



# PROGRAMMING AND OPTIMIZATION FOR INTEL<sup>®</sup> ARCHITECTURE

The Hands-On Workshop (HOW) Series  
Session 2

*Colfax International* — [colfaxresearch.com](http://colfaxresearch.com)

October 2016

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

- ▶ HOW to Program Intel Architecture
  - 01. Parallelism, specialization, guided tour – Oct 24
  - 02. Programming Intel Xeon Phi (KNC, KNL) – Oct 25
- ▶ HOW to Express Parallelism
  - 03. Automatic vectorization – Oct 26
  - 04. Multi-threading with OpenMP – Oct 27
- ▶ HOW to Get Performance
  - 05. Comprehensive demo – Oct 28
  - 06. Scalar & vectorization tuning – Oct 31
  - 07. Multi-threading: common issues – Nov 1
  - 08. Multi-threading: memory aspect – Nov 2
  - 09. Memory traffic – Nov 3
- ▶ HOW to Scale
  - 10. Distributed Computing: MPI – Nov 4

October 2016						
S	M	T	W	H	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					
November 2016						
S	M	T	W	H	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			
■ — Webinar+remote access						

Course page: [colfaxresearch.com/how-16-10](http://colfaxresearch.com/how-16-10)

- ▶ Slides (including this one), code downloads
- ▶ Video of recorded sessions
- ▶ Chat (during webinars or offline)



Additional resources:

- ▶ More workshops like this one: [colfaxresearch.com/training](http://colfaxresearch.com/training)
- ▶ Video courses: [colfaxresearch.com/video-courses](http://colfaxresearch.com/video-courses)

# GET YOUR QUESTIONS ANSWERED

**Chat** (current):  
[colfaxresearch.com/how-16-10](http://colfaxresearch.com/how-16-10)



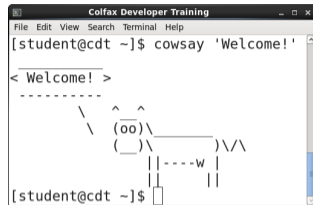
**Forums** (technical):  
[colfaxresearch.com/discussion](http://colfaxresearch.com/discussion)

A screenshot of the Colfax Research website. At the top right, there is a "Log In/Register" link. The main header features the "COLFAX RESEARCH" logo in green, with the tagline "CONTRIBUTING TO INNOVATIONS IN COMPUTING" below it. A dark navigation bar contains the following menu items: READ, WATCH, LEARN, FORUMS (highlighted in green), CONNECT, and JOIN. Below the navigation bar, the heading "Join the Conversation" is followed by a paragraph: "Welcome to Colfax Research forums, an online community for you to engage with HPC experts, software architects, developers, computational researchers, scientists, students and more—so you can acquire new knowledge, share ideas, and build new relationships." Another paragraph below reads: "Tap our experts and your peers to help meet the challenge of optimizing applications on modern hardware. This is the place to browse or post questions (and get answers) related to computational science, parallel programming and code modernization on Intel® Architecture." The final line says: "Welcome aboard. Post questions today!"

**Email** (organizational):  
[training@colfax-intl.com](mailto:training@colfax-intl.com)

# HANDS-ON EXERCISES AND REMOTE ACCESS

- ▶ 96 people receive a remote access token
- ▶ Virtualized Intel Xeon CPU, real Intel Xeon Phi coprocessor (1st gen, KNC), SW tools
- ▶ Can access the system the entire 2 weeks of the workshop



```
Colfax Developer Training
File Edit View Search Terminal Help
[student@cdt ~]$ cowsay 'Welcome!'
< Welcome! >
-----
      \   ^__^
         (oo)\_____)
            (_____)
                )\/)
                ||----w |
                ||     ||

[student@cdt ~]$
```

- ▶ Not among the 96? Stay tuned: follow along with instructor, use own system, or wait for a seat
- ▶ Use it or lose it: if you do not log in for a while, remote access token goes to next student on the list



## **§2. PROGRAMMING COPROCESSORS**



## **OFFLOAD AND NATIVE MODELS**

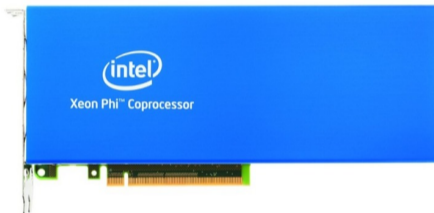
## Intel Xeon Processor



Current: Broadwell  
Upcoming: Skylake

Multi-Core Architecture

## Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

## Intel Xeon Phi Processor, 2nd generation\*



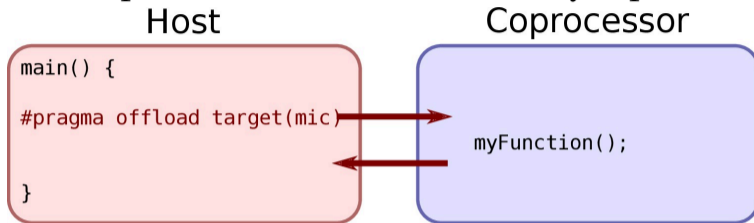
\* socket and coprocessor versions

Knights Landing (KNL)

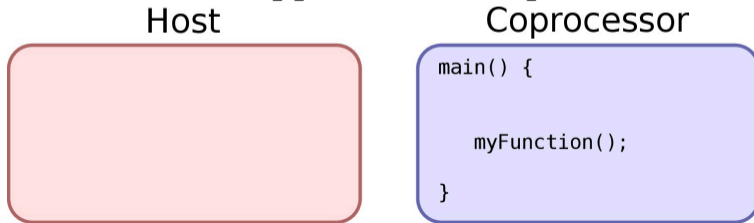
Intel Many Integrated Core (MIC) Architecture

# OFFLOAD AND NATIVE MODELS

- ▶ Offload model (explicit/virtual-shared memory/OpenMP 4.0):



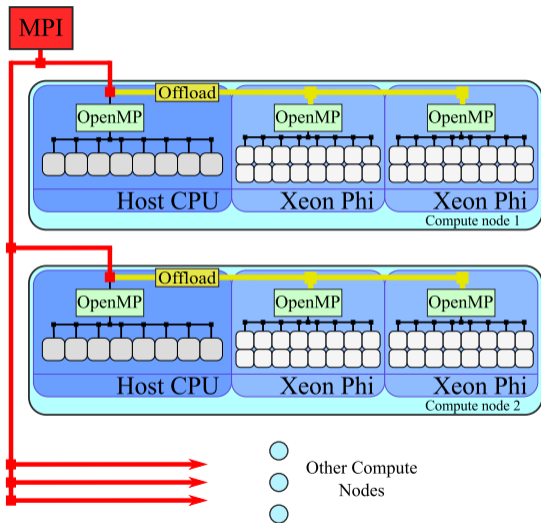
- ▶ Native model (standalone application/MPI process):



# HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

## Option 1: MPI+OpenMP with Offload.

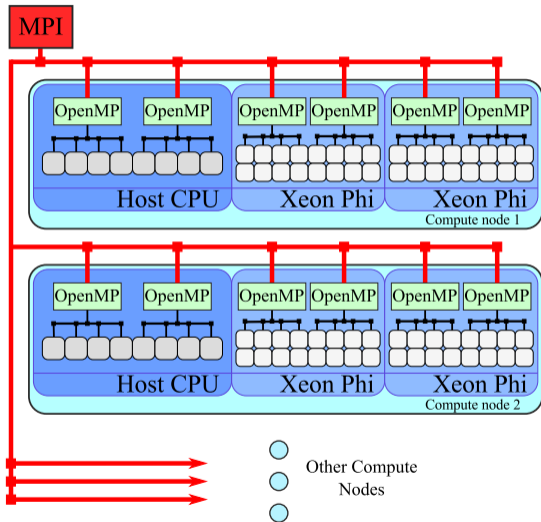
- ▶ MPI processes are multi-threaded with OpenMP.
- ▶ MPI runs only on CPUs.
- ▶ MPI processes offload to coprocessor(s).
- ▶ OpenMP in offload regions.



# HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

## Option 2: Symmetric hybrid MPI+OpenMP.

- ▶ MPI processes on hosts
- ▶ Native MPI processes on the coprocessor.
- ▶ Multi-threading with OpenMP.



# OFFLOAD OR NATIVE: HOW TO DECIDE

<b>Native</b>	<b>Offload</b>
$\leq 16$ GiB All parallel Complex data structures Any arithmetic intensity	$> 16$ GiB Parallel + serial phases Bitwise-copyable data (FLOPs/transfer) $\gg 1000$

Native = same code on CPU and MIC

Offload = must insert directives in code



## **EXPLICIT OFFLOAD (LEO)**

# EXPLICIT OFFLOAD: PRAGMA-BASED APPROACH

“Hello World” in the explicit offload model:

```
1 #include <stdio>
2 int main() {
3     printf("Hello World from host!\n");
4     #pragma offload target(mic)
5     {
6         printf("Hello World from coprocessor!\n"); fflush(stdout);
7     }
8     printf("Bye\n");
9 }
```

Application runs on the host, but some parts of code and data are moved (“offloaded”) to the coprocessor.

Detailed syntax in the [Intel C++ Compiler Reference](#).

# COMPILING AND RUNNING AN OFFLOAD APPLICATION

```
vega@lyra% icpc hello_offload.cc -o hello_offload
vega@lyra% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
```

- ▶ No additional arguments (for Intel compiler)
- ▶ Launch on host as a regular application
- ▶ Code inside of `#pragma offload` is offloaded automatically
- ▶ Console output on coprocessor buffered, mirrored to the host
- ▶ If no coprocessor available, default behavior is error; may be overridden to fall back to host



## **OFFLOADING FUNCTIONS AND DATA**

# OFFLOADING FUNCTIONS

```
1 __attribute__((target(mic))) void MyFunction() {  
2     // ... implement function as usual  
3 }  
4  
5 int main(int argc, char * argv[] ) {  
6     #pragma offload target(mic)  
7     {  
8         MyFunction();  
9     }  
10 }
```

- ▶ Functions used on coprocessor must be marked with the specifier `__attribute__((target(mic)))`
- ▶ Compiler produces a host version and a coprocessor version of such functions (to enable fall-back to host)

# OFFLOADING MULTIPLE FUNCTIONS

```
1 #pragma offload_attribute(push, target(mic))
2 void MyFunctionOne() {
3 // ... implement function as usual
4 }
5
6 void MyFunctionTwo() {
7 // ... implement function as usual
8 }
9 #pragma offload_attribute(pop)
```

- ▶ To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`

# OFFLOADING DATA: LOCAL SCALARS AND ARRAYS

```
1 void MyFunction() {  
2     const int N = 1000;  
3     int data[N];  
4     #pragma offload target(mic)  
5     {  
6         for (int i = 0; i < N; i++)  
7             data[i] = 0;  
8     }
```

- ▶ Scope-local scalars and known-size arrays offloaded automatically
- ▶ Data is copied from host to coprocessor at the start of offload
- ▶ Data is copied back from coprocessor to host at the end of offload
- ▶ Bitwise-copyable data only (arrays of basic types and scalars)  
C++ classes, etc. should use virtual-shared memory model

# OFFLOADING DATA: GLOBAL AND STATIC VARIABLES

```
1 int* __attribute__((target(mic))) data;
2
3 void MyFunction() {
4     static int __attribute__((target(mic))) N;
5     // ...
6 }
7
8 int main() {
9     // ...
10 }
```

- ▶ Global and static variables must be marked with the offload attribute
- ▶ `#pragma offload_attribute(push/pop)` may be used as well

# DATA MARSHALLING FOR DYNAMICALLY ALLOCATED DATA

```
1 double *p1=(double*)malloc(sizeof(double)*N);
2 double *p2=(double*)malloc(sizeof(double)*N);
3
4 #pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
5 {
6     // ... perform operations on p1[] and p2[]
7 }
```

- ▶ #pragma offload recognizes clauses in, out, inout and nocopy
- ▶ Data size (length), alignment, redirection, and other properties may be specified
- ▶ Marshalling is required for pointer-based data

# OPTIONAL OFFLOAD, FALL-BACK TO HOST

```
1 #pragma offload target(mic) optional
2 {
3     printf("Hello World! I have %d logical processors.\n",
4           sysconf(_SC_NPROCESSORS_ONLN )); fflush(stdout);
5 }
```

```
vega@lyra% icpc Offload-Fallback.cc -o Offload-Fallback
vega@lyra% ./Offload-Fallback
Hello World! I have 244 logical processors.
vega@lyra% sudo systemctl stop mpss # Disabling coprocessors
vega@lyra% ./Offload-Fallback
Hello World! I have 48 logical processors.
```



## **MULTIPLE COPROCESSORS WITH EXPLICIT OFFLOAD**

# MULTIPLE COPROCESSORS WITH EXPLICIT OFFLOAD

## ▶ Querying the number of coprocessors:

```
1  const int numDevices = _Offload_number_of_devices();  
2  printf("Number of available coprocessors: %d\n" , numDevices);
```

## ▶ Specifying offload target:

```
1  #pragma offload target(mic: 0)  
2  { /* ... */ }
```

## ▶ Query the device number from within Offload:

```
1  #pragma offload target(mic)  
2  {  
3      const int deviceNum = _Offload_get_device_number();  
4      printf("Hello from coprocessor %d!\n" , deviceNum);  
5  }
```

# MULTIPLE OFFLOADS USING HOST THREADS

```
1  const int nDevices = _Offload_number_of_devices();
2  #pragma omp parallel num_threads(nDevices)
3  {
4      const int i = omp_get_thread_num();
5      #pragma offload target(mic: i)
6          {
7              MyFunction(/*...*/);
8          }
9  }
10 }
```

- ▶ Up to 8 coprocessors, up to 56 host threads
- ▶ All offloads start simultaneously and block the respective thread

# DYNAMIC WORK DISTRIBUTION

```
1  const int nDevices = _Offload_number_of_devices();  
2  omp_set_num_threads(nDevices);  
3  #pragma omp parallel for schedule(dynamic, 1)  
4      for (int i = 0; i < nWorkItems; i++) {  
5          const int iDevice = omp_get_thread_num();  
6          #pragma offload target(mic: iDevice)  
7              {  
8                  MyFunction(i);  
9              }  
10     }
```

- ▶ Up to 8 coprocessors, up to 32 host threads
- ▶ nWorkItems are dynamically scheduled on nDevices



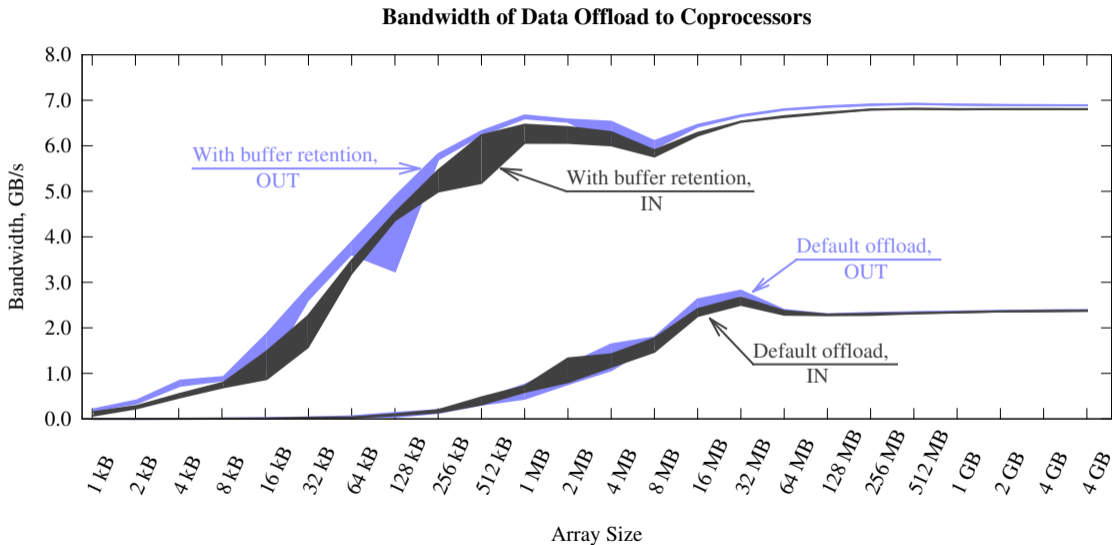
# MEMORY ALLOCATION CONTROL

# MEMORY RETENTION AND DATA PERSISTENCE ON COPROCESSOR

- ▶ By default, memory on coprocessor is allocated before, deallocated after offload
- ▶ Specifiers `alloc_if` and `free_if` allow to avoid allocation/deallocation
- ▶ Data transfer across the PCIe bus rate is  $\approx 7$  GB/s
- ▶ To allocate memory on the coprocessor – 0.5-2.0 GB/s

```
1 #pragma offload target(mic:0) in(p : length(N) alloc_if(1) free_if(0) )
2 { /* allocate memory for array p on coprocessor, do not deallocate */ }
3
4 #pragma offload target(mic:0) in(p : length(N) alloc_if(0) free_if(0) )
5 { /* re-use previously allocated memory buffer on coprocessor */ }
6
7 #pragma offload target(mic:0) in(p : length(0) alloc_if(0) free_if(0) )
8 { /* re-use previously transferred data on coprocessor */ }
9
10 #pragma offload target(mic:0) out(p : length(N) alloc_if(0) free_if(1) )
11 { /* re-use memory and deallocate at the end of offload */ }
```

# OFFLOAD LATENCY WITH AND WITHOUT MEMORY/DATA RETENTION



## PRECAUTIONS WITH PERSISTENT DATA

- ▶ Use explicit zero-based coprocessor number (e.g., `mic:0` as shown below)
- ▶ With multiple coprocessors, if target number is unspecified, any coprocessor can be used, which will result in runtime errors if persistent data cannot be found.

```
1 #pragma offload target(mic:0) in(p : length(N) alloc_if(1) free_if(0) )  
2 { /* allocate memory for array p on coprocessor, do not deallocate */ }
```

- ▶ Do not change the value of the host pointer to a persistent array: the runtime system finds the data on coprocessor using the host pointer value, not variable name.



# **OVERLAPPING COMMUNICATION AND COMPUTATION**

# ASYNCHRONOUS OFFLOAD

- ▶ By default, `#pragma offload` blocks until offload completes
- ▶ Use clause “signal” with any pointer to avoid blocking
- ▶ Use `#pragma offload_wait` to block where needed

```

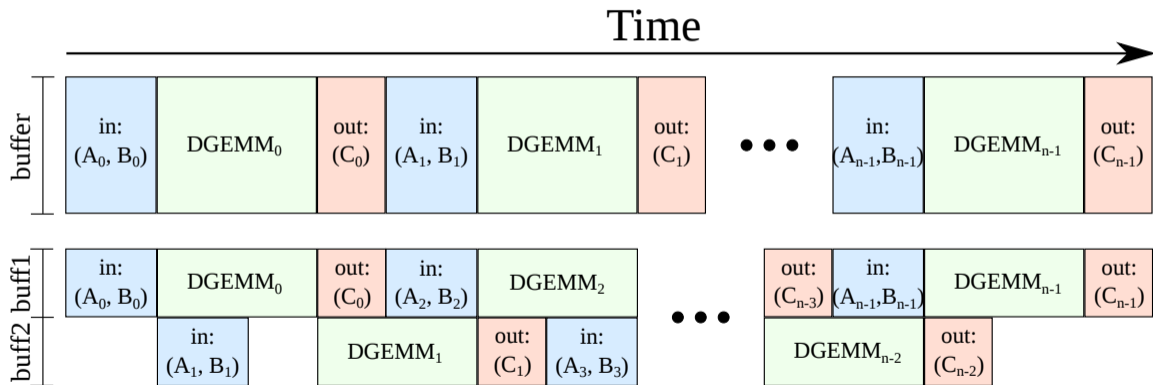
1 float* offload0 = &data[0]; // Any unique pointer value as signal
2 #pragma offload target(mic:0) signal(offload0) in(data : length(N))
3 { /* ... will not block code execution because of clause "signal" */ }
4
5 DoSomethingElse();
6
7 /* Now block until offload signalled by pointer "offload0" completes */
8 #pragma offload_wait target(mic:0) wait(offload0)

```

- ▶ Use the target number to avoid hanging

# OVERLAPPING COMMUNICATION AND COMPUTATION

Usage example: double buffering to mask communication latency



# DOUBLE BUFFERING WITH ASYNCHRONOUS OFFLOAD

```

1  for(int i = 1; i < nMatrices-1; i++) {
2      double* A_T = MatrixA[i]; // Dataset to send next; ...same for B and C
3
4      #pragma offload target(mic:0) signal(A_buff_C) \
5          in(A_buff_C: length(0) alloc_if(0) free_if(0)) ...(same for B and C)
6          { cblas_dgemm(..., A_buff_C, ...); } // Asynchronous offload (COMPUTATION)
7
8      // Send next data set, retrieve previous results (COMMUNICATION):
9      #pragma offload_transfer target(mic:0) in(A_T[0:n*n]: into (A_buff_T[0:n*n]))...
10     #pragma offload_transfer target(mic:0) out(C_buff_T[0:n*n]: into (C_T[0:n*n]))
11     // Wait for asynchronous offload (SYNCHRONIZATION):
12     #pragma offload_wait target(mic:0) wait(A_buff_C)
13
14     if(i%2==1) // Swap Buffers
15         { A_buff_T=A_buff2; A_buff_C =A_buff1; /* ...same for B and C */ }
16     else
17         { A_buff_T=A_buff1; A_buff_C =A_buff2; /* ...same for B and C */ }

```



## **ADDITIONAL OFFLOAD CONTROLS**

# TARGET-SPECIFIC CODE

- ▶ During MIC architecture compilation, preprocessor macro `__MIC__` is defined.
- ▶ Allows to fine-tune application performance or output where necessary

```
1 __attribute__((target(mic))) void MyFunction() {  
2 #ifdef __MIC__  
3     printf("I am running on a coprocessor.\n");  
4     const int tuningParameter = 16;  
5 #else  
6     printf("I am running on the host.\n");  
7     const int tuningParameter = 8;  
8 #endif  
9     // ... Proceed, using the variable tuningParameter  
10 }
```

# OFFLOAD DIAGNOSTICS

```
vega@lyra% export OFFLOAD_REPORT=2
vega@lyra% ./offload-application
Transferring some data to and from coprocessor...
Done. Bye!
[Offload] [MIC 0] [File]                offload-application.cc
[Offload] [MIC 0] [Line]                6
[Offload] [MIC 0] [CPU Time]            0.505982 (seconds)
[Offload] [MIC 0] [CPU->MIC Data]      1024 (bytes)
[Offload] [MIC 0] [MIC Time]           0.000409 (seconds)
[Offload] [MIC 0] [MIC->CPU Data]      1024 (bytes)
vega@lyra%
```

- ▶ Set environment variable `OFFLOAD_REPORT` to 1 or 2 for automatic collection and output of offload information.
- ▶ Unset or set `OFFLOAD_REPORT=0` to disable offload diagnostics

# OFFLOAD DEVICES, SPECIFYING AVAILABLE COPROCESSORS

- ▶ Specify coprocessors to use; For example (using 0 and 1),

```
vega@lyra% export OFFLOAD_DEVICES=0,1
```

- ▶ Disable Offloading

```
vega@lyra% export OFFLOAD_DEVICES=none
```

Disabling Offload is useful for debugging. For example;

```
vega@lyra% icpc Offload-Fallback.cc -o Offload-Fallback
vega@lyra% ./Offload-Fallback
Hello from offload on MIC with 244 logical processors.
vega@lyra% export OFFLOAD_DEVICES=none # Coprocessors disabled
vega@lyra% ./Offload-Fallback
Hello from offload on CPU with 48 logical processors.
```

## ENVIRONMENT VARIABLE FORWARDING WITH OFFLOAD

- ▶ By default, all host environment variables on the host will be copied to the coprocessor when offload starts.
- ▶ In order to have different values for an environment variable on host and coprocessor, set `MIC_ENV_PREFIX`
- ▶ The prefix is dropped when variables are copied to coprocessor

```
vega@lyra% # This sets the value of OMP_NUM_THREADS on the host:
vega@lyra% export OMP_NUM_THREADS=48
vega@lyra%
vega@lyra% # This enables special rules for variable copying:
vega@lyra% export MIC_ENV_PREFIX=XEONPHI
vega@lyra%
vega@lyra% # This sets the value of OMP_NUM_THREADS on the coprocessor:
vega@lyra% export XEONPHI_OMP_NUM_THREADS=240
```



## **OFFLOAD IN OPENMP 4.0**

## OPENMP 4.0 TARGET OFFLOAD

- ▶ Another API for offload: `#pragma omp target`
- ▶ Part of the OpenMP 4.0 standard
- ▶ Designed as portable solution (coprocessors, GPGPUs)
- ▶ On Xeon Phi, uses the same back-end as `#pragma offload`

```
1 #pragma omp target
2 {
3 #pragma omp parallel for
4     for(int i=0; i<size; i++)
5         data[i] = 0;
6 }
```

Application runs on the host, but some parts of code and data are moved (“offloaded”) the coprocessor. Scope-local scalars and stack arrays offloaded automatically.

# CLAUSES OF PRAGMA OMP TARGET

```
1 #pragma omp target [clause[, clause[, ...]]]
```

- ▶ `device(int)` – offload to a specific device (coprocessor)
- ▶ `map([type:] variables)` – create data environment. `type` is `to`, `from`, `tofrom` or `alloc`
- ▶ `if(expr)` – optional offload

Link to [reference manual](#).

# OPENMP 4.0 TARGET DATA MAPPING

Use `#pragma omp target data` to create a device data environment. This allows to keep persistent data on coprocessor. Example:

```
1 #pragma omp target data map(from:data)
2 {
3 #pragma omp target
4 #pragma omp parallel for
5     for(int i=0; i<size; i++) data[i] = 0;
6
7 #pragma omp target
8 #pragma omp parallel for
9     for(int i=0; i<size; i++) data[i] += 1;
10 }
```

data array copied back from coprocessor only once at the end.  
Link to [reference manual](#).

# MOVEMENT OF PERSISTENT DATA

Use `#pragma omp target update` to force data movement within the data environment. Example:

```
1  #pragma omp target data map(from:data)
2  {
3  #pragma omp target
4    { ... }
5
6  #pragma omp target update from(data)
7
8  #pragma omp target
9    { ... }
10 }
```

data array copied from coprocessor between offloads, and at the end.

Link to [reference manual](#).

## OFFLOADING FUNCTIONS WITH #pragma omp target

Use #pragma omp declare target on functions that may be offloaded (similar to \_\_attribute\_\_((target(mic)))). Example:

```
1  #pragma omp declare target
2  void myinit(int* data, int size){
3  #pragma omp parallel for
4      for(int i=0; i<size; i++) data[i] = 0;
5  }
6  #pragma omp end declare target
7
8  int main(int argv, char** argc){
9  ...
10 #pragma omp target map(tofrom:data) map(to:size)
11     myinit(data, size);
12 }
```

Link to [reference manual](#).

## #pragma offload target **VS.** #pragma omp target

1. Different interfaces to the same offload library back-end
2. #pragma offload target is Intel-specific, #pragma omp target is part of a cross-platform standard (although, as today, cross-platform support is not widespread).
3. #pragma offload target allows data/memory persistence outside of the scope of a pragma, #pragma omp target – only within the lexically structured scope (correct us if we're wrong).
4. #pragma offload is a more flexible model and will continue to be supported (see Intel's [communication](#)).

Additional information: [webinar](#).



## **SHARED VIRTUAL MEMORY OFFLOAD MODEL**

# SHARED VIRTUAL MEMORY MODEL

```
1  _Cilk_shared int arr[N]; // This is a virtual-shared array
2
3  _Cilk_shared void Compute() { // This function may be offloaded
4      // ... function uses array arr[]
5  }
6
7  int main() {
8      // arr[] can be initialized on the host
9      _Cilk_offload Compute(); // and used on coprocessor
10     // and the values are returned to the host
11 }
```

- ▶ Alternative to Explicit Offload
- ▶ Data synced from host to coprocessor before the start of offload
- ▶ Data synced from coprocessor to host at the end of offload

# SHARED VIRTUAL MEMORY MODEL

```
1  int* _Cilk_shared data; // Pointer to a virtual-shared array
2
3  int main() {
4      // Working with pointer-based data is illustrated below:
5      data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));
6      _Offload_shared_free(data);
7  }
```

- ▶ Addresses of virtual-shared pointers identical on host and coprocessors
- ▶ Synchronized before and after offload, with page granularity

## REVIEW AND WHAT'S NEXT

- ▶ Native programming = treat coprocessor as a processor
- ▶ Offload = launched on host, some functions on coprocessor
- ▶ Explicit offload – `#pragma offload` or `#pragma omp target` – allows data marshalling; for bitwise-copyable data
- ▶ Shared virtual memory – `_Cilk_shared/_Cilk_offload` – automatic coherence; for complex objects

Next session: expressing data parallelism, vectorization.



**LEARN MORE**

# HOW SERIES: KNIGHTS LANDING

HOW SERIES "KNIGHTS LANDING":

PROGRAMMING AND OPTIMIZATION FOR  
INTEL XEON PHI X200 FAMILY

Free 2-hour video course



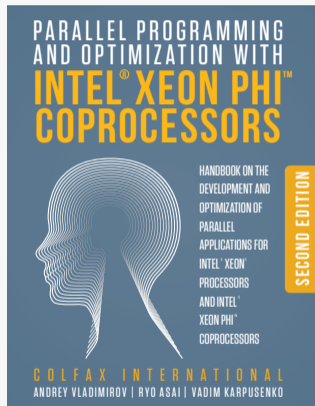
[colfaxresearch.com/how-knl/](http://colfaxresearch.com/how-knl/)

ISBN: 978-0-9885234-0-1 (508 pages, Electronic or Print)

Parallel Programming  
and Optimization with  
Intel® Xeon Phi™  
Coproprocessors

Handbook on the Development and  
Optimization of Parallel Applications  
for Intel® Xeon® Processors  
and Intel® Xeon Phi™ Coprocessors

© Colfax International, 2015



<http://xeonphi.com/book>

**COLFAX RESEARCH**
Log In/Out or Register

READ WATCH LEARN CONNECT JOIN

To search, type and hit enter

**Popular**

**The Hands-On Tutorials (HOT) webinars: details on efficient programming for Intel architecture**

**The Hands-On Workshop (HOW) Series**

**Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation**

**Parallel Programming Book**

Introduction to parallel programming, deep discussion of optimization techniques, exercises.

**Research and Educational Publications**

**Introduction to Intel DAAL, Part 1 of 3: Polynomial Regression with Batch Mode Computation**

**Optimization Techniques for the Intel MIC Architecture, Part 3 of 3: False Sharing and Padding**

**Software Developer's Introduction to the HGST Ultrastar Archive H700 SMR Drives**

**Optimization Techniques for the Intel MIC Architecture, Part 2 of 3: Strip-Mining for Vectorization**

**Optimization Techniques for the Intel MIC Architecture, Part 1 of 3: Multi-Threading and Parallel Reduction**

**Performance to Power and Performance to Cost Ratios with Intel Xeon Phi Coprocessors (and why ix Acceleration May Be Enough)**

**Featured Video**

See Research material re-creation in a streaming mode

g/1702/2e4mawxw/ck.../p/7018

## Consulting

Share

Colfax offers consulting services for enterprises, research help you to:

- Optimize your existing application to take advantage of parallelism, from vectors to cores to clusters and
- Future-proof your application for upcoming innovations
- Accelerate your application using coprocessor tech
- Investigate the potential system configurations that satisfy your cost, power, performance requirements.
- Take a clean slate to develop a novel architecture to realize your computing pro

**Episode 2.1 — Purpose of the MIC architecture**

**Software Developer's Introduction to the HGST Ultrastar Archive H700 SMR Drives**

Introduction by Intel and HGST

In this paper we will discuss the new HGST Ultrastar Archive H700 SMR drives, their features and how they compare to other high capacity hard drives. These drives are well suited for high volume archival applications in a wide range of applications. The paper is available for download.

The HGST Ultrastar Archive H700 SMR drives are well suited for high volume archival applications in a wide range of applications. The paper is available for download.

**Fluid Dynamics with Fortran on Intel Xeon Phi coprocessors**

Introduction by Intel and Fortran

In this demonstration, a Fortran program is executed on a cluster of Intel Xeon Phi coprocessors. The program simulates the flow of a fluid around a cylinder. The results are visualized using a 3D visualization tool. The program is available for download.

**Configuration and Benchmarks of Peer-to-Peer Communication over Gigabit Ethernet and InfiniBand in a Cluster with Intel Xeon Phi Coprocessors**

Introduction by Intel and Mellanox

In this paper we will discuss the configuration and benchmarks of peer-to-peer communication over Gigabit Ethernet and InfiniBand in a cluster with Intel Xeon Phi coprocessors. The results are visualized using a 3D visualization tool. The program is available for download.

**Interview with James Reinders: future of Intel MIC architecture, parallel programming, education**

Introduction by Intel and James Reinders

In this interview, we will discuss the future of Intel MIC architecture, parallel programming, and education. The interview is available for download.

http://colfaxresearch.com/

colfaxresearch.com/how-16-10
LEARN MORE
© Colfax International, 2013–2016