Intel® C++ Composer XE 2011 Getting Started Tutorials

Document Number: 323647-001US

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® Parallel Debugger Extension. The Intel(R) 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® Parallel Debugger Extension

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

- ShowMe video for using Intel® C++ Composer XE with Microsoft Visual Studio*

Prerequisites

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

NOTE. Although the instructions and screen captures in these tutorials refer to the Visual Studio* 2005 integrated development environment (IDE), you can use these tutorials with later versions of Visual Studio.

Required Tools

You need the following tools to use these tutorials:

- Microsoft Visual Studio 2005 or later.
- Intel® C++ Composer XE 2011.
- Sample code included with the Intel® C++ Composer XE 2011.

NOTE.

- Samples are non-deterministic. Your results may vary from the examples shown throughout these tutorials.
- Samples are designed only to illustrate features and do not represent best practices for creating multithreaded code.

Required Skills and Knowledge

These tutorials are designed for developers with a basic understanding of Microsoft Visual Studio, including how to:

- open a project/solution.
- access the **Document Explorer**. (valid in Microsoft Visual Studio 2005 /2008)
- display the **Solution Explorer**.
- compile and link a project.
- ensure a project compiled successfully.

Navigation Quick Start

1

Starting the Intel® C++ Compiler from the Microsoft Visual Studio* IDE

The Intel® C++ Composer XE 2011 integrates into the following versions of the Microsoft Visual Studio* Integrated Development Environment (IDE):

- Microsoft Visual Studio 2010*
- Microsoft Visual Studio 2008*
- Microsoft Visual Studio 2005*

Using the Intel® C++ Composer XE 2011 from Microsoft Visual Studio* IDE

To use the Intel® C++ Compiler do the following:

1. Launch Microsoft Visual Studio*.
2. Open or create a Visual Studio solution in the **Solution Explorer** pane.
3. From the **Project** menu, select **Intel C++ Compiler XE > Use Intel C++**.
4. Click **OK** in the **Confirmation** dialog box. This configures the solution to use the Intel® C++ Compiler. (Visual Studio 2008 or Visual Studio 2005: you can configure the solution to use the Intel® C++ Compiler by clicking on the toolbar icon  . Visual Studio 2010: you can use **Project > Properties General > Platform Toolset** to select the Intel C++ Compiler. This method is equivalent to using the Use Intel C++ menu item except you can make the selection in individual build configurations.)
5. Select **Rebuild Solution** from the Visual Studio **Build** menu.

The results of the compilation display in the Output window.

Setting Intel® C++ Compiler Options

1. Select **Project > Properties**. The Property Pages for your solution display.
2. Locate **C/C++** in the list and expand the heading.
3. Step through the available properties to select your configuration.

Compatibility

The Intel® C++ Compiler processes C and C++ language source files. The Intel® C++ Compiler is fully source- and binary-compatible (native code only) with the Microsoft Visual Studio* C++ compiler. The Intel C++ Compiler only supports native C++ project types provided by Visual Studio development environment. The project types with .NET attributes such as the ones below, cannot be converted to an Intel C++ project:

- Empty Project (.NET)
- Class Library (.NET)
- Console Application (.NET)
- Windows Control Library (.NET)
- Windows Forms Application (.NET)
- Windows Service (.NET)

Refer to the User and Reference Guides for the full list of unsupported features.

Switching between the Installed Compilers

Switching to the Intel® C++ Composer XE 2011

To switch to the Intel® C++ Compiler do the following:

1. Launch Microsoft Visual Studio*.
2. Open the solution.
3. From the **Project** menu, select **Intel C++ Compiler XE > Use Intel C++**.
4. Click **OK** in the **Confirmation** dialog box. This configures the solution to use the Intel® C++ Compiler. (Visual Studio 2008 or Visual Studio 2005: you can configure the solution to use the Intel® C++ Compiler by clicking on the toolbar icon  . Visual Studio 2010: you can use **Project > Properties General > Platform Toolset** to select the Intel C++ Compiler. This method is equivalent to using the Use Intel C++ menu item except you can make the selection in individual build configurations.)

Switching to the Microsoft Visual Studio* C++ Compiler

If you are using the Intel® C++ Compiler, you can switch to the Visual C++ Compiler at any time. Switch compilers by doing the following:

1. Launch Microsoft Visual Studio*.
2. Open the solution.
3. From the **Project** drop-down menu, select **Intel C++ Compiler XE > Use Visual C++**.

This action updates the solution file to use the Microsoft Visual Studio C++ compiler. All configurations of affected projects are automatically cleaned unless you select Do not clean project(s). If you choose not to clean projects, you will need to rebuild updated projects to ensure all source files are compiled with the new compiler.

Starting the Intel® C++ Compiler from the Command Line

Follow these steps to invoke the Intel® C++ Compiler from the command line:

1. Open a command prompt from the **Start>All Programs** menu:
   Intel Parallel Studio XE 2011 >Command Prompt Intel Parallel Studio 2011 >Command Prompt.
2. Invoke the compiler as follows:
   `icl [options... ] inputfile(s) [/link link_options]`

   Use the command `icl /help` to display all available compiler options.

Starting the Intel® Parallel Debugger Extension

The Intel® Parallel Debugger Extension for Microsoft Visual Studio* is a debugging add-on for the Intel® Compiler's parallel code development features. It facilitates developing parallelism into applications based on the Intel® OpenMP* runtime environment.

The Intel® Parallel Debugger Extension provides:

- A new Microsoft Visual Studio* toolbar
- An extension to the Microsoft Visual Studio* Debug menu
- A set of new views and dialogs that are invoked from the toolbar or the menu tree

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

Preparing Applications for Parallel Debugging

You must enable the parallel debug instrumentation with the compiler to enable parallel debugging, such as analyzing shared data or breaking at re-entrant function calls.

To enable the parallel debug instrumentation:

1. Open your application project in Microsoft Visual Studio*.
2. Select **Project > Properties...** from the menu. The Projectname Property Pages dialog box opens.
3. Enable Parallel debug checking.
   1. Select **Configuration Properties > C/C++ > Debug** in the left pane.
   2. Under **Enable Parallel Debug Checks**, select **Yes (/debug:parallel)**.
4. Click OK.
5. Rebuild your application.

Your application is now instrumented for parallel debugging using the features of the Intel® Parallel Debugger Extension.

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` 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

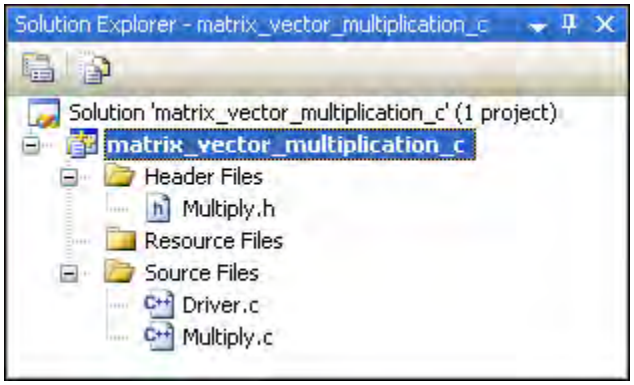
Locating the Samples

To begin this tutorial, open the `vec_samples.zip` archive in the product's `Samples` directory: `<install-dir>\Samples\<locale>\C++\vec_samples.zip`

Use these files for this tutorial:

- `matrix_vector_multiplication_c.sln`
- `matrix_vector_multiplication_c.vcproj`
- `Driver.c`
- `Multiply.c`
- `Multiply.h`

Open the Microsoft Visual Studio solution file, `matrix_vector_multiplication_c.sln`,



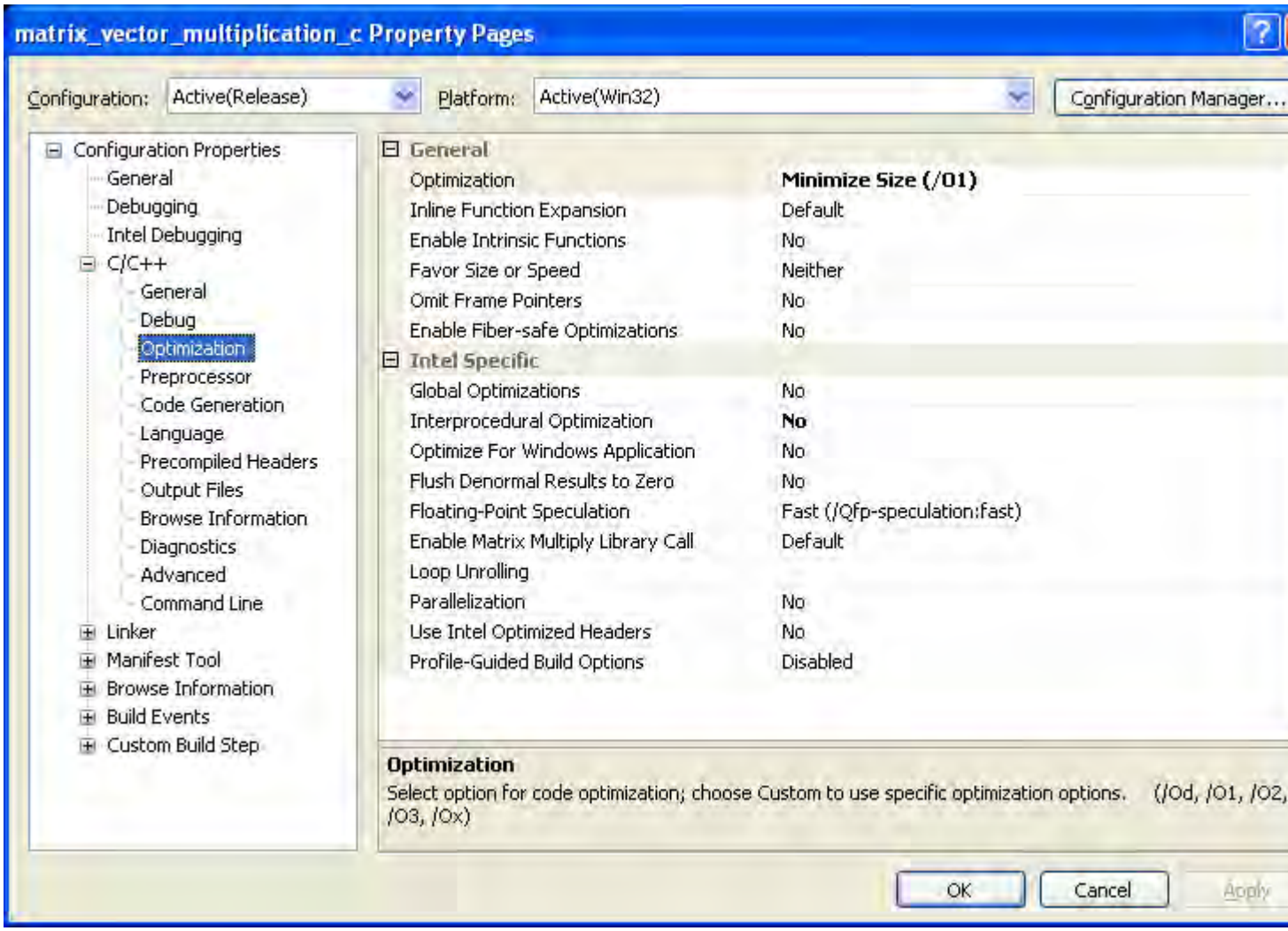
and follow the steps below to prepare the project for the vectorization exercises in this tutorial:

1. Convert to an Intel project by right-clicking on the `matrix_vector_multiplication_c` project and selecting **Intel C++ Composer XE > Use Intel C++**. Click **OK** in the **Confirmation** dialog.
2. Change the Active solution configuration to **Release** using **Build > Configuration Manager**.
3. Clean the solution by selecting **Build > Clean Solution**.

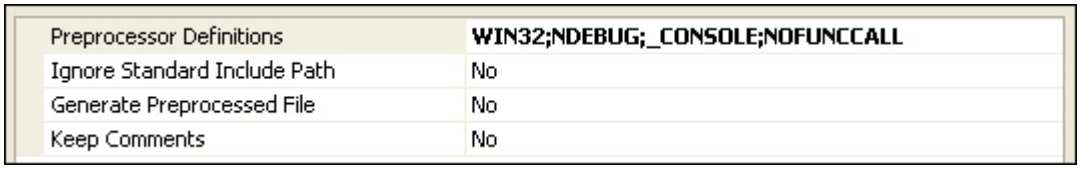
Establishing a Performance Baseline

To set a performance baseline for the improvements that follow in this tutorial, build your project with these settings:

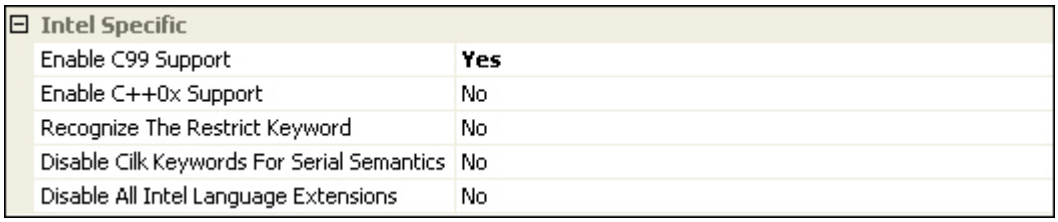
1. Select **Project > Properties > C/C++ > Optimization > General > Optimization > Minimize Size (/O1)**.
2. Select **Project > Properties > C/C++ > Optimization > Intel Specific > Interprocedural Optimization > No**.



3. Add the preprocessor definition, `NOFUNCCALL`, by selecting **Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions**, then adding `NOFUNCCALL` to the existing list of preprocessor definitions.



4. Select **Project > Properties > C/C++ > Langauage > Intel Specific > Enable C99 Support > Yes**.



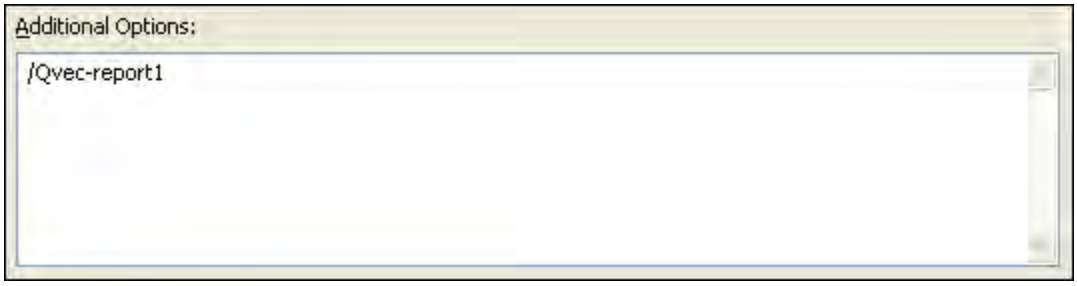
This example uses a variable length array (VLA), and therefore, must be compiled with the `/Qstd=c99` option.

5. Rebuild the project, then run the executable (**Debug > Start Without Debugging**) and record the execution time reported in the output. This is the baseline against which subsequent improvements will be measured.

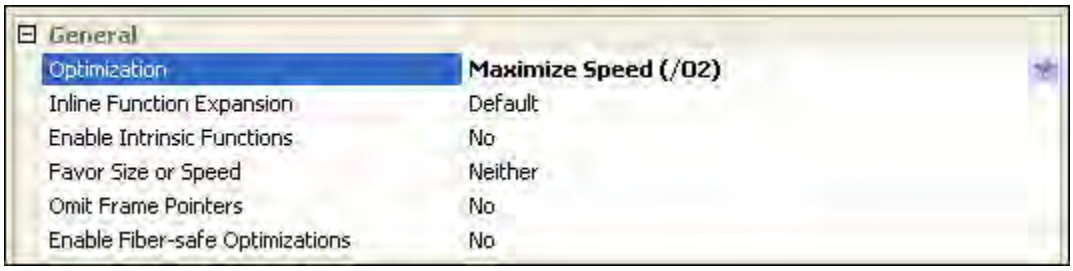
Generating a Vectorization Report

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

Add the `/Qvec-report1` option to the command line by selecting **Project > Properties > C/C++ > Command Line > Additional Options**, then adding `/Qvec-report1`.



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



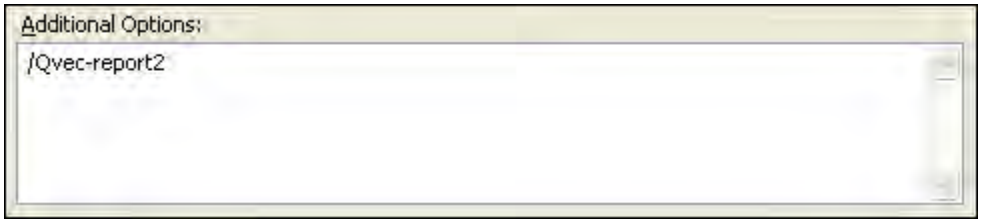
Record the new execution time. The reduction in time is mostly due to auto-vectorization of the inner loop at line 150 noted in the vectorization report:

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

For **C/C++ > Command Line > Additional Options**, change `/Qvec-report1` to `/Qvec-report2`.

Also, for **Linker > Command Line > Additional Options**, add `/Qvec-report2`:



Rebuild your project.

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.

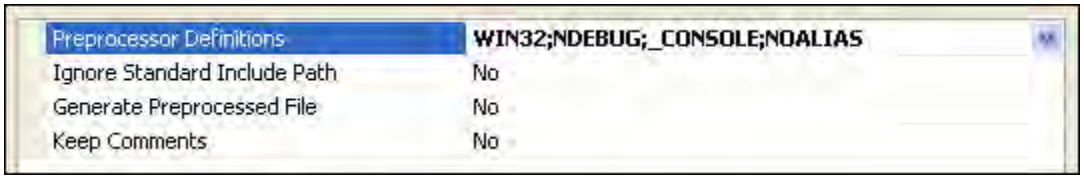
NOTE. For more information on the `/Qvec-report` compiler option, see the Compiler Options section in the Compiler User and Reference Guide.

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 `/Qrestrict` compiler option for `.c` or `.cpp` files, or the `/Qstd=c99` compiler option for `.c` files.

Replace the `NOFUNCCALL` preprocessor definition with `NOALIAS`.



This conditional compilation replaces the loop in the main program with a function call. Rebuild your project, run the executable, 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:

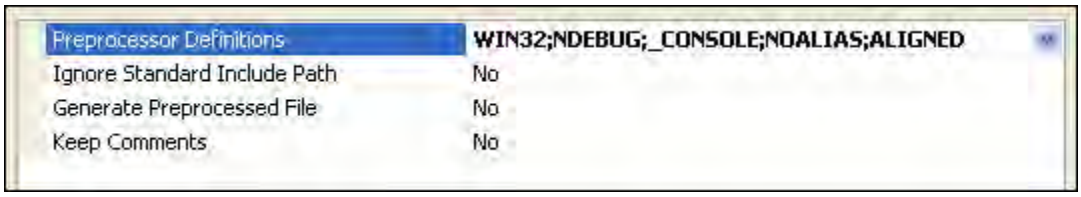
```
FTYPE a[ROW][COLWIDTH] __attribute((aligned(16)));
```

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.

Rebuild the program after adding the `ALIGNED` preprocessor definition to ensure consistently aligned data.



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.

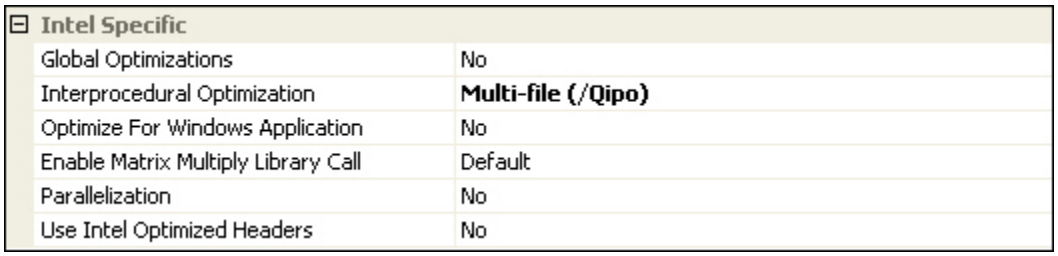
Now, run the executable and record the execution time.

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 `/Qipo` option.

Rebuild the program using the `/Qipo` option to enable interprocedural optimization.

Select **Optimization > Interprocedural Optimization > Multi-file(/Qipo)**



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 preprocessor definition, `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 `/Qguide` 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 `/Qguide` in conjunction with `/Qparallel` 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.

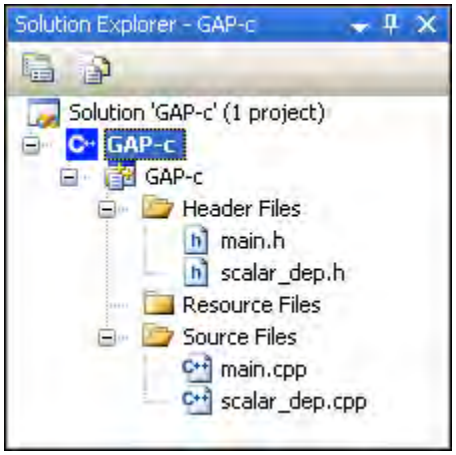
Preparing the Project for Guided Auto-parallelization

To begin this tutorial, open the `GuidedAutoParallel.zip` archive located in the product's Samples directory located at:
`\Samples\<Locale>\C++\`
The following Visual Studio* 2005 project files and source files are included:

- `GAP-c.sln`
- `GAP-c.vcproj`
- `main.cpp`
- `main.h`

- `scalar_dep.cpp`
- `scalar_dep.h`

Open the Microsoft Visual Studio Solution file, `GAP-c.sln`,



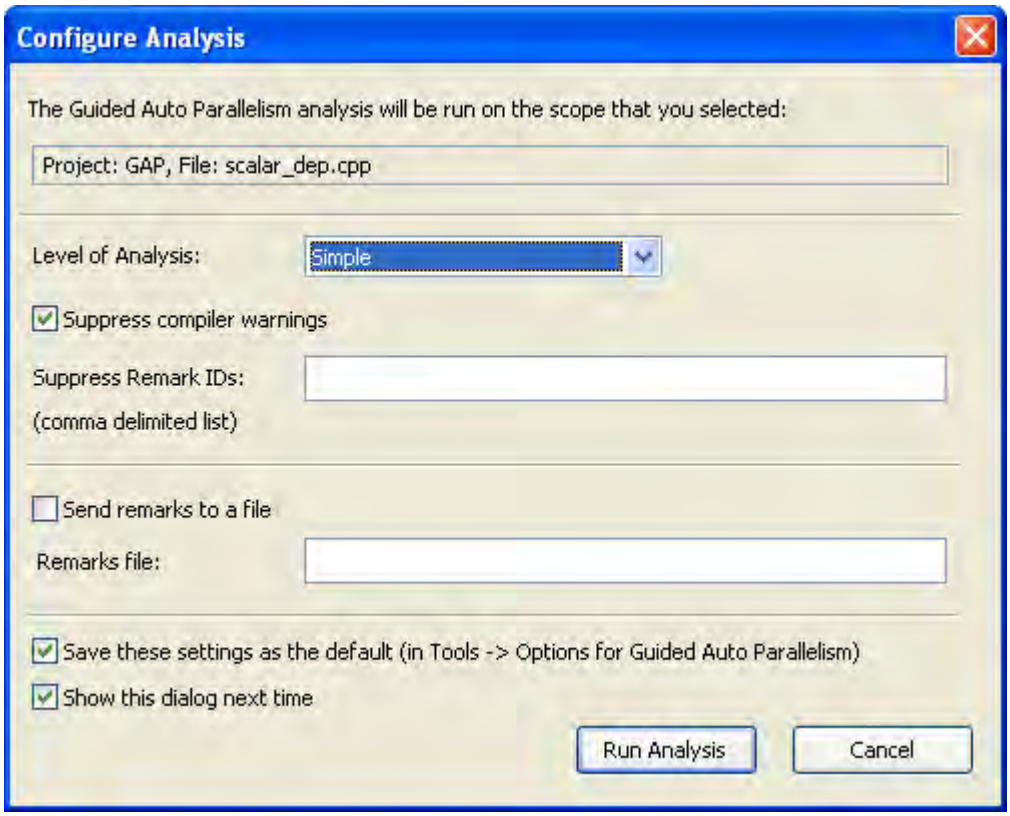
and follow the steps below to prepare the project for Guided Auto-parallelization (GAP).

1. Convert to an Intel project by right-clicking on the GAP-c project and selecting **Intel C++ Composer XE > Use Intel C++**. Click **OK** in the **Confirmation** dialog.
2. Clean the Solution by selecting **Build > Clean Solution**.
3. Since GAP is enabled only with option `/O2` or higher, you will need to change the build configuration to **Release** using **Build > Configuration Manager**.

Running Guided Auto-parallelization

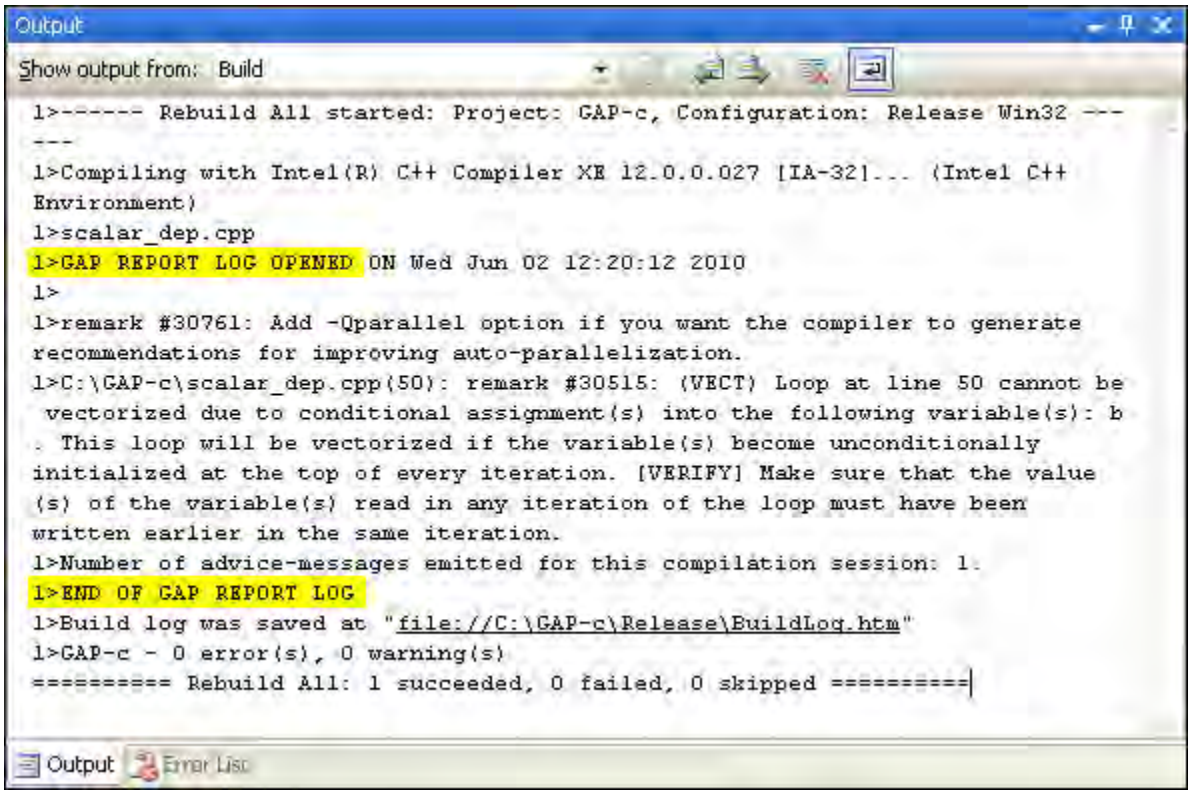
There are several ways to run GAP analysis in Visual Studio, depending on whether you want analysis for the whole solution, the project, a single file, a function, or a range of lines in your source code. In this tutorial, we will use single-file analysis. Follow the steps below to run a single-file analysis on `scalar_dep.cpp` in the GAP-c project:

1. In the GAP-c project, right-click on `scalar_dep.cpp`.
2. Select **Intel C++ Composer XE > Guided Auto Parallelism > Run Analysis on file "scalar_dep.cpp"**
3. If the `/Qipo` option is enabled, the **Analysis with Multi-file optimization** dialog appears. Click **Run Analysis**.
4. On the **Configure Analysis** dialog, click **Run Analysis** using the choices shown here:

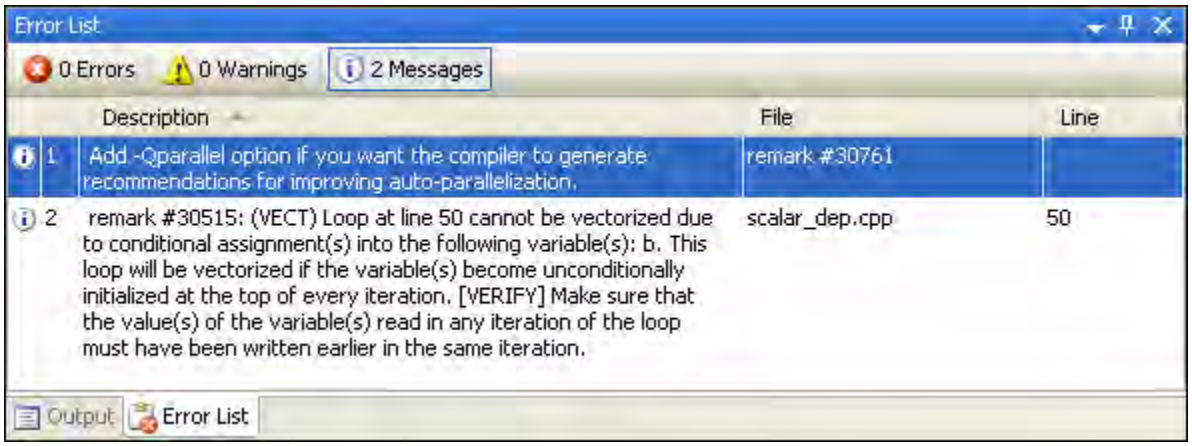


NOTE. If you select **Send remarks to a file**, GAP messages will not be available in the **Output** window or **Error List** window.

See the GAP Report in the Output window. GAP reports in the standard Output window are encapsulated with `GAP REPORT LOG OPENED` and `END OF GAP REPORT LOG`.



Also, see the GAP Messages in the Error List window:

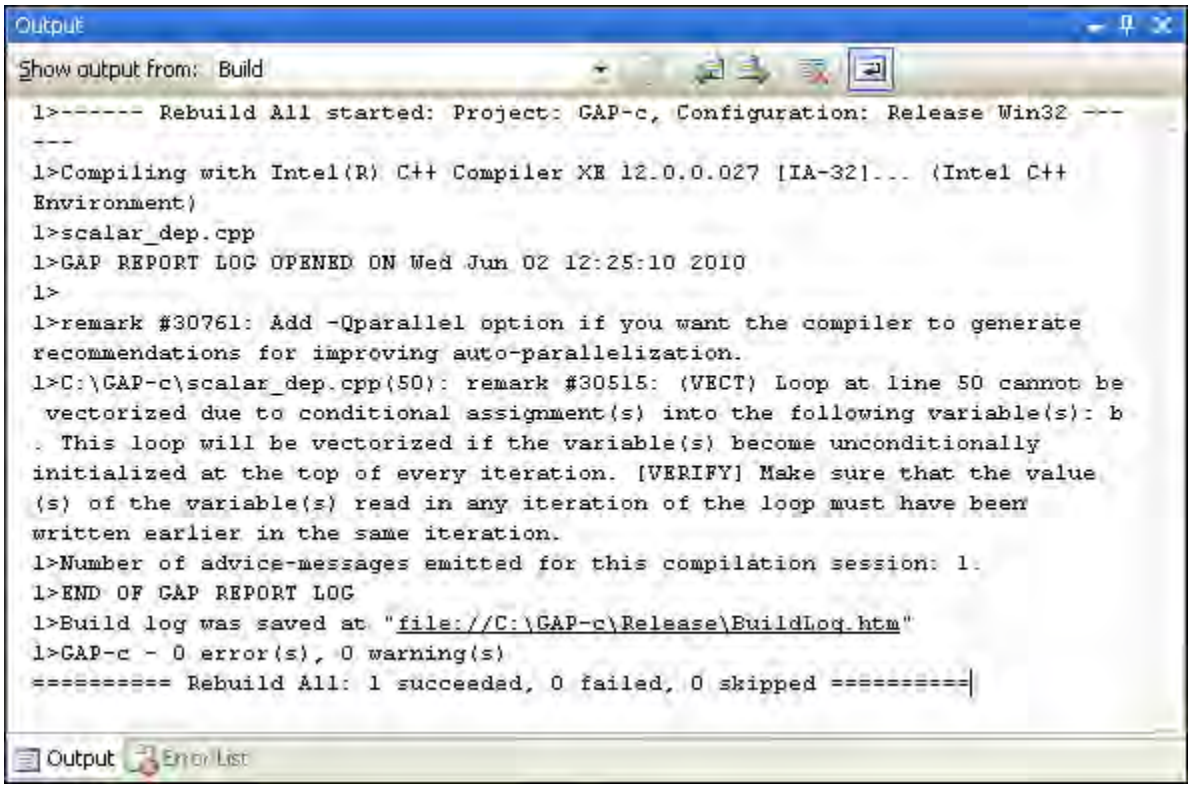


Analyzing Guided Auto-parallelization Reports

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 `/Qparallel` option to improve auto-parallelization. Remark #30515 indicates if variable b can be unconditionally assigned, the compiler will be able to vectorize the loop.

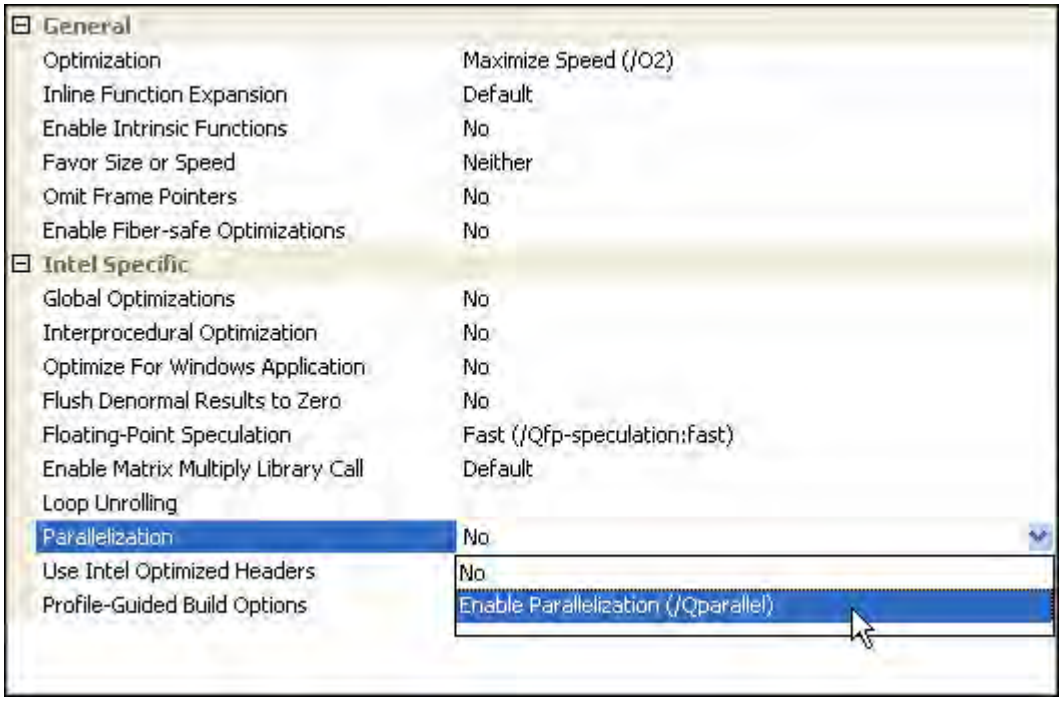


Implementing Guided Auto-parallelization Recommendations

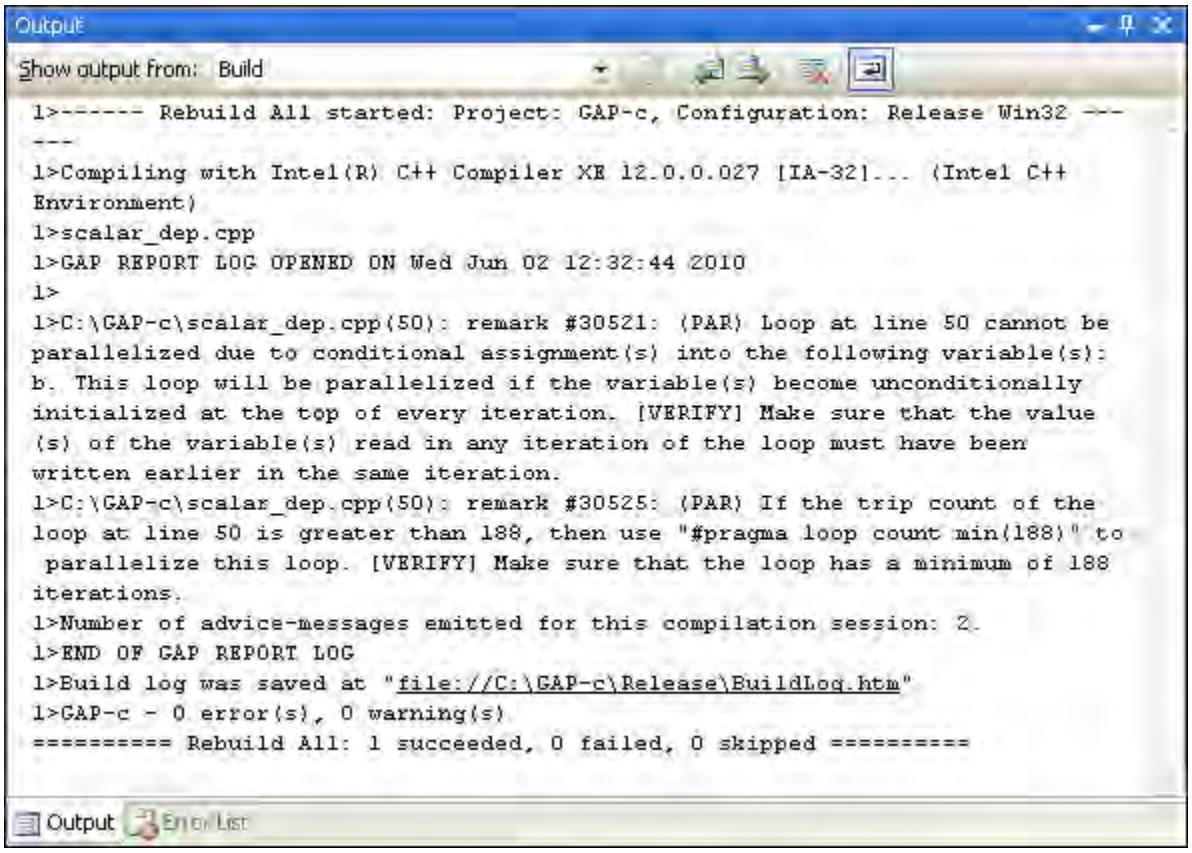
The GAP Report in this example recommends using the `/Qparallel` option to enable parallelization. Follow these steps to enable this option:

1. Right-click on the GAP-c project and select **Properties**

2. On the **Property Pages** dialog, expand the **C/C++** heading and select **Optimization**.
3. In the right-hand pane under **Intel Specific**, select **Parallelization**, then choose **Enable Parallelization (/Qparallel)** and click **OK**.



Now, run the GAP Analysis again and review the GAP Report:



The remark #30521 indicates that loop at line 50 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.

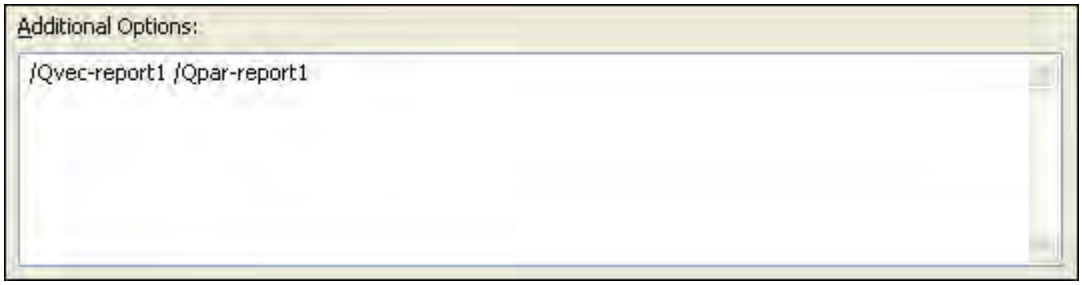
For this loop, the conditional compilation enables parallelization and vectorization of the loop as recommended by GAP:

```
#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;}
  }
```

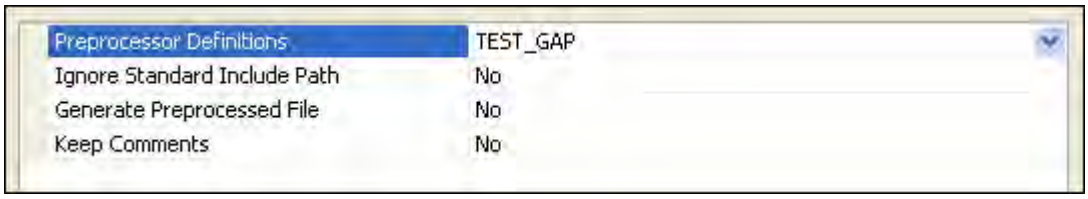
```
#endif
}
```

To verify that the loop is parallelized and vectorized:

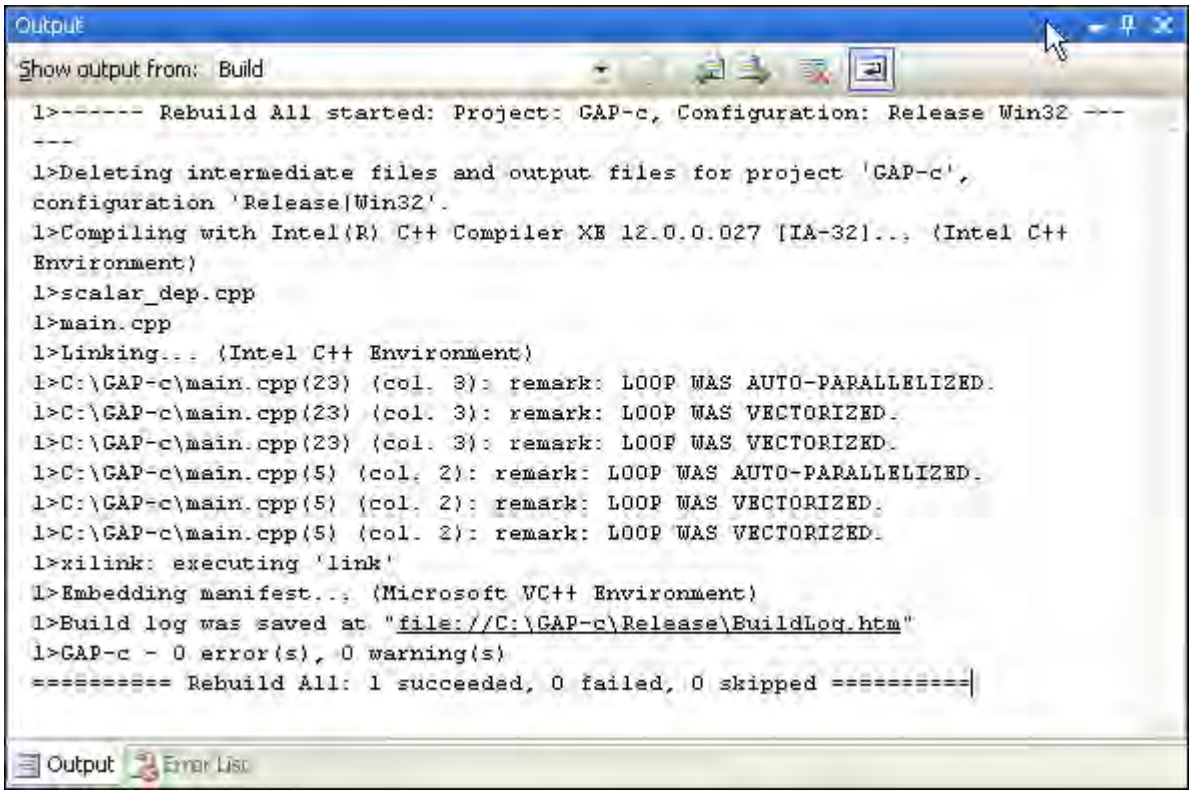
1. Add the options `/Qvec-report1 /Qpar-report1` to the **Linker > Command Line > Additional Options** dialog.



2. Add the preprocessor definition TEST_GAP to compile the appropriate code path.



3. Rebuild the GAP-c project and note the reports in the output window:



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.

Threading Your Applications

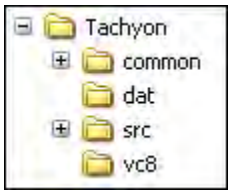
Learning Objectives

In this tutorial, we will be building different parallel implementations of the same function with both the Microsoft Visual C++* Compiler and Intel® C++ Composer XE 2011. When executed, the application will display the execution time required to render the object in the window title. This time is an indication of the speedup obtained with parallel implementations compared to a baseline established with a serial implementation in the first step.

Threading Your Application

Tachyon is a ray-tracer application, rendering objects described in data files. The Tachyon program is located in the product Samples directory: `<install-dir>\Samples\<locale>\C++\Tachyon.zip`.

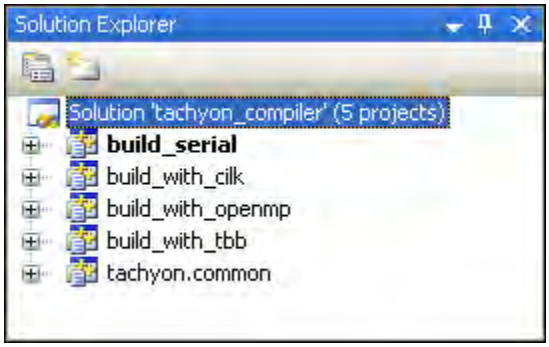
Expand the archive to `\Tachyon`



By default we use `balls.dat` as the input file. Data files are stored in the directory `\Tachyon\dat\`. Originally, Tachyon was an application with parallelism implemented in function `pthread_create()`(source file `\Tachyon\src\Windows\pthread.cpp`) with explicit threads: one for the rendering, and the other for calculations. In this tutorial we implement parallelization on the calculation thread with OpenMP*, Intel® TBB, and Intel® Cilk™ Plus. Parallelization is implemented only for one function `draw_task()`, which you can find in the source file `build_serial.cpp`, in project `build_serial`.

Open the Microsoft Visual Studio* Solution `\Tachyon\vc8\tachyon_compiler.sln`. It includes these projects:

- `build_serial`
- `build_with_cilk`
- `build_with_openmp`
- `build_with_tbb`
- `tachyon.common`



NOTE. Projects `build_with_openmp`, `build_with_tbb` and `build_with_tbbc` use OpenMP, Intel® TBB and Intel® Cilk™ Plus, respectively. In addition to these implementations, there is also an option for users to implement with lambda functionality based on Intel TBB

Follow the steps below to build the serial and Intel® Cilk™ Plus approaches to Tachyon.

Workflow Steps

In the following, we will be building different parallel implementations of the same function with both the Microsoft Visual C++ Compiler and the Intel® C++ Compiler. When executed, the application will display the execution time required to render the object in the window title. This time is an indication of the speedup obtained with parallel implementations compared to a baseline established with a serial implementation in the first step.

Building the Serial Project

1. Set the `build_serial` project as the StartUp project (**Project > Set as StartUp Project**).
2. Set the configuration to Release mode: **Build > Configuration Manager > Active solution configuration: > Release**, then build the `build_serial` project.
3. Execute the application tachyon_compiler.exe with **Debug > Start without Debugging**. Take a note of the time in seconds displayed in the window title. This time to render the image is the baseline for parallelization with the Microsoft Visual C++ Compiler.
4. For projects `build_serial` and "tachyon.common" change compiler to Intel(R) Parallel Composer (**Project > Intel C++ Composer XE 2011 > Use Intel C++ ...**).
5. Rebuild `build_serial` in Release mode (now with Intel Compiler).
6. Execute the application. Note the time to render the image as the baseline for parallelization with the Intel C++ Compiler.

Building with OpenMP*

1. Set the `build_with_openmp` project as StartUp project.
2. For project `build_with_openmp`, change the compiler to Intel C++ Composer XE (**Project > Intel C++ Composer XE > Use Intel C++...**).
3. For the project `build_with_openmp`, make sure the `/Qopenmp` compiler option is set (**Project > Properties > Configuration Properties > C/C++ > Language > OpenMP Support = Generate Parallel Code (/Qopenmp)**).
4. Open source file `build_with_openmp.cpp` in the project `build_with_openmp`.
5. Uncomment OpenMP* pragmas in the routine `draw_task` which create parallel regions and distribute loop iteration within the team of threads.
6. Comment out return inside parallel region in the routine `draw_task`.
7. Uncomment zero assignment to variable `ison` (`ison = 0;`) inside parallel region in the routine `draw_task`.
8. Uncomment return at the end of the routine `draw_task`.
9. Build `build_with_openmp` in Release configuration.
10. Execute the application.
11. Measure performance compared with the serial version.

Options that use OpenMP are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® vs. non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Building with Intel® TBB

1. Set `build_with_tbb` project as StartUp project.

2. For project `build_with_tbb`, change the compiler to Intel C++ Composer XE (**Project > Intel C++ Composer XE > Use Intel C++...**).
3. For the project `build_with_tbb` make sure the Intel® TBB environment is set (**Project > Intel C++ Composer XE > Select Build Components > Use TBB**). See Note below.
4. Open source file `build_with_tbb.cpp` in the project `build_with_tbb`.
5. Uncomment TBB header files.
6. Uncomment class `draw_task`.
7. Comment out routine `draw_task`.
8. Uncomment lines regarding TBB schedule and number of threads in routine `thread_trace`.
9. Uncomment lines regarding grain size in routine `thread_trace`.
10. Uncomment TBB `parallel_for` routine in routine `thread_trace`.
11. Comment out call of routine `draw_task` in routine `thread_trace`.
12. Build `build_with_tbb` in Release configuration.
13. Execute the application.
14. Measure performance compared with the serial version.

NOTE. Double check the following project properties are set:

- **Configuration Properties > C/C++ > General > Additional Include Directories**: contains `$(INTEL_DEF_IA32_INSTALL_DIR)TBB\Include`
- **Configuration Properties > Linker > General > Additional Library Directories**: contains `"$(INTEL_DEF_IA32_INSTALL_DIR)TBB\Lib\ia32\vc8"` for Visual Studio 2005; `"$(INTEL_DEF_IA32_INSTALL_DIR)TBB\Lib\ia32\vc9"` for Visual Studio 2008; `"$(INTEL_DEF_IA32_INSTALL_DIR)TBB\Lib\ia32\vc10"` for Visual Studio 2010;
- For platform x64, the `$(INTEL_DEF_X64_INSTALL_DIR)` is used instead of `$(INTEL_DEF_IA32_INSTALL_DIR)` and the library directory becomes `$(INTEL_DEF_X64_INSTALL_DIR)TBB\Lib\intel64\vc8` for Visual Studio 2005.

Building with Intel® Cilk™ Plus

1. Set the `build_with_cilk` project as the StartUp project.
2. For project `build_with_cilk` change compiler to the Intel C++ Compiler (**Project > Intel C++ Composer XE 2011 > Use Intel C++ ...**).
3. For the project `build_with_cilk` make sure Intel® Cilk™ Plus for Intel® C++ Compiler additional include directory is set (**Project > Properties > Configuration Properties > C/C++ > General > Additional Include Directories = C:\Program Files\Intel\ComposerXE-2011\compiler\include\cilk**).
4. Open source file `build_with_cilk.cpp` in the project `build_with_cilk`.
5. Uncomment Intel® Cilk™ Plus header files.
6. Uncomment routine `draw_task` related to Intel® Cilk™ Plus implementation.
7. Comment out the serial `draw_task()` function

8. Build `build_with_cilk` in Release mode.
9. Execute the application.
10. Measure performance compared with the serial version for Intel(R) Parallel Composer.

Platform and Other Details

The solution for this example was created in Microsoft Visual Studio 2005. If you open the `tachyon_compiler.sln` solution in Microsoft Visual Studio 2008, then it will be converted to a Microsoft Visual Studio 2008 solution.

For Platform Win32

- The executable file for all implementations is `tachyon_compiler.exe` in the `\Tachyon\vc8\Release\` directory.
- Object files are stored in `\Tachyon\vc8\tachyon_compiler\Release\` directory.

For Platform x64

- The executable file for all implementations is `tachyon_compiler.exe` in the `\Tachyon\vc8\x64\Release\` directory.
- Object files are stored in `\Tachyon\vc8\x64\tachyon_compiler\Release\` directory.