EXPLOIT CAPABILITIES WITHIN INTEL® XEON® PROCESSORS

An Introduction to Vectorization with the Intel® C++ Compiler

WHITE PAPER

Q: How do I take advantage of SSE and AVX instructions to speed up my code?

Introduction

This paper introduces vectorization and how C and C++ developers can take advantage of it. The reason to use vectorization is to create more efficient application processing that can increase application performance.

Vectorization techniques can be used by just about any application developer. The first forms of vectorization presented in this paper are those that are the easiest to use. They require no changes to code. Next are libraries, followed by compiler options that offer advice to the programmer on steps to take to deliver vectorization (this paper uses the Intel® C++ Compiler to exemplify these options). Additional topics are introduced that require more programmer intervention in source code and which offer the most programmer control, and frequently, a higher return in performance or efficiency. Here are the topics covered in this paper:

- Auto-vectorization capabilities of the Intel C++ Compiler
- Use of threaded and thread-safe libraries, such as Intel® Math Kernel Library (Intel® MKL) and Intel® Integrated Performance Primitives (Intel® IPP)
- Use of special compiler build-log reports to guide source code changes and use of pragmas
- Guided Auto-Parallelism in the Intel C++ Compiler
- Pragma SIMD statement
- Use of Intel® Cilk™ Plus array notation and elemental function syntax
- Intel SIMD intrinsics

Topics introduced in this paper apply to vectorizing code for Intel IA-32, Intel 64 and the upcoming Intel® MIC architecture. Thus, the vectorization you implement using the Intel C++ Compiler will scale over systems using current and future Intel processors.

Reading materials are mentioned throughout the paper and are presented in a list at the end of the paper.

What is Vectorization?

In computer science, vectorization is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process, where a single instruction can refer to a vector (series of adjacent values)[1] . In effect, it adds a form of parallelism to software in which one instruction or operation is applied to multiple pieces of data. When done on computing systems that support such actions, the benefit is more efficient processing and improved application performance. Many general-purpose microprocessors today feature multimedia extensions that support SIMD (single-instruction-multiple-data) parallelism. And when the hardware is coupled with C++ compilers that support it, developers of scientific, engineering, computational finance, media or graphical applications have an easier time delivering more efficient, better performing software[2].

Performance or efficiency benefits from vectorization depend on the code structure. But, in general, the automatic and near automatic techniques introduced below are most productive in delivering improved performance or efficiency. The techniques offering the most control require greater application knowledge and skill in knowing where they should be applied. But these more intrusive techniques, such as intrinsics, can yield potentially greater performance and efficiency benefit when properly used.

[1] A Guide to Vectorization with Intel® C++ Compilers, page 1, Mark Sabahi, et. al., Intel Corporation.
[2] Vectorization with the Intel Compilers, Intel Developer Services, page 1, Aart J.C. Bik, Intel Corporation.

A Good Way to Start: Intel® Compilers and the Auto-Vectorization Feature

Intel® C++ and Intel® Fortran compilers support SIMD by supporting the Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) on both IA-32 and Intel® 64 processors. Both compilers do auto-vectorization, generating Intel SIMD code to automatically vectorize parts of application software when certain conditions are met. Because no source code changes are required to use auto-vectorization, there is no impact on the portability of your application.

To take advantage of auto-vectorization, applications must be built at default optimization settings (-O2) or higher. No additional or special switch setting is needed. The compiler will automatically look for opportunities to execute multiple adjacent loop iterations in parallel using packed SIMD instructions[3]. If one or more loops have been vectorized, the compiler emits a remark to the build log that identifies the loop and says that the "LOOP WAS VECTORIZED."

When you use Intel compilers on systems that use Intel processors, you get 'free' performance improvements that will automatically take advantage of processing power as the Intel architecture gets more parallel. This is an example of what we mean by 'scaling forward.'

You can try the Intel compilers yourself by downloading an evaluation copy of an Intel compiler and testing it with the sample code included with the compiler[4] or with your own 'loopy' code. The Intel C++ Compiler features easy-to-use "Getting Started" guides that take you step-by-step through the use of the sample code and many compiler features, such as auto-vectorization.

Intel® MKL and Intel® IPP

Another easy way to take advantage of vectorization is to make calls in your applications to the vectorized forms of functions in the Intel® Math Kernel Library. Much of Intel MKL is threaded and supports auto-vectorization to help you get the most of today's multi-core processors. Intel MKL functions are also fully thread-safe, so multiple calls for different threads will not conflict with one another.

Similarly, Intel® Integrated Performance Primitives is another library for C and C++ developers, which also features vectorized functions. Intel® IPP offers libraries that can be called for multimedia, data processing, and communications applications.

Vectorization Reports and Pragmas

Intel compiler build-log reports contain two important kinds of information about vectorization. First, as noted above, they reports which loops were vectorized. Second, and perhaps more useful, an optional report provides information about why some loops were not vectorized. This can be very helpful in providing guidance to restructure code so it will auto-vectorize.

Figure 1. Sample source code and a sample report from the compiler indicating the loop was vectorized. More in "A Guide to Vectorization with Intel® C++ Compilers."

```
#include <math.h>
void quad(int length, float *a, float *b,
float *c, float *restrict x1, float *restrict x2)
  {
     for (int i=0; i<length; i++) {
        float s = b[i]*b[i] - 4*a[i]*c[i];
        if ( s >= 0 ) {
           s = sqrt(s) ;
           x2[i] = (-b[i]+s)/(2.*a[i]);
           x1[i] = (-b[i]-s)/(2.*a[i]);
        }
        else {
           x2[i] = 0.;
           x1[i] = 0.;
        }
     }
  }
> icc -c -restrict -vec-report2 quad.cpp
quad5.cpp(5) (col. 3): remark: LOOP WAS VECTORIZED.
```

[3] Op. cit., Sabahi, et. al., Intel Corporation

[4] After downloading and installing the compiler, find the "example 1" directory. The "Guide to Vectorization …" paper shows you the simple steps to take to see the performance benefit of vectorization.

Figure 2. Example of unvectorizable code with a sample report. More in "A Guide to Vectorization with Intel® C++ Compilers."

```
void no_vec(float a[], float b[], float c[])
{
   int i = 0.;
      while (i < 100) {
          a[i] = b[i] * c[i];
// this is a data-dependent exit condition:
          if (a[i] < 0.0)
              break;
          ++i;
   }
}

> icc -c -O2 -vec-report2 two_exits.cpp
two_exits.cpp(4) (col. 9): remark: loop was not
vectorized: nonstandard loop is not a
vectorization candidate.
```

These reports are also useful to help guide use and placement of the 45+ pragmas that can override assumptions made by the compiler. For developers familiar with their applications, these pragma statements make it easy to declare to the compiler that it is safe to ignore issues such as potential data dependencies. Other pragmas deal with explicit loop counts, allow developers to declare that a loop is safe to vectorize regardless of what the compiler thinks about the performance cost or benefit, and assert that data within the loop are aligned. There is also a statement to tell the compiler to not vectorize a loop and a compiler option to not do any vectorization. These can be useful for 'before' and 'after' performance and results testing.

Descriptions and examples of pragmas supported by the Intel C++ Compiler are provided in Intel C++ Compiler Pragmas section of the Intel® C++ Compiler XE User and Reference Guides.

Figure 3. An C++ example of the ivdep pragma

```
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) {
    cp_a[i] = cp_b[i];
  }
}
```

```
#pragma ivdep
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) {
    cp_a[i] = cp_b[i];
  }
}
```

The example in Figure 3 above makes the point that vectorizing compilers might assume memory regions accessed by the pointer variables cp_a and cp_b may (partially) overlap. A pragma (in lower box) can be used to tell the compiler that the loop won't do this. As it pertains to C++, you can read more in "A Guide to Vectorization with Intel® C++ Compilers, including Section 6.2.2 that provides more information about the *restrict* keyword.

Guided Auto-Parallelism (GAP)

The Intel C++ Compiler also includes an easy-to-use tool to help you vectorize code. It's called Guided Auto-Parallelism (GAP), which is invoked with the "/Qguide" option on Windows and "-guide" on Linux. This causes the compiler to generate diagnostic reports – but no object code or executable – that suggest ways to improve auto-vectorization as well as auto-parallelization and data layout. The advice may include suggestions for source code changes, applying specific pragmas, or applying specific compiler options. In all cases, applying specific advice requires the user to verify that it is safe to apply that particular suggestion.[5] This is a powerful tool to help you extend the auto-vectorization and auto-parallelism capabilities of the compiler for developers who are familiar with the code on which they are working.

Pragma SIMD

Yet another tool is user-mandated vectorization using the #pragma simd statement.[6] This is a feature that enables you to tell the compiler to enforce vectorization of loops. Pragma simd is designed to minimize the amount of source code changes needed in order to obtain vectorized code. It is related to the auto-vectorization discussion above in that it can be used to vectorize loops that the compiler does not normally auto-vectorize even using vectorization hints such as "#pragma vector always" or "#pragma ivdep". You add the pragma to a loop, recompile, and the loop is vectorized. This interesting feature alone makes reading the "Guide to Vectorization" worthwhile. It is listed in the reference section at the end of this paper.

[5] Op. cit, Sabahi, et. al., pg 25
[6] Op. Cit., A Guide to Vectorization with Intel® C++ Compilers, page 27

Intel® Cilk™ Plus: New, Powerful Vectorization Capability

Intel C++ offers more elements of programmer control in implementing vectorization and parallelism. The Intel C++ Compiler includes Intel® Cilk™ Plus, which introduces vector notation for arrays, also called array notation, supporting mathematical operations on arrays without constrained serial ordering, and elemental functions (vector functions) in which the compiler generates code to operate on a short vector of arguments.

Array notation syntax specifies an array section using a set of 3 numbers, either variable or literal, in an array syntax separated by colons[7]. The first is the lower bound where the array section starts, the second is the length of the array, and the third is the stride used to select items from the array. A stride of 1 will select every item from the lower bound contiguously until the array section limit is reached. Specifying a stride of 2 will select every other item.

Figure 4 shows a simple form of array notation. It's a simple vector multiplication, with a lower bound of 0, an array section length of N and an unspecified stride which defaults to 1. In this example, A_0 is set to the product of B_0 and C_0, A_1 is set to the product of B_1 and C_1, etc.

Figure 4. A simple example of array notation showing simple vector multiplication.

```
a[0:N] = b[0:N] * c[0:N];
```

Figure 5 shows a more complex example in which the 10th item of the array X, up to X of 100, is assigned a sin of every other alternating item of the array Y from 20 to 40.

Figure 5. A more sophisticated example of array notation

```
X[0:10:10] = sin(y[20:10:2]);
```

In Figure 5, the compiler knows that the call to the sin function can be done safely in parallel via vector operations. This enables the safe generation of vector code. But it can't assume the same for user functions.

If you have a function that consists of operations to scalar data that follow certain guidelines, you can declare your function an elemental function using the __declspec vector function notation as shown in Figure 6. See the Additional Reading section at the end of this paper for more material on elemental functions.

Figure 6. Example of elemental function syntax

```
__declspec(vector) int foo(int x)  {
        Return(x+1);
}
for(int I = 0; I < size; i++)
        array[i]=foo(array[i]);
```

Regarding elemental functions, it is important that both the function definition and any function declaration use this notation consistently. Doing this will enable the compiler to emit vector code to call multiple elements of foo in an array at a given point in time.

Intrinsics

For C++ developers interested in even more control, Intel offers SIMD intrinsics, vector intrinsics and a large number of other types of intrinsics. Intrinsics are assembly-coded functions that allow you to use C++ function calls and variables in place of assembly instructions. They are expanded inline eliminating function call overhead and thus can improve application performance for those interested in getting quite close to the hardware. They provide the same benefit as using inline assembly, can improve code readability, assist instruction scheduling, and help reduce debugging, relative to using assembly code. Intrinsics are for 'deep-dive' developers interested in squeezing out that last measure of application performance. For more information, please consult the "Overview: Intrinsics Reference" section of the Intel C++ documentation.

It may also be useful to know that using pragmas, directives or intrinsics may vectorize your code but may not lead to enhanced performance. Intel offers an informative, short paper that lays out the requirements for loop vectorization called "Requirements for Vectorized Loops."

[7] Introduction to Intel® Cilk™ Plus, Brandon L. Hewitt. The description in this paragraph, including the examples, are from a 6 minute video that introduces Intel Cilk Plus.

Summary

Other development products from Intel can also help with vectorization and other forms of parallelism. Intel® VTune™ Amplifier XE can help analyze code to find performance bottlenecks and Intel® Inspector XE can help debug parallel code to verify threading correctness.

The performance benefits from vectorization and parallelism can be significant. Intel Software Development products offer flexible capabilities that enable tapping into this performance, some of which are automatic, others that are easy to use and still more that offer extensive programmer control. This paper offers quick survey of these capabilities. Take the time to download Intel Software Development tools, evaluate them, and see for yourself how you can take advantage of parallelism in contemporary computing systems.

Additional Reading and Community

A Guide to Vectorization with Intel® C++ Compilers, Mario Deilmann, Kiefer Kuah, Martyn Corden, Mark Sabahi, all from Intel.

Vectorization with the Intel® Compilers (Part 1), A.J.C Bik, Intel, Intel Software Network Knowledge base and search the title in the keyword search. This article offers good bibliographical references.

The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance, A.J.C. Bik. Intel Press, June, 2004, for a detailed discussion of how to vectorize code using the Intel® compiler.

Vectorization: Writing C/C++ code in VECTOR Format, Mukkaysh Srivastav, Computational Research Laboratories (CRL) - Pune, India. Intel Software Network Knowledge base and search the title in the keyword search

Intel® Cilk™ Plus Introductory Information. Overviews, videos, getting started guide, documentation, white papers and a link to the community.

Elemental functions: Writing data parallel code in C/C++ using Intel® Cilk™ Plus. Robert Geva, Intel Corporation

Intel® C++ Composer XE documentation, Includes documentation for the Intel® C++ Compiler.

Intel Software Network, Search for topics such as "Parallel Programming in the "Communities" menu or "Software Forums" or Knowledge Base in the "Forums and Support" menu.

Requirements for Vectorizable Loops, Martyn Corden, Intel Corporation

The Software Optimization Cookbook, Second Edition, High-Performance Recipes for IA-32 Platforms by Richard Gerber, Aart J.C. Bik, Kevin B. Smith and Xinmin Tian, Intel Press.

Purchase Options: Language Specific Suites

Several suites are available combining the tools to build, verify and tune your application. Single or multi-user licenses and volume, academic, and student discounts are available.

	Suites >>	Intel® Parallel Studio XE	Intel® C++ Studio XE	Intel® Fortran Studio XE	Intel® Cluster Studio XE	Intel® Composer XE	Intel® C++ Composer XE	Intel® Fortran Composer XE
Components	Intel® C / C++ Compiler	●	●		●	●	●	
	Intel® Fortran Compiler	●		●	●	●		●
	Intel® Integrated Performance Primitives[3]	●	●		●	●	●	
	Intel® Math Kernel Library[3]	●	●	●	●	●	●	●
	Intel® Cilk™ Plus	●	●		●	●	●	
	Intel® Threading Building Blocks	●	●		●	●	●	
	Intel® Inspector XE	●	●	●	●			
	Intel® VTune™ Amplifier XE	●	●	●	●			
	Static Security Analysis	●	●	●	●			
	Intel® MPI Library				●			
	Intel® Trace Analyzer & Collector				●			
	Rogue Wave IMSL* Library[2]							●
	Operating System[1]	W, L	W, L	W, L	W, L	W, L	W, L, M	W, L, M

Note: (1)[1] Operating System: W=Windows, L= Linux, M= Mac OS X. (2)[2] Available in Intel® Visual Fortran Composer XE for Windows with IMSL*(3)[3] Not available individually on Mac OS X, it is included in Intel® C++ & Fortran Composer XE suites for Mac OS X*

Evaluate a tool

Download a free evaluation copy of our tools. If you're still uncertain where to begin, we suggest:

1. For suites that include the compiler and libraries along with analysis tools, try Intel Parallel Studio XE or Intel Cluster Studio XE (if you use MPI clusters)
2. If you are not interested in analysis tools, Intel® Composer XE combines the Intel compilers with libraries.

Learning Tools

- Intel® C++ Composer XE 2011 Getting Started Tutorials
  - Windows: http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/start/win/tutorial_comp_cpp_win.pdf
  - Linux: http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/start/lin/tutorial_comp_cpp_lin.pdf
  - Mac OS X: http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/start/mac/tutorial_comp_cpp_mac.pdf
- Intel Learning Lab, collection of tutorials, white papers and more.

About the Author

Chuck Piper is an Intel Product Marketing Engineer specializing in compilers.

Notices

Optimization Notice

Notice revision #20110804

Intel's compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.