**White Paper**

# Using Intel® AVX without Writing AVX

## Introduction and Tools

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 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 application 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 instrinsic 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. A table of equivalent command line switches for each platform can be found in Table 1 - Summary of Intel® Compiler Command Line Switches on page 9.

## Review of Single-Instruction-Multiple-Data (SIMD) Concepts

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.

The generic multiplyValues example helps to illustrate the difference between scalar and vector process using Intel® AVX.

```
void multiplyValues(float *a, float *b, float * c, int size)
{
        for (i = 0; i < size; i++) {
                a[i] = b[i] * c[i];
        }
}
```
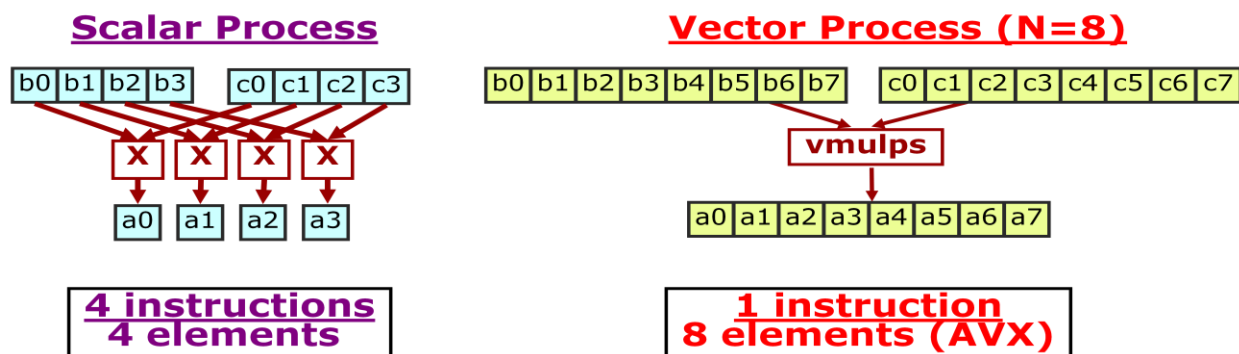


## 1    Recompile for Intel® AVX

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.

## 2   Compiler Auto-vectorization

Compiling the application with the appropriate architecture switch is a great first step towards 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++ ComposeXE 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's 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 with 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 their 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.  The following code is generated when addFloats is compiled with the /QaxAVX option but without the lines containing the simd and vector pragmas:

Without simd and vector pragmas

```
.B46.3::
        vmovss    xmm0, DWORD PTR [rdx+r10*4]
        vaddss    xmm1, xmm0, DWORD PTR [r8+r10*4]
        vaddss    xmm2, xmm1, DWORD PTR [r9+r10*4]
        vaddss    xmm3, xmm2, DWORD PTR [rax+r10*4]
        vmovss    DWORD PTR [rcx+r10*4], xmm3
        inc       r10
        cmp       r10, r11
        jl        .B46.3
```

The assembly code shows the use of scalar versions of Intel® 128-bit AVX instructions. The goal is to produce packed (another word for vector) versions of Intel® 256-bit AVX instructions.  The ss in vaddss indicates that only one set of single precision floating point operands are being added together, a scalar operation.  The algorithm will be more efficient if vaddps is used instead, ps indicates a packed operation on single precision floating point operands.

Adding just "#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. There is still room for more optimization because only four operands are operated on per instruction.

```
    With #pragma simd


.B46.11::
      vmovups    xmm0, XMMWORD PTR [rdx+r10*4]
      vaddps     xmm1, xmm0, XMMWORD PTR [r8+r10*4]
      vaddps     xmm2, xmm1, XMMWORD PTR [r9+r10*4]
      vaddps     xmm3, xmm2, XMMWORD PTR [rax+r10*4]
      vmovups    XMMWORD PTR [rcx+r10*4], xmm3
      vmovups    xmm4, XMMWORD PTR [16+rdx+r10*4]
      vaddps     xmm5, xmm4, XMMWORD PTR [16+r8+r10*4]
      vaddps     xmm0, xmm5, XMMWORD PTR [16+r9+r10*4]
      vaddps     xmm1, xmm0, XMMWORD PTR [16+rax+r10*4]
      vmovups    XMMWORD PTR [16+rcx+r10*4], xmm1
      add        r10, 8
      cmp        r10, rbp
      jb         .B46.11
```

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 application. 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.

# 3    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, then the Intel® Compiler will generate Intel® AVX instructions. C/C++ language extension 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 to operate on. 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
a[:][0:2:2] refers to elements 0 and 2 of the columns for all rows of a2d.
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 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]);
```

C/C++ language extension for array notations simplifies the development of array-based applications. As with auto-vectorization, it's necessary to inspect the instructions generated by the Intel® Compiler to verify that Intel® AVX instructions are being used. Compiling the source with the /S option produces the assembly file. A search for mutliplyValues reveals that there are both a scalar and vectorized version.

The scalar implementation uses scalar (ss) versions of VEX-encode 128-bit Intel® AVX instructions:
```
vmulss     xmm0, xmm0, xmm1
ret
```

The vector implementation uses scalar (ps) versions of VEX-encode 256-bit Intel® AVX instructions:

```
sub        rsp, 40
vmulps     ymm0, ymm0, ymm1
mov        QWORD PTR [32+rsp], r13
lea        r13, QWORD PTR [63+rsp]
and        r13, -32
mov        r13, QWORD PTR [32+rsp]
add        rsp, 40
ret
```

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.

Developers can utilize the compiler to generate Intel® AVX instructions and auto-vectorize their application.  They can also choose to develop with Intel® Cilk™ Plus C/C++ language extension for array notations to tap into the power of Intel® AVX. There is another way that development teams can take advantage of Intel® AVX without writing in assembly language.  The Intel® Integrated

Performance Primitives (Intel® IPP) and the Intel® Math Kernel Libraries (Intel® MKL) offer much to the developer, including support for the latest Intel technologies such as Intel® AVX.

## 4   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. They are 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 BLASS (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.

## 5   Summary

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.

## 6   More Information and References

Intel(R) Advanced Vector Extensions

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

Table 1 - Summary of Intel® Compiler Command Line Switches

| Description of Intel® Compiler Command Line Switch | Windows* | Linux* | Mac OS* X |
|---|---|---|---|
| Generate multiple, processor-specific auto-dispatch code paths for Intel processors if there is a performance benefit | /Qax | -ax | -ax |
| Generate a vectorization report | /Qvec-report<n> | -vec-report<n> | -vec-report<n> |
| Generate an assembly language file | /S | -S | -S |

## 7   Optimization Notice

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.

## 8  Notices

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.  Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions.  Any change to any of those factors may cause the results to vary.  You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel is a trademark of Intel Corporation in the U.S. and/or other countries.
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to:  http://www.intel.com/design/literature.htm