|  |  |
| --- | --- |
|  |  |

The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program   
of the Lobachevsky State University of Nizhni Novgorod   
among the world's research and education centers

Strategic initiative  
“Achieving leading positions in the field of supercomputer technology   
and high-performance computing”

**PROGRAMMING AND OPTIMIZATION   
FOR INTEL XEON PHI**

*Practice 8. Optimization of European option pricing*

Nizhni Novgorod

2014

# Objectives

The objective of this practice is to study principles of optimization of computational applications on the example of European option pricing. This objective includes the following activities:

1. Introduction to a financial market model and basic concepts of financial mathematics.
2. Implementation of basic version of Black-Scholes European option pricing.
3. Step-by-step optimization and parallelization on CPU.
4. Step-by-step optimization and parallelization on Xeon Phi.
5. Analysis of results of optimization and parallelization.

# Abstract

This practice illustrates approaches to optimization of computational applications. We consider European option pricing, which is a fairly simple problem of financial mathematics, in case the result can be computed analytically. From a programming point of view this is a trivial problem (just apply a formula for input data), but it demonstrates that even in such programs computational time can vary by an order of magnitude depending on programming and optimization skills and techniques

First we briefly introduce a model and basic concepts of a financial market and some intuitive descriptions of the option pricing problem. We create a basic implementation, analyze its performance and improve it in a step-by-step fashion: eliminate unnecessary type casts, carry out invariants, perform mathematical transforms that replace heavy math routines with lighter ones, vectorize and parallelize, perform warm-up to reduce overhead on thread creation, try reducing precision of floating-point operations, utilize streaming stores. The effects of these optimization techniques are demonstrated on both CPU and Xeon Phi.

# BRIEF OVERVIEW

We start with problem statement and some basic concepts of financial mathematics. We consider a financial market in continuous time that has two actives: shares (risk actives, *S*) and bonds (risk-free actives, *B*). The market is simulated via a widely used Black-Scholes model which can be written as a system of stochastic differential equations. We explain parameters of the model and their meaning.

Option is a derivative, a contract between sides *P*1 and *P*2 that grants *P*2 a right to buy from or sell to *P*1 shares with fixed price *K* at a future moment *T*. This right costs a price *C* that *P*2 pays to *P*1. Then *K* is called strike price, *C* is called option price, and *T* is called maturity. We consider the simplest option, which is European call-option. The contract is a game of two players *P*1 and *P*2. The latter pays a price *C* and at a future moment *T* that is defined by a contract makes a decision whether to buy shares by price *K* from *P*1 or not. The decision is made based on relation between stock price *ST* and *K*. If *ST* < *K* there is no point in buying by price *K,* the first player gains profit *C*, and the second player suffers loss *C*. In case *ST* > *K* the second player buys shares by price *K* and might gain some profit (depending on relation between *C* and *ST – K*).

The problem is to compute a fair price on such option which is a price that balances gains and losses of both sides. It is logical to find this price as an average gain of *P*2. With this assumption, analytical solution is given by Black-Scholes formula for computing a fair option price at moment *t* = 0:

|  |
| --- |
|  |

where *F* is standard normal distribution function.

We solve a problem of pricing a large portion of options with different parameters in the same market. The scheme of information dependencies for this problem is given.

The main part of this practice is implementation and optimization for CPUs and Xeon Phi. We start with a basic implementation that computes price of one option using Black-Scholes formula. This version will be used as a reference for future optimized versions. We discuss data organization and compare Array of Structures and Structure of Arrays with the latter being preferable for our application (as well as most others). A version for a portion of options is implemented. We argue that single precision floating-point computations are sufficient given the specifics of financial computations. We illustrate a typical mistake of mixing single and double precision constants and functions which usually leads to very poor performance. An elimination of unnecessary type casts using programming language tools leads to performance increase. Mathematically equivalent transformation is used to compute a normal distribution function via erff() routine. This yields significant performance benefit due to lesser complexity and better implementation of erff(). We perform vectorization using compiler directives and **restrict** keyword that was discussed in the previous lectures and practices. On CPU vectorization results in ~ 5.5 times speedup, with ideal speedup in AVX being 8. We apply standard optimization techniques of carrying out invariants and other mathematical transforms. In this example this does not yield further performance benefit because compiler performed similar modifications automatically, but it can be useful for other applications or compilers. Up to this moment our current version is as follows:

void GetOptionPricesV6(float \*pT, float \*pK, float \*pS0, float \*pC)

{

int i;

float d1, d2, erf1, erf2, invf;

float sig2 = sig \* sig;

#pragma simd

for (i = 0; i < N; i++)

{

invf = invsqrtf(sig2 \* pT[i]);

d1 = (logf(pS0[i] / pK[i]) + (r + sig2 \* 0.5f) \*

pT[i]) \* invf;

d2 = (logf(pS0[i] / pK[i]) + (r - sig2 \* 0.5f) \*

pT[i]) \* invf;

erf1 = 0.5f + 0.5f \* erff(d1 \* invsqrt2);

erf2 = 0.5f + 0.5f \* erff(d2 \* invsqrt2);

pC[i] = pS0[i] \* erf1 - pK[i] \* expf((-1.0f) \* r \*

pT[i]) \* erf2;

}

}

We discuss alignment of arrays. In our case the compiler performs it automatically. Finally, we consider a justified reduction of precision of mathematical routines. For our application even single precision is redundant, so we use compiler options for faster computation of mathematical routines. This is achieved by computing lesser amount of correct digits and ignoring some exceptional cases that can not occur in Black-Scholes computation. Performance benefit is ~23%.

We prepare for porting to Xeon Phi. The loop in function GetOptionPricesV6() is easily parallelized due to lack of data dependencies by using #pragma omp parallel for and localization of all variables. Speedup on 16 CPU cores is ~11.3 compared to 1 core.

We discuss warm-up technique frequrently used in benchmarks. Computational time for the optimized version is so small that overhead on creating threads takes a significant share. It can be hidden using the fact that in most implementations of OpenMP threads are not destroyed after a parallel section is over, but are set asleep, which saves time in the next parallel section. Thus, for the second call of GetOptionPrices() will have reduced overhead. This approach is called warm-up. We argue whether warm-up gives fair performance results. As of thread creation, it is fair, because for large applications threads can be created once and all parallel computations will not have this overhead. It is important to notice another effect of warm-up, so-called cache warm-up. In case a function is called several times, some data might reside in cache resulting in unfairly large speedup. We formulate recommendations on how to fairly use warm-up and interpret results. In our case speedup on 16 CPU cores with warm-up is ~13.6 compared to 1 core.

We port the code to Xeon Phi and analyze performance benefits of the previously applied optimizations. Reducing precision gave more than 2 times speedup. Warm-up is also important for Xeon Phi because of larger amount of threads compared to CPU.

Finally, we discuss streaming stores that is writing directly to memory avoiding cache, an important technique for Xeon Phi. In our case the loop has a huge amount of iterations with small ratio between computations and memory access. Given hundreds of threads working on Xeon Phi, there is a high load on memory bus, sometimes even exceeding its peak throughput. Streaming stores for array pC allow to save one read instruction on each iteration, that results in performance benefit.

Our best implementation for Xeon Phi outperforms our best implementation for CPU by a factor of ~4.6 times.

# FOR STUDENTS

Various materials on optimization for Xeon Phi and real-world examples are presented at [1]. Key techniques of performance improvement on Xeon Phi are discussed in [2]. Some examples of optimization for Xeon Phi are given in [3].

# References

1. MIC developer zone. URL: [http://software.intel.com/mic-developer]
2. Common Best Known Methods for Parallel Performance, from Intel® Xeon® to Intel® Xeon Phi™ Processors. URL: [https://software.intel.com/ru-ru/articles/common-best-known-methods-for-parallel-performance-from-intel-xeon-to-intel-xeon-phi]
3. Jeffers J., Reinders J. Intel Xeon Phi Coprocessor High Performance Programming. – Morgan Kaufmann, 2013.

# Individual work

1. Compare AoS and SoA data layouts in Black-Scholes computation.
2. Modify our implementation to computing Call and Put option prices. Experiment with different versions of the code and evaluate influence of our optimizations on performance on CPU and Xeon Phi in this case. Prove that memory bus on Xeon Phi is overloaded if many threads are used.
3. Try different amount of threads on Xeon Phi. Find the optimal configuration.
4. Explain speedup from vectorization on CPU and Xeon Phi.