Multi and Many-core Parallel Execution Models

PRACE MIC Summer School, 8-11 July 2013, CINECA
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 PROPERTY 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, TRADEMARK, OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSIONritical APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS, LIABILITIES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSIONITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENCE IN THE DESIGN, MANUFACTURE, OR DISTRIBUTION OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

- 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, Phi and Cilk, Intel and Intel Sponsors of Tomorrow. and Intel Sponsors of Tomonow. 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 ©2012 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. http://www.intel.com/info/hyperthreading

- Intel® 64 architecture: Requires a system with a 64-bit enabled processor, chipset, BIOS and software. http://www.intel.com/info/em64t


Performance Notice

Multi and Many-Core Programming Models

2 key questions:

• How to address the coprocessor from an application?
  – Focus of today’s talk

• What are the options/specificities to express thread parallelism?
  – Tomorrow’s presentation
Intel® Xeon Phi™ Coprocessor Becomes a Network Node

Intel® Xeon® Processor

Intel® Xeon Phi™ Coprocessor

Virtual Network Connection

Intel® Xeon® Processor

Intel® Xeon Phi™ Coprocessor

Virtual Network Connection

Intel® Xeon Phi™ Architecture + Linux enables IP addressability
Spectrum of Programming Models and Mindsets

Multi-Core Centric

Multi-Core Hosted

General purpose serial and parallel computing

Symmetric

Codes with balanced needs

Many-Core Hosted

Highly-parallel codes

Range of models to meet application needs
Coprocessor only Programming Model

- MPI ranks on Intel® Xeon Phi™ coprocessor (only)
- All messages into/out of the coprocessors
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads used directly within MPI processes

Build Intel Xeon Phi coprocessor binary using the Intel® compiler

Upload the binary to the Intel Xeon Phi coprocessor

Run instances of the MPI application on Intel Xeon Phi coprocessor nodes
**Coprocessor only Programming Model**

- MPI ranks on the Intel® Xeon Phi™ coprocessor(s) only
- MPI messages into/out of the coprocessor(s)
- Threading possible

Build the application for the Intel® Xeon Phi™ Architecture

```bash
# mpiicc -mmic -o test_hello.MIC test.c
```

Upload the coprocessor executable

```bash
# scp ./test_hello.MIC node0-mic0:/tmp
   # Remark: If NFS available no explicit uploads required!
```

Launch the application on the coprocessor from host

```bash
# export I_MPI_MIC=enable
# cat mpi_hosts
node0-mic0
# mpirun -f mpi_hosts -n 2 /tmp/test_hello.MIC
```

Alternatively: login to the coprocessor and execute the already uploaded mpirun there!
Symmetric Programming Model

- MPI ranks on Intel® Xeon Phi™ Architecture and host CPUs
- Messages to/from any core
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* used directly within MPI processes

Build binaries by using the resp. compilers targeting Intel 64 and Intel Xeon Phi Architecture

Upload the binary to the Intel Xeon Phi coprocessor

Run instances of the MPI application on different mixed nodes
Symmetric Programming Model

- MPI ranks on the coprocessor(s) and host CPU(s)
- MPI messages into/out of the coprocessor(s) and host CPU(s)
- Threading possible

Build the application for Intel®64 and the Intel® Xeon Phi™ Architecture separately

```bash
# mpiicc -o test_hello test.c
# mpiicc -mmic -o test_hello.MIC test.c
```

Upload the Intel® Xeon Phi™ coprocessor executable

```bash
# scp ./test_hello.MIC node0-mic0:~/test_hello
```

- Remark: If NFS available no explicit uploads required (look for tips on next slide)!

Launch the application on the host and the coprocessor from the host

```bash
# export I_MPI_MIC=enable
# cat mpi_hosts
node0
node0-mic0
# mpirun -f mpi_hosts -n 2 ~/test_hello
```
**MPI+Offload Programming Model**

- MPI ranks on Intel® Xeon® processors (only)
- All messages into/out of host CPUs
- Offload models used to accelerate MPI ranks
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* within Intel® Xeon Phi™ coprocessor

Homogenous network of heterogeneous nodes

Build Intel® 64 executable with included offload by using the Intel compiler

Run instances of the MPI application on the host, offloading code onto coprocessor

Advantages of more cores and wider SIMD for certain applications
MPI+Offload Programming Model

- MPI ranks on the host CPUs only
- MPI messages into/out of the host CPUs
- Intel® Xeon Phi™ Architecture as an accelerator

Compile for MPI and internal offload

```bash
# mpiicc -o test_hello test.c
```

Latest compiler compiles by default for offloading if offload construct is detected!
- Switch off by `-no-offload` flag

Execute on host(s) as usual

```bash
# cat mpi_hosts
node0

# mpirun -f mpi_hosts -n 2 ~/test_hello
```

MPI processes will offload code for acceleration
Offload Compilation

• Overview
• Offload using Explicit Copies
• Offload using Implicit Copies
• Comparison of techniques
• Heterogeneous Compiler command-line options
• Simultaneous Host/Coprocessor Computing
• Asynchronous Offload and Data Transfer
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® Xeon Phi™ Coprocessor 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 _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 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 coprocessor do not share physical or virtual memory in hardware

• Two offload data transfer models are available:
  1. Explicit Copy
     o Programmer designates variables that need to be copied between host and card in the offload pragma/directive
     o Syntax: Pragma/directive-based
     o C/C++ Example: #pragma offload target(mic) in(data:length(size))
     o Fortran Example: !dir$ offload target(mic) in(a1:length(size))
  2. Implicit Copy
     o Programmer marks variables that need to be shared between host and card
     o The same variable can then be used in both host and coprocessor code
     o Runtime automatically maintains coherence at the beginning and end of offload statements
     o Syntax: keyword extensions based
     o Example: _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)”, 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/bin/compilervars.csh intel64
  SH: source /opt/intel/bin/compilervars.sh intel64
Offload Compilation

- Overview
- Offload using Explicit Copies
- Offload using Implicit Copies
- Comparison of techniques
- Heterogeneous Compiler command-line options
- Simultaneous Host/Coprocessor Computing
- Asynchronous Offload and Data Transfer
## Heterogeneous Compiler – C/C++ Offload using Explicit Copies

<table>
<thead>
<tr>
<th>C/C++ Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offload pragma</td>
<td>Allow next statement to execute on coprocessor or host CPU</td>
</tr>
<tr>
<td>#pragma offload &lt;clauses&gt; &lt;statement&gt;</td>
<td></td>
</tr>
<tr>
<td>Offload transfer</td>
<td>Initiates asynchronous data transfer, or initiates and completes synchronous data transfer</td>
</tr>
<tr>
<td>#pragma offload_transfer &lt;clauses&gt;</td>
<td></td>
</tr>
<tr>
<td>Offload wait</td>
<td>Specifies a wait for a previously initiated asynchronous activity</td>
</tr>
<tr>
<td>#pragma offload_wait &lt;clauses&gt;</td>
<td></td>
</tr>
<tr>
<td>Keyword for variable &amp; function definitions</td>
<td>Compile function for, or allocate variable on, both CPU and coprocessor</td>
</tr>
<tr>
<td><strong>attribute</strong>((target(mic)))</td>
<td></td>
</tr>
<tr>
<td>Entire blocks of code</td>
<td>Mark entire files or large blocks of code for generation on both host CPU and coprocessor</td>
</tr>
<tr>
<td>#pragma offload_attribute(push, target(mic))</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>#pragma offload_attribute(pop)</td>
<td></td>
</tr>
</tbody>
</table>
# Heterogeneous Compiler – Fortran Offload using Explicit Copies

<table>
<thead>
<tr>
<th><strong>Fortran Syntax</strong></th>
<th><strong>Semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Offload directive</td>
<td>![dir$ omp offload &lt;clauses&gt; &lt;statement&gt;] Execute next OpenMP* parallel construct on coprocessor</td>
</tr>
<tr>
<td></td>
<td>![dir$ offload &lt;clauses&gt; &lt;statement&gt;] Execute next statement (function call) on coprocessor</td>
</tr>
<tr>
<td>Offload transfer</td>
<td>![dir$ offload_transfer &lt;clause&gt;] Initiates asynchronous data transfer, or initiates and completes synchronous data transfer</td>
</tr>
<tr>
<td>Offload wait</td>
<td>![dir$ offload_wait &lt;clauses&gt;] Specifies a wait for a previously initiated asynchronous activity</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;] Compile function or variable for CPU and 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_coproc();
f();
unload_coproc();
}
```

Intel® Xeon Phi™ Program

```c
f()
{
    if (coproc_available()){
        send_data_to_coproc();
        start f_part_coproc();
        receive_data_from_coproc();
    } else 
    f_part_host();
}
```

This all happens automatically when you issue a single compile command

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

```c
f_part_coproc()
{a = b + g_coproc();}
```

```c
g_coproc() {...}
```

```c
g() {...}
```
Heterogeneous Compiler – Conceptual Transformation

Source Code

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

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

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

Linux* Host Program

```
main()
{
    copy_code_to_coproc();
    f();
    unload_coproc();
}

f()
{
    if (coproc_available()){
        send_data_to_coproc();
        start f_part_coproc();
        receive_data_from_coproc();
    } else
    f_part_host();
}

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

Intel® Xeon Phi™ Program

```
    f_part_coproc() {a = b + g_coproc();}
    g_coproc() {...}
```

This all happens automatically when you issue a single compile command.

Intel® Xeon Phi™ Coprocessor
Software & Services Group, Developer Products Division
Copyright© 2013, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.
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_coproc();
    f();
    unload_coproc();
}
```

```c
f()
{
    if (coproc_available()){
        send_data_to_coproc();
        start f_part_coproc();
        receive_data_from_coproc();
    } else
    f_part_host();
}
```

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

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

Intel* Xeon Phi™ Program

```c
f_part_coproc() {a = b + g_coproc();}
```

```c
g_coproc() {
    ...
}
```

This all happens automatically when you issue a single compile command.
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/coproc();
    f();
    unload_coproc();
}

f()
{
    if (coproc_available()){
        send_data_to_coproc();
        start f_part_coproc();
        receive_data_from_coproc();
    } else
    f_part_host();
}

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

g() {...}
```

**Intel® Xeon Phi™ Program**

```c
f_part_coproc() {a = b + g_coproc();}

g_coproc() {...}
```

This all happens automatically when you issue a single compile command.
Heterogeneous Compiler – Offload using Explicit Copies – OpenMP* examples

C/C++ OpenMP*

```c
#pragma omp 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)
!$omp do
   A(i) = B(i) * c + d
end do
!$omp end parallel
```
**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**

| Specify pointer length   | `length(element-count-expr)`                    | Copy N elements of the pointer’s type                                    |
| Control pointer memory allocation | `alloc_if ( condition )`                         | Allocate memory to hold data referenced by pointer if condition is TRUE |
| Control freeing of pointer memory | `free_if ( condition )`                          | Free memory used by pointer if condition is TRUE                         |
| Control target data alignment | `align ( expression )`                           | Specify minimum memory alignment on target                               |
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® TBB, OpenMP*, etc.)
  – Intel® Xeon Phi™ versions of Intel® MKL
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 – Data Movement

- Default treatment of \texttt{in/out variables} in a \#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

\begin{Verbatim}
#pragma offload inout(pA:length(n))
{}
\end{Verbatim}
Offload Compilation

- Overview
- Offload using Explicit Copies
- Offload using Implicit Copies
- Comparison of techniques
- Heterogeneous Compiler command-line options
- Simultaneous Host/Coprocessor Computing
- Asynchronous Offload and Data Transfer
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

**Syntax**: 

<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 static</td>
<td>static _Cilk_shared int x;</td>
<td>Visible on both sides, only to code within the file/function</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>
<tr>
<td>Entire blocks of code</td>
<td>#pragma offload_attribute(</td>
<td>Mark entire files or large blocks of code _Cilk_shared using this pragma</td>
</tr>
<tr>
<td></td>
<td>\push, _Cilk_shared)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#pragma offload_attribute(pop)</td>
<td></td>
</tr>
</tbody>
</table>
# 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>Offloading 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>`{ a[i] = b[i] + c[i]; }</td>
<td></td>
</tr>
</tbody>
</table>
Heterogeneous Compiler – Implicit: Offloading using _Cilk_offload Example

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);
    }
}
Offload Compilation

- Overview
- Offload using Explicit Copies
- Offload using Implicit Copies
- Comparison of techniques
- Heterogeneous Compiler command-line options
- Simultaneous Host/Coprocessor Computing
- Asynchronous Offload and Data Transfer
### Heterogeneous Compiler – Comparison of Techniques (1)

<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>Which data is transferred</td>
<td>User has explicit control of data movement at start of each offload directive</td>
</tr>
<tr>
<td>When data movement occurs</td>
<td>Data transfer can overlap with offload execution</td>
</tr>
<tr>
<td>When offload code is copied to card</td>
<td>At first <code>#pragma offload</code></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><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:  
  `_Cilk_shared` and `_Cilk_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       |
Fortran Support is Different

- Main method for parallelism is OpenMP*
  - Use of library calls such as Intel® MKL
  - Intel® MPI
  - (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, 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))), _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

Linking creates one executable with both binaries included!
Offload Compilation

• Overview
• Offload using Explicit Copies
• Offload using Implicit Copies
• Comparison of techniques
• Heterogeneous Compiler command-line options
• Simultaneous Host/Coprocessor Computing
• Asynchronous Offload and Data Transfer
Heterogeneous Compiler – Command-line options

Offload-specific arguments to the Intel® Compiler:

• Generate host only code (by default both host+coprocessor code is generated):
  - no-offload

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

• Add Intel® Xeon Phi™ compiler switches
  - offload-options,mic,compiler,“switches”

• Add Intel® Xeon Phi™ assembler switches
  - offload-options,mic,as:“switches”

• Add Intel® Xeon Phi™ linker switches
  - offload-options,mic,ld,“switches”

Example:

icc -I/my_dir/include -DMY_DEFINE=10 -offload-options,mic,compiler,“-I/my_dir/mic/include -DMY_DEFINE=20” hello.c

Passes “-I/my_dir/mic/include -I/my_dir/include -DMY_DEFINE=10 -DMY_DEFINE=20” to the offload compiler
Heterogeneous Compiler – Command-line option – things to know

• "-openmp" is automatically set when you build
• Don’t need -no-offload if compiling only for Xeon
  – Generates same Xeon only code as previous compilers
  – But -no-offload creates smaller binaries
• Most command line arguments set for the host are set for the coprocessor build
  – Unless overridden by -offload-option,mic,xx="...">
  – Add -watch=mic-cmd to display the compiler options automatically passed to the offload compilation
Offload Compilation

- Overview
- Offload using Explicit Copies
- Offload using Implicit Copies
- Comparison of techniques
- Heterogeneous Compiler command-line options
- **Simultaneous Host/Coprocessor Computing**
- Asynchronous Offload and Data Transfer
Simultaneous Host/Coprocessor Computing - Overview

• #pragma offload statements or _Cilk_offload calls block until the statement completes (may change in future)

• Simultaneous host and coprocessor computing requires multiple threads of execution on the host:
  – One or more to block until their #pragma offload statements or _Cilk_offload calls complete
  – Others to do simultaneous processing on the host

• You can use most multithreading APIs to do this
  – Pthreads*
  – Intel® TBB’s parallel_invoke
  – OpenMP* tasks or parallel sections
  – Intel® Cilk™ Plus
  – etc.
Simultaneous Host/Coprocessor Computing – Using OpenMP*

• Simply use OpenMP* task on host to spawn the offload call
  – Then use OpenMP* for parallelism on the coprocessor

• Use other OpenMP* task calls to simultaneously run code on the host

```c
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    #pragma offload target(mic) ...
    {
        <various serial code>
        #pragma omp parallel for
        for (int i=0; i<limit; i++)
            <parallel loop body>
    }
    #pragma omp task
    {<host code or another offload>}
}```
Simultaneous Host/Coprocessor Computing – Using Intel® Cilk™ Plus

• _Cilk_spawn statements in host code can be combined with statements marked with _Cilk_offload to allow asynchronous processing
  – Example:

```c
x = _Cilk_spawn _Cilk_offload func(y);

// Host code runs simultaneously on host while _Cilk_offload code runs on coprocessor

_Cilk_sync;  // Host here waits for all spawns to complete
```

• Note that the following is not asynchronous, but instead blocks in this host thread until the _Cilk_for completes on the coprocessor

```c
_Cilk_offload _Cilk_for (i=0;i<10000;i++)
{
    <loop_body;>
}
```
Offload Compilation

- Overview
- Offload using Explicit Copies
- Offload using Implicit Copies
- Comparison of techniques
- Heterogeneous Compiler command-line options
- Simultaneous Host/Coprocessor Computing
- Asynchronous Offload and Data Transfer
Asynchronous Offload and Data Transfer

• Only available for the explicit memory model
  - Not part of the Cilk standard
• signal() and wait()
• Available async functionality
  - offload
  - offload_transfer
  - offload_wait
• Examples
  #pragma offload target(mic:0) signal(flg1)
  #pragma offload_transfer target(mic:0) signal(flg2) \ 
  wait(flg1)
  #pragma offload_wait target(mic:0) wait(flg1)
Double Buffering Example

- Overlap computation and communication
- Generalizes to data domain decomposition

![Diagram showing double buffering example with Host and Target data blocks and xfer and process arrows between them for iterations 0, 1, n, n+1, and last iteration.]
Double Buffering Example

```c
int i;
#pragma offload_transfer target(mic:0) in(in1 : length(count) \ 
    alloc_if(0) free_if(0) ) signal(in1)
for (i=0; i<iter; i++)
{
    if (i%2 == 0) {
        #pragma offload_transfer target(mic:0) if(i!=iter-1) \ 
            in(in2 : length(count) alloc_if(0) free_if(0) ) signal(in2)
        #pragma offload target(mic:0) nocopy(in1) wait(in1) \ 
            out(out1 : length(count) alloc_if(0) free_if(0) )
        compute(in1, out1);
    } else {
        #pragma offload_transfer target(mic:0) if(i!=iter-1) \ 
            in(in1 : length(count) alloc_if(0) free_if(0) ) signal(in1)
        #pragma offload target(mic:0) nocopy(in2) wait(in2) \ 
            out(out2 : length(count) alloc_if(0) free_if(0) )
        compute(in2, out2);
    }
}
```
Double Buffering Example

```c
int i;
#pragma offload_transfer target(mic:0) in(in1 : length(count) \  
    alloc_if(0) free_if(0) ) signal(in1)
for (i=0; i<iter; i++)
{
    if (i%2 == 0) {
        #pragma offload_transfer target(mic:0) if(i!=iter-1) \  
        in(in2 : length(count) alloc_if(0) free_if(0) ) signal(in2)
        #pragma offload target(mic:0) nocopy(in1)  
        out(out1 : length(count) alloc_if(0) free_if(0) )  
        compute(in1, out1);
    } else {
        #pragma offload_transfer target(mic:0) if(i!=iter-1) \  
        in(in1 : length(count) alloc_if(0) free_if(0) ) signal(in1)
        #pragma offload target(mic:0) nocopy(in2)  
        out(out2 : length(count) alloc_if(0) free_if(0) )  
        compute(in2, out2);
    }
}
```
Signal, Wait and tag (1)

• Examples
  - #pragma offload_transfer target(mic:0) signal(tagA) \ wait(tag0, tag1) ...
    o Do not start transfer until the operations signaling tag0 and tag1 are complete
    o Upon completion, indicate completion using tagA
  - #pragma offload target(mic:0) signal(tagB) \ wait(tag2, tag3, tag4) ...
    o Do not start transfer until the operations signaling tag2, tag3 and tag4 are complete
    o Upon completion, indicate completion using tagB
Signal, Wait and tag (2)

Notes

- Tags can be a pointer to data of any datatype
  - E.g. char *myFlag; float *array
  - Can be a pointer to an explicit signal or even to the data being transferred.
- If you are using signal() or wait()
  - You must specify an explicit card (e.g. target(mic:0) )
  - #pragma offload_wait must have a wait() parameter

```c
float *inArray = new float[10000]; char mySignal;
#pragma offload_transfer target(mic:0) signal(inArray)\
in(inArray:length(theSize))
#pragma offload target(mic:0) signal(&mySignal) \n
copy(inArray)
```
Explicit Data Copy Details – Persistence of Pointer Data

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

• This behavior can be modified
  – free_if(boolean) controls space deallocation on the coprocessor at the end of the offload
  – alloc_if(boolean) controls space allocation on coprocessor at the start of the offload
  – Use nocopy rather than in/out/inout to indicate that the variable’s value is reused from a previous offload or is only relevant within this offload section
Explicit Data Copy Details – Persistence of Pointer Data Example (1)

- Allocate space on coprocessor, transfer data to, and do not release at end (persist)
- Use persisting data in subsequent offload code
- At end, transfer data from, and deallocate

```c

// Transfer matrices A, B, and C to an Intel(R) Xeon Phi(tm) device and do not deallocate matrices A and B
#pragma offload target(mic) \\ 
  in(transa, transb, M, N, K, alpha, beta, LDA, LDB, LDC) \\ 
  in(A:length(NCOLA * LDA) alloc_if(1) free_if(0)) \\ 
  in(B:length(NCOLB * LDB) alloc_if(1) free_if(0)) \\ 
  inout(C:length(N * LDC)) \\
{
  sgemm(&transa, &transb, &M, &N, &K, &alpha, A, &LDA, B, &LDB, &beta, C, &LDC);
}
```
Explicit Data Copy Details – Persistence of Pointer Data Example (2)

// Transfer matrix C1 to Intel(R) Xeon Phi(TM) device and reuse matrices A and B
#pragma offload target(mic) \
    in(transa1, transb1, M, N, K, alphal, betal, LDA, LDB, LDC1) \
    nocopy (A:length(NCOLA * LDA) alloc_if(0) free_if(0)) \ 
    nocopy (B:length(NCOLB * LDB) alloc_if(0) free_if(0)) \ 
    inout(C1:length(N * LDC1)) 
{
    sgemm(&transal, &transbl, &M, &N, &K, &alphal, A, &LDA, B, &LDB, &betal, C1, &LDC1);
}

// Deallocate A and B on an Intel(R) Xeon Phi(TM) device
#pragma offload target(mic) \
    nocopy (A:length(NCOLA * LDA) alloc_if(0) free_if(1)) \ 
    nocopy (B:length(NCOLB * LDB) alloc_if(0) free_if(1)) \ 
    inout(C1:length(N * LDC1)) 
{
    x = stuff(C1);
}
Explicit Data Copy Details – Persistence of Pointer Data Example (2)

// Transfer matrix C1 to an Intel(R) Xeon Phi(TM) device
// and reuse matrices A and B
#pragma offload target(mic) \
  in(transa1, transb1, M, N, K, alphal, betal, LDA, LDB, LDC1) \
  nocopy(A:length(NCOLA * LDA) alloc_if(0) free_if(0)) \ // e.g. REUSE
  nocopy(B:length(NCOLB * LDB) alloc_if(0) free_if(0)) \ //and RETAIN
  inout(C1:length(N * LDC1))
{
  sgemm(&transa1, &transb1, &M, &N, &K, &alphal, A, &LDA, B, &LDB, &betal, C1, &LDC1);
}

// Deallocate A and B on an Intel(R) Xeon Phi(TM) device
#pragma offload target(mic) \
  nocopy(A:length(NCOLA * LDA) alloc_if(0) free_if(1)) \ // e.g. REUSE
  nocopy(B:length(NCOLB * LDB) alloc_if(0) free_if(1)) \ //and FREE
  inout(C1:length(N * LDC1))
{
  x = stuff(C1);
}
Spectrum of Programming Models and Mindsets

Multi-Core Centric
- Xeon
- Multi-Core Hosted
  - General purpose serial and parallel computing
  - Offload
    - Codes with highly-parallel phases

Symmetric
- Codes with balanced needs
  - Main()
  - Foo()
  - MPI_*()

Many-Core Centric
- MIC
- Many Core Hosted
  - Highly-parallel codes
  - Main()
  - Foo()
  - MPI_*()

Range of models to meet application needs
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 SSE3 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