EXPLOIT CAPABILITIES WITHIN INTEL® XEON® PROCESSORS

An Introduction to Vectorization with the Intel® Fortran Compiler

WHITE PAPER

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

Introduction

This paper defines vectorization and introduces how developers using Fortran can take advantage of it. The reason to use vectorization is typically related to an interest in increasing application performance and creating more efficient application processing.

The paper introduces vectorization techniques that can be used by just about any application developer and uses the Intel® Fortran Compiler to exemplify these uses. 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. 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 vectorization topics mentioned in this paper:

- Auto-vectorization capabilities of the Intel® Fortran Compiler
- Use of threaded and thread-safe libraries, such as Intel® Math Kernel Library (Intel® MKL)
- Use of special compiler build-log reports to guide source code changes and use of pragmas
- Guided Auto-Parallelism in the Intel® Fortran Compiler
- SIMD compiler directive

Topics introduced in this paper apply to vectorizing code for IA-32, Intel® 64 and the upcoming Intel® MIC architectures. Thus, the vectorization you implement using the Intel® Fortran 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 Fortran compilers that support it, developers of scientific and engineering 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 those that may involve compiler directives or other source code changes, 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 architecture 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 -O2) or higher. Add the /Qvec-report1 (-vec-report1) to have the compiler tell you when it vectorized a loop. With these settings, the compiler will 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® Fortran Compiler feature 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.

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

[4] The compiler includes a "Getting Started" tutorial and sample code. If you do the default installation (in this case, on Windows), samples are located in C:\Program Files (x86)\Intel\Composer XE 2011 SP1\Samples\en_US\Fortran\vec_samples.zip.

Intel® MKL

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 (Intel® MKL). Intel® MKL offers linear algebra functions, implemented in LAPACK (solvers and eigensolvers) plus level 1, 2, and 3 BLAS, offering the vector, vector-matrix, and matrix-matrix operations needed for complex mathematical software. A set of vectorized transcendental functions called the Vector Math Library (VML) is also included. These offer greater performance than the libm (scalar) functions, while maintaining the same high accuracy. The Vector Statistical Library (VSL) offers high performance vectorized random number generators for several probability distributions, convolution and correlation routines, and summary statistics functions.

Vectorization Reports

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 (/Qvec-report2 or –vec-report2) 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 followed by a command line to start the Fortran compiler, and a sample report from the compiler indicating the loop was vectorized.

```
subroutine quad(len,a,b,c,x1,x2)
  real(4) a(len),b(len), c(len), x1(len), x2(len), s

  do i=1,len
    s = b(i)**2 - 4.*a(i)*c(i)
    if (s.ge.0.) then
      x1(i) = sqrt(s)
      x2(i) = (-x1(i) - b(i)) *0.5 / a(i)
      x1(i) = ( x1(i) - b(i)) *0.5 / a(i)
    else
      x2(i)=0.
      x1(i)=0.
    endif
  enddo
end


> ifort -c -vec-report2 quad.f90
quad.f90(4): (col. 3) remark: LOOP WAS VECTORIZED.
```

Figure 2. Similar to Figure 1 but, in this case, it's an example of unvectorizable code with a sample report.

```
subroutine no_vec(a, b, c)
  real(4), dimension(*) :: a, b, c
  integer :: i

  do i=1,100
     a(i) = b(i) * c(i)
     if (a(i) < 0.0 )  exit
  enddo

end

> ifort -c -vec-report2 two_exits.f90
two_exits.f90(5): (col. 3) remark: loop was not
vectorized: nonstandard loop is not a vectorization
candidate.
```

Directives

The reports are also useful to help guide use and placement of the many directives included in the Intel® Fortran compiler, not including OpenMP* directives, that can override assumptions made by the compiler. For developers familiar with their applications, directives make it easy to declare to the compiler that it is safe to ignore issues such as potential data dependencies. Other directives deal with 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 Fortran Compiler are provided in the Intel® Fortran Compiler XE 12.1 User and Reference Guides (search for "Compiler Directives").

The IVDEP directive is applied to a DO loop in which the user knows that dependences are in lexical order. For example, if two memory references in the loop touch the same memory location and one of them modifies the memory location, then the first reference to touch the location has to be the one that appears earlier lexically in the program source code. This assumes that the right-hand side of an assignment statement is "earlier" than the left-hand side.

The IVDEP directive informs the compiler that the program would behave correctly if the statements were executed in certain orders other than the sequential execution order, such as executing the first statement or block to completion for all iterations, then the next statement or block for all iterations, and so forth. The optimizer can use this information, along with whatever else it can prove about the dependences, to choose other execution orders.

Guided Auto-Parallelism (GAP)

The Intel® Fortran 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 executables – 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.

SIMD Directive

Yet another tool is user-mandated vectorization using the SIMD directive. This is a feature that enables you to tell the compiler to enforce vectorization of loops. Programs written with SIMD vectorization are very similar to those written using auto-vectorization hints. You can use SIMD vectorization to minimize code changes that you may have to go through in order to obtain vectorized code.

SIMD vectorization uses the !DIR$ SIMD directive to effect loop vectorization. The options –Qsimd- [on Windows*] or –no-simd [on Linux* or Mac* OS] may be used to disable any SIMD directives, for testing and comparisons.

[5] Op. cit, Sabahi, et. al., pg 25

The following example in Figures 3 and 4 show an example using code that does not automatically vectorize the due to the unknown data dependence distance "X". You can use the data dependence assertion via the auto-vectorization hint, !DIR$ IVDEP, to let the compiler decide to vectorize the loop or not, or you can enforce vectorization of the loop using !DIR$ SIMD.

Figure 3. Example: without !DIR$ SIMD produces the output at the bottom of the figure.

```
[D:/simd] cat example1.f
subroutine add(A, N, X)
integer N, X
real       A(N)
DO I=X+1, N
    A(I) = A(I) + A(I-X)
ENDDO
end

Command line entry: [D:/simd] ifort example1.f -
nologo -Qvec-report2

Output:  D:\simd\example1.f(6): (col. 9) remark:
loop was not vectorized: existence of vector
dependence.
```

Figure 4. Example with !DIR$ SIMD produces "LOOP WAS VECTORIZED" report.

```
[D:/simd] cat example1.f
subroutine add(A, N, X)
integer N, X
real       A(N)
!DIR$ SIMD
DO I=X+1, N
    A(I) = A(I) + A(I-X)
ENDDO
end

Command line entry:  [D:\simd] ifort example1.f
-nologo -Qvec-report2
Output:  D:\simd\example1.f(7): (col. 9) remark:
LOOP WAS VECTORIZED.
```

The SIMD directive has optional clauses to guide the compiler on how vectorization must proceed. An expert user might employ these clauses to further guide how the compiler goes about vectorization. In most simple situations, they are not needed. For more information, consult the Intel® Fortran Compiler XE 12.1 User and Reference Guides (search "Directive SIMD").

Summary

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 the tools, evaluate them, and see for yourself how you can take advantage of vectorization in contemporary computing systems.

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.

Additional Reading and Community

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.

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

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.

Evaluate a tool

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

For bundled 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). If you are not interested in analysis tools, Intel® Composer XE combines the Intel compilers with libraries. Try Intel® Parallel Advisor for Windows* to help identify where you code can benefit from parallelism.

Learning Tools

- Intel® Visual Fortran Composer XE 2011 Getting Started Tutorials
    - For Windows
    - For Linux
    - For Mac OS X
- Intel Learning Lab, collection of tutorials, white papers and more.

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*

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.