Parallel Implementation of Moving Averages and Stock Market Prediction

John Jenq
Computer Science Department, Montclair State University, Montclair, New Jersey, USA

Abstract - *In recent years, graphics processing units have made parallel processing affordable with the price of personal desktop computers. This report investigates the computational aspects of calculating simple moving average and exponential moving average operations, two of the most popular financial indicators. In this report, we also investigate the usage of GPU to run artificial neural network as a mean of predicting stock market pricing. Feedforward and Backpropagation artificial neural network was used for this study. Financial data including major stock indices, volumes, pricing, and moving average of stocks were used as input. The future stock prices can be predicted as the output. The speedup factor by adopting GPU and CPU together over traditional CPU alone implementation was not significant. The computation of compute moving averages on GPU was also discussed.*

Keywords: artificial neural network, stock prediction, GPU computing, parallel processing, high performance computing.

1 Introduction

Graphic processing unit (GPU) has transformed a regular PC into a personal supercomputer. For example, in [5], the GeForce GTX 580 can perform single precision operation that reaches more than 1500 GFlops. These computing powers significantly speed up the computational intensive applications with a price of a PC. Many tools have been developed to make the GPU computing much easier than ever before. Personal supercomputing is now a reality to us.

Artificial neural networks are used for pattern recognition, clustering, and optimization. Neural networks can also be used to solve problems which are not easily solved by traditional calculation methods, particularly if there is no strong underlying theory to explain the data. Neural networks have been developed as generalizations of mathematical methods of neural biology, based on the assumption that the information processing occurs at many simple elements called neurons. Signals are passed between neurons over connection links. Each connection link has an associated weight which multiplies the signal transmitted. Each neuron applies an activation function to its net input to determine its output signal.

There are researchers using neural network in financial and economic computations. For example, in [3], Li and Liu used LM BP algorithm to predict the Shanghai stock market. In [9], Wang developed a HLP method that gets stock high low points with different frequencies and amplitudes. The extracted data is then fed into a neural network to forecast the stock direction and price. In [8] Tirados and Jenq used neural networks to predict GDP with ten leading economic indicators as the input. In [4] Lin and Feng combined neural network and pattern matching techniques to analyze and forecast oil stock prices. In [10], Zhou and Zhang used financial indicators such as moving averages, volumes, relative strength index, etc. on neural network to predict future stock prices.

In the past, CPU clusters have been used to achieve high performance computation. GPU computing uses GPU as a co-processor to accelerate CPUs for general purpose scientific and engineering computing. It shifts computation intensive program segments into GPU while keeping the rest of the program segments, which are serial in nature, on the CPU. This kind of hybrid computing improves the performance of many computer applications.

The GPU computation can be used on financial computations as well. Researchers in the financial world find the benefits of using GPU in financial computation. In [6], Peng, et. al., compute option pricing on GPU with backward stochastic differential equation. in [1], Lee, et. al., did financial derivative modeling using GPUs. In [7] Solomon et. al., used trinomial lattice strategy to implement the pricing of European option and American lookback option pricing using GPU. In [2], Lee, et. al., investigated random number generation and the Monte-Carlo simulation to predict future stock prices. They also discussed the out of core case when graphics DRAM is not big enough to hold all the application data.

The rest of the paper is organized as the following. Section 2 discusses methodologies and implementation of an artificial neural network. Parallel implementation of simple moving average and exponential moving average will be discussed as well. Section 3 discusses and analyzes the experimental results. Section 4 gives conclusion remarks.

2 Methodologies and implementations

Three-layer neural network was chosen to implement our prediction system. The inputs are major industrial stock indices and stock indicators. The goal is to forecast future stock prices. Feedforward and backpropagation neural network was used. Backpropagation is a gradient descent neural network method used to minimize the total squared error of the output calculated by the network. The network is developed to achieve a balance between the ability to respond to the input patterns that are used for training and the ability to give good responses to input that is similar, but not identical, to that used in training. Like with multiple regression, backpropagation was used to develop a correlation between the input of stock market data in order to determine the future stock price.

The training of a network by backpropagation involves three stages: the feedforward of the input training pattern, the calculation and backpropagation of the associated error, and the adjustment of the weights. After training, application of the network involves the computations of the feedforward phase only.

To prevent the larger data from dominating the outcome, the raw data will be processed before being fed into the neural network. The raw data has been modified. In the following discussion, *B* represents the raw form of the original data, and C is the normalized version of *B*.

$$C_t = 2\left(\frac{B_t - B_{\min}}{B_{\max} - B_{\min}}\right) - 1 \qquad (1)$$

This transformation map our data to between -1 and 1. The activation function selected is the bipolar sigmoid function, which has a range of (-1, 1), and is defined as

$$f(a) = \frac{2}{1 + exp(-a)} - 1 \qquad (2)$$

$$\frac{df(a)}{da} = \frac{1}{2}[1 + f(a)][1 - f(a)] \qquad (3)$$

2.1 Feedforward Backpropagation Neural Network

The traditional Neural Network algorithm can be simplified as below

```
while (cycle < maxCycle  && averageError > toleranceEerr)
{
      for (int i= 0; i < totalRecords;  i++)
      {
            forwardPropagation();
            backwardPropagation();
            accumulateError();
            updateWeights(learningRate);
      }
      compute averageError;
}
```

To simplify the discussion, let's denote the connection weights between the input layer and hidden layer to be *w*. Also let's denote the connection weights between the hidden layer and output layer as *v*. Note that both *w* and *v* are vectors. To parallelize the code in order to fit into GPU computing, some modifications will be made. Instead of performing backpropagate operation for each pattern to update weights, we will compute the *dw* (the update of weight *w*), *dv*(update of weights *v*) and then do the update of the weights at the end of each cycle. It means that we move the updateWeights(learningRate) function out of the traditional algorithm above. Instead of performing these operations for each record for each cycle, we reduce them to once per cycle. Therefore the total weights to be updated would be the summation of *dw* and *dv*, which are calculated by an individual pattern during the process. Note that the *dw*s are the weights to be added to the *w*, the weights between input layer and hidden layer. And *dv*s are the weights to be added to *v*, the weights between hidden layer and output layer. Here we assume there is one output neuron although more output neurons are possible. The updateWeights function can be done by binary reduction which can be performed in *logN* steps using *N* threads. Although the backwardPropagation and updateWeights functions may be parallelized, the gain of speedup may be limited. This setup allows us to assign one pattern to each thread in order to speed up the process.

Because the binary reduction operation may slow down the whole system performance, it is interesting to find out if assigning more than one pattern to a thread for processing will improve the performance. Once again, it depends on how many clock cycles will be used to synchronize the threads. If more threads are to be synchronized, then we would expect more time for the reduction operation. An interesting aspect is to find out how to organize operations so that we can get the best possible performance.

2.2 Moving Average

Moving averages can be used as financial indicators. There are various types of moving averages, of which two of the most popular are simple moving average and exponential moving average.

2.2.1 Simple Moving Average

Assume the daily closing price for day *t* is C_t. The *n*-day simple moving average can be defined as $SMA(t) = \frac{\sum_{i=t-1}^{i=t-n} C_i}{n}$, where C_i is the closing price at day *i*. So simple moving average can be computed by taking the average closing price of a stock, over the last *N* periods. Popular simple moving

averages are 5, 10, 20, 40, and 200. Let's assume that the last five periods for a stock are 1,3,5,7, and 9. Then, the 5 day simple moving average can be computed as (1+3+5+7+9)/5 = 5. While simple moving average giving all past *n*-day closing prices are weighted equally, the *n*-day exponential moving average assigns more weight to the most recent price, which will be discussed in the next subsection.

Simple moving average on a parallel computer can be done by using Prefix Sum operation. It also known as Scan operation. A Prefix Sum can be defined as the following. Given a set of *N* values $a_1, a_2, a_3, \ldots, a_n$, and an associative operation @, the Prefix Sums operation will compute the *N* quantities (a_1, $a_1@a_2$, $a_1@a_2@a_3$,..., $a_1@a_2@a_3@\ldots@a_n$). By using the Prefix Sum operation, the *N*-day Simple moving average of day *"i"* can be calculate as

$$SMA[i] = (prefixSum[i] - prefixSum[i-N])/N \quad (4)$$

where *SMA[i]* is the *N* day moving average at day *i*. Table 1 gives an example that uses Prefix Sum to compute 3-day moving averages. Assume the missing period (period -1, and -2 in our example) values are 0.

Period	1	2	3	4	5	6	7
Value	1	3	5	7	9	10	12
prefixSum	1	4	9	16	25	35	47
Total of subsequence	1	4	9	15(= 16-1)	21(= 25-4)	26(= 35-9)	31(= 47-16)
Simple Moving Average	1	2.5	3	5	7	8.67	10.33

Table 1 Example of calculation of *SMA* using prefixsum

2.2.2 Exponential Moving Average

Exponential moving average can be defined as $EMA(t) = (P_t - EMA(t-1)) * \alpha + EMA(t-1)$, where P_t is the closing price at day *t*. The α is weighting factor and can be defined as $\alpha = 2/(n+1)$, where *n* is the number of time periods involved in the computation. For example, for 10-day period EMA, $\alpha = 2/(10+1) = 0.181818$, while 20-day EMA, $\alpha = 2/(20+1) = 0.095238$. One can rewrite the above EMA definition using past *n*-day closing prices and α as follows. Note that weights are decayed exponentially and the most recent prices carry more weights in the computation.

$$EMA(t) = \alpha P_t + \alpha(1-\alpha)P_{t-1} + \alpha(1-\alpha)^2 P_{t-2} \ldots + \alpha(1-\alpha)^{n-1} P_{t-n+1}$$

Even though *EMA* can be easily computed by using the method as mentioned in the previous paragraph, however it has serial nature in the computation. In order to deliver a parallel algorithm, let's define $\beta = (1-\alpha)$ where α is the multiplier that we define from the previous discussion. The formula to compute exponential moving average then becomes the following

$$EMA(t) = \alpha P_t + \alpha\beta P_{t-1} + \alpha\beta^2 P_{t-2} \ldots + \alpha\beta^{n-1} P_{t-n+1}$$

We will partition the whole data array into segments with lengths of powers of 2. Assume the leftmost data is the most recent data(the lowest index) and rightmost one (the highest index) is the oldest timing data. By using $O(N)$ memory to store the comupted information, we can therefore compute exponential moving average of P period in $O(log_2 P)$ time. Here, N is the number of records. Actually one can reduce the total memory to $2N$ as we will discuss shortly.

The computation of *EMA* involves two phases. In the first phase, we will generate the required data through iterations. In the first iteration, two pieces of data that are adjacent to each other will be processed to create a combined length-2 information. In the second iteration, combined length-4 segment information can be generated from the combined length-2 information. To simplify, let's assume p is power of two. In $log_2 P$ iterations, one can create $\frac{P}{2} + \frac{P}{4} + .. + 1$ which is a total of $P - 1$ pieces of data information for each segment of length *P*. This information will be used in the second phase of our *EMA* computation. Table 2 gives an illustration of the first phase *EMA* computation. Note that $\beta = (1-\alpha)$.

0	1	2	3	4	5	6	7
	(01)		(23)		(45)		(67)
		(0123)				(4567)	
				(1->7)			
	$c_0 + \beta c_1$	$c_0 + \beta c_1 + \beta^2(c_2 + \beta c_3)$	$c_2 + \beta c_3$	$c_0 + \beta c_1 + \beta^2(c_2 + \beta c_3) + \beta^4(c_4 + \beta c_5 + \beta^2(c_6 + \beta c_7))$	$c_4 + \beta c_5$	$c_4 + \beta c_5 + \beta^2(c_6 + \beta c_7)$	$c_6 + \beta c_7$

Table 2 Phase one of EMA Computation: data store scheme

For a segment of *P* periods of data on a *N* data array, where *P* < *N*, our job is to find the *EMA* for all the data on the data array with *N* data. Our approach is to partition any segment from index *a* to index *b*, where (*b-a+1*) is the period *P*, into at most two segments. It is possible that there is only one segment if the segment starts with index of power of 2 and ends with a power of 2 minus 1. In that case, we can compute the *EMA* value. It is just the value in the array of index $\lceil (a+b)/2.0 \rceil$. For example, if *a* is 56, *b* is 63, and *P* is 8, then index 60 holds the *EMA* value. Refer to Table 2 above to see how the combined values are stored. Note that this is the

best case since there is only one segment. For other *P-1* cases, out of every segment of length *P*, there are two segments. Let's assume these two segments are P_1 and P_2. First of all, we have to find out the cutline index. The first segment P_1 starts with index *a* and ends with an index which is to the power of 2 minus 1. Let's call this index *cutindex*. It can be computed as $b/p \times p$. Consider a segment from index 53 to 60: the resulting cutline index is 56. Note $60/8 \times 8 = 56$. The second segment P_2 starts with an index 56, which is to the power of 2, and ends with *b*, which is 60 in this case. Note that segment P_1 and P_2 can be formed by elements with lengths of power of two. For example, if segment $P_1$is of length 11, then it can be formed as a sum of segments of lengths 1, 2, and 8. Note here that the lengths of the segments from left to right are in increasing order. P_2 can be processed similarly, however the size of the component segments are decreasing. Distinguishing the order of increasing and descreasing of these components is important. For example, if P_2 is a segment of length 14, then the component sub-segments will have lengths of 8, 4, and 2. This is the order to retrieve the information. Note also that these component segments never carry the same length as other component segments in its combination. We can use a loop to mark the partioned segments if they are not zero and then add them up to get the resultant *EMA*. Note that during the loop computation, different power of β need to be applied to the retrieved value so the power of β shall be correctly applied to faithfully reflect the weights assigned to these segments. For the above example, a segment from 53 to 60 is considered. After computing the value of P_1, the resultant computing value from P_2 needs to be multiplied by β^3 before we add the resultant value from the computation of P_2. It is cubed because the segment 53 to 55 has length of 3. The multiplication by the power of β applies to the process of computing the values of P_1 and P_2 with the same reason. Details are omitted.

3 Conclusions

The experiments were conducted on a Intel i7 with Nvidia GeForce 550M. The parallelized moving average and neural network version was constructed using CUDA C. Due to the small size of records and the nature of the neural network, the speedup wasn't observed. The main reason is due to the requirement of synchronization of these threads. It is apparent that the expensive cost of synchronization makes the CPU implement more appealing. We expect that when the number of neurons and number of data increases, we shall get better results. We trust Nvidia will come up with a better solution to deal with the synchronization of threads.

For moving average computation, GPU computing does not provide advantages over CPU computing when there is a small amount of data. If the amount of data by simulation is increased, some improvement can be achieved. We understand that different machines with different models of CPU and GPU can create different results.

4 Conclusions

The parallel versions of moving average computation used in the financial industry and back propagation neural network computation were developed to run on Nvidia GPU using CUDA C. The GPU version of moving average does not give significant speedup over traditional CPU version using prefix sum operation. For neural network training process, GPU computation does not provide significant performance over traditional CPU implementation due to the requirement of thread synchronization. Even if we tried to minimize the number of synchronization by using device kernel calls, then speed up over the traditional CPU approach wouldn't be significant. Actually, in some situations when the number of records are small, the CPU implementation is superior. The implementation of real time predicting system over a huge data set in the financial industry is an interesting and challenging problem for future investigation.

5 References

[1] Myungho Lee, Chin Hong Chun, and Sugwon Hong, " Financial Derivatives Modeling Using GPU's", International Conference on Scalable Computing and Communications; The Eighth International Conference on Embedded Computing, pp 440 - 445

[2] Myungho Lee, Jin-hong Jeon*, Joonsuk Kim, and Joonhyun Song, " Scalable and Parallel Implementation of a Financial Application on a GPU: with focus on out-of-core case", 2010 10th IEEE International Conference on Computer and Information Technology, pp 1323 - 1327

[3] Feng Li, and Cheng Liu, " Application Study of BP Neural Network on Stock Market Prediction", 2009 Ninth International Conference on Hybrid Intelligent Systems, pp 174 - 178

[4] QianYu Lin, and ShaoRong Feng, " Stock market forecasting research based on Neural Network and Pattern Matching", 2010 International Conference on E-Business and E-Government, pp 1940 - 1943

[5] Nvidia, "Nvidia Cuda C Programming Guide", version 4.0, May 6, 2011,

[6] Ying Peng, Bin Gong, Hui Liu, and Bin Dai, "Option Pricing on the GPU with Backward Stochastic Differential Equation", 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming, pp 19 - 23

[7] Steven Solomon, Ruppa K. Thulasiram and Parimala Thulasiraman, " Option Pricing on the GPU", 2010 12th IEEE International Conference on High Performance Computing and Communications, pp 289 - 296

[8] Edward Tirados and John Jenq, "Analysis of Leading Economic Indicator Data and Gross Domestic Product Data Using Neural Network Methods", Journal of Systemics, Cybernetics and Informatics, vol 7, no 4, 2009, pp 51-56

[9] Lei Wang, and Qiang Wang, " Stock market prediction using artificial neural networks based on HLP", 2011 Third International Conference on Intelligent Human-Machine Systems and Cybernetics, pp 116 - 119

[10] Yixin Zhou, and Jie Zhang, "Stock data analysis based on BP neural network", 2010 Second International Conference on Communication Software and Networks, pp 396 - 399