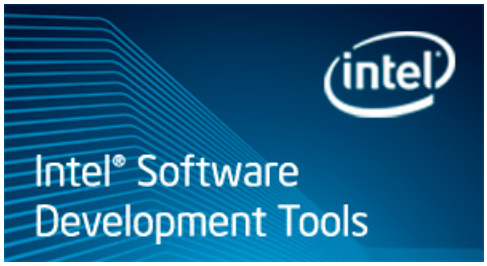

Intel® Fortran Composer XE 2011 Getting Started Tutorials

Document Number: 323651-001US

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

Contents

Legal Information

Introducing the Intel® Fortran Composer XE 2011

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

- Intel® Fortran Compiler
- 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.

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 Linux* operating system, including how to:

- install the Intel® Fortran Composer XE 2011 on a supported Linux distribution. See the Release Notes.
- open a Linux shell and execute fundamental commands including make.
- compile and link Fortran source files.

Navigation Quick Start 1

Getting Started with the Intel® Fortran Composer XE 2011

The Intel® Fortran Compiler XE 12.0 compiles Fortran source files on Linux* operating systems. The compiler is supported on IA-32 and Intel® 64 architectures. The compiler operates only from a command line on Linux* operating systems.

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

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

```
ifort my_source_file.f90
```

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:

- Fortran language support including Fortran 95/90
- Assembler language support
- Access to the registers your application accesses
- Bitfield editor to modify registers
- MMU support

Starting the Intel® Debugger

On Linux*, you can use the Intel Debugger from a Java* GUI application or the command-line.

- To start the GUI for the Intel Debugger, execute the `idb` command from a Linux shell.
- To start the command-line invocation of the Intel Debugger, execute the `idbc` command from a Linux shell.

Tutorial: Intel® Fortran Compiler 2

Using Auto Vectorization

Introduction to Auto-vectorization

For the Intel® Fortran 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 Fortran 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 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>/Fortran/vec_samples/`

Use these files for this tutorial:

- `driver.f90`
- `matvec.f90`

Establishing a Performance Baseline

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

```
ifort -real-size 64 -O1 -vec-report1 matvec.f90 driver.f90 -o MatVector
```

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

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):

```
ifort -real-size 64 -vec-report1 matvec.f90 driver.f90 -o MatVector
```

Record the new execution time. The reduction in time is mostly due to auto-vectorization of the inner loop at line 32 noted in the vectorization report:

```
matvec.f90(32) (col. 3): remark: LOOP WAS VECTORIZED. matvec.f90(38) (col. 6): remark: LOOP WAS VECTORIZED. driver.f90(59) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(61) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(80) (col. 29): 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.

```
ifort -real-size 64 -vec-report2 matvec.f90 driver.f90 -o MatVector
```

The vectorization report indicates that the loop at line 33 in `matvec.f90` did not vectorize because it is not the innermost loop of the loop nest.

```
matvec.f90(32) (col. 3): remark: LOOP WAS VECTORIZED. matvec.f90(33) (col. 3): remark: loop was not vectorized: not inner loop. matvec.f90(38) (col. 6): remark: LOOP WAS VECTORIZED. driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: loop was not vectorized: vectorization possible but seems inefficient. driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: loop was not vectorized: subscript too complex. driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(61) (col. 5): remark: loop was not vectorized: vectorization possible but seems inefficient. driver.f90(61) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(80) (col. 29): remark: LOOP WAS VECTORIZED. driver.f90(74) (col. 7): remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.
```

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 Aligning Data

The vectorizer can generate faster code when operating on aligned data. In this activity you will improve the vectorizer performance by aligning the arrays `a`, `b`, and `c` in `driver.f90` on a 16-byte boundary so 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 insert an alignment directive for `a`, `b`, and `c` in `driver.f90` with the following syntax:

```
!dir$attributes align : 16 :: a,b,c
```

This instructs the compiler to create arrays that it are aligned on a 16-byte boundary, which should facilitate the use of SSE aligned load instructions.

In addition, the column height of the matrix a needs to be padded out to be a multiple of 16 bytes, so that each individual column of a maintains the same 16-byte alignment. In practice, maintaining a constant alignment between columns is much more important than aligning the start of the arrays.

To derive the maximum benefit from this alignment, we also need to tell the vectorizer it can safely assume that the arrays in `matvec.f90` are aligned by using the directive

```
!dir$ vector aligned
```

NOTE. If you use `!dir$ 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 `!dir$ vector aligned` is not used. See the code under the `ALIGNED` macro in `matvec.f90`

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, `!dir$ 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:

```
ifort -real-size 64 -vec-report2 -DALIGNED matvec.f90 driver.f90 -o MatVector
```

matvec.f90(32) (col. 3): remark: LOOP WAS VECTORIZED. matvec.f90(33) (col. 3): remark: loop was not vectorized: not inner loop. matvec.f90(38) (col. 6): remark: LOOP WAS VECTORIZED. driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: loop was not vectorized: vectorization possible but seems inefficient. driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: loop was not vectorized: subscript too complex. driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(61) (col. 5): remark: loop was not vectorized: vectorization possible but seems inefficient. driver.f90(61) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(63) (col. 21): remark: loop was not vectorized: not inner loop. driver.f90(63) (col. 21): remark: LOOP WAS VECTORIZED. driver.f90(80) (col. 29): remark: LOOP WAS VECTORIZED. driver.f90(74) (col. 7): remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.

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.

```
ifort -real-size 64 -vec-report2 -DALIGNED -ipo matvec.f90 driver.f90 -o MatVector
```

Note that the vectorization messages now appear at the point of inlining in `driver.f90` (line 70).

```
driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: loop was not vectorized: vectorization possible but seems inefficient. driver.f90(59) (col. 5): remark: loop was not vectorized: not inner loop. driver.f90(59) (col. 5): remark: loop was not vectorized: subscript too complex. driver.f90(59) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(61) (col. 5): remark: loop was not vectorized: vectorization possible but seems inefficient. driver.f90(61) (col. 5): remark: LOOP WAS VECTORIZED. driver.f90(63) (col. 21): remark: loop was not vectorized: not inner loop. driver.f90(63) (col. 21): remark: LOOP WAS VECTORIZED. driver.f90(73) (col. 16): remark: loop was not vectorized: not inner loop. driver.f90(70) (col. 14): remark: LOOP WAS VECTORIZED. driver.f90(70) (col. 14): remark: loop was not vectorized: not inner loop. driver.f90(70) (col. 14): remark: LOOP WAS VECTORIZED. driver.f90(80) (col. 29): remark: LOOP WAS VECTORIZED.
```

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 changing the command-line option `-real-size 64` to `-real-size 32` 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 `ROWBUF=3` to ensure 16-byte alignment for each row of the matrix a. Otherwise, the directive `!dir$ 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® Fortran 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>/Fortran/guided_auto_parallel.tar.gz`

The following files are included:

- `Makefile`
- `main.f90`
- `scalar_dep.f90`

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 vec` from the command-line, or execute:

```
ifort -c -guide scalar_dep.f90
```

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 ON Mon Aug 2 14:04:34 2010 remark #30761: Add -parallel option if you want the compiler to generate recommendations for improving auto-parallelization. scalar_dep.f90(44): remark #30515: (VECT) Loop at line 44 cannot be vectorized due to conditional assignment(s) into the following variable(s): t. 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.f90`:

```
do i = 1, n
   if (a(i) >= 0) then
       t = i
   end if
   if (a(i) > 0) then
       a(i) = t * (1 / (a(i) * a(i)))
   end if
end do
```

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 `t` 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.f90(44): remark #30515: (VECT) Loop at line 44 cannot be vectorized due to conditional assignment(s) into the following variable(s): t. 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:

```
ifort -c -parallel -guide scalar_dep.f90
```

The compiler emits the following:

GAP REPORT LOG OPENED ON Mon Aug 2 14:04:44 2010 scalar_dep.f90(44): remark #30523: (PAR) Loop at line 44 cannot be parallelized due to conditional assignment(s) into the following variable(s): t. 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 "!dir$ parallel private(t)" to parallelize the loop. [VERIFY] The same conditions described previously must hold. scalar_dep.f90(44): remark #30525: (PAR) If the trip count of the loop at line 44 is greater than 36, then use "!dir$ loop count min(36)" to parallelize this loop. [VERIFY] Make sure that the loop has a minimum of 36 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 44 cannot parallelize because the variable t is conditionally assigned. Remark #30525 indicates that the loop trip count must be greater than 36 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:

```
        do i = 1, n
!dir$ if defined(test_gap)
            t = i
!dir$else
            if (a(i) >= 0) then
                t = i
            end if
!dir$ endif
            if (a(i) > 0) then
                a(i) = t * (1 / (a(i) * a(i)))
            end if
        end do
```

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 w_changes`, or run the following:

```
ifort -c -parallel -Dtest_gap -vec-report1 -par-report1 scalar_dep.f90
```

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

scalar_dep.f90(44) (col. 9): remark: LOOP WAS AUTO-PARALLELIZED. scalar_dep.f90(44) (col. 9): remark: LOOP WAS VECTORIZED. scalar_dep.f90(44) (col. 9): 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.

Using Coarry Fortran

Introduction to Coarray Fortran

The Intel® Fortran Compiler XE supports parallel programming using coarrays as defined in the Fortran 2008 standard. As an extension to the Fortran language, coarrays offer one method to use Fortran as a robust and efficient parallel programming language. Coarray Fortran uses a single-program, multi-data programming model (SPMD).

Coarrays are supported in the Intel® Fortran Compiler XE for Linux* and Intel® Visual Fortran Compiler XE for Windows*.

This tutorial demonstrates how to compile a simple coarray Fortran application using the Intel Fortran Compiler XE, and how to control the number of images (processes) for the application.

Locating the Sample

To begin this tutorial, locate the source file in the product's Samples directory:
`<install-dir>/Samples/<locale>/Fortran/coarray_samples/hello_image.f90`

Copy `hello_image.f90` to a working directory, then continue with this tutorial.

NOTE. The Intel Fortran Compiler implementation of coarrays follows the standard provided in a draft version of the Fortran 2008 Standard. Not all features present in the Fortran 2008 Standard may be implemented by Intel. Consult the Release Notes for a list of supported features.

Compiling the Sample Program

The `hello_image.f90` sample is a hello world program. Unlike the usual hello world, this coarray Fortran program will spawn multiple images, or processes, that will run concurrently on the host computer. Examining the source code for this application shows a simple Fortran program:

```
program hello_image

  write(*,*) "Hello from image ", this_image(), &
              "out of ", num_images()," total images"

end program hello_image
```

Note the function calls to `this_image()` and `num_images()`. These are new Fortran 2008 intrinsic functions. The `num_images()` function returns the total number of images or processes spawned for this program. The `this_image()` function returns a unique identifier for each image in the range 1 to `N`, where `N` is the total number of images created for this program.

To compile the sample program containing the Coarray Fortran features, use the `-coarray` compiler option:

```
ifort -coarray hello_image.f90 -o hello_image
```

If you run the `hello_image` executable, the output will vary depending on the number of processor cores on your system:

```
./hello_image
```

Hello from image 1 out of 8 total images Hello from image 6 out of 8 total images Hello from image 7 out of 8 total images Hello from image 2 out of 8 total images Hello from image 5 out of 8 total images Hello from image 8 out of 8 total images Hello from image 3 out of 8 total images Hello from image 4 out of 8 total images

By default, when a Coarray Fortran application is compiled with the Intel Fortran Compiler, the invocation creates as many images as there are processor cores on the host platform. The example shown above was run on a dual quad-core host system with eight total cores. As shown, each image is a separately spawned process on the system and executes asynchronously.

NOTE. The `-coarray` option cannot be used in conjunction with `-openmp` options. One cannot mix Coarray Fortran language extensions with OpenMP extensions.

Controlling the Number of Images

There are two methods to control the number of images created for a Coarray Fortran application. First, you can use the `-coarray-num-images=N` compiler option to compile the application, where `N` is the number of images. This option sets the number of images created for the application during run time. For example, use the `-coarray-num-images=2` option to the limit the number of images of the `hello_image.f90` program to exactly two:

```
ifort -coarray -coarray-num-images=2 hello_image.f90 -o hello_image
```

```
Hello from image 2 out of 2 total images Hello from image 1 out of 2 total images
```

The second way to control the number of images is to use the environment variable `FOR_COARRAY_NUM_IMAGES`, setting this to the number of images you want to spawn.

As an example, recompile `hello_image.f90` without the `-coarray-num-images` option. Instead, before we run the executable `hello_image`, set the environment variable `FOR_COARRAY_NUM_IMAGES` to the number of images you want created during the program run.

For bash shell users, set the environment variable with this command: `export FOR_COARRAY_NUM_IMAGES=4`

For csh/tcsh shell users, set the environment variable with this command: `setenv FOR_COARRAY_NUM_IMAGES 4`

For example, assuming bash shell:

```
ifort -coarray hello_image.f90 -o hello_image
```

```
export FOR_COARRAY_NUM_IMAGES=4
```

```
Hello from image 1 out of 4 total images Hello from image 3 out of 4 total images Hello from image 2 out of 4 total images Hello from image 4 out of 4 total images
```

```
export FOR_COARRAY_NUM_IMAGES=3
```

```
Hello from image 3 out of 3 total images Hello from image 2 out of 3 total images Hello from image 1 out of 3 total images
```

NOTE. Setting `FOR_COARRAY_NUM_IMAGES=N` overrides the `-coarray_num_images` compiler option.