



Automatic Vectorization using Intel® Composer XE on the Linux* Operating System

Developer Product Division (DPD)

Disclaimer

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation ("Intel") and its contributors ("Contributors") on, as of the date of publication. Intel and the Contributors make no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT, IS PROVIDED "AS IS." NEITHER INTEL, NOR THE CONTRIBUTORS MAKE ANY REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL, THE CONTRIBUTORS, OR THIRD PARTIES. INTEL, AND ITS CONTRIBUTORS EXPRESSLY DISCLAIM ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL, AND ITS CONTRIBUTORS DO NOT WARRANT THAT THIS DOCUMENT IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL, AND ITS CONTRIBUTORS DISCLAIM ALL LIABILITY THEREFOR. INTEL, AND ITS CONTRIBUTORS DO NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIM ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel, its contributors and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate. Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity. LIMITED LIABILITY. IN NO EVENT SHALL INTEL, OR ITS CONTRIBUTORS HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL, OR ANY CONTRIBUTOR HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110228

Intel and Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2010, Intel Corporation. All Rights Reserved.

Time Required	45 minutes
Objective	<p>In this lab session, you will use Intel® Composer XE to get familiar with automatic vectorization</p> <p>After successfully completing this lab's activities, you will be able to:</p> <ul style="list-style-type: none"> • To select the right instruction set extension switch for compiling • To generate and analyze vectorization and optimization reports • To deal with memory aliasing and alignment issues potentially preventing vectorization • To understand the benefit of inter-procedural optimization for best vectorization results

Note: In these instructions we will use some switches like `-O2` explicitly when invoking the compiler although these switches might be set already by default. Doing it in this explicit way will make it more obvious what we are doing. Similar we might use a vectorization switch like `-msse2` explicitly while vectorization for SSE2 (`-msse2`) is enabled by default.

Activity 1.1 – Why Vectorization Failed

In this activity, you enable the compiler to generate diagnostic information on sample code that cannot be vectorized.

1. Navigate to the MatVector folder.
The Mat Vector example calculates the product of a matrix and a vector.
2. Complete the initial build without vectorization and execute the application. We use the switch "`-no-vec`" to explicitly disable automatic vectorization since optimization level `-O2` would enable automatic vectorization for Intel® SSE2 by default.

```
$ gcc -O2 -no-vec Driver.c Multiply.c -o MatVector
$ ./MatVector
```

Record execution time _____.

3. Enable automatic vectorization for SSE2

Automatic Vectorization

Student Workbook

3

Developer Product Division
Software and Service Group
© 2010 Intel® Corporation

```
$ gcc -O2 -msse2 Driver.c Multiply.c -o MatVector
```

```
$ ./MatVector
```

Record execution time _____

Optional: Does it make a difference in case the compiler could select SSE instructions from a more recent instructions set extensions than SSE2 like Intel® SSE4.2 ? Hint: Compare the assembler files (option -S) of both compilations.

4. Generate the vectorization report: Find out which loops in Driver.c and Multiply.c vectorized and which loops didn't vectorize:

```
$ gcc -msse2 -vec_report2 Driver.c Multiply.c -o MatVector
```

Do the messages make sense to you?

Only the loop nest of file Multiply.c is performance critical – we can ignore the other loops in Driver.c. Since outer loops cannot be vectorized, we can focus on the inner loop of this loop nest from now on.

Let's look at the dependencies preventing vectorization of the inner loop: We compile Driver.c separately:

```
$ gcc -msse2 -c Driver.c
```

```
$ gcc -msse2 -vec_report3 Driver.o Multiply.c -o MatVector
```

Verify for some of the dependencies displayed, that they are – based on the information accessible to the compiler - correct.

Activity 1.2 – Vectorize Inner Loop

1. Look at the source code of routine matvec() in Multiply.c. The compiler needs additional information to see, that the assumed dependencies are false dependencies. You have learned about multiple ways to do so. In two different steps use two of them: The 'restrict' keyword and the '-fargument-noalias' switch.

Record execution time for using '-fargument-noalias' switch _____

Record execution time for using 'restrict' keyword _____

2. The performance didn't improve / got even worse. Look at the matvec routine again to understand why: For the compiler, the loop index variables are not incremented by '1' in each iteration. You learned about "non-unit stride" challenges for vectorization. While a "non-unit" stride doesn't prevent vectorization, it results in rather inefficient code frequently. Look at Driver.c to see how inc_x and inc_y are set and simplify the code correspondingly. Recompile the code using -fargument-noalias .

Record execution time _____

3. Have a brief look at the assembler code of our critical routine 'matvec'. Generate the assembler file and rename the file:

```
$ gcc -O2 -msse2 -fargument-noalias /S Multiply.c
$ mv Multiply.s Multiply.s.1.2
```

Do you see the "versioning" for alignment done by the compiler ? Identify the memory load instructions used in the different versions of the loop.

Activity 1.3 – Alignment Improvements

In this activity, you will improve the performance of the code generated by aligning the data

1. Align a, b, and x arrays in Driver.c on a 16 byte boundary so that the vectorizer could use aligned loads rather than the slower unaligned loads (or splitting the loads). We learned that the 'aligned' attribute is one way to do this for Linux:

```
<array declaration> __attribute__((aligned(base))) // instructs the compiler to
enforce an alignment on a 'base'-byte boundary for the array listed
```

```
Example - FTYPE a[ROW][COLWIDTH] __attribute__((aligned(16)))
```

Compile and execute:

```
$ gcc -O2 -msse2 Driver.c Multiply.c -o MatVector -fargument-noalias
$ ./MatVector
```

Record execution time _____

Understand, that the improvement in execution time is due to the run-time check for alignment for the start addresses of the arrays. The compiler had no way to know about the new alignment when compiling Multiply.c since we didn't use IPO – inter-procedural optimization.

2. Edit Multiply.c so that the compiler can assume all SSE data moves are 16-byte aligned. This will make the run-time alignment check redundant. Insert the introduced directive just before inner loop:

```
#pragma vector aligned
```

Compile and execute:

```
$ icc -O2 -msse2 Driver.c Multiply.c -o MatVector -fargument-noalias  
$ ./MatVector
```

Record execution time _____ or any errant behavior _____

3. Look at the array declarations again – in particular in file Multiply.h – to understand why aligning the start addresses of the arrays is not sufficient to make '#pragma vector aligned' a correct assertion. Edit Multiply.h and set COLBUF=1; think about the effect

4. Compile and execute:

```
$ icc -O2 -msse2 Driver.c Multiply.c -o MatVector -fargument-noalias  
$ ./MatVector
```

Record execution time _____

5. Optional: Set COLBUF to '0' again. Instead of the directive "#pragma vector aligned", use the hint '__assume_aligned(...,16)' for arrays a, b and x. Compile, execute the application and note the execution time. Does this make sense to you ?

6. Optional: In case you are working on a system where unaligned moves of SSE data is similar fast as operations from aligned addresses (e.g. Intel® Core™ i7 processor) , investigate the implications for this test code:

Using -xsse4.2, the compiler will know, that the alignment advantages of this architecture can be exploited. Compile the code version of activity 1.2 (declarations not aligned in Driver.c, no hints in Multiply.c)

```
$ icc -O2 -xsse4.2 Driver.c Multiply.c -o MatVector -fargument-noalias  
$ ./MatVector
```

Record execution time _____

Could the compiler do better ?

Activity 2 – Vectorization of Complex Data Type

In this activity, you see that Intel C++ compiler supports complex number data type and complex number arithmetic operations.

1. Navigate to the 'Complex' folder.

2. Compile the initial build without vectorization and execute:

```
$ gcc -O2 -msse2 -no-vec complex.c multiply.c -o complex
$ ./complex
```

3. Use extension Intel® SSE3 and the appropriate vector report to determine whether the loop vectorized, or not.

```
$ gcc -O2 -msse3 -vec_report2 complex.c multiply.c -o complex
$ ./complex
```

4. Record execution time _____

5. Modify the source code to use the `_Complex` data type, to determine whether the loop vectorized, or not.

```
$ gcc -O2 -msse3 -std=c99 -vec_report2 complex.c multiply.c -o complex
$ ./complex
```

Record execution time _____

6. Does this work too for SSE2 ? Re-run last step using `-msse2` and explain difference.

Activity 3 – Using Inter-procedural Optimization

In this activity, we will see, that interprocedural optimization (IPO) can improve vectorization considerably. The application computes the electrostatic potential due to a uniform distribution over a square.

1. Navigate to the 'SquareCharge' folder. Then select either the 'c' or 'fortran' version depending on your preferences
2. Call 'make' to build the binary compiled without IPO and the one with IPO
3. Run sq_no_ipo and sq_with_ipo and compare the run times.
4. Look at the call of function 'trap_int' in the loop body of file twod_int.c (twod_int.f) to understand the benefit of IPO
5. Modify the Makefile to get the in-lining report from compiler: Add `-opt-report-phase ipo_inl` to compilation line for both cases. Re-build the binaries and look at the compiler report to understand, how the compiler reports a successful in-lining.