Accelerating Genetic ANN Training using the CUDA Platform

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 18, 2010

Abstract

We present an implementation of genetic algorithm (GA) training of 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 population and training set sizes. This performance boost enables classifier design to span a wider range of population sizes and number of generations affording better classifiers and improved understanding of long-term trends in genetic algorithm convergence. Finally, we demonstrate this method in the context of the 2009 UC San Diego Data Mining Contest.

1 Introduction

Genetic algorithms (GAs) are a stochastic, evolutionary approach to machine learning classifier training. While greedy methods can get stuck at local extrema, GAs are theoretically capable of asymptotically reaching the global optimum. However, because they require the testing of a population of classifiers at each iteration, and the algorithm is repeated for many generations, 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 a network of nodes connected by weighted directed edges. In a radial basis function (RBF) network architecture, nodes are organized in an input layer, a hidden layers, and an output layer. Each layer is fully connected to the next. The nodes in the

hidden layer calculate the distance between their inputs and a centre parameter, and pass a weighted distance to an RBF. The output node is a summer weighted by the incoming edges. The output is a single real value which can be interpreted as a classification confidence. ANN computation is dominated by the multiplications at the edges, proportional in number to the number of features and the number of hidden nodes, and transcendental functions (exponentiation for Gaussian RBF) at the hidden nodes.

2.2 Genetic Algorithms

Genetic algorithms (GAs) use multiple candidate solutions to simultaneously explore the search space of ANN parameters. The "individuals" 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 is performed by applying the classifier to the training set and calculating a fitness value (classifier performance). The competitive bias imposed by selection attempts to mimic the "survival of the fittest" principle so often seen in Nature.

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 individuals) and to the training set size (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 many independent floating-point units, providing significant speedup to data-parallel compute-intensive applications [1]. Each unit is connected to on-board memory using a very wide bus, enabling high memory bandwidth provided certain particular memory access rules are respected. The combination of these features makes CUDA an ideal platform for GA-ANN training.

2.4 Data Mining Contest

Our demonstration classifier is designed for the 2009 UC San Diego Data Mining Contest "E-commerce Transaction Anomaly Classification" and with training data thereof. The training data consist of 94682 instances of 19 mixed features with 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.

3 Methods

3.1 Classifier Implementation

Each GA iteration consists of the following steps:

1. Computation of ANN outputs
2. Calculation of top 20% threshold value (each individual)
3. Counting of the positive instances with outputs above this threshold (each individual)
4. Genetic algorithm processing – mating, mutation, selection

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

3.2 Computation of ANN Output Values

ANN output values are computed by reading feature values and applying the appropriate operations represented by the network topology and parameters. Feature data are organized to enable memory read coalescing. This part of the process 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).

3.3 Calculation of Top 20% Threshold

To efficiently calculate the top 20% threshold in a list of n ANN output values, we can insert all values into a k-sized minheap where $k = 0.20 * n$, and take the root of the heap as the threshold. Each minheap insertion costs $O(\log k)$ time. Because device shared memory (16 KB) is not large enough for our value of k, we instead use p passes of $\frac{k}{p}$-sized heaps with which we find the top 5% threshold, then the top 10%, etc. The overall complexity of our algorithm is $O(np \log \frac{k}{p})$. Parallelization is across individuals.

3.4 Counting of Positives Above Threshold

By keeping positive instances first in our training set, we avoid the need to consult the class of each instance (saving a costly global memory read). We simply iterate over the first $numPositives$ ANN outputs and count the outputs over the individual's 20% threshold. Parallelization is across both individuals and groups of outputs. For each individual, we keep m counters which accumulate counts for indexes i where $i \equiv 0 \bmod m$, $i \equiv 1 \bmod m$, ..., $i \equiv m - 1 \bmod m$. This enables parallelization even when there are not enough individuals in the population to fully occupy the device. Each batch of m kernels accesses m sequential locations in memory, which results in coalesced reads.

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**
Core2	Intel Core 2 Q9450 CPU
i7	Intel i7 920 CPU
GTX260	NVidia GTX260 GPU
GTX275	NVidia GTX275 GPU

Table 1: Experimental hardware

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 low population sizes, GPU initialization and data copying dominates.

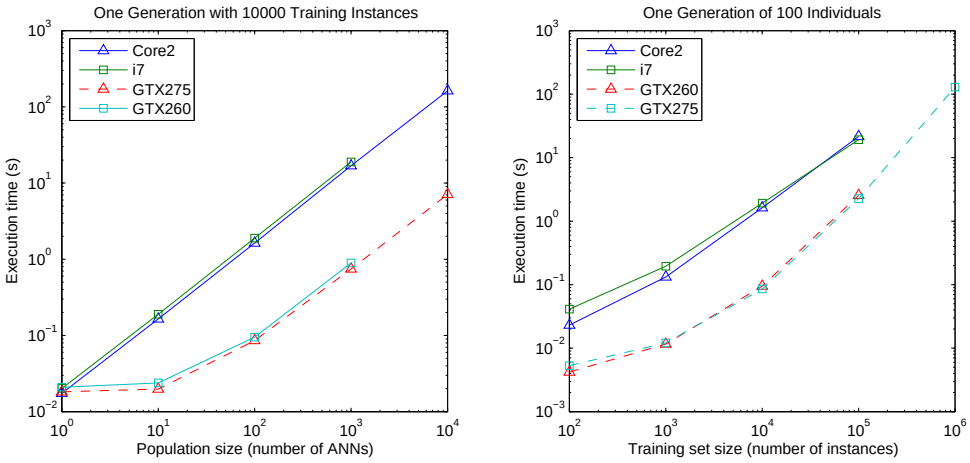

Figure 1: Training performance as a function of population size (left) and training set size (right). Note logarithmic axes.

5 Conclusion

We have presented an implementation of GA training of 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. Additional workaround were necessary for the low amount of fast on-chip GPU memory. Our method scales across population and training set sizes.

References

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