CHAPTER

19 PERFORMANCE OPTIMIZATION OF BLACK-SCHOLES PRICING

Iosif Meyerov*, Alexander Sysoyev*, Nikita Astafiev†, Ilya Burylov†
**Lobachevsky State University of Nizhni Novgorod, Russia, †Intel, Russia*

High-performance computing—although a dynamic and expanding area of computer science, as an industry, tends toward being conservative when it comes to investing in new optimizations—even if they promise a disruptive speedup potential. The emergence of the Intel® Xeon Phi™ coprocessor family, combining many processing cores with a traditional x86 processing architecture, offers the speeds of massively parallel architecture but requires effort in application transformation to extract all of the provided parallelism. The need to transform the application poses questions such as: What data structures and algorithms apply? How to best scale applications? What optimization techniques bring benefits? What applications fit well with new highly parallel architectures? Is it possible to optimize programs simultaneously for host Intel® Xeon® processors and for Intel Xeon Phi coprocessors?

This chapter presents our experiences optimizing the calculation of fair prices for a set of European options with Intel Xeon processors and Intel Xeon Phi coprocessors.

We have chosen this topic for the following reasons: First, European option pricing is traditionally used as a benchmark for verifying the capabilities of new architectures. Second, it is one of the basic elements of financial market analysis, and requires HPC systems computational power and, therefore, has great practical interest. Third, the implementation methodology is quite simple to understand. The option pricing algorithm is based on the prevalent Black-Scholes formula which is well described in financial mathematics textbooks and does not require any special implementation knowledge. Finally, this algorithm takes only a few lines of code, which while implying simplicity, still leaves room for experimentation!

Looking at the description of the Black-Scholes algorithm, it may be difficult to imagine that pitfalls and mysteries still are contained in its implementation and that it provides great opportunities to demonstrate optimization techniques.

In this case study, we will walk step by step through various methods of application optimization, which can, in our opinion, be useful for the development of other software for both host processors and coprocessors. We report both our progress and temporary failures, including promising optimizations that did not have a positive effect. We include these optimization steps in our discussion to stress that implementation of similar methods in other applications can lead to successful improvements. Who would have thought that a mere two hundred lines of code would provide so many learning opportunities!

The chapter is organized as follows: First, a short description of a financial market model is given, the basic concepts are discussed to ensure an understanding of the key elements of the algorithm. The baseline implementation is then described, performance analysis is done, then step-by-step optimizations are applied including the elimination of unnecessary type conversion, loop-invariant code hoisting, equivalent conversions replacing "heavy" mathematical functions by "lighter" ones, vectorization of calculations, parallelization, "warming up" of thread creation to avoid skewing the results with one-time overhead, reduction of the accuracy of floating-point calculations, memory optimization (via the use of streaming stores), and finally, the optimization effects are shown for both the processor and coprocessor. In other words, we start our optimizations with the most generic ones that help any parallel program and illustrate their effect showing the processor run times. Only toward the end of the chapter the difference between coprocessor and processor comes out. This way we illustrate a fruitful optimization methodology: first, extract general parallelism and then focus on architecture-specific fine-tuning.

FINANCIAL MARKET MODEL BASICS AND THE BLACK-SCHOLES FORMULA

FINANCIAL MARKET MATHEMATICAL MODEL

Let us consider a financial market evolving in continuous time and consisting of two types of assets—shares (risk-based assets, S) and bonds (risk-free assets, B). For market modeling, we take the widely used Black-Scholes model. This model, after a number of transformations, is represented by a system of stochastic differential equations that looks as follows:

$$\mathrm{d}B_t = rB_t\,\mathrm{d}t, \quad B_0 > 0 \tag{19.1}$$

$$\mathrm{d}S_t = S_t\left((r-\delta)\mathrm{d}t + \sigma\,\mathrm{d}W_t\right), \quad S_0 > 0 \tag{19.2}$$

Equation (19.1)—an ordinary differential equation, describes the behavior of the price of a bond B_t, which is influenced by r—*interest rate*. Equation (19.2)—a stochastic differential equation describing the evolution of the price of a share S_t. Along with the interest rate r, the equation contains a *dividend rate* δ, *volatility* σ, and *Wiener stochastic process* (WSP) $W=(W_t)_{\{t\geq 0\}}$. The initial share and bond prices (S_0 and B_0, respectively) are predefined constants.

Let us briefly explain the economic logic of the considered model. The first equation shows capabilities for investment in *risk-free assets*—bonds. Imagine that we come to a bank and deposit money at an interest rate determined by the bank depending on inflation, its unique state of affairs and the market as a whole. The second equation shows the influence of two groups of factors on the price of a share (*a risk asset*)—determinate and random. The first part—$S_t(r-\delta)\mathrm{d}t$—is similar to the right part in Eq. (19.1) with the sole difference that on shares, unlike bonds, dividends can be paid, which is reflected in the dividend rate δ.

The term $S_t\sigma\,\mathrm{d}W_t$ is of the greatest interest here. It is included into the equation additively and is used to model the influence of random, hard-to-predict factors in the market.

Volatility σ represents the randomness of processes in the market: $\sigma=0$ means everything is determined, that is, there is no risk. The bigger σ is, the higher the risk (both possible profit and possible losses). Multiplier $\mathrm{d}W_t$—the WSP differential—describes random factors influencing changes in the price of a share at a given point in time t.

The WSP is a mathematical model of Brownian motion in continuous time with these definitions:

1. $W_0 = 0$ with probability 1.
2. W_t—process with independent increments.
3. $W_t - W_s \sim N(0, t-s)$, where $s < t$, and $N(0, t-s)$ is Gaussian distribution with zero mean and variance $t-s$.
4. Process trajectories $W_t(\omega)$—continuous functions of time with probability 1.

For the resulting implementation, here are some key assumptions:

- In the formulas and calculations below as well as for the purpose of simplification, we assume $\delta = 0$ (model without dividends).
- If time is measured in years, then all the interest rates in the equations are numbers from 0.0 to 1.0—annual interest, expressed in fractions.
- We assume that interest rate and volatility are constant—they do not depend on time.

Given the assumption of constant interest rates and volatility, the system of the differential equations (19.1) and (19.2) has an analytical solution. Otherwise, the system has to be solved numerically using one of the known methods (Euler, Runge-Kutta, etc.). When constructing a difference scheme, the only difference from an ODE is that each WSP (Wiener stochastic process) increment is modeled by a random number obtained from $N(0, t-s)$. The solution to Eq. (19.2) looks as follows:

$$S_t = S_0 \mathrm{e}^{\left(r - \frac{\sigma^2}{2}\right)t + \sigma W_t} \tag{19.3}$$

EUROPEAN OPTION AND FAIR PRICE CONCEPTS

An *option* is a derivative financial instrument or a contract between parties P_1 and P_2, which gives the right to P_2 at some point of time t in the future to buy from or to sell to P_1 a share at price K fixed in the contract. In return for this right, P_2 shall pay the fixed amount (fee) C to P_1. K is called the *strike price*, and C—*the option price*.

In this study, the simplest variant of an option—the *European share call option* is considered. The main idea of the contract is a game of two parties, P_1 and P_2. The second party pays an amount C and at some point of time T (*maturity*, fixed in the contract) makes a decision: to buy shares at price K from the first party or not. The decision is made depending on the ratio of price S_T and K. If $S_T < K$, it is not profitable to buy the shares, the first received profit C, and the second incurred losses C. If $S_T > K$, the second party buys from the first party shares at K, in some cases getting a profit (depending on the ratio between C and $S_T - K$).

The main problem is the calculation of a *fair price* of such an option contract, it occurs when there is a balance of gains and losses for each of the parties. It is logical to define such a price as an average gain of party P_2:

$$C = E\left(\mathrm{e}^{-rT} \cdot \max\left(S_T - K, 0\right)\right) \tag{19.4}$$

In Formula (19.4), the fair price of the call option C is obtained as the mathematical expectation E of the *cash flow* function $\max(S_T - K, 0)$ multiplied by a *discounting* factor e^{-rT} which accounts for inflation with interest rate r over the time period $(0, T)$.

BLACK-SCHOLES FORMULA

Under the assumptions made earlier, Eq. (19.4) has an analytical solution known as the Black-Scholes formula for European call option price:

$$C = S_0 F(d_1) - K e^{-rT} F(d_2)$$
$$d_1 = \ln\frac{S_0}{K} + \frac{\left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \tag{19.5}$$
$$d_2 = \ln\frac{S_0}{K} + \frac{\left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

where F is the cumulative normal distribution function.

We will use this formula in further calculations. "What is there to calculate?" you might ask. At first glance everything looks elementary. In practice, everything is not that simple, which will be shown.

OPTIONS PRICING

Certainly, for pricing a single option, one does not need a high-performance computer. In practice, organizations operating in financial markets calculate prices for huge quantities of different options, which may be issued in specific market conditions. Timing of financial calculations significantly influences decision-making speed, so every second counts. Therefore, reducing valuation time for a set of options is a very important factor.

Let us construct a diagram of data dependencies for one option (Figure 19.1).

Generally, when constructing a set of options, it is possible to vary all five parameters (initial price of a share, strike price, interest rate, volatility, maturity). Nevertheless, be cognizant of the fact that

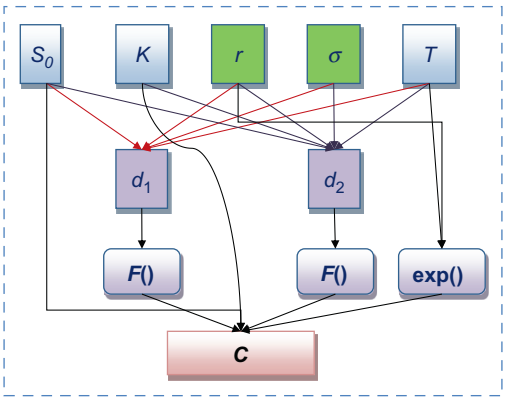

FIGURE 19.1

Data dependence diagram.

CPU	2 x Intel Xeon E5-2690 processors (8 cores, 2.9 GHz)
Coprocessor	Intel Xeon Phi 7110X coprocessor (61 cores)
RAM	64 GB
Operating system	Linux CentOS 6.2
Software	Intel Parallel Studio XE 2013 SP1

FIGURE 19.2

Test infrastructure.

market parameters (interest rate and volatility) at a specific time are identical for all options. Therefore, in the calculations below, we will assume that different options differ only by the initial price of a share, strike price, and maturity.

TEST INFRASTRUCTURE

The computational experiments were performed on the following test infrastructure (Figure 19.2).

CASE STUDY

PRELIMINARY VERSION—CHECKING CORRECTNESS

We set the following parameters (Figure 19.3).

We start our software implementation with the function **`GetOptionPrice()`** that prices one option using the Black-Scholes formula (19.5). It is noteworthy, that here, and in all other functions, we use single precision. Double precision is not required for the nature of the problem (for example, input data usually comes as single precision, type "float"). Later we will see that precision can be decreased further (Figure 19.23).

The nontrivial part of Formula (19.5) is a cumulative normal distribution function F. For its evaluation, the **`cdfnormf()`** function is used.

Let us build the code (Figure 19.4) and see if it works. The result of function execution must equal 20.924.

REFERENCE VERSION—CHOOSE APPROPRIATE DATA STRUCTURES

After ensuring the application is functionally correct, we now proceed to our main goal—estimating a set of options. Let us introduce the following variables (Figure 19.5).

```
const float sig = 0.2f;   // Volatility (0.2 -> 20%)
const float r   = 0.05f;  // Interest rate (0.05 -> 5%)
const float T   = 3.0f;   // Maturity (3 -> 3 years)
const float S0  = 100.0f; // Initial stock price
const float K   = 100.0f; // Strike price
```

FIGURE 19.3

Constants.

```
float GetOptionPrice()
{
  float C;
  float d1, d2, p1, p2;

  d1 = (logf(S0 / K) + (r + sig * sig * 0.5f) * T) /
       (sig * sqrtf(T));
  d2 = (logf(S0 / K) + (r - sig * sig * 0.5f) * T) /
       (sig * sqrtf(T));
       p1 = cdfnormf(d1);
       p2 = cdfnormf(d2);
       C  = S0 * p1 - K * expf((-1.0f) * r * T) * p2;

       return C;
     }
```

FIGURE 19.4

Preliminary version.

```
int numThreads = 1;        // The number of threads
                           // (will be useful later)
int N;                     // The number of options
```

FIGURE 19.5

Variables.

First, we will discuss the data storage requirements of our program. It appears to be trivial. We need four arrays (three to hold input data and one for the results). We also need a few scalar variables.

Data layout in memory can significantly impact software performance. So, in many applications similar to ours, a dilemma arises: whether to use the SoA pattern (structure of arrays) or the AoS pattern (array of structures).

- In the SoA case, data will be organized in memory as follows: the first array comes as a whole block, then the second array, etc.
- In the AoS case: all data related to the first subject domain object comes first, then all data related to the second subject domain object, etc.

Generally, the question of which data layout is better has no predefined answer. However, in certain cases it is possible to make recommendations. The AoS pattern allows localization of memory access and potentially maximal reuse of processor cache, but at the price of more complicated addressing for accessing individual structure members. With the SoA data layout, cache memory can be utilized effectively for several arrays simultaneously in smaller tasks with independent iterations and straightforward memory access requirements. This arrangement simplifies addressing and reduces the total number of machine instructions. Our code belongs to the second type of application, and the SoA pattern helps achieve significant gains in performance(~3×) as will be shown in the second version of **`GetOptionPrices()`** function. Therefore, we choose to use SoA data pattern.

A **`GetOptionPrices()`** function utilizing an SoA data layout is shown in the following example (Figure 19.6).

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
  float *pC)
{
  int i;
  float d1, d2, p1, p2;
  for (i = 0; i < N; i++)
  {
    d1 = (log(pS0[i] / pK[i]) + (r + sig * sig * 0.5) *
          pT[i]) / (sig * sqrt(pT[i]));
    d2 = (log(pS0[i] / pK[i]) + (r - sig * sig * 0.5) *
          pT[i]) / (sig * sqrt(pT[i]));
    p1 = cdfnormf(d1);
    p2 = cdfnormf(d2);
    pC[i] = pS0[i] * p1 - pK[i] *
            exp((-1.0) * r * pT[i]) * p2;
  }
}
```

FIGURE 19.6

Reference version 1.

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970

FIGURE 19.7

Reference version. Time in seconds.

We compile the code using the Intel C ++ Compiler with a **`-O2`** key and launch it on the Intel Xeon host processor. Execution time depends on the number of options, N, shown in Figure 19.7.

REFERENCE VERSION—DO NOT MIX DATA TYPES

A typical mistake made by many C programmers (and not just beginners) is to mix "float" and "double" types when working with floating-point numbers. Let us consider how precision influences performance. Specifically, we consider the performance impact when data are represented in "float" (32-bit single precision) type and "double" (64-bit double precision) type when calling mathematical functions to perform calculations. Notice that in Figure 19.8 code for the **`GetOptionPrice()`** function, the single precision function **`logf()`** is called. Should the programmer call the **`log()`** function instead, the compiler will have to perform precision conversions as follows: function **`log()`** in the C language is passed a "double" argument and returns a "double" result, so the element of **`pT[i]`** array will first be converted from "float" type to "double" after which all calculations will be carried out using the "double" type. Only at the very end of the expression evaluation, during the result assignment to d1, is a backward conversion to "float" type performed. Taking into account that "float" numbers in many cases are processed faster than "doubles" (in particular, the difference becomes noticeable when using vectorization), such unexpected conversions can have an adverse effect on overall performance. We will check the difference in our case.

For this purpose, we modify the code by calling functions **`logf()`**, **`sqrtf()`**, and **`expf()`** and by properly specifying constants (meaning that constant 1.0 that does not have a suffix "f" will be stored as double). The changes are indicated in boldface.

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
  float *pC)
{
  int i;
  float d1, d2, p1, p2;
  for (i = 0; i < N; i++)
  {
    d1 = (logf(pS0[i] / pK[i]) + (r + sig * sig * 0.5f) *
          pT[i]) / (sig * sqrtf(pT[i]));
    d2 = (logf(pS0[i] / pK[i]) + (r - sig * sig * 0.5f) *
          pT[i]) / (sig * sqrtf(pT[i]));
    p1 = cdfnormf(d1);
    p2 = cdfnormf(d2);
    pC[i] = pS0[i] * p1 - pK[i] *
            expf((-1.0f) * r * pT[i]) * p2;
  }
}
```

FIGURE 19.8

Reference version 2.

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970
Do not mix data types	16.776	33.549	50.337	66.989

FIGURE 19.9

Do not mix data types. Time in seconds.

Using the abovementioned infrastructure, the authors obtained the following execution times (Figure 19.9). As Figure 19.9 shows, the execution time was slightly reduced if compared to reference version in all data sizes. Let us look at the **`GetOptionPrice()`** function profile collected with Intel VTune Amplifier XE (Figure 19.10) to better understand the reasons.

The VTune *Basic Hotspots* analysis shows that most of the time is spent in the **`cdfnormf()`** function that hides the overhead of the double precision **`exp()`**, **`log()`**, **`sqrt()`** computations. Nevertheless, the technique is generally applicable and we have used it previously to improve performance by several times in certain cases. The conclusion: Do not mix data types if possible!

VECTORIZE LOOPS

Vectorization of performance-critical code is one of the most important and very effective optimization techniques. We can ask the compiler to use a set of AVX instructions by adding the **-mavx** switch into the compilation line (the **-O2** switch by default assumes only SSE2 instructions). However, our experiments show that the use of this flag did not change the execution time.

To understand the reasons, we add a request for a vectorization report from the compiler with the **-vec-report3** switch. The digit "3" corresponds to the level of detail of the report (the range is 1-6). Note the sixth report type (**−vec-report6**) offers additional recommendations concerning vectorization.

The vectorization report when compiling the code in Figure 19.8 is shown in Figure 19.11. The compiler reports a possible dependency between arrays in our key loop. Our experience is that diagnostics

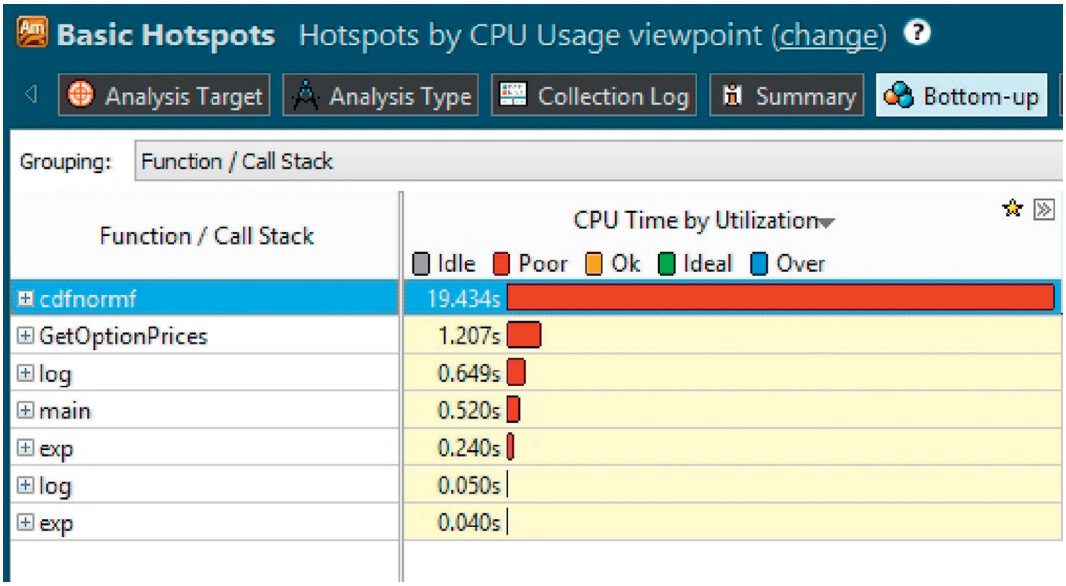

FIGURE 19.10

GetOptionPrice() function reference version profile, cdfnormf() is used.

```
sh-4.1$ icc -O2 -openmp -mavx -vec-report3 main.cpp -o option_prices
main.cpp(279): (col. 3) remark: loop was not vectorized: nonstandard loop is not
 a vectorization candidate
main.cpp(99): (col. 3) remark: loop was not vectorized: nonstandard loop is not
a vectorization candidate
main.cpp(115): (col. 3) remark: loop was not vectorized: nonstandard loop is not
 a vectorization candidate
main.cpp(131): (col. 3) remark: loop was not vectorized: nonstandard loop is not
 a vectorization candidate
sh-4.1$
```

FIGURE 19.11

Vectorization report.

from the Intel C/C ++ compilers very often helps the programmer identify exactly what hampers vectorization. In this case, we have a bit more work to do to identify the root issue.

Based on the vectorization report, let us "persuade" the compiler to vectorize the loop (Figure 19.12). Different ways can be used to "explain" to the compiler that there are no dependencies between arrays. Let us consider these ways in more detail.

1. Using the keyword **restrict** in declaration of the formal parameters of function (**restrict** is a keyword in the C99 standard). It will tell the compiler it is safe to vectorize the loop because there are no overlapping memory references.
2. Using the directive **#pragma ivdep** before the loop telling the compiler to ignore possible data dependencies. This directive will be ignored if the compiler can "prove" the existence of

```
for (i = 0; i < N; i++)
{
  ...
}
```

FIGURE 19.12

Main computational loop.

a dependency. Caution should be exercised when using **ivdep** as it can cause code with subtle, undetected "real" dependencies to produce incorrect results when vectorized.

3. Using the directive **#pragma simd** before the loop. This directive requests the compiler to vectorize the code regardless of any dependencies. Therefore, the programmer is responsible to "guarantee" against any issues preventing vectorization. This directive is the most aggressive compiler optimization, but responsibility for results is assumed by the programmer. Use this method only when you are sure no data dependencies exist.
4. Apart from the abovementioned methods, it is possible to specify a special option –**ansi-alias,** indicating to the compiler that there are no overlapping (in memory) arrays in the program. However, this option affects the **whole** source file and can unexpectedly "break" the code, if used without caution.

We can successfully vectorize the code in Figure 19.8 using any of the presented techniques. The results are provided in Figure 19.13.

It should be noted that the execution time of the vectorized version is about 8% lower than the scalar version on an Intel Xeon processor. The Intel Xeon E5-2690 processor used in our experiments supports AVX instructions allowing eight single precision floating-point elements to be processed at the same time. Why was not the code accelerated by 8x? We infer that in this example, the overhead of data packing into vector registers and unpacking the results must be negligible because of the use of the SoA pattern which keeps the data contiguously in memory.

To understand the lower than expected performance increase, we profile the application using Intel VTune Amplifier XE both before and after vectorization (see Figures 19.10 and 19.14).

Inspecting the profile, we notice that **`logf()`** and **`expf()`** functions were replaced by their respective vector analogs from the SVML (short vector math library) runtime library from the Intel Compiler. VTune also reports that the time-consuming **`cdfnormf()`** function remained scalar. This explains the lack of significant speedup from vectorization as **`cdfnormf()`** is responsible for the 90% of the runtime. *Note*: future versions of the compiler could improve the vectorization of **cdfnormf( ).**

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970
Do not mix data types	16.776	33.549	50.337	66.989
Vectorize loops	15.445	30.977	46.608	62.141

FIGURE 19.13

Vectorize loops. Time in seconds.

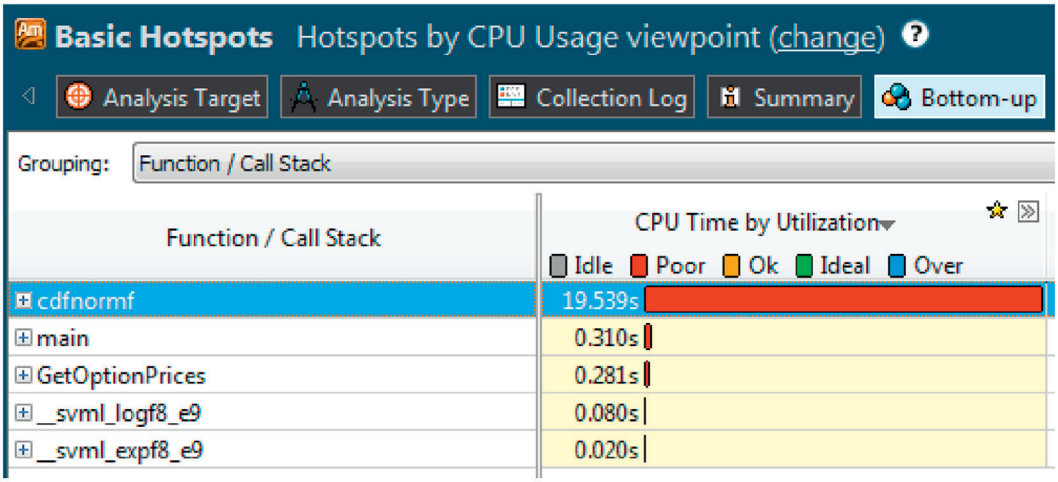

FIGURE 19.14

GetOptionPrice() function profile, cdfnormf() is used, after vectorization.

The lack of a vectorized math function analog is quite illustrative. We cannot expect further improvements without achieving full vectorization of the hot spots, especially on the coprocessor which can process 16 single precision floating-point elements per vector. However, we are able to get past this problem with the help of basic arithmetic and the profiler.

USE FAST MATH FUNCTIONS: ERFF() VS. CDFNORMF()

It is common that some math functions in a compiler may be better optimized than others. In our example, the calculations performed with the help of **cdfnormf()** function can be achieved using the **`erff()`** function (Figure 19.15).

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
  float *pC)
{
  int i;
  float d1, d2, erf1, erf2;
  for (i = 0; i < N; i++)
  {
    d1 = (logf(pS0[i] / pK[i]) + (r + sig * sig * 0.5f) *
         pT[i]) / (sig * sqrtf(pT[i]));
    d2 = (logf(pS0[i] / pK[i]) + (r - sig * sig * 0.5f) *
         pT[i]) / (sig * sqrtf(pT[i]));
    erf1 = 0.5f + 0.5f * erff(d1 / sqrtf(2.0f));
    erf2 = 0.5f + 0.5f * erff(d2 / sqrtf(2.0f));
    pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
            pT[i]) * erf2;
  }
}
```

FIGURE 19.15

erff version.

The next example recodes the GetOptionPrices() using the following formula:

$$\texttt{cdfnorm}(x) = 0.5 + 0.5\,\mathrm{erf}\left(\frac{x}{\sqrt{2}}\right) \tag{19.6}$$

Using the infrastructure mentioned in Figure 19.2, the authors obtained the run times shown in Figure 19.16. We expected the **`erff()`** function to be faster than **`cdfnormf()`** given its numerical properties allow simpler floating-point approximation. However, that was not the only reason for the speedup. Let us inspect the VTune profile again.

Figure 19.17 shows that the compiler has employed an SVML vector analog for **`erff()`**, which provided a significant speedup resulting in 29× improvement versus the scalar code with **`cdfnormf()`**. Not bad! This result also shows a potential for further improvement with a coprocessor given its even wider vector instructions.

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17,002	34,004	51,008	67,970
Do not mix data types	16,776	33,549	50,337	66,989
Vectorize loops	15.445	30.977	46.608	62.141
Use fast math functions + improved vectorization	0.522	1.049	1.583	2.091

FIGURE 19.16

Use fast math functions. Time in seconds.

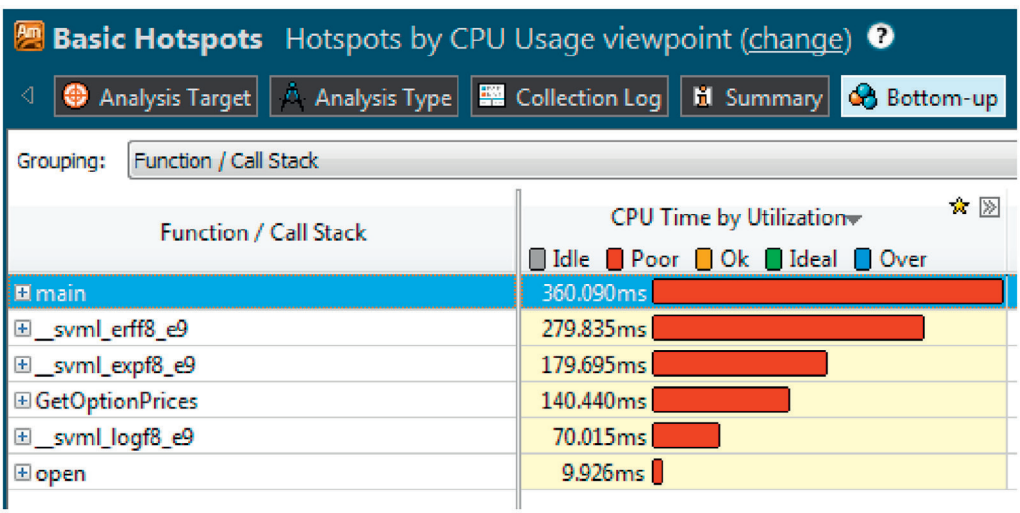

FIGURE 19.17

GetOptionPrice() function profile, erff() is used, after vectorization.

EQUIVALENT TRANSFORMATIONS OF CODE

The program can be optimized a little bit more by lifting "invariant" one-time calculations out of the loop. In this case, **`1.0f / sqrtf(2.0f)`**.

We can evaluate this expression separately by introducing the following constant (Figure 19.18):

There is also the noteworthy method of replacing division by multiplication. In some situations, this optimization leads to a considerable execution time reduction. In this example, we can replace **`1/sqrtf()`** expression by the **`invsqrtf()`** function. We can check, whether the compiler performed these substitutions automatically.

As is shown in Figure 19.19, the execution time of the new code version almost equals the time obtained earlier. Most likely, the compiler had performed the necessary substitutions in the code without our help. That can be confirmed by inspecting the compiler's assembler listing (e.g., add **-Fa** switch to the command line).

Despite the fact that the execution time hardly changed, the loop invariants hoisting and replacement of division by multiplication optimizations we performed are generally useful and in the case of a less "intelligent" compiler, they often provide a benefit. Retaining these optimizations may help when porting to a different architecture so we will take this version (Figure 19.20) as the new base line, plus we will include them in our upcoming coprocessor experiments.

ALIGN ARRAYS

Another potentially useful software optimization deals with data alignment. SIMD register-size aligned data accesses are performed much faster by the processor than unaligned ones. In some cases, the compiler and/or hardware can minimize the performance impact, but often significant performance increases—especially for vector codes—can be achieved by ensuring alignment. For this reason, it is worthwhile to check memory address alignment. To guarantee alignment, we replace **`new/delete`** operators with calls to the **`memalign()/free()`** functions. The rest of the code does not change (Figure 19.21).

```
const float invsqrt2 = 0.707106781f;
```

FIGURE 19.18

"1.0f / sqrtf(2)" constant.

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970
Do not mix data types	16.776	33.549	50.337	66.989
Vectorize loops	15.445	30.977	46.608	62.141
Use fast math functions + improved vectorization	0.522	1.049	1.583	2.091
Equivalent transformations	0.538	1.071	1.614	2.133

FIGURE 19.19

Equivalent transformations of code. Time in seconds.

```
void GetOptionPrices(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;
   }
}
```

FIGURE 19.20

New base line version.

```
int main(int argc, char *argv[])
{
  pT = (float *)memalign(32, 4 * N * sizeof(float));
//  pT  = new float[4 * N];

  ...

  free(pT);
//  delete [] pT;
  return 0;
}
```

FIGURE 19.21

Data alignment.

The recommended value of alignment (the first parameter in **`memalign()`** function) depends on the width of the SIMD registers in use. For SSE instructions, use 16 bytes, for AVX instructions—32 bytes, and for the coprocessor instruction set—64 bytes.

It is also useful to add one more directive into the code before the loop:

`#pragma vector aligned`

This directive will inform the compiler that the arrays in the loop are aligned and corresponding aligned memory read/write instructions can be used. It should be noted that if we "deceive" the compiler, the software could crash in an attempt to use aligned access on misaligned data.

In this case, the compiler had already coped with the alignment: Figure 19.22 shows no difference between the last two experiments, but it does not mean that it will always be like this, and for all compilers. Our experience indicates you should align your data whenever possible!

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970
Do not mix data types	16.776	33.549	50.337	66.989
Vectorize loops	15.445	30.977	46.608	62.141
Use fast math functions + improved vectorization	0.522	1.049	1.583	2.091
Equivalent transformations	0.538	1.071	1.614	2.133
Align arrays	0.539	1.072	1.617	2.135

FIGURE 19.22

Align data. Time in seconds.

REDUCE PRECISION IF POSSIBLE

Earlier we said that there is no need to use **`double`** type in the solution of this particular problem. Now, we go further. Here **`float`** type single precision is also excessive (as not more than four decimal digits are used in a subject domain). Precision decrease provides potential for further optimization of calculations.

The following Intel compiler command-line options influence the precision of calculation of mathematical functions.

```
icc … -fimf-precision=low -fimf-domain-exclusion=31
```

The **`-fimf-precision=low`** option tells the compiler to use implementations of mathematical functions with 11 accurate bits in mantissa (out of 24 bits available in single precision), which more closely corresponds to the precision of the input parameters. The **`-fimf-domain-exclusion`** option allows special values in mathematical functions (such as infinity, NaN, and extreme values of arguments) to be excluded from processing. This command-line option can be used when it is safe to assume that the program does not have to deal with such extreme values.

Figure 19.23 shows that the use of the lower precision command-line options reduced the application runtime by approximately 23%. Note that the reduced precision also impacted the numeric result

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970
Do not mix data types	16.776	33.549	50.337	66.989
Vectorize loops	15.445	30.977	46.608	62.141
Use fast math functions + improved vectorization	0.522	1.049	1.583	2.091
Equivalent transformations	0.538	1.071	1.614	2.133
Align arrays	0.539	1.072	1.617	2.135
Reduce precision	0.438	0.871	1.314	1.724

FIGURE 19.23

Reduce precision if possible. Time in seconds.

of the **`GetOptionPrices()`** function: we got 20.920 instead of a reference 20.924, but the difference does not occur until the 5th decimal digit. It has to be specially mentioned though that proper accuracy analysis should always be carried out in order to safely use the precision controls.

WORK IN PARALLEL

Just like vectorization helps saturate the core resources available to the single-threaded application, thread-level parallelization helps to distribute the work across the multiple cores and single-core resources available in SMT architectures. Parallelization of the computational loop across processing units (e.g., threads and cores) is rather simple due to the absence of any dependencies between iterations. It is simply enough to add **omp parallel for** pragma before the loop (as noted in boldface in Figure 19.24 below) and to localize all writable variables via the **private** list.

Note that **-openmp** compiler command-line switch was added earlier, so additional command line changes are not required.

As Figure 19.25 shows, scaling improves with the growth of data volume, from 7.59x to 11.27x for 240 million samples.

USE WARM-UP

The times of execution in the last experiment became rather short; therefore the overhead costs for thread creation may include a significant contribution to the parallel region execution time. Let us try and remove that overhead. The approach uses the fact that the majority of OpenMP implementations do not destroy the threads created for the first parallel section of code but put them into a sleep state, so they can be resumed much more quickly on subsequent use. This is known as creating a "thread pool."

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
  float *pC)
{
  int i;
  float d1, d2, erf1, erf2, invf;
  float sig2 = sig * sig;
#pragma simd
#pragma omp parallel for private(d1, d2, erf1, erf2,
invf)
  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;
  }
}
```

FIGURE 19.24

OpenMP version.

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970
Do not mix data types	16.776	33.549	50.337	66.989
Vectorize loops	15.445	30.977	46.608	62.141
Use fast math functions + improved vectorization	0.522	1.049	1.583	2.091
Equivalent transformations	0.538	1.071	1.614	2.133
Align arrays	0.539	1.072	1.617	2.135
Reduce precision	0.438	0.871	1.314	1.724
Work in parallel (16 cores)	0.058	0.084	0.126	0.153

FIGURE 19.25

Work in parallel. Time in seconds, 16 cores are used.

Accordingly, if **`GetOptionPrices()`** is called two times in a row in **`main()`** function, then the second call overhead costs will be minimized. This approach is called "warm-up."

Is warm-up fair for benchmarking? The authors' experience is that real production programs always create threads long before formula computations are executed, so thread creation overhead is reasonable to exclude for benchmarking measurements. However, additional savings may also come from the data cache warm-up: two subsequent calls of the same function using the same data may result in different execution times due to the fact that the first call will suffer from cache misses, while the second one will likely benefit from cache hits due to the data remaining in high-speed cache.

Is this good or bad? On one hand, it is not completely fair: there is a well-known benchmarking mistake—carrying out single-threaded and multi-threaded experiments (operating on the same dataset) within a single process. The resulting super-linear speedup immediately disappears when experiments are done correctly—each in its own process. On the other hand, in real programs, other setup computations are typically carried out around the hot-spot function gradually loading target calculation data into the cache.

Generally, we recommend the following:

1. In case of long execution times, e.g., seconds, warm-up is not needed.
2. If execution times are small (fractions of a second), and the number of threads is large, it makes sense to apply the warm-up, otherwise the real world algorithm execution time will be hidden by the overheads.
3. If it is suspected that preloading the cache is distorting the performance timing results, the programmer can introduce a special dummy parallel section outside and prior to the measured region for the sole purpose of thread creation. This approach only removes the thread startup overhead without touching the caches.

The difference between "Reduce precision" and "Reduce precision + warm cache" results in Figure 19.26 illustrates the benefits of operating in "warm" cache. Warm-up also allowed us to exclude the thread creation overhead from the measurements in parallel versions (last row in Figure 19.26), and see that the computational part indeed scaled by 13x on 16 cores.

N	60 000 000	120 000 000	180 000 000	240 000 000
Reference version	17.002	34.004	51.008	67.970
Do not mix data types	16.776	33.549	50.337	66.989
Vectorize loops	15.445	30.977	46.608	62.141
Use fast math functions + improved vectorization	0.522	1.049	1.583	2.091
Equivalent transformations	0.538	1.071	1.614	2.133
Align arrays	0.539	1.072	1.617	2.135
Reduce precision	0.438	0.871	1.314	1.724
Reduce precision + warm cache	0.409	0.812	1.226	1.603
Work in parallel (16 cores)	0.058	0.084	0.126	0.153
Parallel, warm cache, threads creation overhead excluded	0.033	0.062	0.091	0.118

FIGURE 19.26

Use warm-up. Time in seconds.

USING THE INTEL XEON PHI COPROCESSOR—"NO EFFORT" PORT

The coprocessor is showcased best through its effective use of a large number of threads (from 120 to 240), provided that the code also vectorizes well. Let us see what a simple recompilation based "no effort port" could give us. Furthermore, we will analyze the influence of optimizations applied on the processor. We start experiments from the "Equivalent transformations" version, taking it as a baseline.

To build our program for a coprocessor, we simply add **-mmic** switch to the compilation line. In addition, we should not forget to increase the alignment factor in **`memalign()`** call from 32 to 64 for the coprocessor architecture.

Figure 19.27 contains the timings of the sequential version on the coprocessor compared to the best sequential version on the processor. It should be noted that for the coprocessor transition to calculations

N	60 000 000	120 000 000	180 000 000	240 000 000
The best sequential CPU version	0.409	0.812	1.226	1.603
Equivalent transformations	1.544	3.089	4.633	6.174
Align arrays	1.545	3.091	4.634	6.179
Reduce precision	0.676	1.352	2.027	2.703
Reduce precision + warm cache	0.422	0.845	1.269	1.690

FIGURE 19.27

Using Intel Xeon Phi coprocessor. Time in seconds, serial version.

with reduced precision allows 2.3x gains versus 23% on the CPU. It is also shown that warm cache gives a more considerable gain (about 60%). Finally, it is noteworthy that the execution times of the sequential versions on the coprocessor and processor are almost equal.

USE INTEL XEON PHI COPROCESSOR: WORK IN PARALLEL

Naturally, the potential of the coprocessor is clearly seen when using all its sources of parallelism: both SIMD and its many-cores with SMT. Now we look at parallel versions on the coprocessor. It should be noted that, so far, we have not applied any optimization specific to the coprocessor. We just recompiled the same source version of the code we used for the processor and ran it on the coprocessor in native mode.

As Figure 19.28 shows, the version without warm-up is hardly accelerated, which supports the assumption that the thread creation overheads are comparable to the execution time of the program.

Adding the warm-up, we exclude the one-time overheads from the measurements and are now able to see how the computational part scales: speedup varies from 50.5 to 60.4. Note that there is good scaling until the thread count reaches the number of cores. Adding more threads to fully utilize the execution pipelines on each core somewhat improves the situation in the 120 threads case (i.e., two threads per core), but things get worse when we fully subscribe the machine with 240 threads (i.e., four threads per core) as shown in Figure 19.28. It is clear that, what worked well for 16 threads on the processor does not scale well to 240 threads. Let us see, if there is anything we can do to improve this for the coprocessor.

N	60 000 000	120 000 000	180 000 000	240 000 000
Work in parallel, 60 threads	0.134	0.149	0.164	0.175
Speedup	5.0336	9.050	12.331	15.437
60 threads, no overhead	0.008	0.017	0.025	0.033
Speedup	50.585	51.178	51.783	51.546

N	60 000 000	120 000 000	180 000 000	240 000 000
Work in parallel, 120 threads	0.234	0.255	0.257	0.255
Speedup	2.885	5.303	7.883	10.590
120 threads, no overhead	0.007	0.014	0.021	0.028
Speedup	59.422	59.587	60.389	59.839

N	60 000 000	120 000 000	180 000 000	240 000 000
Work in parallel, 240 threads	0.532	0.527	0.533	0.558
Speedup	1.269	2.564	3.800	4.842
240 threads, no overhead	0.008	0.016	0.024	0.031
Speedup	53.286	54.248	53.969	53.964

FIGURE 19.28

Use Intel Xeon Phi Coprocessor. Time in seconds, 60, 120, 240 threads are used.

USE INTEL XEON PHI COPROCESSOR AND STREAMING STORES

`GetOptionPrices()` function works with four arrays (**`pT`**, **`pK`**, **`PS0`**, **`pC`**). Three of the arrays are used as read-only inputs, and one (**`pC`**) is written to as the output. Please note that with the current benchmark data organization, the lengthy **`pC`** array is only accessed once in the loop. Thus, the **`pC`** array data does not need to be cached and may be tagged nontemporal. In other words, we now come to the conclusion that our example has hit the memory bandwidth wall which prevents it from scaling with increased thread count.

Streaming store instructions on the coprocessor should help us to save bandwidth by eliminating cache coherency traffic (known as "RFO—read-for-ownership" (traffic)). Computing X options requires 3X reads and X writes, plus X RFO requests to maintain cache coherency upon writes, total 5X. The use of streaming stores eliminates the RFO requests and reduces the traffic to 4X. If our benchmark is memory bandwidth limited, then we would expect 5X/4X = 1.25 speedup from this optimization. We got close to that in case of the largest data set: 0.031/0.026 = 1.19, see Figure 19.29 (Figure 19.30).

As a result, evaluation of 240,000,000 options using the coprocessor version is approximately 4.5 times faster than the host processor version, but we hit the memory bandwidth wall which prevents efficient scaling of the program beyond 65.4x.

SUMMARY

The Black-Scholes formula for pricing European options is a *de facto* standard financial benchmark. Despite the fact that several assumptions in the model invented by Black and Scholes are rarely satisfied in real life, the Black-Scholes formula is still widely used in practice. A typical problem being solved is pricing of millions of options simultaneously. Such a problem is time consuming, requiring significant compute power and, therefore, is relevant to the HPC field. This raises the challenge of a high-performance implementation of Black-Scholes formula on modern multicore and many-core hardware.

N	60 000 000	120 000 000	180 000 000	240 000 000
The best parallel CPU version	0.033	0.062	0.091	0.118
Work in parallel + Warm-up, 120 threads	0.007	0.014	0.021	0.028
Work in parallel + Warm-up, 240 threads	0.008	0.016	0.024	0.031
Work in parallel + Warm-up, streaming stores, 120 threads	0.007	0.013	0.019	0.026
Work in parallel + Warm-up, streaming stores, 240 threads	0.007	0.013	0.019	0.026

FIGURE 19.29

Use Intel Xeon Phi coprocessor and streaming stores. Time in seconds.

```
void GetOptionPrices(float *pT, float *pK, float *pS0,
  float *pC)
{
  int i;
  float d1, d2, erf1, erf2, invf;
  float sig2 = sig * sig;
#pragma simd
#pragma vector nontemporal
#pragma omp parallel for private(invf, d1, d2, erf1,
erf2)
  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;
  }
}
```

FIGURE 19.30

Streaming stores version.

This chapter presented a study of performance optimization of Black-Scholes formula computation on Intel Xeon processors and Intel Xeon Phi coprocessors from a very basic to a heavily tuned implementation. Step-by-step analysis of typical optimization techniques showed performance improvements on both processor and coprocessor. The final optimized version computes 2067 million options per second on the processor and 9231 million options per second on the coprocessor in single precision floating point. We illustrated how the performance limiting factors changed with application optimizations—namely the processor application is compute bound while the coprocessor memory bandwidth appears to be the limiting factor after the compute intensive mathematical calculations are tuned.

The study with the source codes has been published as a part of "Programming on Intel Xeon Phi coprocessors" tutorial in Russian (http://hpc-education.unn.ru/ru/obuchenie/courses/xeon-phi) and the code is also available at: http://lotsofcores.com. Work was prepared in UNN HPC Center in collaboration with Intel Numerics engineers.

FOR MORE INFORMATION

- Knuth, D., 1997. The Art of Computer Programming, Seminumerical Algorithms, vol. 2, third ed. Addison-Wesley Professional, 784 p.
- Li, S. Achieving Superior Performance on Black-Scholes Valuation Computing using Intel® Xeon Phi™ Coprocessors. https://software.intel.com/en-us/articles/case-study-achieving-superior-performance-on-black-scholes-valuation-computing-using.
- Jeffers, J., Reinders, J., 2013. Intel® Xeon Phi™ Coprocessor High Performance Programming. Copyright, Morgan Kaufman.

- Smelyanskiy, M., Sewall, J., Kalamkar, D.D., et al., 2012. Analysis and optimization of financial analytics benchmark on modern multi- and many-core IA-based architectures. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 1154-1162.
- Bastrakov, S., Meyerov, I., Gergel, V., et al., 2013. High performance computing in biomedical applications. Procedia Comput. Sci. 18, 10-19.
- Black, F., Scholes, M.S., 1973. The pricing of options and corporate liabilities. J. Polit. Econ. 81(3), 637-654.