An Introduction to the Intel® Xeon Phi™ Coprocessor

INFIERI-2013  - July 2013

Leo Borges  (leonardo.borges@intel.com)
Intel Software & Services Group
Introduction

High-level overview of the Intel® Xeon Phi™ platform:
Hardware and Software

Intel Xeon Phi Coprocessor programming considerations:
Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Intel in High-Performance Computing

Dedicated, Renowned Applications Expertise
Large Scale Clusters for Test & Optimization
Tera-Scale Research
Exa-Scale Labs

Broad Software Tools Portfolio
Defined HPC Application Platform
Platform Building Blocks

Manufacturing Process Technologies
Leading Performance, Energy Efficient
Many Integrated Core Architecture

A long term commitment to the HPC market segment
HPC Processor Solutions

Multi-Core

**Xeon**
General Purpose Architecture
Leadership Per Core Performance
FP/core CAGR via AVX
Multi-Core CAGR

<table>
<thead>
<tr>
<th></th>
<th>EN</th>
<th>EP</th>
<th>EP 4S</th>
<th>Xeon EX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>General purpose perf/watt</td>
<td>Max perf/watt w/ Higher Memory BW / freq and QPI ideal for HPC</td>
<td>Additional compute density</td>
<td>Additional sockets &amp; big memory</td>
</tr>
</tbody>
</table>

Many-Core

**Intel® Xeon Phi™ Coprocessor**
Trades a “big” IA core for multiple lower performance IA cores resulting in higher performance for a subset of highly parallel applications

Common Intel Environment
Portable code, common tools
Highly Parallel Applications
Markets, Types, & Hardware

**Parallel Application Types**

- **Fine Grain**
- **Coarse Grain**
- **Embarrassingly Parallel**

**Hardware Options for Parallel Application Types**

- Communication
- Compute Process

Highly Parallel Compute Kernels

- Black-Scholes
- Sparse/Dense Matrix Mult
- FFTs
- Vector Math
- LU Factorization

Intel® MIC Architecture

Copyright© 2013, Intel Corporation. All rights reserved. *Other brands and names are the property of their respective owners.*
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

>50 in-order cores
  • Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family
  • Two pipelines
    - Dual issue with scalar instructions
  • One-per-clock scalar pipeline throughput
    - 4 clock latency from issue to resolution

4 hardware threads per core
  • Each thread issues instructions in turn
  • Round-robin execution hides scalar unit latency
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

- >50 in-order cores
  - Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core
- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

- >50 in-order cores
  - Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core
- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

- >50 in-order cores
  - Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family

- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit

>50 in-order cores

- Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family

- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency
Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread vector unit

- Optimized
  - Single and Double precision
- All new vector unit
  - 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
  - 32 512-bit wide vector registers
    - Hold 16 singles or 8 doubles per register
- Fully-coherent L1 and L2 caches
Reminder: Vectorization, What is it? (Graphical View)

Vectorization
- One Instruction
- Eight Mathematical Operations

Scalar
- One Instruction
- One Mathematical Operation

for (i=0; i<=MAX; i++)
    c[i] = a[i] + b[i];

1. Number of operations per instruction varies based on the which SIMD instruction is used and the width of the operands.
Data Types for Intel® MIC Architecture

now

16x floats
8x doubles

now

16x 32-bit integers
8x 64-bit integers

Takeaway: Vectorization is very important
Individual cores are tied together via fully coherent caches into a bidirectional ring.

- **Bidirectional ring**: 115 GB/sec
- **Distributed Tag Directory (DTD)** reduces ring snoop traffic
- **PCIe port** has its own ring stop

- **GDDR5 Memory**: 16 memory channels - Up to 5.5 Gb/sec
  - 8 GB 300ns access

- **L1**: 32K I- D-cache per core
  - 3 cycle access
  - Up to 8 concurrent accesses

- **L2**: 512K cache per core
  - 11 cycle best access
  - Up to 32 concurrent accesses

**Takeaway**: Parallelization and data placement are important.
## Intel® Xeon Phi™ Coprocessor x100 Family Reference Table

<table>
<thead>
<tr>
<th>Processor Brand Name</th>
<th>Codename</th>
<th>SKU #</th>
<th>Form Factor, Thermal</th>
<th>Board TDP (Watts)</th>
<th>Max # of Cores</th>
<th>Clock Speed (GHz)</th>
<th>Peak Double Precision (GFLOP)</th>
<th>GDDR5 Memory Speeds (GT/s)</th>
<th>Peak Memory BW</th>
<th>Memory Capacity (GB)</th>
<th>Total Cache (MB)</th>
<th>Enabled Turbo</th>
<th>Turbo Clock Speed (GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel® Xeon Phi™ Coprocessor x100 Knights Corner</td>
<td>7120P</td>
<td></td>
<td>PCIe Card, Passively Cooled</td>
<td>300</td>
<td>61</td>
<td>1.238</td>
<td>1208</td>
<td>5.5</td>
<td>352</td>
<td>16</td>
<td>30.5</td>
<td>Y</td>
<td>1.333</td>
</tr>
<tr>
<td></td>
<td>7120X</td>
<td></td>
<td>PCIe Card, No Thermal Solution</td>
<td>300</td>
<td>61</td>
<td>1.238</td>
<td>1208</td>
<td>5.5</td>
<td>352</td>
<td>16</td>
<td>30.5</td>
<td>Y</td>
<td>1.333</td>
</tr>
<tr>
<td></td>
<td>5120D</td>
<td></td>
<td>PCIe Dense Form Factor, No Thermal Solution</td>
<td>245</td>
<td>60</td>
<td>1.053</td>
<td>1011</td>
<td>5.5</td>
<td>352</td>
<td>8</td>
<td>30</td>
<td>N</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>3120P</td>
<td></td>
<td>PCIe Card, Passively Cooled</td>
<td>300</td>
<td>57</td>
<td>1.1</td>
<td>1003</td>
<td>5.0</td>
<td>240</td>
<td>6</td>
<td>28.5</td>
<td>N</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>3120A</td>
<td></td>
<td>PCIe Card, Actively Cooled</td>
<td>300</td>
<td>57</td>
<td>1.1</td>
<td>1003</td>
<td>5.0</td>
<td>240</td>
<td>6</td>
<td>28.5</td>
<td>N</td>
<td>N/A</td>
</tr>
<tr>
<td>Previously Launched and Disclosed</td>
<td>5110P*</td>
<td></td>
<td>PCIe Card, Passively Cooled</td>
<td>225</td>
<td>60</td>
<td>1.053</td>
<td>1011</td>
<td>5.0</td>
<td>320</td>
<td>8</td>
<td>30</td>
<td>N</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*Please refer to our technical documentation for Silicon stepping information.
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Intel® Parallel Studio XE 2013 and Intel® Cluster Studio XE 2013

- Industry-leading performance from advanced compilers
- Comprehensive libraries
- Parallel programming models
- Insightful analysis tools
Comprehensive set of SW tools

**Code Analysis**
- Advisor XE
- VTune Amplifier XE
- Inspector XE
- Trace Analyzer

**Libraries & Compilers**
- Math Kernel Library
- Integrated Performance Primitives
- Intel Compilers

**Programming Models**
- Intel Cilk Plus
- Threading Building Blocks
- OpenMP
- OpenCL
- MPI
- Offload/Native/MYO
Preserve Your Development Investment
Common Tools and Programming Models for Parallelism

Develop Using Parallel Models that Support Heterogeneous Computing
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations:

- Native
- Offload
  - Explicit block data transfer
  - Offloading with Virtual Shared Memory

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector
Spectrum of Programming & Execution Models

### Multicore Centric
- (Intel® Xeon® processors)
- Multi-core-hosted
- **General purpose**
  - Serial and parallel computing
- Codes with highly-parallel phases
  - `Main()`, `Foo()`, `MPI_*()`

### Many-core Centric
- (Intel® Many Integrated Core co-processors)
- Many-core-hosted
- **Codes with balanced needs**
  - `Main()`, `Foo()`, `MPI_*()`
- **Highly-parallel codes**
  - `Main()`, `Foo()`, `MPI_*()`

### Range of Models to Meet Application Needs
- Multicore
- Many-core

Optimization Notice

23
Intel® Xeon Phi™ Coprocessor runs *either* as an accelerator for offloaded host computation

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

**Diagram**

- **Host Processor**
  - Host-side offload application
    - User code
  - Offload libraries, user-level driver, user-accessible APIs and libraries
  - User-level code
  - System-level code
  - Intel® Xeon Phi™ Coprocessor support libraries, tools, and drivers
    - Linux* OS
    - PCI-E Bus

- **Intel® Xeon Phi™ Coprocessor**
  - Target-side offload application
    - User code
  - Offload libraries, user-accessible APIs and libraries
  - User-level code
  - System-level code
  - Intel® Xeon Phi™ Coprocessor communication and application-launch support
    - Linux* OS
    - PCI-E Bus

*Other brands and names are the property of their respective owners.*
**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

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

**Intel® Xeon Phi™ Coprocessor**
- Target-side "native" application
  - User code
  - Standard OS libraries plus any 3rd-party or Intel libraries
- Virtual terminal session
- System-level code
- User-level code

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

**Host Processor**
- System-level code
- User-level code

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

**Linux* OS**

**PCI-E Bus**

**IB fabric**

**ssh or telnet connection to coprocessor IP address**
The Intel® Manycore Platform Software Stack (Intel® MPSS) provides Linux* on the coprocessor

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
```

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

Intel MPSS supplies a virtual FS and native execution

```
sudo scp /opt/intel/composerxe/lib/mic/libiomp5.so root@mic0:/lib64
scp 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

```c++
#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

```c++
#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

**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

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

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

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

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

h() {
    ...
}
```
Example: Compiler Assisted Offload

- Offload section of code to the coprocessor.

```c
float pi = 0.0f;
#pragma offload target(mic)
#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);
}
pi /= count;
```

- Offload any function call to the coprocessor.

```c
#pragma offload target(mic) \ 
    in(transa, transb, N, alpha, beta) \ 
    in(A:length(matrix_elements)) \ 
    in(B:length(matrix_elements)) \ 
    in(C:length(matrix_elements)) \ 
    out(C:length(matrix_elements) alloc_if(0))
{
    sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, 
          &beta, C, &N);
}
```
Example: Compiler Assisted Offload

• An example in Fortran:

```fortran
!DEC$ ATTRIBUTES OFFLOAD : TARGET( MIC ) :: SGEMM
!DEC$ OMP OFFLOAD TARGET( MIC ) &
!DEC$ IN( TRANSA, TRANSB, M, N, K, ALPHA, BETA, LDA, LDB, LDC ), &
!DEC$ IN( A: LENGTH( NCOLA * LDA ) ), &
!DEC$ IN( B: LENGTH( NCOLB * LDB ) ), &
!DEC$ INOUT( C: LENGTH( N * LDC ) )
CALL SGEMM( TRANSA, TRANSB, M, N, K, ALPHA, &
            A, LDA, B, LDB BETA, C, LDC )
```
Example – share work between coprocessor and host using OpenMP*

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

Top level, runs on host
Runs on coprocessor
Runs on host
## Pragmas and directives mark data and code to be offloaded and executed

<table>
<thead>
<tr>
<th></th>
<th>C/C++ Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Offload pragma</strong></td>
<td>#pragma offload &lt;clauses&gt; &lt;statement&gt;</td>
</tr>
<tr>
<td></td>
<td>Allow next statement to execute on coprocessor or host CPU</td>
</tr>
<tr>
<td><strong>Variable/function offload properties</strong></td>
<td><strong>attribute</strong>((target(mic)))</td>
</tr>
<tr>
<td></td>
<td>Compile function for, or allocate variable on, both host CPU</td>
</tr>
<tr>
<td></td>
<td>and coprocessor</td>
</tr>
<tr>
<td><strong>Entire blocks of data/code defs</strong></td>
<td>#pragma offload_attribute(push, target(mic))</td>
</tr>
<tr>
<td></td>
<td>#pragma offload_attribute(pop)</td>
</tr>
<tr>
<td></td>
<td>Mark entire files or large blocks of code to compile for both</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Fortran Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Offload directive</strong></td>
<td>!dir$ omp offload &lt;clauses&gt; &lt;statement&gt;</td>
</tr>
<tr>
<td></td>
<td>Execute OpenMP* parallel block on coprocessor</td>
</tr>
<tr>
<td></td>
<td>!dir$ offload &lt;clauses&gt; &lt;statement&gt;</td>
</tr>
<tr>
<td></td>
<td>Execute next statement or function on coproc.</td>
</tr>
<tr>
<td><strong>Variable/function offload properties</strong></td>
<td>!dir$ attributes offload:&lt;mic&gt; :: &lt;ret-name&gt; OR &lt;var1,var2,..&gt;</td>
</tr>
<tr>
<td></td>
<td>Compile function or variable for CPU and coprocessor</td>
</tr>
<tr>
<td><strong>Entire code blocks</strong></td>
<td>!dir$ offload begin &lt;clauses&gt;</td>
</tr>
<tr>
<td></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) / manadatory</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>

### Modifiers

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Syntax</th>
<th>Semantics</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 \text{ bytes} )</code></td>
<td>Specify minimum memory alignment on coprocessor</td>
</tr>
<tr>
<td>Array partial allocation &amp; variable relocation</td>
<td><code>alloc( array-slice ) into( var-expr )</code></td>
<td>Enables partial array allocation and data copy into other vars &amp; ranges</td>
</tr>
</tbody>
</table>
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
Example: Virtual Shared Memory

- Shared between host and Xeon Phi

```c
// Shared variable declaration
__Cilk_shared T in1[SIZE];
__Cilk_shared T in2[SIZE];
__Cilk_shared T res[SIZE];

__Cilk_shared void compute_sum()
{
    int i;
    for (i=0; i<SIZE; i++) {
        res[i] = in1[i] + in2[i];
    }
}

(...)

// Call compute sum on Target
__Cilk_offload compute_sum();
```
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:

```
_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>
<th>Description</th>
</tr>
</thead>
</table>
| Function              | ```
int _Cilk_shared f(int x){ return x+1; }
```                                                                           | Code emitted for host and target; may be called from either side              |
| Global                | `_Cilk_shared int x = 0;`                                               | Datum is visible on both sides                                              |
| File/Function static  | ```
static _Cilk_shared int x;
```                                                                               | Datum visible on both sides, only to code within the file/function           |
| Class                 | ```
class _Cilk_shared x {...};
```                                                                             | Class methods, members and operators available on both sides                 |
| Pointer to shared data| ```
int _Cilk_shared *p;
```                                                                               | `p` is local (not shared), can point to shared data                         |
| A shared pointer      | ```
int *_Cilk_shared p;
```                                                                               | `p` is shared; should only point at shared data                              |
| Entire blocks of code | ```
#pragma offload_attribute( push, _Cilk_shared)
```                                                                     | Mark entire files or blocks of code _Cilk_shared using this pragma          |
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>
<tbody>
<tr>
<td>Offloading a function call</td>
<td>x = _Cilk_offload func(y);</td>
</tr>
<tr>
<td></td>
<td>func executes on coprocessor if possible</td>
</tr>
<tr>
<td></td>
<td>x = _Cilk_offload_to (card_num) func(y);</td>
</tr>
<tr>
<td></td>
<td>func must execute on specified coprocessor or an error occurs</td>
</tr>
<tr>
<td>Offloading asynchronously</td>
<td>x = _Cilk_spawn _Cilk_offload func(y);</td>
</tr>
<tr>
<td></td>
<td>func executes on coprocessor; continuation available for stealing</td>
</tr>
<tr>
<td>Offloading a parallel for-loop</td>
<td>_Cilk_offload _Cilk_for(i=0; i&lt;N; i++){</td>
</tr>
<tr>
<td></td>
<td>a[i] = b[i] + c[i];</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>Loop executes in parallel on coprocessor. The loop is implicitly “un-inlined” as a function call.</td>
</tr>
</tbody>
</table>
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Options for Thread Parallelism

- Intel® Math Kernel Library
- OpenMP*
- Intel® Threading Building Blocks
  - Intel® Cilk™ Plus
  - OpenCL*
- Pthreads* and other threading libraries

Ease of use / code maintainability

Programmer control

Choice of unified programming to target Intel® Xeon® and Intel® Xeon Phi™ Architecture!
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism: OpenMP

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
OpenMP* on the Coprocessor

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

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

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

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

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

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

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

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

Or define MIC – specific versions of env vars using

- `MIC_ENV_PREFIX=MIC` (no underscore)
- Values on MIC no longer default to values on host
- Set values specific to MIC using
  - `export MIC_OMP_NUM_THREADS=120` (all cards)
  - `export MIC_2_OMP_NUM_THREADS=180` for card #2, etc
  - `export MIC_3_ENV="OMP_NUM_THREADS=240|KMP_AFFINITY=balanced"`
Stack Sizes for Coprocessor

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

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

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

Allows OpenMP threads to be bound to physical or logical cores

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

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

- See compiler documentation for (much) more detail
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism: TBB

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Intel® Threading Building Blocks

Widely used C++ template library for parallelism

C++ Library for parallel programming
• Takes care of managing multitasking

Runtime library
• Scalability to available number of threads

Cross-platform
• Windows, Linux, Mac OS* and others

http://threadingbuildingblocks.org
Intel® Threading Building Blocks

- **Generic Parallel Algorithms**
  Efficient scalable way to exploit the power of multi-core without having to start from scratch

- **Concurrent Containers**
  Common idioms for concurrent access
  - a scalable alternative serial container with a lock around it

- **Task scheduler**
  The engine that empowers parallel algorithms that employs task-stealing to maximize concurrency

- **Threads**
  - Thread-safe timers
  - OS API wrappers

- **Miscellaneous**
  Thread-safe timers

- **TBB Flow Graph**

- **Thread Local Storage**
  Scalable implementation of thread-local data that supports infinite number of TLS

- **Synchronization Primitives**
  User-level and OS wrappers for mutual exclusion, ranging from atomic operations to several flavors of mutexes and condition variables

- **Memory Allocation**
  Per-thread scalable memory manager and false-sharing free allocators
**parallel_for usage example**

```cpp
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
using namespace tbb;

class ChangeArray{
  int* array;
public:
  ChangeArray(int* a): array(a) {}
  void operator()(const blocked_range<int>& r) const {
    for (int i = r.begin(); i != r.end(); i++) {
      Foo (array[i]);
    }
  }
};

int main (){
  int a[n];
  // initialize array here...
  parallel_for(blocked_range<int>(0, n), ChangeArray(a));
  return 0;
}
```

- ChangeArray class defines a for-loop body for parallel_for.
- blocked_range – TBB template representing 1D iteration space.
- As usual with C++ function objects the main work is done inside operator().
- A call to a template function parallel_for<Range, Body>: with arguments
  Range → blocked_range
  Body → ChangeArray
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism: MKL

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Intel® MKL is industry’s leading math library

- **Linear Algebra**
  - BLAS
  - LAPACK
  - Sparse solvers
  - ScalAPACK

- **Fast Fourier Transforms**
  - Multidimensional (up to 7D)
  - FFTW interfaces
  - Cluster FFT

- **Vector Math**
  - Trigonometric
  - Hyperbolic
  - Exponential, Logarithmic
  - Power/Root
  - Rounding

- **Vector Random Number Generators**
  - Congruential
  - Recursive
  - Wichmann-Hill
  - Mersenne Twister
  - Sobol
  - Neiderreiter
  - Non-deterministic

- **Summary Statistics**
  - Kurtosis
  - Variation coefficient
  - Quantiles, order statistics
  - Min/max
  - Variance-covariance
  - ...

- **Data Fitting**
  - Splines
  - Interpolation
  - Cell search

* 2011 & 2012 Evans Data N. American developer surveys

---

*Other brands and names are the property of their respective owners.*
MKL Usage Models on Intel® Xeon Phi™ Coprocessor

- **Automatic Offload**
  - No code changes required
  - Automatically uses both host and target
  - Transparent data transfer and execution management

- **Compiler Assisted Offload**
  - Explicit controls of data transfer and remote execution using compiler offload pragmas/directives
  - Can be used together with Automatic Offload

- **Native Execution**
  - Uses the coprocessors as independent nodes
  - Input data is copied to targets in advance
MKL Execution Models

Multicore Centric

- **Multicore Hosted**
  - General purpose serial and parallel computing
  - **Offload**
    - Codes with highly-parallel phases

- **Symmetric**
  - Codes with balanced needs

Many-Core Centric

- **Many Core Hosted**
  - Highly-parallel codes

- **Native**

---

*Other brands and names are the property of their respective owners.*
## Work Division Control in MKL Automatic Offload

<table>
<thead>
<tr>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_mic_set_Workdivision(MKL_TARGET_MIC, 0, 0.5)</td>
<td>Offload 50% of computation only to the 1\text{st} card.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_MIC_0_WORKDIVISION=0.5</td>
<td>Offload 50% of computation only to the 1\text{st} card.</td>
</tr>
</tbody>
</table>
How to Use MKL with Compiler Assisted Offload

• The same way you would offload any function call to the coprocessor.

• An example in C:

```c
#pragma offload target(mic) \ 
in(transa, transb, N, alpha, beta) \ 
in(A:length(matrix_elements)) \ 
in(B:length(matrix_elements)) \ 
in(C:length(matrix_elements)) \ 
out(C:length(matrix_elements) alloc_if(0))
{
    sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);
}
```
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
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
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

Homogenous network of many-core CPUs

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
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

Heterogeneous network of homogeneous CPUs

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
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

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
Introduction

High-level overview of the Intel® Xeon Phi™ platform:
Hardware and Software

Intel Xeon Phi Coprocessor programming considerations:
Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Intel® Trace Analyzer and Collector

Compare the event timelines of two communication profiles

Blue = computation
Red = communication

Chart showing how the MPI processes interact
Intel® Trace Analyzer and Collector Overview

Intel® Trace Analyzer and Collector helps the developer:

• Visualize and understand parallel application behavior
• Evaluate profiling statistics and load balancing
• Identify communication hotspots

Features

• Event-based approach
• Low overhead
• Excellent scalability
• Comparison of multiple profiles
• Powerful aggregation and filtering functions
• Fail-safe MPI tracing
• Provides API to instrument user code
• MPI correctness checking
• Idealizer
Full tracing functionality on Intel® Xeon Phi™ coprocessor
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Collecting Hardware Performance Data

Hardware counters and events
• 2 counters in core, most are thread specific
• 4 outside the core (uncore) that get no thread or core details
• See PMU documentation for a full list of events

Collection
• Invoke from Intel® VTune™ Amplifier XE
• If collecting more than 2 core events, select multi-run for more precise results or the default multiplexed collection, all in one run
• Uncore events are limited to 4 at a time in a single run
• Uncore event sampling needs a source of PMU interrupts, e.g. programming cores to CPU_CLK_UNHALTED

Output files
• Intel VTune Amplifier XE performance database
Intel® VTune™ Amplifier XE offers a rich GUI

Menu and Tool bars
Analysis Type
Viewpoint currently being used
Tabs within each result
Grid area
Current grouping
Stack Pane
Timeline area
Filter area
Intel® VTune™ Amplifier XE on Intel® Xeon Phi™ coprocessors

Adjust Data Grouping
- Function - Call Stack
- Module - Function - Call Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack
... (Partial list shown)

No Call Stacks Yet

Double Click Function to View Source

Filter by Timeline Selection (or by Grid Selection)

Filter by Module & Other Controls

Copyright© 2013, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Intel® VTune™ Amplifier XE displays event data at function, source & assembly levels

- Time on Source / Asm
- Quick Asm navigation: Select source to highlight Asm
- Quickly scroll to hot spots. Scroll Bar “Heat Map” is an overview of hot spots
- Right click for instruction reference manual
- Click jump to scroll Asm
Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions
Conclusions: Intel® Xeon Phi™ Coprocessor supports a variety of programming models

The familiar Intel development environment is available:

- Intel® Composer: C, C++ and Fortran Compilers
- OpenMP*
- Intel® MPI Library support for the Intel® Xeon Phi™ Coprocessor
  - Use as an MPI node via TCP/IP or OFED
- Parallel Programming Models
  - Intel® Threading Building Blocks (Intel® TBB)
  - Intel® Cilk™ Plus
- 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 Developer site:
http://software.intel.com/mic-developer

One Stop Shop for:

- Tools & Software Downloads
- Getting Started Development Guides
- Video Workshops, Tutorials, & Events
- Code Samples & Case Studies
- Articles, Forums, & Blogs
- Associated Product Links

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

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

• Developer’s Quick Start Guide
• Programming Overview
• New User Forum at


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

Intel Premier Support  https://premier.intel.com
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
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:

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

To reuse that variable and keep it again:

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

To reuse one more time, then discard:

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