CUDA-accelerated Genetic Feedforward-ANN Training for Data Mining

Catalin Patulea, Robert Peace, and James Green
School of Systems and Computer Engineering
Carleton University
Ottawa, Canada K1S 5B6
cat@vv.carleton.ca, *rpeace@sce.carleton.ca*, *jrgreen@sce.carleton.ca*

April 19, 2010

Abstract

We present an implementation of genetic algorithm (GA) training of feedforward artificial neural networks (ANNs) targeting commodity graphics cards (GPUs). By carefully mapping the problem onto the unique GPU architecture, we achieve order-of-magnitude speedup over a conventional CPU implementation. Furthermore, we show that the speedup is consistent across a wide range of data set sizes, making this implementation ideal for large data sets. This performance boost enables the genetic algorithm to search a larger subset of the solution space, which results in more accurate pattern classification. Finally, we demonstrate this method in the context of the 2009 UC San Diego Data Mining Contest, achieving a world-class lift on a data set of 94682 e-commerce transactions.

1 Introduction

Genetic algorithms (GAs) are a stochastic, evolutionary approach to machine learning for pattern classification. While greedy methods will get stuck at local extrema, GAs are theoretically capable of asymptotically reaching the global optimum [1]. However, because they require the evaluation of approximately 50 to 100 classifiers at each iteration, and the algorithm is repeated for several thousand iterations, they are extremely compute-intensive. This is exacerbated by the use of artificial neural networks (ANNs) as the classifier because ANNs themselves are very compute-intensive.

Compared to a traditional sequential implementation of GA training of ANNs, the use of graphical processing units (GPUs) as parallel processors provides significant performance improvements at a fraction of the price of alternatives such as cluster computing. GPUs, however, use a specialized programming paradigm which must be taken into account to leverage their full processing power. In this paper, we show an implementation of GA training of ANNs which achieves an order-of-magnitude speedup over a sequential algorithm.

2 Background

2.1 Artificial Neural Networks

ANNs are composed of nodes connected by weighted directed edges. In a feed-forward network architecture, nodes are organized into an input layer, one or more hidden layers, and an output layer. Each layer is fully connected to the next. Each of the hidden nodes at hidden layer n perform a transcendental function with inputs equal to the outputs of layer $n-1$ each multiplied by the weight associated with the edge through which they are fed from layer $n-1$ to layer n. In a radial basis function (RBF) network, the nodes in each hidden layer calculate the distance between their inputs and a centre vector, and pass a weighted distance to a RBF. Nodes in a sigmoid function network perform a similar operation with a sigmoid function in place of the RBF. The output node is typically linear, computing a weighted sum of its inputs. The output of the ANN is a single real value which is interpreted as a classification confidence. ANN computation is dominated by the number of multiplications at the directed edges, proportional to the number of features in the data set and the number of hidden nodes, and by the transcendental functions at the hidden nodes, which are typically exponentiations. ANN training has been implemented previously on GPU hardware, however these efforts have focused on training through backpropagation [2] as opposed to genetic algorithms.

2.2 Genetic Algorithms as ANN Training Agents

Genetic algorithms (GAs) use populations of candidate solutions to simultaneously explore the search space of ANN parameters [1]. The candidate ANNs are randomly mutated, mated and selected during each of several generations. Mutation and mating is done using a problem-specific representation of candidate solutions and "genetic operators." Selection requires calculating the fitness of each candidate ANN then preferentially selecting candidate ANNs based on fitness. The fitness of a classifier is a classifier accuracy metric such as sensitivity. The competitive bias imposed by selection attempts to mimic the "survival of the fittest" principle often seen in nature. Because GAs perform an unbiased search of a solution space and ANNs can be of arbitrary complexity, this technique is suitable for data mining applications using large or complex data sets.

The computation required for GA training of ANNs is proportional to the number of generations (10^1-10^4), to the size of each generation (10^1-10^2 candidates) and to the size of the data set (10^2-10^6 instances).

2.3 The CUDA Platform

NVidia's Compute Unified Device Architecture (CUDA) is a programming platform for massively parallel GPUs found in off-the-shelf graphics cards. GPUs consist of several dozen independent floating-point units, providing significant speedup to data-parallel compute-intensive applications [3]. Each unit is connected to on-board memory using a very wide bus, enabling high memory bandwidth provided certain memory access rules are respected. These features make CUDA an ideal platform for GA-ANN training.

2.4 Sample Application

Our demonstration classifier is designed for the 2009 UC San Diego Data Mining Contest "E-commerce Transaction Anomaly Classification" [4] and with training data thereof. The training data consist of 94682 instances with 19 features of mixed types and a 1:50 binary class imbalance. The evaluation metric is "lift at 20%", which can be understood as the ratio of the true positive rate in the top 20% ranked instances to the overall positive rate of the data set. Lift at 20% is commonly used in marketing research in order to select the subset of a population which is most likely to respond to solicitation.

3 Implementation of GA Training of ANNs

Our GA begins by generating an initial population of random candidate ANNs, each of which is a set of parameters describing one ANN. Feature values for all training instances are loaded into GPU global memory in preparation for ANN computations. Then, the following steps are executed in sequence for each generation of the GA:

1. Compute the ANN output corresponding to each training instance, as calculated by each candidate ANN in the population.
2. Find the threshold that defines the highest 20% outputs for each candidate's set of output values.
3. Compute the number of top 20% instances which are truly positive, again for each candidate ANN. This allows us to calculate lift, which is used as the fitness value for each candidate ANN.
4. Apply genetic operators: mating, mutation and selection. The selected candidate ANNs become the new population, to be used in the next generation.

Items 1, 2 and 3 were all implemented on the CUDA platform and are described in detail below. Performance results are given as the combined time for one generation of these three items but exclude one-time initialization. Item 4 is not very computationally intensive and was not parallelized.

3.1 Computation of ANN Output Values

ANN output values are computed by reading the training instances as real-valued feature vectors and applying the appropriate operations represented by the network topology and parameters. Feature data are organized to efficiently use memory bandwidth. ANN output calculation is particularly well-suited to the CUDA architecture because it has a very high ratio of mathematical operations to memory accesses (160 multiplications, 4 exponentiations and 80 bytes of memory accesses for our ANN topology). In addition, each output value depends only on the current ANN parameters and one training instance's feature values. Therefore, parallelization of this step scales particularly well with hardware capabilities.

3.2 Calculation of Top 20% Threshold

Next, we must calculate the k^{th} largest value in each candidate's list of output values, where $k = 0.20 * n$ and n is the number of output values (number of training instances). We can accomplish this by iterating through all n unsorted output values and inserting each into a binary minheap of fixed size k. The heap size is kept fixed by removing the root (smallest node) after each insertion. At the end of this process, the root of the heap is the k^{th} largest output value. The insertion and removal can be combined into a single operation which costs $O(\log k)$ time. The heap occupies $4 * k$ bytes of device shared memory.

Because fast on-chip memory (16 KB) is not large enough for our value of k ($4 * 18936 =$ 74 KB), we instead use p passes of $\frac{k}{p}$-sized heaps. Each pass examines only values which are less than the threshold calculated by the preceeding pass. For example, the third pass examines only values below the 10% threshold and finds the top 5% threshold of this subset of values. This threshold therefore is the 15% threshold of the full list. The threshold calculated by the last pass is the desired 20% threshold.

The overall complexity of our algorithm is $O(np \log \frac{k}{p})$. Each top 20% calculation is performed independently for each candidate ANN. Therefore, the parallel performance of this step scales with hardware capabilities when there are enough candidates to occupy the entire device.

3.3 Computation of Number of Truly Positive Top 20% Instances

To compute the number of truly positive instances in the top 20% of scores, if training instances are unsorted, we must read from memory each instance's true class and corresponding ANN output value and compare the output value to the candidate's threshold. For n training instances, this requires $2 * n$ memory accesses. However, by keeping the n_p positive instances first in memory, we need only read the first n_p ANN outputs, knowing that they correspond to the positive instances. This reduces the number of memory accesses to n_p, which is significant particularly for training data with low positive rate (in our case, $2*n = 189364$ while $n_p = 2080$). Sorting the training instances by class is a cheap operation which is performed only once during initialization and can be done by the host CPU.

Counting of positive instances above the candidate's threshold is performed independently, in parallel, for each candidate. However, because there may not be enough candidates to fully occupy the GPU, we also split the counting process for each candidate into 256 independent sub-counts: the first sub-counter processes training instances 0, 256, 512, ..., the second sub-counter instances 1, 257, 513, ..., such that the sub-counts can be executed in parallel. The striped assignment of training instance indices to sub-counters is necessary to take advantage of the full memory bandwidth of the GPU. The host CPU then aggregates the sub-counts by summing, which is a very lightweight operation.

Finally, each count is divided by the number of positive instances to give a positive rate in the top 20%. Then, the ratio of this positive rate to the overall positive rate is the lift of each candidate. These operations are also comparatively lightweight and executed on the host CPU.

4 Results

4.1 Experimental Setup

To evaluate the performance of our system, we compared execution time of equivalent algorithms running on commodity desktop CPUs and GPUs. Each series represents one type of experimental hardware (Table 1). Each data point is an average over 10 runs of the algorithm with the same parameters.

Legend Label	Hardware	Clock	Free RAM	OS
Core2	Intel Core 2 Q9450 CPU	2.66 GHz	7.5 GB	Linux 2.6 (64-bit)
i7	Intel i7 920 CPU	2.66 GHz	2.4 GB	Windows 7 (64-bit)
GTX260	NVidia GTX260 GPU	1.24 GHz	N/A	Windows 7 (64-bit)
GTX275	NVidia GTX275 GPU	1.4 GHz	N/A	Linux 2.6 (64-bit)

Table 1: Experimental hardware. Free RAM is OS-reported available memory before running each experiment.

4.2 Performance Results

In most cases, the GPU implementation resulted in approximately an order of magnitude speedup over a CPU implementation (Figure 1). For population sizes below 10, observed speedup decreases and no speedup is observed in the single-candidate case. This is due to GPU initialization and data copying dominating execution time. However, in these cases, both implementations are still quite fast (20 ms per generation).

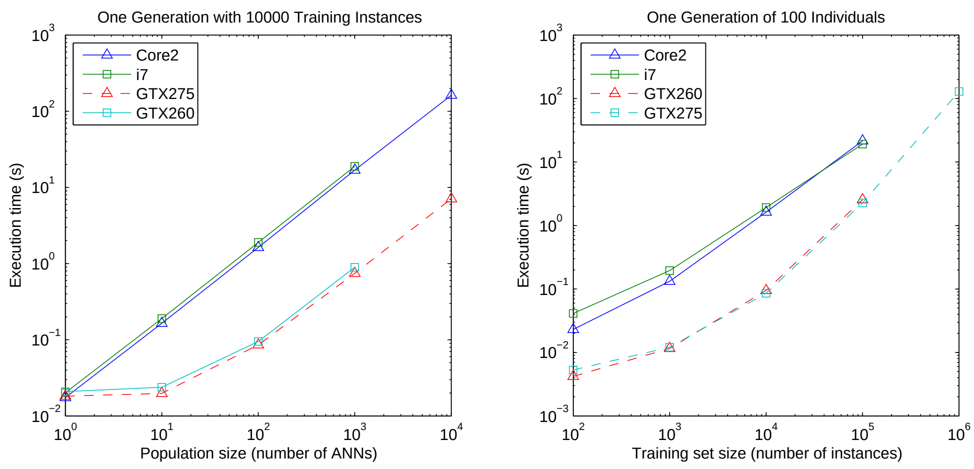

Figure 1: Training performance as a function of population size (left) and training set size (right). Note logarithmic axes.

5 Conclusion

We have presented an implementation of GA training of feedforward ANN classifiers for the CUDA platform for GPU programming. By carefully designing memory organization, algorithm computational load and memory access patterns, we have obtained a 10-fold speedup compared to a conventional sequential CPU implementation. A multipass approach was required for part of our system to compensate for the limited amount of fast on-chip GPU memory. Our method scales across population and training set sizes and is expected to be useful in other data intensive machine learning and data mining tasks.

References

[1] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the eleventh international joint conference on artificial Intelligence*, volume 123, pages 762–767. Citeseer, 1989.

[2] Chris Oei, Gerald Friedland, and Adam Janin. Parallel Training of a Multi-Layer Perceptron on a GPU. 2009.

[3] T. Halfhill. Parallel processing with cuda: Nvidia's high performance computing platform uses massive multithreading. *Microprocessor Report*, 2008.

[4] UC San Diego. 2009 UC San Diego Data Mining Contest.