Using GPUs in Machine Learning

Author
Geetika Bansal
Computer Science Department
San Jose State University
San Jose, CA 95112
669-222-9632
geetikagarg07@gmail.com

ABSTRACT

Machine learning algorithms have been known to perform better with more free parameters - to tune - and more training data. But learning algorithms are often too slow for large scale applications and thus the size of training models (free parameters) and data is limited in practice. So usage of GPUs to improve the speeds of these algorithms has attracted a lot of attention recently. I focused on a couple of papers analyze the performance improvements by adapting the learning algorithms for GPUs. I ran the experiments using the source code provided publicly under Theano framework

1. INTRODUCTION

I considered well known unsupervised learning model, deep beliefs networks. 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 considered in the literature generally contain many fewer free parameters or are trained on a fraction of the available input examples. 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). But using more parameters or more training data, might produce very significant performance benefits. 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 multi-core 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 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, the power of modern graphics processors (GPUs) is exploited to tractably learn large DBN. 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 as shown in Figure 2. Also,they have large number of ALUs which help in different and complex computations as shown in figure 1. The 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 trade off—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.

Figure 1: GPUs have more ALUs than CPUs

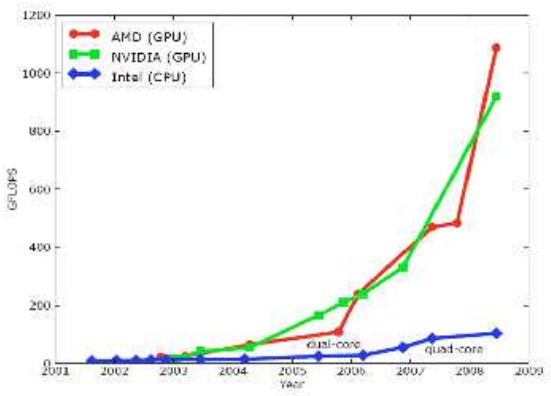

Figure 2: peak memory bandwidth

2. COMPUTATIONS WITH GPUs

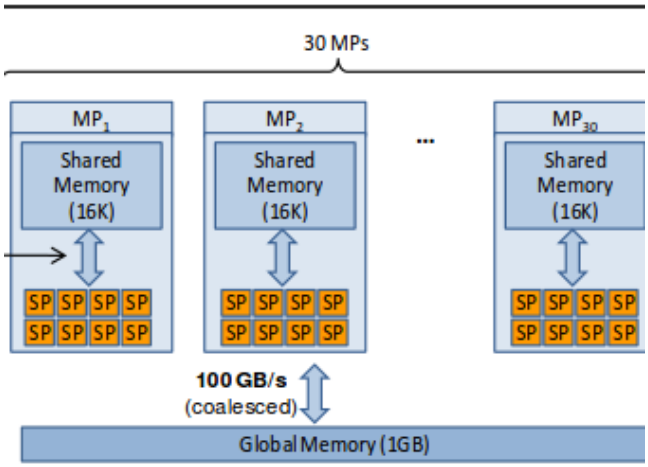

Figure 3: Nvidia GPU

The principles of GPU computing are illustrated using Nvidia's CUDA programming model (Harris, 2008). Figure 3 and Figure 4 shows a simplified schematic model 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 application is distributed into number of "grids" and 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 the threads have their private or the "local memory". This memory is the fastest memory in terms of access and is private to every thread. 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," or also known as "inter-block 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 simultaneous 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). 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. So, if we are doing 25 different matrix multiplications, we should be able to perform memory transfers in large chunks (by transferring all inputs together,and transferring all outputs together), such that 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.

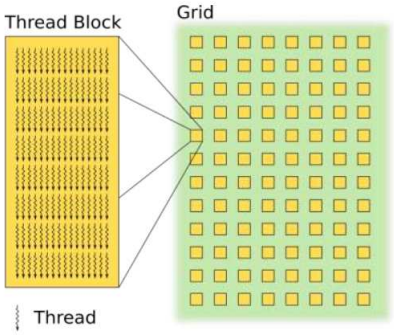

Figure 4: Programming Model GPU

3. DEEP BELIEFS NETWORK AND BOLTZMANN MACHINE

DBN are multilayer neural network models that learn hierarchical representations for their input data. Hinton 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).

3.1 Restricted Boltzmann Machine

Boltzmann Machines (BMs) are the neural networks which is a generative model I.e after the iterations , the model or the trained system is almost alike as that of the input samples. BM is a particular form of long-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a

non-parametric one), we consider that some of the variables are never observed (they are called hidden). By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM). Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections. That is, no visible node is connected to other visible node and no hidden node is connected to other hidden node. A graphical depiction of an RBM is shown below in Figure 5. RBMs are used in unsupervised learning.

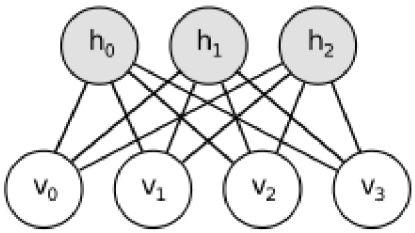

Figure 5: RBM

The energy function $E(v,h)$ of an RBM is defined as:

$$E(v,h) = -b'v - c'h - h'Wv$$

Where W represents the weights connecting hidden and visible units and b,c are the offsets of the visible and hidden layers respectively .

This translates directly to the following free energy formula:

$$\mathcal{F}(v) = -b'v - \sum_i \log \sum_{h_i} e^{h_i(c_i + W_i v)}.$$

Because of the specific structure of RBMs, visible and hidden units are conditionally independent given one-another. Using this property, we can write:

$$p(h|v) = \prod_i p(h_i|v)$$
$$p(v|h) = \prod_j p(v_j|h).$$

RBMs with binary units:

In the commonly studied case of using binary units (where **vj** and $h_i \in \{0,1\}$), a probabilistic version of the usual neuron activation function:

$$P(h_i = 1|v) = sigm(c_i + W_i v)$$
$$P(v_j = 1|h) = sigm(b_j + W'_j h)$$

The free energy of an RBM with binary units further simplifies to:

4. GPUs FOR UNSUPERVISED LEARNING

The algorithm 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 ransfers 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.

Following template algorithm can be used to apply GPUs to unsupervised learning tasks:

Parallel unsupervised learning on GPUs

1) 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 EXPERIMENTAL RESULTS

I compared GPU-based algorithm against CPU-based methods using the following multi core hardware:

GPU used: GeForce GT 720M

CPU used: Intel(R) Core(TM) i5-3337U CPU @1.80GHz

The restricted boltzmann machine was trained under Theano framework. The data set for which it was trained was the publicly available dataset from MNIST database which has 50,000 vectors of size 784 each. These vectors are basically, images of handwritten characters. Data is taken is batches and trained one by one using the weight matrix or the filter. The filters learnt by the model can be visualized. This amounts to plotting the weights of each unit as a gray-scale image (after reshaping to a square matrix). Below(Figure 6) is a filter after the 15 epochs(cycle):

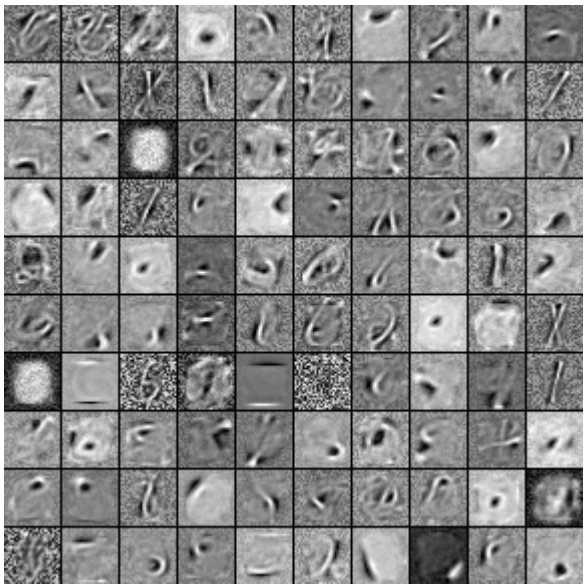

Figure 6: Filter after 15 epochs

Now, this learnt system can be used with data and sampling of the data can be done using this trained system . The samples generated by the RBM after training will look like as in Figure 7.

Figure 7: Sample generated

TABLE 1: Comparison between CPU and GPU time for learning.

Hardware used	Complexity (no.of hidden nodes)	Time take to learn(in seconds)
CPU	500	1155.988942
GPU	500	2300.276344
CPU	2000	5459.449917
GPU	2000	2600.956223
CPU	5000	11382.501643
GPU	5000	5381.133367

While doing this experiment, I changed the complexity of the model by changing the no. of hidden nodes in the model and then computed the time taken to train the system on CPU and on GPU. From the table 1, we can see that the speed up of the GPU is around 2.1. When we plot this in a graph, the comparison between the time taken by CPU and GPU is shown in Figure 8.and the speed up of the GPUs is shown in figure 9.

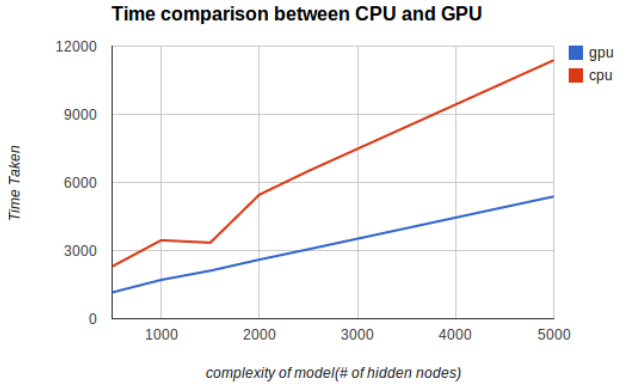

Figure 8: Comparison of GPU and CPU

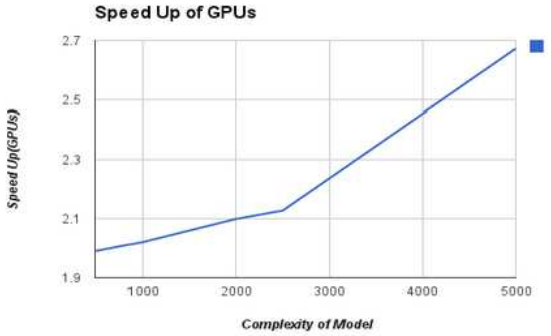

Figure 9: Speed-up of GPU

6. SUMMARY

Graphics processors are able to exploit fined-grained parallelism than current multi core 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 (Chuet 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).

7. REFERENCES

[1] http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture3.pdf

[2] https://www.udacity.com/course/viewer#!/c-cs344/l-95446457/m-95293899

[3] http://deeplearning.net/software/theano/

[4] http://ai.stanford.edu/~ang/papers/icml09-LargeScaleUnsupervisedDeepLearningGPU.pdf

[5] http://deeplearning.net/tutorial/gettingstarted.html

[6] http://www.deeplearning.net/tutorial/rbm.html