Avoiding AVX-SSE Transition Penalties

Transitioning between 256-bit Intel® AVX instructions and legacy Intel® SSE instructions within a program may cause performance penalties because the hardware must save and restore the upper 128 bits of the YMM registers. This paper discusses how and why these transition penalties occur, methods to detect AVX-SSE transitions, and methods to remove transitions or avoid the transition penalties. It also discusses the implications that CPU dispatching can have on AVX-SSE transitions, and provides general recommendations to avoid issues when using Intel® AVX.

1. Introduction to AVX-SSE Transition Penalties

Intel® Advanced Vector Extensions (Intel® AVX) is a new SIMD instruction set extension available as part of the 2nd generation Intel® Core™ processor family. Intel® AVX features wider 256-bit vectors and new instructions, and uses the new Vector Extension (VEX) extensible instruction encoding format, which adds supports for three (or more) operand instructions. Intel® AVX also includes 128-bit VEX encoded instructions equivalent to all legacy Intel® Streaming SIMD Extensions (Intel® SSE) 128-bit instructions.

When using Intel® AVX instructions, it is important to know that mixing 256-bit Intel® AVX instructions with legacy (non VEX-encoded) Intel® SSE instructions may result in penalties that could impact performance. 256-bit Intel® AVX instructions operate on the 256-bit YMM registers which are 256-bit extensions of the existing 128-bit XMM registers. 128-bit Intel® AVX instructions operate on the lower 128 bits of the YMM registers and zero the upper 128 bits. However, legacy Intel® SSE instructions operate on the XMM registers and have no knowledge of the upper 128 bits of the YMM registers. Because of this, the hardware saves the contents of the upper 128 bits of the YMM registers when transitioning from 256-bit Intel® AVX to legacy Intel® SSE, and then restores these values when transitioning back from Intel® SSE to Intel® AVX (256-bit or 128-bit). The save and restore operations both cause a penalty that amounts to several tens of clock cycles for each operation.

There are several different situations where AVX-SSE transitions might occur, such as when 256-bit Intel® AVX intrinsic instructions or inline assembly are mixed with any of the following:

   a. 128-bit intrinsic instructions
   b. Intel® SSE inline assembly
   c. C/C++ floating point code that is compiled to Intel® SSE
   d. Calls to functions or libraries that include any of the above.

Additionally, AVX-SSE transitions may occur if code containing 256-bit Intel® AVX instructions is executing and an interrupt occurs where the interrupt's service routine (ISR) contains legacy Intel® SSE instructions. In the case where ISRs cause AVX-SSE transition penalties, there is nothing the application developer can do to avoid the penalties. ISR developers should be aware of this potential penalty when using XMM/YMM registers within their routines, and should use the same methods discussed below to avoid AVX-SSE transition penalties, as well as ensuring they save and restore the entire YMM state when necessary.

It is often possible to remove AVX-SSE transitions by converting legacy Intel® SSE instructions to their equivalent VEX encoded instructions. When it is not possible to remove the transitions, it is often possible to avoid the penalty by explicitly zeroing the upper 128-bits of the YMM registers, in which case the hardware does not save these values. Methods to avoid the AVX-SSE transition penalty are discussed in depth in section 3.

Consider the following example where we use both 128-bit and 256-bit intrinsic instructions. The assembly that is generated (also shown below) contains mostly Intel® AVX instructions (prefixed with "v"). However, it also contains a legacy Intel® SSE instruction (movaps). Immediately before the movaps instruction the hardware will save the contents of the upper 128 bits of the YMM registers. The hardware will restore these values when it sees the next Intel® AVX instruction, which will come on the next iteration. The following code was compiled at the command line with the Intel® Compiler version 12.0.4 using –O3.

Figure 1. C source and disassembly for example, showing the location of AVX-SSE transitions.

```
float* a; float* b; float* c; // allocate and initialize memory
for (int i = 0; i < size; i += 4) {
        __m128  av_128 = _mm_load_ps(a + i);
        __m128  bv_128 = _mm_load_ps(b + i);
        __m256d av_256 = _mm256_cvtps_pd(av_128);
        __m256d bv_256 = _mm256_cvtps_pd(bv_128);
        __m256d cv_256 = _mm256_sqrt_pd(_mm256_add_pd(_mm256_mul_pd(av_256, av_256),
            _mm256_mul_pd(bv_256, bv_256)));
        __m128  cv_128 = _mm256_cvtpd_ps(cv_256);
        _mm_store_ps(c + i, cv_128);
}
```

```
loop:   vcvtps2pd (%rbx,%rax,4), %ymm0          ⟵  SSE-AVX transition (after first iteration)
        vcvtps2pd (%rcx,%rax,4), %ymm1
        vmulpd    %ymm0, %ymm0, %ymm2
        vmulpd    %ymm1, %ymm1, %ymm3
        vaddpd    %ymm2, %ymm3, %ymm4
        vsqrtps   %xmm4, %xmm5
        vcvtpd2ps %ymm5, %xmm6
        movaps    %xmm6, (%rdx,%rax,4)          ⟵  AVX-SSE transition
        addq      $4, %rax
        cmpq      $1048576, %rax
        jl        loop
```

2. Detecting AVX-SSE Transitions

2.1. Using Intel® Software Development Emulator

Intel® Software Development Emulator (Intel® SDE) is a command line tool for Windows* and Linux* that developers can use to detect dynamic AVX-SSE transitions in their programs, even on processors that do not support Intel® AVX. Intel® SDE will report the number of AVX-SSE and SSE-AVX transitions for a specific block within a function. Command line usage and sample output detailing information on AVX-SSE transitions can be found in the figure below. The advantages of using Intel® SDE is that it is free, it is very simple and quick to use, and it can be used on processors that do not support Intel® AVX; the disadvantage of using Intel® SDE is that it does not show the specific instructions that cause transitions. For more information, see the Intel® Software Development Emulator website.

Figure 2. Command to use Intel® SDE to detect AVX-SSE transitions, and sample output from Intel® SDE.

```
sde –oast avx-sse-transitions.out –- user-application [args]
```

```
        Penalty    Dynamic     Dynamic
          in  AVX to SSE  SSE to AVX  Static          Dynamic         Previous
        Block  Transition   Transition  Icount Executions  Icount         Block
================ ============= ============= ============ ======== ========== ======== =================
    0x13ff510b5        1         0      18      1      18        N/A
#Penalty detected in routine: main @ 0x13ff510b5
    0x13ff510d1    262143     262143     11    262143  2883573     0x13ff510d1
#Penalty detected in routine: main @ 0x13ff510d1
# SUMMARY
# AVX_to_SSE_transition_instances:     262144
# SSE_to_AVX_transition_instances:     262143
# Dynamic_insts:             155387299
# AVX_to_SSE_instances/instruction:     0.0017
# SSE_to_AVX_instances/instruction:     0.0017
# AVX_to_SSE_instances/100instructions:  0.1687
# SSE_to_AVX_instances/100instructions:  0.1687
```

2.2. Using Intel® vTune™ Amplifier XE

The 2nd generation Intel® Core™ processor family has support for hardware events that correspond to the transitions from 256-bit Intel® AVX to Intel® SSE (OTHER_ASSISTS.AVX_TO_SSE) and from Intel® SSE to Intel® AVX (OTHER_ASSISTS.SSE_TO_AVX). Developers can use Intel® vTune™ Amplifier XE on a 2nd generation Intel® Core™ processor to utilize these hardware events to detect AVX-SSE transitions. To utilize these events in Intel® vTune™ Amplifier XE, you will need to create a new hardware event-based custom analysis using the following steps, annotated in Microsoft* Visual Studio 2010 SP1 on the figure below (note: SP1 is required when using Intel® AVX in Microsoft Visual Studio* 2010):

1. Create a New Analysis
2. Click "New …" and select "New Hardware Event-based Sampling Analysis"
3. Click "Add Event", select the OTHER_ASSISTS.AVX_TO_SSE and OTHER_ASSISTS.SSE_TO_AVX events, and click "OK"
4. Click "Start" to start the analysis

When the analysis has completed you will see the event counts by function, which you can use to determine which functions have AVX-SSE transitions. You can also click on any function to view hotspots for these specific events in the source or disassembly, which can tell you exactly which instructions are causing transitions.

The advantage of using Intel® vTune™ Amplifier XE to detect AVX-SSE transitions is that it can show you the precise location in your source code and disassembly that is causing transitions. The disadvantage of Intel® vTune™ Amplifier XE is that it must be used on a processor that supports Intel® AVX in order to detect AVX-SSE transition events.

Figure 3. Steps to create a custom analysis to detect AVX-SSE transitions using Intel® vTune™ Amplifier XE.
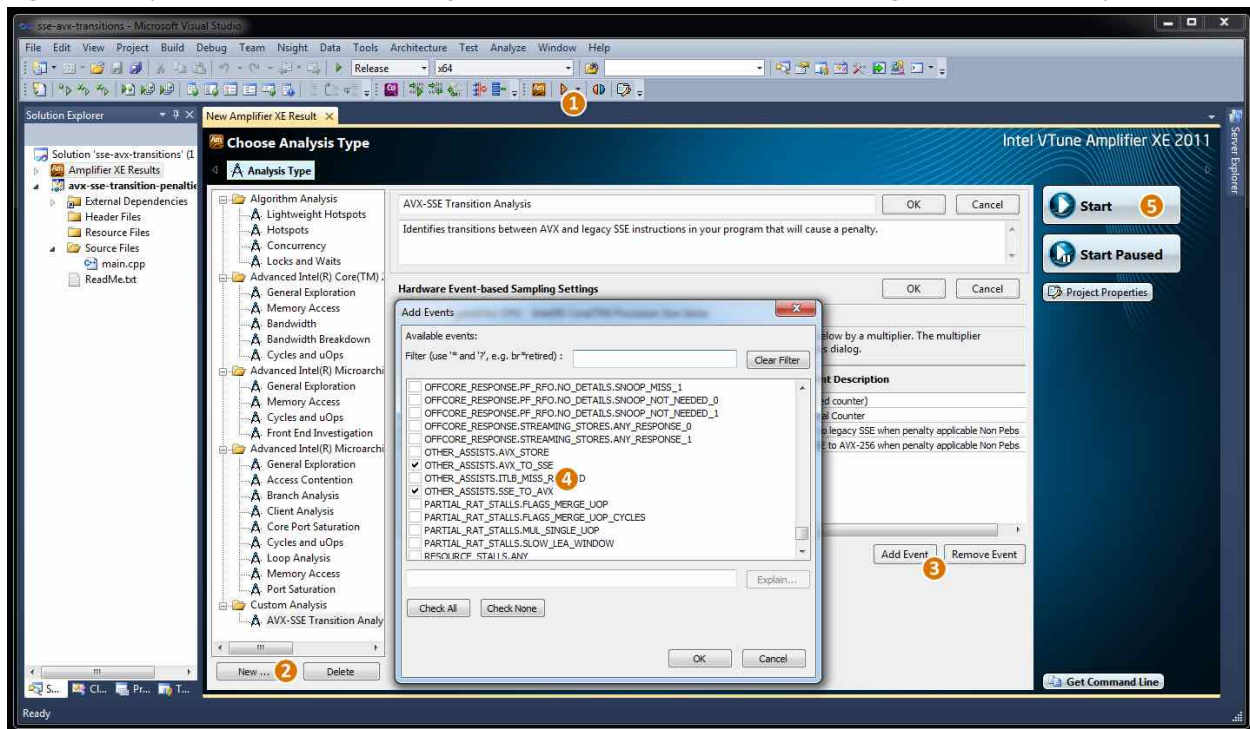
3. Methods to Avoid AVX-SSE Transition Penalties

3.1. Method 1: Automatically Converting to VEX with Compiler Flags

There are several methods to either remove AVX-SSE transitions or to remove the penalty from transitions. The easiest method to avoid the AVX-SSE transition penalty is to compile the relevant source files with the Intel® Compiler using either the –xavx (/Qxavx on Windows*) or –mavx (/arch:avx on Windows*) flag. These flags tell the Intel® Compiler to generate instructions that are specialized for processors that support Intel® AVX; the –xavx flag tells the Intel® Compiler to also attempt to *optimize* the code for processors that support Intel® AVX.

When these flags are used the compiler will automatically generate VEX-encoded instructions rather than legacy Intel® SSE instructions where appropriate, which removes the transition between Intel® AVX and Intel® SSE within those files. They also tell the compiler to automatically insert vzeroupper instructions, which zero out the upper 128 bits of the YMM registers (see next section). When these flags are used, the compiler will insert a vzeroupper instruction at the beginning of a function containing Intel® AVX code if none of the arguments are a YMM register or __m256/__m256d/__m256i datatype; the compiler will also insert a vzeroupper instruction at the end of functions if the returned value is not a YMM register or __m256/__m256d/__m256i datatype. Inserting vzeroupper instructions prevents AVX-SSE transitions from occurring when calling the functions in those files from routines that may have legacy Intel® SSE instructions. For more information on Intel® Compiler flags for processor-specific optimizations, see the Intel® Compiler documentation.

The advantage to this method is that the compiler does it automatically. Additionally, this is the only method that can force 128-bit intrinsic instructions to generate VEX encoded instructions (when not using –xavx or –mavx, 128-bit intrinsic instructions are not guaranteed to generate VEX encoded instructions). In some situations compilers will compile C/C++ floating point code to Intel® SSE instructions as opposed to x87 instructions. If C/C++ floating point code would be compiled to Intel® SSE instructions, then using the –xavx or –mavx flag is the only method that can force the compiler to produce VEX encoded instructions. C/C++ floating point code compiled to x87 instructions will not cause transition penalties.

A disadvantage of this method is that it requires access to the relevant source files, so it cannot avoid AVX-SSE transitions resulting from calls to functions that are not compiled with the –xavx or –mavx flag. Another possible disadvantage is that all Intel® SSE code within a file compiled with the –xavx or –mavx flag will be converted to VEX format and will only run on Intel® AVX supported processors. If a file contains code intended to run on multiple different generation processors then you should consider separating the functionality into separate files and compiling each file with the relevant compiler flag; also, see section 4 on CPU dispatching.

Returning to our example, by compiling the file with the –xavx flag, the compiler will now generate the vmovaps instruction rather than the movaps instruction, which removes the AVX-SSE transition. Prior to removing the transition this code took more than 230 cycles per iteration. After compiling with –xavx the code now takes approximately 70 cycles per iteration[1].

---

[1] On a 2.3 GHz Intel® Core™ i7 running Mac OS X 10.6.8, compiled using Intel® Compiler 12.0.4 with –O3.

Figure 4. When using –xavx, the compiler will use the VEX encoded version of 128-bit instructions.

```
loop:   vcvtps2pd (%rbx,%rax,4), %ymm0
        vcvtps2pd (%rcx,%rax,4), %ymm1
        vmulpd    %ymm0, %ymm0, %ymm2
        vmulpd    %ymm1, %ymm1, %ymm3
        vaddpd    %ymm2, %ymm3, %ymm4
        vsqrtps   %xmm4, %xmm5
        vcvtpd2ps %ymm5, %xmm6
        vmovaps   %xmm6, (%rdx,%rax,4)
        addq      $4, %rax
        cmpq      $1048576, %rax
        jl        loop
```

3.2. Method 2: Automatically Converting to VEX with Pragmas

Another method of automatically converting to VEX with the Intel® Compiler is to use the Intel® specific pragma: #pragma intel optimization_parameter target_arch=avx; this pragma is new to Intel® Compiler 12.1. When placed at the head of a function, this pragma has the effect of applying –mavx to that function only. This will cause VEX encoded instructions to be automatically generated where appropriate within this function, and a vzeroupper instruction to be automatically inserted at the beginning and end of the function.

The advantage of this method is that it can be applied at the function level as opposed to the file level, as is the case with –xavx and –mavx. As a result, it is not necessary to separate functionality intended to run on multiple different generation processors into multiple different files. A disadvantage of this method is that, like –xavx and –mavx, this method requires access to the relevant source files, so it cannot avoid AVX-SSE transitions resulting from calls to functions that are not accessible. Another disadvantage of this method is that if a function marked with this pragma is chosen for inlining by the Intel® Compiler, at present the Intel® Compiler will not apply –mavx to that code. This can be avoided by explicitly preventing the Intel® Compiler from inlining the function by using the __declspec(noinline) keyword.

Figure 5. Example of using the optimization_parameter pragma and __declspec(noinline)

```
#pragma intel optimization_parameter target_arch=avx
__declspec(noinline) void function_with_avx() { ... }
```

3.3. Method 3: Zeroing Registers

In many cases it may not be possible to remove the transition from Intel® AVX to Intel® SSE, such as when it is necessary to call to a library that uses legacy Intel® SSE. In those cases, intrinsic instructions or inline assembly can be used to call the vzeroupper instruction, which zeros out the upper 128 bits of the YMM registers (similarly, the vzeroall instruction can be used, which zeros out all 256 bits of the YMM registers). When the upper 128 bits of the YMM registers are set to zero by the vzeroupper instruction, the hardware does not need to save those values, so the hardware assists do not occur. The vzeroupper instruction must be used after 256-bit Intel® AVX code and before Intel® SSE code, which will remove both the save and the restore operations. Zeroing out the YMM registers with other methods, such as with XORs, will not prevent AVX-SSE transition penalties.

An advantage of this method is that it is the only way to avoid the AVX-SSE transition penalty when using functions or libraries that contain legacy Intel® SSE and that are not under your control.

Another advantage is that this method can be implemented without writing assembly by using the intrinsic instructions _mm256_zeroupper() and _mm256_zeroall(). The disadvantage of this method is that the vzeroupper instructions must be correctly placed to avoid all transition penalties.

To resolve the issue in our example code we must add a call to vzeroupper (using the _mm256_zeroupper() intrinsic instruction) immediately after our last 256-bit Intel® AVX intrinsic instruction and before our 128-bit intrinsic instruction. After adding the code to zero the upper 128 bits of the YMM registers this code takes approximately 70 cycles per iteration.

Figure 6. Zeroing registers to avoid the AVX-SSE transition penalty.

```
float* a; float* b; float* c; // allocate and initialize memory
for (int i = 0; i < size; i += 4) {
        __m128  av_128 = _mm_load_ps(a + i);
        __m128  bv_128 = _mm_load_ps(b + i);
        __m256d av_256 = _mm256_cvtps_pd(av_128);
        __m256d bv_256 = _mm256_cvtps_pd(bv_128);
        __m256d cv_256 = _mm256_sqrt_pd(_mm256_add_pd(_mm256_mul_pd(av_256, av_256),
            _mm256_mul_pd(bv_256, bv_256)));
        __m128  cv_128 = _mm256_cvtpd_ps(cv_256);
        _mm256_zeroupper();
        _mm_store_ps(c + i, cv_128);
}
```

3.4. Method 4: Manually Converting Assembly to VEX

The final method to avoid the AVX-SSE transition penalty is to manually convert any legacy Intel® SSE assembly instructions to their VEX encoded equivalent, which removes the AVX-SSE transition. Information on VEX encoded instructions can be found in the [Intel® Architectures Software Developer's Manuals](#).

An advantage to manually converting to VEX is that it allows you to selectively convert the assembly within a file, as opposed to having all converted with –xavx. Additionally, if for some reason using –xavx or the pragma is not possible or ideal, then manually converting assembly to VEX

```
loop:   vcvtps2pd (%rbx,%rax,4), %ymm0
        vcvtps2pd (%rcx,%rax,4), %ymm1
        vmulpd    %ymm0, %ymm0, %ymm2
        vmulpd    %ymm1, %ymm1, %ymm3
        vaddpd    %ymm2, %ymm3, %ymm4
        vsqrtps   %xmm4, %xmm5
        vcvtpd2ps %ymm5, %xmm6
        vzeroupper
        movaps    %xmm6, (%rdx,%rax,4)
        addq      $4, %rax
        cmpq      $1048576, %rax
        jl        loop
```

is the only option. Another advantage to manually converting to VEX is that it allows you to take advantage of the non-destructive three-operand forms in your assembly. The disadvantages of this method are that it must be done manually, it can only be done in assembly code, and that the code will only run on processors that support Intel® AVX.

4. AVX-SSE Transitions and CPU Dispatching

In many cases it is ideal to have multiple versions of a given function, where each is optimized for certain CPU features (e.g. Intel® SSE2, Intel® AVX, etc.). For instance, this might be useful when you would like to have an Intel® AVX and non-AVX version of a function, so that you can take advantage of Intel® AVX but still support non-AVX processors. In such cases, CPU dispatching is used to "dispatch" execution to the most appropriate version of the function based on which CPU the program is running on. There are three methods of implementing CPU dispatching: automatically with the Intel® Compiler, manually using the Intel® Compiler's manual-dispatching feature, or manually with a custom mechanism provided by the developer. We will discuss the automatic and manual CPU dispatching using the Intel® Compiler and the implications these have on AVX-SSE transitions; these methods are not guaranteed to work with other compilers, and developers should understand that CPU dispatching may be their own responsibility on other compilers.

4.1. Intel® Compiler's Auto-Dispatching Feature

To take advantage of the Intel® Compiler auto-dispatching feature, use the –axavx flag (/Qaxavx on Windows*). This flag directs the Intel® Compiler to look for opportunities to optimize the existing code using any of the Intel® SIMD extensions, up to and including Intel® AVX. The Intel® Compiler will generate optimized processor-specific versions of existing functions when it finds sufficient performance benefit, and will also generate functionality to auto-dispatch to the appropriate function at execution. The Intel® Compiler will always generate a generic function containing the original code, but may or may not generate any particular processor-specific version. For more information on the Intel® Compiler's auto-dispatching feature, see [Intel® compiler options for SSE generation and processor-specific optimizations](#).

When using the Intel® Compiler's auto-dispatching feature, the compiler will decide on a function-by-function basis whether it will produce auto-dispatched processor-specific versions. If the compiler targets a function for auto-dispatch and generates a code path optimized for Intel® AVX, then Intel® AVX instructions will be generated as appropriate, all relevant instructions within that function will automatically be VEX encoded, and vzeroupper instructions will automatically be inserted at the beginning and end of the function. However, if a function is not targeted for auto-dispatch and the developer has manually added Intel® AVX intrinsic instructions, then it is not guaranteed that all relevant instructions within that function will be VEX encoded, and vzeroupper instructions will not automatically be inserted. It is important to understand that just using –axavx does not guarantee that the Intel® Compiler will optimize your code for Intel® AVX, and your program may still have the same AVX-SSE transitions it would if you did not use –axavx (using –axavx is not the same as using –xavx).

4.2. Intel® Compiler's Manual-Dispatching Feature

The Intel® Compiler's manual-dispatching feature allows the developer to explicitly define processor-specific versions of a function. The Intel® Compiler will then automatically generate functionality to dispatch to the appropriate version during execution. Manual dispatching can be helpful when you want to explicitly define an Intel® AVX version of a function, but also want to explicitly support other processors that do not support Intel® AVX (for instance, an Intel® AVX version, an Intel® SSE version, and a generic version).

Manual dispatching is implemented using the __declspec(cpu_dispatch()) and __declspec(cpu_specific()) keywords. The __declspec(cpu_dispatch(cpuid,...)) keyword should be placed above a stub of the function to be dispatched; the cpuid parameters should specify all the specific processors that are being explicitly targeted. The __declspec(cpu_specific(cpuid, ...)) keyword should be placed above processor-specific implementations of the function; the cpuid of one or more specific targeted processors must be supplied. The core_2nd_gen_avx cpuid is used to target processors that support Intel® AVX. For more information and an example on the Intel® Compiler's manual dispatching feature, see [How to manually target 2nd generation Intel® Core™ processors with support for Intel® AVX](#).

Within function versions that specify the core_2nd_gen_avx cpuid, all relevant intrinsic instructions and inline assembly[2] will automatically be VEX encoded and vzeroupper instructions will automatically be inserted at the beginning and end of the function. Any functions that do not specify the core_2nd_gen_avx cpuid but do contain Intel® AVX intrinsic instructions would be targeted for processors that do not support Intel® AVX and would generate an exception at runtime.

Table 1. Effects that different compiler flags and dispatching scenarios have on code generation.

situation	128-bit intrinsics & FP code	Intel® SSE inline assembly	vzeroupper in function[1]
default	non-VEX	non-VEX	no
-xavx and -mavx	VEX	VEX	yes
pragma with target_arch=avx	VEX	VEX[2]	yes
ICC auto-dispatching (targeted)	VEX	VEX[2]	yes
ICC auto-dispatching (not targeted)	non-VEX	non-VEX	no
ICC manual dispatching (core_2nd_gen_avx)	VEX	VEX[2]	yes
ICC manual dispatching (not core_2nd_gen_avx)	non-VEX	non-VEX	no

5. Summary & Recommendations

There is a performance penalty when switching between 256-bit Intel® AVX instructions and Intel® SSE instructions because the hardware saves and restores the upper 128 bits of the YMM registers. To remove this penalty you can convert all legacy Intel® SSE instructions to their VEX encoded equivalents using the –xavx or -mavx flag with the Intel® Compiler, the new Intel® specific pragma, or by manually converting assembly. In cases where you cannot avoid the transition, you can remove

---

[1] At the beginning if none of the functions arguments are a YMM register or a __m256/__m256d/__m256i datatype; at the end if the returned value is not a YMM register or __m256/__m256d/__m256i datatype.
[2] In these scenarios the Intel® Compiler does not currently convert Intel® SSE inline assembly to VEX-encoded instructions. The intended behavior is that Intel® SSE inline assembly be VEX-encoded. This issue is under investigation and will be resolved shortly.

the penalty by zeroing the YMM registers after 256-bit Intel® AVX instructions and before Intel® SSE instructions using the vzeroupper instruction.

To minimize issues when using Intel® AVX, it is recommended that you compile any source files intended to run on processors that support Intel® AVX with the –xavx flag. If your code contains functions intended to be run on multiple different generation processors, then it is recommended that you use the new Intel® specific pragma as opposed to compiling with -xavx. Additionally, you should use the VEX encoded form of 128-bit instructions to avoid AVX-SSE transitions. Even if your code does not contain legacy Intel® SSE code, when you have completed your use of 256-bit Intel® AVX within your code you should zero the registers as soon as possible using the vzeroupper instruction or their intrinsic instructions; this can help you avoid introducing transitions in the future or causing transitions in programs that may use your code. Finally, when developing a program that includes Intel® AVX, it is recommended that you always check for AVX-SSE transitions with Intel® Software Development Emulator or Intel® vTune™ Amplifier XE.

6.   About the Author

Patrick Konsor is an Application Engineer on the Apple Enabling Team at Intel in Santa Clara, and specializes in software optimization. Patrick received his BS in computer science from the University of Wisconsin-Eau Claire. He enjoys reading and cycling in his free time (go Schlecks!).

7. Notices

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

Optimization Notice

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

NOTE: additional notices and disclaimers may be needed, depending on the content of the collateral. Generally, they can be found in the following areas, and must be appended to the Notices section of the paper.

General Notices: *Notices placed in all materials distributed or released by Intel*
Benchmarking and Performance Disclaimers: *Disclaimers for Intel materials that use benchmarks or make performance claims.*
Technical Collateral Disclaimers: *Disclaimers that should be included in Intel technical materials that describe the form, fit or function of Intel products.*
Technology Notices: *Notices for Intel materials when the benefits or features of a technology or program are described. Note for technology disclaimers - if every product being discussed (e.g., ACER ULV) has the particular technology/feature, then you can remove the requirements statement in the disclaimer. If you have multiple technical disclaimers, you can consolidate the "your performance may vary" statements and only put in a single "your mileage may vary".*