Option Pricing Simulation on GPGPU

Alexander, Bogdanov
Plekhanov Russian University of Economics Stremyanny lane, 36, Moscow, 117997, Russia,
St.-Petersburg State University, Universitetskaya emb. 7/9, St.-Petersburg, Russia
e-mail: bogdanov@csa.ru

Vladimir, Rukovchuk
Plekhanov Russian University of Economics Stremyanny lane, 36, Moscow, 117997, Russia
e-mail: vrukovchuk@gmail.com

Dmitry, Khmel
Plekhanov Russian University of Economics Stremyanny lane, 36, Moscow, 117997, Russia
e-mail: dima91x@gmail.com

Stanislav, Bogdanov
Plekhanov Russian University of Economics Stremyanny lane, 36, Moscow, 117997, Russia
e-mail: stan.bogdanov@gmail.com

ABSTRACT
The speed of computation is the key issue in trading. The appearance of GPGPU opens new possibilities for substantial increase in performance of pertinent algorithms. Although porting of heavy applications on GPGPU is a difficult task, we managed to effectively implement GPGPU both for solution of Black - Scholes equation and direct simulation of the process.

Keywords
Financial mathematics, GPGPU/ porting of applications, Black - Scholes equation, direct simulation.

1. INTRODUCTION
In the world of finance, one of the main factors affecting trading is time. Delaying information for a split second can cost millions of dollars. Thus, it is obvious that with such a time value, the speed of building a solution to the Black - Scholes equation also plays a big role. Particular attention should be paid to finding methods that are the least time consuming. Moreover, it is obvious that the choice of the method may depend on the values of the parameters of the problem.

At the moment there are two main approaches to modeling the behavior of options using the Black – Scholes equation: the finite difference method and the approach using the path integral.

2. APPROACHES TO MODELING THE BLACK-SCHOLES EQUATION
Finite difference method. This method will be considered on the example of an Asian option. The pricing problem of the Asian option is widely discussed [1], [2], [3]. One of the problems of the finite difference method for an Asian option is the formulation of correct boundary conditions. Hagger [4], [5], [6] cites detailed discussions on the derivation of the physical boundary conditions for the Asian option in his papers.

Functional integration. The method of functional integration is one of the main ones in modern quantum physics. The functional integral itself was first described by Richard Feynman in 1942 [7]. To date, this approach is very popular in financial mathematics. Numerical procedures were developed to search for the price of exotic options, depending on the price dynamics of the underlying asset [8], [9].

The Monte Carlo method [10], [11], [12], [13] is used to solve the partial differential equation and calculate the functional integral. However, despite the simplicity of implementation, it has several disadvantages. The first is the poor convergence of the method and, accordingly, the need to generate very long chains of random numbers. As a result, in practical problems a very careful approach is required when choosing a random number sensor. Another significant disadvantage of this method is the impossibility of an a priori estimate of accuracy.

3. APPLICATION OPTIMIZATION ON HETEROGENEOUS SYSTEMS
Due to the constant increase in the diversity of accelerators, development tools must, to one degree or another, ensure the portability of applications between devices. This can manifest itself in two forms: either an already compiled program is launched on different platforms without recompiling, or the program is compiled from one source code for different platforms. The second method also includes compiling platform-dependent fragments into dynamic libraries for different platforms with the subsequent selection of the appropriate fragment for a particular platform. Porting without recompilation requires a unified intermediate language, and porting with recompilation requires a unified high-level API.

The most portable API is still OpenCL. Although the degree of support varies for different platforms, and many platforms still do not support OpenCL 2.0, it is difficult to find a platform oriented at accelerating parallel data processing but not supporting OpenCL at all. In the case of mobile platforms (especially on Android), there are essentially no portable OpenCL alternatives. Despite the good portability, OpenCL has many disadvantages. For example, because of the compilation at runtime, the device driver must contain an almost complete C compiler, and the source code of the kernels is open. The kernel works in its own context, even if it runs on a host, which requires unnecessary copying of data from the host into this context. Computing kernels have to be written in a special language that is a subset of C. It creates

potential vulnerabilities in data transfer between the host and the device that are not detected by static analyzers. It also means that compiler cannot perform context-sensitive optimization. Apple even declared OpenCL deprecated in IoS 12 [3]. Many developers prefer to use NVIDIA CUDA. CUDA is free of many of the drawbacks of OpenCL, in particular, it supports the use of C ++ (including C ++ 17) in the kernels and does not separate the kernel code from the host code, which allows data transfer to be type safe. The question of comparing the performance of CUDA and OpenCL kernels remains open in general, but cuBLAS (linear algebra library as part of 4 CUDA) outperforms both user-made implementations on OpenCL and CUDA, which is also an argument in favor of CUDA [4]. But the choice of CUDA over time can lead to complications: the computing accelerators ecosystem is evolving, new hardware solutions are emerging, and switching to other platforms may become a necessity. For large projects, this can be a really serious problem. An interesting solution is offered by the ROCm project: it contains a HIP – tool that allows you to semi-automatically convert CUDA code into portable C++ code and compile this portable C++ for AMD and NVIDIA, or you can write initially portable applications. At the moment this tool is only applicable for NVIDIA and AMD, although the idea can be implemented for other platforms [5]. For projects designed for long-term support, standardized APIs originally designed as portable are more interesting. Such API can be provided by the Parallelism project in the C++ standardization framework. Its idea is that the interfaces provided by the standard C++ library for algorithms are already well suited for parallel algorithms, an example of which is Boost.Compute and other similar libraries. So, this interface can be extended for automatic parallelization [6]. The CPU-oriented part of this project is already included in the C++17 standard, and there is a potential for expanding the approach to the GPUs and other devices [7]. The main problem here is that the level of abstraction is too high to allow you to manually optimize the code for specific devices. Even the choice of the device is actually inaccessible to the user, the compiler chooses it. But thanks to this project, the C ++ memory model is now expanding towards greater support for parallel algorithms, including for accelerators. A beter compromise option is the SYCL standard - a heterogeneous-oriented API for C ++, released in 2014.

3.1 Ways to manage the process of computing

The following approaches to managing the calculation process have been implemented:

Simple Loop. A simple loop on a set of matrices, at each iteration, a pair of source matrices is transmitted to the device, processed on it, and the result is transmitted to the host. All tasks are performed sequentially. Obviously, for small matrices overhead must be greater than for large ones.

One-In One-Out. All source matrices are written to one large buffer, the result is also written to one buffer, the computational kernels are started asynchronously and can be executed in parallel. Overhead costs should be less dependent on the size of the matrices, but in practice the use of this approach is rarely possible.

One-In Many-Out. The original matrices are also recorded in one buffer, which is transferred to the accelerator once, but the computational kernel is started and the data is transferred to the host sequentially for each pair of initial matrices (the kernels are executed sequentially). This allows the program to perform calculations and transmit to the host the results of previous calculations in parallel.

Auto Transfer Simple Loop. Similar to Simple Loop, but instead of explicit data transfer, SYCL data automatic control was used. In this case, only the total operation time is measured, since SYCL does not measure data transfer times separately with automatic control.

3.2 Ways to optimize computational kernels

Computational kernels are implemented for the operation of matrix multiplication and matrix multiplication by a vector.

Simple Kernel. The classic loop used to multiply matrices by the CPU. The only optimization is to change the traversal order so as to streamline the memory access.

Local Memory Kernel. Classic block algorithm taken from CUDA documentation. At each step, two fragments of the original matrices are placed in the local memory of the working group (shared memory in terms of CUDA), the intermediate result is 8 calculated and also placed in the local memory. In the case of multiplication by a vector, the local memory is not needed for the left matrix. The block size was calculated automatically based on accelerator characteristics.

Private Memory Kernel. It is similar to Local Memory Kernel, but the intermediate result is placed directly into private (register) memory.

4. FEATURES OF THE MODELING OF THE BLACK-SCHOLES EQUATION ON GPGPU

A full description of the model itself is given in our other articles [14]. Here I would like to highlight the features of the application of these approaches on hybrid systems.

The main problem of calculating an option through a functional integral by the Monte Carlo method is that in order to calculate the integral it is necessary to generate a set of random numbers in such a way that the distribution density of each of them depends on the previous one.
The distribution function of a random variable:

$$f(x) = exp\left\{-\frac{1}{2\sigma^2 \triangle t}\sum_{k=1}^{n+1}[x - (x_{k-1} + \mu \triangle t)]^2\right\} \quad (1)$$

To eliminate the dependence on x_{k-1} we note that

$$x_n = \xi_n \sigma\sqrt{\triangle t} + \left(r - \frac{\sigma^2}{2}\right)\triangle t + x_{n-1}, \xi_n \in N(0,1) \quad (2)$$

This formula can be represented as:

$$x_n = \sigma\sqrt{\triangle t}\sum_{i=1}^{n}\xi_i + n\left(r - \frac{\sigma^2}{2}\right)\triangle t + x_0 \quad (3)$$

Thus, if we use formula (3) to calculate an option on a GPU, we can solve the problem with dependencies, and as a result we get a typical problem for a GPU based on the implementation of a large number of stochastic processes. Having a large array with the resulting data for each process, the problem arises of finding their sum. For this purpose, you can use parallel reduction algorithms, which can also be executed on the GPU several times faster than on the CPU. In this case, knowing the hierarchy of memory and the architecture of the graphics processor, you can speed up the reduction algorithm several more times. So, as the main

limiting factor in performance is memory access. Efficiency can be increased by using shared memory between threads within a single block [15].

For all applications, an important factor for achieving high performance on the GPU is the ability to bypass the "bottleneck", transferring large amounts of data to the memory of the graphics device. Fortunately, for our task, only random numbers are used as input, which can be generated directly on the device using the CURAND library. It contains optimized functions for generating pseudo- and quasi-random numbers with a period of 267 [16].

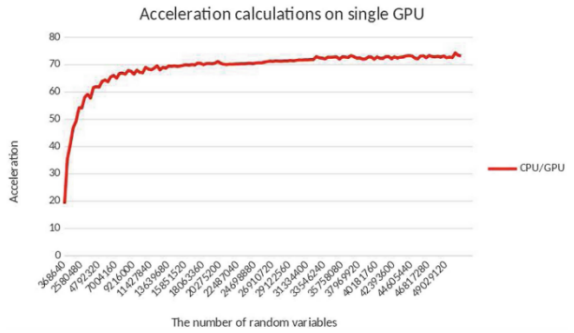

Fig. 1. Comparison graph of sequential and parallel algorithms for the calculation path integral using Monte Carlo method.

The second option pricing approach using the explicit finite difference method has also been implemented on the GPU. For more details, see [17]. In order to perform calculations with the maximum speed, constant time steps and spatial variables were chosen. Experiments have shown that the explicit method allows us to achieve sufficient accuracy.

$$C_{jk}^{n+1} = (1 - \triangle\tau\sigma^2 j^2 - r\triangle\tau)C_{jk}^{n} + \left(\frac{\triangle\tau\sigma^2 j^2}{2} + \frac{rj\triangle\tau}{2}\right)C_{j+1,k}^{n} + \left(\frac{\triangle\tau\sigma^2 j^2}{2} - \frac{rj\triangle\tau}{2}\right)C_{j-1,k}^{n} + \frac{j\Delta S\Delta\tau}{2\Delta I}C_{j,k+1}^{n} - \frac{j\Delta S\Delta\tau}{2\Delta I}C_{j,k-1}^{n} \quad (4)$$

To implement this approach on the GPU, we will pay attention to the following features of the problem under consideration (4):

- the coefficients of the finite-difference scheme are calculated independently of one another, and they do not depend on time;
- the values C_{jk}^{n+1} are independent within each time layer.

Since the coefficients of the finite-difference scheme do not depend on time, it would be advisable to keep them in the permanent memory of the graphics processor. In this case, thanks to the caching mechanism, there is no need to constantly read them from the global memory. Only in this case it is necessary to determine the number of sampling points that can fit in the internal memory of the device without overflowing it.

This is the second feature of the graphics processor. The values were calculated by blocks of 16 × 16. These changes drastically reduce the number of accesses to global GPU memory.

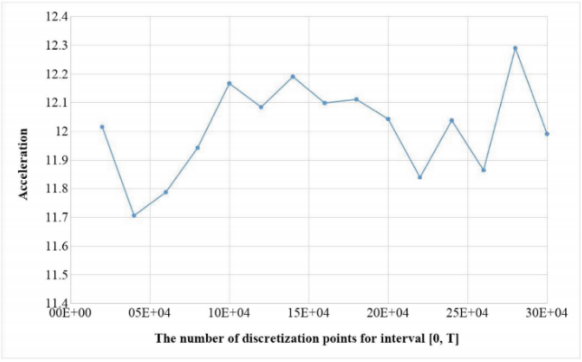

Fig. 2. Comparison graph of sequential and parallel algorithms for finite difference method.

REFERENCES

[1] B. Alziary, J.-P. Decamps, P.-F. Koehl, "A P.D.E. Approach to Asian Options: Analytical and Numerical Evidence"

[2] E. Barucci, S. Polidoro, V. Vespri, "Some results on partial differential equations and asian options"

[3] W. Henao, J. G. Lee, M. Moon, A. Narboni, N. Petrosyan, "Numerical methods for pricing of asian options"

[4] J. Hugger, "A fixed strike Asian option and comments on its numerical solution"

[5] J. Hugger, "The boundary value formulation of the Asian Call Option"

[6] J. Hugger, "Wellposedness of the boundary value formulation of a fixed strike Asian option"

[7] R. P. Feynman, "The Principle of Least Action in Quantum Mechanics, Ph.D" thesis, Princeton, May 1942.

[8] C. John, "Hull Options, Futures and other derivatives. Seventh edition"

[9] P. Wilmott, J. Dewynne and S. Howinson, "Option Pricing: Mathematical Models and Computation" Oxford Financial Press, Oxford, 1993.

[10] Н. П. Бусленко, Д. И. Голенко, И. М. Соболь, В. Г. Срагович, Ю. А. Шрейдер "Метод статистических испытаний (метод Монте-Карло)" Физматгиз. 1962.

[11] В. Ф. Кузнецов, "Решение задач теплопроводности методом Монте-Карло". М.:ОНТИ ИАЭ, 1973

[12] V. Linetsky, "The path integral approach to financial modeling and options pricing"

[13] G. Montagna, O. Nicrosini, "Path Integral Way to Option Pricing"

[14] A. Bogdanov, E. Stepanov, D. Khmel, "Assessment of the dynamics of Asian and European options on the hybrid system" J. Phys: Conf. Ser. 681(1), 012007 (2016)

[15] A. Boreskov, A. Harlamov, "Parallelnye vychysleniya na GPU" Moscow University Press, Moscow (2012)

[16] CUDA Toolkit documentation (n. d.). http://docs.nvidia.com/cuda/curand. Accessed 27 May 2019

[17] E. Stepanov, D. Khmel, V. Mareev, N. Storublevtcev, and A. Bogdanov, "Porting the algorithm for calculating an asian option to a new processing architecture" Vol. 10963 LNCS 2018, p. 113-122.