DEVELOPER ROCK STARS:
Shannon Cepeda | Wendy Doerner

Help Future-Proof Performance of Your Application with

VECTORIZATION in Six Steps

by Wendy Doerner and Shannon Cepeda,
Software Technical Consulting Engineers, Intel

Learn how to get a significant performance boost from processing data in parallel inside a single CPU core. Then, combine with threading and/or cluster parallelism for even more performance.

The Parallel Universe has featured many articles about parallelism aimed at exploiting multiple cores through threading and cluster programming. This article takes a look at exploiting an equally important form of parallelism known as vector parallelism. Vectorization is parallelism *within a single CPU core* and is a key form of hardware support for data parallelism.

With vectorization, certain operations can be performed on multiple pieces of data at once. It is accomplished by using special instructions called (SIMD) Single Instruction, Multiple Data operations. SIMD instructions, and the hardware that goes along with them, has been present in Intel® processors for over a decade. Some examples of our SIMD instructions sets are Intel® Streaming SIMD Extensions (Intel® SSE), first introduced in 1999 and expanded several times, and Intel® Advanced Vector Extensions (Intel® AVX), introduced last year.

How Does Vectorization Work?

In the typical scalar (non-vectorized) case, each variable you use will each be stored in its own CPU register. If you perform an operation on two variables, such as addition, the two register quantities are added and the result stored back into a register. The vectorized version of this example would first fill a register with multiple variables to be added, which is called "packing" the register. For example, on processors supporting Intel AVX, up to eight single, precision, floating-point data elements can be packed into one register. Then, using one SIMD instruction, these data elements can be combined with another packed register full of elements, generating multiple results at once. Being able to do these operations in parallel rather than separately can result in significant performance gains for suitable code.

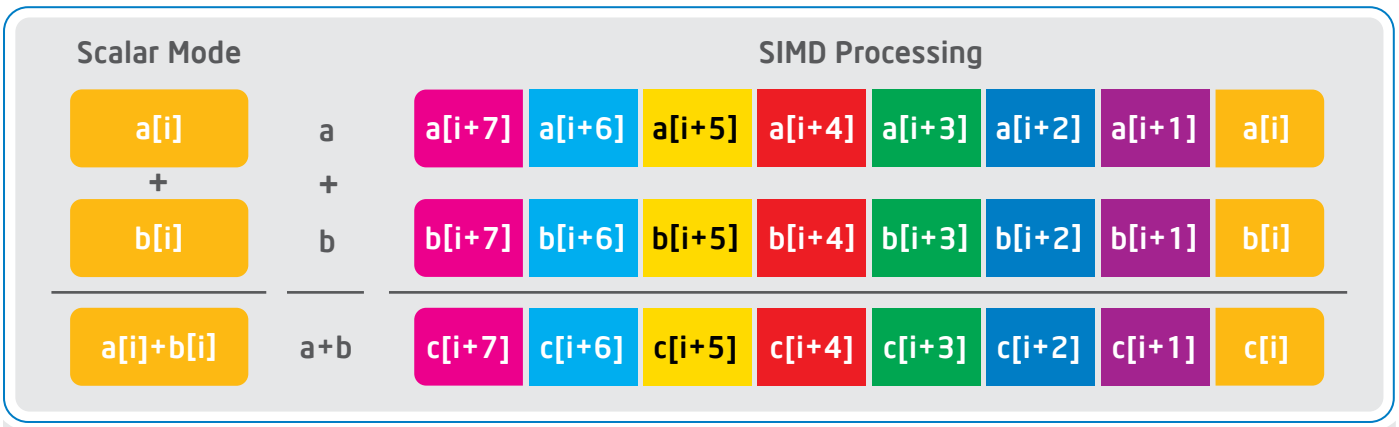

Figure 1: A SIMD addition applied to array elements

"Vectorization allows you to process data in parallel within a single CPU core. Alone, it can provide a significant performance boost and it can also be combined with threading and/or cluster parallelism... Using one of our recommended methods to vectorize your code gives you two advantages: increased performance now, and forward scaling for the future."

As **Figure 1** shows, vectorization is typically applied to array or vector data elements that are processed in loops. Besides addition, there are SIMD instructions for many mathematical functions, logical operations, and even string operations. The SIMD instruction sets are designed for applications that process large data sets. Applications in the scientific, engineering, financial, media, and graphical domain areas may be candidates for vectorization, as well as any others fitting the description above.

How Do I Vectorize My Code?

Developers can access the SIMD instructions in their applications in a variety of ways. Traditionally, vectorization has been accomplished by manipulating SIMD instructions and registers directly, using assembly code or intrinsics provided by the compiler. This method required developers to become experts in SIMD architecture and to hand-tune the code for various CPUs. In addition to requiring significant effort to develop and maintain the code, vectorizing using assembly or intrinsics also has the disadvantage that code is not portable across compilers. Because code must be written for a targeted set of SIMD hardware, the vectorization achieved with this method would *not* scale forward, meaning it must be re-implemented for new CPUs.

Fortunately, today you can choose from several other vectorization methods that require less effort and *do* scale forward. The techniques below rely on a vectorizing compiler, and we recommend our Intel® Compiler, available for C++ or Fortran. The Intel Compiler includes an advanced vectorizer, as well as several options, reports, and extensions to support vectorization. Our "six steps to vectorization" process utilizes several of these features.

Methods that Scale Forward

When you vectorize your code with one of the methods below, it will be vectorizable without changes for future architectures and CPUs as well. When compiling on those future architectures, the Intel Compiler (or a compiler fully supporting the method) will make the appropriate choice about how to vectorize.

- Using the Intel Compiler auto-vectorizer: When enabled, the compiler auto-vectorizer will look for opportunities to vectorize loops with no changes required to your source code. This method may be all that is needed for vectorization-friendly code. However, the compiler will not vectorize loops if it can't prove that it will be safe to perform the operations in parallel. For this reason, you may see even more vectorization by following up this method with one of the techniques below.
- Using a high-level construct provided by Intel® Cilk™ Plus: Cilk Plus is a set of language extensions for C and C++ (and in one case, Fortran) that support parallelism and vectorization. Currently Cilk Plus is fully supported by the Intel Compiler, and partially implemented in GCC. It is an open standard (for more details, see cilk.com). Cilk Plus provides a variety of constructs that can be applied to your code to give the compiler information that it needs to vectorize. Many of these constructs are simple to add to your code—involving only a change in notation or the addition of a pragma.
- Using a high-level Fortran construct: Fortran includes several vectorization-friendly features, such as array notations and the DO CONCURRENT loop. The Cilk Plus mandatory vectorization directive (!DIR$ SIMD) is also available for Fortran. Like Cilk Plus, these constructs are used to give information to the compiler so that it knows when it can vectorize.

BLOG highlights

Serial Equivalence of Cilk Plus programs

ROBERT GEVA, Intel

There is a trend in the C++ community to grow capabilities thru more libraries and as much as possible, avoid adding language keywords. Consistent with these trends are the Intel® Threading Building Blocks and the Microsoft Parallel Patterns Library.* The question arises, then, why implement Intel® Cilk™ Plus as language extensions rather than a library?

One of the answers is that the language is implemented by compilers, and compilers can provide certain guarantees. One such guarantee is serial equivalence. Every Cilk Plus program that uses the three taking keywords for parallelism has a well-defined serial elision. The serial elision is defined by replacing each cilk_spawn and each cilk_sync with white spaces, and each cilk_for with the for keyword. Obviously, the serial elision of a Cilk Plus program is a valid C/C++ program.

A program has a determinacy race if two logically parallel strands both access the same memory location and at least one of them modifies the memory location. If a Cilk Plus parallel program has no determinacy race, then it will produce the same results as its serial elision. What are the compiler's contributions to the serial equivalence guarantees? Consider the following code illustration ...

SEE THE REST OF ROBERT'S BLOG:

Visit **Go-Parallel.com**

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

The Six-Step Process for Vectorizing Your Application

The methods above, and the Intel Compiler are part of a six-step process we have designed for vectorizing an application. You can try this method on your entire application, or selectively on parts of it. To help you follow the process we have documented the steps online in our *Vectorization Toolkit*. The toolkit also includes links to additional resources for each step. Check out the toolkit online at: http://software.intel.com/en-us/articles/vectorization-toolkit/.

Step 1. Measure Baseline Release Build Performance

You need to have a baseline for performance so you know if changes to introduce vectorization are effective. In addition, you should have a baseline to set your ultimate performance goals, so that you know when you have achieved them.

A release build should be used instead of a debug build. A release build will contain all the optimizations in your final application, and may alter the hotspots or even the code that is executed. For instance, a release build may optimize away a loop in a hotspot that otherwise would be a candidate for vectorization.

A release build is the default in the Intel Compiler. You have to specifically turn off optimizations by doing a DEBUG build on Windows* (or using the -Zi switch) or using the -Od switch on Linux* or Mac OS* X. If using the Intel Compiler, ensure you are using optimization levels 2 or 3 (-O2 or –O3) to enable the auto-vectorizer.

Step 2. Determine Hotspots Using Intel® VTune™ Amplifier XE

You can use Intel® VTune™ Amplifier XE, our performance profiler, to find the most time-consuming functions in your application. The "Hotspots" analysis type is recommended; although "Lightweight Hotspots" would work as well (it will profile the whole system, as opposed to just your application).

Identifying which areas of your code are taking the most time will allow you to focus your optimization efforts in the areas where performance improvements will have the most effect. Generally you want to focus only on the top few hotspots, or functions taking at least 10% of your application's total time. Make note of the hotspots you want to focus on for the next step.

Step 3. Determine Loop Candidates Using Intel Compiler Vec-Report

The vectorization report (or vec-report) of the Intel Compiler can tell you whether or not each loop in your code was vectorized. Ensure that you are using Compiler optimization level 2 or 3 (-O2 or –O3) to enable the auto-vectorizer. Run the vec-report and look at the output for the hotspots you determined in **Step 2**. If there are loops in your hotspots that did not vectorize, check whether they have math, data processing, or string calculations on data in parallel (for instance in an array). If they do, they might benefit from vectorization. Move to **Step 4** if any candidates are found.

To run the vec-report, use the "-vec-report2" or "/Qvec-report2" option.

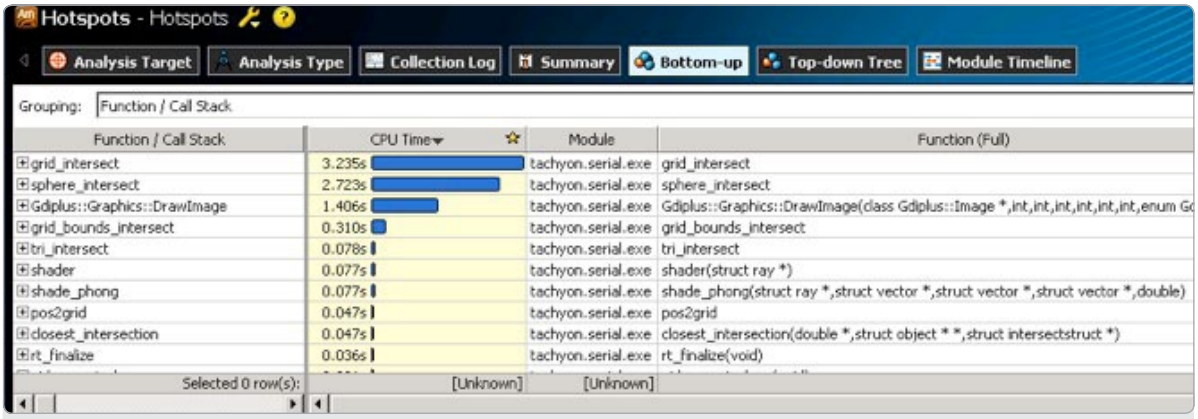

Figure 2: Intel® VTune™ Amplifier XE Hotspots view

```
.\main.cpp(30): warning : LOOP WAS VECTORIZED.
.\scalar_dep.cpp(80): warning : LOOP WAS VECTORIZED.
.\main.cpp(47): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: nonstandard loop is not a vectorization candidate.
.): warning : loop was not vectorized: nonstandard loop is not a vectorization candidate.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
```

Figure 3: Output from the Intel® Compiler vec-report

Note that the Intel Compiler can be run on just a portion of the code and will be compatible with the native compilers (gcc on Linux and Mac OS X and Microsoft Visual Studio* on Windows).

Step 4. Get Advice Using the Intel Compiler GAP Report and Toolkit Resources

Run the Intel Compiler Guided Auto-parallelization (or GAP) report to see suggestions from the compiler on how to vectorize your loop candidates from Step 3. Examine the advice and refer to additional toolkit resources as needed.

Run the GAP report using the "guide" or "/Qguide" options for the Intel Compiler.

Note: You can run the Intel Compiler on just parts of your application if needed.

```
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;}
}
```

Figure 4: Example of a nonvectorizing loop

```
scalar_dep.cpp(80): warning #30515: (VECT)
Assign a value to the variable(s) "b" at
the beginning of the body of the loop in
line 80. This will allow the loop to be
vectorized. [VERIFY] Make sure that, in the
original program, the variable(s) "b"
read in any iteration of the loop has been
defined earlier in the same iteration.
```

Figure 5: GAP advice for the loop in Figure 4

Step 5. Implement GAP Advice and Other Suggestions (Such as Using Elemental Functions and/or Array Notations)

Now that you know the GAP report suggestions for your loop, it's time to implement them if possible.

The report may suggest making a code change. Make sure the change would be "safe" to do. In other words, make sure the change does not affect the semantics or safety of your loop. One way to ensure that the loop has no dependencies that may be affected is to see if executing the loop in backwards order would change the results. Another is to think about the calculations in your loop being in a scrambled order. If the results would be changed, your loop has dependencies and vectorization would not be "safe." You may still be able to vectorize by eliminating dependencies in this case.

Modify your source to give additional information to the Compiler or optimize your loop for better vectorization.

At this point you may introduce some of the high-level constructs provided by Cilk Plus or Fortran. You can find links to full details of the available constructs in the online Vectorization Toolkit.

Step 6: Repeat!

Iterate through the process as needed until performance is achieved or until there are no good candidates left in your hotspots.

```
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;}
}
```

Figure 6: The loop from Figure 4, modified to vectorize

Conclusion

Vectorization allows you to process data in parallel within a single CPU core. Alone, it can provide a significant performance boost and it can also be combined with threading and/or cluster parallelism. Using vectorization is important for performance on current Intel CPUs, such as those in the Intel® Xeon® and Intel® Core™ processor families. In the future, it will be an even more critical component of performance for those processors, as well as for utilizing the Intel® Many Integrated Core architecture.

Using one of our recommended methods to vectorize your code gives you two advantages: increased performance now, and forward scaling for the future. Please visit our Vectorization Toolkit to see the latest advice, processes, and resources to help with your vectorization effort. See the webinar: Future-Proof Your Application's Performance ›

Optimization Notice

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.

Notice revision #20110804