Using GPUs to speedup sparse coding algorithms applied to self-taught learning problems

Anand Madhavan

Abstract

In this work, we present a new combination of sparse coding algorithms designed to exploit the GPU and apply it to a self-taught learning [1] problem. We build on top of the iterative approach of solving two convex optimization problems alternatingly (originally suggested in [2]). The first and most important improvement in this work is a parallelized coordinate descent algorithm for the generalized L_1-regularized least-squares optimization problem, used to solve for the sparse coefficients. We demonstrate the improved speedups against the feature-sign search algorithm described in [2]. The second improvement is in a projected gradient descent implementation for the basis learning portion. Finally, an application of this implementation on a self-taught learning problem using unlabelled handwritten digits to predict handwritten letters is demonstrated.

1 Introduction

An efficient sparse coding approach based on alternating convex optimization over two subsets of the variables was described in [2] and was applied to solving the self-taught learning problem in [1]. The approach however can still take a long time to solve. As an illustration, when it was used to characterize natural images, the algorithm takes 10 hours to solve for 2000 bases (each 20x20 pixels) using off-the-shelf desktops in 2008 [1]. In this project, we present a way of improving the performance of this approach by specifically designing algorithms that exploit the compute power of commodity GPUs. We then apply it to natural images to show performance comparisons with prior Matlab implementations and finally demonstrate an application to handwritten digits.

To briefly describe the sparse coding problem, let $X \in \mathbb{R}^{kxm}$ be the input matrix (each column is an input vector), let $B \in \mathbb{R}^{kxn}$ be the basis matrix (each column is a basis vector $\in \mathbb{R}^k$), and let $S \in \mathbb{R}^{nxm}$ be the coefficient matrix (each column is a coefficient vector $\in \mathbb{R}^n$). Then the basis vectors and the sparse coefficients can be found as the solution to the following optimization problem:

$$\text{minimize}_{B,S} \frac{1}{2\sigma^2} \| X - BS \|_F^2 + \beta \sum_{i,j} \phi(S_{i,j})$$

$$\text{subject to } \sum_i B_{i,j}^2 \leq c, \forall j = 1,..., n$$

where $\phi(.)$ is the sparsity function and β is a constant. Using an L_1 penalty ($\| S_{i,j} \|_1$) as the sparsity function, the optimization problem is convex in B (while holding S fixed) and convex in S (while holding B fixed), but not convex in both simultaneously. This was solved in [2], by alternatingly solving first for the bases B (as a least squares problem with quadratic constraints) using the Lagrange dual, and then for the coefficients S (as a regularized least squares problem) using the feature-sign search algorithm. However both these methods are not very conducive to massive parallelism due to the presence of inverse terms in their closed form solutions [2]. Generic libraries for solving for the inverse of a matrix on the GPU do not currently exist (for example not part of CUBLAS [3]. Further, since feature-sign exploits sparsity, it is not the most efficient algorithm on generic regularized least squares problems ()a wider problem of interest in the learning and optimization community).

2 L_1-regularized least-squares: A parallelized coordinate descent algorithm

We first consider the step of solving the L_1-regularized least-squares problem for the coefficients S keeping the bases B fixed. Consider an alternative to feature-sign search, the popular, coordinate descent algorithm. This algorithm on its own is not particularly parallel since it involves proceeding along one coordinate after the other.

However, we could alternatively find the descent directions along all the coordinates in any given step and chose a descend direction that is the resultant direction suggested by all of them. This is an idea that lends itself to a high degree of parallelism (i) allowing simultaneous computations of the coefficients for many inputs and (ii) allowing for the simultaneous computation of the descent direction, with each coordinate being computed in a separate thread. We present this algorithm as a solution. Thus the optimization problem can be simplified to:

$$\text{minimize}_x f(x) \equiv \tfrac{1}{2} \parallel y - Ax \parallel^2 + \gamma \parallel x \parallel_1$$

one equation per training example, where γ is a constant (regularization coefficient) and $x \in \mathbb{R}^{\mathrm{n}}$, $y \in \mathbb{R}^{\mathrm{k}}$ and $A \in \mathbb{R}^{\mathrm{kxn}}$. We optimize above w.r.t one x_j at a time (while keeping the others fixed) and determine the new descent direction as follows:

$$x_j^* = \begin{cases} 0 & if \mid -y^T a^{(j)} \mid < \gamma \\ \frac{y^T a^{(j)} - \gamma}{a^{(j)T} a^{(j)}} & if\ (-y^T a^{(j)}) < -\gamma \\ \frac{y^T a^{(j)} + \gamma}{a^{(j)T} a^{(j)}} & if\ (-y^T a^{(j)}) > \gamma \end{cases}$$

where $a^{(j)} \in \mathbb{R}^{\mathrm{k}}$ (jth column of the matrix A). We perform a line search along the direction $d = x - x^*$ to find a step size, α (where $0 < \alpha \leq 1$), that reduces the value of the objective function. Since d is a descent direction, such a step size can always be found [1]. This is then repeated until the change in the cost function is within a specified tolerance.

This algorithm, is easier to parallelize as each thread can compute the simple coordinate-wise optimum. The line search in turn can also be parallelized, or we can just take a pre-determined step size in that direction. Further our algorithm can be used as a substitute to feature-sign search as a whole, or as part of the feature-sign search (instead of the analytical solution that uses an inverse in [2]).

3 GPU implementation of the parallelized coordinate descent algorithm

We use Nvidia's CUDA enabled GPUs for our implementation. These come with many streaming multiprocessors (SMs) [4] each with many scalar cores (SPs) (see Figure 1). Threads running on the cores within an SM can communicate with each other using shared memory. The unit of thread execution on the SP is a kernel and this is executed in concert with many other threads that form a 'block'. Many blocks can be scheduled on the GPU, however a block is assigned only to one SM and hence threads across blocks should not rely on communications with each other. Blocks are scheduled by the hardware as and when SMs have resources available to handle them.

We exploit the parallelism available in the many SMs by creating as many blocks as there are training data (m). This works very well since we don't require communications across training data during coefficient computations. We exploit the many SPs in an SM, by scheduling as many threads in a block as there are coordinates (n). This allows for the coordinates to be shared during computations (such as computation of x vector norms). Temporary data (mostly vectors of size n) are stored in the

[1] We state it here without providing proof in the interest of space

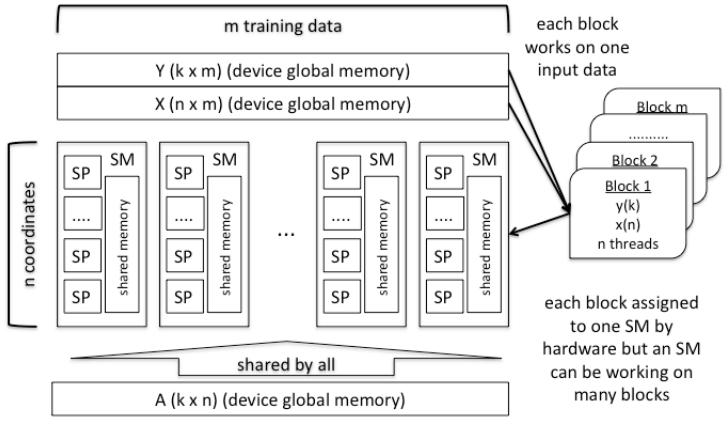

Figure 1: Simplified schematic illustrating the blocks, threads and various portions of the matrices A, Y and X used by them.

shared memory space of an SM and is shared across all the cores of that SM. The A, X and Y matrices above are stored in global device memory and is shared across all the SMs. The kernel function that is executed on each thread is then repeated until the minimum of the cost function is found. Figure 1 illustrates this.

We benchmark this solution using various sizes of m, n and k and compare its performance on the GPU against the equivalent Matlab version on the CPU. These measurements were made on a Macbook Pro with an Intel Core2 duo machine with an 8600M GT GPU, thus making it a 'fair' enough comparison. Figure 2 shows the speedup graphs. Speedups of upto 7x are achieved even on a modest Macbook pro (with 32 SM cores, compared to say 240 cores on a Tesla 280 GTX)

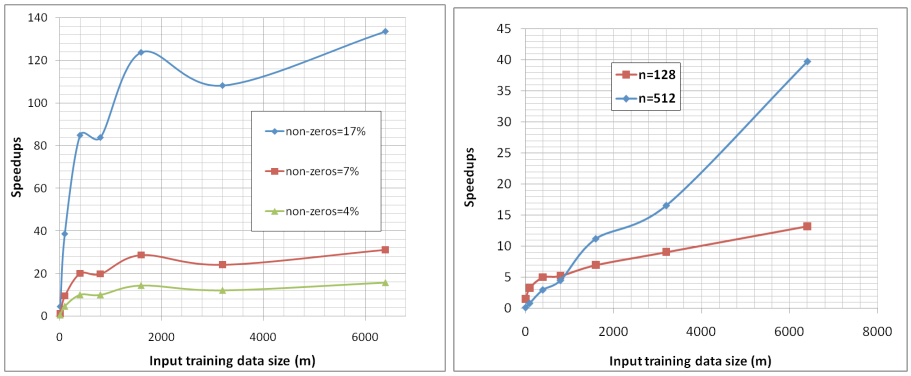

Figure 2: Left: Speedup of the parallelized coordinate descent on GPU vs feature-sign search on CPU (for 512 basis vectors and k=196). Right: Speedup of the same algorithm on the CPU vs on the GPU.

4 Learning bases using projected gradient descent

As the second step of the optimization in sparse coding, we solve for the basis keeping the coefficients fixed. This is a least squares problem with quadratic constraints.

$$\text{minimize}_x \| X - BS \|_F^2$$

$$\text{subject to } \sum_{i=1}^{k} B_{i.j}^2 \leq c, \quad \forall\, j = 1, ..., m$$

This can be solved very efficiently using the Lagrange dual [2]. However the solution involving the Lagrange dual requires taking an inverse, which is not very parallelizable and in the spirit of exploring parallelizable algorithms in this work, we consider using a projected gradient descent approach instead. This gives us a closed form solution for the gradient w.r.t B, and results in the following update rule for the projected gradient descent:

$$B := B - \eta \nabla_B \parallel X - BS \parallel_F^2$$

constrained at each step by scaling B down such that

$$\sum_{i=1}^{k} B_{i,j}^2 \leq c, \quad \forall\, j = 1, ..., m$$

$$\text{where } \nabla_B \parallel X - BS \parallel_F^2 = -2(X - BS)S^T$$

When implemented this algorithm using CUBLAS libraries [3], this can provide significant speedups compared to a CPU version but only when run on large number of inputs sizes (m). Typically however the Lagrange-dual closed form solution can in general be faster. However this is not much of an issue since basis computation takes a much smaller fraction (and if not we decrease the iterations to convergence accordingly, with not much loss in iterative convergence) of the total time, which is often dominated by the coefficient computation part. For example on a problem of size m=500 (k=400, n=50), the Lagrange-dual takes around 2.75 seconds, while the CUBLAS version takes 0.25 seconds (incidentally the Matlab version of the projected gradient descent takes around 131 seconds). The benefits are better with increasing size. The numbers are rudimentary and so were not reliable enough to plot.

5 Evaluation on natural images

We then proceed to evaluate the combined GPU implementations from sections 3 and 4 on 62 natural images. This is learnt by breaking up the images into patches of 20x20 each. Figure 3 shows the 512 basis vectors learnt. The entire run takes a few hours on a 8800 GT machine. The convergence profile is shown as well in figure 3.

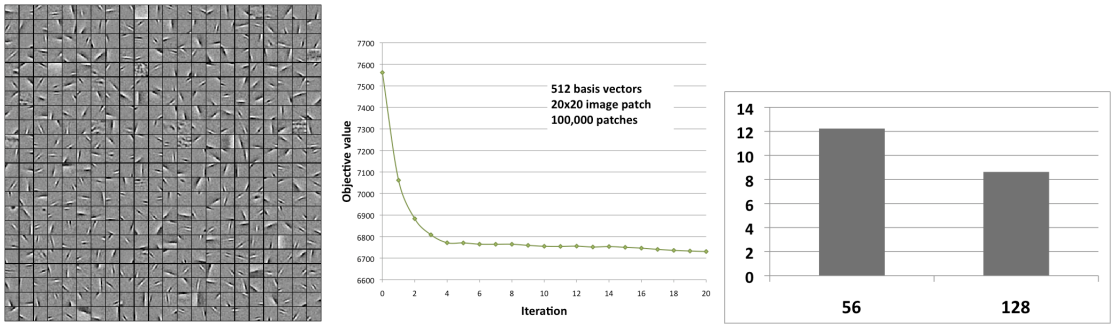

Figure 3: *Left:* 512 basis learnt using natural images (using 20x20 patches) using a combined GPU code using alternating kernel and CUBLAS code. *Middle:* Joint convergence of the algorithms. *Right:* Sample speedups against Matlab versions (of our algorithms) for a smaller problem size (10000 images, 16x16 patch, 56 and 128 basis vectors case shown).

6 Application to Self Taught Learning on handwritten letters

Finally, we apply the implementation to a self-taught learning problem using handwritten letters and images. We are interested in classifying handwritten letters as a, b, c etc (right-most tile in figure 4). We however are in a (albeit made-up) scenario where we have only say a few labelled handwritten letters and we seek to automatically characterize them. Coming up with the basis vector using just a handful of letters however is not feasible and so we instead train it on a large number of handwritten digits instead. Figure 4 (left-most) shows a sampling of the digits used. The entire 784-dimension (28x28) images were initially used for learning the basis. However, these produced highly overfitted basis that looked very much like the handwritten digits (center-left in figure 4). Thus we apply PCA to these digits and obtained a much much better basis vectors that capture strokes instead of just the digits. Center-right image in figure 4 shows the final resulting basis obtained (9 of the 512 basis are shown).

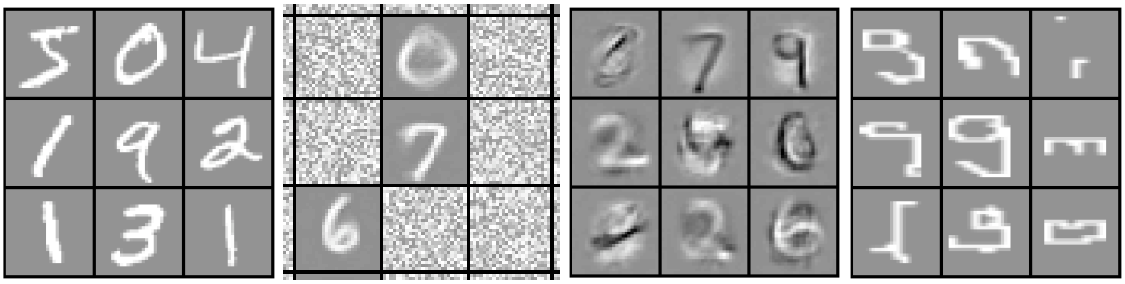

Figure 4: *Leftmost*: Sample of the 60000 handwritten digits used to train the basis. *Center-left*: Sample of the overfitted basis when using all 784 pixels. *Center-right:* Sampling of 9 of the 512 basis vectors learnt using handwritten digits. Right: Sampling of the letters that we are interested in labeling using self-taught learning.

We then use these basis to determine coefficients for the labelled letters and feed these coefficients as labelled features into an SVM (we use SVM-multiclass for this purpose [5]). Each time we require to classify a letter, we compute its coefficients in a similar fashion and use these as features for classification by the SVM. It is worth noting that in finding the coefficients, we again use the parallelized coordinate descent algorithm, which makes this step very fast. The prediction results for this step are tabulated in table 2.

# of training examples	Accuracy on letters
100	33.29%
500	51.88%
1000	59.07%

Table 2: 196 features (after reduction using PCA), 26 classes, 512 basis vectors

7 Conclusion and suggested future work

Conclusions

- The parallelized coordinate descent algorithm was shown to be a fast algorithm, conducive to parallelism on the GPU. This algorithm on its own is valuable in a variety of instances:

    1. As a solution within feature-sign search algorithm itself [2].

2. As a replacement to the entire coefficient computation step within sparse coding [2].
3. In finding the coefficients during real-time classification of data using self-taught learning as demonstrated in section 6.
4. As a solution the generic L1 regularized least square problem applicable to many such other areas.

- We obtained speedups of 7x over the feature-sign search algorithm even on a modest GPU on a macbook pro. Running this on more powerful GPUs proportionately scales and so provides more benefits when applied to larger problems.
- Running the problem end to end on a self-taught learning application also provided speedups that were not measured, but were definitely perceptible. On the natural images which we did measure, we saw speedups of around 12x.
- Together these provide compelling reasons for us to use GPUs in machine learning (specifically in brain-based probabilistic models).

Future work

- Future work should definitely involve exploiting even more parallelism (for example in the line-search within the two algorithms)
- Applications to larger problem sizes such as textual classification and audio classifications.
- With improved speeds using commodity GPUs, we can think of real-time classification tasks onboard robots and other devices.

Acknowledgements

Immense thanks to Rajat Raina for providing invaluable guidance, help with formalisms and insights into the problem, as well as help with optimizing the Matlab version of the implementation of the coordinate descent algorithm. Thanks to Honglak Lee for graciously providing his GPU and machine for benchmarking. Also thanks to Dr. Andrew Ng for encouraging a project like this in CS 229.

References

[1] Rajat Raina, Alexis Battle, Honglak Lee, Benjamin Packer, and Andrew Y. Ng. Self-taught learning: transfer learning from unlabeled data. In *ICML*, pages 759–766, 2007.

[2] Honglak Lee, Alexis Battle, Rajat Raina, and Andrew Y. Ng. Efficient sparse coding algorithms. In *In NIPS*, pages 801–808. NIPS, 2007.

[3] In *CUBLAS Library 1.1*. Nvidia Inc, 2007.

[4] In *CUDA Programming Guide 2.0*. Nvidia Inc, 2008.

[5] Svm-multiclass: http://www.cs.cornell.edu/people/tj/svm