Optimizing for the Intel® Xeon Phi™ Coprocessor
based on the same, familiar tools and methods that you already use for multi-core systems

Martyn Corden
Intel Technical Consulting Engineer
Introduction

• Intel’s C/C++ and Fortran compilers support the same optimization features for the Intel® Xeon Phi™ coprocessor as for the host processor.
  • SIMD optimizations
    – Vectorization
    – Intel® Cilk™ Plus
  • OpenMP*
  • Other loop optimizations, optimized math libraries, IPO, ...

Programming and optimizing for the Intel® Xeon Phi™ Coprocessor is a simple extension of programming and optimizing for the CPU host
Parallel programming is the same on coprocessor and host
SIMD 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**

*Other brands and names are the property of their respective owners.*
Support for SIMD Parallelism

For good performance, it’s not sufficient to use all the cores, you need to use the 512 bit SIMD registers and instructions

Vector classes and intrinsics are supported for C/C++
- See micvec.h and zmmmintrin.h in the include/mic directory
- Just include <immintrin.h>, the compiler takes care of the rest.

Auto-vectorization for Intel® MIC architecture works just like for SSE or AVX on the host
- Data alignment should be to 512 bits, instead of 128 or 256

Because of the greater SIMD width, vectorization is even more important on Intel® MIC architecture than on Intel® Xeon® processors. The Intel compiler now supports

Explicit Vector Programming
- Via Intel® Cilk™ Plus language extensions
- Via the SIMD constructs from OpenMP 4.0 RC2
Auto-vectorization

• The vectorizer for Intel® MIC architecture works just like for SSE or AVX on the host, for C, C++ and Fortran
  • Enabled at default optimization level (-O2)
  • Data alignment should be to 64 bytes, instead of 16 (see later)
  • More loops can be vectorized, because of masked vector instructions, gather/scatter instructions, fused multiply-add (FMA)
  • Try to avoid 64 bit integers (except as addresses)

• Vectorized loops may be recognized by:
  • Vectorization and optimization reports (simplest), e.g.
    -vec-report2 or -opt-report-phase hpo
  • Unmasked vector instructions (there are no separate scalar instructions; masked vector instructions are used instead)
  • Gather & scatter instructions
  • Math library calls to libsvml
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)
- 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/
## Vectorizable math functions

<table>
<thead>
<tr>
<th>Function 1</th>
<th>Function 2</th>
<th>Function 3</th>
<th>Function 4</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`
Problems with Pointers

- Hard for compiler to know whether arrays or pointers might be aliased (point to the same memory location)
  - Aliases may hide dependencies that make vectorization unsafe
- In simple cases, compiler may generate vectorized and unvectorized loop versions, and test for aliasing at runtime
- Otherwise, compiler may need help:
  - `-fargument-noalias` & similar switches
  - Use Intel® Cilk™ Plus array notation
  - "restrict" keyword with `-restrict` or `-std=c99` or by inlining
    - and now `__restrict__`
  - `#pragma ivdep` asserts no potential dependencies
    - Compiler still checks for proven dependencies
  - `#pragma simd` asserts no dependencies, period (see later)

```c
void saxpy (float *x, float *y, float* restrict z, float *a, int n) {
  #pragma ivdep
  for (int i=0; i<n; i++) z[i] = *a*x[i] + y[i];
}
```
Intel® Compilers:
some useful loop optimization pragmas/directives

- **IVDEP**
  - Ignore vector dependency
- **LOOP COUNT**
  - Advise typical iteration count(s)
- **UNROLL**
  - Suggest loop unroll factor
- **DISTRIBUTE POINT**
  - Advise where to split loop
- **VECTOR**
  - **Aligned**
    - Assume data is aligned
  - **Always**
    - Override cost model
  - **Nontemporal**
    - Advise use of streaming stores
- **NOVECTOR**
  - Do not vectorize
- **NOFUSION**
  - Do not fuse loops
- **INLINE/FORCEINLINE**
  - Invite/require function inlining
- **SIMD ASSERT**
  - \"vectorize or die\" 😊 (see later)

Use where needed to help the compiler, guided by optimization reports
How to Align Data

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

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

And tell the compiler...

#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

__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
Vectorization and Intel® Cilk™ Plus

Vectorization is so important ➔ consider explicit vector programming

- Intel® Cilk™ Plus array notation
  - Compiler can assume LHS does not alias RHS (unlike Fortran)
    \[
    r[n:n] = \text{sqrtf}(x[0:n]^2 + y[0:n]^2);
    \]
- and elemental functions
  - Allow vectorization over function calls, without inlining
    ```c
    __attribute__((vector)) float myfun(float a, float x, float y){...
    z[:] = myfunl(a[:], b[:], c[:]);
    ```
- #pragma simd
  - Directs compiler to vectorize if at all possible
  - Overrides all dependencies and cost model
    - More aggressive than pragmas ivdep and vector always
  - Semantics modeled on OpenMP parallel pragmas
    - Private and reduction clauses required for correctness
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.
Explicit Vector Programming: Intel® Cilk™ Plus pragma example

Using `#pragma simd` (C/C++) or ![DIR]$ SIMD (Fortran) or `#pragma omp simd` (OpenMP* 4.0 RC2)

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];
    }
}

loop was not vectorized: existence of vector dependence.

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

SIMD LOOP WAS VECTORIZED.

• Use when you **KNOW** that a given loop is safe to vectorize
• The Intel Compiler will vectorize if possible (will ignore dependency or efficiency concerns)
• Minimizes source code changes needed to enforce vectorization
Clauses for SIMD directives

The programmer (i.e. you!) is responsible for correctness
- Just like for race conditions in OpenMP* loops

Available clauses (both OpenMP and Intel versions)
- **PRIVATE**
- **LASTPRIVATE** --- like OpenMP
- **REDUCTION**
- **COLLAPSE** (OpenMP 4.0 RC2 only; for nested loops)
- **LINEAR** (additional induction variables)
- **SAFELEN** (OpenMP 4.0 RC2 only)
- **VECTORLENGTH** (Intel only)
- **ALIGNED** (OpenMP 4.0 RC2 only)
- **ASSERT** (Intel only; default for OpenMP 4.0 RC2)
Vector Elemental Function

Compiler generates vector version of a scalar function that can be called from a vectorized loop:

```c
__attribute__((vector(uniform(y, xp, yp))))
float func(float x, float y, float xp, float yp) {
    float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
    denom = 1./sqrtf(denom);
    return denom;
}
```

```
#pragma simd  private(x)  reduction(+:sumx)
for (i=1; i<nx; i++)  {
    x = x0 + (float)i * h;
    sumx = sumx + func(x, y ,xp, yp);
enddo
```

y, xp and yp are constant, x can be a vector

These clauses are required for correctness, just like for OpenMP*

SIMD LOOP WAS VECTORIZED.
Clauses for Vector Functions

__attribute__((vector)) (Intel)
#pragma omp declare simd (OpenMP* 4.0 RC2)

Available clauses (both OpenMP and Intel versions)

- LINEAR (additional induction variables)
- UNIFORM (arguments that are loop constants)
- REDUCTION
- PROCESSOR (Intel)
- VECTORLENGTH (Intel)
- MASK / NOMASK (Intel)
- INBRANCH / NOTINBRANCH (OpenMP 4.0 RC2)
- SIMDLEN (OpenMP 4.0 RC2)
- ALIGNED (OpenMP 4.0 RC2)
SIMD 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
  – With compiler directives and hints
  – With explicit vector programming
  – 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
  – Typically a simple recompilation
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
  - \(-\text{opt-report-phase hlo} -\text{opt-report 3}\) 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
  - \(-\text{opt-prefetch=\text{n}} (4 = \text{most aggressive}) \) to control
  - \(-\text{opt-prefetch=\text{0}} \) or \(-\text{no-opt-prefetch} \) to disable
Prefetching - manual

• Use intrinsics

  \_mm\_prefetch((char \*) \&a[i], hint);
  See xmmmintrin.h for possible hints (for L1, L2, non-temporal, …)

  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):

  #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
Streaming Stores

• Write directly to memory bypassing cache
  • for “nontemporal” data that are not read and will not be reused
  • avoids “read for ownership” to get line into cache
    - Reduces memory bandwidth requirements
    - Keeps cache available for useful work, avoids “pollution”
• #pragma vector nontemporal (v1, v2, ...) hint to compiler

• No Streaming Stores:
  448 Bytes read/write per iteration
• With Streaming Stores:
  320 Bytes read/write per iteration
• –vec-report6 shows what the compiler did

```c
for (int chunkBase = 0; chunkBase < OptPerThread; chunkBase += CHUNKSIZE) {
    #pragma simd vectorlength(CHUNKSIZE)
    #pragma vector aligned
    #pragma vector nontemporal (CallResult, PutResult)
    for(int opt = chunkBase; opt < (chunkBase+CHUNKSIZE); opt++)
    {
        float CNDD1, CNDD2;
        float CallVal =0.0f, PutVal = 0.0f;
        float T = OptionYears[opt];
        float X = OptionStrike[opt];
        float S = StockPrice[opt];
        ...
        CallVal = S * CNDD1 - XexpRT * CNDD2;
        PutVal = CallVal + XexpRT - S;
        CallResult[opt] = CallVal;
        PutResult[opt] = PutVal;
    }
}
```

bs_test_sp.c(215): (col. 4) remark: vectorization support: streaming store was generated for CallResult.
bs_test_sp.c(216): (col. 4) remark: vectorization support: streaming store was generated for PutResult.
More reports

-opt-report-phase hlo
summarizes loop optimizations including loop interchange, fusion, distribution, unrolling, multi-versioning, cache blocking, prefetching, etc.

-opt-report-phase ipo_inl
summarizes function inlining

-opt-report-phase offload
gives a compile time summary of which data are copied to and from the coprocessor

OFFLOAD_REPORT=2 (environment variable on host)
gives run-time summary of data copied to and from the coprocessor and computation time on the coprocessor

Main optimization opportunity for the data offload:
Don’t transfer data that you don’t need! (e.g., copy in but not out)
OpenMP* on the Coprocessor

• The basics work just like on the host CPU
  • For both native and offload models
  • Need to specify -openmp

• There are 4 hardware thread contexts per core
  • Need at least 2 x ncore threads for good performance
    - For all except the most memory-bound workloads
    - Often, 3x or 4x (number of available cores) is best
    - Very different from hyperthreading on the host!
    - -opt-threads-per-core=n advises compiler how many threads to optimize for

• If you don’t saturate all available threads, be sure to set KMP_AFFINITY to control thread distribution
OpenMP defaults

- **OMP_NUM_THREADS** defaults to
  - 1 x ncore for host (or 2x if hyperthreading enabled)
  - 4 x ncore for native coprocessor applications
  - 4 x (ncore-1) for offload applications
    - one core is reserved for offload daemons and OS

- Defaults may be changed via environment variables or via API calls on either the host or the coprocessor
Target OpenMP environment (offload)

Use target-specific APIs to set for coprocessor target only, e.g.

omp_set_num_threads_target() (called from host)
omp_set_nested_target() etc

• Protect with #ifdef __INTEL_OFFLOAD, undefined with –no-offload
• Fortran: USE MIC_LIB and OMP_LIB
  C: #include <offload.h>

Or define MIC – specific versions of env vars using

MIC_ENV_PREFIX=MIC (no underscore)

• Values on MIC no longer default to values on host
• Set values specific to MIC using

export MIC_OMP_NUM_THREADS=120 (all cards)
export MIC_2_OMP_NUM_THREADS=180 for card #2, etc
export MIC_3_ENV="OMP_NUM_THREADS=240|KMP_AFFINITY=balanced"
Stack Sizes for Coprocessor

For the main thread, (thread 0), default stack limit is 12 MB
- In offloaded functions, stack is used for local or automatic arrays and compiler temporaries
- To increase limit, export MIC_STACKSIZE (e.g. =100M)
  - default unit is K (Kbytes)
- For native apps, use ulimit –s  (default units are Kbytes)

For worker threads: default stack size is 4 MB
- Space only needed for those local variables or automatic arrays or compiler temporaries for which each thread has a private copy
- To increase limit, export OMP_STACKSIZE=10M  (or as needed)
- Or use dynamic allocation (may be less efficient)

Typical error message if stack limits exceeded:
  offload error: process on the device 0 was terminated by SEGFAULT
Thread Affinity Interface

Allows OpenMP threads to be bound to physical or logical cores

• export environment variable `KMP_AFFINITY=`
  - physical use all physical cores before assigning threads to other logical cores (other hardware thread contexts)
  - compact assign threads to consecutive h/w contexts on same physical core (eg to benefit from shared cache)
  - scatter assign consecutive threads to different physical cores (eg to maximize access to memory)
  - balanced blend of compact & scatter (currently only available for Intel® MIC Architecture)

• Helps optimize access to memory or cache
• Particularly important if all available h/w threads not used
  - else some physical cores may be idle while others run multiple threads

• See compiler documentation for (much) more detail
**Example – share work between coprocessor and host using OpenMP**

```c
omp_set_nested(1);
#pragma omp parallel private(ip)
{
#pragma omp sections
{
#pragma omp section
/*
   use pointer to copy back only part of potential array,
to avoid overwriting host */
#pragma omp offload target(mic) in(xp) in(yp) in(zp) out(ppot:length(np1))
#pragma omp parallel for private(ip)
   for (i=0;i<np1;i++)
       ppot[i] = threed_int(x0,xn,y0,yn,z0,zn,nx,ny,nz,xp[i],yp[i],zp[i]);
}
#pragma omp section
#pragma omp parallel for private(ip)
   for (i=0;i<np2;i++)
       pot[i+np1] =
           threed_int(x0,xn,y0,yn,z0,zn,nx,ny,nz,xp[i+np1],yp[i+np1],zp[i+np1]);
}
```

Top level, runs on host

- Runs on coprocessor
- Runs on host
Debugging with OpenMP*

• Test your OpenMP* threaded app on the host before moving to the coprocessor

• Debug with –O0 –openmp
  • Unlike most other optimizations, OpenMP threading is not disabled at –O0

• If debugging with print statements
  • print out the thread number with omp_get_thread_num()
  • the internal I/O buffers are threadsafe (with –openmp), but the order of print statements from different threads is undetermined.

• Use Intel® Inspector XE on the host to detect race conditions and other threading and memory errors
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

Trapping of floating-point exceptions in vector instructions is not supported

The bits of the SIMD floating-point control word that mask/unmask floating-point protections are protected

• If you try to unmask exceptions, your app will seg fault
• Unmasking by compiler switches such as –fp-trap or –fpe0 is disabled for native builds or for target part of an offload build
• The exception flags still get set, and you can test on these
• Otherwise, the computation just continues with QNaNs, infinities, etc

- fp-model except or –fp-model strict preserves exception semantics
  - Generates x87 instead of vector instructions, big performance impact
  - May be useful for debugging

Denormals are supported

• Needs –no-ftz or –fp-model precise (like on host)
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

-fp-model fast=2 enables some more aggressive optimizations
• Faster inlined versions of some math functions
  – May not give standard behavior for extreme or exceptional arguments

Floating-point results on Intel® Xeon Phi™ may not be bit-for-bit identical to results obtained on Intel® Xeon processors
• Most common cause is fused multiply-add (FMA) instructions
  – Not disabled by -fp-model precise
  – Can disable for testing with -no-fma
    – With some impact on performance
• Implementation of math functions might also differ
• To get close, try -fp-model precise -no-fma -fimf-precision=high
  – But most parallel reductions will still cause differences
Resources

http://software.intel.com/mic-developer
- Developer’s Quick Start Guide
- Programming Overview
- User Forum at


Intel® Composer XE 2013 for Linux* User and Reference Guides

Intel Premier Support http://premier.intel.com
Questions?
Legal Disclaimer & Optimization Notice

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