Overview: Programming Environment for Intel® Xeon Phi™ Coprocessor
One Source Base, Tuned to many Targets

Compilers, Libraries, Parallel Models

- Multicore
- Many-core
- Cluster

- Multicore CPU
- Multicore CPU
- Intel® MIC Architecture
- Multicore Cluster
- Multicore and Many-core Cluster
# Intel® Parallel Studio XE 2013 and Intel® Cluster Studio XE 2013

<table>
<thead>
<tr>
<th>Phase</th>
<th>Product</th>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build</td>
<td>Intel® Advisor XE</td>
<td>Threading design assistant (Studio products only)</td>
<td>• Simplifies, demystifies, and speeds parallel application design</td>
</tr>
</tbody>
</table>
|             | Intel® Composer XE | • C/C++ and Fortran compilers  
• Intel® Threading Building Blocks  
• Intel® Cilk™ Plus  
• Intel® Integrated Performance Primitives  
• Intel® Math Kernel Library | • Enabling solution to achieve the application performance and scalability benefits of multicore and forward scale to many-core |
|             | Intel® MPI Library† | High Performance Message Passing (MPI) Library                         | • Enabling High Performance Scalability, Interconnect Independence, Runtime Fabric Selection, and Application Tuning Capability |
| Verify & Tune | Intel® VTune™ Amplifier XE | Performance Profiler for optimizing application performance and scalability | • Remove guesswork, saves time, makes it easier to find performance and scalability bottlenecks |
|             | Intel® Inspector XE | Memory & threading dynamic analysis for code quality  
Static Analysis for code quality | • Increased productivity, code quality, and lowers cost, finds memory, threading, and security defects before they happen |
|             | Intel® Trace Analyzer & Collector† | MPI Performance Profiler for understanding application correctness & behavior | • Analyze performance of MPI programs and visualize parallel application behavior and communications patterns to identify hotspots |
Native Models

Parallel programming is the same on coprocessor and host
Intel® MIC Centric
Native MIC Programming

Enabled by –mmic compiler option

• Fully supported by compiler vectorization, Intel® MKL, OpenMP®, Intel® TBB, Intel® Cilk Plus, Intel® MPI, ...
  • No Intel® Integrated Performance Primitives library yet

• For sure, an option for some applications:
  • Needs to fit into memory (up to 16GB today)
  • Should be highly parallel code
  • Serial parts are slower on MIC than on host
  • Limited access to external environment like I/O
  • Native MIC file system exists in memory only
  • NFS allows external I/O but limited bandwidth
Parallel programming is the same on coprocessor and host
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

Multi-Core Hosted
Codes with highly-parallel phases

Main( ) Foo( ) MPI_*() Main( ) Foo( ) MPI_*() Main( ) Foo( ) MPI_*()

Multi-core (Xeon)

Many-core (MIC)

Range of models to meet application needs

Copyright © 2014, Intel Corporation. All rights reserved.*Other brands and names are the property of their respective owners.
Offload Models

• Intel® Xeon Phi™ supports two offload models:
  – **Explicit:**
    Data transfers from host to/from coprocessor are initiated by programmer
  – **Implicit:**
    Data is (virtually) shared (VSHM) between host and coprocessor

• Also called LEO (Language Extensions for Offload)
Explicit Offload Model

- The programmer explicitly control data and function movement between the host and target(s)
  - Data is copied (not shared)
  - Must be bitwise copy-able (pointers NOT relocated)
- Supported for Fortran, C/C++
Explicit Offload Model - Use

Explicit offloading requires user to manage data persistence:

- Data/Functions marked as...
  
  **C/C++:**
  - `#pragma offload_attribute(push, target(mic))`
  - `#pragma offload_attribute(pop))`
  - `_attribute__((target(mic)))`

  **Fortran:**
  - `!DIR$ OPTIONS /OFFLOAD_ATTRIBUTE_TARGET=mic`
  - `!DIR$ ATTRIBUTES OFFLOAD:mic :: <subroutine>`

Will exist on both the host and target systems and copied between host and target when referenced.

- Named targets
  - `target(mic)`: runtime picks the card
  - `target(mic:n)`: explicitly name the logical card number `n`
Explicit Offload Model - Use

Pure data transfer:
• #pragma offload_transfer target(mic0)
• !DIR$ offload_transfer target(mic0)
• Asynchronous transfers:
  Clauses signal(<id>) & wait(<id>)

Offloading code:
• #pragma offload target(mic0) <code_scope>
• !DIR$ offload target(mic0) <code_scope>
Explicit Offload Model – Memory

- Use `in/out/inout/nocopy` clauses to specify data transfer and direction.
- Use `alloc_if([0|1])` and `free_if([0|1])` to conditionally manage memory allocation on the target.
- With `into(...)` clause you can specify data to be moved to other variables/memory.

What can persist?

<table>
<thead>
<tr>
<th>Type of variable</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static or global</td>
<td>Yes</td>
</tr>
<tr>
<td>Heap</td>
<td>Yes</td>
</tr>
<tr>
<td>Scalar on stack</td>
<td>Yes</td>
</tr>
<tr>
<td>Array on stack</td>
<td>Not yet</td>
</tr>
</tbody>
</table>
Explicit Offload Model – Example

Allocate persisted data on heap:
Create structure to track persistent data, pass it as a parameter between functions

```c
struct pcache {
    int *p1;
};

#define N 128
int main(int argc, char **argv)
{
    struct pcache *share = start_job();
    continue_job(share);
    return 0;
}
```

Declaration of struct type with link to dynamic data

Two separate functions operate on same data
struct pcache *start_job() {
    int *A = (int *)malloc(N * sizeof(int));
    for (i=0; i < N; ++i) A[i] = i;

    struct pcache *mycache =
        (struct pcache *)malloc(sizeof(pcache));
    mycache->p1 = A;

#pragma offload target(mic:0) in(A:length(N) free_if(0))
{
    for (i=0; i < N; ++i) {
    }
}
return mycache;
}
Explicit Offload Model – Example cont’d

```c
void continue_job(struct pcache *mine) {
    int i;
    int *A = mine->p1;

    #pragma offload target(mic:0) \
    in(A:length(0) alloc_if(0) free_if(0))
    {
        for (i=0;i < N; ++i) {
        }
    }
}
```

- Only minimal changes required!
- The same code also runs on host:
  - Compile with **-no-offload** or
  - run on host without target

Get the pointer to existing data from the “cache”

(Re-)Use already allocated data
Implicit Offload Model

- Implicit Offloading: Virtual Shared Memory
  - User code declares data objects to be shared:
    - Data allocated at same address on host and target
    - Modified data is copied at synchronization points
    - Allows sharing of complex data structures
    - No data marshaling necessary
- Supported only for C and C++
- Can’t propagate exceptions between host & target
#define N 20000.0

_Cilk_shared float FindArea(float r) {
    float x, y, area;
    unsigned int seed = __cilkrts_get_worker_number();

    cilk::reducer_opadd<int> inside(0);
    cilk_for(int i = 0; i < N; i++) {
        x = (float)rand_r(&seed)/RAND_MAX;
        y = (float)rand_r(&seed)/RAND_MAX;
        x = 2.0 * x - 1.0;
        y = 2.0 * y - 1.0;
        if (x * x + y * y < r * r) inside++;
    }

    area = 4.0 * inside.get_value() / N;
    return area;
}
int main(int argc, char **argv)
{
    // Get r1 & r2 from user...
    Area1 = cilk_spawn _Cilk_offload FindArea(r1);
    Area2 = FindArea(r2);
    cilk_sync;

    float Donut = Area1 - Area2;
    float PI = 3.14159265;
    float AreaR = PI * (r2 * r2 - r1 * r1);
    float Accuracy = 100 * (1 - fabs(Donut - AreaR)/AreaR);
    printf("Area1=%lf, Area2=%lf\n", Area1, Area2);
    printf("Donut =%lf, Accuracy = %lf\n", Donut, Accuracy);
}
Implicit Offload Model

Preferred if data to be transferred is complex:
It can use pointers, and any composition of structs and pointers!

- Allocation via `_Offload_shared_[aligned_]malloc(…)`
- De-allocation via `_Offload_shared_[aligned_]free(…)`

Target can be specified as well:
`__Cilk_offload_to(<target-number>)`

Same as for explicit offload model:
- Only minimal changes required!
- The same code also runs on host:
  - Compile with `-no-offload` or
  - run on host without target
What is needed?

Install Intel® Manycore Platform Software Stack package:

- Provides cross-compile environment
- Also contains Eclipse* IDE/Visual Studio* integration for debugging
- Use compilers from Intel® Composer XE 2013 and later

For offload models, pass options via `-offload-option`

Example:

```
$ icc test.c -O2 -offload-option,mic,compiler,"-O3 -vec-report3"
```

Native build also possible by option `-mmic`

More information reg. offload models:

OpenMP 4.0 TARGET – Offload Execution Model

The code section to be executed on accelerators are marked by a target construct.

- A target region is executed by a single thread, called the initial device thread
- Parallelism on accelerator is specified by traditional and extended Openmp-parallel constructs
- The task that encounters the target construct waits at the end of the construct until execution of the region completes
- If a target device does not exist or is not supported by the implementation, the target region is executed by the host device
- A data environment is set up via map clause

```
#pragma omp target [clause[,] clause],...
    structured-block
Clauses:
device(scalar-integer-expression)       // to select the device
map(alloc|to|from|tofrom : list) if(scalar-expr)  // data environment
```
#pragma omp target map(to:b[0:count])) map(to:c,d) map(from:a[0:count])
{
#pragma omp parallel for
    for (i=0; i<count; i++) {
        a[i] = b[i] * c + d;
    }
}

#pragma omp target data device(0) map(alloc:tmp[0:N]) map(to:input[0:N]) map(from:result)
{
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
    {
        tmp[i] = some_computation(input[i], i);
    }

    do_somel_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:result)
    for (i=0; i<N; i++)
    {
        result += final_computation(tmp[i], i)
    }
}
Support for SIMD Parallelism

But there is more than the parallelism offered by the many cores:

For good performance, it’s not sufficient to use all the cores, you need to use the 512 bit SIMD registers and (vector) instructions

Vector Processing Unit for Intel® IMCI - 512-bit Vector Execution Engine

- 16 lanes of 32-bit single precision and integer operations
- 8 lanes of 64-bit double precision and integer operations
- 32 512-bit general purpose vector registers in 4 threads
- 8 16-bit mask registers in 4 threads for predicated execution

Read/Write
- One vector length (512-bits) per cycle from/to Vector Registers
- One operand can be from the memory

IEEE 754 Standard Compliance
- 4 rounding Model, even, 0, +∞, -∞
- Hardware support for SP/DP denormal handling

Intel® MIC Vector Types

Intel® Many Integrated Core (Intel® MIC):

First Generation

16x single precision FP
8x double precision FP
16x 32 bit integer

Note:

It just has begun – more to come, but only what’s needed!

High level language complex type can also be used; compiler cares about details (halves the potential vector length)

Use 32 bit integers where possible (short & char types will be converted implicitly, though)
Many Ways to Exploit SIMD Parallelism

Compiler: Auto-vectorization (no change of code)

Compiler: Auto-vectorization hints (#pragma vector, ...)

Compiler: Intel® Cilk™ Plus Data Parallel Extensions

SIMD intrinsic class (e.g.: F32vec, F64vec, ...)

Vector intrinsic (e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

Assembler code (e.g.: [v]addps, [v]addss, ...)

Ease of use

Programmer control
Auto-vectorization

- The vectorizer for Intel® MIC architecture works just like for SSE or AVX on the host, for C, C++ and Fortran
  - Enabled at default optimization level (-O2)
  - Data alignment should be to 64 bytes, instead of 16 (see later)
  - More loops can be vectorized, because of masked vector instructions, gather/scatter instructions, fused multiply-add (FMA)
  - Try to avoid 64 bit integers (except as addresses)

- Vectorized loops may be recognized by:
  - Vectorization and optimization reports (simplest), e.g.
    - vec-report2 or -opt-report-phase hpo
  - Unmasked vector instructions (there are no separate scalar instructions; masked vector instructions are used instead)
  - Gather & scatter instructions
  - Math library calls to libsvml
**Intel® Cilk™ Plus**

### Task parallelism

**Simple Keywords**
Set of keywords, for expression of task parallelism:
- `cilk_spawn`
- `cilk_sync`
- `cilk_for`

**Reducers (Hyper-objects)**
Reliable access to nonlocal variables without races
- `cilk::reducer_opadd<int> sum(3);`

### Data parallelism

**Array Notation**
Provide data parallelism for sections of arrays or whole arrays
- `mask[:] = a[:] < b[:] ? -1 : 1;`

**Elemental Functions**
Define actions that can be applied to whole or parts of arrays or scalars

### Execution parameters
Runtime system APIs, Environment variables, pragmas
**Intel® Cilk™ Plus Array Notation**

**Simple example:**
Serial version:

```c
float dot_product(unsigned int size, float A[size], float B[size])
{
    int i;
    float dp = 0.0f;
    for (i=0; i<size; i++) {
        dp += A[i] * B[i];
    }
    return dp;
}
```

Array Notation version:

```c
float dot_product(unsigned int size, float A[size], float B[size])
{
    return __sec_reduce_add(A[:] * B[:]);
    // A[: ] can also be written as A[0:size]
}
```
**Intel® Cilk™ Plus Array Notation Syntax**

- Use a “:” in array subscripts to operate on multiple elements
- Array notation returns a subset of the referenced array
- “length” specifies number of elements of subset
- “stride”: distance between elements for subset
- “length” and “stride” are optional (all & stride 1 are default)

```
A[:]
A[start_index : length]
A[start_index : length : stride]
```

Explicit Data Parallelism Based on C/C++ Arrays
Intel® Cilk™ Plus

Task parallelism

Simple Keywords
Set of keywords, for expression of task parallelism:
- `cilk_spawn`
- `cilk_sync`
- `cilk_for`

Reducers (Hyper-objects)
Reliable access to nonlocal variables without races
```cpp
cilk::reducer_opadd<int> sum(3);
```

Data parallelism

Array Notation
Provide data parallelism for sections of arrays or whole arrays
```cpp
mask[:] = a[:] < b[:] ? -1 : 1;
```

SIMD Enabled Functions
Define actions that can be applied to whole or parts of arrays or scalars

Execution parameters
Runtime system APIs, Environment variables, pragmas
Intel® Cilk™ Plus SIMD Enabled Functions

Serial Code

```cpp
void saxpy(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i) y[i] += a * x[i];
}
```

```cpp
saxpy(2.0, x[0:n], y[0:n]);
```

User-defined Elemental Function

```cpp
__declspec(vector(uniform(a)))
void saxpy(float a, float x, float &y)
{
    y += a * x;
}
```

Compiler generates:

```cpp
void saxpy_v(float a, float x[VL], float y[VL])
{
    y[:] += a * x[:];
}
```

Compiler generates:

```cpp
parallel for(i = 0; i < n; i += VL)
{
    saxpy_v(2.0, &x[VL], &y[VL]);
}
```
__attribute__((vector))  (Intel)

#pragma omp declare simd  (OpenMP* 4.0)

Available clauses  (both OpenMP and Intel versions)
- LINEAR  (additional induction variables)
- UNIFORM  (arguments that are loop constants)
- REDUCTION
- PROCESSOR  (Intel)
- VECTORLENGTH  (Intel)
- MASK / NOMASK  (Intel)
- INBRANCH / NOTINBRANCH  (OpenMP 4.0)
- SIMDLEN  (OpenMP 4.0)
- ALIGNED  (OpenMP 4.0)
Intel® Cilk™ Plus

Task parallelism

Simple Keywords
Set of keywords, for expression of task parallelism:
- cilk_spawn
- cilk_sync
- cilk_for

Reducers
(Hyper-objects)
Reliable access to nonlocal variables without races
```cpp
cilk::reducer_opadd<int> sum(3);
```
Intel® Cilk™ PlusPragma/Directive

C/C++: #pragma simd [clause [,clause]…]
Fortran: !DIR$ SIMD [clause [,clause]…]
Without any clause, the directive “enforces” vectorization of the loop, ignoring all dependencies (even if they are proved!)

Example:

```c
void addfl(float *a, float *b, float *c, float *d, float *e, int n)
{
    #pragma simd
    for(int i = 0; i < n; i++)
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Without SIMD directive, vectorization likely fails since there are too many pointer references to do a run-time check for overlapping (compiler heuristic). The compiler won’t create multiple versions here. Using the directive asserts the compiler that none of the pointers are overlapping.
# Intel® Cilk™ Plus Pragma/Directive - Clauses

The programmer (i.e. you!) is responsible for correctness
Just like for race conditions in OpenMP* loops

<table>
<thead>
<tr>
<th>Available clauses</th>
<th>(both OpenMP and Intel versions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIVATE</td>
<td></td>
</tr>
<tr>
<td>LASTPRIVATE</td>
<td>--- like OpenMP</td>
</tr>
<tr>
<td>REDUCTION</td>
<td></td>
</tr>
<tr>
<td>COLLAPSE</td>
<td>(OpenMP 4.0 only; for nested loops)</td>
</tr>
<tr>
<td>LINEAR</td>
<td>(additional induction variables)</td>
</tr>
<tr>
<td>SAFELEN</td>
<td>(OpenMP 4.0 only)</td>
</tr>
<tr>
<td>VECTORLENGTH</td>
<td>(Intel only)</td>
</tr>
<tr>
<td>ALIGNED</td>
<td>(OpenMP 4.0 only)</td>
</tr>
<tr>
<td>ASSERT</td>
<td>(Intel only; default for OpenMP 4.0 )</td>
</tr>
</tbody>
</table>
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
Thread Affinity Interface

Allows OpenMP threads to be bound to physical or logical cores

- export environment variable **KMP_AFFINITY**=
  - **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
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

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

http://software.intel.com/mic-developer
• Developer’s Quick Start Guide
• Programming Overview
• 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 © 2014, 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