An Introduction to the Intel® Xeon Phi™ Coprocessor

Robert Reed
Intel Technical Consulting Engineer
Statements and Assumptions

This workshop is not comprehensive training
• Looking at breath and not depth
• We will limit the depth of questions
  – Contact your Intel representative for in-depth training

All attendees are experienced software developers
• Single-machine parallel programming
• Production environment

Attendees will be writing code for the Intel® Xeon Phi™ Coprocessor
Topics covered during these sessions

Introduction to hardware and software architecture
Native and offload execution
Compiling for Intel® Xeon Phi™ coprocessors
Optimization, Parallelization, Vectorization
Debugging

Intel® Math Kernel Library (Intel® MKL)
MPI – Programming and analysis
Performance Analysis using Intel® VTune™ Amplifier XE
Questions and wrap-up
Fundamentals: reviewing the hardware and software layers and execution models

High-level overview of the Intel® Xeon Phi™ platform

• Hardware
• Software

Intel Xeon Phi Coprocessor programming considerations

• Native
• Offload
  - Explicit block data transfer
  - Offloading with Virtual Shared Memory
• Programming models: a list of tools
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

Scalar unit based on Intel® Pentium® processor family
- Two pipelines (U and V)
  - Dual-issue on scalar instructions
- Scalar pipeline 1 clock latency
- 64-bit data path

4 hardware threads per core
- Thread context: GPRs, ST0-7, etc.
- “Smart” round-robin scheduling
  - Prefetch buffers 2 instr-bundles / context
  - Next ready context selected in order
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

Instruction decoder is fully pipelined but was designed as a 2-cycle unit
- Allows significant increase to maximum core frequency, but...
  - Core cannot issue instructions from same context in adjacent cycles
  - Means minimum two threads per core to achieve full compute potential
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread vector unit

A new vector unit: 512 bits wide!

- 32 512-bit vector registers per context
  - Each holds 16 floats or 8 doubles
  - ALUs support int32/float32 operations, float64 arithmetic, int64 logic ops
  - Ternary ops including Fused-Multiply-Add
  - Broadcast/swizzle support
  - 8 vector mask registers for per lane conditional operations
  - Most ops: 4-cycle latency, 1-cycle thrput
    - Matches 4-cycle round robin of integer unit
  - Mostly IEEE 754 2008 compliant
  - Not supported: MMX™ technology, Streaming SIMD Extensions (SSE), Intel® Advanced Vector Extensions (Intel AVX)
Individual cores are tied together via fully coherent caches into a bidirectional ring

**L1** 32K I/D-cache per core
- 1 cycle access latency
- 3 cycle addr-gen interlock
- 8-way associativity
- 64-byte cache line
- ~38 concurrent access/core

**L2** 512K cache per core
- 11 cycle raw latency
- 8-way associativity
- 64-byte cache line
- Streaming HW prefetcher
- ~38 concurrent access/core

**GDDR5 Memory**
- 16 32-bit channels
- Up to 5.5 GT/sec
- 8 GB - 300ns access

**Bidirectional ring**
- 200 GB/sec
- Distributed Tag Directory (DTD)
- reduces ring snoop traffic
- Gen2x16 PCI Express* 64-256 byte packets
- peer-to-peer R/W
Intel® Xeon Phi™ Coprocessor can run as an accelerator for offloaded host computation

Advantages
- More memory available
- Better file access
- Host faster on serial code
- Better uses resources

Host-side offload application
- User code
- Offload libraries, user-level driver, user-accessible APIs and libraries

Target-side offload application
- User code
- Offload libraries, user-accessible APIs and libraries

Intel® Xeon Phi™ Coprocessor support libraries, tools, and drivers

Intel® Xeon Phi™ Coprocessor communication and application-launch support

Linux* OS
PCI-E Bus
Or Intel® Xeon Phi™ Coprocessor runs as a native or MPI* compute node via IP or OFED

Advantages
- Simpler model
- No directives
- Easier port
- Good kernel test

ssh or telnet connection to coprocessor IP address

User-level code
System-level code

Intel® Xeon Phi™ Coprocessor support libraries, tools, and drivers

User code
Standard OS libraries plus any 3rd-party or Intel libraries

Virtual terminal session

Target-side “native” application

Use if
- Not serial
- Modest memory
- Complex code
- No hot spots

User-level code
System-level code

Intel® Xeon Phi™ Coprocessor communication and application-launch support

Intel® Xeon Phi™ Coprocessor

Host Processor

System-level code
User-level code

Linux* OS
PCI-E Bus

IB fabric

Linux* OS
PCI-E Bus
Or utilize both host and coprocessors in Symmetric Mode using MPI*

**Diagram:**
- **Host Processor**
  - Host-side MPI application
    - User code
    - Standard OS libraries plus any 3rd-party or Intel libraries
  - User-level code
  - System-level code
  - Intel MPI support libraries
  - Linux* OS
  - IB fabric
  - PCI-E Bus
- **Intel® Xeon Phi™ Coprocessor**
  - Target-side MPI application
    - User code
    - Standard OS libraries plus any 3rd-party or Intel libraries
  - User-level code
  - System-level code
  - Intel® Xeon Phi™ Coprocessor MPI support libraries
  - Linux* OS
  - PCI-E Bus

*Other brands and names are the property of their respective owners.
Or, now in Beta, use your Intel® Xeon Phi™ Coprocessor from a Windows* host machine.

Host Processor:
- Available in both offload and MPI coprocessor node configurations
- ssh or telnet connection to coprocessor IP address

Intel® Xeon Phi™ Coprocessor:
- Target-side “native” application
  - User code
  - Standard OS libraries plus any 3rd-party or Intel libraries
- Virtual terminal session
- Not available yet
  - Symmetric Mode
  - Host support for multiple coprocessors

Windows* OS
- System-level code
- User-level code
- Intel® Xeon Phi™ Coprocessor support libraries, tools, and drivers

Linux* OS
- System-level code
- User-level code
- Intel® Xeon Phi™ Coprocessor communication and application-launch support

IB fabric
- PCI-E Bus

Available in both offload and MPI coprocessor node configurations

Copyright © 2013, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Fundamentals: reviewing the hardware and software layers and execution models

High-level overview of the Intel® Xeon Phi™ platform

• Hardware
• Software

Intel Xeon Phi Coprocessor programming considerations

• Native
• Offload
  - Explicit block data transfer
  - Offloading with Virtual Shared Memory
• Programming models: a list of tools
Authenticated users can treat it like another node

```
ssh mic0 top
```

```
Mem: 298016K used, 7578640K free, 0K shrd, 0K buff, 100688K cached
CPU: 0.0% usr 0.3% sys 0.0% nic 99.6% idle 0.0% io 0.0% irq 0.0% sirq
Load average: 1.00 1.04 1.01 1/2234 7265
```

```
PID PPID USER STAT VSZ %MEM CPU %CPU COMMAND
7265 7264 fdkew R 7060 0.0 14 0.3 top
43 2 root SW 0 0.0 13 0.0 [ksoftirqd/13]
5748 1 root S 119m 1.5 226 0.0 ./sep_mic_server3.8
5670 1 micuser S 97872 1.2 0 0.0 /bin/cci_daemon --coiuser=micuser
7261 5667 root S 25744 0.3 6 0.0 sshd: fdkew [priv]
7263 7261 fdkew S 25744 0.3 241 0.0 sshd: fdkew@notty
5667 1 root S 21084 0.2 5 0.0 /sbin/sshd
5757 1 root S 6940 0.0 18 0.0 /sbin/getty -L -l /bin/noauth 1152
1 0 root S 6936 0.0 10 0.0 init
7264 7263 fdkew S 6936 0.0 6 0.0 sh -c top
```

Intel MPSS supplies a virtual FS and native execution

```
sudo scp /opt/intel/composerxe/lib/mic/libiomp5.so root@mic0:/lib64
cscp a.out mic0:/tmp
ssh mic0 /tmp/a.out my-args
```

Add `--mmic` to compiles to create native programs

```
icc -O3 -g --mmic -o nativeMIC myNativeProgram.c
```
Alternately, use the offload capabilities of Intel® Composer XE to access coprocessor

Offload directives in source code trigger Intel Composer to compile objects for both host and coprocessor

#pragma offload target(mic) inout(A:length(2000))
!DIR$ OFFLOAD TARGET(MIC) INOUT(A: LENGTH(2000))

When the program is executed and a coprocessor is available, the offload code will run on that target

• Required data can be transferred explicitly for each offload
• Or use Virtual Shared Memory (_Cilk_shared) to match virtual addresses between host and target coprocessor

Offload blocks initiate coprocessor computation and can be synchronous or asynchronous

#pragma offload_transfer target(mic) in(a: length(2000)) signal(a)
!DIR$ OFFLOAD_TRANSFER TARGET(MIC) IN(A: LENGTH(2000)) SIGNAL(A)
_Cilk_spawn _Cilk_offload asynch-func()
Offload directives are independent of function boundaries

**Host**
Intel® Xeon® processor

```c
f() {
    #pragma offload
    a = b + g();
    h();
}
```

```c
__attribute__((target(mic)))
g() {
    ...
}
```

```c
h() {
    ...
}
```

**Target**
Intel® Xeon Xeon Phi™ coprocessor

```c
f_part1() {
    a = b + g();
}
```

```c
__attribute__((target(mic)))
g() {
    ...
}
```

**Execution**
- If at first offload the target is available, the target program is loaded
- At each offload if the target is available, statement is run on target, else it is run on the host
- At program termination the target program is unloaded
Pragmas and directives mark data and code to be offloaded and executed

<table>
<thead>
<tr>
<th>Pragmas and Directives</th>
<th>C/C++ Syntax</th>
<th>Fortran Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offload pragma</td>
<td>#pragma offload &lt;clauses&gt; &lt;statement&gt;</td>
<td>!dir$ omp offload &lt;clauses&gt; &lt;statement&gt;</td>
</tr>
<tr>
<td></td>
<td>Allow next statement to execute on coprocessor or host CPU</td>
<td>Execute OpenMP* parallel block on coprocessor</td>
</tr>
<tr>
<td>Variable/function offload properties</td>
<td><strong>attribute</strong>((target(mic)))</td>
<td>!dir$ offload &lt;clauses&gt; &lt;statement&gt;</td>
</tr>
<tr>
<td></td>
<td>Compile function for, or allocate variable on, both host CPU and coprocessor</td>
<td>Execute next statement or function on coproc.</td>
</tr>
<tr>
<td>Entire blocks of data/code defs</td>
<td>#pragma offload_attribute(push, target(mic))</td>
<td>!dir$ attributes offload:&lt;mic&gt; :: &lt;ret-name&gt; OR &lt;var1,var2,...&gt;</td>
</tr>
<tr>
<td></td>
<td>#pragma offload_attribute(pop)</td>
<td>Compile function or variable for CPU and coprocessor</td>
</tr>
<tr>
<td>Mark entire files or large blocks of code to compile for both</td>
<td>#pragma offload_attribute(begin &lt;clauses&gt;)</td>
<td>!dir$ offload begin &lt;clauses&gt;</td>
</tr>
<tr>
<td></td>
<td>!dir$ end offload</td>
<td>!dir$ end offload</td>
</tr>
</tbody>
</table>
Options on offloads can control data copying and manage coprocessor dynamic allocation

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple coprocessors</td>
<td><code>target(mic[:unit] )</code></td>
<td>Select specific coprocessors</td>
</tr>
<tr>
<td>Conditional offload</td>
<td><code>if (condition) / mandatory</code></td>
<td>Select coprocessor or host compute</td>
</tr>
<tr>
<td>Inputs</td>
<td><code>in(var-list modifiers_opt)</code></td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>Outputs</td>
<td><code>out(var-list modifiers_opt)</code></td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>Inputs &amp; outputs</td>
<td><code>inout(var-list modifiers_opt)</code></td>
<td>Copy host to coprocessor and back when offload completes</td>
</tr>
<tr>
<td>Non-copied data</td>
<td><code>nocopy(var-list modifiers_opt)</code></td>
<td>Data is local to target</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifiers</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify copy length</td>
<td><code>length(N)</code></td>
<td>Copy N elements of pointer’s type</td>
</tr>
<tr>
<td>Coprocessor memory allocation</td>
<td><code>alloc_if ( bool )</code></td>
<td>Allocate coprocessor space on this offload (default: TRUE)</td>
</tr>
<tr>
<td>Coprocessor memory release</td>
<td><code>free_if ( bool )</code></td>
<td>Free coprocessor space at the end of this offload (default: TRUE)</td>
</tr>
<tr>
<td>Control target data alignment</td>
<td><code>align ( N bytes )</code></td>
<td>Specify minimum memory alignment on coprocessor</td>
</tr>
<tr>
<td>Array partial allocation &amp;</td>
<td><code>alloc ( array-slice )</code> into ( var-expr )</td>
<td>Enables partial array allocation and data copy into other vars &amp; ranges</td>
</tr>
<tr>
<td>variable relocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Offloaded data have some restrictions and directives to channel their transfer

Offload data are limited to scalars, arrays, and “bitwise-copyable” structs (C++) or derived types (Fortran)
• No structures with embedded pointers (or allocatable arrays)
• No C++ classes beyond the very simplest
• Fortran 2003 object constructs also off limits, mostly
• Data exclusive to the coprocessor has no restrictions

Offload data includes all scalars & named arrays in lexical scope, which are copied both directions automatically
• IN, OUT, INOUT, NOCOPY are used to limit/channel copying
• Data not automatically transferred:
  - Local buffers referenced by local pointers
  - Global variables in functions called from the offloaded code
• Use IN/OUT/INOUT to specify these copies – use LENGTH
alloc_if() and free_if() provide a means to manage coprocessor memory allocs

Both default to true: normally coprocessor variables are created/destroyed with each offload

A common convention is to use these macros:

```c
#define ALLOC alloc_if(1)
#define FREE free_if(1)
#define RETAIN free_if(0)
#define REUSE alloc_if(0)
```

To allocate a variable and keep it for the next offload:

```
#pragma offload target(mic) in(p:length(n) ALLOC RETAIN)
```

To reuse that variable and keep it again:

```
#pragma offload target(mic) in(p:length(n) REUSE RETAIN)
```

To reuse one more time, then discard:

```
#pragma offload target(mic) in(p:length(n) REUSE FREE)
```
To handle more complex data structures on the coprocessor, use Virtual Shared Memory

An identical range of virtual addresses is reserved on both host and coprocessor: changes are shared at offload points, allowing:

- Seamless sharing of complex data structures, including linked lists
- Elimination of manual data marshaling and shared array management
- Freer use of new C++ features and standard classes
Virtual Shared Memory uses special allocation to manage data sharing at offload boundaries

Declare virtual shared data using _Cilk_shared allocation specifier

Allocate virtual dynamic shared data using these special functions:

```c
_Offload_shared_malloc(), _Offload_shared_aligned_malloc(),
_Offload_shared_free(), _Offload_shared_aligned_free()
```

Shared data copying occurs automatically around offload sections

- Memory is only synchronized on entry to or exit from an offload call
- Only modified data blocks are transferred between host and coprocessor

Allows transfer of C++ objects

- Pointers are transportable when they point to “shared” data addresses

Well-known methods can be used to synchronize access to shared data and prevent data races within offloaded code

- E.g., locks, critical sections, etc.

This model is integrated with the Intel® Cilk™ Plus parallel extensions

Note: Not supported on Fortran - available for C/C++ only
# Data sharing between host and coprocessor can be enabled using this Intel® Cilk™ Plus syntax

<table>
<thead>
<tr>
<th>What</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td><code>int _Cilk_shared f(int x){ return x+1; }</code></td>
</tr>
<tr>
<td></td>
<td>Code emitted for host and target; may be called from either side</td>
</tr>
<tr>
<td>Global</td>
<td><code>_Cilk_shared int x = 0;</code></td>
</tr>
<tr>
<td></td>
<td>Datum is visible on both sides</td>
</tr>
<tr>
<td>File/Function static</td>
<td><code>static _Cilk_shared int x;</code></td>
</tr>
<tr>
<td></td>
<td>Datum visible on both sides, only to code within the file/function</td>
</tr>
<tr>
<td>Class</td>
<td><code>class _Cilk_shared x {...};</code></td>
</tr>
<tr>
<td></td>
<td>Class methods, members and operators available on both sides</td>
</tr>
<tr>
<td>Pointer to shared data</td>
<td><code>int _Cilk_shared *p;</code></td>
</tr>
<tr>
<td></td>
<td><code>p</code> is local (not shared), can point to shared data</td>
</tr>
<tr>
<td>A shared pointer</td>
<td><code>int * _Cilk_shared p;</code></td>
</tr>
<tr>
<td></td>
<td><code>p</code> is shared; should only point at shared data</td>
</tr>
<tr>
<td>Entire blocks of code</td>
<td><code>#pragma offload_attribute(push, _Cilk_shared)</code></td>
</tr>
<tr>
<td></td>
<td><code>;</code></td>
</tr>
<tr>
<td></td>
<td><code>#pragma offload_attribute(pop)</code></td>
</tr>
<tr>
<td></td>
<td>Mark entire files or blocks of code _Cilk_shared using this pragma</td>
</tr>
</tbody>
</table>
**Intel® Cilk™ Plus syntax can also specify the offloading of computation to the coprocessor**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Example</th>
</tr>
</thead>
</table>
| Offloading a function call       | `x = __Cilk_offload func(y);`  
  `func` executes on coprocessor if possible |
| Offloading asynchronously        | `x = __Cilk_offload_to (card_num) func(y);`  
  `func` *must* execute on specified coprocessor or an error occurs |
| Offloading a parallel for-loop   | `__Cilk_offload __Cilk_for(i=0; i<N; i++){`  
  `a[i] = b[i] + c[i];`  
  `}`  
  Loop executes in parallel on coprocessor.  
  The loop is implicitly “un-inlined” as a function call. |
Intel® Composer XE manages offload code complexity and gives hooks to help developers organize and understand

- The compiler, on detecting offload constructs in a file, will automatically compile the file twice, for host and coprocessor; to suppress, use –no-offload switch

- Offload options can be selectively set for target only
  - offload-option,mic,compiler,”-vec-report2”

- Offload options available for compiler, ld and as

- Dual objs, libs thru xiar & xild merge to single a.out

- To identify unresolved symbols in offload programs
  - offload-option,mic,compiler,”-z defs”

- To compile whole files for offload
  - offload-attribute-target=mic

- Compiler predefined macros mark host/target
  - Target builds define __MIC__
  - __INTEL_OFFLOAD marks host-side builds where offload builds are enabled
Intel® Xeon Phi™ Coprocessor supports a variety of programming models

The familiar Intel development environment is available:

- Intel® Composer: C, C++ and Fortran Compilers
- Parallel Programming Models
  - OpenMP*
  - Intel Threading Building Blocks (Intel TBB)
  - Intel Cilk™ Plus
- Intel MPI Library support for the Intel Xeon Phi™ Coprocessor
  - Use as an MPI node via TCP/IP or OFED
- Intel support for gdb on Intel Xeon Phi Coprocessor
- Intel Performance Libraries (e.g. Intel Math Kernel Library)
  - Three versions: host-only, coprocessor-only, heterogeneous
- Intel VTune™ Amplifier XE for performance analysis
- Standard runtime libraries, including pthreads*
Intel® Xeon Phi™ Coprocessor supports a variety of programming models

The familiar Intel development environment is available:
• Intel® Composer: C, C++ and Fortran Compilers
• Parallel Programming Models
  – OpenMP*
  – Intel Threading Building Blocks (Intel TBB)
  – Intel Cilk™ Plus
• Intel MPI Library support for the Intel Xeon Phi™ Coprocessor
  – Use as an MPI node via TCP/IP or OFED
• Intel support for gdb on Intel Xeon Phi Coprocessor
• Intel Performance Libraries (e.g. Intel Math Kernel Library)
  – Three versions: host-only, coprocessor-only, heterogeneous
• Intel VTune™ Amplifier XE for performance analysis
• Standard runtime libraries, including pthreads*
OpenMP* parallelism can be easily finessed into offload parallelism

OpenMP is a simple way to add parallelism

```c
#pragma omp parallel for
for (i=1; i < N-1; ++i)
    B2[i] = m * B1[i-1] + n * B1[i] + p * B1[i+1];

!$OMP PARALLEL DO
do I=2, N-1
    B2(I) = m * B1(I-1) + n * B1(I) + p * B1(I+1)
end do
```
OpenMP* parallelism can be easily finessed into offload parallelism

OpenMP is a simple way to add parallelism

```c
#pragma omp parallel for
for (i=1; i < N-1; ++i)
    B2[i] = m * B1[i-1] + n * B1[i] + p * B1[i+1];

!$OMP PARALLEL DO
do I=2, N-1
    B2(I) = m * B1(I-1) + n * B1(I) + p * B1(I+1)
end do
```

Offload easily shifts that parallelism to the coprocessor

```c
#pragma omp parallel for
for (i=1; i < N-1; ++i)
    B2[i] = m * B1[i-1] + n * B1[i] + p * B1[i+1];

#pragma offload target(mic: CPID) if (N > Limit)
#pragma omp parallel for
for (i=1; i < N-1; ++i)
    B2[i] = m * B1[i-1] + n * B1[i] + p * B1[i+1];

!DIR$ OFFLOAD BEGIN target(mic : CPID) IF (N .GT. Limit)
!$OMP PARALLEL DO
do I=2, N-1
    B2(I) = m * B1(I-1) + n * B1(I) + p * B1(I+1)
end do
!DIR$ END OFFLOAD
```
Intel offers other parallel models: Intel® Threading Building Blocks

Intel TBB provides parallel_for...

```cpp
parallel_for (blocked_range<float>(0, N-1), [] (const blocked_range<float> &r) {
    for (float &i = r.begin(); i != r.end(); ++i)
        B2[i] = m * B1[i-1] + n * B1[i] + p * B1[i+1];
}
```

...and a whole lot more

scalable memory allocator

parallel_pipeline()

spin_mutexes

queuing_mutexes

thread-safe

concurrent containers

parallel_reduce()

thread local storage

task cancelation

flow graph
Intel offers other parallel models: Intel® Cilk™ Plus

- A simple syntax provides fork-join parallelism
  
  ```cilk
  cilk_for (int i = 1; i < N-1; ++i) {
    B2[i] = m * B1[i-1] + n * B1[i] + p * B1[i+1];
  }
  ```

- As with Intel TBB, arrays are partitioned between threads, work stealing balances loads, splitting work between threads

  - Intel Cilk Plus also has Hyperobjects
  - C/C++ Extensions for Array Notation

  ```cilk
  y[0:M-K] = 0;
  for (j=0; j<K; j++){
    y[0:M-K] += x[j:M-K] * c[j];
  }
  ```
For further reading, plus video tutorials

Check out the Intel® Developer Zone for Intel® Xeon Phi™ coprocessors

http://software.intel.com/mic-developer
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