Intel® Many Integrated Core (MIC) Architecture

Offload Compilation
Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPEITY 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.

- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards 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.
- Sandy Bridge and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user
- 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. For more information go to http://www.intel.com/performance
- Intel, Core, Xeon, VTune, Cilk, Intel and Intel Sponsors of Tomorrow. and Intel Sponsors of Tomorrow. logo, and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright ©2011 Intel Corporation.

- Hyper-Threading Technology: Requires an Intel® HT Technology enabled system, check with your PC manufacturer. Performance will vary depending on the specific hardware and software used. Not available on all Intel® Core™ processors. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading

- Intel® 64 architecture: Requires a system with a 64-bit enabled processor, chipset, BIOS and software. Performance will vary depending on the specific hardware and software you use. Consult your PC manufacturer for more information. For more information, visit http://www.intel.com/info/em64t

- Intel® Turbo Boost Technology: Requires a system with Intel® Turbo Boost Technology capability. Consult your PC manufacturer. Performance varies depending on hardware, software and system configuration. For more information, visit http://www.intel.com/technology/turboboost
Module Goals

- You can list two pragmas or keywords that cause code to run on the Intel® MIC Architecture
- You can explain how data movement is controlled between the host and card
Module Outline

• High-Level overview of the Heterogeneous Compiler
  – Overview
  – Offload using Explicit Copies
  – Offload using Implicit Copies
  – Comparison of Techniques
  – Heterogeneous Compiler command-line options
  – Intel® MIC Architecture Native Compiler
Offload Compilation

• High-Level overview of the Heterogeneous Compiler
  – Overview
  – Offload using Explicit Copies
  – Offload using Implicit Copies
  – Comparison of Techniques
  – Heterogeneous Compiler command-line options
  – Intel® MIC Architecture Native Compiler
Heterogeneous Compiler – Programming Model Overview

• Add pragmas and new keywords to working host code to make sections of code run on the Intel® MIC Architecture coprocessor
  – Similar to adding parallelism to serial code using OpenMP® pragmas or Intel® Cilk™ Plus keywords
  – Again, the Intel® MIC Architecture is best suited for highly-parallel vectorized code

• The Intel® Compiler generates code for both target architectures at once
  – The resulting binary runs whether or not a coprocessor is present
    o Unless you use _Offload_to or #pragma offload target(mic:cardnumber)
  – The compiler adds code to transfer data automatically to the coprocessor and to start your code running (with no extra coding on your part)
  – Hence the term “Heterogeneous Compiler” or “Offload Compiler”

• You can make further optimizations to your code that ensure full use of both the host and coprocessor
Heterogeneous Compiler – Data Transfer Overview

- **The host CPU and the Intel® MIC Architecture coprocessor do not share physical or virtual memory in hardware**

- Two offload data transfer models are available:
  1. **Explicit Copy**
     - Programmer designates variables that need to be copied between host and card in the offload directive
     - Syntax: Pragma/directive-based
     - C/C++ Example: #pragma offload target(mic) in(data:length(size))
     - Fortran Example: !dir$ offload target(mic) in(a1:length(size))
  2. **Implicit Copy**
     - Programmer marks variables that need to be shared between host and card
     - The same variable can then be used in both host and coprocessor code
     - Runtime automatically maintains coherence at the beginning and end of offload statements
     - Syntax: keyword extensions based
     - Example: _Shared double foo; _Offload func(y);
Heterogeneous Compiler – Overview

Summary

• Programmer designates code sections to offload
  – No further programming/API usage is needed
  – The compiler and the runtime *automatically* manage setup/teardown, data transfer, and synchronization

• Code marked for offload *is not guaranteed to run* on the coprocessor
  – If the coprocessor is unavailable, the offload section runs entirely on host
    o Unless you use `Offload_to` or `#pragma offload target(mic:cardnumber)`, in which case the coprocessor must be present for the code to run
    o Future feature: The runtime may use coprocessor load to decide whether or not to execute code marked for “offload” on the coprocessor

• Setting up the compiler build environment

  **CSH:** source /opt/intel/composerxe_mic/bin/compilervars.csh intel64
  **SH:** source /opt/intel/composerxe_mic/bin/compilervars.sh intel64
Offload Compilation

- High-Level overview of the Heterogeneous Compiler
  - Overview
  - Offload using Explicit Copies
  - Offload using Implicit Copies
  - Comparison of techniques
  - Heterogeneous Compiler command-line options
  - Intel® MIC Architecture Native Compiler
# Heterogeneous Compiler – Offload using Explicit Copies

<table>
<thead>
<tr>
<th></th>
<th><strong>C/C++ Syntax</strong></th>
<th><strong>Semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Offload pragma</td>
<td><code>#pragma offload &lt;clauses&gt; &lt;statement&gt;</code></td>
<td>Allow next statement to execute on Intel® MIC Architecture or host CPU</td>
</tr>
<tr>
<td>Keyword for</td>
<td><code>__attribute__((target(mic)))</code></td>
<td>Compile function for, or allocate variable on, both CPU and Intel® MIC</td>
</tr>
<tr>
<td>variable &amp;</td>
<td></td>
<td>Architecture</td>
</tr>
<tr>
<td>function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>definitions</td>
<td><code>#pragma offload_attribute(push, target(mic))</code></td>
<td>Mark entire files or large blocks of code for generation on both host CPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Intel® MIC Architecture</td>
</tr>
<tr>
<td>Entire blocks of</td>
<td><code>#pragma offload_attribute(pop)</code></td>
<td></td>
</tr>
<tr>
<td>code</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Fortran Syntax

<table>
<thead>
<tr>
<th></th>
<th><strong>Semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Offload directive</td>
<td>Execute next OpenMP* parallel construct on Intel® MIC Architecture</td>
</tr>
<tr>
<td></td>
<td>Execute next statement (function call) on Intel® MIC Architecture</td>
</tr>
<tr>
<td>Keyword for</td>
<td>Compile function or variable for CPU and Intel® MIC Architecture</td>
</tr>
<tr>
<td>variable/function</td>
<td></td>
</tr>
<tr>
<td>definitions</td>
<td></td>
</tr>
</tbody>
</table>
Heterogeneous Compiler – Conceptual Transformation

Source Code

```c
main()
{
  f();
}
```

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

```c
attribute__((target(mic))) g()
{
}
```

Linux* Host Program

```c
main()
{
  copy_code_to_mic();
  f();
  unload_mic();
}
```

```c
f()
{
  if (mic_available()){
    send_data_to_mic();
    start f_part_mic();
    receive_data_from_mic();
  } else
    f_part_host();
}
```

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

```c
g() {
...}
```

Intel ®MIC Program

```c
f_part_mic()
{
  a = b + g_mic();
}
```

```c
g_mic() {
...}
```

This all happens automatically when you issue a single compile command.
Heterogeneous Compiler – Offload using Explicit Copies – OpenMP* & Intel® Cilk™ Plus examples

**C/C++ OpenMP**

```c
#pragma offload target(mic)
#pragma omp parallel for
for (i=0; i<count; i++)
{
    a[i] = b[i] * c + d;
}
```

**Fortran OpenMP**

```fortran
!dir$ omp offload target(mic)
!$omp parallel do
    do i=1, count
        A(i) = B(i) * c + d
    end do
!$omp end parallel
```

**C/C++ Intel® Cilk™ Plus**

Note: There is no equivalent `_Cilk_for` notation in the explicit offload model.*

*There is one in the implicit model.
## Heterogeneous Compiler – Offload using Explicit Copies – Modifiers

Variables and pointers restricted to scalars, structs of scalars, and arrays of scalars

<table>
<thead>
<tr>
<th>Clauses / Modifiers</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target specification</td>
<td>target( name[:card_number] )</td>
<td>Where to run construct</td>
</tr>
<tr>
<td>Conditional offload</td>
<td>if (condition)</td>
<td>Boolean expression</td>
</tr>
<tr>
<td>Inputs</td>
<td>in(var-list modifiers&lt;sub&gt;opt&lt;/sub&gt;)</td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>Outputs</td>
<td>out(var-list modifiers&lt;sub&gt;opt&lt;/sub&gt;)</td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>Inputs &amp; outputs</td>
<td>inout(var-list modifiers&lt;sub&gt;opt&lt;/sub&gt;)</td>
<td>Copy host to coprocessor and back when offload completes</td>
</tr>
<tr>
<td>Non-copied data</td>
<td>nocopy(var-list modifiers&lt;sub&gt;opt&lt;/sub&gt;)</td>
<td>Data is local to target</td>
</tr>
</tbody>
</table>

### Modifiers

<table>
<thead>
<tr>
<th>Specify pointer length</th>
<th>length(element-count-expr)</th>
<th>Copy N elements of the pointer’s type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control pointer memory allocation</td>
<td>alloc_if ( condition )</td>
<td>Allocate memory to hold data referenced by pointer if condition is TRUE</td>
</tr>
<tr>
<td>Control freeing of pointer memory</td>
<td>free_if ( condition )</td>
<td>Free memory used by pointer if condition is TRUE</td>
</tr>
<tr>
<td>Control target data alignment</td>
<td>align ( expression )</td>
<td>Specify minimum memory alignment on target</td>
</tr>
</tbody>
</table>
float reduction(float *data, int numberOff)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(numberOff))
    {
        #pragma omp parallel for reduction(+:ret)
        for (int i=0; i < numberOff; ++i)
            ret += data[i];
    }
    return ret;
}
Heterogeneous Compiler – Offload using Explicit Copies - Rules & Limitations

• The Host↔Coprocessor data types allowed in a simple offload:
  - Scalar variables of all types
    - Must be *globals or statics* if you wish to use them with nocopy, alloc_if, or free_if (i.e. if they are to persist on the coprocessor between offload calls)
  - Structs that are bit-wise copyable (no pointer data members)
  - Arrays of the above types
  - Pointers to the above types

• What is allowed *within* coprocessor code?
  - All data types can be used (incl. full C++ objects)
  - Any parallel programming technique (Pthreads*, Intel® TBB, OpenMP*, etc.)
  - Intel® MIC Architecture versions of Intel® IPP and Intel® MKL
Heterogeneous Compiler – Offload using Explicit Copies – Data Movement

- Default treatment of \texttt{in/out} variables in a \texttt{#pragma offload} statement
  - At the start of an offload:
    - Space is allocated on the coprocessor
    - \texttt{in} variables are transferred to the coprocessor
  - At the end of an offload:
    - \texttt{out} variables are transferred from the coprocessor
    - Space for both types (as well as \texttt{inout}) is \texttt{deallocated} on the coprocessor

```c
#pragma offload inout(pA:length(n))
{
  ...\}
```

- Optimization Notice
  - Intel Confidential – NDA Presentation
Offload Compilation

- High-Level overview of the Heterogeneous Compiler
  - Overview
  - Offload using Explicit Copies
  - Offload using Implicit Copies
  - Comparison of techniques
  - Heterogeneous Compiler command-line options
  - Intel® MIC Architecture Native Compiler
Heterogeneous Compiler – Offload using Implicit Copies (1)

- Section of memory maintained at the same virtual address on both the host and Intel® MIC Architecture coprocessor
- Reserving same address range on both devices allows
  - Seamless sharing of complex pointer-containing data structures
  - Elimination of user marshaling and data management
  - Use of simple language extensions to C/C++
Heterogeneous Compiler – Offload using Implicit Copies (2)

- When “shared” memory is synchronized
  - Automatically done around offloads (so memory is only synchronized on entry to, or exit from, an offload call)
  - Only modified data is transferred between CPU and coprocessor
- Dynamic memory you wish to share must be allocated with special functions: `__Offload_shared_malloc`, `__Offload_shared_aligned_malloc`, `__Offload_shared_free`, `__Offload_shared_aligned_free`
- Allows transfer of C++ objects
  - Pointers are no longer an issue when they point to “shared” data
- 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
### Heterogeneous Compiler – Implicit: Keyword `_Shared` for Data and Functions

<table>
<thead>
<tr>
<th>What</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td><code>int _Shared f(int x) { return x+1; }</code></td>
<td>Versions generated for both CPU and card; may be called from either side</td>
</tr>
<tr>
<td>Global</td>
<td><code>_Shared int x = 0;</code></td>
<td>Visible on both sides</td>
</tr>
<tr>
<td>File/Function static</td>
<td><code>static _Shared int x;</code></td>
<td>Visible on both sides, only to code within the file/function</td>
</tr>
<tr>
<td>Class</td>
<td><code>class _Shared x {...};</code></td>
<td>Class methods, members, and and operators are available on both sides</td>
</tr>
<tr>
<td>Pointer to shared data</td>
<td><code>int _Shared *p;</code></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 *_Shared p;</code></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, _Shared)</code></td>
<td>Mark entire files or large blocks of code <code>_Shared</code> using this pragma</td>
</tr>
</tbody>
</table>
## Heterogeneous Compiler – Implicit: Offloading using `_Offload`

<table>
<thead>
<tr>
<th>Feature</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offloading a function call</td>
<td><code>x = _Offload func(y);</code></td>
<td><code>func</code> executes on coprocessor if possible</td>
</tr>
<tr>
<td></td>
<td><code>x = _Offload_to (card_number) func(y);</code></td>
<td><code>func must</code> execute on specified coprocessor</td>
</tr>
<tr>
<td>Offloading asynchronously</td>
<td><code>x = _Cilk_spawn _Offload func(y);</code></td>
<td>Non-blocking offload</td>
</tr>
<tr>
<td>Offload a parallel for-loop</td>
<td><code>_Offload _Cilk_for(i=0; i&lt;N; i++) { a[i] = b[i] + c[i]; }</code></td>
<td>Loop executes in parallel on target. The loop is implicitly outlined as a function call.</td>
</tr>
</tbody>
</table>
Heterogeneous Compiler – Implicit: Offloading using `_Offload` Example

```c
void findpi()
{
    int count = 10000;

    // Initialize shared global
    // variables
    _Shared float pi = 0.0f;

    // Compute pi on target
    _Offload compute_pi(count);

    pi /= count;
}

// Shared variable declaration for pi
_Shared float pi;

// Shared function declaration for
// compute
_Shared void compute_pi(int count)
{
    int i;

    //pragma omp parallel for 
    reduction(+:pi)
    for (i=0; i<count; i++)
    {
        float t = (float)((i+0.5f)/count);
        pi += 4.0f/(1.0f+t*t);
    }
}
```
Offload Compilation

- High-Level overview of the Heterogeneous Compiler
  - Overview
  - Offload using Explicit Copies
  - Offload using Implicit Copies
  - Comparison of techniques
  - Heterogeneous Compiler command-line options
  - Intel® MIC Architecture Native Compiler
## Heterogeneous Compiler – Comparison of Techniques (1)

<table>
<thead>
<tr>
<th></th>
<th>Offload via Explicit Data Copying</th>
<th>Offload via Implicit Data Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language Support</strong></td>
<td>Fortran, C, C++ <em>(C++ functions may be called, but C++ classes cannot be transferred)</em></td>
<td>C, C++</td>
</tr>
</tbody>
</table>
| **Syntax**                     | Pragmas/Directives:  
  • `#pragma offload in C/C++`  
  • `!dir$ omp offload directive in Fortran`                                                      | Keywords:  
  `_Shared` and `_Offload`                          |
| **Used for...**                | Offloads that transfer contiguous blocks of data                                                | Offloads that transfer all or parts of complex data structures, or many small pieces of data |
## Heterogeneous Compiler – Comparison of Techniques (2)

<table>
<thead>
<tr>
<th>Offload via Explicit Data Copying</th>
<th>Offload via Implicit Data Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offloaded data allowed</td>
<td>Scalars, arrays, bit-wise copyable structures</td>
</tr>
<tr>
<td>When data movement occurs</td>
<td>User has explicit control of data movement at start of each offload directive</td>
</tr>
<tr>
<td>When offload code is copied to card</td>
<td>At first pragma offload</td>
</tr>
</tbody>
</table>
Fortran Support is Different

- Main method for parallelism is OpenMP*
  - Use of library calls such as Intel® MKL
  - Eventually, Intel® MPI
  - (Be careful to set stack size parameters)
- Offload with implicit copying is not supported
  - no _Shared or _Offload keywords
  - no simple way to offload non-sequence derived types
- Use explicit copy with offload directives
  - !DIR$ OFFLOAD
  - Cannot use length parameter to offload part of an array; create a Fortran pointer and use that
    E.g. !DIR$ OFFLOAD IN(FPTR:length(n):free_if(.false.))
- Otherwise, much the same as C
Heterogeneous Compiler – Reminder of What is Generated

Note that for both techniques, the compiler generates two binaries:

- The host version
  - includes **all functions/variables** in the source code, whether marked `#pragma offload`, `__attribute__((target(mic)))`, `_Shared`, `_Offload`, or not

- The coprocessor version
  - includes **only functions/variables** marked `#pragma offload`, `__attribute__((target(mic)))`, `_Offload`, or `_Shared` in the source code
Offload Compilation

- High-Level overview of the Heterogeneous Compiler
  - Overview
  - Offload using Explicit Copies
  - Offload using Implicit Copies
  - Comparison of techniques
- Heterogeneous Compiler command-line options
- Intel® MIC Architecture Native Compiler
Heterogeneous Compiler – Command-line options

Offload-specific arguments to the Intel® Compiler:

• Generate host+coprocessor code (by default only host code is generated):
  - offload-build

• Produce a report of offload data transfers at compile time (not runtime)
  - opt-report-phase:offload

• Add Intel® MIC Architecture compiler switches
  - offload-copts: “switches”

• Add Intel® MIC Architecture archiver switches
  - offload-aropts: “switches”

• Add Intel® MIC Architecture linker switches
  - offload-ldopts: “switches”

Example:

```bash
icc -g -O2 -mkl -offload-build -offload-copts:"-g -03"
  -offload-ldopts:"-L/opt/intel/composerxe_mic/mkl/lib/mic"
foo.c
```
Heterogeneous Compiler – Command-line option – things to know

• "-openmp" is automatically set when you build with "-offload-build"
• Most command line arguments set for the host are set for the coprocessor build
  – Unless overridden by -offload-copts="..."
Offload Compilation

- High-Level overview of the Heterogeneous Compiler
  - Overview
  - Offload using Explicit Copies
  - Offload using Implicit Copies
  - Comparison of techniques
  - Heterogeneous Compiler command-line options
  - Intel® MIC Architecture Native Compiler
Intel® MIC Architecture Native Compilation

• Purpose:
  – Build standalone programs for execution directly on coprocessor
  – Create coprocessor-specific libraries for use by offloaded code sections

• Use:
  – Invoke the compiler with `-mmic` rather than `-offload-build` to generate purely “native” code

• Caveats:
  – Standalone programs and their data need to be copied manually by the user using `micfileio` or copied using an offload proxy
  – Shared libraries, such as `libiomp5.so` (which has no static counterpart) may need to be copied manually, even if you link your program statically.

• Performance analysis procedure is unchanged from offload code

• Debugging requires that you manually attach to the program while it runs on the coprocessor
$ icc -mmic -openmp omp_app.cpp
$ micfileio upload a.out /tmp/a.out
Uploading a.out to /tmp/a.out completed.
$ micfileio upload
   /opt/intel/composerxe_mic/compiler/lib/mic/libiomp5.so
   /tmp/libiomp5.so
Uploading /opt/intel/composerxe_mic/compiler/lib/mic/libiomp5.so to /tmp/libiomp5.so completed.

As super user, start minicom, connect to the card, then the run program

# cd /tmp
# ls
a.out  libiomp5.so
# export LD_LIBRARY_PATH=/tmp
# ./a.out Testarg
A big OpenMP hello to Testarg from 90 threads!
Module Summary

Topics we covered:
- High-Level overview of the Heterogeneous Compiler
  - Overview
  - Offload using Explicit Copies
  - Offload using Implicit Copies
  - Comparison of techniques
  - Heterogeneous Compiler command-line options
  - Intel® MIC Architecture Native Compiler
Module Discussion

Can you

- List two pragmas or keywords that will cause code to run on the Intel® MIC architecture?
- Explain how data movement is controlled between the host and card?
Suggested Labs and Samples

Labs
- Converting Code for Offload (Lesson 1)
- Monitoring the Coprocessor (Lesson 2)
- Building and Running a “Native” Intel® MIC Architecture Application (Lesson 3)

Additional information may be found at [https://mic-dev.intel.com](https://mic-dev.intel.com)

Compiler reference guide at:
/opt/intel/composerxe_mic/Documentation/en_US/mic

Example code showing various offload subtleties can be found at
- /opt/intel/composerxe_mic/Samples/en_US/C++/mic_samples
- /opt/intel/composerxe_mic/Samples/en_US/Fortran/mic_samples
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101
Backup