Using AVX without Writing AVX Code

<keywords: Intel Advanced Vector Extensions, AVX, vectorization, auto-vectorization, Intel MKL, Intel IPP, SIMD>

Abstract

Intel® Advanced Vector Extensions (Intel® AVX) is a new 256-bit instruction set extension to Intel® Streaming SIMD Extensions (Intel® SSE) and is designed for applications that are Floating Point (FP) intensive. Intel® SSE and Intel® AVX are both examples of Single Instruction Multiple Data instruction sets. Intel® AVX was released as part of the 2nd generation Intel® Core™ processor family. Intel® AVX improves performance due to wider 256-bit vectors, a new extensible instruction format (Vector Extension or VEX), and by its rich functionality.

The instruction set architecture supports three operands which improves instruction programming flexibility and allows for non-destructive source operands. Legacy 128-bit SIMD instructions have also been extended to support three operands and the new instruction encoding format (VEX). An instruction encoding format describes the way that higher-level instructions are expressed in a format the processor understands using opcodes and prefixes. This results in better management of data and general purpose applications like those for image, audio/video processing, scientific simulations, financial analytics and 3D modeling and analysis.

This paper discusses options that developers can choose from to integrate Intel® AVX into their applications without explicitly coding in low-level assembly language. The most direct way that a C/C++ developer can access the features of Intel® AVX is to use the C-compatible intrinsic instructions. The intrinsic functions provide access to the Intel® AVX instruction set and to higher-level math functions in the Intel® Short Vector Math Library (SVML). These functions are declared in the immintrin.h and ia32intrin.h header files respectively. There are several other ways that an application programmer can utilize Intel® AVX without explicitly adding Intel® AVX instructions to their source code. This document presents a survey of these methods using the Intel® C++ Composer XE 2011 targeting execution on a Sandy Bridge system. The Intel® C++ Composer XE is supported on Linux*, Windows*, and Mac OS* X platforms. Command line switches for the Windows* platform will be used throughout the discussion.

This article is part of the larger series, "Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

A vector or SIMD enabled-processor can simultaneously execute an operation on multiple data operands in a single instruction. An operation performed on a single number by another single number to produce a single result is considered a scalar process. An operation performed simultaneously on N numbers to produce N results is a vector process (N > 1). This technology is available on Intel processors or compatible, non-Intel processors that support SIMD or AVX instructions. The process of converting an algorithm from a scalar to vector implementation is called vectorization.

Advice

Recompile for Intel® AVX

The first method to consider is to simply recompile using the /QaxAVX compiler switch. The source code does not have to be modified at all. The Intel® Compiler will generate appropriate 128 and 256-bit Intel® AVX VEX-encoded instructions. The Intel® Compiler will generate

multiple, processor-specific, auto-dispatched code paths for Intel processors when there is a performance benefit. The most appropriate code will be executed at run time.

Compiler Auto-vectorization

Compiling the application with the appropriate architecture switch is a great first step toward building Intel® AVX ready applications. The compiler can do much of the vectorization work on behalf of the software developer via auto-vectorization. Auto-vectorization is an optimization performed by compilers when certain conditions are met. The Intel® C++ Compiler can perform the appropriate vectorization automatically during code generation. An excellent document that describes vectorization in more detail can be found at A Guide to Vectorization with Intel(R) C++ Compilers. The Intel Compiler will look for vectorization opportunities whenever the optimization level is /O2 or higher.

Let's consider a simple matrix-vector multiplication example that is provided with the Intel® C++ Composer XE that illustrates the concepts of vectorization. The following code snippet is from the matvec function in Multiply.c of the vec_samples archive:

```
void matvec(int size1, int size2, FTYPE a[][size2], FTYPE b[], FTYPE x[])
{
	for (i = 0; i < size1; i++) {
		b[i] = 0;

		for (j = 0;j < size2; j++) {
			b[i] += a[i][j] * x[j];
		}
	}
}
```

Without vectorization, the outer loop will execute size1 times and the inner loop will execute size1*size2 times. After vectorization with the /QaxAVX switch the inner loop can be unrolled because four multiplications and four additions can be performed in a single instruction per operation. The vectorized loops are much more efficient than the scalar loop. The advantage of Intel® AVX also applies to single-precision floating point numbers as eight single-precision floating point operands can be held in a ymm register.

Loops must meet certain criteria in order to be vectorized. The loop trip count has to be known when entering the loop at runtime. The trip count can be a variable, but it must be constant while executing the loop. Loops have to have a single entry and single exit, and exit cannot be dependent on input data. There are some branching criteria as well, e.g., switch statements are not allowed. If-statements are allowed provided that they can be implemented as masked assignments. Innermost loops are the most likely candidates for vectorization, and the use of function calls within the loop can limit vectorization. Inlined functions and intrinsic SVML functions increase vectorization opportunities.

It is recommended to review vectorization information during the implementation and tuning stages of application development. The Intel® Compiler provides vectorization reports that provide insight into what was and was not vectorized. The reports are enabled via /Qvec-report=<n> command line option, where n specifies the level of reporting detail. The level of detail increases with a larger value of n. A value of n=3 will provide dependency information, loops vectorized, and loops not vectorized. The developer can modify the implementation based on the information provided in the report; the reason why a loop was not vectorized is very helpful.

The developer's intimate knowledge of his or her specific application can sometimes be used to override the behavior of the auto-vectorizer. Pragmas are available that provide additional information to assist with the auto-vectorization process. Some examples are: to always vectorize a loop, to specify that the data within a loop is aligned, to ignore potential data dependencies, etc. The addFloats example illustrates some important points. It is necessary to review the generated assembly language instructions to see what the compiler generated. The Intel Compiler will generate an assembly file in the current working directory when the /S command line option is specified.

```
void addFloats(float *a, float *b, float *c, float *d, float *e, int n)  {
       int i;
#pragma simd
#pragma vector aligned
       for(i = 0; i < n; ++i)       {
              a[i] = b[i] + c[i] + d[i] + e[i];
       }
}
```

Note the use of the simd and vector pragmas. They play a key role to achieve the desired Intel® AVX 256-bit vectorization. Adding "#pragma simd" to the code helps as packed versions of Intel® 128-bit AVX instructions are generated. The compiler also unrolled the loop once which reduces the number of executed instructions related to end-of-loop testing. Specifying "pragma vector aligned" provides another hint that instructs the compiler to use aligned data movement instructions for all array references. The desired 256-bit Intel® AVX instructions are generated by using both "pragma simd" and "pragma vector aligned." The Intel® Compiler chose vmovups because there is no penalty for using the unaligned move instruction when accessing aligned memory on the 2nd generation Intel® Core™ processor.

With `#pragma simd` and `#pragma vector aligned`

```
.B46.4::
        vmovups   ymm0, YMMWORD PTR [rdx+rax*4]
        vaddps    ymm1, ymm0, YMMWORD PTR [r8+rax*4]
        vaddps    ymm2, ymm1, YMMWORD PTR [r9+rax*4]
        vaddps    ymm3, ymm2, YMMWORD PTR [rbx+rax*4]
        vmovups   YMMWORD PTR [rcx+rax*4], ymm3
        vmovups   ymm4, YMMWORD PTR [32+rdx+rax*4]
        vaddps    ymm5, ymm4, YMMWORD PTR [32+r8+rax*4]
        vaddps    ymm0, ymm5, YMMWORD PTR [32+r9+rax*4]
        vaddps    ymm1, ymm0, YMMWORD PTR [32+rbx+rax*4]
        vmovups   YMMWORD PTR [32+rcx+rax*4], ymm1
        add       rax, 16
        cmp       rax, r11
        jb        .B46.4
```

This demonstrates some of the auto-vectorization capabilities of the Intel® Compiler. Vectorization can be confirmed by vector reports, the simd assert pragma, or by inspection of generated assembly language instructions. Pragmas can further assist the compiler when used by developers with a thorough understanding of their applications. Please refer to A Guide to Vectorization with Intel(R) C++ Compilers for more details on vectorization in the Intel Compiler. The Intel® C++ Compiler XE 12.0 User and Reference Guide has additional information on the use of vectorization, pragmas and compiler switches. The Intel Compiler can do much of the vectorization work for you so that your application will be ready to utilize Intel® AVX.

Intel® Cilk™ Plus C/C++ Extensions for Array Notations

The Intel® Cilk™ Plus C/C++ language extension for array notations is an Intel-specific language extension that is applicable when an algorithm operates on arrays, and doesn't require a specific ordering of operations among the elements of the array. If the algorithm is expressed using array notation and compiled with the AVX switch, the Intel® Compiler will generate Intel® AVX instructions. C/C++ language extensions for array notations are intended to allow users to directly express high-level parallel vector array operations in their programs. This assists the compiler in performing data dependence analysis, vectorization, and auto-parallelization. From the developer's point of view, they will see more predictable vectorization, improved performance and better hardware resource utilization. The combination of C/C++ language extension for array notations and other Intel® Cilk™ Plus language extensions simplify parallel and vectorized application development.

The developer begins by writing a standard C/C++ elemental function that expresses an operation using scalar syntax. This elemental function can be used to operate on a single element when invoked without C/C++ language extension for array notation. The elemental function must be declared using "__declspec(vector)" so that it can also be invoked by callers using C/C++ language extension for array notation.

The multiplyValues example is shown as an elemental function:

```
__declspec(vector) float multiplyValues(float b, float c)
{
	return b*c;
}
```

This scalar invocation is illustrated in this simple example:

```
	float a[12], b[12], c[12];
a[j] = multiplyValues(b[j], c[j]);
```

The function can also act upon an entire array, or portion of an array, by utilizing of C/C++ language extension for array notations. A section operator is used to describe the portion of the array on which to operate. The syntax is: [<lower bound> : <length> : <stride>]

Where lower bound is the starting index of the source array, length is the length of the resultant array, and stride expresses the stride through the source array. Stride is optional and defaults to one.

These array section examples will help illustrate the use:

```
	float a[12];
```

`a[:]` refers to the entire a array

`a[0:2:3]` refers to elements 0 and 3 of array a.

`a[2:2:3]` refers to elements 2 and 5 of array a

`a[2:3:3]` refers to elements 2, 5, and 8 of array a

The notation also supports multi-dimensional arrays.

```
float a2d[12][4];
```

`a2d[:][:]` refers to the entire a2d array

`a2d[:][0:2:2]` refers to elements 0 and 2 of the columns for all rows of a2d.

The array notation makes it very straightforward to invoke the multiplyValues using arrays. The Intel® Compiler provides the vectorized version and dispatches execution appropriately. Here are some examples. The first example acts on the entire array and the second operates on a subset or section of the array.

This example invokes the function for the entire array:

```
a[:] = multiplyValues(b[:], c[:]);
```

This example invokes the function for a subset of the arrays:

```
a[0:5] = multiplyValues(b[0:5], c[0:5]);
```

These simple examples show how C/C++ language extension for array notations uses the features of Intel® AVX without requiring the developer to explicitly use any Intel® AVX instructions. C/C++ language extension for array notations can be used with or without elemental functions. This technology provides more flexibility and choices to developers, and utilizes the latest Intel® AVX instruction set architecture. Please refer to the Intel® C++ Compiler XE 12.0 User and Reference Guide for more details on Intel® Cilk™ Plus C/C++ language extension for array notations.

Using the Intel® IPP and Intel® MKL Libraries

Intel offers thousands of highly optimized software functions for multimedia, data processing, cryptography, and communications applications via the Intel® Integrated Performance Primitives and Intel® Math Kernel Libraries. These thread-safe libraries support multiple operating systems and the fastest code will be executed on a given platform. This is an easy way to add multi-core parallelization and vectorization to an application, as well as take advantage of the latest instructions available for the processor executing the code. The Intel® Performance Primitives 7.0 includes approximately 175 functions that have been optimized for Intel® AVX. These functions can be used to perform FFT, filtering, convolution, correlation, resizing and other operations. The Intel® Math Kernel Library 10.2 introduced support for Intel® AVX for BLAS (DGEMM), FFT, and VML (exp, log, pow). The implementation has been simplified in Intel® MKL 10.3 as the initial call to mkl_enable_instructions is no longer necessary. Intel® MKL 10.3 extended Intel® AVX support to DGMM/SGEMM, radix-2 Complex FFT, most real VML functions, and VSL distribution generators.

If you are already using, or are considering using these versions of the libraries, then your application will be able to utilize the Intel® AVX instruction set. The libraries will execute Intel® AVX instructions when run on a Sandy Bridge platform and are supported on Linux*, Windows*, and Mac OS* X platforms.

More information on the Intel® IPP functions that have been optimized for Intel® AVX can be found in http://software.intel.com/en-us/articles/intel-ipp-functions-optimized-for-intel-avx-intel-advanced-vector-extensions/ More information on Intel® MKL AVX support can be found in Intel(R) AVX Optimization in Intel(R) MKL V10.3.

Usage Guidelines

The need for greater computing performance continues to drive Intel's innovation in micro-architectures and instruction sets. Application developers want to ensure that their product will be able to take advantage of advancements without a significant development effort. The methods, tools, and libraries discussed in this paper provide the means for developers to benefit from the advancements introduced by Intel® Advanced Vector Extensions without having to write a single line of Intel® AVX assembly language.

Additional Resources

Intel® Software Network Parallel Programming Community

Intel(R) Advanced Vector Extensions

Using AVX Without Writing AVX

Intel(R) Compilers

Intel(R) C++ Composer XE 2011 - Documentation

How to Compile for Intel(R) AVX

A Guide to Vectorization with Intel(R) C++ Compilers

Intel(R) Integrated Performance Primitives Functions Optimized for Intel(R) Advanced Vector Extensions

Enabling Intel(R) Advanced Vector Extensions Optimizations in Intel(R) MKL

Intel(R) AVX Optimization in Intel(R) MKL V10.3