.3] for a more detailed treatment.

2.3 Support Vector Machines using a Linear Kernel

In both our previous models, we assumed that the probability density takes on the form of the logistic function, however this is not necessarily a valid assumption. Using the theory of support vector machines [2], we assume that there is some probability density $p(Y_i = 1|X_i)$. We don't know anything about this explicit probability density nor do we care to. Instead a hyperplane is found through the data such that we maximize the minimum distance between two adjacent points on either side of that hyperplane. Maximizing this distance will yield an optimum separator. Once we have the seperator, the points on one side of this hyperplane are in one class while points on the other are in a different class. The main advantage of support vector machines are that they are non-parametric distributions, so we do not assume anything on the probability density $p(Y_i = 1|X_i)$.

2.4 Support Vector Machines using a Polynomial Kernel

All the models until now assumed that the best separator through the plane was a linear function of the data. In the actual calculations for the support vector machines, the value of the feature vectors enters through a dot product $X_i^T X_i$. This assumption can be further relaxed to give more interesting non-linear models. We can choose to relate the X_i using a more complicated inner-product such as a polynomial function $(X_i^T X_i + 1)^d$ and maximize the minimum distance over this surface through the hyperspace. [1] As we will see in the results, this model yields the best results, implying that the other constraints were too limiting to reason interesting features about the data. [2]

[1] A Gaussian radial basis function was also tried as the kernel, but a degree 3 ($d = 3$) polynomial produced the best results so we only present that data here.

[2] Both support vector machine models used a cost factor of 4. The cost factor is the factor by which $Y = 1$ classification errors outweigh $Y = 0$ classification errors.

3 Experimental Setup

In order to get a good data set and prime the learning machines properly, a collection of 88 sparse matrices was drawn from a variety of applications such as circuit design, chemical engineering, and document parsing. A full description of these matrices and their properties can be found in Richard Vuduc's PhD thesis[9].

Previous work [6] found that only a few matrices from this suite were amenable to cache blocking thus there are not enough matrices in the blocking class to properly prime the machines. In order to circumvent this problem, the matrices were randomly permuted to augment the standard set of matrices. First the rows were randomly permuted. This only changes the order in which the elements in the source vector x are accessed but the inherent spatial and temporal locality of accesses remains. Next the columns of the matrix are randomly permuted. This operation destroys any locality in accesses to x that might have existed. The last permutation randomly permutes both the rows and the columns.

We also run our experiments on three different processors: the Intel Itanium 2, the Intel Pentium 4, and the AMD Opteron. Since the performance depends heavily on the architecture that is used, the results on multiple platforms are interesting. However the various architectural parameters are not taken into account because on one perticular platform all these parameters are the same and will not influence the learning process. Our models are going to find different parameters for each platform and not try to find one overarching parameter that stretches across all platforms. For brevity, the exact features of the architectures, the compiler, and the compiler flags are not described here because they are not relevant to this work.

For each matrix, we perform an exhaustive search across different block sizes. In the row (column) dimension the block sizes range from 1024 to the number of rows (columns) by powers of 2. In addition to all the square block sizes, we iterate overall the rectangular block sizes as well. The statistical models for the generative models and the logistic regression were directly implemented using MATLAB primitive operations while the support vector machine was implemented using the SVMLight programs [4] and the complimentary MATLAB interface [8].

4 Matrix Features

In this section we explore the various matrix features (*i.e.*the X_i vector) that are used in the different models. In addition to just naming what they are we describe them and explain what intuitions they can give us about the blocked performance. The parameters are as follows:

- number of rows (m): This gives us an indication of the size of the destination vector y.
- number of columns (n): This gives us an indication of the size of the source vector x.
- number of nonzeros (k): This gives us an indication about the number of times we will access the source vector x.
- nonzero ratio : A measure of how sparse the matrix is. It is calculated as follows: $\rho = k/(m * n)$. From previous work, one of the thresholds on when to apply the optimization depends on the density of the matrix. If a matrix is too dense or too sparse, the optimization has no effect.
- bandwidth : A measure of how compact the rows are and how much of the source vector a particular row spans. It is calculated by finding the largest difference between the first and last nonzeros within a row overall the rows.
- correlation coefficient: We calculate the standard correlation coefficient between all the indices of the nonzeros. This is a measure of how well the nonzeros are clustered along a particular diagonal (and not necessarily the main diagonal) of the matrix. The closer the elements are to a diagonal, the less likely blocking will be useful, since the access to the source vector already yield high locality.
- average distance between nonzeros: We calculate the average number of columns between adjacent nonzeros. This is a measure of the spatial locality in the accesses to the source vector. The lower the distance, the higher the spatial locality.
- average number of nonzeros per block of columns: The columns are divided into groups of 4 columns. We then calculate the average number of nonzeros within each group across all the groups. This is a rough measure of how much reuse there is in the matrix and how much temporal locality the matrix exhibits.

Factor analysis [5, Chapter 14] was also tried to see whether the feature space could be reduced, however this showed very little positive results. To increase the parameter space, each parameter was normalized with respect to the maximum value within that parameter. Every possible pair of parameters was then linearly combined using a naive average. This was also found not to help. Thus the data in the following section is directly based on the parameters above, normalized with respect to the maximum within each parameter.

5 Experimental Results

There are two different metrics of success that can be used. The first is the number of false positives and false negatives each model produces. A second metric is how much potential performance is lost or realized by using the models. Thus for matrices that show less than a 10% potential speedup with blocking, false negatives don't really harm the performance and the misprediction penalty is not that large. However false negatives that do not capture 100% potential gains in performance are important. Therefore the second metric is a lot more relevent to measuring how successful the model is within this optimization space.

For each platform and statistical model we create a histogram of how successful that statistical technique is. We partition the speedups into bins that are 10% wide. The y axis is the number of matrices that exhibit a speedup within that 10% range. A speedup is considered positive if the model correctly classifies the matrix or else it is considered negative. A negative speedup *does not* represent a slow down; rather it is a measure of how much potential speedup was not realized because of the misclassification. The y axis has also been truncated to accentuate the matrices with potential speedups larger than 10%. Since our training set is so small, we are forced to report the numbers for the training set only. Future work will increase the number of matrices and set a small set of them aside for testing purposes.

5.1 Generative Models

Figure 1 shows the performance of the generative model. On platforms such as the Opteron and Pentium 4 there are a lot of misclassifications with this technique. On the Opteron, we misclassify a matrix that has a potential speedup of 600%! Results are nearly as poor on the Pentium 4 and the Itanium 2; we misclassify matrices that have very large potential speedups. This implies that the decision to block does not imply a certain set of features nor can the data be easily separated into clusters.

5.2 Logistic Regression

Figure 2 shows that the logistic regression models do a lot better at predicting performance. We correctly classify more matrices, including the previously mentioned matrix with a potential for 600% improvement on the Opteron. However there are still many misclassifications. A reason that the logistic function does better at classifying the data is that it explicitly takes each error of classification into account and tries to minimize that error whereas the generative models try explicitly cluster the data. The difference in success rates of both the models, implies that the data is not well clustered but reasonably separable thus it makes sense to classify the data.

5.3 Support Vector Machines using a Linear Kernel

Figure 3 shows that we do even better and classify more matrices correctly by using support vector machines with linear kernels. There are still a few matrices that show significant speedups that we are unable to capture, but in general the number of matrices that we have misclassified has dropped. This leads one to believe that there is no fixed distribution on the data and that non-parametric models would produce good results. However the large losses in potential speedup, indicate that a simple hyperplane through the data might not be the optimal solution and more advanced models should be considered.

5.4 Support Vector Machines using a Polynomial Kernel

The final set of models try to fit a degree 3 polynomial through the data. As seen in Figure 4, these perform the best at correctly classifying the data with very few mispredictions. The low degree polynomial implies that the data is mostly separable. In fact on the Pentium 4, we do not misclassify any matrices that have potential speedups of greater than 10%. On the Opteron however the linear kernel and the polynomial kernel yield the same results. Thus by our metric of success this is the best model.

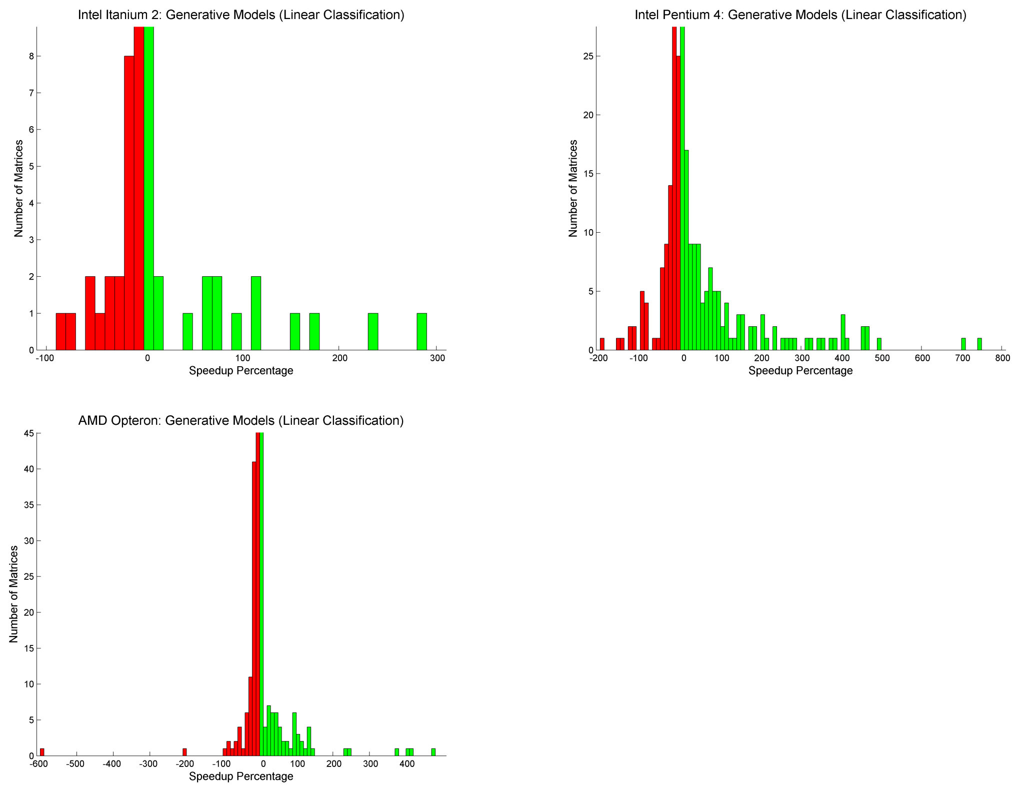


Figure 1: The histograms for the generative models described in Section 2.1

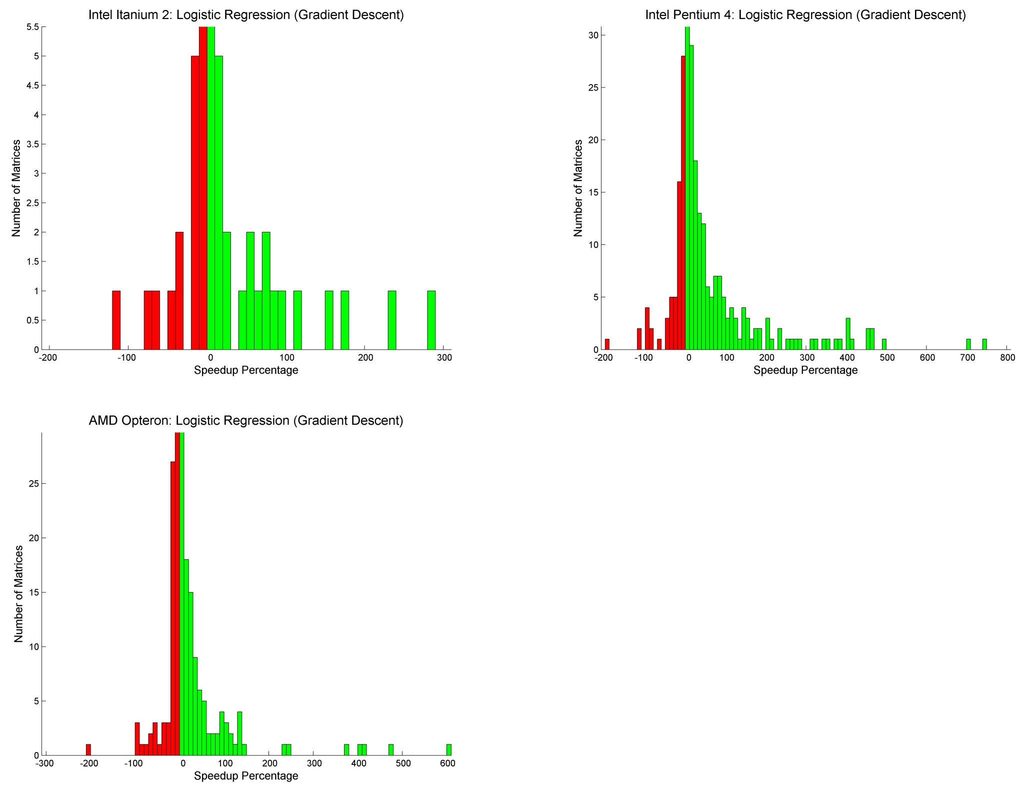
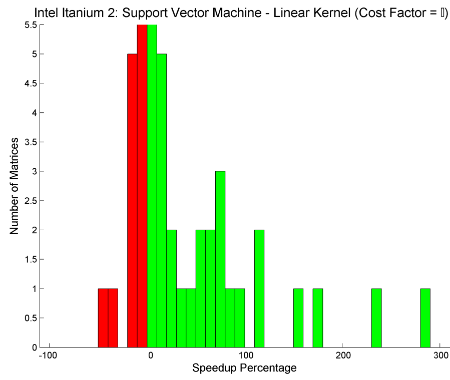
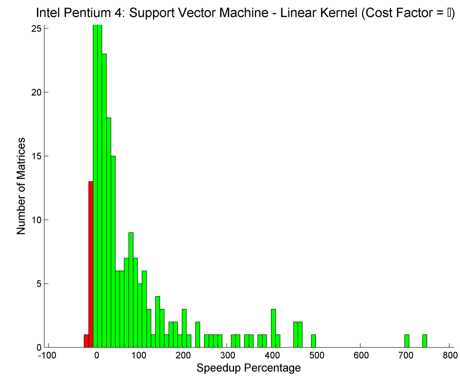


Figure 2: The histograms for the logistic regression models described in Section 2.2





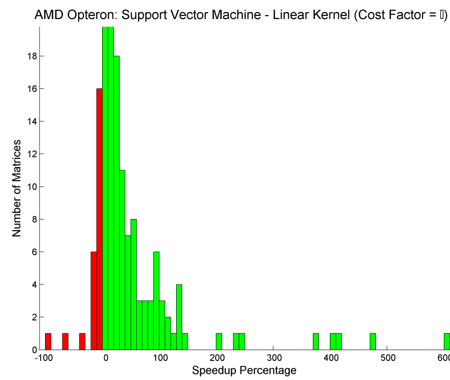
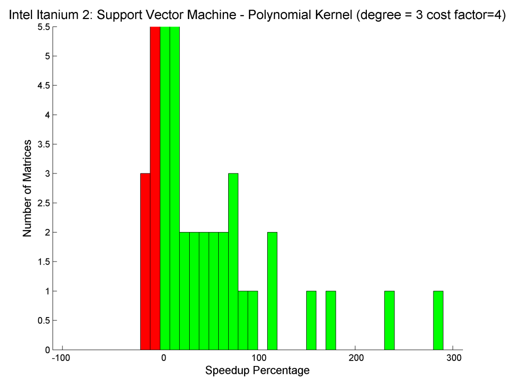
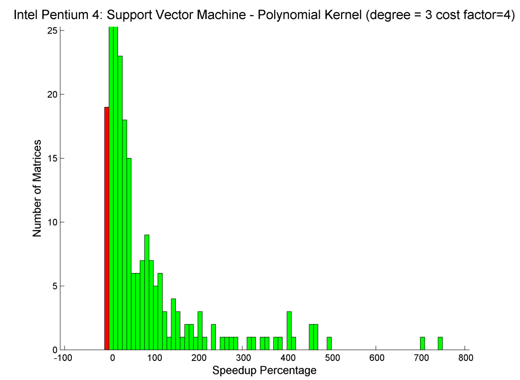


Figure 3: The histograms for the support vector machines described in Section 2.3





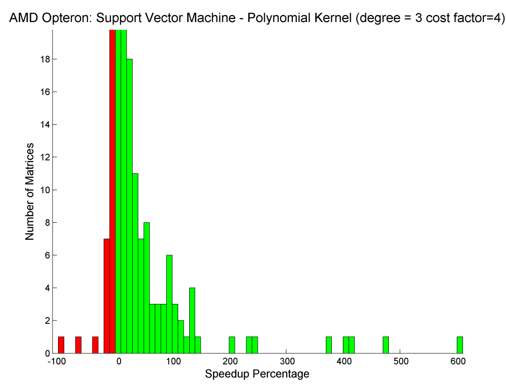


Figure 4: The histograms for the support vector machines described in Section 2.4.

6 Conclusion and Future Work

The main aim of this work has been as follows: given a sparse matrix we wish to classify it as amenable to cache blocking. The success of the various models leads us to some intuitions on what the data set looks like. The fact that the naive generative models failed to give good results indicate that the data are not well clustered. Because the logistic regression models produce better results, we can reason that even though the data are not well clustered, it is separable and that it makes sense to try to separate the models into two classes. Furthermore, the support vector machines with linear kernels imply that relaxing the assumption that there is a fixed parameterized probability distribution is a good idea. However this method still has a few misclassifications leading us to believe that a linear separator between the classes might not be the most optimal. We finally found that a polynomial surface that separates the data produce the best results.

The statistical learning techniques have taken a huge step forward in solving some of the open problems presented by Nishtala *et al.* in trying to reason when to cache block a sparse matrix. There is still however much future work to be done. The first step is to find more matrices through which the models can be primed. This would negate the need to artificially augment our matrix suite. Another task would be to take the classification result and then try to predict the optimal block size. In addition, one can imagine adding more parameters into the model that would try to capture more about the structure of the matrix. One goal throughout the implementation of the models was to keep them generic enough so that they can be applied to a different set of classification problems in other areas of sparse linear algebra.

References

[1] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC. see `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[2] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[3] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication.* PhD thesis, University of California, Berkeley, May 2000.

[4] T. Joachims. Svm light version 6.0. *http://svmlight.joachims.org/*, 2004.

[5] M. I. Jordan. *Introduction to Probabilistic Graphical Models.* unpublished draft, Berkeley, CA, 2004.

[6] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance Modeling and Analysis of Cache Blocking in Sparse Matrix Vector Multiply. Technical Report UCB/CSD-04-1335, University of California, Berkeley, Berkeley, CA, USA, June 2004.

[7] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, NASA Ames Research Center, Moffett Field, CA, 1990.

[8] A. Schwaighofer. A matlab interface to svm light to version 4.0. *http://www.cis.tugraz.at/igi/aschwaig/software.html*, 2004.

[9] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels.* PhD thesis, University of California, Berkeley, 2003.

[10] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. of Supercomp.*, 1998.