Parallel Training of a Multi-Layer Perceptron on a GPU

Chris Oei, Gerald Friedland, and Adam Janin

TR-09-008

October 2009

Abstract

This report presents a parallelized implementation of a back propagation training algorithm running on the NVIDIA graphics card GPU and compares it with several CPU implementations, including highly optimized software packages that do their work on the CPU.

Parallel Training of a Multi-Layer Perceptron on a GPU

Chris Oei, Gerald Friedland, Adam Janin

October 7, 2009

Abstract

This report presents a parallelized implementation of a back propagation training algorithm running on the NVIDIA graphics card GPU and compares it with several CPU implementations, including highly optimized software packages that do their work on the CPU.

1 Introduction

Speed has always been the Achilles heel of artificial neural networks. For many problems, MLPs are too slow to provide viable solutions. However, these algorithms are highly parallelizable, and modern-day graphics processing units (GPUs) can offer tremendous speed gains at relatively low cost. We compare the speeds of a GPU implementation of a back propagation learning algorithm against highly optimized CPU implementations.

In the sections that follow, we (2) review related work by other researchers, (3) outline our approach and hardware setup, (4) normalizing the file formats across neural network tools, (5) discuss floating point representation problems, and (5) provide results of timing tests.

2 Related Work

QuickNet, developed at ICSI [1], is a highly optimized multi-threaded implementation of a MLP designed specifically for speech-related learning tasks. As such, both the program and its data files have features that are important for speech processing, but are not typically used in the typical benchmarks and sample problems for MLPs. It takes some processing to convert the file format used in PROBEN1 into the QuickNet pfile format.

Aside from QuickNet and the CUDA back propagation software, the fastest implementation known to the authors is the Fast Artificial Neural Network Library (FANN) by Steffen Nissen [4]. Most research on back propagation optimization occurred around that time; this was before GPUs became popular enough and advanced enough to make high-performance computing easy to do.

At that same time, Moore's law still held true; CPUs were getting faster exponentially, and there was less of a need to go through the trouble to parallelize. Today, however, CPUs have stopped getting faster, and performance increases can only be achieved by parallelization.

FANN relies on optimized inner loops and cache optimization and (in their fixed point implementation) and a lookup function for doing sigmoids.

We used the FANN version 2.0.0 Debian / Ubuntu binary packages [4] to test against our benchmarks.

We also benchmarked a popular toolkit for machine learning, The Waikato Environment for Knowledge Analysis (WEKA) [2, 3] which was written in Java. Although this toolkit is not optimized, we thought it was worthwhile comparing; since it is popular and easy to use, it may be a good choice for problems smaller than a certain size. For the very largest problems we tested, however, WEKA was two orders of magnitude slower than the CUDA implementation.

3 Our Approach

Our approach relies on two key items. The first is the highly parallelized matrix operations (BLAS) provided by NVIDIA's CUDA (CUBLAS). We use primarily the sgemm (level 3) functions, which multiply a 2 dimensional matrix by a 2 dimensional matrix.

The second is a fast exponential algorithm used in the sigmoid nonlinearities and softmax functions. Although for very large networks, these $O(N)$ tasks are dominated by the $O(N^2)$ matrix multiplies, for most practical pursposes these exponentials take a significant fraction of the total time.

Whereas FANN under the fastest settings uses a piecewise linear activation function, and QuickNet uses an approximation that relies on the IEEE float format [6], we use the CUDA exponentiation. If we have enough GPU cores, this step is an $O(1)$ computation instead of $O(N)$.

methodology

Table 1: Comparing Neural Net Packages

Neural Network	Matrix Multiplies	Exponentiation
FANN	internal library	internal library
QuickNet	ATLAS BLAS	Schraudolph
mlp cuda	CUBLAS	CUDA provided

3.1 Our Hardware and Setup

- NVIDIA GTX 280 graphics card.
- Lenovo D10

  - 2 GB RAM
  - Dual Quad core Xeon 2GHz
- Ubuntu 8.04 32-bit

Note that we are using a 32-bit operating system on our 64-bit machine. At the time we were doing the experiment, we had stability problems with the 64-bit version of CUDA, and so we downgraded our operating system and the CUDA software to the 32-bit versions.

3.2 Power Supply

One of the trickier parts of using high-end graphics cards is that they draw a lot of power and may overwhelm the computer's power supply. In order to be most cost effective, entry-level desktops are often configured with a power supply that is sufficient for powering the pre-installed components and little else.

And even if the power supply is capable of handling the load, the graphics card may have multiple power cables to spread the load; each power cable and power rail must be able to hold up its share of the load. Also, the graphics card may have power connectors that are incompatible with the motherboard's power connectors, and adapters must be used; furthermore, some adapters will take two power sockets on the motherboard and convert them to a single power connector on the graphics card.

Given these complexities, we were initially uncertain that all of our components – from the power supply to the rails to the connectors – could handle the load from the graphics card. Our bios had a temperature monitoring system, and we checked that temperature under various GPU loads to make certain we did not overheat. We did not do lengthy unsupervised computations until we were certain.

3.3 Benchmark

We chose the PROBEN1 benchmark [5] as our primary benchmark because it has been used by a number of other machine learning software packages, and because the author has a wide range of "real-world" problems as test cases.

The PROBEN1 benchmark, however, is too small for the economies of scale of CUBLAS and QuickNet to show their power. Much of the optimization for mlp-cuda and QuickNet are for the $O(N^2)$ component, which is not entirely dominant when N is around 10 or so. The $O(N)$ components include exponentiation.

In order to explore larger net and data sizes, we used the Tandem 29 corpus, which mapped to a network of 378 input neurons and 71 output neurons. We chose to have a single hidden layer consisting of 2048 neurons, and a bunch size of 8192.

4 Format Normalization

Converting the PROBEN1 data file format into the formats that FANN and WEKA use was straightforward. The FANN format is described in [4] and the WEKA format is described in [3, 2].

The pfile format has two columns that are not typically used by other neural network software: sentence number and frame number. In order to convert the PROBEN1 data to the pfile format, we assume that the data is a single sentence composed of many, many frames. The opposite choice (assuming many, many sentences with a single frame each) does not produce the same behavior, since QuickNet shuffles frames within a sentence, but does not shuffle sentences.

In the PROBEN1 test set, there are two different types of problems. QuickNet handles each rather differently. These types are:

1. classification Each input frame maps to exactly one category, and there are a finite number of categories. For example: detecting a person's gender from the sound of their voice, determining whether a mushroom is edible or not, and the classical 2-bit XOR problem. For these problems, we use QuickNet's "hard target" training.

2. estimation each input frame maps to a vector of real numbers. For example, converting Fahrenheit to Celsius, or calculating the fuel efficiency of a car. For these problems, we use QuickNet's "soft target" training.

Mathematically speaking, the difference between the two types are blurred, since we can create an approximate representation of a real number using a large but finite number of categories; and of course that is what a computer does. However, estimation problems typically have some sort of smoothness in the mapping, such that the accuracy can be represented by the L2 norm. Categorization problems typically do not have such a topology – all items in the set are equidistant.

For classification problems, we feed the input data as a feature file and the target data as a label file.

For estimation problems, we feed the input data as a feature file, and the target data as a separate feature file. Label files can only contain discrete targets, and are therefore unsuitable for estimation problems.

Now that we have described the setup of our experiment, we discuss some of the difficulties

5 Floating Point Representation

Back propagation is, for the most part, a numerically robust algorithm. A small amount of noise caused by roundoff errors do not typically affect the result very much; for this reason, we are able to use single-precision floats instead of double precision. There are three major exceptions, all of which involve overflows or underflows of some sort.

5.1 NaN

As we increase the bunch size, the system appears to become more numerically unstable. Sometimes, this reduces the final score that the network gets, and sometimes this causes the results to be partially or even entirely invalid.

Part of the problem is that floating point overflows cause NaNs to appear in the calculations. And since a NaN multiplied, divided, added, or subtracted from another NaN results in another NaN, these errors eventually spread throughout all the weights and biases of the network, rendering the results unusable.

Lowering the learning rate helps avoid these NaNs, but of course this can increase the number of epochs it takes to learn the pattern.

We believe that the root cause of the NaNs are the exponential functions we use in calculating the sigmoid and softmax nonlinearities. We could solve the problem by capping the maximum value of the exponential function that we use, and this is what we tried at first. However, this approach come with a price: it makes $e^x = e^y$ for all x and y greater than the cutoff; this distorts the parameter space for learning and it also causes problems in the scoring of softmax problems.

Detecting these NaNs is another issue. If we attempt to detect them at all times in all calculations, we will take a performance hit. If we ignore them entirely, we can end up with spurrious results.

5.2 Subnormals

Another issue can cause the neural network to slow dramatically, even though it gets the correct results. Floating point numbers too small to be represented are flagged by the CPU as subnormals. They are for practical purposes equal to zero and behave mathematically as such, yet they take far longer than ordinary floats to perform computations on.

We can deal with this problem in several different ways. We could once again choose a cutoff value and set all scalar values smaller than that cutoff to zero. Or we could also set a compiler flag which turns off the processing of subnormals in the CPU.

5.3 Saturation

In the reverse propagation, the deltas (errors) are multiplied by a nonlinearity as they pass through the hidden layer. If y is the value of the sigmoid, the factor is:

$$y(1 - y)$$

But if the argument of the sigmoid is too large y will be either 0 or 1, the factor will be 0, and therefore no more "learning" will take place for that particular hidden unit. This means that once a hidden unit saturates, it will no longer learn properly.

5.4 Re-implementation of softmax

For the reasons mentioned in the previous sections, we came up with a slightly different method of calculating softmax.

To get some intuition on how NaNs arise, here are some example calculations. For x sufficiently large:

$$expf(x) = inf$$
$$expf(x) + expf(x) = inf$$
$$expf(x) - expf(x) = NaN$$
$$expf(x)/expf(x) = NaN$$
$$1/expf(x) = 0$$

Softmax is typically calculated using:

$$s_n = \frac{exp(x_n)}{\sum_i exp(x_i)}$$

If $exp(x_i) = inf$ for any i, then some of these values will be NaN. We propose to fix this problem using the following.

Let x_{max} be the largest value of x in the set. If we divide each of the terms in both the numerator and the denominator in the equation above, we get:

$$s_n = \frac{exp(x_n - x_{max})}{\sum exp(x_i - x_{max})}$$

Although this approach is susceptible to a slow-down due to subnormals, it is well-behaved in that it does not generate NaNs and the system assigns $s_n \approx 1$ and the rest to zero if x_n is much larger than all the other x values.

6 Results

We measured the speed of our network in millions of connection updates per second (MCUPS). The results are in Table 1. Note that the performance gains seen in the CUDA implementation are only visible in the largest problems. For small problems, CUDA's overhead (such as initializing the kernel and copying memory back and forth between CPU memory and GPU memory) wipe out the performance gains of the fast matrix multiply and exponentiation.

The code that we used, as well as our test data, can be downloaded from http://www.icsi.berkeley.edu/chrisoei.

From these results, we see that QuickNetoutperforms FANNby signifcant amounts only on the Tandem 29, mushroom, and gene problems above; these are the three largest test sets.

Timing Tests

Table 2: MCUPS

Test suite	test name	WEKA	FANN	QuickNet	QuickNet(8 threads)	mlp-cuda
PROBEN1	building	18	64	82	85	
	cancer	16	33	39	31	2
	card	26	153	223	204	41
	diabetes	11	31	38	29	2
	flare	23	98	130	112	
	gene	28	214	497	769	198
	glass	8	36	41	39	3
	heart	29	122	178	164	21
	horse	28	159	193	174	46
	mushroom	28	212	544	893	201
	soybean	28	129	307	289	136
	thyroid	26	70	135	146	9
Tandem 29			226	1355	5251	11000

7 Conclusion and Future Work

The problems we faced with NaNs, infs, subnormals, and saturation are not specifically CUDA or GPU problems; these problems exist in the CPU implementations as well. However, a solution that works for the CPU implementation may not work in CUDA, as CUDA floats are not entirely IEEE-compliant. Also, since we are using the GPU implementation to explore larger data sets than we could before, we are more likely to run into these problems. Back propagation involves summing over matrix elements along a particular dimension, and then exponentiating that sum; the more elements we add together, the more likely we will spill over the floating point capacity. Part of our future work will involve finding ways to eliminate these floating point problems.

References

[1] P. Farber. Quicknet on multispert: fast parallel neural network training. Technical report, Tech. Rep. TR-97-047, ICSI, 1997, 1997.

[2] S.R. Garner. Weka: The waikato environment for knowledge analysis. In *Proc. of the New Zealand Computer Science Research Students Conference*, pages 57–64. Citeseer, 1995.

[3] G. Holmes, A. Donkin, and I.H. Witten. Weka: A machine learning workbench. In *Proceedings of the Second Australia and New Zealand Conference on Intelligent Information Systems*, pages 357–361. Citeseer, 1994.

[4] S. Nissen. Implementation of a fast artificial neural network library (fann). *Report, Department of Computer Science University of Copenhagen (DIKU)*, 31, 2003.

[5] L. Prechelt. Proben1—a set of neural network benchmark problems and benchmarking rules. *Fakultat fur Informatik, Universit at Karlsruhe, 76128*, 1994.

[6] N.N. Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999.