Tutorial: Using Auto Vectorization with Intel® C++ Compiler

Contents

Chapter 1: Tutorial: Using Auto Vectorization Overview

Introduction to Auto Vectorization 3
Tutorial: Linux* and macOS* 4
Preparing the Sample Application 4
Establishing a Performance Baseline 4
Generating a Vectorization Report 4
Improving Performance by Pointer Disambiguation 6
Improving Performance by Aligning Data 7
Improving Performance with Interprocedural Optimization 9
Tutorial: Windows* Version 9
Preparing the Sample Application 9
Establishing a Performance Baseline 10
Generating a Vectorization Report 10
Improving Performance by Pointer Disambiguation 12
Improving Performance by Aligning Data 13
Improving Performance with Interprocedural Optimization 14
Additional Exercises 15
Notices and Disclaimers 15

Tutorial: Using Auto Vectorization Overview

1

The Intel® Compiler has an auto-vectorizer that detects operations in the application that can be done in parallel and converts sequential operations to parallel operations by using the Single Instruction Multiple Data (SIMD) instruction set.

In this tutorial, you will use the auto-vectorizer to improve the performance of the sample application. You will compare the performance of the serial version and the version that was compiled with the auto-vectorizer.

NOTE `-qopt-report-` support is only available for the `icc` compiler. It is not available for `icx`.

About This Tutorial	This tutorial demonstrates how you can improve the performance of the sample project by using the features of the Intel® Compiler. You can use the techniques in this tutorial to improve your application.
Estimated Duration	15-20 minutes.
Learning Objectives	After you complete this tutorial, you should be able to: • Establish a performance baseline. • Generate a vectorization report. • Improve performance by using pointer disambiguation, aligning data, and interprocedural optimization.

Introduction to Auto Vectorization

For the Intel® 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.

Intel® Advisor can assist with vectorization and show optimization report messages with your source code. See https://software.intel.com/content/www/us/en/develop/tools/advisor.html for details.

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 `m` or `x`.

Vectorization is enabled with the compiler at optimization levels of `O2` (default level) and higher for both Intel® Microprocessors and non-Intel® Microprocessors. 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

This tutorial is available in Linux*, macOS*, and Windows* versions.

Tutorial: Linux* and macOS* Version

Use this section of the tutorial for Linux and macOS applications.

NOTE The `ifx` compiler is not currently available for macOS*.

Preparing the Sample Application

In this tutorial, you will use the following files:

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

Setting the Environment Variables

Before you can use the compiler, you must first set the environment variables by running the compiler environment script `compilervars.sh` or `compilervars.csh` with an argument that specifies the target architecture.

To set the environment variables:

1. Open a terminal session.
2. Run one of the scripts with the appropriate architecture argument. The following example uses `compilervars.sh`:

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

   where `<install-dir>` is the directory structure containing the `/bin` directory and `<arg>` is one of the following architecture arguments:

`ia32`	Compiler and libraries for IA-32 architecture only
`intel64`	Compiler and libraries for Intel® 64 architecture only

NOTE IA-32 is not available on macOS*.

Establishing a Performance Baseline

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

```
icc -O1 -std=c99 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.

Generating a Vectorization Report

A vectorization report shows what loops in your code were vectorized and explains why other loops were not vectorized. To generate a vectorization report, use the `qopt-report-phase=vec` compiler options together with `qopt-report=1` or `qopt-report=2`.

Together with `qopt-report-phase=vec`, `qopt-report=1` generates a report with the loops in your code that were vectorized while `qopt-report-phase=vec` with `qopt-report=2` generates a report with both the loops in your code that were vectorized and the reason that other loops were not vectorized.

Because vectorization is turned off with the `O1` option, the compiler does not generate a vectorization report. To generate a vectorization report, compile your project with the `O2`, `qopt-report-phase=vec`, `qopt-report=1` options:

```
icc -std=c99 -O2 -D NOFUNCCALL -qopt-report=1 -qopt-report-phase=vec Multiply.c
Driver.c -o MatVector
```

NOTE
We replace the call to the Matvec function with an inline equivalent with `-D NOFUNCCALL`.

Recompile the program and then execute `MatVector`. Record the new execution time. The reduction in time is mostly due to auto-vectorization of the inner loop at line 145 noted in the vectorization report `matvec.optrpt`:

```
LOOP BEGIN at Driver.c(140,5)
   remark #25460: No loop optimizations reported

   LOOP BEGIN at Driver.c(143,9)
      remark #25460: No loop optimizations reported

      LOOP BEGIN at Driver.c(145,13)
         remark #15300: LOOP WAS VECTORIZED
      LOOP END

      LOOP BEGIN at Driver.c(145,13)
      <Remainder loop for vectorization>
      LOOP END
   LOOP END
LOOP END
```

NOTE
Your line and column numbers may be different.

`qopt-report=2` with `qopt-report-phase=vec,loop` returns a list that also includes loops that were not vectorized or multi-versioned, along with the reason that the compiler did not vectorize them or multi-version the loop.

Recompile your project with the `qopt-report=2` and `qopt-report-phase=vec,loop` options.

```
icc -std=c99 -O2 -D NOFUNCCALL -qopt-report-phase=vec,loop -qopt-report=2 Multiply.c
Driver.c -o MatVector
```

The vectorization report `Multiply.optrpt` indicates that the loop at line 37 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 49 were generated, and one version was vectorized.

```
LOOP BEGIN at Multiply.c(37,5)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at Multiply.c(49,9)
   <Peeled loop for vectorization, Multiversioned v1>
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Multiversioned v1>
      remark #25228: Loop multiversioned for Data Dependence
      remark #15300: LOOP WAS VECTORIZED
   LOOP END
```

```
   LOOP BEGIN at Multiply.c(49,9)
   <Alternate Alignment Vectorized Loop, Multiversioned v1>
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Remainder loop for vectorization, Multiversioned v1>
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Multiversioned v2>
      remark #15304: loop was not vectorized: non-vectorizable loop instance from
multiversioning
      remark #25439: unrolled with remainder by 2
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Remainder, Multiversioned v2>
   LOOP END
```

NOTE

- Your line and column numbers may be different.
- For more information on the `qopt-report` and `qopt-report-phase` compiler options, see the *Compiler Options* section in the *Intel®C++ Compiler Developer Guide and Reference*.

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. Aliasing is not the only source of potential dependencies. In fact, `Multiply.c` does have other dependencies. In this case however, removal of the dependency created by aliasing allows the compiler to resolve the other loop dependency.

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.

Remove the `-D NOFUNCCALL` to restore the call to `matvec()`, then add the `-D NOALIAS` option to the command line.

```
icc -std=c99 -qopt-report=2 -qopt-report-phase=vec -D NOALIAS 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.optrpt` now shows:

```
LOOP BEGIN at Multiply.c(37,5)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at Multiply.c(49,9)
   <Peeled loop for vectorization>
```

```
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
      remark #15300: LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Alternate Alignment Vectorized Loop>
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Remainder loop for vectorization>
   LOOP END
LOOP END
```

NOTE Your line and column numbers may be different.

Now that the compiler has been told that the arrays do not overlap, it uses idiom-recognition to resolve the loop dependency and proceeds 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 `aligned 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. Use `-qopt-report=4` to see the change in aligned references.

```
icc -std=c99 -qopt-report=4 -qopt-report-phase=vec -D NOALIAS -D ALIGNED Multiply.c
Driver.c -o MatVector
```

`Multiply.optrpt` before adding the `#pragma vector aligned` shows:

```
LOOP BEGIN at Multiply.c(49,9)
   <Peeled loop for vectorization>
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
      remark #15388: vectorization support: reference a[i][j] has aligned access
[ Multiply.c(50,21) ]
      remark #15388: vectorization support: reference x[j] has aligned access
[ Multiply.c(50,31) ]
      remark #15305: vectorization support: vector length 2
      remark #15399: vectorization support: unroll factor set to 4
      remark #15309: vectorization support: normalized vectorization overhead 1.031
      remark #15300: LOOP WAS VECTORIZED
      remark #15442: entire loop may be executed in remainder
      remark #15448: unmasked aligned unit stride loads: 2
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 10
      remark #15477: vector cost: 4.000
      remark #15478: estimated potential speedup: 2.380
      remark #15488: --- end vector cost summary ---
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Alternate Alignment Vectorized Loop>
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Remainder loop for vectorization>
   LOOP END
```

And after adding `-D ALIGNED`:

```
LOOP BEGIN at Multiply.c(49,9)
      remark #15388: vectorization support: reference a[i][j] has aligned access
[ Multiply.c(50,21) ]
      remark #15388: vectorization support: reference x[j] has aligned access
[ Multiply.c(50,31) ]
      remark #15305: vectorization support: vector length 2
      remark #15399: vectorization support: unroll factor set to 4
      remark #15309: vectorization support: normalized vectorization overhead 0.594
      remark #15300: LOOP WAS VECTORIZED
      remark #15448: unmasked aligned unit stride loads: 2
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 10
      remark #15477: vector cost: 4.000
      remark #15478: estimated potential speedup: 2.410
      remark #15488: --- end vector cost summary ---
   LOOP END

   LOOP BEGIN at Multiply.c(49,9)
   <Remainder loop for vectorization>
      remark #15388: vectorization support: reference a[i][j] has aligned access
[ Multiply.c(50,21) ]
      remark #15388: vectorization support: reference x[j] has aligned access
[ Multiply.c(50,31) ]
      remark #15335: remainder loop was not vectorized: vectorization possible but seems
inefficient. Use vector always directive or -vec-threshold0 to override
      remark #15305: vectorization support: vector length 2
      remark #15309: vectorization support: normalized vectorization overhead 2.417
   LOOP END
```

NOTE Your line and column numbers may be different.

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 -qopt-report=2 -qopt-report-phase=vec -D NOALIAS -D ALIGNED -ipo
Multiply.c Driver.c -o MatVector
```

Note that the vectorization messages now appear at the point of inlining in `Driver.c` (line 150) and this is found in the file `ipo_out.optrpt`.

```
LOOP BEGIN at Driver.c(152,16)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at Multiply.c(37,5) inlined into Driver.c(150,9)
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at Multiply.c(49,9) inlined into Driver.c(150,9)
         remark #15300: LOOP WAS VECTORIZED
      LOOP END

      LOOP BEGIN at Multiply.c(49,9) inlined into Driver.c(150,9)
      <Remainder loop for vectorization>
         remark #15335: remainder loop was not vectorized: vectorization possible but seems
inefficient. Use vector always directive or -vec-threshold0 to override
      LOOP END
   LOOP END
LOOP END

LOOP BEGIN at Driver.c(74,5) inlined into Driver.c(159,5)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

NOTE Your line and column numbers may be different.

Now, run the executable and record the execution time.

Tutorial: Windows* Version

Use this section of the tutorial for Windows applications.

Preparing the Sample Application

In this tutorial, you will use the following files:

- `vec_samples_20xx.sln` or `vec_samples.vcproj`
- `Driver.c`
- `Multiply.c`
- `Multiply.h`

1. Start Microsoft* Visual Studio and open the solution file `vec_samples_20xx.sln`.
2. Convert to an Intel project by right-clicking on the `vec_samples` project and selecting **Intel Compiler > Use Intel C++**. Click **OK** in the **Confirmation** dialog.
3. Change the Active solution configuration to **Release** using **Build > Configuration Manager**.
4. 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**. The project property pages window appears.
2. Select **Configuration Properties** > **C/C++** > **Optimization** .
3. For **Optimization**, select **Minimum size** from the dropdown list.
4. For **Configuration Properties** > **C/C++** > **Optimization [Intel C++]** > **Interprocedural Optimization**, select **No**.
5. Select **Configuration Properties** > **C/C++** > **Language [Intel C++]**.

   For **Enable C99 Support**, select **Yes**.
6. Select **Configuration Properties** > **C/C++** > **Code Generation** .

   For **Floating Point Model**, select **Fast (/fp:fast)**.

   The `/fp:fast` option sets the compiler to aggressively optimize floating point arithmetic operations. Using the `/fp:precise` or `/fp:strict` options may limit opportunities for auto vectorization.
7. Rebuild the project, then run the executable (**Debug > Start Without Debugging**). 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 shows what loops in your code were vectorized and explains why other loops were not vectorized. To generate a vectorization report, use the `Qopt-report` and `Qopt-report-phase:vec` compiler options.

Together with `Qopt-report-phase:vec`, `Qopt-report:1` generates a report with the loops in your code that were vectorized while `Qopt-report:2` generates a report with both the loops in your code that were vectorized and the reason that other loops were not vectorized.

To use these options:

1. In your project's property pages select **Configuration Properties** > **C/C++** > **Diagnostics [Intel C++]**.
2. For **Optimization Diagnostics Level**, select **Level 1 (/Qopt-report:1)**.
3. For **Optimization Diagnostics Phase**, select **Vectorization (/Qopt-report-phase:vec)**.

Because vectorization is turned off with the `O1` option, the compiler does not generate a vectorization report. To generate a vectorization report, build your project with the `O2` option:

To set the `O2` option:

1. In your project's property pages select **Configuration Properties** > **C/C++** > **Optimization** .
2. For **Optimization**, select **Maximize Speed**.

For the purpose of showing the report, we'll also replace the call to `matvec()` in `Driver.c` with the equivalent C code by defining the preprocessor macro `NOFUNCCALL`. To do this, add `NOFUNCCALL` with a semicolon to the list of user defined macros at **Project** > **Properties** > **C/C++** > **Preprocessor** > **Preprocessor Definitions**.

Rebuild your project and then run the executable (**Debug > Start Without Debugging**). Record the new execution time. The reduction in time is mostly due to auto-vectorization of the inner loop at line 145 noted in the **Compiler Optimization Report** window, as well as in the `*.optrpt` files in the object directory.

For example, the following messages appear in `driver.optrpt`:

```
LOOP BEGIN at Driver.c(140,2)
Driver.c(145,2):remark #25460: No loop optimizations reported
```

```
LOOP BEGIN at Driver.c(143,3)
Driver.c(148,3):remark #25460: No loop optimizations reported

LOOP BEGIN at Driver.c(145,4)
Driver.c(150,4):remark #15300: LOOP WAS VECTORIZED
LOOP END
```

NOTE Your line and column numbers may be different.

The `Qopt-report:2` option returns a list that also includes loops that were not vectorized, along with the reason why the compiler did not vectorize them. Add the `Qopt-report:2` option in the same way you added `Qopt-report:1` above, instead selecting **Level 2 (/Qopt-report:2)**.

Rebuild your project.

The vectorization report indicates that the loop at line 37 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 49 were generated, and one version was vectorized.

The following messages appear in `Multiply.optrpt`:

```
LOOP BEGIN at Multiply.c(37,5)
Multiply.c(37,5):remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at Multiply.c(49,9)
Peeled loop for vectorization, Multiversioned v1
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Multiversioned v1
Multiply.c(49,9):remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Alternate Alignment Vectorized Loop, Multiversioned v1
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Remainder loop for vectorization, Multiversioned v1
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Multiversioned v2
Multiply.c(49,9):remark #15304: loop was not vectorized: non-vectorizable loop instance
from multiversioning
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Remainder, Multiversioned v2
LOOP END
LOOP END
```

NOTE

- Your line and column numbers may be different.
- For more information on the `Qopt-report` and `Qopt-report-phase` compiler options, 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. Aliasing is not the only source of potential dependencies. In fact, `Multiply.c` does have other dependencies. In this case however, removal of the dependency created by aliasing allows the compiler to resolve the other loop dependency.

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.

Remove the NOFUNCCALL preprocessor definition to reinsert the call to `matvec()`. Add the `NOALIAS` preprocessor definition to the compiler options.

Rebuild your project, run the executable, and record the execution time reported in the output. `Multiply.optrpt` shows:

```
LOOP BEGIN at Multiply.c(37,5)
Multiply.c(37,5):remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at Multiply.c(49,9)
Peeled loop for vectorization
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Multiply.c(49,9):remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Alternate Alignment Vectorized Loop
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Remainder loop for vectorization
LOOP END
LOOP END
```

> **NOTE** Your line and column numbers may be different.

Now that the compiler has been told that the arrays do not overlap, it uses idiom-recognition to resolve the loop dependency and proceeds 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 `__align declspec`, which has the following syntax:

```
__declspec(align(16)) float array[30];
```

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:

```
__declspec(align(16)) FTYPE a[ROW][COLWIDTH];
```

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.

Use `/Qopt-report:4` to see the report reflect the updated references (in your project's property pages select **Configuration Properties** > **C/C++** > **Diagnostics [Intel C++]** and for **Optimization Diagnostics Level**, select **Level 4 (/Qopt-report:4)**). Rebuild the program after adding the `ALIGNED` preprocessor definition to ensure consistently aligned data.

Multiply.optrpt before using #pragma vector aligned:

```
LOOP BEGIN at Multiply.c(49,9)
Peeled loop for vectorization
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Multiply.c(50,13):remark #15388: vectorization support: reference a[i][j] has aligned access
Multiply.c(50,13):remark #15388: vectorization support: reference x[j] has aligned access
Multiply.c(49,9):remark #15305: vectorization support: vector length 2
Multiply.c(49,9):remark #15399: vectorization support: unroll factor set to 4
Multiply.c(49,9):remark #15309: vectorization support: normalized vectorization overhead
1.031
Multiply.c(49,9):remark #15300: LOOP WAS VECTORIZED
Multiply.c(49,9):remark #15442: entire loop may be executed in remainder
Multiply.c(49,9):remark #15448: unmasked aligned unit stride loads: 2
Multiply.c(49,9):remark #15475: --- begin vector cost summary ---
Multiply.c(49,9):remark #15476: scalar cost: 10
Multiply.c(49,9):remark #15477: vector cost: 4.000
Multiply.c(49,9):remark #15478: estimated potential speedup: 2.380
Multiply.c(49,9):remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at Multiply.c(49,9)
```

```
Alternate Alignment Vectorized Loop
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Remainder loop for vectorization
LOOP END
```

Multiply.optrpt after adding `ALIGNED` to the preprocessor definitions:

```
LOOP BEGIN Multiply.c(49,9)
Multiply.c(50,13):remark #15388: vectorization support: reference a[i][j] has aligned access
Multiply.c(50,13):remark #15388: vectorization support: reference x[j] has aligned access
Multiply.c(49,9):remark #15305: vectorization support: vector length 2
Multiply.c(49,9):remark #15399: vectorization support: unroll factor set to 4
Multiply.c(49,9):remark #15309: vectorization support: normalized vectorization overhead
0.594
Multiply.c(49,9):remark #15300: LOOP WAS VECTORIZED
Multiply.c(49,9):remark #15448: unmasked aligned unit stride loads: 2
Multiply.c(49,9):remark #15475: --- begin vector cost summary ---
Multiply.c(49,9):remark #15476: scalar cost: 10
Multiply.c(49,9):remark #15477: vector cost: 4.000
Multiply.c(49,9):remark #15478: estimated potential speedup: 2.410
Multiply.c(49,9):remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at Multiply.c(49,9)
Remainder loop for vectorization
Multiply.c(50,13):remark #15388: vectorization support: reference a[i][j] has aligned access
Multiply.c(50,13):remark #15388: vectorization support: reference x[j] has aligned access
Multiply.c(49,9):remark #15335: remainder loop was not vectorized: vectorization possible
but seems inefficient. Use vector always directive or /Qvec-threshold0 to override
Multiply.c(49,9):remark #15305: vectorization support: vector length 2
Multiply.c(49,9):remark #15309: vectorization support: normalized vectorization overhead
2.417
LOOP END
```

NOTE Your line and column numbers may be different.

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 [Intel C++]** > **Interprocedural Optimization** > **Multi-file(/Qipo)**.

Note that the vectorization report now appears in `ipo_out.optrpt`.

```
LOOP BEGIN at Driver.c(152,9)
Driver.c(152,9):remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at Multiply.c(37,5) inlined into Driver.c(150,9)
Multiply.c(37,5):remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at Multiply.c(49,9) inlined into Driver.c(150,9)
Multiply.c(50,13):remark #15388: vectorization support: reference a[0][i][j] has aligned
access
Driver.c(150,9):remark #15388: vectorization support: reference x[j] has aligned access
Multiply.c(49,9):remark #15305: vectorization support: vector length 2
Multiply.c(49,9):remark #15399: vectorization support: unroll factor set to 4
```

```
Multiply.c(49,9):remark #15309: vectorization support: normalized vectorization overhead
0.594
Multiply.c(49,9):remark #15300: LOOP WAS VECTORIZED
Multiply.c(49,9):remark #15448: unmasked aligned unit stride loads: 2
Multiply.c(49,9):remark #15475: --- begin vector cost summary ---
Multiply.c(49,9):remark #15476: scalar cost: 9
Multiply.c(49,9):remark #15477: vector cost: 4.000
Multiply.c(49,9):remark #15478: estimated potential speedup: 2.000
Multiply.c(49,9):remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at Multiply.c(49,9) inlined into Driver.c(150,9)
Remainder loop for vectorization
Multiply.c(50,13):remark #15388: vectorization support: reference a[0][i][j] has aligned
access
Driver.c(150,9):remark #15388: vectorization support: reference x[j] has aligned access
Multiply.c(49,9):remark #15335: remainder loop was not vectorized: vectorization possible
but seems inefficient. Use vector always directive or /Qvec-threshold0 to override
Multiply.c(49,9):remark #15305: vectorization support: vector length 2
Multiply.c(49,9):remark #15309: vectorization support: normalized vectorization overhead
2.417
LOOP END
LOOP END
LOOP END
```

NOTE Your line and column numbers may be different.

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 that shows how the compiler can optimize performance with various vectorization techniques.

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
Notice revision #20201201