Large-scale Deep Unsupervised Learning using Graphics Processors

Rajat Raina   RAJATR@CS.STANFORD.EDU
Anand Madhavan   MANAND@STANFORD.EDU
Andrew Y. Ng   ANG@CS.STANFORD.EDU
Computer Science Department, Stanford University, Stanford CA 94305 USA

Abstract

The promise of unsupervised learning methods lies in their potential to use vast amounts of unlabeled data to learn complex, highly nonlinear models with millions of free parameters. We consider two well-known unsupervised learning models, deep belief networks (DBNs) and sparse coding, that have recently been applied to a flurry of machine learning applications (Hinton & Salakhutdinov, 2006; Raina et al., 2007). Unfortunately, current learning algorithms for both models are too slow for large-scale applications, forcing researchers to focus on smaller-scale models, or to use fewer training examples.

In this paper, we suggest massively parallel methods to help resolve these problems. We argue that modern graphics processors far surpass the computational capabilities of multicore CPUs, and have the potential to revolutionize the applicability of deep unsupervised learning methods. We develop general principles for massively parallelizing unsupervised learning tasks using graphics processors. We show that these principles can be applied to successfully scaling up learning algorithms for both DBNs and sparse coding. Our implementation of DBN learning is up to 70 times faster than a dual-core CPU implementation for large models. For example, we are able to reduce the time required to learn a four-layer DBN with 100 million free parameters from several weeks to around a single day. For sparse coding, we develop a simple, inherently parallel algorithm, that leads to a 5 to 15-fold speedup over previous methods.

1. Introduction

We consider two well-known unsupervised learning models, deep belief networks (DBNs) and sparse coding, that can learn hierarchical representations of their input (Olshausen & Field, 1996; Hinton & Salakhutdinov, 2006). With the invention of increasingly efficient learning algorithms over the past decade, these models have been applied to a number of machine learning applications, including computer vision, text modeling and collaborative filtering, among others. These models are especially well-suited to problems with high-dimensional inputs, over which they can learn rich models with many latent variables or layers. When applied to images, these models can easily have tens of millions of free parameters, and ideally, we would want to use millions of unlabeled training examples to richly cover the input space. Unfortunately, with current algorithms, parameter learning can take weeks using a conventional implementation on a single CPU. Partly due to such daunting computational requirements, typical applications of DBNs and sparse coding considered in the literature generally contain many fewer free parameters (e.g., see Table 1), or are trained on a fraction of the available input examples.

In our view, if the goal is to deploy better machine learning applications, the difficulty of learning large models is a severe limitation. To take a specific case study, for two widely-studied statistical learning tasks in natural language processing—language modeling and spelling correction—it has been shown that simple, classical models can outperform newer, more complex models, just because the simple models can be tractably learnt using orders of magnitude more input data (Banko & Brill, 2001; Brants et al., 2007).

Analogously, in our view, scaling up existing DBN and sparse coding models to use more parameters, or more training data, might produce very significant performance benefits. For example, it has been shown that sparse coding exhibits a qualitatively different and highly selective behavior called "end-stopping" when

Table 1. A rough estimate of the number of free parameters (in millions) in some recent deep belief network applications reported in the literature, compared to our desired model. To pick the applications, we looked through several research papers and picked the ones for which we could reliably tell the number of parameters in the model. All the models do not implement exactly the same algorithm, and the applications cited may not have used the largest-scale models possible, so this is not an exact comparison; but the order of magnitude difference between our desired model and recent work is striking.

Published source	Application	Params
Hinton et al., 2006	Digit images	1.6mn
Hinton & Salakhutdinov	Face images	3.8mn
Salakhutdinov & Hinton	Sem. hashing	2.6mn
Ranzato & Szummer	Text	3mn
Our model		100mn

the model is large, but not otherwise (Lee et al., 2006). There has been a lot of recent work on scaling up DBN and sparse coding algorithms, sometimes with entire research papers devoted to ingenious methods devised specifically for each of these models (Hinton et al., 2006; Bengio et al., 2006; Murray & Kreutz-Delgado, 2006; Lee et al., 2006; Kavukcuoglu et al., 2008).

Meanwhile, the raw clock speed of single CPUs has begun to hit a hardware power limit, and most of the growth in processing power is increasingly obtained by throwing together multiple CPU cores, instead of speeding up a single core (Gelsinger, 2001; Frank, 2002). Recent work has shown that several popular learning algorithms such as logistic regression, linear SVMs and others can be easily implemented in parallel on multicore architectures, by having each core perform the required computations for a subset of input examples, and then combining the results centrally (Dean & Ghemawat, 2004; Chu et al., 2006). However, standard algorithms for DBNs and sparse coding are difficult to parallelize with such "data-parallel" schemes, because they involve iterative, stochastic parameter updates, where any update depends on the previous updates. This makes the updates hard to massively parallelize at a coarse, data-parallel level (e.g., by computing the updates in parallel and summing them together centrally) without losing the critical stochastic nature of the updates. It appears that *fine-grained parallelism* might be needed to successfully parallelize these tasks.

In this paper, we exploit the power of modern graphics processors (GPUs) to tractably learn large DBN and sparse coding models. The typical graphics card shipped with current desktops contains *over a hundred* processing cores, and has a peak memory bandwidth several times higher than modern CPUs. The
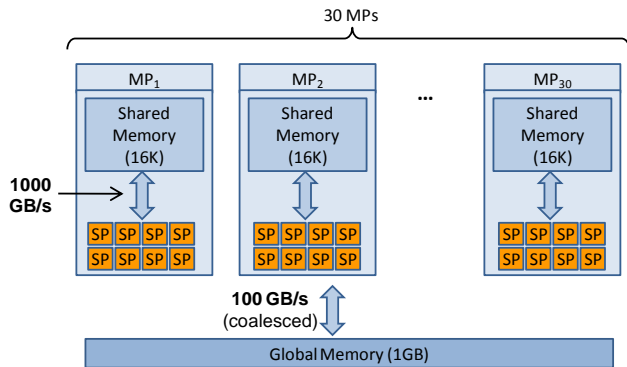

Figure 1. Simplified schematic for the Nvidia GeForce GTX 280 graphics card, with 240 total cores (30 multiprocessors with 8 stream processors each).

hardware can work concurrently with thousands of threads, and is able to schedule these threads on the available cores with very little overhead. Such fine-grained parallelism makes GPUs increasingly attractive for general-purpose computation that is hard to parallelize on other distributed architectures.

There is of course a tradeoff—this parallelism is obtained by devoting many more transistors to data processing, rather than to caching and control flow, as in a regular CPU core. This puts constraints on the types of instructions and memory accesses that can be efficiently implemented. Thus, the main challenge in successfully applying GPUs to a machine learning task is to redesign the learning algorithms to meet these constraints as far as possible. While a thorough introduction to graphics processor architecture is beyond the scope of this paper, we now review the basic ideas behind successful computation with GPUs.

2. Computing with graphics processors

We illustrate the principles of GPU computing using Nvidia's CUDA programming model (Harris, 2008). Figure 1 shows a simplified schematic of a typical Nvidia GPU. The GPU hardware provides two levels of parallelism: there are several multiprocessors (MPs), and each multiprocessor contains several stream processors (SPs) that run the actual computation. The computation is organized into groups of threads, called "blocks", such that each block is scheduled to run on a multiprocessor, and within a multiprocessor, each thread is scheduled to run on a stream processor.

All threads within a block (and thus executing on the same multiprocessor) have shared access to a small amount (16 KB) of very fast "shared memory," and they can synchronize with each other at different points in their execution. All threads also have access to a much larger GPU-wide "global memory" (currently up to 4 GB) which is slower than the shared memory, but is optimized for certain types of simul-

taneous access patterns called "coalesced" accesses. Briefly, memory access requests from threads in a block are said to be coalesced if the threads access memory in sequence (i.e., the k-th thread accesses the k-th consecutive location in memory).[1] When memory accesses are coalesced, the hardware can perform them in parallel for all stream processors, and the effective access speed (between the stream processors and the global memory) is several times faster than the access speed between a CPU and RAM.

Since GPU computation and within-GPU memory accesses themselves are highly parallel, in many algorithms, the main bottleneck arises in transferring data between RAM and the GPU's global memory. For example, the total time taken to multiply two 1000x1000 matrices using our GPU configuration (and a vendor-supplied linear algebra package) is roughly 20 milliseconds, but the actual computation takes only 0.5% of that time, with the remaining time being used for transfer in and out of global memory. A partial solution is to perform memory transfers only in large batches, grouped over several computations. In our example, if we were doing 25 different matrix multiplications and were able to perform memory transfers in large chunks (by transferring all inputs together, and transferring all outputs together), then as much as 25% of the total time is spent in computation. Thus, efficient use of the GPU's parallelism requires careful consideration of the data flow in the application.

3. Preliminaries

We now introduce the unsupervised learning problems we consider in this paper, and analyze the specific issues faced in applying GPUs to those problems. We consider an unsupervised learning task where we are given a large unlabeled dataset $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$, with each input $x^{(i)} \in \mathbb{R}^k$. The goal is to learn a model for the inputs x, and to then apply the model to specific machine learning tasks. For example, each unlabeled input $x^{(i)} \in \mathbb{R}^{900}$ might represent a 30x30 pixel image of a handwritten character (represented as a vector of pixel intensities). We might want to learn a model for the complex 900-dimensional space of inputs, and then use this model to classify new handwritten characters using only very little labeled data.

3.1. Deep Belief Networks

DBNs are multilayer neural network models that learn hierarchical representations for their input data. Hin-

ton et al. (2006) proposed an unsupervised algorithm for learning DBNs, in which the DBN is greedily built up layer-by-layer, starting from the input data. Each layer is learnt using a probabilistic model called a restricted Boltzmann machine (RBM). Briefly, an RBM contains a set of stochastic hidden units h that are fully connected in an undirected model to a set of stochastic visible units x. Assuming binary-valued units, the RBM defines the following joint distribution:

$$P(x, h) \quad \propto \quad \exp\left(\sum_{i,j} x_i w_{ij} h_j + \sum_i c_i x_i + \sum_j b_j h_j\right)$$

where the weights w and biases b and c are parameters to be tuned. The conditional distributions can be analytically computed:

$$P(h_j|x) \quad = \quad \text{sigmoid}(b_j + \sum_i w_{ij} x_i) \qquad (1)$$
$$P(x_i|h) \quad = \quad \text{sigmoid}(c_i + \sum_j w_{ij} h_j) \qquad (2)$$

Maximum likelihood parameter learning for an RBM can be efficiently approximated by contrastive divergence updates (Hinton, 2002), where we start with the unlabeled examples as the visible units, alternately sample the hidden units h and visible units x using Gibbs sampling (Equations 1-2), and update the parameters as:

$$w_{ij} \quad := \quad w_{ij} + \eta\left(\langle x_i h_j\rangle_{\text{data}} - \langle x_i h_j\rangle_{\text{sample}}\right) \quad (3)$$
$$c_i \quad := \quad c_i + \eta\left(\langle x_i\rangle_{\text{data}} - \langle x_i\rangle_{\text{sample}}\right) \qquad (4)$$
$$b_j \quad := \quad b_j + \eta\left(\langle h_j\rangle_{\text{data}} - \langle h_j\rangle_{\text{sample}}\right) \qquad (5)$$

where η is the learning rate, $\langle\cdot\rangle_{\text{data}}$ represents expectations with the visible units tied to the input examples, and $\langle\cdot\rangle_{\text{sample}}$ represents expectations after $T \geq 1$ iterations of Gibbs sampling. Since each update requires a Gibbs sampling operation, and the updates have to be applied over many unlabeled examples to reach convergence, unsupervised learning of the parameters can take several days to complete on a modern CPU.

3.2. Sparse Coding

Sparse coding is an algorithm for constructing succinct representations of input data (Olshausen & Field, 1996). Using our earlier example, if each input $x^{(i)} \in \mathbb{R}^{900}$ represents a handwritten character image, sparse coding attempts to learn that each handwritten character is composed of only a few building blocks, such as pen strokes (instead of 900 arbitrary intensity values). Such a higher-level representation can then be applied to classification tasks, where it leads to good results even with limited labeled data (Raina et al., 2007; Bradley & Bagnell, 2008).

Specifically, given inputs $x \in \mathbb{R}^k$, sparse coding attempts to find basis vectors $b = \{b_1, b_2, \ldots, b_n\}$, $b_j \in \mathbb{R}^k$ such that each input x can be represented as a linear combination of a few basis vectors: $x \approx \sum_j a_j b_j$, where $a_j \in \mathbb{R}$ represents the activation of basis b_j,

---

[1] For simplicity, we ignore certain other technical conditions that are easy to obey in practice. We also omit discussion of two other types of memory—constant and texture memory—that are optimized for other specific types of access patterns that we do not use in our applications.

and most of the a_j values are zero (or, the vector a is sparse). The basis vectors are found by solving the following optimization problem (Lee et al., 2006):

$$\text{minimize}_{b,a} \quad \tfrac{1}{2}\sum_i \|x^{(i)} - \sum_j a_j^{(i)} b_j\|^2 + \beta \sum_{i,j} |a_j^{(i)}|$$
$$\text{s.t.} \quad \|b_j\| \leq 1, \quad \forall j \in \{1,...,n\}$$

where the first term in the objective function encourages good reconstruction ($x^{(i)} \approx \sum_j b_j a_j^{(i)}$), and the second term encourages sparsity by penalizing non-zero activations (Tibshirani, 1996). The optimization problem is *not* jointly convex in both b and a variables, but it is convex in either one of those variables, if the other is kept fixed. This suggests an alternating minimization algorithm with two steps: first, keeping b fixed, we optimize over a, which leads to an L_1-regularized least squares problem, that can be solved using custom-designed solvers (Efron et al., 2004; Lee et al., 2006; Andrew & Gao, 2007). Then, we keep a fixed, and optimize over b using convex optimization techniques (Lee et al., 2006). For problems with high-dimensional inputs and large numbers of basis vectors, the first step is particularly time consuming as it involves a non-differentiable objective function, and the overall learning algorithm can take several days.

4. GPUs for unsupervised learning

Both the above algorithms repeatedly execute the following computations: pick a small number of unlabeled examples, compute an update (by contrastive divergence or by solving a convex optimization problem), and apply it to the parameters. To successfully apply GPUs to such unsupervised learning algorithms, we need to satisfy two major requirements. First, memory transfers between RAM and the GPU's global memory need to be minimized, or grouped into large chunks. For machine learning applications, we can achieve this by storing all parameters permanently in GPU global memory during learning. Unlabeled examples usually cannot all be stored in global memory, but they should be transferred only occasionally into global memory in as large chunks as possible. With both parameters and unlabeled examples in GPU global memory, the updates can be computed without any memory transfer operations, with any intermediate computations also stored in global memory.

A second requirement is that the learning updates should be implemented to fit the two level hierarchy of blocks and threads, in such a way that shared memory can be used where possible, and global memory accesses can be coalesced. Often, blocks can exploit data parallelism (e.g., each block can work on a separate input example), while threads can exploit more fine-grained parallelism because they have access to very fast shared memory and can be synchronized

(e.g., each thread can work on a single coordinate of the input example assigned to the block). Further, the graphics hardware can hide memory latencies for blocks waiting on global memory accesses by scheduling a ready-to-run block in that time. To fully use such latency hiding, it is beneficial to use a large number of independently executing blocks. In some cases, as discussed for sparse coding in Section 6, we can completely redesign the updates to be inherently parallel and require less synchronization between threads.

We thus arrive at the following template algorithm for applying GPUs to unsupervised learning tasks:

Algorithm 1 Parallel unsupervised learning on GPUs

Initialize parameters in global memory.
while convergence criterion is not satisfied **do**
    Periodically transfer a large number of unlabeled examples into global memory.
    Pick a few of the unlabeled examples at a time, and compute the updates in parallel using the GPU's two-level parallelism (blocks and threads).
end while
Transfer learnt parameters from global memory.

---

5. Learning large deep belief networks

We apply Algorithm 1 to learning large DBNs using the contrastive divergence updates in Equations (3-5). The parameters w, c and b for all the DBN layers are maintained permanently in global memory during training. The updates require repeated Gibbs sampling using the distributions in Equations (1-2). These distributions can be rewritten using matrix notation:

$$P(h|x) = \text{vectorSigmoid}(b + w^T x)$$
$$P(x|h) = \text{vectorSigmoid}(c + wh)$$

where vectorSigmoid$(\cdot)$ represents the elementwise sigmoid function, and x, h are vectors containing an element corresponding to each visible and hidden unit respectively. The above computations can be batched together for several examples for further efficiency. The matrix operations can be performed in parallel using optimized linear algebra packages for the GPU, and the sigmoid computation and sampling can be done by a simple parallelization scheme where each block works on a single example, and each thread in the block works on a single element of the example. Finally, once the samples have been generated, the updates can again be applied in parallel using linear algebra packages: e.g., $w := w + \eta \left(\langle x^T h \rangle_{\text{data}} - \langle x^T h \rangle_{\text{sample}} \right)$

We extend our method to learning deep belief networks with "overlapping patches" (Figure 2). This model is most easily understood with hidden and visible units arranged in a 2-D array (e.g., when the input is an
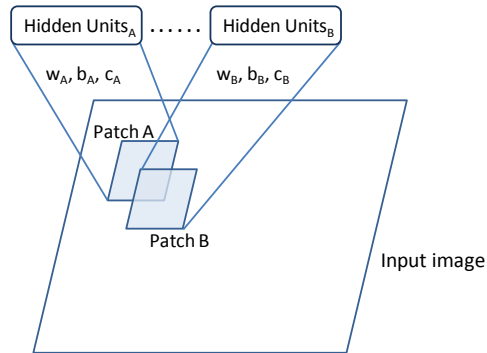
Figure 2. A schematic of the overlapping patches model. Two patches A and B in the input image are shown, with each patch connected to a different set of hidden units. The connections are parameterized by their own sets of parameters w_A, b_A, c_A and w_B, b_B, c_B.

image and each visible unit is a pixel). The input image is tiled by equally-spaced, equal-sized patches (or receptive fields), and each patch is fully connected to a unique group of hidden units. There is no weight sharing in this model, and each connection is parameterized by a free parameter. Because of the overlapping patches, all the parameters in the model depend on each other, making learning hard. However, Gibbs sampling can still be performed in parallel for this model: each visible unit depends on hidden units at many different locations, but the sampling operation $x|h$ can be implemented using only coalesced global memory accesses (implementation details omitted).

These overlapping patch RBMs can be stacked on top of each other, such that the second-layer RBM contains hidden units connected locally to first-layer hidden units, and so on. The resulting deep networks have a very large number of units, but only sparse, local connections, which make learning tractable even for models with more than 100 million parameters.

Experimental Results: We compare our GPU-based algorithm against CPU-based methods using the following multicore hardware:

- **GPU**: Nvidia GeForce GTX 280 graphics card with 1GB memory. Dual-core CPU @ 3.16GHz. Reported results show the total running time (including all computation, memory transfer, etc.).
- **Single CPU**: Single core @ 3.16GHz.
- **Dual-core CPU**: Two cores, each @ 3.16GHz. (Identical machine as for the GPU result.)

The CPU-based method was implemented using two highly optimized multithreaded linear algebra packages: ATLAS BLAS (Whaley et al., 2001) and Goto BLAS (Goto & Van De Geijn, 2008). Consistent with previous results, we found that Goto BLAS was faster (Bengio, 2007), so we report CPU results using it. As input, we used a large dataset of natural

images (van Hateren & van der Schaaff, 1997) and obtained input examples by randomly extracting square image patches of the required size. Following previous work, we used Gaussian visible units and binary hidden units, and trained a sparse RBM by adding an additional penalty term to the objective (Lee et al., 2007)—however, these modifications do not affect the running time results significantly. For learning, we performed one-step contrastive divergence updates using a mini-batch of 192 examples.

Table 2 shows the running time for processing 1 million examples for RBMs of varying size (denoted by number of visible units × number of hidden units). The GPU method is between 12 to 72 times faster. The speedup obtained is highest for large RBMs, where the computations involve large matrices and can be more efficiently parallelized by using a large number of concurrent blocks (which allows the graphics hardware to better hide memory latencies). The largest model in Table 2 has 45 million parameters, and our GPU method can update these parameters using a million examples in about 29 minutes. In comparison, our multicore CPU takes more than a day per million examples. Since we would ideally want to use tens of millions of training examples for learning such a large model, the CPU method is impractical for such tasks.

Table 3 shows a similar running time comparison for two "overlapping patch" models (see table caption for details). The GPU method is about 10 times faster than the dual-core CPU. This speedup is somewhat lower than the speedup observed for a fully connected RBM (Table 2), because Gibbs sampling in the overlapping patch model requires many operations involving small matrices (one weight matrix per patch), instead of only a few operations involving large matrices. Using the overlapping patch model, we can learn a four-layer DBN with 96 million parameters, and 25600, 82944, 8192, 4608 and 1024 units respectively in the input layer and the four successive hidden layers. Such models are at least an order of magnitude larger than previously published work on DBNs.

Finally, we note that the overlapping patches model can be modified to share parameters in all patches, such that, for example, $w_A = w_B$ in Figure 2. If overlapping patches are tiled one pixel apart, this model is identical to the convolutional RBM model (Desjardins & Bengio, 2008; Lee et al., 2009). Contrastive divergence learning in this model can be implemented by using convolutions to perform the Gibbs sampling operation $h|x$. For small to medium filter (patch) sizes, spatial convolution can be implemented very efficiently using GPUs, by having each block read a filter into shared memory, then reading the input image column-

Table 2. Average running time in seconds for processing 1 million input examples for learning an RBM, with contrastive divergence updates applied in batches of 192 examples each. The size of the RBM in each column is denoted by the number of visible units × number of hidden units. The GPU speedup is computed w.r.t. the fastest CPU-based result.

Package	Architecture	576x1024	1024x4096	2304x16000	4096x11008
Goto BLAS	Single CPU	563s	3638s	172803s	223741s
Goto BLAS	Dual-core CPU	497s	2987s	93586s	125381s
GPU		38.6s	184s	1376s	1726s
GPU Speedup		**12.9x**	**16.2x**	**68.0x**	**72.6x**

Table 3. Average time in seconds for processing 1 million examples for the overlapping patch model, with contrastive divergence updates applied in batches of 192 examples each. The model size in each column is denoted by the number of visible units × number of hidden units (but note that the units are not fully connected). The two models were created by taking 144x144 pixel and 192x192 pixel inputs respectively; the size of each patch is 24x24 pixels, there are 192 hidden units connected to each patch, and neighboring patches are 8 pixels apart. Overall, the models have 28 million and 54 million free parameters respectively.

Package	Arch.	20736x49152	36864x92928
Goto	Single CPU	38455s	77246s
Goto	Dual-core	32236s	65235s
GPU		3415s	6435s
GPU Speedup		**9.4x**	**10.1x**

by-column into shared memory, and finally aggregating the output elements affected by that filter and that input image column. It can be shown that by ordering operations in this way, we use only fast shared memory accesses and coalesced global memory accesses.[2] For example, on computing the convolution of 32 128x128 images with 32 16x16 filters, our GPU implementation of spatial convolution (including the time to transfer images/filters into GPU memory) is over 100 times faster than either spatial convolution implemented in C or FFT-based convolution in Matlab.

6. Parallel sparse coding

We now consider the sparse coding optimization problem discussed in Section 3.2. Following the template in Algorithm 1, we maintain the basis parameters b permanently in global memory, and transfer input examples to GPU global memory periodically in large batches. Following the alternating minimization method, each update itself consists of two steps: the first, simpler part of the update involves optimizing over b, given fixed a:

$$\text{minimize}_b \quad \sum_i \|x^{(i)} - \sum_j a_j^{(i)} b_j\|^2 \quad \text{s.t.} \|b_j\| \leq 1, \forall j$$

We solve this problem using projected gradient descent, where we follow the gradient of the quadratic objective function, and project at each step to the

feasible set.[3] This method is guaranteed to converge to the optimal b and can be straightforwardly implemented using a GPU linear algebra package.

The other part of the update involves optimizing over a, given fixed b. Since the activation $a^{(i)}$ for each example $x^{(i)}$ is now independent of the activations for other examples, it suffices to consider the following canonical L_1-regularized least squares problem for a single input example x:

$$\text{minimize}_a \quad \frac{1}{2}\|x - \sum_j a_j b_j\|^2 + \beta \sum_j |a_j| \quad (6)$$

The objective function is not differentiable because of the second term. This problem has recently received wide attention because of its robust feature selection properties (Tibshirani, 1996; Ng, 2004), and custom algorithms have been designed to solve it (Efron et al., 2004; Lee et al., 2006; Andrew & Gao, 2007). Some of these algorithms use sparse linear algebra operations to achieve efficiency. We instead present a very different algorithm that is inherently parallel and thus uses the GPU hardware more efficiently.

6.1. Parallel L_1-regularized least squares

Our algorithm is based on the observation that in the optimization problem in Equation (6), if we vary only one of the activations a_j, while keeping the other activations fixed, the optimal value a_j^* can be easily computed (Friedman et al., 2007). Letting B be a matrix with b_j as its j-th column, and $r_j = b_j^T b_j$:

$$a_j^* = \begin{cases} 0 & \text{if } |g_j - r_j a_j| \leq \beta \\ (-g_j + r_j a_j + \beta)/r_j & \text{if } g_j - r_j a_j > \beta \\ (-g_j + r_j a_j - \beta)/r_j & \text{if } g_j - r_j a_j < -\beta \end{cases}$$

where $g = \nabla_a \frac{1}{2}\|x - \sum_j a_j b_j\|^2 = B^T B a - B^T x$. The updates can be efficiently performed in parallel by having thread j compute just one coordinate a_j^*. Further, since we usually batch several examples together, we can precompute the matrix $B^T B$, the vector $B^T x$ and the vector r once in parallel, store the result in global memory, and perform only efficient accesses to compute all a_j^* values.[4]

---

[2]For larger filter sizes FFT-based convolution is generally better, and a GPU FFT package can be used.

[3]The projection operation is particularly simple: for each basis vector b_j, if $\|b_j\| > 1$ then rescale b_j to have norm 1, otherwise keep b_j unchanged.

[4]To see why, note that to compute a_j^*, thread j needs to

Thus, we propose the following iterative algorithm: at each iteration, starting at the current activation values $a = \hat{a}$, we compute all the optimal coordinate values a_j^* *in parallel* as outlined above. Then, we perform a line search in the direction of vector $d = a^* - \hat{a}$. The line search consists of finding a step size $t > 0$ such that the value of the objective function at the point $a = \hat{a} + td$ is lower than the value at $a = \hat{a}$. This line search, including the function evaluations, can be run in parallel.[5] We then move to the new point $a = \hat{a} + td$, and iterate. We declare convergence when the objective value decreases by less than a 10^{-6} fraction of the previous objective value.

Since the direction d is a nonnegative linear combination of descent directions along the coordinate axes: $d_j = a_j^* - \hat{a}_j$, d must itself be a descent direction for the objective function. Thus, at each iteration, a step size $t > 0$ can always be found that reduces the value of the objective function, and the overall algorithm is guaranteed to converge to the optimal solution.

This algorithm uses fine-grained parallelism by having each thread compute just one coordinate of the solution. Such highly multithreaded execution is especially well-suited for graphics processors, as the hardware is able to hide memory latency (for threads blocked on memory accesses) by scheduling other threads that are not blocked on memory accesses, and leads to high utilization of the available cores.

Experimental Results: We again compare our method against a multicore CPU baseline (Lee et al., 2006). We used optimized Matlab code provided by Lee et al. For the CPU multicore results, we executed the same Matlab code with multithreading enabled.

Table 4 shows the running time for applying sparse coding basis updates (including both basis and activation optimization) for $m = 5000$ examples, with mini-batches of 1000 examples. Each example $x \in \mathbb{R}^{1024}$

compute $g_j - r_j a_j = \sum_t (B^T B)_{tj} a_t - (B^T x)_j - r_j a_j$. Consider the elements thread j accesses: (i) $(B^T B)_{tj}$: Accesses can be coalesced if $B^T B$ is stored in row-major order. (ii) By maintaining a in shared memory, all threads can access the same element a_t simultaneously, as well as access the elements a_j that are different for each thread. (For the interested reader, we add that this avoids "bank conflicts" in shared memory. See CUDA reference manual for details.) (iii) $(B^T x)_j$ and r_j: Can be coalesced as thread j accesses the j-th location.

[5]Details: By substituting $a = \hat{a} + td$ in the original objective function, the line search reduces to minimizing a 1-D function of the form $f(t) = \alpha_2 t^2 + \alpha_1 t + \alpha_0 + \beta\|\hat{a} + td\|_1$, where the values $\alpha_2, \alpha_1, \alpha_0$ can be computed in parallel. For the 1-D line search over $f(t)$, we simply try a fixed set of positive step sizes, and pick the largest step size that reduces the value of the objective function.

Table 4. Average running time for updating sparse coding parameters on 5000 input examples. The GPU speedup is computed w.r.t. the fastest CPU-based result. The sparsity value refers to the average percentage of the 1024 activations that were nonzero at the optimal solution; different sparsity was obtained by using different β values. Note that 3-10% is a reasonable range as it corresponds to around 30 to 100 nonzero activations per input example.

Method	Sparsity≈3%	6%	10%
Single CPU	215s	403s	908s
Dual-core	191s	375s	854s
GPU	37.0s	41.5s	55.8s
Speedup	**5.2x**	**9.0x**	**15.3x**

was obtained via a randomly sampled 32x32 pixel natural image patch. We used $n = 1024$ basis vectors, initialized randomly. The majority of sparse coding time in Lee et al.'s method is taken by the activation learning step, especially when many activations are nonzero at the optimum. By effectively parallelizing this step, our GPU method is up to 15 times faster than a dual-core implementation.

7. Discussion

Graphics processors are able to exploit finer-grained parallelism than current multicore architectures or distributed clusters. They are designed to maintain thousands of active threads at any time, and to schedule the threads on hundreds of cores with very low scheduling overhead. The map-reduce framework (Dean & Ghemawat, 2004) has been successfully applied to parallelize a class of machine learning algorithms (Chu et al., 2006). However, that method relies exclusively on data parallelism—each core might work independently on a different set of input examples—with no further subdivision of work. In contrast, the two-level parallelism offered by GPUs is much more powerful: the top-level GPU blocks can already exploit data parallelism, and GPU threads can further subdivide the work in each block, often working with just a single element of an input example.

GPUs have been applied to certain problems in machine learning, including SVMs (Catanzaro et al., 2008), and supervised learning in convolutional networks (Chellapilla et al., 2006). To continue this line of work, and to encourage further applications of deep belief networks and sparse coding, we will make our source code publicly available.

References

Andrew, G., & Gao, J. (2007). Scalable training of L_1-regularized log-linear models. *International Conference on Machine Learning* (pp. 33–40).

Banko, M., & Brill, E. (2001). Scaling to very very large corpora for natural language disambiguation. *Annual Meeting of the Association for Computational Linguistics* (pp. 26–33).

Bengio, Y. (2007). Speeding up stochastic gradient descent. *Neural Information Processing Systems Workshop on Efficient Machine Learning.*

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006). Greedy layer-wise training of deep networks. *Neural Information Processing Systems* (pp. 153–160).

Bradley, D., & Bagnell, J. A. (2008). Differentiable sparse coding. *Neural Information Processing Systems* (pp. 113–120).

Brants, T., Popat, A. C., Xu, P., Och, F. J., & Dean, J. (2007). Large language models in machine translation. *Conference on Empirical Methods in Natural Language Processing (EMNLP-CoNLL).*

Catanzaro, B. C., Sundaram, N., & Keutzer, K. (2008). Fast support vector machine training and classification on graphics processors. *International Conference on Machine Learning* (pp. 104–111).

Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing. *International Workshop on Frontiers in Handwriting Recognition.*

Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G. R., Ng, A. Y., & Olukotun, K. (2006). Map-reduce for machine learning on multicore. *Neural Information Processing Systems* (pp. 281–288).

Dean, J., & Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. *Operating System Design and Implementation* (pp. 137–150).

Desjardins, G., & Bengio, Y. (2008). Empirical evaluation of convolutional RBMs for vision. *Tech Report.*

Efron, B., Hastie, T., Johnstone, I., & Tibshirani, R. (2004). Least angle regression. *Ann. Stat.*, *32*, 407.

Frank, D. (2002). Power-constrained CMOS scaling limits. *IBM Jour. of Res. and Devel.*, *46*, 235–244.

Friedman, J., Hastie, T., Hfling, H., & Tibshirani, R. (2007). Pathwise coordinate optimization. *Ann. App. Stat.*, *2*, 302–332.

Gelsinger, P. (2001). Microprocessors for the new millennium: Challenges, opportunities and new frontiers. *ISSCC Tech. Digest*, 22–25.

Goto, K., & Van De Geijn, R. (2008). High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, *35*, 1–14.

Harris, M. (2008). Many-core GPU computing with NVIDIA CUDA. *Int. Conf. Supercomputing* (p. 1).

Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, *14*, 1771–1800.

Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*, 1527–1554.

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*, 504–507.

Kavukcuoglu, K., Ranzato, M., & LeCun, Y. (2008). Fast inference in sparse coding algorithms with applications to object recognition. *NYU Tech Report.*

Lee, H., Battle, A., Raina, R., & Ng, A. Y. (2006). Efficient sparse coding algorithms. *Neural Information Processing Systems* (pp. 801–808).

Lee, H., Chaitanya, E., & Ng, A. Y. (2007). Sparse deep belief net model for visual area V2. *Neural Information Processing Systems* (pp. 873–880).

Lee, H., Grosse, R., Ranganath, R., & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *International Conference on Machine Learning (to appear).*

Murray, J. F., & Kreutz-Delgado, K. (2006). Learning sparse overcomplete codes for images. *J. VLSI Signal Processing Systems*, *45*, 97–110.

Ng, A. Y. (2004). Feature selection, L_1 vs. L_2 regularization, and rotational invariance. *International Conference on Machine Learning* (pp. 78–85).

Olshausen, B. A., & Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images.*Nature*, *381*, 607–609.

Raina, R., Battle, A., Lee, H., Packer, B., & Ng, A. Y. (2007). Self-taught learning: Transfer learning from unlabeled data. *International Conference on Machine Learning* (pp. 759–766).

Ranzato, M. A., & Szummer, M. (2008). Semi-supervised learning of compact document representations with deep networks. *International Conference on Machine Learning* (pp. 792–799).

Salakhutdinov, R., & Hinton, G. (2007). Semantic Hashing. *SIGIR Workshop on Information Retrieval and Applications of Graphical Models.*

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *J. R. Stat. Soc. B.*, *58*, 267–288.

van Hateren, J. H., & van der Schaaff, A. (1997). Independent component filters of natural images compared with simple cells in primary visual cortex. *Royal Soc. Lond. B*, *265*, 359–366.

Whaley, R. C., Petitet, A., & Dongarra, J. J. (2001). Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, *27*, 3–35.