Efficient Asian Option Pricing with CUDA

Artur Yuzhanin, Ivan Gankevich, Eduard Stepanov, Vladimir Korkhov
Faculty of Applied Mathematics and Control Processes
St. Petersburg State University
St. Petersburg, Russian Federation
artur.yuzhanin@gmail.com, igankevich@ya.ru, e.an.stepanov@gmail.com, vladimir@csa.ru

POSTER PAPER

In this paper the Monte Carlo methods of the Asian option pricing are considered. Among them are pricing method with path integral and partial differential equation. Simulation algorithms running on the CPU sequentially and algorithms running on the GPU in parallel using the CUDA technology were analyzed and compared.

Asian option; Black–Scholes–Merton model; Black–Scholes equation; path integral; partial differential equation; CUDA; parallel computing;

I. Introduction

In order to solve scientific and engineering problems, amounts of computations increases extremely fast, and therefore new approaches, such as new mathematical models and computing technologies are required to be improved or created, to cope with assigned tasks.

For that purposes in economics Fischer Black, Myron Scholes and Robert Merton developed Black–Scholes–Merton model in 1973, which gives theoretical estimate of the price for European-style options with Black–Scholes equation [1], [2]. Unfortunately, in the case of non-European-style options, for example Asian, which is kind of exotic option, the analytical solution of Black – Scholes equation is not possible to deduce, so the price of non-European-style options cannot be obtained. In order to solve this problem, Monte Carlo methods (MC) were developed. The implementation of MC requires a lot of computational power.

To speed up the computation-intensive tasks (such as MC simulations), new approaches, such as SIMD (Single Instruction Multiple Data) and SIMT (Single Input Multiple Thread), have been suggested. Large computing clusters implement SIMD, whereas SIMT is implemented in GPGPU, for example, by Nvidia CUDA. CUDA supports Linux and Windows operation systems and a several programming languages C++, Python and Fortan. Since CUDA released, it has been successfully used in problems of computation in finite difference schemes for differential equations [3], objects recognition problems [4], neural networks problems [5], etc. Therefore, we tried to decrease the computational costs of the MC methods of the Asian option pricing with CUDA technology.

II. Gentle introduction to options

Option is a contract, under which a buyer acquires the right to buy or sell the financial instrument, called the underlying asset, at a pre-agreed price at some point of time in the future, which is determined by the contract. The seller sells his obligation to buy or sell an asset.

The Black – Scholes equation in the case of the Asian option.

$$\frac{\partial C}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} + rS\frac{\partial C}{\partial S} + S\frac{\partial C}{\partial A} - rC = 0$$

The Problem of the option pricing involves the determining the following basic definitions and designation:

- S (S_0) is the (current) price of the underlying asset,
- r is the risk-free interest rate,
- σ is the volatility (the statistical measure of the tendency of the variability of the price),
- $C(S, A, t)$ is the option price,
- $A(S, t)$ is the summed set of the prices of the asset at the every moment of the time of the monitoring,
- t is a time in years.

To estimate theoretical price of an option the following are also required:

- T is the expiration (the date or the period of time, in which a seller is to fulfill the obligations under the option contract),
- K is the exercise price of an option at which the owner of the option can buy or sell the asset in the future.

The Asian option is based on the idea of the amount of the payment, which is determined by the average price of the asset for a period. In majority cases, the average price is defined as the arithmetic average:

$$A(0,T) = \frac{1}{T}\int_0^T \omega(t)dt$$

In the case of discrete points of monitoring at the moments in time $(t_1, t_2, \ldots, t_n)$:

$$A(0,T) = \frac{1}{T}\sum_{i=0}^{n} \omega(t_i)$$

III. Preparing computation

TABLE I. Device's characteristics

Specification type	Value
Model	GeForce GT 630M 2 GB GDDR5
Die name	GF117 GL
Architecture	Fermi
Fabrication process	40 nm
Memory clock (effective)	900 (3600)MHz
Memory bandwidth	57.6 GB/s
CUDA Compute Capability	2.1
Shared Memory Size	48 KB
CUDA cores	96
Single presision compute power	307.2 GFLOPS

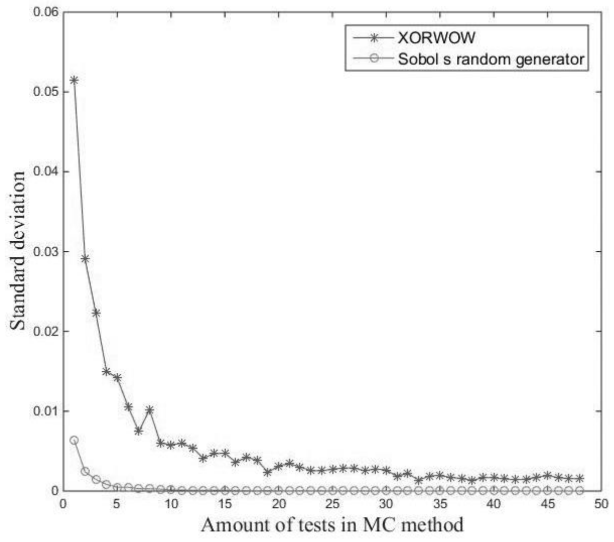

Figure 1. The dependence of the standard deviation of a European option prices on the number of tests in the Monte Carlo. Stochastic differential equations.

An important point in a MC simulation is the choice of the random number generator. CUDA provides cuRand library and five random number generators, four of which are pseudo-random number generators (PRNG) and one is quasi-random number generator (QRNG) [6].

There are several fundamental differences between implementations of a QRNG and a PRNG. When using a PRNG, the probability of the generating each random variable in the range $[0, RAND\_MAX]$ is constant. Generation of some random variable does not affect the probability of being generated in the future steps.

Thus, we can say the random sequences created by a PRNG are statistically independent.

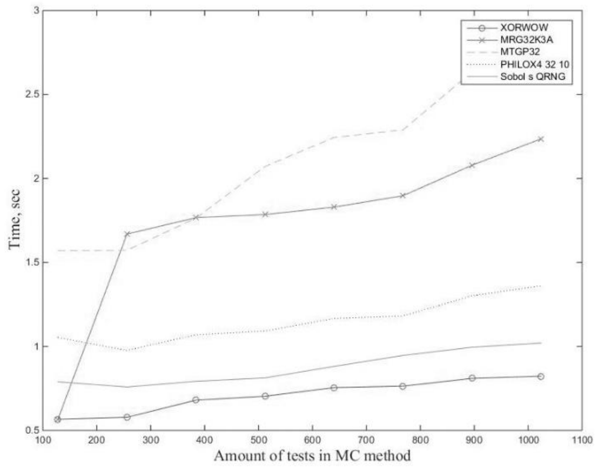

Figure 2. The dependence of the elapsed time of CUDA algoritm on amount of test in MC. Stochastic differential equations. European option.

In the case of a QRNG usage, when some random variable is generated, the probability of getting the same random variable at the next step decreases. Therefore, the random variables generated by QRNG are more evenly distributed within the generation interval, that also has a positive effect in the case of a QRNG usage in a MC simulations [7], [8], [9]. Figure 1 confirms that fact: in the case of the Sobol's QRNG usage the standard deviation of the option price decreases faster than in the case of the XORWOW PRNG usage. Figure 2 also illustrates what the Sobol's QRNG is almost the fastest RNG presented by cuRand. Therefore, the Sobol's QRNG is used throughout this paper.

In addition, we tested every algorithms with sets of data, where every parameter is the uniformly distributed random variable over the several intervals.

We implemented serial algorithms with pure C++.

IV. Monte-Carlo Methods of numerical processing and CUDA implementation

The trivial model of the parallelization is to transfer all input data (current prices, risk rates etc.) to the device memory and to make one thread process the price of one option. Then launch the kernel function, which implements full algorithm, with sufficient amount of the threads.

A. Some optimization notes.

We improved the performance with another computation model: we used two kernel functions in this implementation (which are *computing_kernel* and *pricing_kernel* will be considered a few later). It is possible to implement the full algorithms using one kernel function only, but dividing the kernel into separate few kernels causes several possibilities for optimization and hence a performance increasing.

CUDA allows threads to cooperate with each other within the block and a buffer in the shared memory makes this cooperation possible. In addition, an option price is calculated with a threadblock, not only a thread, and calculated prices are stored in the shared memory. This approach provides more efficiency by using decreasing of the accesses to the global memory, than trivial parallel model. Then the average of the obtained prices over all threads within the block is calculated using classic parallel reduction algorithm. Obtained values transfer back to the global memory between kernel calls. Between kernel calls values transfer back to the global memory.

Required constants and data are stored in the constant and the local memory.

It is also important to select the block size properly. It is recommended to set the block size as an integer multiple of warp size.

Moreover, before the kernel starts, we have sorted the arrays of the input data using the CPU in the increasing order of the elements of the array *T*. The benefit of it is that sorted this way array of expiration dates allows to decrease the threads divergence. Because the kernel is not performed "truly parallel". I.e. when the kernel starts, threads are being grouped into warps and are performed simultaneously within the warp, if they execute the same instruction. So, if the threads within warp execute for example two different instructions, threads with different instruction wait for the threads with another instruction to proceed. The number of tests in MC method depends on expiration date. Further, it will be clear with the algorithms the further the expiration date the more MC tests are to be done. Therefore, sorting will allow threads to execute in "more parallel".

We moved the same for all threads constants to the constant memory, so program can cache them.

Also we assume what there is 252 trading days in a year.

Consider several MC methods of the calculation the Asian option price.

B. Pricing using path integral [10], [11], [12].

1) Calculate:

$$x_0 = \ln(S_0)\,,\ \ \mu = r - \frac{\sigma^2}{2}.$$

2) Generate random variables $x_i\,, i = \overline{1,n}$, with following densities distribution:

$$\frac{1}{\sqrt{2\pi\sigma^2\Delta t}}\exp\left\{-\frac{1}{2\sigma^2\Delta t}[x_i - (x_{i-1} + \mu\Delta t)]^2\right\}$$

3) Use full set $(x_0, \ \dots, x_n, x_{n+1})$ to calculate for $j = \overline{1,N}$

$$A_j = \frac{\Delta t}{T}\sum_{k=1}^{n+1}\int_0^1 V(\tau)\,\omega(\tau)\,e^{(x_k - x_{k-1})\tau + x_{k-1}}d\tau$$

where $V(\tau) = \left(1 + \sigma^2\Delta t(1-\tau)\frac{\tau}{2}\right)$

4) Calculate the option price

$$C(S,t) = e^{-rT}\frac{1}{N}\sum_{j=1}^{N} max\left(A_j - K, 0\right)$$

5) Repeat steps until the value of the option price obtained at the current step not differ from the value obtained in previous step to a sufficiently small variable

The serial code proceed every option consequentially and every step of the MC method is proceeded in the same way. We try to add parallelism with CUDA at the scale of proceeding the options. It means what all options are proceeded simultaneously. In addition, every option uses several threads at the same time, for example, to obtain value at step 3 of the MC method.

Now let us consider the pseudo-code of the CUDA implementation of the MC method with path integral.

Algorithm 3. Asian call options pricing, path integral

Input: arrays: *S* (current prices S_0), *r* (risk-free interest rates), *v* (volatilities), *T* (days to expiration), *K* (strike prices), *P* (prices of option of previous period of monitoring); scalars: *nMC* (amount tests in MC method), *N* (amount of options), *Ng* (amount of options could be processed simultaneously over the grid).
Output: arrays: *C*

```
1. Sort arrays S, r, v, T, K, P using quick sort by the elements of
   T as a sorting key
2. Transfer the data from the host memory S, r, v, T, K, P to the
   allocated regions in the global memory dev_S, dev_r, dev_v,
   dev_T, dev_K, dev_P. Allocate memory for dev_C,
   dev_prevC, dev_sum, dev_isDone, dev_isDone_reducted.
   Initialize states of the array of dev_states of
   curandStateSobol32 type
3. nMCPerThread := nMC / blockSize
4. Set sufficient value of eps.
5. Copy nMCPerThread and eps to c_nMCPerThread and
   c_eps variables resided in the constant memory
6. Set stride := N / Ng - 1 and ISDONE := false
7. while ISDONE = false do
8.     for jstride = 0 : stride do
9.         pricing_kernel <<<gridSize, blockSize>>> (jstride,
           dev_sum, dev_isDone, dev_states, dev_S, dev_r,
           dev_v, dev_K, dev_T, dev_P)
10.    end for
11.    Synchronize the threads within the grid
12.    computing_kernel <<<gridSize, blockSize>>> (dev_C,
       dev_sum, dev_isDone, dev_prevC, dev_r, dev_T)
13.    Synchronize the threads within the grid
14.    reduction_kernel <<<gridSize, blockSize>>>
       (dev_isDone, dev_isDone_reducted)
15.    Synchronize the threads within the grid
16.    Transfer data from d_isDone_reducted to
       isDone_reducted
17.    If all values in isDone_reducted are set as true then
       ISDONE := true
18. end while
19. Copy values from dev_C to C
```

We sort the arrays as it was said in the optimization notes and copy the variables to the fast cache in the constant memory.

Since the size of the shared memory is limited by 48kB, when choosing a block size of 128 threads, we see what tested GeForce GT630M makes it possible to simultaneously calculate the price of 192 options only. So, the pseudo-code above illustrated what with variables *Ng* and *stride*. We proceed 192 options simultaneously, then next 192 options and so on.

Also the steps 11 – 17 shows we also use parallel reduction for the array of the flags of type bool of every option, to ensure all options has been proceeded. For that purpose, function *reduction_kernel* is used. Function *computing_kernel* (step 12) implements only step 4 of MC algorithm trivially, therefore does not worth considering. But we will consider realization of *pricing_kernel*, as long as it is not straightforward as in the case of the *reduction_kernel* and *computing_kernel.*

Algorithm 4. Kernel function: *pricing_kernel*

```
1. Define the positions of the current thread tid and the block
   bid within the grid. Calculate the size of the block blockSize
   and the position of the thread within the block idx. Set the
   index of the option: optInd := Ng * stride + bid

2. if dev_isDone[optInd] = false do
3.     Allocate memory for array sbuffer_c [blockSize] in shared
       memory
4.     Copy input data into variables resided in local memory
       localState, rate, vol, price, days
5.     constSummand := (rate - vol * vol / 2) * deltaT
6.     dev := vol * sqrt(time/ (time + 1) / 252)
       summed_A := 0
7.     for iterationCounter := 0 : c_nMCPerThread do
8.         sbuffer_c[idx] := log(price)
9.         for daysCounter = 1 : time + 1 do
10.            Generate standard normal random variable r_val
11.            sbuffer_c[idx] := sbuffer_c[idx] +
               constSummand + dev * r_val
12.            Compute variable A as integral as described
               at the step 3 of the algorithm.
               summed_A := summed_A + A
13.        end for
14.        summed_A := summed_A * deltaT / T
15.        sbuffer_c[idx] := max(summed_A - strike,0)
16.        Synchronize the threads within the block
17.        Run the parallel reduction for sbuffer_c array
18.        if idx = 0 do
19.            dev_sum [optInd] := dev_sum [optInd] +
               (sbuffer_c[idx] / blockSize)
20.        end if
21.        Synchronize the threads within the block
22.    end for
23.    dev_states[tid] := localState
24. end if
```

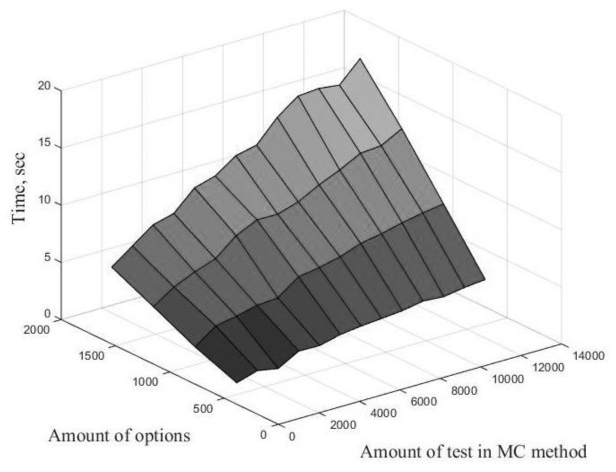

Figure 3. The dependence of the elapsed time of CUDA algoritm on amount of test in MC and amount of prices of options calculated sumiltanious. Path inegral.

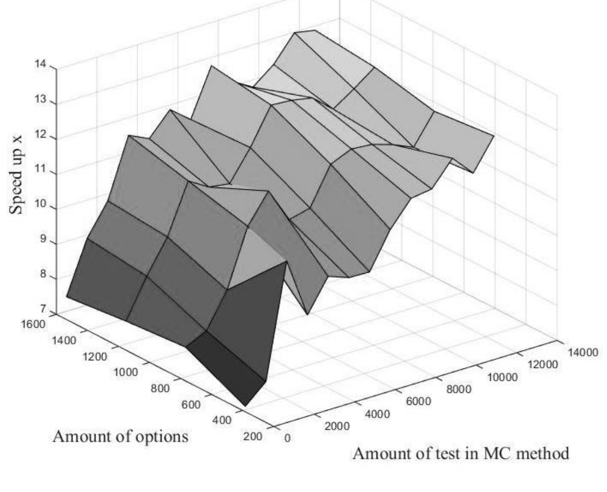

Figure 4. Acceleration of computations of CUDA parallel algorithm relatively serial algorithm. Stochastic differential equation

To generate the random variable with specify densities distributions we use the Box – Muller transformation (steps 10 – 11). To integrate the expression in the step 3 of the MC method interpolation Newton polynomial is used. It allows to construct the function of dynamics of the price of an underlying asset $\omega(\tau)$. Worth saying, what the bottleneck of this algorithm is the construction the function $\omega(\tau)$ and the integrating. We did not use nested parallelism in this paper. It could has positive effect on efficiency and could be considered in further research.

Then the parallel reduction is used to obtain the average of prices over the threads within the threadblock (steps 16 - 21).

Figure 3 and Figure 4 shows the elapsed time of the CUDA implementation and acceleration relatively the serial implementation. Acceleration is not constant for over set of variables. CUDA implementation executes 11 – 13 times faster than serial. Acceleration may increase with increasing parameters (the tests per thread and amount of options). Transfer and sorting times are taken into account.

C. Pricing using partial differential equation [13].

1) Construct the grid $[0, S_{max}] \times [0, A_{max}]$ *by dividing invervals into* N_s *and* N_a *subintervals.*

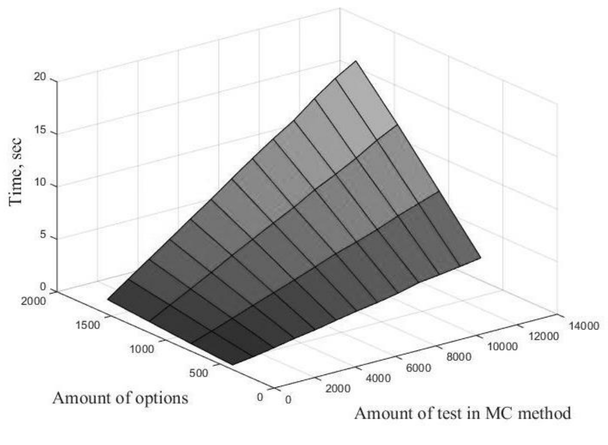

Figure 5. The dependence of the elapsed time of CUDA algoritm on amount of test in MC and amount of prices of options calculated sumiltanious. PDE.

2) Calculate:

$$Q_{i,j,n} = \frac{1}{\frac{\sigma^2 S_i^2}{(\Delta S)^2} + \frac{rS_i}{\Delta S} + \frac{S_i}{\Delta A}}$$

3) Select a point P on the grid. Calcualte p_1, p_2 u p_3 probabilities of transitions from the P to P_1, P_2 u P_3 points.

$$\Gamma_{i+1,j,n} = \frac{\sigma^2 S_i^2}{2(\Delta S)^2} + \frac{rS_i}{\Delta S} \quad p_1 = p_{i+1,j,n} = \Gamma_{i+1,j,n} Q_{i,j,n},$$

$$\Gamma_{i-1,j,n} = \frac{\sigma^2 S_i^2}{2(\Delta S)^2} \quad p_2 = p_{i-1,j,n} = \Gamma_{i-1,j,n} Q_{i,j,n},$$

$$\Gamma_{i,j+1,n} = \frac{S_i}{\Delta A} \quad p_3 = p_{i,j+1,n} = \Gamma_{i,j+1,n} Q_{i,j,n}.$$

4) Move to the selected point.

a) If during the transitions the sum of the obtained values $Q_{i,j,k}$ becomes greater than T, then stop the process. Calculate the option price:

$$C(S,A,0) = e^{-r\tau} \max\left(\frac{A}{T} - K, 0\right).$$

b) Upon reaching one of the borders consider the option price in the one of the following ways:

$$C(0,A,\tau) = \max\left(\frac{A}{T} - K, 0\right),$$

$$C(S,A_{max},\tau) = \max\left(\frac{A_{max}}{T} - K, 0\right) + \frac{S}{rT}(e^{-rT} - 1),$$

$$C(S_{max},A,\tau) = \max\left(\frac{A}{T} - K, 0\right) + \frac{S_{max}}{rT}(e^{-rT} - 1),$$

$$C(S,0,\tau) = \frac{S}{rT}(e^{-rT} - 1).$$

c) Otherwise continue the transitions to the next point.

5) When get the set of $C_1, C_2, \dots, C_N$, calculate the price of option

$$C = e^{-rT} \frac{1}{N} \sum_{k=1}^{N} C_k.$$

Pseudo-code of CUDA implementation will be considered next.

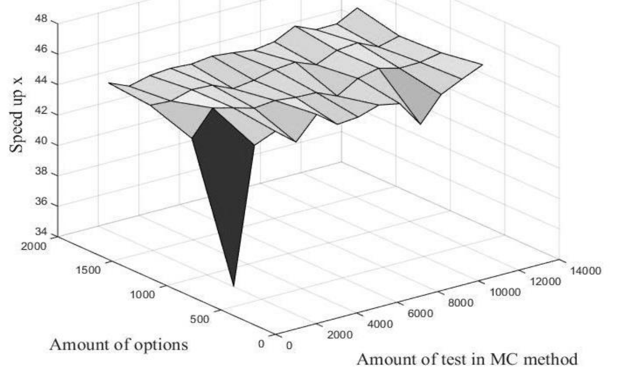

Figure 6. Acceleration of computations of CUDA parallel algorithm relatively serial algorithm. PDE.

Algorithm 5. Asian call options pricing, partial differential equation

Input: arrays: *S* (current prices S_0), *r* (risk-free interest rates), *v* (volatilities), *T* (days to expiration), *K* (strike prices); scalars: *nMC* (amount tests in MC method), *N* (amount of options), *Ng* (amount of options could be processed simultaneously over the grid)

Output: arrays: *C*

```
1. nMCPerThread := nMC / blockSize and copy into
   c_nMCPerThread resided in the constant memory
2. Sort arrays S, r, v, T, K using quick sort by the elements of
   T as a sorting key
3. Transfer the data from host memory S, r, v, T, K, A to
   allocated regions in global memory dev_S, dev_r, dev_v,
   dev_T, dev_K, dev_A
4. Allocate memory for dev_C. Initialize states of array
   dev_states of curandStateSobol32 type
5. for jstride = 0 : N / Ng do
6.     for counter = 0 : nMCPerThread - 1 do
7.         pricing_kernel <<<gridSize, blockSize>>> (jstride,
           dev_sum, dev_cond, dev_states, dev_S, dev_r,
           dev_v, dev_K, dev_T)
8.         Synchronize the threads within the grid
9.     end for
10. end for
11. Synchronize the threads within the grid
12. compute_kernel <<< gridSize, blockSize>>> (dev_C,
    dev_r, dev_T)
13. Copy the values from dev_C to C
```

We prepare to computations at steps 1 – 4. As in the previous case *Ng* and *jstride* are used to proceed just a part of a set of options. We use *compute_kernel* to implement step 5 of the MC method and *pricing_kernel* for steps 1 – 4 of the method, which is considered further.

Algorithm 6. Kernel function: *pricing_kernel*

```
1. Define the positions of the current thread tid and the block
   bid within the grid. Calculate the size of the block blockSize
   and the position of the thread within the block idx. Set the
   index of the option: optInd := Ng * stride + bid
2. Select Amax, Smax, Ns, Na
3. delta_S := Smax/Ns, delta_A := Amax / Na and evalTime := 0
4. Allocate memory for the arrays sbuffer_c [blockSize],
   sbuffer_S [blockSize], sbuffer_A [blockSize] in shared
   memory
5. Copy input data into variables resided in local memory
   localState, rate, vol, price, day, A, sbuffer_c [idx] := price
   and sbuffer_A [idx] := A
6. Select a point on the grid with coordinates
   (sbuffer_c [idx], sbuffer_A [idx]) or closest point.
7. while (done = false) do
8.     Chose a point to move to
9.     Depends on the selected pointed calculate
          sbuffer_S[idx] := sbuffer_S [idx] + delta_S
          sbuffer_S [idx] := sbuffer_S [idx] - delta_S
          or sbuffer_A [idx] := sbuffer_A [idx] + delta_A
10.    evalTime := evalTime + Q
11.    if evalTime >= days / 252 or sbuffer_S[idx] =< 0 or
          sbuffer_A[idx] =< 0 or sbuffer_S[idx] >= Smax or
          sbuffer_A[idx] >= Amax do
12.          Calculate price of option sbuffer_c[idx] as
             described above.
13.          done := true;
14.    end if
15. end while
16. Run the parallel reduction for sbuffer_c over the block
17. if idx = 0 do
18.     dev_C[optInd] := sbuffer_c[idx] / blockSize
19. end if
```

According to the description of the algorithm, the most requested values throughout the algorithm are coordinates of the point on the grid and the option price. Therefore, transfer of coordinate values to shared memory and subsequent access to the fast-accessible shared memory instead of the global memory is a good idea. (Steps 1 – 8). At the step 11 we mean what steps 2 and 3 of the MC method are implemented. Step 12 illustrates how the point moves over the grid. If one of the conditions at step 14 is satisfied, then obtain option price with expression of 4.a or boundary conditions of 4.b.

The bottleneck of this algorithm is the choice of the parameters *Amax*, *Smax*, *Ns*, *Na*. We tried *Amax* as current value of *A* multiple 252, *Smax* as *S* + 100, *Ns* and *Na* could vary.

Then the parallel reduction is used to obtain average price.

Due to Figure 6, the evaluated acceleration is up to 45 times in comparison with the serial algorithm and does not change with variety of parameters. Figure 5 also shows dependence of elapsed time of CUDA implementation of parameters. The transfer and sorting times are taken into account.

V. Conclusions

Using CUDA allows increasing performance of computation even on a weak graphics card. The most important conclusion of this paper is what field of the application of the proposed approaches is not limited by the scope of the options price computing. Proposed approaches of computation could be also successfully applied, for example, in finite difference schemes and methods of calculation of the path integrals.

Also worth saying what implementation could be transposed to newer devices with more advanced architecture. Main restriction is the size of shared memory. Implementation may be improved with newer graphic cards mainly due to bigger size of shared memory.

CUDA allows easy algorithms scaling by variety of kernel parameters (the size of configured grid and blocks). It is clear what the more multiprocessors and cuda-cores are available for computation the more efficient implementations are.

We recommend to use Sobol's QRNG in order to decrease standard deviation and not to and avoid loss performance.

For now, the question of the accuracy of the computation of each method is out of the scope. It is the direction for further research.

References

[1] F. Black, M. Scholes ,"The pricing of options and corporate liabilities" Journal Political Economy. 1973. Vol. 81. P. 637-659

[2] R.C. Merton, "Theory of rational option pricing" Bell Journal of Economics and Management Science. 1973. Vol. 4

[3] Richter C, Schops S, Clemens M. "GPU acceleration of finite difference schemes used in coupled electromagnetic/thermal field simulations". IEEE Trans Magn 2013; 49(5):1649–52

[4] Orchard G, Martin J, Vogelstein R, Etienne-Cummings R. "Fast neuromimetic object recognition using FPGA outperforms GPU implementations" IEEE Trans Neural Networks Learn Syst 2013; 24(8):1239–52.

[5] Mei S, He M, Shen Z. "Optimizing hopfield neural network for spectral mixture unmixing on GPU platform" IEEE Geosci Remote Sens Lett 2014; 11(4):818–22.

[6] John Cheng, Max Grossman, Ty McKercher. "Professional CUDA c programming" ISBN: 978-1-118-73932-7, p. 350

[7] Hongmei Chi, Peter Beerli, Deidre W. Evans, and Micheal Mascagni. "On the scrambled sobol's sequences" Lecture Notes in Computer Science 3516, 775-782, Springer 2005

[8] Stephen Joe, Frances Y. Kuo "Remark on algorithm 659: implementing Sobol's quasirandom sequence generator" ACM transactions on mathematical software, Vol. 29, N 1, 2003 , p. 49

[9] Sobol,I.M. "Distribution of points in a cube and approximate evaluation of integrals". U.S.S.R Comput. Maths. Math. Phys. 7: 86–112 (in English), Zh. Vych. Mat. Mat. Fiz. 7: 784–802 (in Russian)

[10] Vadim Linetsky , "The path integral approach to financial modeling and options pricing" Computational Economics 11: 129–163, 1998

[11] Guido Montagna, Oreste Nicrosini, "A path integral way to option pricing" Physica A: Statistical Mechanics and its Applications Volume 310, Issues 3–4, 15 July 2002, Pages 450–466

[12] Malvin H. Kalos, Paula A. Whitlock "Monte Carlo methods" WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim ISBN: 978-3-527-40760-6

[13] Daniel Zwillinger "Handbook of differential equations", Academic Press, 1997