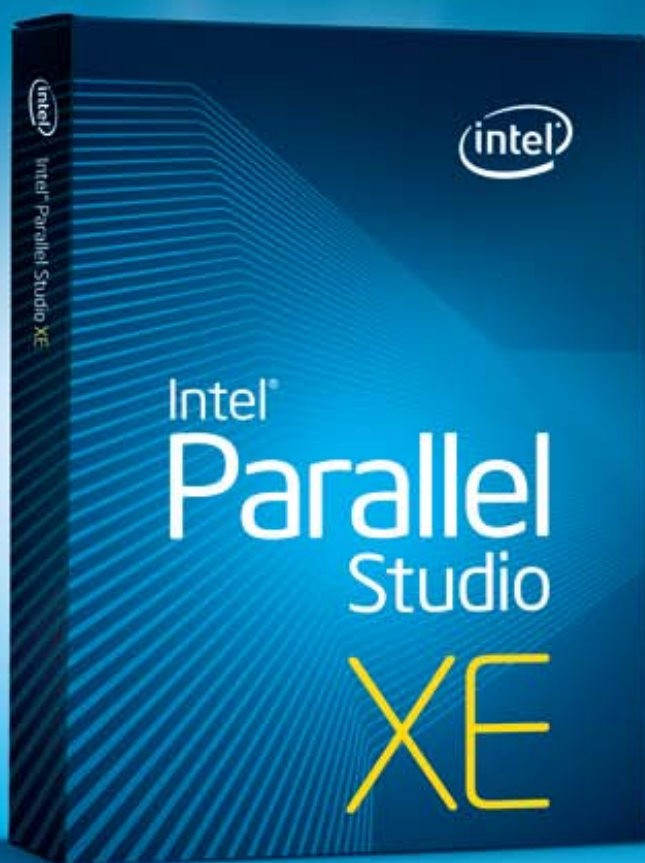




Intel® Cilk™ Plus: A Simple Path to Parallelism





Compiler extensions to simplify task and data parallelism

Intel Cilk Plus adds simple language extensions to express data and task parallelism to the C and C++ language implemented by the Intel® C++ Compiler, which is part of Intel® Parallel Composer and Intel® Parallel Studio, as well as Intel® Composer XE and Intel® Parallel Studio XE. These language extensions are powerful, yet easy to apply and use in a wide range of applications.

Intel Cilk Plus includes the following features and benefits:

Feature	Benefit
Simple keywords	Simple, powerful expression of task parallelism: <i>cilk_for</i> – Parallelize for loops <i>cilk_spawn</i> – Specify the start of parallel execution <i>cilk_sync</i> – Specify the end of parallel execution
Hyper-objects (Reducers)	Eliminates contention for shared reduction variables amongst tasks by automatically creating views of them for each task and reducing them back to a shared value after task completion
Array Notation	Data parallelism for whole arrays or sections of arrays and operations thereon
Elemental Functions	Enables data parallelism of whole functions or operations which can then be applied to whole or parts of arrays

When to use Intel Cilk Plus over other Parallel Methods?

Use Intel Cilk Plus when you want the following

- simple expression of opportunities for parallelism, rather than control of execution to perform operations on arrays
- higher performance obtainable with inherent data parallelism semantics – array notation
- to use native programming, as opposed to managed deployment: no managed runtime libraries – you express the intent
- to mix parallel and serial operations on the same data

Intel Cilk Plus involves the compiler in optimizing and managing parallelism. The benefits include:

- code is easier to write and comprehend because it is better integrated into the language through the use of keywords and intuitive syntax
- the compiler implements the language semantics, checks for consistent use and reports programming errors
- integration with the compiler infrastructure allows many existing compiler optimizations to apply to the parallel code. The compiler understands these four parts of Intel Cilk Plus, and is therefore able to help with compile time diagnostics, optimizations and runtime error checking.

Intel Cilk Plus has an [open specification](#) so other compilers may also implement these exciting new C/C++ language features.

How is the Performance and Scaling?

Here are two real application examples. One is a Monte Carlo simulation utilizing Intel Cilk Plus, where the array notation allows the compiler to vectorize, that is, utilize Intel® Streaming SIMD Extensions (Intel® SSE) to maximize data-parallel performance, while adding the `cilk_for` causes the driver function of the simulation to be parallelized, maximizing use of the multiple processor cores for task-level parallelism. The second is a Black-Scholes algorithm using the elemental functions of Intel Cilk Plus, where a scalar function gets vectorized to utilize Intel SSE, and `cilk_for` parallelizes the vectorized code over multiple cores. Both applications get a dramatic speedup with very little effort!

	Scalar Code	Intel Cilk Plus Data-Parallel	Intel Cilk Plus Task-Parallel
Monte Carlo Simulation ¹	5.8 seconds	2.8 seconds = 1.9x speedup	0.50 seconds = 11.6x speedup (16 virtual cores)
Black-Scholes ²	2.1 seconds	1.1 seconds = 1.9 x speedup	0.14 seconds = 15x speedup

Compiler: Intel Parallel Composer 2011 with Microsoft* Visual Studio* 2008

¹System Specifications: Intel® Xeon® W5580 processor, 3.2 GHz, 8 cores, hyper-threading enabled, 6 GB RAM, Microsoft* Windows* XP Professional x64 Version 2003, service pack 2.

²System Specifications: Intel Xeon E5462 processor, 2.8 GHz, 8 cores, 8 GB RAM, Microsoft Windows Server 2003 x64 Edition.

Try It Yourself

This guide will help you begin adding Intel Cilk Plus to your application using Intel Parallel Studio. It will show you the following examples:

- a simple quick-sort implementation with the `cilk_spawn` and `cilk_sync` keywords
- Black-Scholes application demonstrating elemental functions with the `cilk_for` keyword
- Monte Carlo Simulation to show both the array notation syntax and the `cilk_for` keywords

Topic
Install Intel Parallel Studio
Get the Sample Applications
Implement Parallelism with Intel Cilk Plus Keywords
Performance Through Elemental Functions
Putting It All Together – Performance & Parallelism
Summary
Additional Resources

Install Intel Parallel Studio 2011

Install and set up Intel Parallel Studio:

1. [Download](#) an evaluation copy of Intel Parallel Studio 2011
2. Install Intel Parallel Studio by clicking on the `parallel_studio_setup.exe` (can take about 15-30 minutes depending on your system).

Note: The examples in this guide use Microsoft® Visual Studio® on Windows®, but Intel Cilk Plus is also available for Linux® in the Intel® C++ Compiler in Intel Composer XE for Linux®.

Get the Sample Applications

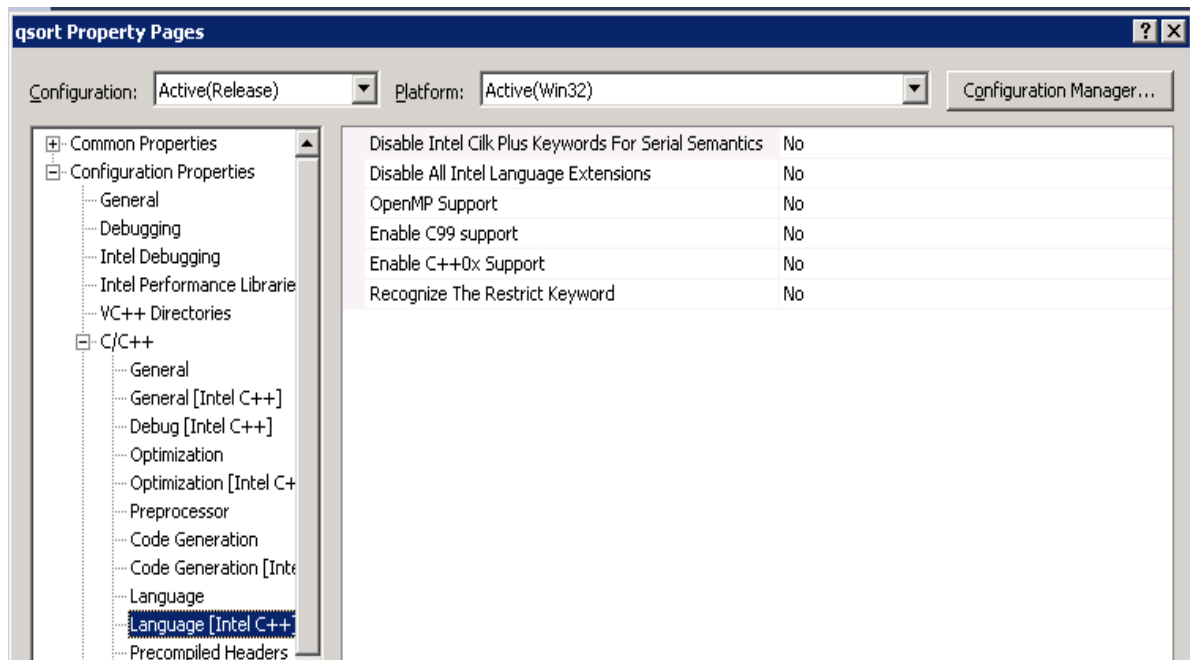
Install the sample applications:

1. Download the "[BlackScholesSample.zip](#)" sample file to your local machine. This sample shows how the Intel C++ Compiler can vectorize, or generate a single instruction multiple data (SIMD) parallel version of, the key function. Then it shows how to parallelize this vectorized function using `cilk_for`.
2. Download the "[MonteCarloSample.zip](#)" sample file to your local machine. This sample shows a serial/scalar kernel for the Monte Carlo simulation, an Intel Cilk Plus `cilk_for` version of the driver function, and an Intel Cilk Plus array notation version of the kernel.
3. Find the Intel Cilk Plus Samples, `Cilk.zip`, in the installation directory of Intel Parallel Composer. In a typical installation, this is `c:\Program Files\Intel\Parallel Studio 2011\Composer\Samples\en_US\C++\Cilk.zip`.
4. Extract the files from each zip file to a writable directory or share on your system, such as a `My Documents\Visual Studio 200x\Intel\samples` folder.
5. After extracting all of the samples, you should see the following directories:
 - Cilk – with multiple sample directories underneath it. We will use `qsort`.
 - BlackScholesSample
 - MonteCarloSample

Build the Samples:

Each sample has a Microsoft® Visual Studio® 2005 solution file (`.sln`) that can be used with Visual Studio 2005, 2008 and 2010.

- 1) Build the solutions with Intel Parallel Composer 2011, using the Intel C++ Compiler in the **Release** (optimized) configuration settings.
- 2) In each of these solution files, the Intel Cilk Plus language extensions are enabled.
 - a) Right-click the solution, select **Properties** and expand **the Configuration Properties > C/C++ > Language [Intel C++]**.
 - b) Set the **Disable Intel Cilk Plus Keywords for Serial Semantics** to **No**.
 - c) Set the **Disable All Intel Language Extensions** to **No**. Following are the Configuration Properties for `qsort`.



- 3) Run the applications from within Microsoft Visual Studio. Go to **Debug > Start Without Debugging**

Implement Parallelism with the Intel Cilk Plus Keywords

Now, we are going to quickly add task parallelism using the keywords *cilk_spawn* and *cilk_sync*.

1. Load the qsort solution into Microsoft Visual Studio.
2. Open the qsort.cpp file and look at sample_qsort() routine:

```
void sample_qsort(int * begin, int * end)
{
    if (begin != end) {
        --end; // Exclude last element (pivot) from partition
        int * middle = std::partition(begin, end,
                                     std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle); // move pivot to middle
        cilk_spawn sample_qsort(begin, middle);
        sample_qsort(++middle, ++end); // Exclude pivot and restore end
        cilk_sync;
    }
}
```

Look at the use of the *cilk_spawn* and *cilk_sync*. To parallelize the quick-sort algorithm, the *cilk_spawn* will take care of the task creation and scheduling of tasks to threads for you, while the *cilk_sync* indicates the end of the parallel region at which point the tasks complete

and serial execution resumes. In this example, the two calls to `sample_qsort()` between the `cilk_spawn` and `cilk_sync` can be executed in parallel, depending on the resources available to the Intel Cilk Plus runtime.

At the time of writing, the Intel Cilk Plus keywords are specific to the Intel C++ Compiler, though we have published the specification to encourage other tools vendors to adopt it (see [Additional Resources](#) below). A key property of the Intel Cilk Plus keywords is that even if you disable them, their serial semantics guarantees the application will still run correctly in serial mode – without modifying the code! Thus, you can easily check serial and parallel runtime performance and stability by disabling/enabling them in the **Property Pages** as seen in the previous section: **Disable Intel Cilk Plus Keywords for Serial Semantics = Yes** – to turn parallel execution off. In addition, if you want to compile files in which you added Intel Cilk Plus keywords and reducer hyper-objects using another compiler that does not support them, add the following code to the top of the file in which the keywords are used:

```
#ifndef __cilk
#include <cilk/cilk_stub.h>
#endif
#include <cilk/cilk.h>
```

The `cilk_stub.h` header file will essentially comment out the keywords so that other compilers will compile the files without any further source code changes. See the “Using Intel Cilk Plus” section of the Intel C++ Compiler 12.0 User and Reference Guide and other samples in the Intel Cilk Plus sample directory, and other Intel Parallel Studio samples, to learn about reducer hyper-objects. These are simple, powerful objects used to protect shared variables among Intel Cilk Plus tasks, without the need for barriers or synchronization code.

Performance with Intel Cilk Plus Elemental Functions and `cilk_for`

In this sample, we will show an implementation of Intel Cilk Plus elemental functions and demonstrate the special capability of the Intel Compiler to vectorize function calls.

In order to write an Elemental function, the code has to indicate to the compiler that a vector version needs to be generated. This is done by using the `__declspec(vector)` syntax as shown below:

```
__declspec(vector) float saxpy_elemental(float a, float x, float y)
{
    return(a * x + y);
}

Here is how to invoke such vectorized elemental functions using array
notation:

z[:] = saxpy_elemental(a, b[:], c[:]);
```

The compiler provides a vector implementation of the standard functions in the math library. You do not have to declare them in any special way. The compiler can identify their use in an array notation call site and invoke the provided vector version, as shown below:

```
a[:] = sin(b[:]);           // vector math library function
a[:] = pow(b[:], c);       // b[:]**c via a vector math library function
```

(Many routines in the Intel math libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors).

After this brief introduction, let's take a look at the Black-Scholes example using elemental functions.

1. Load the CilkPlus-BlackScholesSample solution into Microsoft Visual Studio.
2. First, open the BlackScholes-Serial.cpp file that contains the original or serial version of the code that is not vectorized

The routine `test_option_price_call_and_put_black_scholes_Serial()` invokes the call and put option calculations using the original or serial version of the routines:

```
double option_price_call_black_scholes(...) {...}
double option_price_put_black_scholes(...) {...}

void test_option_price_call_and_put_black_scholes_Serial(
    double S[NUM_OPTIONS], double K[NUM_OPTIONS],
    double r, double sigma, double time[NUM_OPTIONS],
    double call_Serial[NUM_OPTIONS], double put_Serial[NUM_OPTIONS])
{
    for (int i=0; i < NUM_OPTIONS; i++) {
        // invoke calculations for call-options
        call_Serial[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);

        // invoke calculations for put-options
        put_Serial[i] = option_price_put_black_scholes(S[i], K[i], r, sigma, time[i]);
    }
}
```

Now open the BlackScholes-ArrayNotationsOnly.cpp file and look at the Intel Cilk Plus version using elemental functions with array notation:

```
__declspec(vector) double option_price_call_black_scholes(...) {...}
__declspec(vector) double option_price_put_black_scholes(...) {...}

void test_option_price_call_and_put_black_scholes_ArrayNotations(
    double S[NUM_OPTIONS], double K[NUM_OPTIONS],
    double r, double sigma, double time[NUM_OPTIONS],
    double call_ArrayNotations[NUM_OPTIONS], double put_ArrayNotations[NUM_OPTIONS])
{
    // invoking vectorized version of the call functions optimized for given core's SIMD
    call_ArrayNotations[:] = option_price_call_black_scholes_ArrayNotations(S[:], K[:], r, sigma, time[:]);

    // invoking vectorized version of the put functions optimized for given core's SIMD
    put_ArrayNotations[:] = option_price_put_black_scholes_ArrayNotations(S[:], K[:], r, sigma, time[:]);
}
```

The `__declspec(vector)` is added to the original serial code of call and put option calculations to tell the compiler to generate a vectorized version of these functions. These vectorized functions are then invoked using the array notation syntax where the original for-loop is now replaced with matching array sections. The array section syntax, `call_ArrayNotations[:]`, `put_ArrayNotations[:]` and similar syntax for the function arguments indicates that the operations should be applied to each element of the arrays. The compiler calls the vector version of the function, that is, the elemental function, and it figures out the appropriate loop bounds for the vectorized loop automatically.

Note that the original algorithm has not changed at all! This simple change gave the code close to a 2x speedup on this sequential code running on a single core of the system described in the second footnote to the table on page 2. The Intel Cilk Plus elemental functions offer great simplicity by enabling you to express to the compiler which functions should be tuned for data parallelism.

Vectorization is enabled at default optimization levels for both Intel microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel microprocessors compared to non-Intel microprocessors. The vectorization can also be affected by certain compiler options, such as `/arch` or `/Qx` (Windows) or `-m` or `-x` (Linux and Mac OS X).

3. We will now take this vectorized sequential code and introduce threading to parallelize it and take advantage of multi-core systems. Open the `BlackScholes-CilkPlus.cpp` file that demonstrates how easy it is to parallelize the code using just one keyword `cilk_for` (that replaces `for` in the original code):

```
void test_option_price_call_and_put_black_scholes_CilkPlus(
    double S[NUM_OPTIONS], double K[NUM_OPTIONS],
    double r, double sigma, double time[NUM_OPTIONS],
    double call_CilkPlus[NUM_OPTIONS], double put_CilkPlus[NUM_OPTIONS])
{
    // invoking vectorized version of the call and put functions optimized for given core's SIMD
    // and then spreading it across multiple cores using cilk_for for optimized multi-core performance
    cilk_for (int i=0; i<NUM_OPTIONS; i++) {
        call_CilkPlus[i] = option_price_call_black_scholes_ArrayNotations(S[i], K[i], r, sigma, time[i]);
        put_CilkPlus[i] = option_price_put_black_scholes_ArrayNotations(S[i], K[i], r, sigma, time[i]);
    }
}
```

This simple, yet powerful, change automatically parallelizes the loop and executes the sequentially optimized code of the previous step on multiple cores. Note that the Intel Cilk Plus runtime determines the number of cores available on the system and distributes the load accordingly. In our example, we saw a further speedup of close to 8x on the 8-core system described in the table.

Putting It All Together – Performance and Parallelism

Now we will look at a more complex example – a financial Monte Carlo Simulation. This example combines the array notation and the keywords of Intel Cilk Plus to give you both parallelization of the main driver loop using `cilk_for`, and the array notation for the simulation kernel to allow vectorization.

Array notation provides a way to operate on slices of arrays using a syntax the compiler understands and subsequently optimizes, vectorizes, and in some cases parallelizes.

This is the basic syntax:

```
[<lower bound> : <length> : <stride>]
```

where the *<lower bound>*, *<length>*, and *<stride>* are optional, and have integer types. The array declarations themselves are unchanged from C and C++ array-definition syntax. Here are some example array operations and assignments:

Operations:

```
a[:] * b[:] // element-wise multiplication
a[3:2][3:2] + b[5:2][5:2] // matrix addition of 2x2 subarrays within a and b starting at
a[3][3] and b[5][5]
a[0:4][1:2] + b[0][1] // adds a scalar b[0][1] to each element of the array section in a
```

Assignments:

```
a[:, :] = b[:, 2][:] + c;
e[:] = d; // scalar variable d is broadcast to all elements of array e
```

Now, take a look at the MonteCarloSample application.

1. Load the MonteCarloSample solution into Microsoft Visual Studio
2. Open the file, mc01.c and look at the function, `Pathcalc_Portfolio_Scalar_Kernel`. In the scalar kernel function, the scalar array declarations are as follows:

```
__declspec(align(64)) float B[nmat], S[nmat], L[n];
```

In order to utilize array operations, we are going to change the operation of the function, and work on “stride” elements, instead of working on a single element. We start by changing B, S and L from single dimensional to two dimensional arrays:

```
__declspec(align(64)) float B[nmat][vlen], S[nmat][vlen], L[n][vlen];
```

where we have simply specified the size (`nmat` or `n`) and length (`vlen`). We also have to declare some other arrays in the new function, `Pathcalc_Portfolio_Array_Kernel()` to handle the many scalar accumulations and assignments in the computation loops within the kernel.

The resulting code is very similar to the scalar version except for the array section specifiers. For example, here is a comparison of one of the loops in the kernel:

Scalar version:

```
for (j=nmat; j<n; j++) {
    b = b/(1.0+delta*L[j]);
    s = s + delta*b;
    B[j-nmat] = b;
    S[j-nmat] = s;
}
```

Array notation version:

```
for (j=nmat; j<n; j++) {
    b[:] = b[:] / (1.0 + delta * L[j][:]);
    s[:] = s[:] + delta * b[:];
    B[j-nmat][:] = b[:];
    S[j-nmat][:] = s[:];
}
```

With some straightforward changes to the code, we have implemented array operations that work on multiple elements at a time, allowing the compiler to use SIMD code and gain a substantial speedup over the serial, scalar kernel implementation.

- Now, let's parallelize the calling loop using *cilk_for*. Take a look at the function, `Pathcalc_Portfolio_Scalar()`. All we need to do is replace the *for* with *cilk_for*. This is done for you in the `Pathcalc_Portfolio_Cilk()` function:

```
void Pathcalc_Portfolio_CilkArray(FPPREC *restrict z,
                                  FPPREC *restrict v,
                                  FPPREC *restrict L0,
                                  FPPREC * restrict lambda)
{
    int stride = SIMDVLEN, path;
    DWORD startTime = timeGetTime();

    cilk_for (path=0; path<npath; path+=stride) {
        Pathcalc_Portfolio_Array_Kernel(stride,
                                        L0,
                                        &z[path*nmat],
                                        lambda,
                                        &v[path]);
    }

    perf_cilk_array = timeGetTime()-startTime;
}
```

On the 8 core system with hyper-threading described in the table on page 2, adding array notation to enable SIMD data parallelism increased performance by close to a factor of two compared to the scalar version. Adding the *cilk_for* keyword to parallelize the calls to the kernel showed good scaling, and resulted in a further speedup of close to a factor of eight on the same system. This is an illustration of the power and simplicity of Intel Cilk Plus.

Summary

Parallelism in a C or C++ application can be simply implemented using the Intel Cilk Plus keywords, reducer hyper-objects, array notation and elemental functions. It allows you to take full advantage of both the SIMD vector capabilities of your processor and the multiple cores, while reducing the effort needed to develop and maintain your parallel code.









Additional Resources

Please visit the [user forum](#) for Intel Cilk Plus.

See the Intel Cilk Plus documentation for all of the details about the syntax and semantics:

- [Intel Parallel Studio Documentation](#), especially the “Using Intel Cilk Plus” section under “Creating Parallel Applications” in the Intel C++ Compiler 12.0 User and Reference Guide.
- Intel Cilk Plus Tutorial – installed with Intel Parallel Composer and available online along with the Intel Parallel Studio Documentation listed above.

Finally, take a look at our [open specification](#) for Intel Cilk Plus and feel free to comment about it at the email link provided, or on the Cilk Plus user forum listed above.

Related Links	
Intel® Software Network Forums	
Intel® Software Products Knowledge Base	
Intel Software Network Blogs	
Intel Parallel Studio Website	
Intel® Threading Building Blocks Website	
Parallelism blogs, papers and videos	



Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101