Intel® Xeon Phi™ Coprocessor
Offloading Computation
Offloading Computation Topics

• Programming Model Overview
• Offload Using Explicit Copies
• Offload Using Implicit Copies
• Comparison of Techniques
• Heterogeneous Compiler Command-line Options
• Intel® MIC architecture Native Compiling
• Which should I use?
Module Outline

- Programming Model Overview
  - Offload Using Explicit Copies
  - Offload Using Implicit Copies
  - Comparison of Techniques
  - Heterogeneous Compiler Command-line Options
  - Intel® MIC Architecture Native Compiling
  - Which should I use?
Heterogeneous Compiler – Programming Model Overview

- Add pragmas and new keywords to working host code to make sections of code run on the Intel® Xeon Phi™ 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
    - Unless you use `_Cilk_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 further optimize your code to ensure full use of the host and coprocessor
Heterogeneous Compiler – Data Transfer Overview

- **The host CPU and the Intel® Xeon Phi™ 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 pragma/directive*
     - Syntax: Pragma/directive-based
     - C/C++ Example:  
       ```c
       #pragma offload target(mic) in(data:length(size))
       ```
     - Fortran Example:  
       ```fortran
       !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:  
       ```c
       _Cilk_shared double foo; _Cilk_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 `_Cilk_offload_to` or “#pragma offload target(mic:cardnumber)”, then 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

  SH: source /opt/intel/composerxe/bin/compilervars.sh intel64
Module Outline

- Programming Model Overview
- **Offload Using Explicit Copies**
- Offload Using Implicit Copies
- Comparison of Techniques
- Heterogeneous Compiler Command-line Options
- Intel® MIC Architecture Native Compiling
- Which should I use?
### 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>#pragma offload &lt;clauses&gt; &lt;statement&gt;</td>
<td>Allow next statement to execute on Intel® Xeon Phi™ coprocessor or host CPU</td>
</tr>
<tr>
<td>Keyword for variable &amp;</td>
<td><strong>attribute</strong>((target(mic)))</td>
<td>Compile function for, or allocate variable on, both CPU and Intel® Xeon Phi™</td>
</tr>
<tr>
<td>function definitions</td>
<td></td>
<td>coprocessor</td>
</tr>
<tr>
<td>Entire blocks of code</td>
<td>#pragma offload_attribute(push, target(mic))</td>
<td>Mark entire files or large blocks of code for generation on both host CPU and</td>
</tr>
<tr>
<td></td>
<td>:: #pragma offload_attribute(pop)</td>
<td>Intel® Xeon Phi™ coprocessor</td>
</tr>
</tbody>
</table>

#### Fortran Syntax

<table>
<thead>
<tr>
<th></th>
<th><strong>Fortran Syntax</strong></th>
<th><strong>Semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Offload directive</td>
<td>!dir$ omp offload &lt;clause&gt; &lt;statement&gt;</td>
<td>Execute next OpenMP* parallel construct on Intel® MIC Architecture</td>
</tr>
<tr>
<td></td>
<td>!dir$ offload &lt;clauses&gt; &lt;statement&gt;</td>
<td>Execute next statement (function call) on Intel® Xeon Phi™ coprocessor</td>
</tr>
<tr>
<td>Keyword for variable/function definitions</td>
<td>!dir$ attributes offload:&lt;mic&gt; :: &lt;return-name&gt; OR &lt;var1,var2,...&gt;</td>
<td>Compile function or variable for CPU and Intel® Xeon Phi™ coprocessor</td>
</tr>
</tbody>
</table>
Heterogeneous Compiler – Conceptual Transformation

Source Code

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

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

_attribute__((target(mic))) g()
{
    ...
}
```

Linux* Host Program

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

```
int f()
{
    ...
    if (mic_available()){
        send_data_to_mic();
        start offl_part_mic();
        receive_data_from_mic();
    } else
    offl_part_host();
}
```

Intel *MIC Program

```
offl_part_host()
{
a = b + g();
}
```

```
offl_part_mic()
{
a = b + g_mic();
}
```

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

This all happens automatically when you issue a single compile command.
Put in illustration of what code is generated in terms of host and coprocessor!

Code compiled for host

offload (host version)

offload (host version)

Code compiled for host

offload (coprocessor version)

offload (coprocessor version)
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 *direct* 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><code>target( name[:card_number] )</code></td>
<td>Where to run construct</td>
</tr>
<tr>
<td>Conditional offload</td>
<td><code>if (condition)</code></td>
<td>Boolean expression</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>

## Modifiers

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify pointer length</td>
<td><code>length(element-count-expr)</code></td>
<td>Copy N elements of the pointer’s type</td>
</tr>
<tr>
<td>Control pointer memory allocation</td>
<td><code>alloc_if ( condition )</code></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><code>free_if ( condition )</code></td>
<td>Free memory used by pointer if condition is TRUE</td>
</tr>
<tr>
<td>Control target data alignment</td>
<td><code>align ( expression )</code></td>
<td>Specify minimum memory alignment on target</td>
</tr>
</tbody>
</table>
Heterogeneous Compiler – Offload using Explicit Copies – Modifier Example

```c
float reduction(float *data, int numberOf)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(numberOf))
    {
        #pragma omp parallel for reduction(+:ret)
        for (int i=0; i < numberOf; ++i)
            ret += data[i];
    }
    return ret;
}
```

Note: copies numberOf*sizeof(float) elements to the coprocessor, not numberOf bytes – the compiler knows data’s type
Heterogeneous Compiler – Offload using Explicit Copies - Rules & Limitations

• The Host⇔Coprocessor data types allowed in a simple offload:
  – Scalar variables of all types
    o 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® Thread Building Blocks, OpenMP*, etc.)
  – Intel® MIC Architecture versions of Intel® Integrated Performance Primatives and Intel® Math Kernel Library
Heterogeneous Compiler – Offload using Explicit Copies – Data Movement

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

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

- Programming Model Overview
- Offload Using Explicit Copies
- **Offload Using Implicit Copies**
- Comparison of Techniques
- Heterogeneous Compiler Command-line Options
- Intel® MIC Architecture Native Compiling
- Which should I use?
Heterogeneous Compiler – Offload using Implicit Copies (1)

- Section of memory maintained at the same virtual address on both the host and Intel® Xeon Phi™ 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 _Cilk_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>int _Cilk_shared f(int x) { return x+1; }</td>
<td>Versions generated for both CPU and card; may be called from either side</td>
</tr>
<tr>
<td>Global</td>
<td>_Cilk_shared int x = 0;</td>
<td>Visible on both sides</td>
</tr>
<tr>
<td>File/Function</td>
<td>static _Cilk_shared int x;</td>
<td>Visible on both sides, only to code within the file/function</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>class _Cilk_shared x {...};</td>
<td>Class methods, members, and operators are available on both sides</td>
</tr>
<tr>
<td>Pointer to shared data</td>
<td>int _Cilk_shared *p;</td>
<td>p is local (not shared), can point to shared data</td>
</tr>
<tr>
<td>A shared pointer</td>
<td>int *_Cilk_shared p;</td>
<td>p is shared; should only point at shared data</td>
</tr>
</tbody>
</table>
| Entire blocks of code| #pragma offload_attribute(push, _Cilk_shared)
#pragma offload_attribute(pop) | Mark entire files or large blocks of code _Cilk_shared using this pragma |
## Heterogeneous Compiler – Implicit: Offloading using _Cilk_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 = _Cilk_offload func(y);</code></td>
<td><code>func</code> executes on coprocessor if possible</td>
</tr>
<tr>
<td></td>
<td><code>x = _Cilk_offload_to (card_num) func(y);</code></td>
<td><code>func</code> <strong>must</strong> execute on specified coprocessor</td>
</tr>
<tr>
<td>Offload asynchronously</td>
<td><code>x = _Cilk_spawn _Cilk_offload func(y);</code></td>
<td>Non-blocking offload</td>
</tr>
<tr>
<td>Offload a parallel for-loop</td>
<td><code>_Cilk_offload _Cilk_for(i=0; i&lt;N; i++)</code></td>
<td>Loop executes in parallel on target. The loop is implicitly outlined as a function call.</td>
</tr>
<tr>
<td></td>
<td><code>{  a[i] = b[i] + c[i];  }</code></td>
<td></td>
</tr>
</tbody>
</table>
void findpi()
{
    int count = 10000;

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

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

    pi /= count;
}

// Shared variable declaration for pi
_Cilk_shared float pi;

// Shared function declaration for
// compute
_Cilk_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);
    }
}
Module Outline

• Programming Model Overview
• Offload Using Explicit Copies
• Offload Using Implicit Copies
• Comparison of Techniques
• Heterogeneous Compiler Command-line Options
• Intel® MIC Architecture Native Compiling
• Which should I use?
## 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>
<tr>
<td><strong>Syntax</strong></td>
<td>Pragmas/Directives: • #pragma offload in C/C++ • !dir$ omp offload directive in Fortran</td>
<td>Keywords: _Cilk_shared and _Cilk_offload</td>
</tr>
<tr>
<td><strong>Used for...</strong></td>
<td>Offloads that transfer contiguous blocks of data</td>
<td>Offloads that transfer all or parts of complex data structures, or many small pieces of data</td>
</tr>
</tbody>
</table>
### Heterogeneous Compiler – Comparison of Techniques (2)

<table>
<thead>
<tr>
<th></th>
<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>
<td>All data types (pointer-based data, structures, classes, locks, etc.)</td>
</tr>
<tr>
<td>When data movement occurs</td>
<td>User has explicit control of data movement at start of each offload directive</td>
<td>All <code>_Cilk_shared</code> data synchronized at start and end of <code>_Cilk_offload</code> statements</td>
</tr>
<tr>
<td>When offload code is copied to card</td>
<td>At first <code>#pragma offload</code></td>
<td>At program startup</td>
</tr>
</tbody>
</table>
Fortran Support is Different

• Main method for parallelism is OpenMP*
  – Use of library calls such as Intel® Math Kernel Library
  – Intel® MPI Library
  – (Be careful to set stack size parameters)

• Offload with implicit copying is not supported
  – no _Cilk_shared or _Cilk_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, similar to 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))), __Cilk_shared, __Cilk_offload, or not

- The coprocessor version
  - includes only functions/variables marked #pragma offload, __attribute__((target(mic))), __Cilk_offload, or _Cilk_shared in the source code
Module Outline

- Programming Model Overview
- Offload Using Explicit Copies
- Offload Using Implicit Copies
- Comparison of Techniques

**Heterogeneous Compiler Command-line Options**

- Intel® MIC Architecture Native Compiling
- Which should I use?
Heterogeneous Compiler – Command-line options

Offload specific arguments to the Intel® Compiler:

• Generate only host code (by default host+coprocessor code is generated):
  - no-offload
• Produce a report of offload data transfers at compile time (not runtime)
  - opt-report-phase:offload
• To explicitly specify options for a particular tool at compile time
  - offload-option,target,tool,”option-list”
• Add Intel® MIC Architecture compiler switches
  - offload-option,mic,compiler,“switches”
• Add Intel MIC Architecture assembler switches
  - offload-option,mic,as,“switches”

Example:

```sh
icc -I/my_dir/include -DMY_DEFINE=10
  -offload-option,mic,compiler="-I/my_dir/mic/include -DMY_DEFINE=20"
hello.c
```

Passes “-I/my_dir/mic/include -DMY_DEFINE=10
  -DMY_DEFINE=20” to the offload compiler
Module Outline

- Programming Model Overview
- Offload Using Explicit Copies
- Offload Using Implicit Copies
- Comparison of Techniques
- Heterogeneous Compiler Command-line Options
- **Intel® MIC Architecture Native Compiling**
- Which should I use?
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 to generate purely “native” code

• Caveats:
  – Standalone programs and data must be copied manually using scp
  – 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, or use through terminal emulator
Intel® MIC Architecture Native Compilation Communication

• Communication
  – Host <==> Coprocessor supported
  – Coprocessor <==> Coprocessor supported
  – External <==> Coprocessor supported

• Addressing

<table>
<thead>
<tr>
<th>Card</th>
<th>Network Interface</th>
<th>MIC IP Address</th>
<th>Host IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mic0</td>
<td>172.31.1.1</td>
<td>172.31.1.254</td>
</tr>
<tr>
<td>1</td>
<td>mic1</td>
<td>172.31.2.1</td>
<td>172.31.2.254</td>
</tr>
</tbody>
</table>

• Example

ssh -i /opt/intel/mic/id_rsa root@172.31.1.1
Intel® MIC Architecture Native Compilation

NFS Mounting

• Host
  – Install/execute NFS Server
  – Add dir to be exported to /etc/exports
  – Add mic to /etc/hosts.allow

• Coprocessor
  – Create mount point
  – Add dir to /etc/fstab

HOST /etc/exports:
/mic/mic0fs 172.31.1.1/24(rw,no_root_squash)

HOST /etc/hosts.allow:
ALL: 172.31.1.1

MIC card /etc/fstab:
172.31.1.254:/mic/mic0fs /mic0fx nfs rsize=8192,wsize=8192,nolock,intr 0 0
Intel® MIC Architecture Native Compilation - Example

$ icc -mmic -openmp omp_app.cpp
$ su
# scp -i /opt/intel/mic/id_rsa a.out root@172.31.1.1:/tmp/
a.out 100% 111KB 111.5KB/s 00:00
# scp -i /opt/intel/mic/id_rsa /opt/intel/composerxe/lib/mic/
  libiomp5.so root@172.31.1.1:/tmp/
libiomp5.so 100% 939KB 939.2KB/s 00:01
#

Connect to the card, then the run program

# ssh -i /opt/intel/mic/id_rsa root@172.31.1.1
~ # cd /tmp
/tmp # ls
a.out   libiomp5.so
/tmp # export LD_LIBRARY_PATH=/tmp
/tmp # ./a.out Testarg
A big OpenMP hello to Testarg from 90 threads!
Module Outline

• Programming Model Overview
• Offload Using Explicit Copies
• Offload Using Implicit Copies
• Comparison of Techniques
• Heterogeneous Compiler Command-line Options
• Intel® MIC Architecture Native Compiling

• Which should I use?
When to use Native vs. offload

• Native only
  + Easy to port – just change some switches, add libs, use micnativeloadex; focus on parallelization once ported
  - Scalar and IO-intensive parts run slowly
  - Memory limited to 8 GB

• Host with offload
  + Better perf when serial/scalar fraction is significant
  + Enables memory footprint >> working set, e.g. via pipelining
  + Path to hybrid, since no reverse offload
  - Higher porting cost
  - Needs identifiable hotspots

• “Hybrid” host+device
  + Best perf for parallel cases
  - More effort to manage concurrency with async or multithreading
Offloading Computation Summary

- Programming Model Overview
- Offload Using Explicit Copies
- Offload Using Implicit Copies
- Comparison of Techniques
- Heterogeneous Compiler Command-line Options
- Intel® MIC Architecture Native Compiling
- Which should I use?
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.