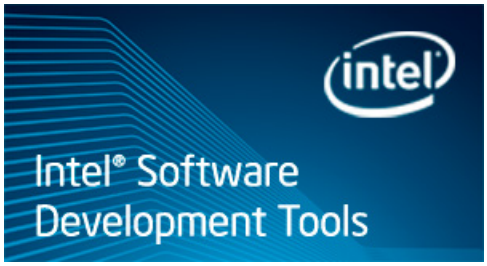

Intel® C++ Composer XE 2011 Getting Started Tutorials

Document Number: 323649-001US

World Wide Web: http://developer.intel.com

Legal Information

Contents

Legal Information

Introducing the Intel® C++ Composer XE 2011

This guide shows you how to start the Intel® C++ Composer XE 2011 and begin debugging code using the Intel® Debugger. The Intel C++ Composer XE 2011 is a comprehensive set of software development tools that includes the following components:

- Intel® C++ Compiler
- Intel® Integrated Performance Primitives
- Intel® Threading Building Blocks
- Intel® Math Kernel Library
- Intel® Debugger

Optimization Notice

The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized 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.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized 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.

Optimization Notice
Intel recommends that you evaluate other library products to determine which best meets your requirements.

Check http://software.intel.com/en-us/articles/intel-software-product-tutorials/ for the following:

- Printable version (PDF) of this Getting Started Tutorial

Prerequisites

You need the following tools, skills, and knowledge to effectively use these tutorials.

Required Skills and Knowledge

These tutorials are designed for developers with a basic understanding of the Mac OS* X, including how to:

- install the Intel® C++ Composer XE 2011 on a supported Mac OS* X version. See the Release Notes.
- open a Mac OS* X command-line shell and execute fundamental commands including make.
- compile and link C/C++ source files.

Navigation Quick Start

1

Getting Started with the Intel® C++ Composer XE 2011

The Intel® C++ Compiler XE 12.0 for Mac OS* X compiles C and C++ source files on Mac OS* X operating systems. The compiler is supported on IA-32 and Intel® 64 architectures.

You can use the Intel C++ Compiler XE 12.0 in the Xcode* integrated development environment or from the command line. This tutorial assumes you are using Xcode*, but supplies general instructions for starting the compiler from a command line.

Using the Compiler in Xcode*

You must first create or choose an existing C or C++ Xcode* project. These instructions assume you are creating a new project.

1. Launch Xcode.
2. Choose **New Project** from the **File** menu. When the **New Project Assistant** window appears, select a project template under **Application**; for example, select **Command Line Tool**. Click **Choose**.
3. Click **Next**, then name your project (`hello_world`, for example) and specify a save location. Click **Save**.
4. From within the project, highlight the target you want to change in the **Groups & Files** list under the **Target** group.
5. Double-click the target you want to change in the **Groups & Files** list under the **Target** group.
6. In the **Target Info** window, click **Rules**.
7. To add a new rule, click the **+** button at the bottom, left-hand corner of the **Target Info** window.

   From the new **Rule** section:

   - under **Process**, choose **C++ source files**
   - under **Using**, choose **Intel® C++ Compiler XE 12.0**

8. Choose **Build** from the **Build** menu or click the **Build and Go** button in the toolbar. To view the results of your build, choose **Build Results** from the **Build** menu in the Xcode toolbar.

See the Building Applications with Xcode* section in the compiler documentation for more information about using the compiler with the Xcode integrated development environment.

Using the Compiler from the Command Line

Start the compiler from a command line by performing the following steps:

1. Open a terminal session.
2. Set the environment variables for the compiler.
3. Invoke the compiler.

One way to set the environment variables prior to invoking the compiler is to "`source`" the compiler environment script, `compilervars.sh` (or `compilervars.csh`):

```
source <install-dir>/bin/compilervars.sh <arg>
```

where `<install-dir>` is the directory structure containing the compiler `/bin` directory, and `<arg>` is the architecture argument listed below.

The environment script takes an argument based on architecture. Valid arguments are as follows:

- `ia32`: Compilers and libraries for IA-32 architectures only
- `intel64`: Compilers and libraries for Intel® 64 architectures only

To compile C source files, use a command similar to the following:

```
icc my_source_file.c
```

To compile C++ source files, use a command similar to the following:

```
icpc my_source_file.cpp
```

Following successful compilation, an executable is created in the current directory.

Getting Started with the Intel® Debugger

The Intel® Debugger (IDB) is a full-featured symbolic source code application debugger that helps programmers:

- Debug programs
- Disassemble and examine machine code and examine machine register values
- Debug programs with shared libraries
- Debug multithreaded applications

The debugger features include:

- C/C++ language support
- Assembler language support
- Access to the registers your application accesses
- Bitfield editor to modify registers
- MMU support

Starting the Intel® Debugger

On Mac OS* X, you can use the Intel Debugger only from the command-line. To start the command-line invocation of the Intel Debugger, execute the `idbc` command.

Tutorial: Intel® C++ Compiler 2

Using Auto Vectorization

Introduction to Auto-vectorization

For the Intel® C++ Compiler, vectorization is the unrolling of a loop combined with the generation of packed SIMD instructions. Because the packed instructions operate on more than one data element at a time, the loop can execute more efficiently. It is sometimes referred to as auto-vectorization to emphasize that the compiler automatically identifies and optimizes suitable loops on its own.

Using the `-vec` (Linux* OS) or the `/Qvec` (Windows* OS) option enables vectorization 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 than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` or `/Qx` (Windows) or `-m` or `-x` (Linux and Mac OS X).

Vectorization is enabled with the Intel C++ Compiler at optimization levels of `-O2` and higher. Many loops are vectorized automatically, but in cases where this doesn't happen, you may be able to vectorize loops by making simple code modifications. In this tutorial, you will:

- establish a performance baseline
- generate a vectorization report
- improve performance by pointer disambiguation
- improve performance by aligning data
- improve performance using Interprocedural Optimization

Locating the Samples

To begin this tutorial, locate the source files in the product's `Samples` directory:
`<install-dir>/Samples/<locale>/C++/vec_samples/`

Use these files for this tutorial:

- `Driver.c`
- `Multiply.c`
- `Multiply.h`

Establishing a Performance Baseline

To set a performance baseline for the improvements that follow in this tutorial, compile your sources with these compiler options:

```
icc -O1 -std=c99 -DNOFUNCCALL Multiply.c Driver.c -o MatVector
```

Execute `MatVector` and record the execution time reported in the output. This is the baseline against which subsequent improvements will be measured.

This example uses a variable length array (VLA), and therefore, must be compiled with the `-std=c99` option.

Generating a Vectorization Report

A vectorization report tells you whether the loops in your code were vectorized, and if not, explains why not.

Because vectorization is off at `-O1`, the compiler does not generate a vectorization report, so recompile at `-O2` (default optimization):

```
icc -std=c99 -DNOFUNCCALL -vec-report1 Multiply.c Driver.c -o MatVector
```

Record the new execution time. The reduction in time is mostly due to auto-vectorization of the inner loop at line 150 noted in the vectorization report:

```
Driver.c(150) (col. 4): remark: LOOP WAS VECTORIZED. Driver.c(164) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(81) (col. 2): remark: LOOP WAS VECTORIZED.
```

The `-vec-report2` option returns a list that also includes loops that were not vectorized, along with the reason why the compiler did not vectorize them.

```
icc -std=c99 -DNOFUNCCALL -vec-report2 Multiply.c Driver.c -o MatVector
```

The vectorization report indicates that the loop at line 45 in `Multiply.c` did not vectorize because it is not the innermost loop of the loop nest. Two versions of the innermost loop at line 55 were generated, but neither version was vectorized.

```
Multiply.c(45) (col. 2): remark: loop was not vectorized: not inner loop. Multiply.c(55) (col. 3): remark: loop was not vectorized: existence of vector dependence. Multiply.c(55) (col. 3): remark: loop skipped: multiversioned. Driver.c(140) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(140) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(141) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(145) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(148) (col. 3): remark: loop was not vectorized: not inner loop. Driver.c(150) (col. 4): remark: LOOP WAS VECTORIZED. Driver.c(164) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(81) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(69) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(54) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(55) (col. 3): remark: loop was not vectorized: vectorization possible but seems inefficient.
```

NOTE. For more information on the `-vec-report` compiler option, see the Compiler Options section in the Compiler User and Reference Guide.

Improving Performance by Pointer Disambiguation

Two pointers are aliased if both point to the same memory location. Storing to memory using a pointer that might be aliased may prevent some optimizations. For example, it may create a dependency between loop iterations that would make vectorization unsafe. Sometimes, the compiler can generate both a vectorized and a non-vectorized version of a loop and test for aliasing at runtime to select the appropriate code path. If you know that pointers do not alias and inform the compiler, it can avoid the runtime check and generate a single vectorized code path. In `Multiply.c`, the compiler generates runtime checks to determine whether or not the pointer `b` in function `matvec(FTYPE a[][COLWIDTH], FTYPE b[], FTYPE x[])` is aliased to either `a` or `x` . If `Multiply.c` is compiled with the NOALIAS macro, the restrict qualifier of the argument `b` informs the compiler that the pointer does not alias with any other pointer, and in particular that the array `b` does not overlap with `a` or `x`.

NOTE. The `restrict` qualifier requires the use of either the `-restrict` compiler option for `.c` or `.cpp` files, or the `-std=c99` compiler option for `.c` files.

Replace the `NOFUNCCALL` macro with `NOALIAS`.

```
icc -std=c99 -vec-report2 -DNOALIAS Multiply.c Driver.c -o MatVector
```

This conditional compilation replaces the loop in the main program with a function call. Execute `MatVector` and record the execution time reported in the output.

Multiply.c(45) (col. 2): remark: loop was not vectorized: not inner loop. Multiply.c(55) (col. 3): remark: LOOP WAS VECTORIZED. Driver.c(140) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(140) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(141) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(164) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(145) (col. 2): remark: loop was not vectorized: nonstandard loop is not a vectorization candidate. Driver.c(81) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(69) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(54) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(55) (col. 3): remark: loop was not vectorized: vectorization possible but seems inefficient.

Now that the compiler has been told that the arrays do not overlap, it knows that it is safe to vectorize the loop.

Improving Performance by Aligning Data

The vectorizer can generate faster code when operating on aligned data. In this activity you will improve performance by aligning the arrays `a`, `b`, and `x` in `Driver.c` on a 16-byte boundary so that the vectorizer can use aligned load instructions for all arrays rather than the slower unaligned load instructions and can avoid runtime tests of alignment. Using the `ALIGNED` macro will modify the declarations of `a`, `b`, and `x` in `Driver.c` using the `__attribute` keyword, which has the following syntax:

```
float array[30] __attribute((aligned(base, [offset])));
```

This instructs the compiler to create an array that it is aligned on a "base"-byte boundary with an "offset" (Default=0) in bytes from that boundary. Example:

```
FTYPE a[ROW][COLWIDTH] __attribute((aligned(16)));
```

In addition, the row length of the matrix, a, needs to be padded out to be a multiple of 16 bytes, so that each individual row of a is 16-byte aligned. To derive the maximum benefit from this alignment, we also need to tell the vectorizer it can safely assume that the arrays in `Multiply.c` are aligned by using `#pragma vector aligned`.

NOTE. If you use `#pragma vector aligned`, you must be sure that all the arrays or subarrays in the loop are 16-byte aligned. Otherwise, you may get a runtime error. Aligning data may still give a performance benefit even if `#pragma vector aligned` is not used. See the code under the `ALIGNED` macro in `Multiply.c`

If your compilation targets the Intel© AVX instruction set, you should try to align data on a 32-byte boundary. This may result in improved performance. In this case, `#pragma vector aligned` advises the compiler that the data is 32-byte aligned.

Recompile the program after adding the `ALIGNED` macro to ensure consistently aligned data.

```
icc -std=c99 -vec-report2 -DNOALIAS -DALIGNED Multiply.c Driver.c -o MatVector
```

Multiply.c(45) (col. 2): remark: loop was not vectorized: not inner loop. Multiply.c(55) (col. 3): remark: LOOP WAS VECTORIZED. Driver.c(140) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(140) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(140) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(141) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(164) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(145) (col. 2): remark: loop was not vectorized: nonstandard loop is not a vectorization candidate. Driver.c(81) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(69) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(72) (col. 3): remark: LOOP WAS VECTORIZED. Driver.c(54) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(55) (col. 3): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(60) (col. 3): remark: loop was not vectorized: not inner loop. Driver.c(61) (col. 4): remark: LOOP WAS VECTORIZED.

Now, run the executable and record the execution time.

Improving Performance with Interprocedural Optimization

The compiler may be able to perform additional optimizations if it is able to optimize across source line boundaries. These may include, but are not limited to, function inlining. This is enabled with the `-ipo` option.

Recompile the program using the `-ipo` option to enable interprocedural optimization.

```
icc -std=c99 -vec-report2 -DNOALIAS -DALIGNED -ipo Multiply.c Driver.c -o MatVector
```

Note that the vectorization messages now appear at the point of inlining in `Driver.c` (line 155).

Driver.c(145) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(155) (col. 3): remark: loop was not vectorized: not inner loop. Driver.c(155) (col. 3): remark: LOOP WAS VECTORIZED. Driver.c(164) (col. 2): remark: LOOP WAS VECTORIZED. Driver.c(54) (col. 2): remark: loop was not vectorized: not inner loop. Driver.c(55) (col. 3): remark: loop was not vectorized: vectorization possible but seems inefficient. Driver.c(60) (col. 3): remark: LOOP WAS VECTORIZED. Driver.c(69) (col. 2): remark: loop was not vectorized: vectorization possible but seems inefficient.

Now, run the executable and record the execution time.

Additional Exercises

The previous examples made use of double precision arrays. They may be built instead with single precision arrays by adding the macro, `FTYPE=float`. The non-vectorized versions of the loop execute only slightly faster the double precision version; however, the vectorized versions are substantially faster. This is because a packed SIMD instruction operating on a 16-byte vector register operates on four single precision data elements at once instead of two double precision data elements.

NOTE. In the example with data alignment, you will need to set `COLBUF=3` to ensure 16-byte alignment for each row of the matrix a. Otherwise, `#pragma vector aligned` will cause the program to fail.

This completes the tutorial for auto-vectorization, where you have seen how the compiler can optimize performance with various vectorization techniques.

Using Guided Auto-parallelization

Introduction to Guided Auto-parallelization

Guided Auto-parallelization (GAP) is a feature of the Intel® C++ Compiler that offers selective advice and, when correctly applied, results in auto-vectorization or auto-parallelization for serially-coded applications. Using the `-guide` option with your normal compiler options at `-O2` or higher is sufficient to enable the GAP technology to generate the advice for auto-vectorization. Using `-guide` in conjunction with `-parallel` will enable the compiler to generate advice for auto-parallelization.

In this tutorial, you will:

1. prepare the project for Guided Auto-parallelization.
2. run Guided Auto-parallelization.
3. analyze Guided Auto-parallelization reports.
4. implement Guided Auto-parallelization recommendations.

Preparing the Project for Guided Auto-parallelization

To begin this tutorial, open the source file archive located at:

`<install-dir>/Samples/<locale>/C++/guided_auto_parallel.tar.gz`

The following files are included:

- `Makefile`
- `main.cpp`
- `main.h`
- `scalar_dep.cpp`

- scalar_dep.h

Copy these files to a directory on your system where you have write and execute permissions.

Running Guided Auto-parallelization

You can use the -guide option to generate GAP advice. From a directory where you can compile the sample program, execute make gap_vec_report from the command-line, or execute:

```
icpc -c -guide scalar_dep.cpp
```

The GAP Report appears in the compiler output. GAP reports are encapsulated with GAP REPORT LOG OPENED and END OF GAP REPORT LOG.

GAP REPORT LOG OPENED remark #30761: Add -parallel option if you want the compiler to generate recommendations for improving auto-parallelization. scalar_dep.cpp(51): remark #30515: (VECT) Loop at line 51 cannot be vectorized due to conditional assignment(s) into the following variable(s): b. This loop will be vectorized if the variable(s) become unconditionally initialized at the top of every iteration. [VERIFY] Make sure that the value(s) of the variable(s) read in any iteration of the loop must have been written earlier in the same iteration. Number of advice-messages emitted for this compilation session: 1. END OF GAP REPORT LOG

Analyzing Guided Auto-parallelization Reports

Analyze the output generated by GAP analysis and determine whether or not the specific suggestions are appropriate for the specified source code. For this sample tutorial, GAP generates output for the loop in scalar_dep.cpp:

```
for (i=0; i<n; i++) {
    if (A[i] > 0) {b=A[i]; A[i] = 1 / A[i]; }
    if (A[i] > 1) {A[i] += b;}
  }
```

In this example, the GAP Report generates a recommendation (remark #30761) to add the -parallel option to improve auto-parallelization. Remark #30515 indicates if variable b can be unconditionally assigned, the compiler will be able to vectorize the loop.

GAP REPORT LOG OPENED remark #30761: Add -parallel option if you want the compiler to generate recommendations for improving auto-parallelization. scalar_dep.cpp(51): remark #30515: (VECT) Loop at line 51 cannot be vectorized due to conditional assignment(s) into the following variable(s): b. This loop will be vectorized if the variable(s) become unconditionally initialized at the top of every iteration. [VERIFY] Make sure that the value(s) of the variable(s) read in any iteration of the loop must have been written earlier in the same iteration. Number of advice-messages emitted for this compilation session: 1. END OF GAP REPORT LOG

Implementing Guided Auto-parallelization Recommendations

The GAP Report in this example recommends using the `-parallel` option to enable parallelization. From the command-line, execute `make gap_par_report`, or run the following:

```
icpc -c -guide -parallel scalar_dep.cpp
```

The compiler emits the following:

GAP REPORT LOG OPENED ON Wed Jul 28 14:33:09 2010 scalar_dep.cpp(51): remark #30523: (PAR) Loop at line 51 cannot be parallelized due to conditional assignment(s) into the following variable(s): b. This loop will be parallelized if the variable(s) become unconditionally initialized at the top of every iteration. [VERIFY] Make sure that the value(s) of the variable(s) read in any iteration of the loop must have been written earlier in the same iteration. [ALTERNATIVE] Another way is to use "#pragma parallel private(b)" to parallelize the loop. [VERIFY] The same conditions described previously must hold. scalar_dep.cpp(51): remark #30525: (PAR) If the trip count of the loop at line 51 is greater than 188, then use "#pragma loop count min(188)" to parallelize this loop. [VERIFY] Make sure that the loop has a minimum of 188 iterations. Number of advice-messages emitted for this compilation session: 2. END OF GAP REPORT LOG

In the GAP Report, remark #30523 indicates that loop at line 51 cannot parallelize because the variable b is conditionally assigned. Remark #30525 indicates that the loop trip count must be greater than 188 for the compiler to parallelize the loop.

Apply the necessary changes after verifying that the GAP recommendations are appropriate and do not change the semantics of the program.

For this loop, the conditional compilation enables parallelization and vectorization of the loop as recommended by GAP:

```
#ifdef TEST_GAP
#pragma loop count min (188)
  for (i=0; i<n; i++) {
        b = A[i];
    if (A[i] > 0) {A[i] = 1 / A[i];}
    if (A[i] > 1) {A[i] += b;}
  }

#else

for (i=0; i<n; i++) {
    if (A[i] > 0) {b=A[i]; A[i] = 1 / A[i]; }
    if (A[i] > 1) {A[i] += b;}
  }
#endif

}
```

To verify that the loop is parallelized and vectorized:

- Add the compiler options `-vec-report1 -par-report1`.
- Add the conditional definition `TEST_GAP` to compile the appropriate code path.

From the command-line, execute `make final`, or run the following:

```
icpc -c -parallel -DTEST_GAP -vec-report1 -par-report1 scalar_dep.cpp
```

The compiler's `-vec-report` and `-par-report` options emit the following output, confirming that the program is vectorized and parallelized:

scalar_dep.cpp(43) (col. 3): remark: LOOP WAS AUTO-PARALLELIZED. scalar_dep.cpp(43) (col. 3): remark: LOOP WAS VECTORIZED. scalar_dep.cpp(43) (col. 3): remark: LOOP WAS VECTORIZED.

For more information on using the `-guide`, `-vec-report`, and `-par-report` compiler options, see the Compiler Options section in the Compiler User Guide and Reference.

This completes the tutorial for Guided Auto-parallelization, where you have seen how the compiler can guide you to an optimized solution through auto-parallelization.