Getting Started with Intel® Cilk™ Plus Array Notations

Introduction

Array Notations is an Intel-specific language extension that is a part of Intel® Cilk™ Plus feature supported by the Intel® C++ Compiler that provides ways to express data parallel operation on ordinary declared C/C++ arrays. By using array notations, you can improve the performance of your application through Vectorization. Vectorization is the key to improving your applications' performance through taking advantage of the processor's capability to operate on multiple array (or vector) elements at a time. The Intel® Compilers provide unique capabilities to enable vectorization. The programmer may be able to help the compiler to vectorize more loops through a simple programming style and by the use of compiler features designed to assist vectorization. This article discusses how to use the Array Notations feature from the Intel® Cilk™ Plus, to help the compiler to vectorize C/C++ code and improve performance.

Document Organization

To demonstrate the usage and performance characteristics of Array Notations, you will:

- Establish a performance baseline by building and running the scalar (non-vectorized) version of a loop.
- Improve application performance by vectorizing the loop using Array Notations
- Improve application performance by creating a short vector form of the loop using a constant trip-count

System Requirements

To compile and run the example and exercises in this document you will need Intel® C++ Composer XE 2011 Update 9 or higher, and an Intel® Pentium 4 Processor or higher with support for Intel® SSE2 or higher instruction extensions. **The exercises in this document were tested on a first generation Intel® Core i5 system supporting 128-bit vector registers.** The instructions in this tutorial show you how to build and run the example with the Microsoft Visual Studio* 2008. The example provided can also be built from the command line on Windows*, Linux*, and Mac OS* X using the following command line options:

```
Windows: icl /Qvec-report2 ArrayNotation.cpp main.cpp

Linux* and Mac OS* X: icc –vec-report2 ArrayNotation.cpp main.cpp
```

Array Notations

Array notations is an Intel-specific language extension that is a part of Intel® Cilk™ Plus feature supported by Intel® C++ Compiler. It provides ways to express data-parallel operations on ordinary declared C/C++ arrays.

Array notations has the following benefits:

- Provides ways to express data parallel operations on ordinary declared C/C++ arrays.
- Enables compiler parallelizing and vectorizing a program and exploiting the data parallelism with minimal and simple changes in the code.
- Resolves any possible data dependency ambiguity that the compiler might get into when encountering normal loops.
- Maps parallel constructs to the underlying SIMD hardware to achieve predictable performance.

  Array notations can be used when operations that involve arrays do not require a specific order among the elements of the array(s). By default the compiler generates Single Instruction Multiple Data (SIMD) vector instructions for SSE2 instruction set. To target other specific vector instructions you can use the /Qx or /Qax compiler options.

Usage of array notations for an array such as int a[10] is as follows:

a[0:10] – Full range of the array from index 0 (lower bound) to index 9 (length). Same as a[:] which is a shorthand for the entire array.

For example to multiply arrays "a" and "b" and store the product in array "c", we can use the following statement:

```
c[:] = a[:] * b[:];
```

a[0:5:2] – Selects the following a[0], a[2], a[4], a[6], a[8] since we mention an extra third parameter "2" (stride) which is the factor by which to increment the index. We stop after index 8 because we have accrued 5 elements as specified in the second parameter in the notation.

For more information on Array Notation Extensions please refer to **Key Features > Intel® Cilk™ Plus > Extensions for Array Notations** in "Intel® C++ Compiler XE 12.1 User and Reference Guides"

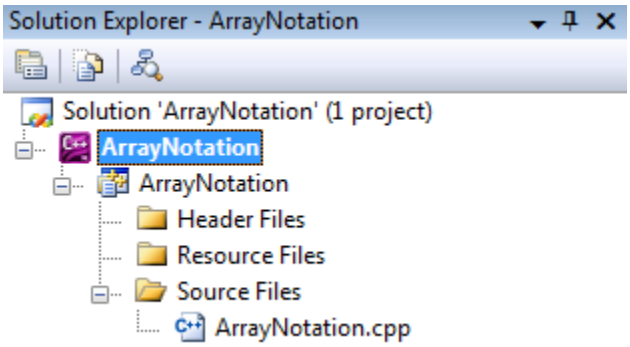
Locating the Samples

To begin this tutorial, open the ArrayNotation.zip archive attached:

Use the following files for this tutorial:

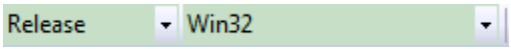
- ArrayNotation.sln
- ArrayNotation.vcproj
- ArrayNotation.cpp

1. Open the Microsoft Visual Studio* 2008 solution file, ArrayNotation.sln



and follow the steps below to prepare the project for the Array Notation exercises in this tutorial:

2. Select "Release" "Win32" configuration



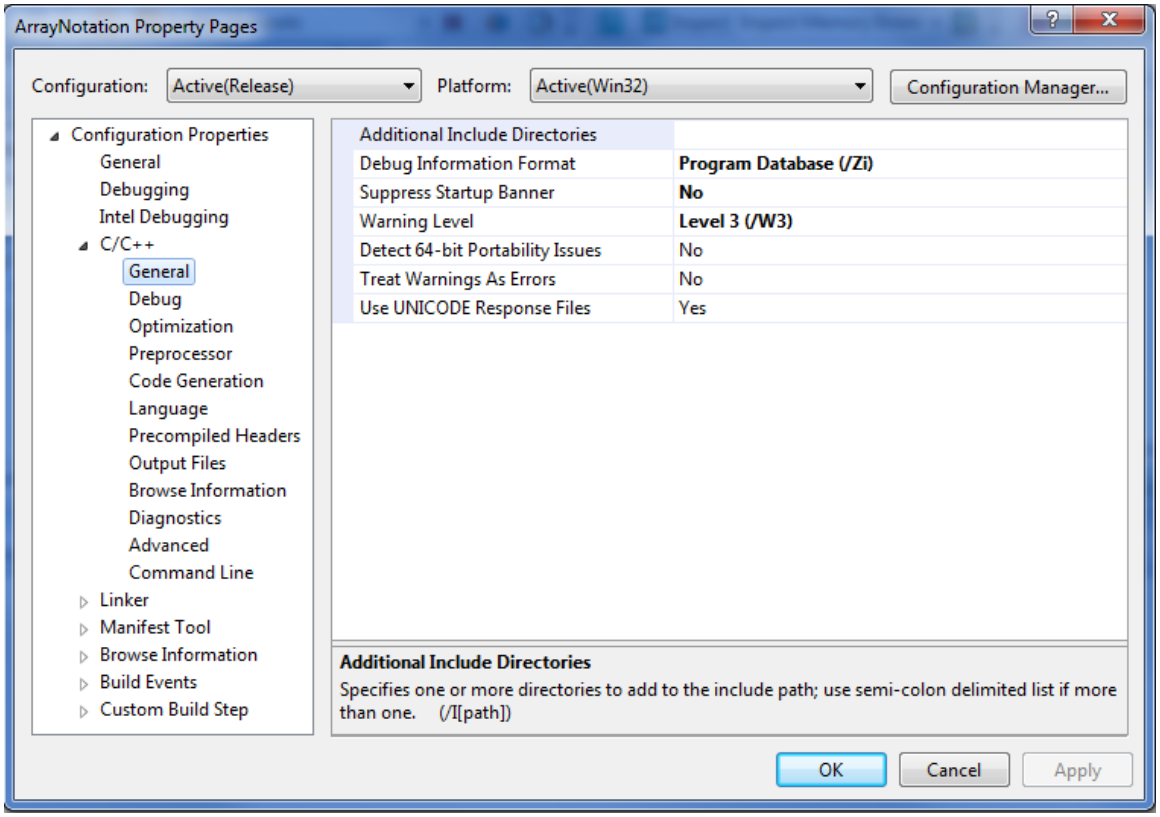
3. Clean the solution by selecting **Build > Clean Solution**.

You just deleted all of the compiled and temporary files association with the solution. Cleaning a solution ensures that the next build is a full build rather than changing existing files.

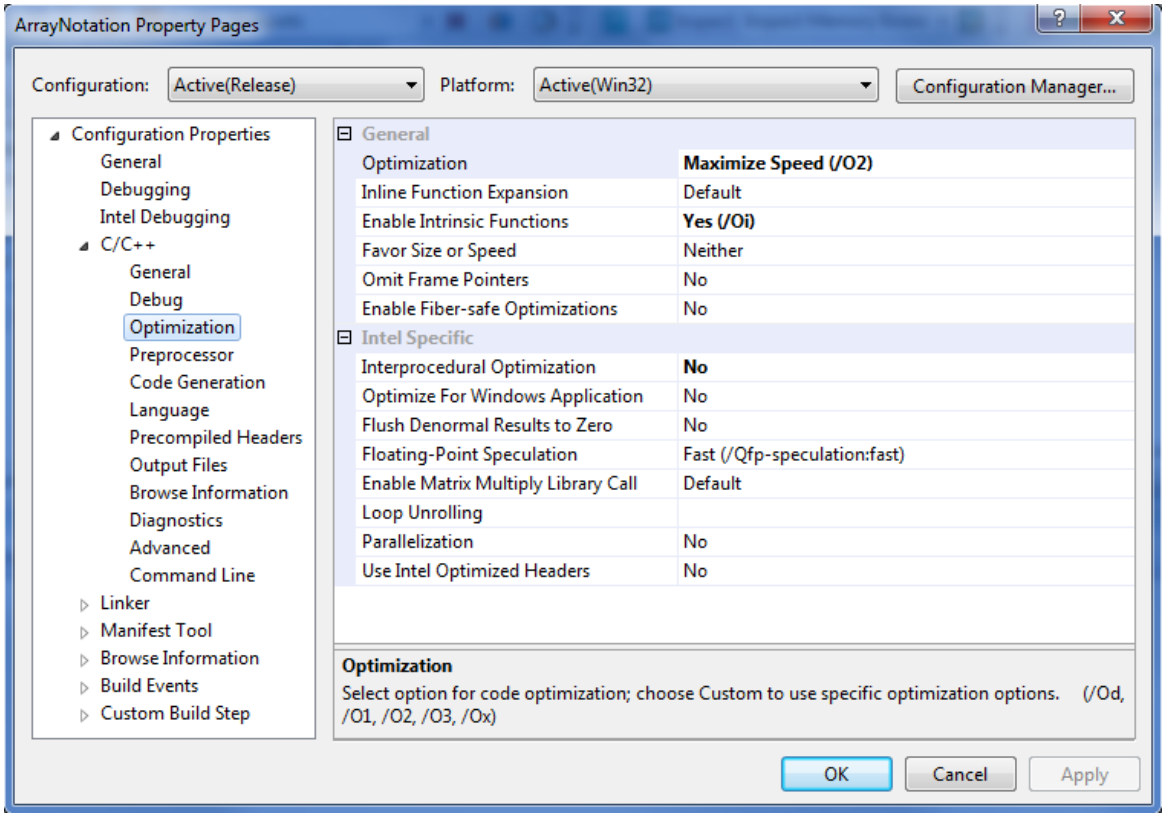
Establishing a Performance Baseline

To set a performance baseline for the improvements that follow in this tutorial, build the default scalar implementation with these settings:

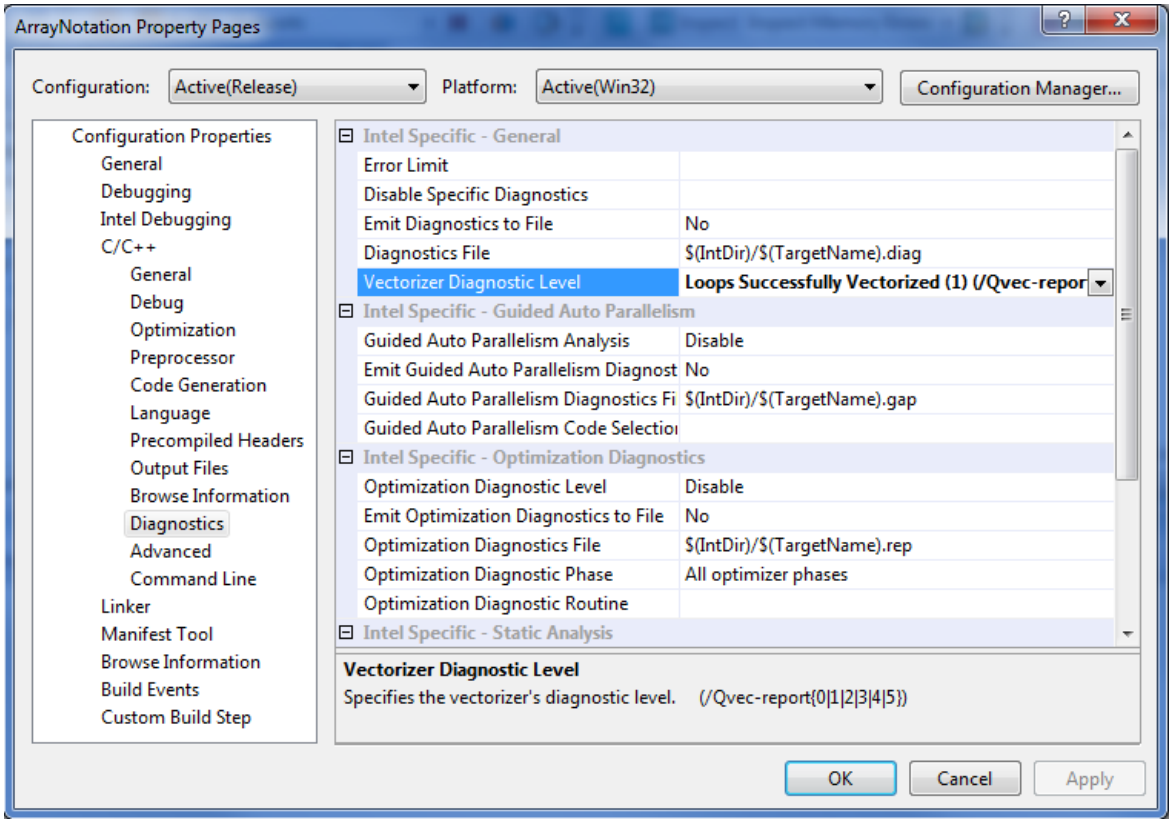
1. Select **Project > Properties > C/C++ > General > Suppress Startup Banner > No**.



2. Select **Project > Properties > C/C++ > Optimization > General > Optimization > Maximize Speed (/O2).**



3. Select **Project > Properties > Diagnostics > Vectorizer Diagnostic Level > Loops Successfully Vectorized (1) (/Qvec-report2)**



4. Rebuild the project, then run the executable (**Debug > Start Without Debugging**). Running the program results in starting a window that displays the program's execution time in seconds. Record the execution time reported in the output. This is the baseline against which subsequent improvements will be measured. Below is the vectorization report for the scalar implementation:

```
1> main.cpp(62): (col. 3) remark: LOOP WAS VECTORIZED.
1> main.cpp(65): (col. 15) remark: LOOP WAS VECTORIZED.
1> main.cpp(68): (col. 2) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.
1> ArrayNotation.cpp(38): (col. 2) remark: loop was not vectorized: existence of vector dependence.
```

The hotspot in this example is the loop at line number 38 that does not vectorize due to assumed vector dependency. Without Inter-Procedural analysis, the compiler does not know if there are memory overlaps among the arrays passed in as arguments to the scalar function. We can improve the application performance by vectorizing the above loop using Array Notations.

The vectorizer can generate faster code when operating on aligned data. The program uses the following syntax where ALIGNMENT=16 to request the compiler to align all arrays on a 16-byte boundary:

```
__declspec(align(ALIGNMENT)) T A[S], B[S], C[S];
```

When data is aligned, the vectorizer can use aligned load instructions for all arrays rather than the slower unaligned load instructions and can avoid runtime tests of alignment. To derive the maximum benefit from this alignment, we also need to tell the vectorizer it can safely assume that the arrays are aligned by using the following syntax:

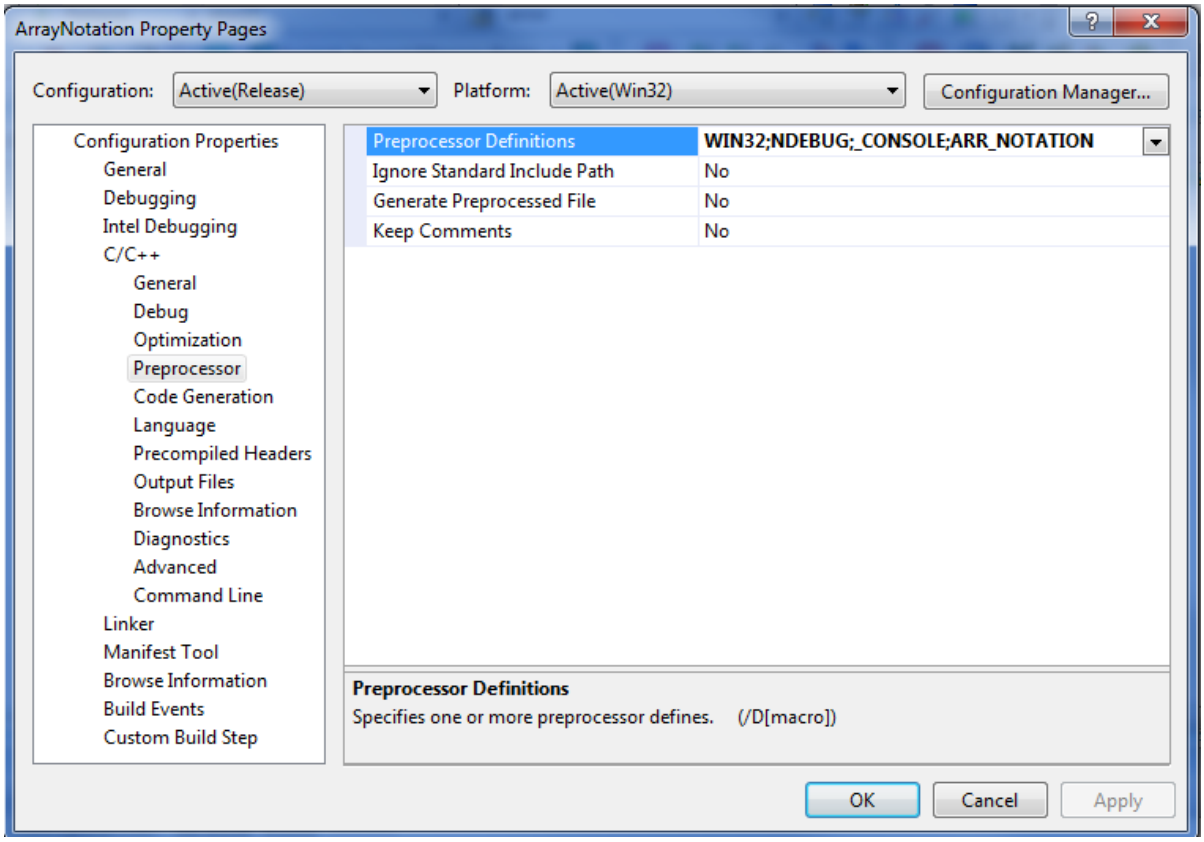
```
__assume_aligned(A,ALIGNMENT);
__assume_aligned(B,ALIGNMENT);
__assume_aligned(C,ALIGNMENT);
```

NOTE. While it is a good practice to align data for improved performance, it is not a requirement for vectorization or the use of array notations. Many unaligned loops vectorize just fine, and in many cases, alignment is not feasible – when you pass a section of an array, for example. If you use __assume_aligned, you must be sure that all the arrays or subarrays in the loop have the specified alignment (e.g. 16-byte aligned in this example). Otherwise, you may get a runtime error. If your compilation targets the **Intel® AVX** instruction set, you should try to align data on a 32-byte boundary. For the **Intel® MIC** architecture, you should try to align on 64-byte boundary. This may result in improved performance.

Improving Performance by Using Array Notations

Here we re-write the loop at line 71 using the array notations with the default vector length. On a CPU with 128-bit vector registers size the default vector length is 4 (e.g. loading four 32-bit float data elements into vector registers).

1. Select Project **> Properties > C/C++ > Preprocessor > Preprocessor Definitions**, and add a new macro "**ARR_NOTATION**".



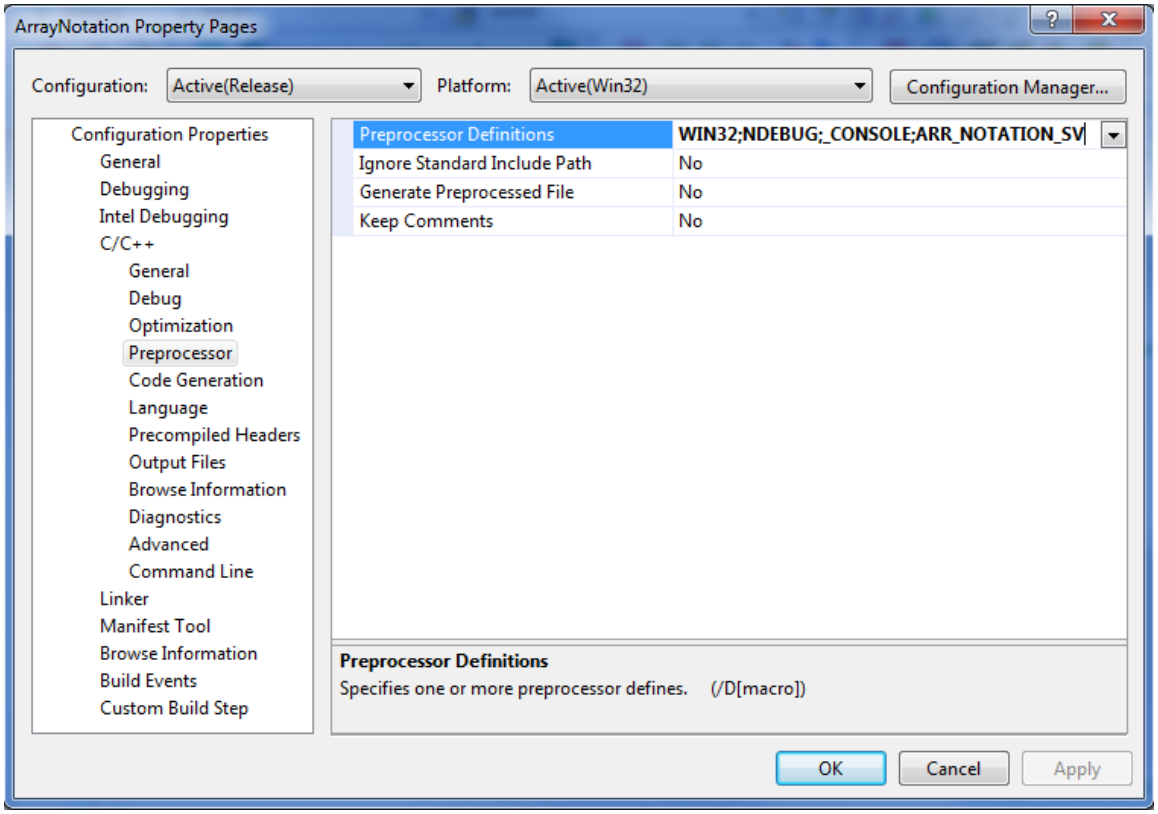
2. Rebuild the project, then run the executable (**Debug > Start Without Debugging**) and record the execution time reported in the output. The execution time for this implementation should be less than the time taken for the baseline (Scalar) version. This is because array notations version will make use of the SIMD registers and SIMD instruction set to handle operations on vector operands rather than normal scalar operations done in the baseline case. The vectorization report shows that the array notation version of the loop at line 56 got vectorized resulting in improved performance:

1> main.cpp(62): (col. 3) remark: LOOP WAS VECTORIZED.
1> main.cpp(65): (col. 15) remark: LOOP WAS VECTORIZED.
1> main.cpp(80): (col. 2) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.
1> ArrayNotation.cpp(56): (col. 5) remark: LOOP WAS VECTORIZED.

Improving Performance by Creating a Short Vector Form of the Loop Using a Constant Trip-count

You can improve the performance further by making the implicit inner loop in the previous step in this example a short constant trip-count loop and get the desired unrolling. In this exercise we will recompile the code with a TCOUNT=16 to see the performance effects.

1. Select **Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions**. Remove the macro named **ARR_NOTATION** and add a new macro "**ARR_NOTATION_SV**".



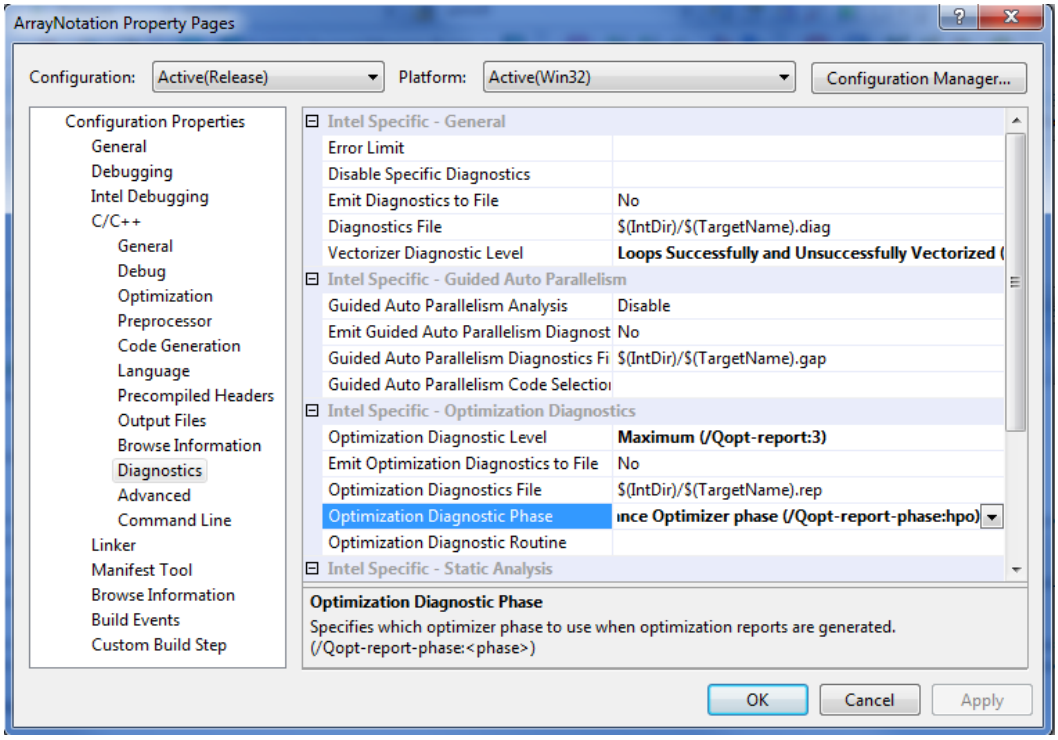
2. Rebuild the project, then run the executable (**Debug > Start Without Debugging**) and record the execution time reported in the output.

1> main.cpp(62): (col. 3) remark: LOOP WAS VECTORIZED.
1> main.cpp(65): (col. 15) remark: LOOP WAS VECTORIZED.
1> main.cpp(92): (col. 2) remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.
1> ArrayNotation.cpp(74): (col. 5) remark: LOOP WAS VECTORIZED.

1> ArrayNotation.cpp(73): (col. 3) remark: loop was not vectorized: not inner loop.

The execution time for this implementation should be less than the previous version. The reason is that in the previous case the compiler unrolled the loop by a factor of 2 while in this case the loop is unrolled by a factor of 4 due to TCOUNT=16. For a target CPU with 128-bit vector registers, for the loop in this example, the compiler first vectorizes the loop by loading four 32-bit elements of float data type into the 128-bit vector registers, and then unrolls the vectorized loop by 4 if TCOUNT=16. To see the loop unrolling factor chosen by the compiler:

1. Select **Project > Properties > Diagnostics > Optimization Diagnostic Level > Maximum (/Qopt-report:3)** and **Project > Properties > Diagnostics > optimization Diagnostic Phase > The High Performance Optimizer phase (/Qopt-report-phase:hpo)**.



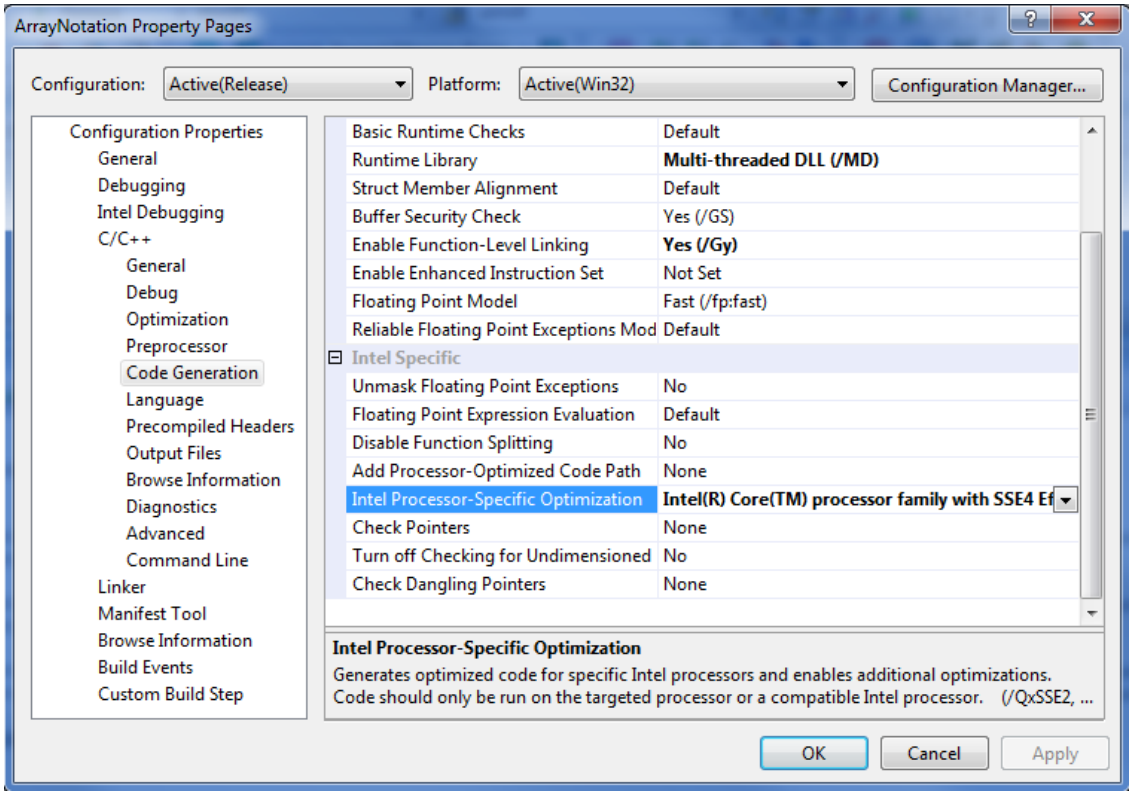
Below is the optimization report we get about the loop unroll factor:

1>ArrayNotation.cpp(74:5-74:5):VEC:?shortvector@@YAXQAM00MQAH@Z: vectorization support: unroll factor set to 4
1>ArrayNotation.cpp(74): (col. 5) remark: LOOP WAS VECTORIZED.

The loop unrolling by the right factor can take advantage of the Instruction Level Parallelism (ILP). That's how we gain more performance than the previous case.

You can also have the compiler to generate processor-specific instructions depending on the target architecture type as follows:

1. Select **Project > Properties > C/C++ > Code Generation > Intel Processor-Specific Optimization.** Select the appropriate target instruction set. By default it is set to SSE2.



References

For more information on Array Notation, vectorization, Intel Compiler automatic vectorization, Elemental Functions and examples of using other Intel® Cilk™ Plus constructs refer to:

"SIMD Parallelism using Array Notation"
"Intel® Cilk™ Plus Language Extension Specification"
"A Guide to Autovectorization Using the Intel® C++ Compilers"
"Requirements for Vectorizing Loops"
"Requirements for Vectorizing Loops with #pragma SIMD"
"Getting Started with Intel® Cilk™ Plus SIMD Vectorization and Elemental Functions"
"Elemental functions: Writing data parallel code in C/C++ using Intel® Cilk™ Plus"
"Using Intel® Cilk™ Plus to Achieve Data and Thread Parallelism"

Optimization Notice

Intel's compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804