Evolving Neural Networks on GPUs

Johannes Hofmann
Friedrich-Alexander-University Erlangen-Nuremberg
Email: johannes.b.hofmann@stud.informatik.uni-erlangen.de

I. Introduction

Financial Time Series prediction attempts to model the behavior of financial markets using, among other things, tools like technical, inter-market, and fundamental indicators. Accurate prediction, however, is difficult for a number of reasons: financial markets are influenced, often in a non-linear, sometimes time-lagged fashion, by factors including interest and exchange rates, the rate of economic growth, and a number of industrial commodities.

Neural Networks (NN) are a well-established method to attempt to conquer these difficulties [1]. Using unsupervised learning like backpropagation or the Levenberg-Marquardt algorithm NNs can be trained to model a market using historic market data; complex models, however, require the use of large NNs, the training of which requires large amounts of historic data — leading to long training periods.

As the size of the network needs to be maintainable there can only be a limited number of inputs, leaving the network designer with the question what inputs to select from a large amount of technical indicators. The network topology also has a great impact on network's modeling ability. There exist no exact methods for finding the "right" inputs or the "right" network topology: practitioners have to use heuristics until they arrive at a combination of inputs and topology that satisfies their requirements.

Evolutionary Computing can provide a solution to this dilemma by maintaining a population of NNs with different inputs and topologies. However, as we've already mentioned, the time required to train a single network can be substantial, so the notion of training a whole population over a number of generations can render the algorithm infeasible for ordinary processors in terms of execution time.

To overcome this problem we present an implementation which uses the GPU to speed up the search. The algorithm implements a new, NN-oriented, evolutionary search which is based on ideas borrowed from Grammatical Evolution (GE) [2], as well as other DNA-inspired concepts.

II. Algorithm

A number of partial[1] phenotypes are maintained in a population of individuals using variable-length binary genomes. These phenotypes are created in two stages: *transcription* maps successive pairs of eight bits onto integer values which are used in a context-sensitive fashion during *translation* to chose appropriate actions for constructing a NN topology.

While transcription is the same as in GE we decided to customize the translation operator: because NNs are not *per se* sentences of a language, which can be described by a grammar, we use stop codons[2] during the mapping process to decide whether to add inputs to a neuron, create a new neuron, or create a new network layer.

[1]In our algorithm the genotype determines only NN inputs and NN topology, network weights are not encoded in it.

[2]This design is based on the concept of stop codons used in the translation phase during gene-expression in living cells.

Once the skeleton phenotype (NN input and structure) has been created the missing weights to complement the NN are generated during evaluation with the help of the backpropagation algorithm.

Backpropagation initializes the network's weights with random values, then performs gradient descent in batch to minimize the network error, which is calculated using a validation data set to prevent overfitting. Gradient descent can only find local minima, so in an attempt to increase the odds of identifying the global minimum backpropagation is performed R times. The best value of all iterations is used as the neural network's fitness, and the corresponding weights are saved with the individual to enable later use of the network for prediction on out-of-sample data.

The evolutionary search is performed using stock evolutionary operators: tournament selection, ripple crossover and bit mutation for variation, and a generational scheme featuring k-elitism as replacement operator.

During evaluation we used the algorithm to evolve a network which attempts to predict the next day's closing value of the German stock index (DAX). A number of technical indicators including stock indexes like the Dow Jones Industrial Average, industrial commodities such as oil and aluminum, and a number of moving averages were used to train the network.

III. Implementation

A. CPU

Additional to the sequential algorithm we implemented a parallel version for the CPU to allow for a fair comparison of both architectures. This was done using OpenMP [3] to parallelize most of the evolutionary operators (seeding, parent selection, crossover, evaluation); the only components requiring manual tuning were random number generation and the replacement operator using k-elitism: because parallel performance of glibc's `drand48_r` was more than poor we decided to implement a parallel Mersenne twister for random number generation; location of the k fittest individuals was performed using `_gnu_parallel::min_element` from the Standard Template Library's `parallel/algorithm` suite.

Speedups for the parallel version are included in Table I.

B. GPU

The whole algorithm is run on the GPU, every generation the elite is transfered to the CPU and stored, allowing the search to be stopped at any time.

Random number generation is done with the CURAND library and parallelization of all evolutionary operators is straightforward: seeding, parent selection, crossover, and mutation can be considered embarrassingly parallel tasks, which is why in these operators each individual can be mapped onto a thread (two individuals per thread in the case of crossover) and handled independently. Because these operators' design is simple there is low divergence, which is why their parallelization scales very well. Elite location in the replacement operator is implemented with the help of the Thrust library [4].

Parallelization of the fitness function should not be based on the already mentioned "one thread per individual" scheme because the number of iterations of the gradient descent algorithm varies greatly with NN structure and seeding of weights. This leaves threads with a low number of iterations stuck until the thread with the most iterations has finished, thus causing divergent warps and thereby essentially degrading parallelism to the multiprocessor level.

In batch learning, instead of applying the calculated weight adjustments[3] directly after a training set has been presented to the network, averaged weight deltas over all training sets are used to update the neural network weights.

We can use this property to implement the parallelization of the evaluation function on an individual-per-block level: each thread in a block processes some of an individual's training data to produce accumulated weight deltas, which are then shared and summed up (using best reduction practices [5]), and finally averaged to update the weights of the network.

Calculation of the network error on the validation dataset is done they same way: each thread handles some of the validation data and sums up the individual errors in a local variable, then the local errors are shared between threads and summed up to calculated the total network error.

We use the constant and L1 cache, if available, to store training and validation data, but we were able to achieve a substantial speedup on GPUs with no caches as well, despite low occupancy caused by high kernel register usage: memory latency can easily be hidden because operations associated with neural networks are very expensive in terms of floating-point computation, i.e. they take longer to calculate than it takes the next input to arrive.

IV. Results

We evaluated our implementation using an AMD Opteron 2435 hexacore and an Intel Xeon X5650 hexacore on the CPU side, and an NVIDIA GeForce 480 GTX (Fermi; caches) as well as an NVIDIA Tesla C1060 (pre-Fermi; no caches) on the GPU side. Speedup results are shown in Table I and were calculated using the average time of ten runs.

	AMD (1)	Intel (1)	AMD (6)	Intel (6)
Intel (1)	1.1			
AMD (6)	5.1	4.7		
Intel (6)	5.6	5.2	1.1	
Tesla C1060	112.1	91.3	19.6	17.6
GTX 480	309.3	285.3	61.3	55.0

TABLE I
SPEEDUP OF THE HARDWARE IN THE LEFT COLUMN IN RELATION TO THE HARDWARE IN THE TOP ROW (NUMBERS IN PARENTHESIS INDICATE NO. OF CORES USED) USING $\mu = 100$, $R = 4$, AND 10 GENERATIONS.

For other algorithms people have suggested the use of `-use_fast_math` to speed up calculation on the GPU. While a better speedup could be achieved (62 instead 55 when comparing the GeForce GTX 480 to the Xeon CPU) we found that solution quality degraded too much (see Fig. 1) to include the results in a competitive comparison.

The phenomenon of solution quality degradation is easily explained by high-precision floating-point calculations being required when propagating δ values backwards through the network during training: typically the value decreases by an order of magnitude for each hidden layer it passes; should the value become small enough to be influenced by imprecise calculation the training of the network is corrupted.

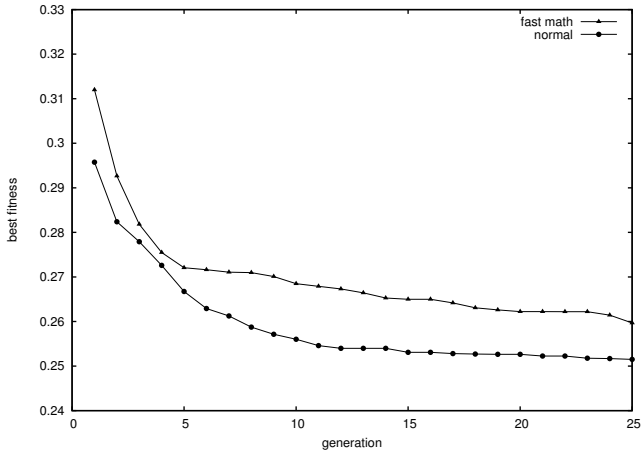

Fig. 1. Fitness development over 25 generations ($\mu = 1000, R = 8$).

V. Conclusion

Although we deliberately chose the CPUs from today's high-end range, they are clearly outperformed by both GPUs, despite their higher price tag — $688 for the Opteron 2434 and $875 for the Xeon X5650 compared to the fast GeForce GTX 480 for $300.

However, when it comes to lengthy calculations initial acquisition cost of hardware is often negligible when put in relation to cost caused by hardware operation, i.e. power usage.

To come up with a good NN for prediction, we ran the algorithm with a population of 5000 individuals and 32 backpropagation instances over 20 generations. On the GeForce GTX 480 the configuration executed for about 17 hours. This setup would have required an estimated 38.5 days on the fast Xeon CPU amounting to a total of 87.9 kWh used power, compared to 4.4 kWh consumed by the GTX 480.[4]

In conclusion, all obtained results back up our hypothesis of the algorithm being infeasible to run on the CPU, thus justifying the time invested in parallelizing it for the GPU.

References

[1] B. K. Wong, V. S. Lai, and J. Lam, "A bibliography of neural network business applications research: 1994-1998," *Computers & OR*, vol. 27, no. 11-12, pp. 1045–1076, 2000. [Online]. Available: http://dx.doi.org/10.1016/S0305-0548(99)00142-2

[2] M. O'Neill, "Automatic programming in an arbitrary language: Evolving programs with grammatical evolution," Ph.D. dissertation, University Of Limerick, Ireland, Aug. 2001.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[4] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.3.0. [Online]. Available: http://www.meganewtons.com/

[5] M. Harris, "Optimizing parallel reduction in cuda," *NVIDIA Developer Technology*, 2008. [Online]. Available: http://www.mendeley.com/research/optimizing-parallel-reduction-cuda/

[3] A weight adjustment, or weight delta, Δw_{ij} between node i and j is calculated as $\Delta w_{ij} = -\eta x_i \delta_j$ with η a manually set learning rate, x_i the output of node i, and δ_j the contributing error of node j.

[4] Power usage was calculated using only TDP values of the CPU respectively the GPU, neglecting the power required to operate the hardware, to allow a direct comparison of both architectures.