Performance optimization of Black Scholes calculation

Manel Fernández

Intel HPC Software Workshop Series 2016
HPC Code Modernization for Intel® Xeon and Xeon Phi™
February 18th 2016, Barcelona
Development cycle

Analysis
- Intel® VTune™ Amplifier, Intel® Advisor

Design (introduce parallelism)
- Intel® IPP, Cilk Plus, MKL, OpenMP, TBB
- Intel® Compilers

Debug for correctness
- Intel® Parallel Inspector XE
- Debugger, Static Analysis

Tune for performance
- Intel® VTune™ Amplifier, Intel® Advisor
General workflow optimization

Step-by-step optimization workflow ...

Baseline
Modification of publicly available (sometimes “naïve”) code – compare non-Intel and Intel compiler

Optimized/Vectorized
Baseline code modifications for effective use of SSE/AVX by Intel Compiler; alternative Array Section Notation of Intel Cilk Plus

Threaded
Auto parallelization with Intel Compiler, OpenMP*, TBB or Cilk threading of Intel Compilers

Intel® Performance Library Enabled
Code modifications to enable e.g. Intel® MKL features like VSL – Vector Statistical Library or IPP multi-media routines
Case study: Black Scholes calculation

Myron Scholes, Robert Merton won 1997 Nobel Economics Prize for their “Pioneering work in Derivative Pricing”

Techniques/sample code we use is derived from code published by Bernt Arne Odegaard, http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/
Black Scholes option value model

Assumptions
• The option is European, exercisable only on the maturity date.
• No dividends are paid by the stock
• Short term interest rate is known constant
• No transaction fee
• The stock price follows ‘Geometric Brownian Motion’

Financial application computes values for huge number of input parameter sets
• Two steps: Initialization and computation
• Initialization (meaningful random data) can be more expensive than option value computation
Black-Scholes-Merton formula
Model for the future value of options

Problem definition

**call**: European call option, allows buyer to buy option
**put**: European put option, allows seller to sell option

\[
\text{call} = \text{CND}(d_1)S_0 - \text{CND}(d_2)Ke^{-rT} \\
\text{put} = \text{CND}(-d_2)Ke^{-rT} - \text{CND}(-d_1)S_0
\]

where

- \( S_0 \): Current stock price
- \( K \): Option strike price
- \( r \): Risk free rate
- \( \sigma \): Volatility
- \( T \): Time to expiry

\[
\begin{align*}
d_1 &= \frac{1}{\sigma \sqrt{T}} \left[ \ln \frac{S_0}{K} + \left( r + \frac{\sigma^2}{2} \right) T \right] \\
d_2 &= \frac{1}{\sigma \sqrt{T}} \left[ \ln \frac{S_0}{K} + \left( r - \frac{\sigma^2}{2} \right) T \right]
\end{align*}
\]

\( \text{CND}(x) \): Cumulative normal distribution with variance \( x \)
Normal distribution

Distribution

\[ \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \]

Cumulative distribution

\[ \frac{1}{2} \left[ 1 + \text{erf} \left( \frac{x-\mu}{\sqrt{2\sigma^2}} \right) \right] \]
General workflow optimization

Step-by-step optimization workflow ...

- **Baseline**
  - Modification of publicly available (sometimes “naïve”) code – compare non-Intel and Intel compiler

- **Optimized/Vectorized**
  - Baseline code modifications for effective use of SSE/AVX by Intel Compiler; alternative Array Section Notation of Intel Cilk Plus

- **Threaded**
  - Auto parallelization with Intel Compiler, OpenMP*, TBB or Cilk threading of Intel Compilers

- **Intel® Performance Library Enabled**
  - Code modifications to enable e.g. Intel® MKL features like VSL – Vector Statistical Library or IPP multi-media routines
Baseline: initialization [1]

```c
void initBSdataRand(int vl, double* x, double* c,
                     double* t, double* variance, double* rate)
{
    for (int i=0; i<vl; i++)
    {
        x[i] = random_normal_mean_sigma(75.0, 10.0);
        c[i] = random_normal_mean_sigma(80.0, 10.0);
        t[i] = random_uniform_0_1() * 0.2 + 1.8;
        variance[i] = random_normal_mean_sigma(0.50, 0.20);
        rate[i] = random_normal_mean_sigma(0.05, 0.015);
    }
}
```
double random_uniform_0_1(void) {
    return ( double(rand()) / double(RAND_MAX) ) ; // rand()
};

double random_normal(void) {
    double U1, U2, V1, V2, S=2;
    while (S>=1) {
        U1 = random_uniform_0_1();  U2 = random_uniform_0_1();
        V1 = 2.0*U1-1.0;            V2 = 2.0*U2-1.0;
        S = pow(V1,2)+pow(V2,2);
    };
    return( V1*sqrt((-2.0*log(S))/S) ) ;
};

double random_normal_mean_sigma(double mean, double sigma) {
    return sigma*random_normal()+mean;
}
void calc_bsf(int vl, double* x, double* c, double* t, 
    double* variance, double* r, double *vcall, double *vput)
{
    double time_sqrt, d1, d2;
    for (int i=0; i<vl; i++)
    {
        time_sqrt = sqrt(t[i]);
        d1 = (log(x[i]/c[i])+r[i]*t[i])/
            (variance[i]*time_sqrt)+0.5*variance[i] * time_sqrt;
        d2 = d1-(variance[i]*time_sqrt);
        vcall[i] = x[i] * cnd(d1) - c[i] * exp(-r[i]*t[i]) * cnd(d2);
        vput[i] = exp(-r[i]*t[i]) * c[i] * cnd(-d2) - x[i] * cnd(-d1);
    }
}
Baseline: calculation [2]

double cnd(const double& z)
{
    double b1 = 0.31938153, b2 = -0.356563782;
    double b3 = 1.781477937, b4 = -1.821255978;
    double b5 = 1.330274429, p = 0.2316419;
    double c2 = 0.3989423, a = fabs(z);
    double t = 1.0/(1.0+a*p);
    double b = c2 * exp((-z)*(z/2.0));
    double n = (((b5*t+b4)*t+b3)*t+b2)*t+b1)*t;
    n = 1.0-b*n;
    if ( z < 0.0 ) n = 1.0 - n;
    return n;
};
Test system used

Lenovo T420 notebook.
- Intel® T2520M processor
- 2 cores, Hyperthreading enabled
- SandyBridge architecture
- 2.53 GHz
- 4.00 GB memory

Microsoft Windows® 7 operating system

Intel® Parallel Studio XE 2016 Beta, Update 1
Baseline: performance

<table>
<thead>
<tr>
<th></th>
<th>cyc(option) Initialization</th>
<th>cyc(option) calculation</th>
<th>cyc(option) sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS VS 2013 SP4 - CL, base line</td>
<td>1055</td>
<td>403</td>
<td>1458</td>
</tr>
<tr>
<td>Intel 16.0 Beta- ICL, base line</td>
<td>942</td>
<td>409</td>
<td>1351</td>
</tr>
</tbody>
</table>

Where is the time spent?

- Intel® VTune™ Amplifier XE for performance analysis
- We compile Intel-compiler version but explicitly switch off in-lining
  - /Qip-no-lining
- Performance data can be misleading otherwise
  - And for this sample code it really does...
- For now, we only need basic hotspot analysis
Intel® VTune™ Amplifier XE
Supports for all steps of a systematic performance analysis

Where is my application…

**Spending Time?**
- Focus tuning on functions taking time (hotspots)
- See call stacks
- See time on source

**Wasting Time?**
- See cache misses on your source
- See functions sorted by # of cache misses

**Waiting Too Long?**
- See locks by wait time
- Red/Green for CPU utilization during wait

Advanced profiling for scalable multicore performance
Hotspots

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Calculation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Random_normal</td>
<td>19.9</td>
<td>Exp 11.4</td>
</tr>
<tr>
<td>getpd_noexit</td>
<td>15.2</td>
<td>cnd 9.1</td>
</tr>
<tr>
<td>Log</td>
<td>13.9</td>
<td></td>
</tr>
<tr>
<td>random_uniform_0_1</td>
<td>7.6</td>
<td></td>
</tr>
<tr>
<td>Random_normal_mean</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>initBSdata_Rand</td>
<td>1.5</td>
<td></td>
</tr>
</tbody>
</table>
General workflow optimization

Step-by-step optimization workflow ...

Baseline
Modification of publicly available (sometimes "naïve") code – compare non-Intel and Intel compiler

Optimized/Vectorized
Baseline code modifications for effective use of SSE/AVX by Intel Compiler; alternative Array Section Notation of Intel Cilk Plus

Threaded
Auto parallelization with Intel Compiler, OpenMP*, TBB or Cilk threading of Intel Compilers

Intel® Performance Library Enabled
Code modifications to enable e.g. Intel® MKL features like VSL – Vector Statistical Library or IPP multi-media routines
Optimizing the calculation

Compute very same sub-expressions only once
User “Error Function” routine erf() of Intel libimf to compute cnd():
\[ \text{cnd}(x) = 0.5 \times \text{erf}\left( \frac{x}{\sqrt{2.0}} \right) \]
Expression reduction and simplification like:
Save two calls of cnd() due to:
\[ \text{cnd}(-x) = 1 - \text{cnd}(x) \]

<table>
<thead>
<tr>
<th></th>
<th>initialization</th>
<th>calculation</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICL, base line</td>
<td>942</td>
<td>409</td>
<td>1351</td>
</tr>
<tr>
<td>ICL, “optimized”</td>
<td>961</td>
<td>417</td>
<td>1378</td>
</tr>
</tbody>
</table>

No gain / bad move: ICL is doing excellent job optimizing baseline version
We continue to use this version nevertheless since erf() is available as vectorized version (svml library)
Automatic vectorization: concept

Transforming sequential code to exploit the vector SIMD/AVX processing capabilities

\[
\text{for (i=0; i<MAX; i++)} \\
\quad c[i] = a[i] + b[i];
\]
Intel® Advisor XE 2016
A design/analysis tool for threading and vectorizing your code

Survey analysis
- See what prevents vectorization
- Detect vectorization issues
- Source/assembly integration
- Optimization reports
- Automatic recommendations

Trip-count analysis
- How many iterations in a loop
- Quantify peel/main/remainder

Deeper analyses
- Correctness analysis to see if a loop can be safely vectorized
- Memory access pattern (MAP) to figure out actual vectorization stride

Complete tutorial in latest Intel’s magazine “The Parallel Universe” (Issue 22)
Automatic vectorization for BS

Doesn’t work for ‘Initialization’ since there is no vectorized version of ‘rand()’

For ‘Calculation’-loop, compiler needs assertion, that arguments do not overlap.
Many potential ways:
  • ‘restrict’ keyword
  • pragma ‘ivdep’ in front of loop
  • Compiler switch /Qalias-args-, Oa, ...

Declaration (mm_malloc) and “pragma vector aligned” take care of required alignment

Vectorization for AVX ( /Qxavx )

/Qopt-report informs about success of vectorization

<table>
<thead>
<tr>
<th></th>
<th>initialization</th>
<th>calculation</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICL, “optimized”</td>
<td>961</td>
<td>417</td>
<td>1378</td>
</tr>
<tr>
<td>ICL, vectorized for AVX</td>
<td>913</td>
<td>77</td>
<td>990</td>
</tr>
</tbody>
</table>
General workflow optimization

Step-by-step optimization workflow ...

Baseline
Modification of publicly available (sometimes “naïve”) code – compare non-Intel and Intel compiler

Baseline code modifications for effective use of SSE/AVX by Intel Compiler; alternative Array Section Notation of Intel Cilk Plus

Optimized/Vectorized

Threaded
Auto-parallelization with Intel Compiler, OpenMP*, TBB or Cilk threading of Intel Compilers

Intel® Performance Library Enabled
Code modifications to enable e.g. Intel® MKL features like VSL – Vector Statistical Library or IPP multi-media routines
Threading by OpenMP*

Intel® Composer XE fully supports OpenMP* 4.0
- w/ minor feature exceptions

Switch: /QopenMP

```c
// assume N=12
#pragma omp parallel for
for(i = 1, i < N+1, i++)
    c[i] = a[i] + b[i];
```

Threads are assigned an independent set of iterations
Threads must wait at the end of work-sharing construct
void initBSdataRand(int vl, double* x, double* c, double* t, double* variance, double* rate)
{
    #pragma OMP parallel for
    for (int i=0; i<vl; i++)
    {
        x[i] = random_normal_mean_sigma(75.0,10.0);
        c[i] = random_normal_mean_sigma(80.0,10.0);
        t[i] = random_uniform_0_1() * 0.2 + 1.8;
        variance[i] = random_normal_mean_sigma(0.50,0.20);
        rate[i] = random_normal_mean_sigma(0.05,0.015);
    }
}
OpenMP in BS code

Can be used both, for ‘initialization’ and ‘calculation’ phase
The ‘calculation’ loop will be both, parallelized and vectorized, then
We need to make sure, we don’t introduce race conditions
  • Variables shared by multiple loop iterations?
  • rand() is not “thread-safe” - or?

<table>
<thead>
<tr>
<th></th>
<th>initialization</th>
<th>calculation</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICL, vectorized</td>
<td>913</td>
<td>77</td>
<td>990</td>
</tr>
<tr>
<td>ICL, vectorized + OpenMP threading</td>
<td>330</td>
<td>31</td>
<td>361</td>
</tr>
</tbody>
</table>

Huge gain: 4 OMP threads, but 2 cores + HT only!
Correctness checking

Did our threading introduce race conditions / deadlocks?

In general difficult to verify by pure code inspection or execution tests

- Library calls w/o source code
- Non-deterministic behavior
- Isn’t random number generation routine `rand()` known to keep state making it non re-entrant?
Intel® Inspector XE

Finds hard to detect coding defects
• Memory leaks and memory corruption
• Threading data races and deadlocks
• Static Security Analysis for source code checking

Supports different implementations of threading
• Native threads and Intel® Parallel Building Blocks

Works on standard builds and binaries
Can identify over 250 security errors

Advanced memory checking
Find errors like memory leaks and corruption
Make changes to source code in context of error
Thread checking
Finds data races and deadlocks
Security Static Analysis
Finds security errors including Buffer overruns and uninitialized variables
Intel® Cilk Plus™

Combines and integrates a task-parallel and data (vector) parallel programming language extension for C/C++:

- **Task parallelism**
  - Realized by Cilk as defined by MIT Cilk project
    - Only 3 simple keywords: `cilk_for`, `cilk_spawn`, `cilk_cync`
    - Hyperobjects like reducers for reductions etc
  - Exploits core/thread parallelism
  - In general no deterministic program execution

- **Data parallelism**
  - Realized by
    - C Array Notation: Array sections in C/C++
    - SIMD Pragma: new pragma class for vectorization
    - Elemental function: Functions operating on array elements in parallel
  - Exploits SSE/AVX parallelism
    - Not restricted to a single core however
  - Deterministic execution model
Implementation of cilk_for

cilk_for (int i=0; i<8; ++i) { f(i); }
void initBSdataRand(int vl, double* x, double* c, double* t, double* variance, double* rate) {
    cilk_for (int i=0; i<vl; i++) {
        x[i] = random_normal_mean_sigma(75.0,10.0);
        c[i] = random_normal_mean_sigma(80.0,10.0);
        t[i] = random_uniform_0_1() * 0.2 + 1.8;
        variance[i] = random_normal_mean_sigma(0.50,0.20);
        rate[i] = random_normal_mean_sigma(0.05,0.015);
    }
}
Cilk Plus tasking in BS code

Like OpenMP, works for both phases
Calculation loop again will be both, parallelized and vectorized, then
Very simple to add: Replace ‘for’ by ‘cilk_for’ – that’s all!

<table>
<thead>
<tr>
<th></th>
<th>initialization</th>
<th>calculation</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICL, vectorized</td>
<td>913</td>
<td>77</td>
<td>990</td>
</tr>
<tr>
<td>ICL, vectorized + OpenMP threading</td>
<td>330</td>
<td>31</td>
<td>361</td>
</tr>
<tr>
<td>ICL, vectorized + Cilk tasking</td>
<td>318</td>
<td>34</td>
<td>352</td>
</tr>
</tbody>
</table>

Cilk Plus tasking doing as well as OpenMP!
General workflow optimization

Step-by-step optimization workflow ...

Baseline
Modification of publicly available (sometimes “naïve”) code – compare non-Intel and Intel compiler

Optimized/Vectorized
Baseline code modifications for effective use of SSE/AVX by Intel Compiler; alternative Array Section Notation of Intel Cilk Plus

Threaded
Auto parallelization with Intel Compiler, OpenMP*, TBB or Cilk threading of Intel Compilers

Intel® Performance Library Enabled
Code modifications to enable e.g. Intel® MKL features like VSL – Vector Statistical Library or IPP multi-media routines
Intel® Math Kernel Library

Part of Intel® Composer XE (both, C/C++ and Fortran version)

Intel’s engineering, scientific, and financial math library

- Linear equation Solvers (BLAS, LAPACK)
- Eigenvector/eigenvalue solvers (BLAS, LAPACK)
- Some quantum chemistry needs (dgemm)
- PDEs, signal processing, seismic, solid-state physics (FFTs)
- General scientific, financial
  - vector transcendental functions (VML)
  - vector random number generators (VSL)
- Sparse Solvers (PARDISO - DSS and ISS)

Multi-thread via OpenMP*

Tuned for Intel® processors – current and future

Intel® AVX optimizations

- All BLAS level 3 functions, LU/Cholesky/QR & eigensolvers in LAPACK, FFTs of lengths 2^n, Mixed Radix FFTs (3, 5, 7), VML/VSL
Intel® MKL: Vector Statistical Library (VSL)

Functions for:
- Generating vectors of pseudorandom and quasi-random numbers
- Convolution & Correlation

Excellent multi-core scaling

Parallel computation support – some functions
User can supply own BRNG or transformations

<table>
<thead>
<tr>
<th>Performance Comparison of Random Number Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Processor</td>
</tr>
<tr>
<td>Standard C rand() function</td>
</tr>
<tr>
<td>Intel® MKL VSL random number generator</td>
</tr>
<tr>
<td>Intel® MKL + OpenMP version (12 threads)</td>
</tr>
</tbody>
</table>

Distribution Generators

<table>
<thead>
<tr>
<th>Continuous</th>
<th>Discrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform, Gaussian (two methods), Exponential, Laplace, Weibull, Cauchy, Rayleigh, Lognormal, Gumbel, Gamma, Beta</td>
<td>Uniform, UniformBits, Bernoulli, Geometric, Binomial, Hypergeometric, Poission, PoissonV, NegBinomial</td>
</tr>
</tbody>
</table>

Basic RNGs

<table>
<thead>
<tr>
<th>Pseudo RNGs</th>
<th>Quasi RNGs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCG31, GFSR250, MRG32, MCG59, WH, MT19937, MT2203</td>
<td>Sobol-quasi, Niederreiter quasi</td>
</tr>
</tbody>
</table>

Intel® Xeon® processor 2X6 Cores, 3.3 GHz
12MB L2 cache, 18 GB memory
Fedora 9 X86_64, Intel® MKL 10.3
MKL VSL in BS code

For initialization phase: Replace slow rand() calls

Multiple “streams” for OMP-threading

Fully thread save

- While rand() is thread-safe too on latest Windows*/Linux* release, the seed() initialization call makes e.g. OpenMP parallelism tricky to implement!

We use single-threaded MKL routines to avoid over-subscription!

<table>
<thead>
<tr>
<th></th>
<th>initialization</th>
<th>calculation</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICL, vectorized</td>
<td>913</td>
<td>77</td>
<td>990</td>
</tr>
<tr>
<td>ICL, vectorized + OpenMP threading</td>
<td>330</td>
<td>31</td>
<td>361</td>
</tr>
<tr>
<td>ICL, vectorized + OpenMP + MKL VSL</td>
<td>57</td>
<td>32</td>
<td>89</td>
</tr>
</tbody>
</table>

Most productive optimization step for initialization phase!!
Potential further tuning...

All FP data as single precision!

- Vectorization would show even larger benefit: 8 versus 4 elements in each vector!!

Use LA (lower accuracy) mode erf() call from MKL VML component for ‘calculation’

- Not done here because semantic would be changed
- But frequently, financial simulation codes can deal fine with lower precision

Use /Qprec-div- for faster division

Exploit hyper-threading: Very likely no benefit

Use EBS (event based sampling) of Intel® Vtune™ Amplifier XE for u-architectural tuning
Intel® MKL: Vector Math Library (VML)

Highly optimized implementations of computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.)

Operates on a vector unlike libm

Multiple accuracy modes
- High accuracy (HA) ~53 bits accurate
- Lower accuracy (LA), faster ~51 bits accurate
- Enhanced Performance (EP) ~26 bits accurate
- Routine-level mode controls
- New VML overflow reporting feature
- Denormal paths speedup via VML FTZ/DAZ setting

Special value handling $\sqrt{-a}$, $\sin(0)$, and so on

Can improve performance of non-linear programming and integrals computations applications.
BS computation using array sections

“Explicit vector code”, alternative to auto-/guided-vectorization
  • Supported by Intel® C/C++ Compiler since multiple years

Array Section Notation

<array base> [ <lower bound>:<length> [:<stride>] ]
[ <lower bound>:<length> [:<stride>] ] ...

Note that length is chosen: Not ‘upper bound’ as in Fortran [lower bound : upper bound]

A[:] // All elements of vector A
C[:][5] // Column 5 of matrix C
D[0:3:2] // Elements 0,2,4 of vector D
E[0:3][0:4] // 12 elements from E[0][0] to E[2][3]
Function maps with array sections
“Elemental functions”

Compiler can convert a user-supplied scalar function to vector function, when called with array notation arguments

```c
// Plain C scalar function declared with __declspec(vector)
__declspec(vector) float saxpy (float a, float x, float y)
{
    return (a * x + y);
}

Z[:]= saxpy(A, X[:], Y[:]); // Call scalar function with 
// array notation parameters
```

Compiler automatically maps the function across multiple array elements (in example, the function becomes “a * x[:] + y[:]”)

Built-in support for vectorized math functions
Array sections in BS code
A few sample lines

```c
void create_input_data(double S[NUM_OPTIONS], ...) {
    srand(SEED);
    S[:] = (double)rand()/(double)RAND_MAX*(SH-SL) + SL;
    K[:] = (double)rand()/(double)RAND_MAX*(KH-KL) + KL;
    time[:] = (double)rand()/(double)RAND_MAX*(TH-TL) + TL;
    call_Serial[:] = 0.0;
    call_ArrayNotations[:] = 0.0;
    call_CilkPlus[:] = 0.0;
}
```
## Timing summary

<table>
<thead>
<tr>
<th></th>
<th>initialization</th>
<th>calculation</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS VS 2013 - CL, base line</td>
<td>1055</td>
<td>403</td>
<td>1458</td>
</tr>
<tr>
<td>ICL, base line</td>
<td>942</td>
<td>409</td>
<td>1351</td>
</tr>
<tr>
<td>ICL, vectorized</td>
<td>913</td>
<td>77</td>
<td>990</td>
</tr>
<tr>
<td>ICL, vectorized + OpenMP threading</td>
<td>330</td>
<td>31</td>
<td>361</td>
</tr>
<tr>
<td>ICL, vectorized + Cilk tasking</td>
<td>318</td>
<td>34</td>
<td>352</td>
</tr>
<tr>
<td>ICL, vectorized + OpenMP + MKL VSL</td>
<td>57</td>
<td>32</td>
<td>89</td>
</tr>
</tbody>
</table>

Source code freely available: Please contact presenter in case you want to have the source code and build scripts to reproduce the measurements.
BS on Intel® Xeon® and Xeon Phi™ architectures

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

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

Performance data from “High Performance Parallelism Peals, Vol1, Chapter 19: Performance optimization of Black-Scholes pricing”, by Iosif Meyerov, Et Al.
Summary

Intel® Parallel Studio XE provides complete tool box for whole program development cycle

Black-Scholes sample code optimized by a factor of 16+ by systematic use of Intel developer tools

Code easy to parallelize too using new programming models like Intel® Cilk Plus including Array Notation and OpenMP* data parallel extensions

Xeon Phi results confirm significant performance improvement using multi-threaded paradigms.