Data Types for Intel® MIC Architecture

- **16x floats**
- **8x doubles**
- **16x 32-bit integers**
- **8x 64-bit integers**
- **64x 8-bit bytes**
- **32x 16-bit shorts**

Not implemented
## Options for Vectorization

<table>
<thead>
<tr>
<th>Option</th>
<th>Ease of use / code maintainability (depends on problem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® MKL Libraries</td>
<td></td>
</tr>
<tr>
<td>Array Notation: Intel® Cilk™ Plus</td>
<td></td>
</tr>
<tr>
<td>Automatic vectorization</td>
<td></td>
</tr>
<tr>
<td>Semiautomatic vectorization with annotation: #pragma vector, #pragma ivdep, and #pragma simd</td>
<td></td>
</tr>
<tr>
<td>C/C++ Vector Classes (F32vec16, F64vec8)</td>
<td></td>
</tr>
<tr>
<td>Vector intrinsics (mm_add_ps, addps)</td>
<td></td>
</tr>
</tbody>
</table>
Intel® Cilk™ Plus Array Notation - Major Concepts

• Data-parallel array notations provide a natural mapping between data-parallel algorithms and underlying parallel hardware

• Based on the concept of array-section notation:
  
  `<array>[<lower bound> : <length> : <stride>]`
  
  `<array>[<lower bound> : <length> : <stride>]`...

• Fortran Connection
  
  - Section specifier is [lower bound:length] pair (memcpy style)
  - Fortran uses [lower bound:upper bound]

  `a[:]`     // All elements of vector a
  
  `b[2:6]`   // Elements 2 to 7 of vector b
  
  `c[:][5]`  // Column 5 of matrix c
  
  `d[0:3:2]` // Elements 0, 2, and 4 of vector d={0,1,2,3,4}

• Language extension supported by the Intel® Compiler
The fragment:

\[ a[0:s]+b[2:s] \]

Is equivalent to the following pseudo code:

```
unordered_loop (i=0;i<s;i++)
   a[i]+b[i+2]
```

The fragment:

\[ a[0:s]*c \]

Is equivalent to the following pseudo code:

```
unordered_loop (i=0;i<s;i++)
   a[i]*c
```

Likewise:

```
a[:]=\text{pow}(b[:],c); \quad // \quad b[:]**c
\text{c[:]}=a[b[:]]; \quad \quad // \quad \text{gather elements of a into c,}
\quad \quad \quad \quad // \quad \text{according to index array b}
```
**Intel® Cilk™ Plus Array Notation - Array Section Reductions**

- Reductions combine array section elements to generate a scalar result
  - Nine built-in reduction functions supporting basic C data-types:
  - Supports user-defined reduction function

**Reductions**

- **Built-in**
  
  _sec_reduce_add(a[:]), _sec_reduce_mul(a[:])_
  _sec_reduce_min(a[:]), _sec_reduce_max(a[:])_
  _sec_reduce_min_ind(a[:]), _sec_reduce_max_ind(a[:])_
  _sec_reduce_all_zero(a[:]), _sec_reduce_all_nonzero(a[:])_
  _sec_reduce_any_nonzero(a[:])_

- **User-defined**

  result _sec_reduce (initial, a[:], fn-id)
  void _sec_reduce_mutating(reduction, a[:], fn-id)
Intel® Cilk™ Plus Array Notation - Example Reducer

Original Code:
float reduction(float *data, size_t size)
{
    float ret = 0.f;
    for (int i = 0; i < size; i++)
        ret += data[i];
    return ret;
}

Intel® Cilk™ Plus Array Notation version:
#include <cilk/cilk.h>
float reduction(float *data, size_t size)
{
    float ret = 0.f;
    ret = __sec_reduce_add(data[0:size]);
    return ret;
}
Explicit Vector Programming: Intel® Cilk™ Plus Array notation

void addit(double* a, double* b, int m, int n, int x)
{
    for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
    }
}

void addit(double* a, double * b, int m, int n, int x)
{
    // I know x<0
    a[m:n] = b[m:n] + a[m-x:n];
}

loop was not vectorized: existence of vector dependence.

- Array notation asks the compiler to vectorize
  - asserts this is safe (for example, x<0)
  - Improves readability

LOOP WAS VECTORIZED.
Vector and Elemental Processing

Vector Processing

Elemental Processing

Natural in case of scatter, or with sync. primitives

\[
z[0:10:10] = a[20:10:2] + y[x[0:10]];
\]

Kernel Function

\[
y[0:10:10] = \sin(x[20:10:2]);
\]

* The Intel Cilk Plus Array section syntax is [offset:size:stride] whereas F90 uses [lbound:ubound:stride].
Semi-automatic Vectorization by Annotation - Vector/Elemental Function Annotation

• Syntax  __attribute__((vector([(clauses)]))))
  • Creates a vectorized version of a scalar function definition
  • Adapt input argument and return values correspondingly
• Enables vectorization of the loop that calls the function

<table>
<thead>
<tr>
<th>Programmer</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Write a standard C/C++ function with a scalar syntax</td>
<td>1. Generates vector code with SIMD Operation.</td>
</tr>
<tr>
<td>2. Annotate it with a <strong>attribute</strong>((vector))</td>
<td>2. Compiler generated function operates on one vector at a time (as opposed to one scalar value).</td>
</tr>
<tr>
<td></td>
<td>3. Execute on a single core, or use the task scheduler to execute on multiple cores</td>
</tr>
</tbody>
</table>
// Convert C scalar function to vector version
__attribute__((vector)) float saxpy (float a, float x, float y) {
    return (a * x + y);
}

void main() {
    float a = 5.0f;
    float X[size], Y[size], Z[size];
    Z[:] = saxpy(a, X[:], Y[:]); // Call scalar function with
    // Array Notation parameters
Clauses for Vector Functions

• __attribute__((vector)) (Intel)

• #pragma omp declare simd (OpenMP* 4.0 RC1)

• Available clauses (both OpenMP and Intel versions)
  - LINEAR (additional induction variables)
  - UNIFORM (arguments that are loop constants)
  - PROCESSOR (Intel)
  - VECTORLENGTH (Intel)
  - MASK / NOMASK (Intel)
  - INBRANCH / NOTINBRANCH (OpenMP 4.0 RC1)
  - SIMDLEN (OpenMP 4.0 RC1)
  - ALIGNED (OpenMP 4.0 RC1)
## Vectorizable math functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>ceil</td>
<td>fabs</td>
<td>round</td>
</tr>
<tr>
<td>acosh</td>
<td>cos</td>
<td>floor</td>
<td>sin</td>
</tr>
<tr>
<td>asin</td>
<td>cosh</td>
<td>fmax</td>
<td>sinh</td>
</tr>
<tr>
<td>asinh</td>
<td>erf</td>
<td>fmin</td>
<td>sqrt</td>
</tr>
<tr>
<td>atan</td>
<td>erfc</td>
<td>log</td>
<td>tan</td>
</tr>
<tr>
<td>atan2</td>
<td>erfinv</td>
<td>log10</td>
<td>tanh</td>
</tr>
<tr>
<td>atanh</td>
<td>exp</td>
<td>log2</td>
<td>trunc</td>
</tr>
<tr>
<td>cbrt</td>
<td>exp2</td>
<td>pow</td>
<td></td>
</tr>
</tbody>
</table>

Also float versions, such as `sinf()`

Uses short vector math library, `libsvml`
Compiler-based Automatic Vectorization

- You write your code in a vectorizable friendly form
  - Compiler vectorizes your code for SIMD execution without hints
  - Program correctness is paramount – the compiler makes conservative assumptions
  - Odds of success higher when code looks like Fortran – no branches, no function calls
- If your code is ANSI compliant, use the “–ansi-alias” compiler option
- Use –vec-report2 to learn if autovectorization occurred, and –vec-report3 to learn why vectorization did not occur
- Robust
  - Code written to vectorize without hints should vectorize on other Intel® architectures
Vectorization Reports

• By default, both host and target compilations may generate messages for the same loop, e.g.

• icc -vec-report2 test_vec.c

• test_vec.c(10): (col. 1) remark: LOOP WAS VECTORIZED.

• test_vec.c(10): (col. 1) remark: *MIC* LOOP WAS VECTORIZED.

• To get a vectorization report for the offload target compilation, but not for the host compilation:

• icc –vec-report0 –offload-option,mic,compiler,”-vec-report2 test_vec.c

• test_vec.c(10): (col. 1) remark: *MIC* LOOP WAS VECTORIZED.

• test_vec.c(20): (col. 1) remark: *MIC* loop was not vectorized: existence of vector dependence.

• test_vec.c(20): (col. 1) remark: *MIC* PARTIAL LOOP WAS VECTORIZED.
Common vectorization messages

“Loop was not vectorized” because:

- “Low trip count”
- “Existence of vector dependence”
  - Possible dependence of one loop iteration on another, e.g.
    for (j=1; j<MAX; j++) a[j] = a[j] + c * a[j-n];
- "vectorization possible but seems inefficient"
- “Not Inner Loop”

- It may be possible to overcome these using switches, pragmas, source code changes or explicit vector programming
Vector instructions

Compile with –S to see assembly code (if you like)

• A vectorized loop contains instructions like
  
  vfmadd213ps %zmm23, %zmm8, %zmm2 # fma instruction
  vaddps %zmm25, %zmm2, %zmm0 # single precision add

• In a scalar loop, these instructions will be masked, e.g.
  
  vfmadd213ps %zmm17, %zmm20, %zmm1{%k1}
  vaddps %zmm23, %zmm1, %zmm0{%k1}

• Example of vectorized math function for Intel® MIC architecture:
  
  call __svml_sinf16 # calculates sin(x) for 16 floats
  call __svml_sinf16_mask
Requirements for Auto-Vectorization

- Innermost loop of nest (a few simple exceptions)
- Straight-line code (masked assignments OK)
- Avoid:
  - Function/subroutine calls (unless inlined or vector)
  - Non-mathematical operators
  - Data-dependent loop exit conditions
    - Iteration count should be known at entry to loop
  - Loop carried data dependencies (Reduction loops OK)
  - Non-contiguous data (indirect addressing; non-unit stride)
    - Inefficient
  - Inconsistently aligned data
- Directives/pragmas can help:
  - #pragma ivdep ...... ignore potential dependencies
  - #pragma vector always ignore efficiency heuristics
  - aligned assume data aligned
  - Compiler can generate runtime alignment and dependency tests for simple loops (but less efficient)
- See http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/
Semi-automatic Vectorization by Annotation

• The compiler makes conservative assumptions when attempting vectorization

• The programmer can provide some directives to loosen those assumptions
  – The compiler takes the directives and compares them with its analysis of the code

• Some annotation indicates mandatory actions
  – Result in an error message if not met

• Some annotation provide relatively fine control

• Some annotation can have unintended consequences.

• Convincing the compiler to vectorize hot code is very important for achieving best performance from the Intel® MIC Architecture
## Semi-automatic Vectorization by Annotation - Options

<table>
<thead>
<tr>
<th><strong>#pragma</strong></th>
<th><strong>Semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma ivdep !DEC$ ivdep ! Fortran</td>
<td><strong>Ignore vector dependences</strong> unless they are proven by the compiler</td>
</tr>
<tr>
<td>#pragma vector always [assert]</td>
<td><strong>If the loop is vectorizable, ignore any benefit analysis</strong> If the loop did not vectorize, give a compile-time error message via assert</td>
</tr>
<tr>
<td>#pragma novector</td>
<td>Specifies that a <strong>loop should never be vectorized</strong>, even if it is legal to do so, when avoiding vectorization of a loop is desirable (when vectorization results in a performance regression)</td>
</tr>
<tr>
<td>#pragma vector aligned / unaligned</td>
<td>Instructs the compiler to <strong>use aligned (unaligned) data movement instructions</strong> for all array references when vectorizing</td>
</tr>
<tr>
<td>#pragma vector temporal / nontemporal</td>
<td>Directs the compiler to <strong>use temporal/non-temporal</strong> (that is, streaming) <strong>stores</strong> on systems based on IA-32 and Intel® 64 architectures; optionally takes a comma separated list of variables</td>
</tr>
</tbody>
</table>
Semi-automatic Vectorization by Annotation
- #pragma simd

- **Syntax:** #pragma simd [<clause-list>]
  - Mechanism to force vectorization of a loop
  - Programmer: asserts a loop will be vectorized
  - Compiler: vectorizes the loop if possible or gives an error

<table>
<thead>
<tr>
<th>Clause</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No clause</td>
<td>Enforce vectorization of innermost loops; ignore dependencies etc</td>
</tr>
<tr>
<td>vectorlength ( (n_1[, n_2]... )</td>
<td>Select one or more vector lengths (range: 2, 4, 8, 16) for the vectorizer to use.</td>
</tr>
<tr>
<td>private ( (\text{var}_1, \text{var}_2, ..., \text{var}_N) )</td>
<td>Scalars private to each iteration. Initial value broadcast to all instances. Last value copied out from the last loop iteration instance.</td>
</tr>
<tr>
<td>linear ( (\text{var}_1: \text{step}_1, ..., \text{var}_N: \text{step}_N) )</td>
<td>Declare induction variables and corresponding positive integer step sizes (in multiples of vector length)</td>
</tr>
<tr>
<td>reduction ( (\text{operator}:\text{var}_1, \text{var}_2,..., \text{var}_N) )</td>
<td>Declare the private scalars to be combined at the end of the loop using the specified reduction operator</td>
</tr>
<tr>
<td>[no]assert</td>
<td>Direct compiler to assert when the vectorization fails. Default is not to assert for SIMD pragma.</td>
</tr>
</tbody>
</table>
C/C++ Vector Classes

• The Intel® C/C++ Compiler provides C++ Classes that wrap Intel® MIC Architecture vector registers and vector intrinsics
  – Class interface for _mm512
  – Class constructors use broadcast intrinsic functions
  – Loaded operator for basic arithmetic and bitwise operations: +,-,*, /, &, |, !, ^
  – Provide transcendental functions interface – exp(a) wraps __mm512_exp_ps(a)
  – Defined reduction operations such as reduce_add(), reduce_and(), reduce_min()

• Classes
  – F32vec16, F64vec8, I32vec16, Is32vec16, Iu32vec16, I64vec8

• You will need to use #ifdef __MIC__ to contain Intel® MIC Architecture-specific class usage
C/C++ Vector Classes - Benefits

• Lets you write C/C++ objects that give intrinsic-level performance
• Enables array indexing [] operations
• Use operator overloading for operations among objects:
  \[ c = a + b; \]
• Scalar-object requires converting Scalar to vector using broadcast primitives
• Compiler performs full optimization before code generation
  – In contrast, intrinsics bypass compiler optimization (except register allocation) and are translated directly into Intel® MIC Architecture assembly code
C/C++ Vector Classes - Example

float a[16], r[16];
...
for i = 0 to 15
  r[i] = exp(a[i])

Intrinsics - #include <immintrin.h>
...
__m512 vr, va = &a[0];
vr = __mm512_exp_ps(va);
*(__mm512 *)&(r[0]) = vr;

Vector Classes - #include <micvec.h>
...
F32vec16 vr, va = &a[0];
vr = exp(va);
*(F32vec16 *)&(r[0]) = vr;
C++ Vector Classes and SIMD Generic Programming

- The Intel Compiler provides vector classes for all SIMD lengths
  - Support SSE2 and later 128-bit SIMD -- F32vec4, F64vec2
  - Support Intel AVX-based 256-bit SIMD -- F32vec8, F64vec4
  - Just add #include <dvec.h>

- SIMD Function definitions can abstract out SIMD length
  - Create a template that takes a Vector Class, and fundamental type as inputs
    - Instead of F32vec16 foo(F32vec16 a), only on Intel MIC architecture
    - Try generic SIMDType foo_t<SIMDType, BasicType>(SIMDType a)
    - F32vec4 res = foo_t<F32vec4, float>(a) while working on NHM/WSM
    - F32vec8 res = foo_t<F32vec8, float>(a) while working on Sandy Bridge
    - F32vec16 res = foo_t<F32vec16, float>(a) while working on Knights Ferry
  - Compiler creates a version of the template for each class the user instantiates
    - int laneN = sizeof(SIMDType)/sizeof(BasicType); // the num. of SIMD lanes
    - int alignN = sizeof(SIMDType)/sizeof(char); // minimum SIMD alignment
    - SIMDType Tvec = *(SIMDType*)&Tmem[0]; // read SIMD-full of data from Tmem
    - *(SIMDType *)&(Tmem[0]) = Tvec; //write SIMD-full of data to Tmem, which point to BasicType

- Benefit
  - Same code template can create different binaries on different architectures
  - Same code template for single precision and double precision
  - Uses vector class constructor/methods for intrinsic function calls
Compiler Intrinsic Functions for the Intel® MIC Architecture

• C interface to Intel® MIC Architecture assembly
  – Advantage over inline assembly in that the compiler can optimize interaction with other code (register allocation)
  – Operates on C data types
    o __m512 (__m512, __m512i, and __m512d)
    o __mmask, 16bit mask registers

• Conventions
  – __cdecl _mm512_mask_<vop>_<suffix>(v1_old, k1, v2, v3);
  – __cdecl _mm512_<vop>_<suffix>(v1_old, v2, v3);
  – __cdecl _mm512_<vop>_<suffix>(v2, v3);
  – <vop> basic vector operation; e.g., add for add, sub for subtract
  – <suffix> type of data the instruction operates on
    o ps: packed single-precision; pd: packed double-precision

• Use #ifdef __MIC__ to contain Intel® MIC Architecture-specific intrinsics

• Please refer to the Intel® Composer XE 2011 for Linux* Including Intel® MIC Architecture User's Guide for a complete list of all intrinsic functions
Compiler Intrinsic Functions for the Intel® MIC Architecture - Example

```c
#include <immintrin.h>
__m512i z, x, y;
__mmask k = _mm512_cmplt_ps(a, b);
z = _mm512_mask_mul_ps(x, k, x, y);

for i = 1 to 16
    if a[i] < b[i]
        z[i] = x[i] * y[i];
```
Which Technique Should I use to add SIMD-based Data Parallelism to my Code?

- It depends...but in very general terms:
  - If you can *let Intel maintain* SIMD operation
    - Use Intel® MKL to automatically support future generations of Intel® hardware
  - For *smaller blocks of code* and loops
    - Try compiler-based full autovectorization & semi-automatic vectorization by annotation
  - For *larger blocks of code*
    - Try Intel® Cilk™ Plus array notation
      - Forces you to code in a vectorization-friendly manner, which is also good for “future-proofing” your code
      - Can abstract away many vector length considerations
  - Need to *maintain the greatest control* over vectorization
    - C++ Vector classes and intrinsics
      - But someone has to maintain this code and add support as new architectures come out
How to Align Data

● Allocate memory on heap aligned to n byte boundary:
  ```c
  void* _mm_malloc(int size, int n)
  int posix_memaligned(void **p, size_t n, size_t size)
  ```

● Alignment for variable declarations:
  ```c
  __attribute__((aligned(n))) var_name or __declspec(align(n)) var_name
  ```

● And tell the compiler...
  ```c
  #pragma vector aligned
  ```
  • Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
    – May cause fault if data are not aligned
  ```c
  __assume_aligned(array, n)
  ```
    – Compiler may assume array is aligned to n byte boundary

n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE
How to Align Data (Fortran)

• align array on an “n”-byte boundary (n must be a power of 2)
• !dir$ attributes align:n :: array
  – Works for dynamic, automatic and static arrays (not in common)
• For a 2D array, choose column length to be a multiple of n, so that consecutive columns have the same alignment (pad if necessary)
• -align array64byte compiler tries to align all array types

• And tell the compiler...
  • !dir$ vector aligned
  • Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
  – May cause fault if data are not aligned
  – !dir$ assume_aligned array:n [,array2:n2, …]
  Compiler may assume array is aligned to n byte boundary

n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE
Prefetching - automatic

- Compiler prefetching is on by default for the Intel® Xeon Phi™ coprocessor at –O2 and above
  - Prefetches issued for regular memory accesses inside loops
  - But not for indirect accesses  a[index[i]]
  - More important for Intel Xeon Phi coprocessor (in-order) than for Intel® Xeon® processors (out-of-order)
  - Very important for apps with many L2 cache misses
- Use the compiler reporting options to see detailed diagnostics of prefetching per loop
  - -opt-report-phase hlo -opt-report 3  e.g.
    - Total # of lines prefetched in main for loop at line 49=4
    - Using noloc distance 8 for prefetching unconditional memory reference in stmt at line 49
    - Using second-level distance 2 for prefetching spatial memory reference in stmt at line 50
- -opt-prefetch=n (4 = most aggressive) to control
- -opt-prefetch=0 or –no-opt-prefetch to disable
Prefetching - manual

• Use intrinsics

```c
_mm_prefetch((char *) &a[i], hint);
```

See xmmmintrin.h for possible hints (for L1, L2, non-temporal, ...)

```c
MM_PREFETCH(A, hint)  // for Fortran
```

• But you have to figure out and code how far ahead to prefetch

• Also gather/scatter prefetch intrinsics, see zmmmintrin.h and compiler user guide, e.g. _mm512_prefetch_i32gather_ps

• Use a pragma / directive (easier):

```c
#pragma prefetch a [:hint[:distance]]
```

```
!DIR$ PREFETCH A, B, ...
```

• You specify what to prefetch, but can choose to let compiler figure out how far ahead to do it.

• Hardware L2 prefetcher is also enabled by default

• If software prefetches are doing a good job, then hardware prefetching does not kick in
Vectorization Summary

• The importance of SIMD parallelism is increasing
  – Moore’s law leads to wider vectors as well as more cores
  – Don’t leave performance “on the table”
  – Be ready to help the compiler to vectorize, if necessary
    o With compiler directives and hints
    o With explicit vector programming
    o Use Intel® VTune™ Amplifier XE to find the best places (hotspots) to focus your efforts - tomorrow’s presentation

• No need to re-optimize vectorizable code for new processors
  o Typically a simple recompilation
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY
ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS
DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR
IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES
RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY
PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core,
VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

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