Parallel Computation of Binomial Security Pricing Model

CMU 15-418/ 618 Final Project

Authors and Contributors: Qifang "Charlie" Cai, Xiqiao Shan

Special Thanks to Prof. David Handron from CMU 21-370

Introduction

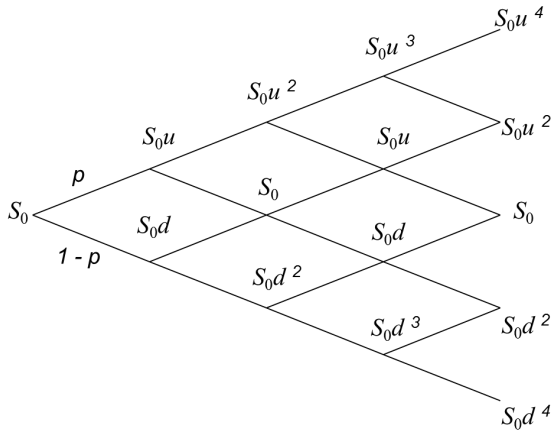
Although Fintech just becomes a hot topic in recent years, it does not mean people who work in a bank never really care about technology. In fact, the financial industry has a long history of implementing cutting edge technology to create competitive edges and win in the market. In the area of trading, there exist many firms that utilize the power of computers and trade in either high-frequency or quantitative manner. Pricing is essential in trading. With correct model and correct price, one can exploit the inefficiency in financial market and make a fortune by fixing the price and providing liquidity to the market. This mispricing arbitrage exists in the market and can be discovered by comparing the result of your own pricing model against the current market price. However, the opportunity vanishes rather quickly, often disappearing within a couple seconds. In order to capture the opportunity, one needs to process information fast with the help of a computer. It's challenging to implement an simple pricing engine with high performance and a good level of accuracy.

The problems that this project is trying to address are how to speed-up the calculation of a binomial pricing model and how to conduct small scale machine learning given a high performance pricing model. To create a high performance pricing model, our team converts the traditional sequential binomial model into simplified parallel-computing model using Nvidia GPU and CUDA API. Once the high-performance model is in place, apply the Stochastic Gradient Descent approach to conduct small scale machine learning on real market data. Others might have done

similar things before, but most of the information remains proprietary and is hard to search related information on the Internet. This project helps us better understand the concept of binomial security pricing model and its pros and cons. Moreover, this pricing engine can serve as a prototype core of a black-box trading system if people find the result to be useful.

Quick Financial Theory

In the setting of a binomial model, the evolution of stock price follows a coin toss process. Given a up factor (u) and a down factor (d), the result of a coin toss at each time will move the stock price to the next time period by a factor of either up (u) and down (d).



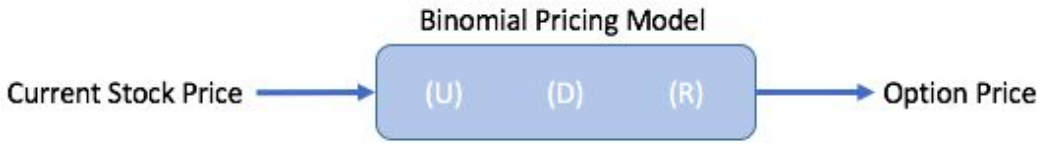
Once the stock price tree is in place, one can proceed and calculate the current theoretical value of the option. Calculation relies on the idea of backward induction - the current price of the security is the current risk neutral expected value of the security.

$$\tilde{p} = \frac{1 + r - d}{u - d}, \quad \tilde{q} = \frac{u - 1 - r}{u - d}.$$

$$S_n(\omega_1 \dots \omega_n) = \frac{1}{1+r}\left[\tilde{p}S_{n+1}(\omega_1 \dots \omega_n H) + \tilde{q}S_{n+1}(\omega_1 \dots \omega_n T)\right].$$

(risk neutral probability and backward induction formula)

It is convenient to view the pricing model as a system with three parameters (up factor, down factor, and interest rate), one input (current stock price), and one output (current theoretical price of the option).

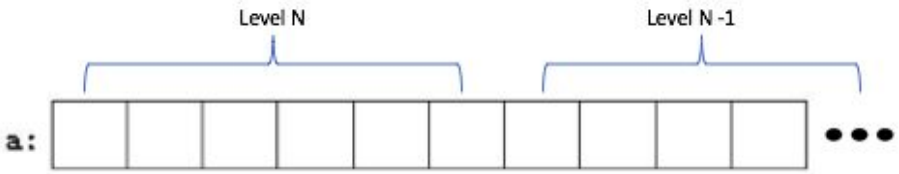


Design and Approach

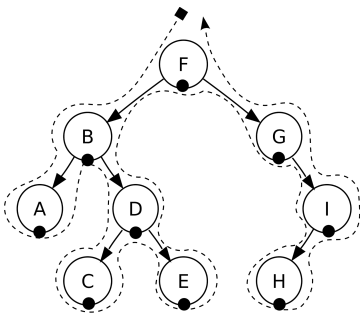
- Sequential Model

To make the binomial pricing model more accurate, the model grows to be **600 levels**. During the first iteration, the binomial tree is represented as a real tree data structure using linked list. However, this data structure turns out taking a lot of unnecessary memory space and has poor locality. Later the idea of linked list is replaced by a simple array. Still, it takes around 1.45 MB of allocated space for one of this array. The indexes in the array represent their positions in the binomial tree while the array stores actual values. Values in the same level from the tree are stored adjacent to each other. Here is a quick visualization.

The conversion between index and position in the tree is done by modulating the index number with each level's number of nodes, starting from level 600 all the way to level 1. This is a linear search method.



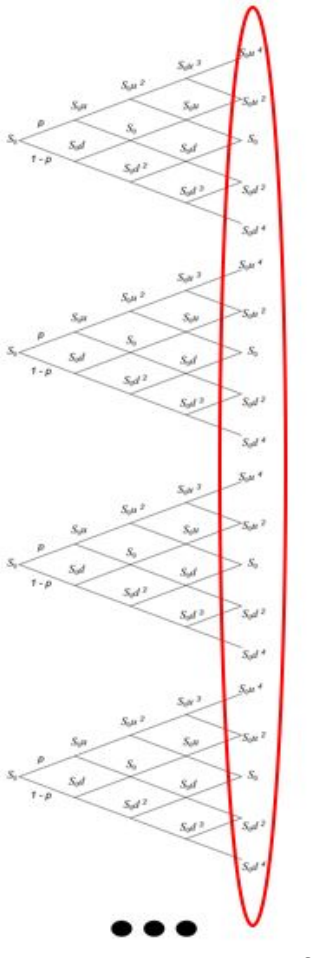
In order to conduct backward induction on this 600-level binomial tree, the sequential model adopts depth first search approach (DFS). The value of the derivative is first calculated at all the leaves (level-600) then their parent nodes. Once a node finishes calculating its own derivative



price, the price is written to the array at the respected index, effectively overwriting the array. After the entire backward induction process finishes, the current theoretical price of financial derivative is located at the end of the data array. Applying this sequential model, it takes around 250 millisecond to finish one complete model calculation.

- CUDA Parallel Model

The CUDA parallel implementation of the binomial model inherits some idea from the sequential implementation. It recycles the array data structure for simplicity and saving space. All array spaces are pre-allocated in the GPU memory. At the same time, parallel implementation uses a similar index calculation algorithm as its sequential brother, but given a parent's index, the algorithm only focus on getting the value and index of dependent children nodes. The Nvidia GTX 1080 GPU is quite spacious and has great performance. As a result, the parallel build of the model allocates place for up to 10 different model configuration. Literally, the parallel implementation can contain 10 different binomial pricing models at one time. During calculation, all models will calculate at the same pace.



Parallel computation is synchronized level by level. Figure on the right is a good illustration of the parallelism. At each level, all existing models in the GPU will run the same calculation algorithm, performing the backward induction formula for all the nodes at the same level. For a binomial model that has 600 level, the entire calculation can finish in 600 iteration. Notice that different iterations might not take the same amount of time to finish. This calculation approach is a significant improvement from the sequential DFS.

Stochastic Gradient Descent Machine Learning

We have tried different ways in training parameters. Also, based on classic methods, we added our **own approach** to optimize those methods to better fit our problem. During the process, we recorded our iterations on those methods and also our **own feeling** about pros and cons of these methods.

- 1. Trivial Training

Description:

The most direct way is to give each parameter a fixed change when error is large. The direction of change (positive or negative) is decided by some simple and trivial way (e.g: a negative change comes after a positive change and then a positive change, over and over).

Pros and Cons:

- Pros: Very easy to implement. Easy to get rid of local minimum when the graph of error equation is very steep.
- Cons: Almost impossible to find the global minimum. Hard to control the process of adjusting parameters. Hard to do minor change when error is small.

Outcome & Result:

We only tried out this method on the first two days before we go for Gradient Descent. The result we got showed large error and low performance. Therefore, we did not record detailed data about this method.

- 2. Gradient Descent

Description:

Gradient Descent is a first-order iterative optimization algorithm. Proportional steps to the negative gradient on different points of the function leads to local minimum.

Implementation and Details:

(1) Basics:

There are two key points about Gradient Descent: Function and derivative. However, our calculating model has 600 levels with dependency on neighbour levels. It is extremely hard to get a general mathematical function that describes how theoretical price was calculated. Then, without a mathematical function for the whole process, we could not directly get derivative from a mathematical function. Instead, we decided to use derivative definition to calculate partial derivative for each parameter:

$$f_x = \frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h, y) - f(x, y)}{h}$$

$$f_y = \frac{\partial f}{\partial y} = \lim_{h \to 0} \frac{f(x, y+h) - f(x, y)}{h}$$

To be specific, our error function looks like:

$$error = func\,(u,\ d,\ r)$$

For example, if we have 10 different initial price (s0 in the binomial model, stated above), we could get 10 different partial derivatives for u (same for d and r) at current point of (u, d, r) using the model we defined above. And then we could use the average of the 10 derivative to be the partial derivative of u at current point. Similar method could be found at this link.

Then, a problem of accuracy came. How large is a delta (“h” in picture) should we select? After we researched on the magnitude of parameters in binomial model, we find the up factor is usually within 1.0 to 1.8 and down factor is usually within 0.5 to 1.0. After several tests on the granularity of delta (“h” in picture), we found out 0.00001 is a delta of appropriate granularity for u and d. Selecting a large delta gives rise to low accuracy. Meanwhile selecting an extremely small delta gives rise to accuracy problem when we use partial derivative to calculate the change of parameter:

$$\Delta u = \frac{\partial u}{\partial f} * LearningRate_u$$

If we use extreme small delta for calculating derivative, then the derivative gonna have more decimal digits. However, our learning rate for each parameter also

has lots of decimal digits. So, the combination of decimal digits gives rise to low accuracy. Then, we can get to a point with lower gradient by:

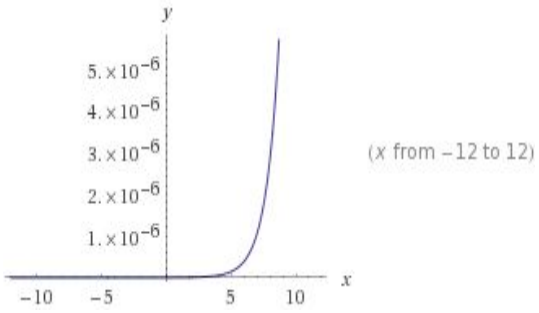
$$u_{lowergradient} = u - \Delta u \qquad d_{lowergradient} = d - \Delta d \qquad r_{lowergradient} = r - \Delta r$$

Following the same steps until we achieved expected error.

(2) Methods to speed up gradient descent:

- Adjust learning rate according to error: As we tested our code, we found out it is very slow to reach an expected error by current gradient descent process. The reason is that the learning rate for all three parameters are fixed. If our initial guesses are quite inaccurate (well, real life condition), the initial error could be over one million. Therefore, it took lots of time to get to a lower range of error. Then, we try to adjust learning rate according to the magnitude of error. After several trials, we found out this equation would be a good fit, :

$$learningRate = pow(e, error / 100000) / 10000$$



This ensures a large learning rate when error is large and a minor learning rate when error is very close to the expected error.

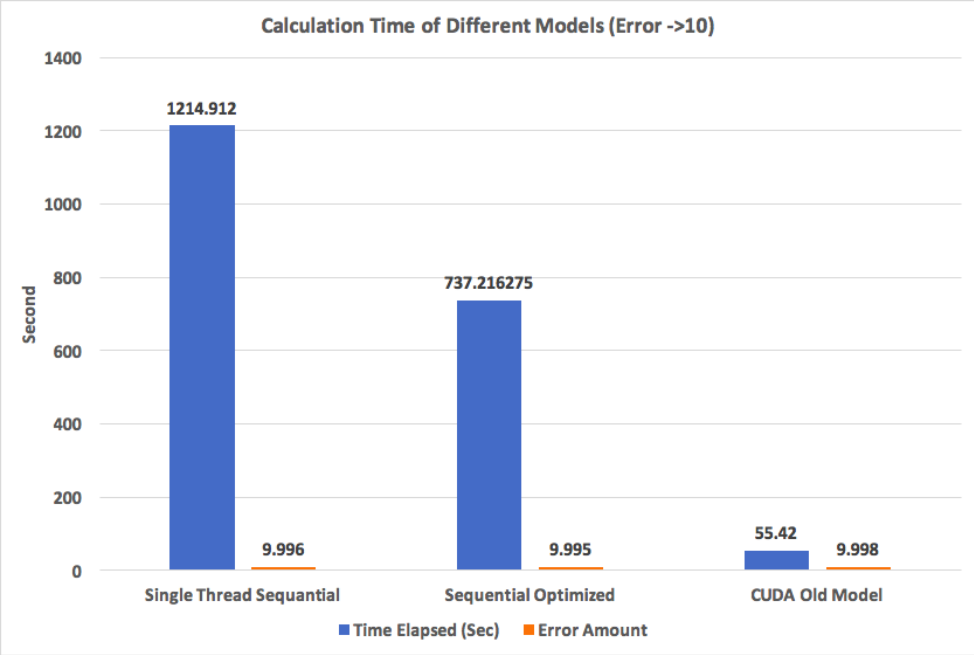
- Utilize parallel binomial model: A big bottleneck of training parameters is the calculation of derivatives using sequential binomial model. Therefore, when we finished the code of parallel binomial model, we applied parallel model to gradient descent process to calculate derivatives of a set of data in parallel. The performance graph will be showed in “Outcome & Result”.

Pros and Cons:

- Pros: Easy to implement and easy to understand. Easy to apply parallelism for calculations. Comparably good performance and accurate result.

- Cons: Gradient Descent has boundedness. It is so easy to jump to a local minima and never get rid of it. Therefore, the result of a single gradient descent process relies much on the accuracy of the local minima it converges to.

Outcome & Result:



- The first pair shows the original gradient descent process with sequential binomial model.
- The second pair shows the gradient descent process with learning rate adjustment and sequential binomial model.
- The third pair shows the gradient descent process with learning rate adjustment and CUDA parallel binomial model.

- 3. Optimized Gradient Descent:

Description:

Even though our gradient descent process could reach a low error (about 10) using CUDA parallel model within 1 minute, it is very hard to reach a lower error. When the error reaches about 9 (average of lots of speculation), it will go up and never converge again. After analyzed the derivatives, learning rate and all three parameters after thousands of calculation, we found out the graph of error function: $error = func(u,d,r)$ could be very steep around the global minima or some optimal

local minimas. Then, the derivative of each parameter is very large and a single learning rate adjustment equation is not enough to handle this situation.

Implementation and Details:

(1) Basics:

Though the normal way to do gradient descent is to follow the following equation:

$$\Delta u = \frac{\partial u}{\partial f} * LearningRate_u \qquad u_{lowergradient} = u - \Delta u$$

The essence of this method is to navigate the parameters towards a lower gradient position over and over again. The equation is just a calculation of the magnitude of change for a parameter at a time. Therefore, after the direction of movement is decided by the derivative, we could manually adjust the parameter using our own equation.

(2) Methods to optimize current gradient descent process:

- Add r to CUDA model: As we iterated on our CUDA binomial model, we found out applying a third parameter r into calculation instead of an almost fixed r will give higher accuracy of the model. The performance improvement with new CUDA model will be showed in "Outcome & Result".
- New equation for changing parameter when error is small: As stated above, there are maximum and minimum limit for up factor (u) and down factor (d) for our binomial model. Therefore, we could simply move a small percentage of the gap between current parameter value and the corresponding limit (maximum limit for positive change and minimum limit for negative change). Then equation goes like this:

  $$\Delta u = f_{percentage}(error, epsilon) * gap / f_{divisor}(error, epsilon)$$

  $$gap = abs(u - u_{max}) \; OR \; gap = abs(u - u_{min})$$

  And then, an appropriate percentage function is quite important for this method. We wanted our $f_{percentage}$ to have a value according to current error but not linear, because linear change has poor accuracy when number is very small. Also, we

need a function that has value from 0 to 1 (exclusive). After lots of test, we found out the arc tangent function is a good fit:

$$f_{percentage} = arctan(\ error\ /\ f_{divisor}(error,\ epsilon)\ )\ /\ (\pi\ /\ 2)$$

Here, the divisor function acts as a balancer to make larger change when error is large as well as very smooth change when error is small:

$$f_{divisor} = f_{divisorWeight}(error) * pow(constant_1,\ (error\ -\ epsilon)\ /\ f_{divisorBalancer}(error,\ epsilon))\ +\ constant_2$$

$$constant_1 = 0.999999999999\ \ constant_2 = 1.000000000001$$

It is obvious to see $constant_1 + constant_2 = 1$. Actually, the result of $f_{divisor}$ is close to 1.0 when error is very large, which means nearly no effect. However, when error is small, $f_{divisor}$ has a value of close to $f_{divisorWeight}$, which gives large division for $f_{percentage}$:

$$f_{divisorWeight} = 100\ -\ 90\ *\ (\ arctan\ (error)\ /\ (\pi\ /\ 2)\ )$$

$$f_{divisorBalancer} = 0.75\ *\ pow\ (\ constant_1\ ,\ error - epsilon)\ +\ 0.25$$

Function divisor weight works together with function divisor balancer to ensure a large change of parameter when error is large but very smooth change when error is very small. And we still find there is potential of speed up by modifying these two functions to more appropriate ones according to our speculation and test.

To conclude, the change of parameter in optimized gradient descent process is the minimum of these two values (u as an example):

$$\frac{\partial u}{\partial f} * LearningRate_u$$

$$f_{percentage}\ (error,\ epsilon) * gap_u\ /\ f_{divisor}\ (error,\ epsilon)$$

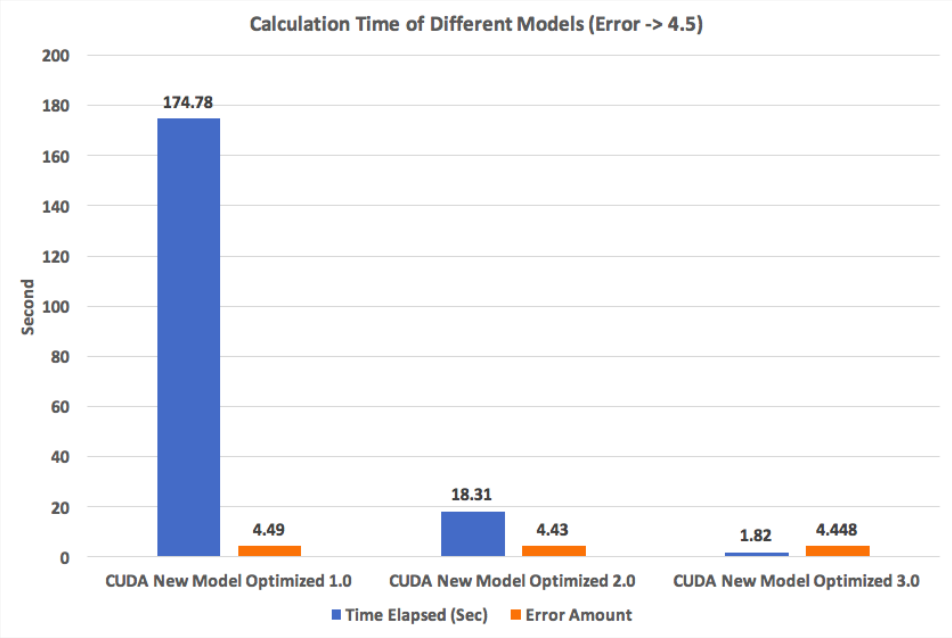
We know it is hard to digest these equations for people not participating in our project. These equations are results of thousands of tests and trials as we got more and more familiar with the thinking process of gradient descent and machine learning.

These equations is actually a good fit for our problem with the data set we get for stock option.

Pros and Cons:

- Pros: Good performance and accuracy for data set of stock option
- Cons: Not generally fit for other financial product. Still hard to get rid of local minimas.

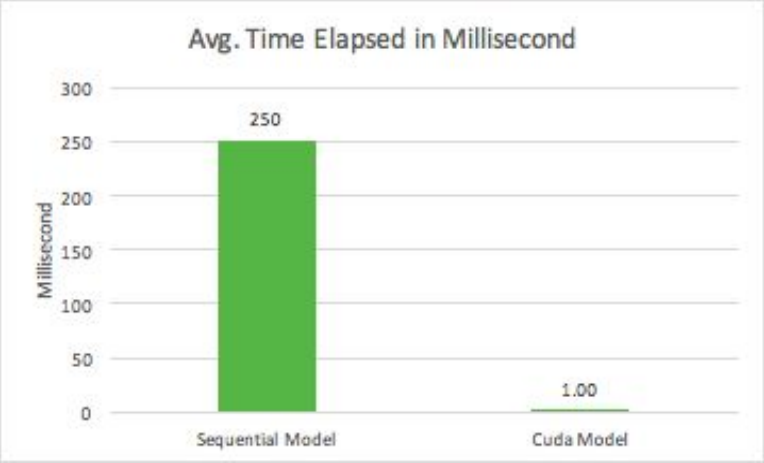
Outcome & Result:



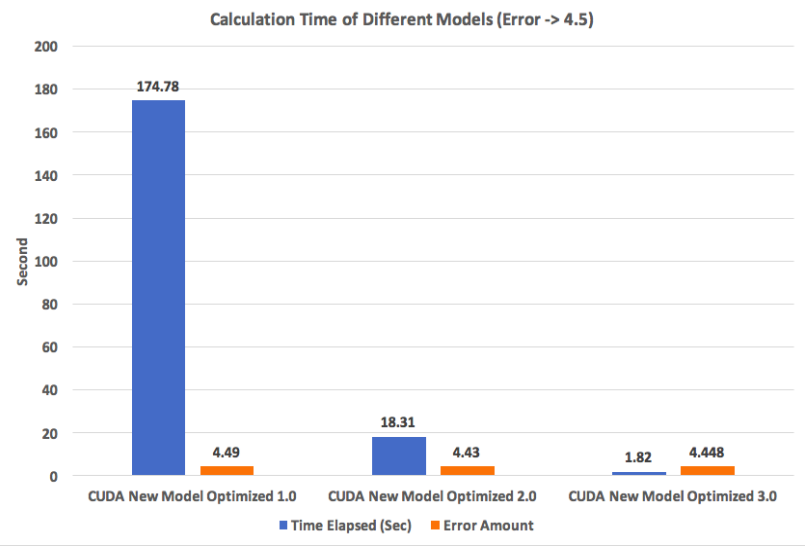
- The first pair shows the training time with CUDA new model and normal gradient descent process for error 4.5
- The second pair shows the training time with CUDA new model and earlier version of optimized gradient descent process for error 4.5
- The third pair shows the best training time with CUDA new model and latest version of optimized gradient descent process for error 4.5

Experiment and Result

This graph shows the speedup of CUDA parallel pricing model. It takes the sequential model around 250 millisecond to finish full binomial tree calculation. GPU CUDA program can finish a similar process in just 1 millisecond! (The output takes around 10 millisecond, however, we calculate a group of 10 model at the same time. The average is 1 millisecond per model each time.) We simply place a time before and after the calculation function to record the calculation time. We think this dramatic performance increase shows the power of GPU parallel computing. There's more computing cores in the GPU and those cores have high data bandwidth. The program is optimized to take advantage of locality. As a result, most of the array data stays in the L1 level cache and is shared across all cores in hardware.

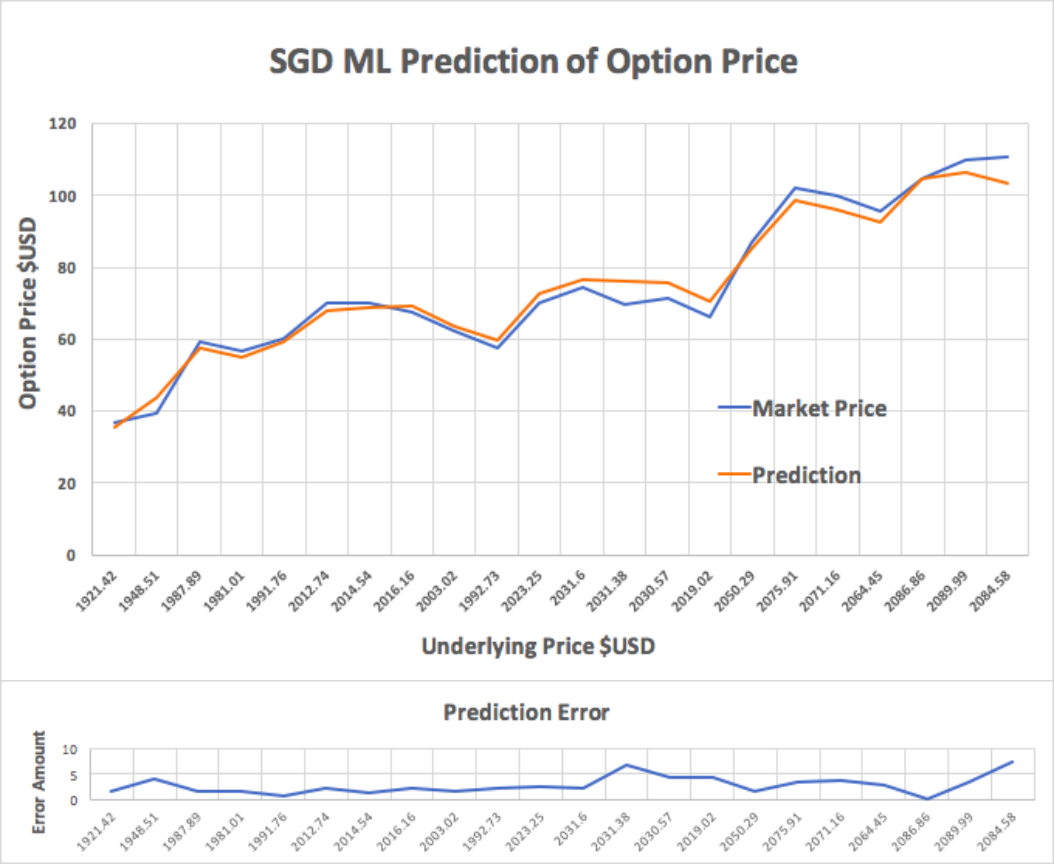


This graph on the right is the same as the figure introduced in the SGD introduction. However, this experiment pushes the model further. As you can see, the error threshold is reduced to around 4.5, which forces the machine learning algorithm to compute more. The best result that we experienced in the experiment is 1.82 second, which produces



parameters U = 1.00526483, D = 0.996038098 and R = -0.000051635. With this best result parameters, we make a prediction on the market price of the stock option. This result is as follow.

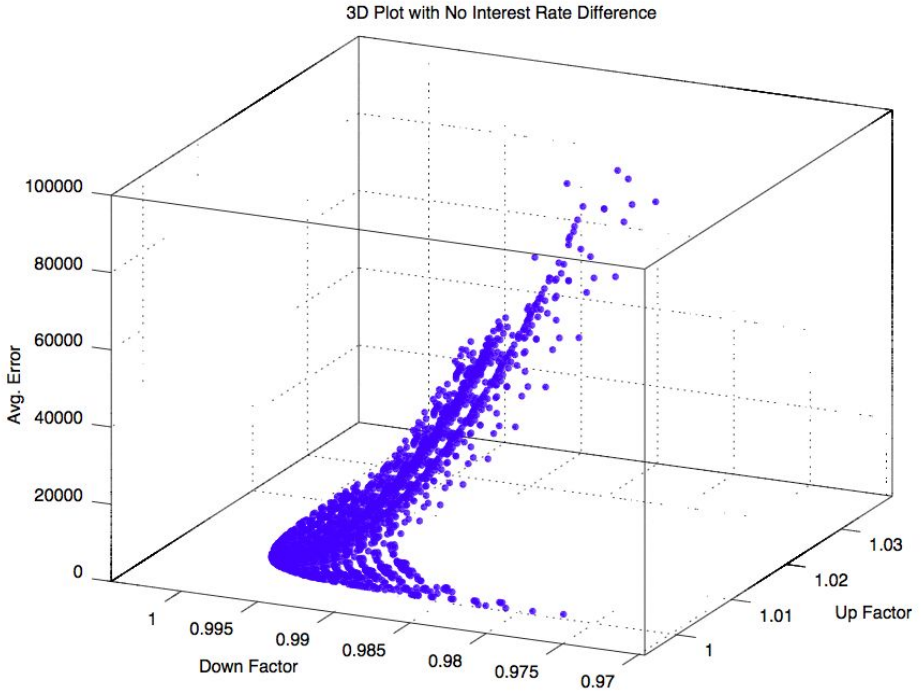
The beginning 10 data points was used as the training data in machine learning.

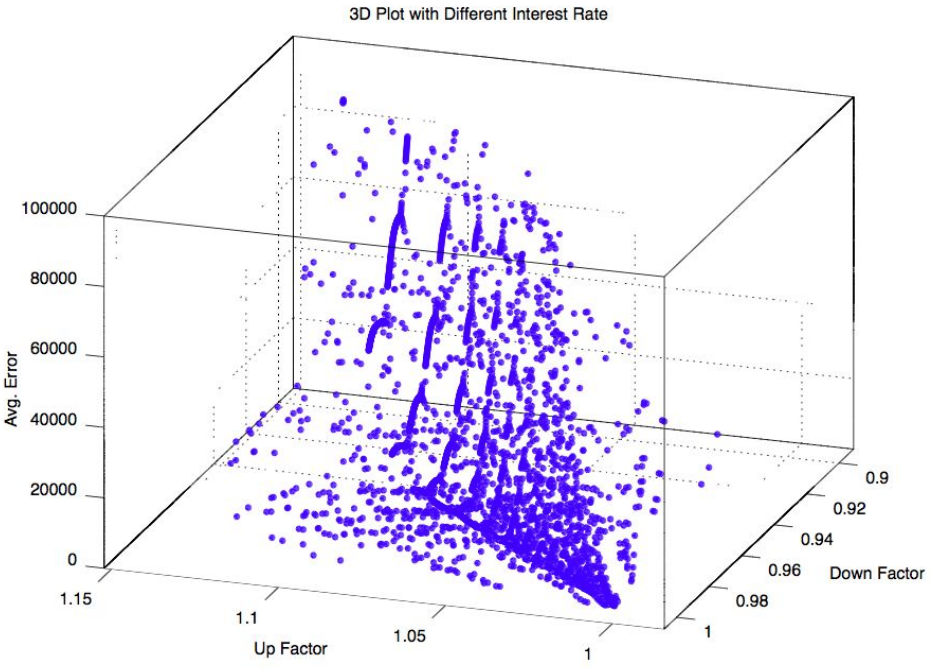


The rest of the 12 data points is the prediction of the model. The parameters withstand well against the real market data. The security that we focus on is the call option of the SPX (S&P 500 Index) that matures at 1/15/2016 with strike price of $2000. It's hard to obtain option data for free so we can only find the end of day quote from 10/1/2015 to 10/30/2015.

To visualize the machine learning process, we have tried to plot all the model calculation. We fix the z axis as a representation of the error. The x and y axises are the binomial tree model parameters. The first graph is an attempt that have the interest rate fixed. This force the binomial model to only have two dimensions of freedom. The second graph we have is an attempt that allows machine learning on all three model

parameters. Both of them shows the ideal SGD bowl shape. Every single dot on the 3D plot represent a calculation in the model. More calculation is concentrated around the global minimum at the bottom of the graph.





Lessons Learned

- Modeling

- The CUDA implementation of the binomial model works surprising well. It has performed beyond expectation. On average there is a 250 times speedup between the sequential and parallel build. This high performance makes our machine learning experiment possible. We begin to really appreciate the computing power of GPU.
- We can even try to speed up the GPU model by utilizing the block shared memories. Given the time scope of this project we can't try to implement a second CUDA model and compare the performance. Another idea that we have is to treat the calculations across binomial model level as a transformation of matrix. This approach is calculation heavy but can significantly reduce the memory requirement in the model.

- SGD:

- We learned a lot about the thinking process of machine learning. Work on ML might not need much code, but need lots of mathematical analysis and pressure tests. We spent a lot time tried and tested on different training equations and finally found out an acceptable set of equations that fit our problem and current data set. However, there are lots of work to do to make the training process suitable to larger data set and apply the training process to other financial product.
- Also, we learnt that doing ML project required strong mathematical skills. Even our project is not complex, we spent lots of time figuring out which is better and which is worse. By doing this project, our mathematical skills got improved.
- Selecting correct training model and training method is extremely important. Each model has its pros and cons. Also, each model could be utilized to diverse problems based on different iterations. There is always new things to do on old models.

- Patience and carefulness: Important for everything, but we found these extremely important for ML. One decimal digit counts and one more minute might bring you BIG BANG.
- We are surprised to have a final result of less than 2 seconds for training three parameters using CUDA. We think we are on the good track to parallelize program with lots of data and lots of iterative operations. If there is opportunity, we hope we could go deeper in this field.

Conclusions and Future Work

Given the proprietary nature of financial option data, we can only find these 22 free data points online. However, with this data, we believe we have successfully demonstrate the idea of parallel computing on binomial pricing model. Moreover, we have successfully scale up the performance using Nvidia GPU and CUDA API. The significant increase in performance helps us better conduct the machine learning on real world market data. This project serves as a prototype experiment of parallelized pricing engine and help us better understand the power of GPU.

Regarding future works, we do not apply much parallelism to the Stochastic Gradient Descent Machine Learning part. As we mentioned to Prof.Railing in the poster session, we hope we could utilize what we learnt from “Lecture 26: Parallel Deep Neural Networks” to apply parallelism to our machine learning part with better models instead of Gradient Descent. At the same time, we can increase the models stored in the GPU, growing the number from 10 models to maybe 50 models. This increase of coexisting models can allow us to perform a larger parallel computation across all models.

Acknowledgment/ Reference

Stochastic Calculus for Finance I: The Binomial Asset Pricing Model by Steven E. Shreve.

Gradient Descent Wikipedia: https://en.wikipedia.org/wiki/Gradient_descent

Introduction to Gradient Descent and linear Regression:

https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/

Distribution of Total Credit

50% - 50% credit distribution. We had great teamwork and a great time working on this project.