View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2015 Zhentao Xu

PRICING EUROPEAN OPTIONS USING MONTE CARLO METHODS

BY

ZHENTAO XU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor William Gropp

ABSTRACT

European-style options are quite popular nowadays. Calculating their theoretical price is not an easy task because there are many sources of uncertainty. However, we can model these uncertainties with random numbers. In this paper I discuss my implemention of two options-pricing programs using Monte Carlo methods, one for a CPU and the other for a GPU. I also optimize them to reduce their running time. Finally I compare the performance of those two programs.

To my parents, for their love and support.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Bill Gropp, for the continuous support of my Senior and Master's Theses, for his patience, motivation, enthusiasm and immense knowledge. I could not have imagined having a better advisor and mentor for my study.

Besides my advisor, I would like to thank my family: my parents Jianping Xu and Meijun Hua, for giving birth to me in the first place and supporting me spiritually throughout my life.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS vi

CHAPTER 1 BACKGROUND 1
1.1 Financial Options 1
1.2 Black-Scholes Model 1
1.3 Monte Carlo Method 2

CHAPTER 2 CPU ALGORITHM 3
2.1 Sequential Algorithm 3
2.2 Parallel Algorithm 5
2.3 Conclusion 7

CHAPTER 3 GPU ALGORITHM 8
3.1 GPGPU 8
3.2 The Algorithm 9

CHAPTER 4 GPU VS. CPU 14
4.1 Performance Comparison 14
4.2 Conclusion 16

REFERENCES 17

LIST OF ABBREVIATIONS

GPU	Graphics Processing Unit
CPU	Central Processing Unit
PDE	Partial Differential Equation

CHAPTER 1

BACKGROUND

1.1 Financial Options

An option is a contract which gives the buyer the right, but not obligation, to buy or sell an instrument at a specified strike price on or before a specified date. There are two kinds of options: call option and put option. They give the owner the right to buy and sell something at a specified price, repsectively.

European-style options is one of the most widely used type of options. It is an kind of option that may only be exercised on expiration. One of the most important problems in Finance is options pricing, that is, calculating the theoretical price of an option. For an European-style call option, its price usually depends on the following factors:

- S: asset value
- E: exercise price
- r: continuously compounded interest rate
- σ: market volatility
- T: expiry time of the option

1.2 Black-Scholes Model

In 1973, Fischer Black and Myron Scholes derived a PDE, which is now called Black-Scholes model [1], to estimate the theoretical price of an option. They made a few simplifying assumptions and came up with the following formula [2]:

$$C(S,t) = S \cdot N(d_1) - E \cdot e^{-r(T-t)} \cdot N(d_2),$$

where

$$d_1 = \frac{\log(S/E) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}},$$

$$d_2 = d_1 - \sigma\sqrt{T-t}.$$

and $N(\cdot)$ is Normal(0, 1) distribution function

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{s^2}{2}} \mathrm{d}s.$$

1.3 Monte Carlo Method

The Monte Carlo method [3] is a class of computational algorithms that use repeated random sampling to obtain numerical results. Since the price of an option will be impacted by various sources of uncertainty, it makes sense to model such uncertainty using random numbers. In my experiment, I will use Monte Carlo methods to price European-style call options. Specifically, I will generate M random samples and then use the average value to estimate the option price. However, the average value might not be the true option price. Therefore, I will compute the standard deviation and use it to obtain a 95% confidence interval. Mathematically speaking, let μ be the average value and σ be the standard deviation. A 95% confidence interval would be

$$[\mu - 1.96 \cdot \sigma/\sqrt{M}, \mu + 1.96 \cdot \sigma/\sqrt{M}].$$

CHAPTER 2

CPU ALGORITHM

2.1 Sequential Algorithm

Let M be the number of trials, P_i be the price at ith trial. The pseudocode of the algorithm is listed as follows:

for $i = 1$ to M **do**

$S_i = S_0 e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}\xi_i}$

$P_i = e^{-rT} \max(S_i - E, 0)$

end for

$\mu = \texttt{mean}(P)$

$s = \texttt{stddev}(P, \mu)$

In the above code, ξ_i is the ith random number from Normal(0, 1) distribution.

2.1.1 Computing Standard Deviation

There are two simple ways to compute the standard deviation, namely,

$$s = \sqrt{\frac{1}{n(n-1)}\left(n\sum_{i=1}^{n} P_i^2 - \left(\sum_{i=1}^{n} P_i\right)^2\right)} \tag{2.1}$$

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(P_i - \mu)^2} \tag{2.2}$$

I implemented both algorithms, however, their results differ:

Table 2.1: Standard Deviation

Array Size	(2.1)	(2.2)	Welford
100	0.904419	0.817973	0.813750
1000	1.014951	1.030125	1.029917
10000	1.000119	1.000029	1.000087
100000	0.999448	0.998896	0.998886
1000000	0.999972	0.999945	0.999994

I then implemented Welford Algorithm below, which is more stable according to [4]:

$U_1 \leftarrow P_1, V_1 \leftarrow 0$
for $i = 1$ to M **do**
  $U_i \leftarrow U_{i-1} + (P_i - U_{i-1})/i$
  $V_i \leftarrow V_{i-1} + (P_i - U_{i-1}) * (P_i - U_i)$
end for
$s \leftarrow V_M/(M-1)$

It turns out that Welford Algorithm's result is almost identical with (2.2). I think the reason is cancellation: we lost more significant digits when using (2.1). Although (2.2) is more accurate, it is slow because the array must be scanned to compute the mean first. Therefore, I use Welford Algorithm to compute the standard deviation in my sequential algorithm.

Below are the running time of my sequential algorithm:

Table 2.2: Running Time of Sequential Algorithm

Array Size	Time in seconds
1×10^6	0.0741999
2×10^6	0.147027
5×10^6	0.370285
1×10^7	0.740026
2×10^7	1.48209
5×10^7	3.71825
1×10^8	7.40798

2.2 Parallel Algorithm

Since the M trials are independent of each other, it is possible to parallelize the algorithm using pthreads to make it run faster. To do this, I created an array of size M to store the result of M trials. Then I divide the array into n segments where n is the number of threads. Each thread will calculate the Black-Scholes value in each array entry in its portion and sum them up. After this I will add those partial sums up and compute the mean. In the end, I will use the mean and the trial array to compute the standard deviation. The pseudocode of the parallel algorithm is listed below:

Let n be the total number of threads, k be the current thread number.
$start_index \leftarrow k(M/n)$,
$end_index \leftarrow \min(M, (k+1) * (M/n))$,
$partial_sum \leftarrow 0$,
for $i = start_index$ to $end_index - 1$ **do**
$S_i = S_0 e^{(r - \frac{1}{2}\sigma^2)T + \sigma\sqrt{T}\xi_i}$
$P_i = e^{-rT} \max(S_i - E, 0)$
$partial_sum \leftarrow partial_sum + P_i$
end for
$sum[k] \leftarrow partial_sum$

Below are the running time of my program using two, three and four threads respectively:

Table 2.3: Running Time Comparison

Array Size	Two Threads	Three Threads	Four Threads
1×10^6	0.0369596	0.0248208	0.0189137
2×10^6	0.073669	0.049392	0.037178
5×10^6	0.187248	0.127107	0.0959623
1×10^7	0.378646	0.251982	0.191258
2×10^7	0.749793	0.501524	0.380512
5×10^7	2.15965	1.26027	0.944356
1×10^8	4.50759	2.80522	1.89112

Figure 2.1: Running Time

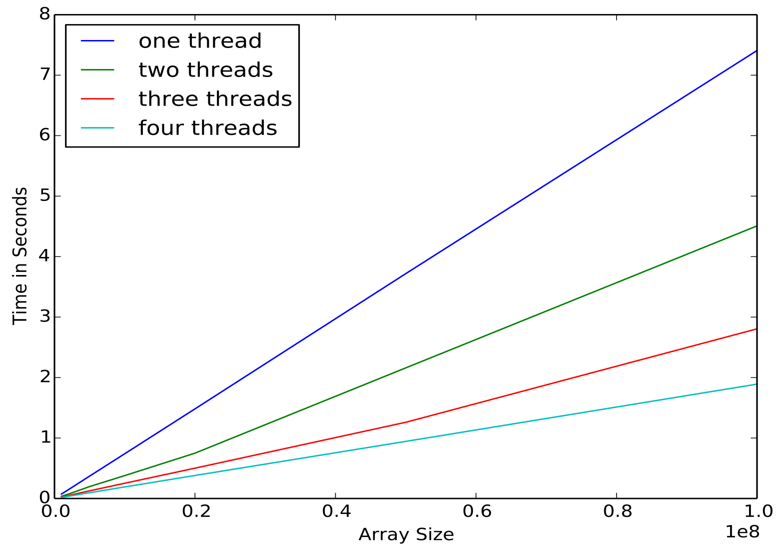
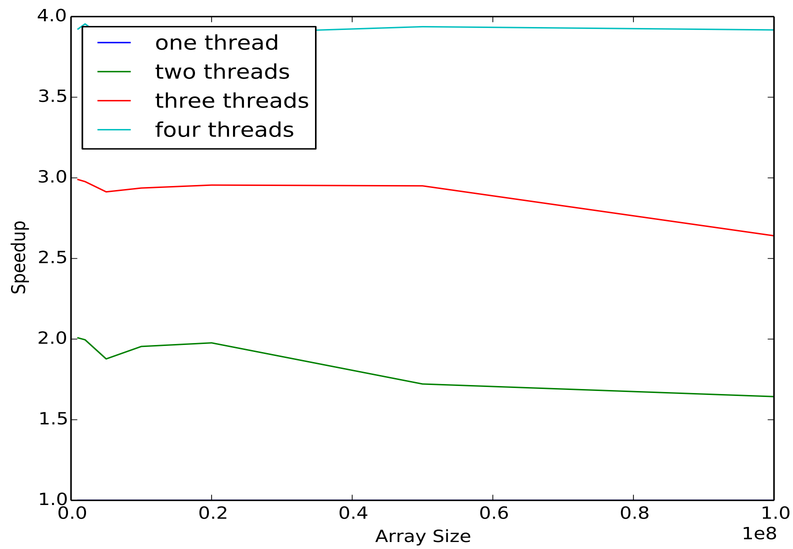


Figure 2.2: Speed up



2.3 Conclusion

Let n be the number of threads of execution, $B \in [0, 1]$ be the fraction of algorithm that is strctly serial. According to Amdahl's law [5], the theoretical speed-up $S(n)$ we can get by running the algorithm on n threads is

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)}.$$

In my program, B is small. The serial part of the algorithm merely parses arguments and allocates/deallocates memory. Therefore, the theoretical speed-up should be very close to the number of cores, as observed in the experiments.

CHAPTER 3

GPU ALGORITHM

3.1 GPGPU

General-Purpose computing on Graphics Processing Units, or GPGPU, is the utilization of GPU to perform computation in applications traditionally handled by the CPU. GPU does extremely well in processing large amounts of data. As such, I implemented the Black-Scholes model on GPU.

Different GPUs have different capabilities. My computer is equipped with NVIDIA GT 750M. I can run the algorithm on it without many difficulties.

The architecture of a GPU is very different from that of a CPU. In a GPU, there are a few multiprocessors, each of which contains some stream processors. For example, my GPU has 2 multiprocessors and 384 stream processors. Each processor can execute a sequential thread. However, the code is actually executed in groups of 32 threads, which is called a warp. There are a few facts [6] that should be taken into consideration when developing programs for NVIDIA GPUs:

- Threads are grouped into blocks which are grouped into grids.
- Each thread and block has a unique local index in its block and grid, respectively. These indices are usually used to compute array indices.
- If one or more threads in a warp is executing a different instruction from others, the warp must be partitioned into groups of threads based on instructions being executed. The groups of threads are executed one after the other.

3.2 The Algorithm

3.2.1 Generating Random Numbers

In the CPU algorithm, each thread will generate a random number and immediately use it to compute Black-Scholes value. On the GPU, however, I am using the cuRAND library to generate all M random numbers from a Normal(0, 1) distribution and store them in the array before calling the kernel to compute the Black-Scholes value for each array element. Below is the C code of the algorithm that shows how this is done:

Listing 3.1: Generate Random Numbers on GPU

```
double *trials_d;
curandGenerator_t gen;
cudaMalloc((void**)&trials_d, M * sizeof(double));
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(gen,(int)time(NULL));
curandGenerateNormalDouble(gen, trials_d, M, 0.0, 1.0);
```

3.2.2 Computing Black-Scholes Values

As I mentioned in the above section, threads should be grouped into blocks. How they are grouped will impact the performance of the program. CUDA supports 1D, 2D and 3D blocks. I am going to group the threads into 1D blocks because I am using a 1D array to store the Black-Scholes values. Then I will launch a kernel to compute Black-Scholes values. At this time, `trial_d` array has been filled with random numbers generated in the last section.

Listing 3.2: Launching Kernel

```
int number_of_blocks = (M / thread_per_block) +
    (M % thread_per_block == 0)? 0 : 1;
black_scholes_value<<<number_of_blocks,
                        thread_per_block>>>
    (trials_d, S, E, r, sigma, T, M);
```

In `black_scholes_value` function, each thread will figure out the part of the `trials_d` array it has to process. Below is the CUDA code that does it. It

begins by calculating how many array entries each thread has to process. Then it will figure out the start and end indices for the entire block and thread. In the end, it uses a for loop to calculate Black-Scholes values.

Listing 3.3: Black Scholes Value

```
int idx;
int elements_per_thread = (M / (gridDim.x * blockDim.x))
    + ((M % (gridDim.x * blockDim.x) == 0)? 0: 1);
int block_start_index = elements_per_thread * blockIdx.x
    * blockDim.x;
int block_end_index = min(elements_per_thread *
    (blockIdx.x + 1) * blockDim.x, M);
int thread_start_index = block_start_index
    + threadIdx.x * elements_per_thread;
int thread_end_index = min(block_start_index +
    (threadIdx.x + 1) * elements_per_thread,
    block_end_index);
double current_value;
for(idx = thread_start_index; idx < thread_end_index;
    idx++)
{
    current_value = S * exp((r - (sigma * sigma) / 2.0)
        * T + sigma * sqrt(T) * trials_d[idx]);
    trials_d[idx] = exp(-r * T) *
        ((current_value - E < 0.0) ? 0.0 :
        current_value - E);
}
```

The value of `thread_per_block` will impact the performance of the kernel. Below are the results of the experiments I have done when setting $M = 100000$:

Table 3.1: Different Values of `thread_per_block`

Thread Number Per Block	Time in Milliseconds
1024	2.772768
512	2.492928
256	4.648800
128	5.181248
64	6.055808
32	6.478784
16	14.119616
8	23.309153
4	38.754753

The code is executed in groups of 32 threads, therefore, when the thread number per block is less than 32, the performance is severely impacted.

3.2.3 Reduction

In the last chapter, I showed that there are three ways to calculate the standard deviation. One of them is unstable. In the other two algorithms, the Welford Algorithm computes the ith estimate of the standard deviation based on $(i-1)$th estimate, which cannot be easily adapted to run on GPU. Therefore, I will use

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{M}(P_i - \mu)^2}$$

to compute standard deviation.

The entire process can be decomposed into four steps:

1. sum the whole array and compute the mean
2. subtract mean from each array entry and square it
3. sum the whole array and compute the mean
4. divide by $(M-1)$ and then take the square root

The first step corresponds to computing μ; the second step corresponds to computing $(P_i - \mu)^2$ for each array entry and the third step corresponds to computing $\sum_{i=1}^{M}(P_i - \mu)^2$.

The second step is not difficult. Just like `black_scholes_value`, each thread process a small portion of the entire array. The fourth step is straight forward as well. The difficult part are steps one and three, which involve reductions.

The maximum number of threads in a block is 1024. Therefore inside each block I created an array of length 1024 in shared memory. The reason I use shared memory is because I need to visit some array entries more than once, visiting shared memory is much faster than visiting global memory. Then I set all the entries to be 0. After that I use a for loop to reduce the whole array. Below is the C code that does it.

Listing 3.4: Reduction

```
__shared__ double sdata[1024];
int tid = threadIdx.x;
/*each of the 1024 threads will sum a portion of
  the trials_d array and store the result in sdata
  array*/
int shared_memory_entry_number = /*actual number
    of array entries in the sdata array*/;
for(int stride = shared_memory_entry_number >> 1;
    stride > 0; stride >>= 1,
    shared_memory_entry_number >>= 1)
{
    if(tid < stride)
    {
        sdata[tid] += sdata[tid + stride];
    }
    if(tid == 0 && shared_memory_entry_number & 1 == 1)
    {
        sdata[tid] += sdata[shared_memory_entry_number
            - 1];
    }
    __syncthreads();
}
```

In the code above, `shared_memory_entry_number` is the actual number of array entries in the `sdata` array, `stride` will be initialized to half of it. Each

array entry of `sdata` whose index idx is greater than `stride` will be added to `sdata[`idx`-stride]`. If `shared_memory_entry_number` is an odd number, then the first thread will add the last array entry to the first entry. Each iteration will reduce the length of the array by 50%. Therefore it is an $O(\log n)$ algorithm.

CHAPTER 4

GPU VS. CPU

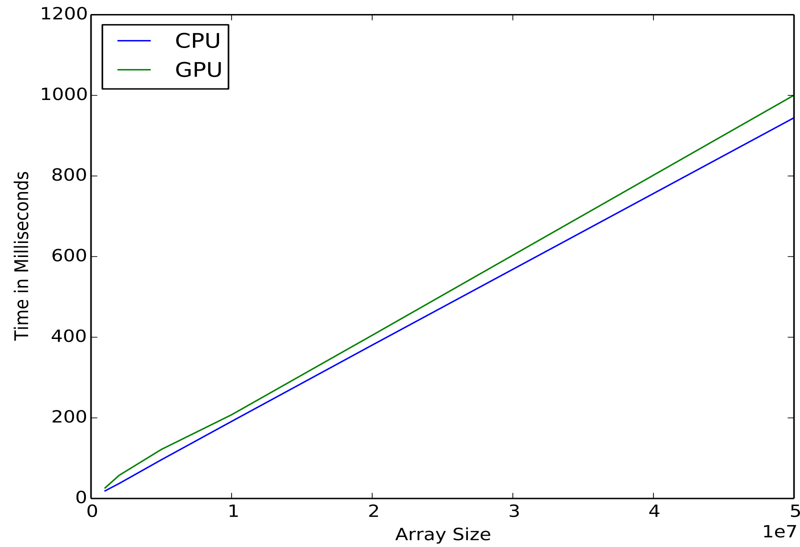
4.1 Performance Comparison

For the same array size M, the GPU actually performs a little worse than CPU. This is reasonable because the GPU program spends a lot of time doing reductions, which is memory intensive and does not utilize the computation power of GPU.

Table 4.1: CPU vs. GPU 1

Array Size	CPU Time	GPU Time
1×10^6	0.0189137	0.0262843
2×10^6	0.037178	0.0574118
5×10^6	0.0959623	0.1216183
1×10^7	0.191258	0.2077592
2×10^7	0.380512	0.4049033
5×10^7	0.944356	1.0003793

Figure 4.1: CPU vs GPU 1

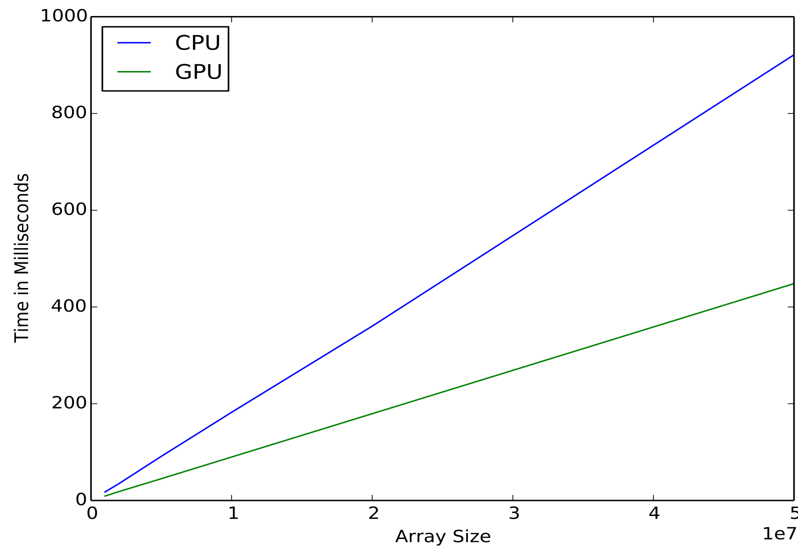


If we only compare the time they use to calculate the Black-Scholes values, things are quite different:

Table 4.2: CPU vs. GPU 2

Array Size	CPU Time	GPU Time
1×10^6	0.0177	0.00917
2×10^6	0.035125	0.018541
5×10^6	0.0912769	0.0450343
1×10^7	0.1823	0.0898428
2×10^7	0.360406	0.1793895
5×10^7	0.921276	0.4482882

Figure 4.2: CPU vs GPU 2



4.2 Conclusion

The GPU performs very well in computationally intensive tasks. Calculating Black-Scholes values requires a lot of floating-point operations but relatively few memory operations. Hence GPU outperforms CPU. However, things are different when calculating the mean and standard deviation. In this case, the CPU does better. Sometime it makes sense to decompose the task into different parts and run computationally intensive tasks and memory intensive tasks on GPU and CPU, respectively.

REFERENCES

[1] Wikipedia, "Black-Scholes model - Wikipedia, the free encyclopedia," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model

[2] D. J. Higham, "Black-Scholes Option Valuation for Scientific Computing Students," *Computing in Science and Engineering*, vol. 6, pp. 72–79, 2004.

[3] Wikipedia, "Monte Carlo methods for option pricing - Wikipedia, the free encyclopedia," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Monte_Carlo_methods_for_option_pricing

[4] D. Knuth, *The Art of Computer Programming, Vol 2, 3rd edition.* Reading, MA: Addison-Wesley, 1997.

[5] Wikipedia, "Amdahl's law - wikipedia, the free encyclopedia," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Amdahl%27s_law

[6] M. Wolfe, "Understanding the CUDA data parallel threading model," 2012. [Online]. Available: https://www.pgroup.com/lit/articles/insider/v2n1a5.htm