Parallelization of European Monte-Carlo Options Pricing on Graphics Processing Units

Chat Niramarnsakul, Prabhas Chongstitvatana and Mark Curtis
Department of Computer Engineering and Department of Finance
Chulalongkorn University Bangkok, Thailand
Relativity.c@gmail.com, Prabhas.c@chula.ac.th, Markpcurt@yahoo.co.uk

***Abstract*— Using GPU computing for option pricing has been a critical problem for a long time, specifically, in Monte Carlo simulation which is the most widely used solution for option pricing problem. In general, option pricing must be performed in real time. Recent multi-core CPUs can provide a high computing power, but the option pricing on recent multi-core CPUs is far from responding in real time. The development of Graphic Processing Units promises a much higher computing power than multi-core CPUs for specialized problems. In this paper, we present methods to compress the data inputs for GPU in computing European options pricing which save large memory bandwidth and give the results in an acceptable time. The experimental result shows that the overall speedup is about 900X.**

Keywords-GPU computing; Monte Carlo; speedup; parallelization;

I. INTRODUCTION

More and more time consuming on options pricing model have been making a big hit on financial engineering on these consecutive years because of the numerous exercising dates and several numerical conditions[1]. The demand of a varieties in quantitative pricing models [2], and the use of complicated numerical methods, usually takes several hours to get a solution[3]. Monte Carlo simulation is widely accepted due to its broad applicability to those pricing models with complex boundary conditions [4, 5].

In this paper, we choose Monte-Carlo to solve the Black Scholes model because of its ability to be parallelized naturally. Nevertheless, the amount of price paths that can be simulated to quickly respond to the dynamic market in real-time heavily relies on the data processing capability of the processor. For these intensive computational applications, a stream processor has an absolute advantage over the traditional CPU [6]. Many previous works[7] reported the result of speed up of CPU with the annual increasing rate of 1.4x for, known as Moore's Law, and compared the result with GPUs. GPUs computational capability has been compounding at an annual rate of 1.7x. This work reports the implementation of Monte-Carlo method to solve the Black Scholes model on GPU. The experimental result shows that the overall speedup is 900X.

II. RELATED WORK

Craig et al.[8] gave a simple implementation for Black-Scholes Model and Binomial Model with Cg, and got a 10x speedup compared with CPU implementation. Stanimire [9] implemented two probability-based simulation models, Ising model and Percolation model, and received a 3x speedup compared to CPU.

After a GPU programming more user-friendly RapidMind was invented, Michael et al. published their experimental results on the implementation of European options pricing via quasi-Monte Carlo sampling and reported a 30x speedup over the CPU implementation, which proves the high efficiency of RapidMind.

In 2007, NVIDIA released their new product with the computing unified device architecture (CUDA) which provides much better support for general purpose computation and greatly relieves the pain of traditional graphics programming. Among those works based on CUDA, Gregoire et al. [10] implemented a Trinomial Option Pricing Model which is 31.9 times faster than the CPU implementation.

NVIDIA released [11] CUDA SDK, an MC-based solution for European options pricing, which achieved a peak performance of 10 billion path samples per second. However, it didn't show performance in term of speed up compared to CPU, it only showed the optimal value of path per option. For that reason, it has to be some optimal point which we can improve the speed up compared to CPU. We investigate many methods and parameters tuning to improve the performance on an ordinary PC, Nvidia Geforce8600GT with Intel duo core processor.

III. EUROPEAN OPTION PRICING WITH MONTE CARLO SIMULATION

A Monte Carlo model generates random price paths of the underlying asset, each of which results in a payoff for the option. The average of these payoffs can be discounted to yield an expectation value for the option[12]. According to the law of large numbers, the more price paths are calculated, the more accurate the expectation value will be.

$S(t)$ depicts the price of a certain stock at time t. A European option gives the holder the right to buy the stock

at a fixed or strike price X at future time $t = T$, the current time being $t = 0$. If at time T, $S(T) > X$ the holder is able to exercise the option for a profit of $S(T) - X$. However, if $S(T) \leq X$ the option expires worthless. The 'payoff' p to the option holder at time $t = T$ is therefore as shown in (1).

$$p = \max(0, S(T) - X) \tag{1}$$

(1) is known as the 'payoff function', where max() returns the maximum of two input values.

To find the expected value E of the option at current time $t = 0$, a continuously compounded interest rate r should be considered. The expected value of the option to the holder at time t can be shown in (2).

$$E = e^{-rT} p \tag{2}$$

In calculating the expected value of the option[13], T, X and r are parameters already known. Random variable $S(T)$ is unknown, but it may be characterized by using the well known Black-Scholes model via N is a Gaussian distributed(Normal distributed) random variable of zero mean and unit standard deviation with known volatility μ. This is shown in (3).

$$S(T) = S(0)e^{(r-\frac{1}{2}\mu^2)T+\mu\sqrt{T}N} \tag{3}$$

As $S(0)$ is the current price of the stock it is also a known quantity. Substituting into (1) and assuming a sequence of n Gaussian random numbers $N1$, $N2$...Nn, the calculation shown in (4) can be used to find p_n.

$$p_n = \max(0, S(0)e^{(r-\frac{1}{2}\mu^2)T+\mu\sqrt{T}N_n} - x) \tag{4}$$

In[14] each p_n can be assigned in each step, usually called number of simulation n. Means of p_n can be gathered by summation of each p_n divided by number of simulation. Multiplied by(2) to make it become present value E(5).

$$E = e^{-rT\frac{\sum_{i=1}^{n} p_n}{n}} \tag{5}$$

This final value E is the price of European call option.

IV. GPU IMPREMENTATION

The implementation of GPU is divided into three steps.

Step1 : Managing set of normal distributed random values. The Mersenne Twister [15] is used to generate random numbers that are uniformly distributed between 0 and MAXINT ($2^{32} - 1$).The uniformly distributed random numbers are transformed into a Gaussian (normal) distribution with $\mu = 0$ and $\sigma = 1$. Then allocate these uniform distributed values into size-n array where n is the number of simulation noticing that if n is a large number, memory allocation has to be sufficient enough to hold a large array. Once the array completely allocated, then they are ready to be sent into GPU.

Step2 : Parallelization on GPU. At the beginning step, all data including T, X, r, μ are already kept by GPU registers to make an advantage on timing fetching data in or out of GPU. According to (3), this equation can be parallelized to each thread because each p_n can be calculated individually.

```
for (i = threadID; i < n; i+=next threadID)
{
   for (j = 0; j < n; j+=1)
   {
      Calculate option_j pn in step i
      Check if pn price outperform strike price
            If outperform, sum it to array_j pn
   }
}
```

Figure 1. Parallelization on GPU

Challenging emerges in this step due to the capability of GPU's memory concerning number of options input that can be run simultaneously in each thread. As the number of simulation increases lesser number of options input can be performed because of memory insufficiency and inadequate number of register. In this paper, the maximum input that can concurrently be run in each thread is 64 including normal distributed random array at a million number of simulations.

Step3 : Gathering and merging each p_n.

```
for (i = blockID; i >0; i /=2)
{
   for (j = 0; j < n; j+=1)
   {
      if threadID < i
      sum pn j +=  data j in threadID+ 1
   }
}
for (k = 0; k < n; k+=1)
{
   Collect each option value p from array
   k at BlockID = 0
   For Option k, p Multiplied by (5)
}
```

Figure 2. Gathering *pn* Algorithm

Algorithm starts from gathering data in every block by a factor of 2 until no data left in other blocks except in the first block. The algorithm works consecutively on each p_n by gradually summing up data in each thread in each block. When the process is finished, get p for each option j from the BlockID0. Each European call option can finally be calculated by algorithm in Fig.2 after multiplied by (5)

V. PERFORMANCE EVALUATION

TABLE I. COMPUTATION TIME IN MILLISECOND

NO_SIMS	1000000						
input(n)	1	2	4	8	16	32	64
CPU	8574	16156	32047	64998	129288	258376	522974
TpB 16	78.7	86.8	100.8	149.2	198.3	364.7	996.6
TpB 32	80.2	83	99.6	161.1	206.7	287.5	1432.5

Table 1 shows computation time in millisecond(ms) calculated by CPU Intel duo core processor @ 2.2 GHz and GPU Nvidia Geforce8600GT at Thread per Block(TpB) 16 and 32. If we increase TpB more than 32, it will lead to incorrect result due to the lack of registers. This topic is out of this project scope. As the number of option increases CPU takes more computation time distinctly compared to GPU which gradually affects the computation time.

TABLE II. OVERALL SPEED UP

NO_SIMS	1000000						
input(n)	1	2	4	8	16	32	64
S_TpB16	108.9	186.1	317.9	435.6	651.9	708.4	524.7
S_TpB32	106.9	194.6	321.7	403.4	625.5	898.6	365.1

Table 2 describes the maximum point of speedup, as number of input escalates the speedup factor is also increase. However, at 64 inputs both factors in TpB16 and TpB32 are less than the point at 32 inputs. This point is the optimal point of speedup with a speedup factor of 898.6 for TpB 32. The optimal point is at 32 inputs. Computation time for each option is 8.98 milliseconds. It can be calculated as 111.3 options per second or 10^8 paths(number of simulation) per second shown in Tab.2.

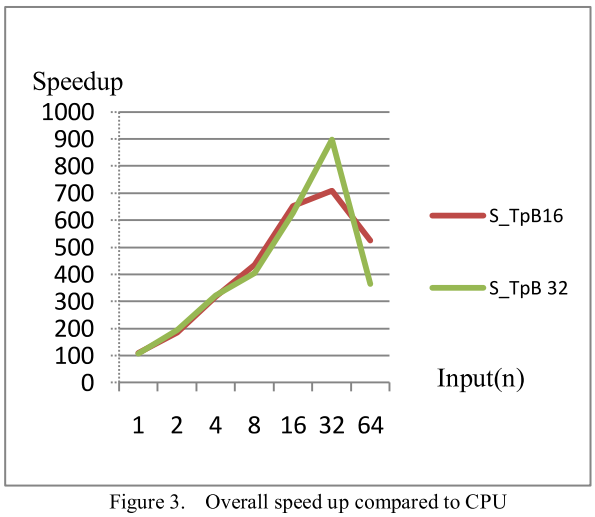

Figure 3. Overall speed up compared to CPU

VI. CONCLUSION

This work shows the algorithm design and implementation of a parallel Monte Carlo option pricing model on Nvidia Geforce8600GT architecture. The result of GPU implementation outperforms CPU implementation in term of computation time, number of option and speedup. The speed up is as high as 900X. Noticing that reconfigurable technology cannot always be efficacious or that GPU may not be used efficiently. Generally, finance computing is based on cost and utilization, therefore a powerful GPU might not always be suitable as the first choice. Nevertheless, it is worth mentioning that work shows massively parallel computing is suitable and is highly efficient for this problem. It remains a challenge for future research in computing a large set of option pricing models.

REFERENCES

[1] Tangman D Y, Gopaul A, Bhuruth M. Numerical pricing of options using high-order compact finite difference schemes. J Comput Appl Math, 2008, 218: 270–280

[2] Samuli I, Jari T. Efficient numerical methods for pricing American options under stochastic volatility. Num Method Part Differ Equ, 2007, 24: 104–126

[3] Black F, Myron S. The pricing of options and corporate liabilities. J Politic Econ, 1973, 81: 637–654

[4] Phelim P B. Options: A Monte Carlo approach. J Financ Econ, 1977, 4: 323–338

[5] Acworth P, Broadie M, Glasserman P. A comparison of some Monte Carlo and Quasi-Monte Carlo methods for option pricing. In: Proceedings of the 1996 Conference on Monte Carlo and Quasi-Monte Carlo Methods. New York: Springer, 1998

[6] NVIDIA CUDA Programming Guide v2.0, 2009, 1: 10–11

[7] John D O, David L, Naga G, et al. A survey of general-purpose computation on graphics hardware. In: Eurographics, 2005, State of Art Reports. 2005. 21–51

[8] Craig K, Matt P. Options pricing on the GPU, GPU Gems 2, Chapter 45, Addison Wesley Professional, 2005

[9] Stanimire T, Michael M, Robert B, et al. Benchmarking and implementation of probability-based simulations on programmable graphics cards. Comput Graph, 2005, 29: 71–80

[10] Gregoire J, Ecole C P, Tuan N, et al. Parallelized trinomial option pricing model on GPU with cuda, 2008, available at http://www3.imperial.ac.uk

[11] Victor P, Mark H. Black-Scholes Option Pricing. NVIDIA CUDK SDK, June 2007

[12] P. Glasserman, *Monte Carlo Methods in Financial Engineering*.Springer Science, 2000.

[13] F. Black and M. Scholes, "The pricing of options and corporate liabilities," in *Journal of Political Economy*, vol. 81, no. 3, 1973, p. 637 to 654.

[14] Bongki M, Joel S. Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In: Proceedings of the Scalable High Performance Computing Conference, Knoxville, TN, 1994. 176–183

[15] Makoto Matsumoto and Takuji Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. On Modeling and Computer Simulation*, 8(1):3-30, January 1998.