Programming and Optimization for Intel® Architecture

The Hands-On Workshop (HOW) Series

Andrey Vladimirov, PhD, and Ryo Asai
Colfax International — @colfaxintl

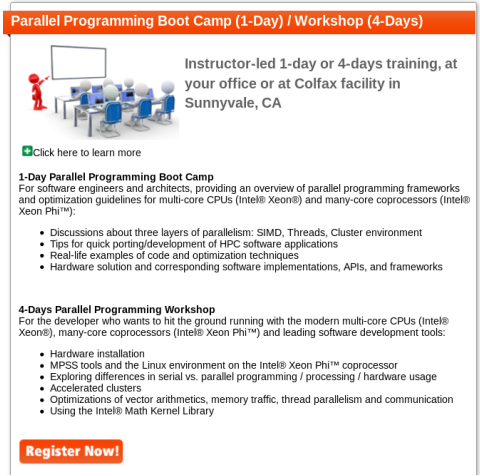
March 2016 , Rev. 02a

 © Colfax International, 2013–2016

About This Document

This document represents the materials of a Web-based training “Programming and Optimization with Intel Architecture“ developed and run by Colfax International.

© Colfax International, 2013-2015



colfaxresearch.com/how-series

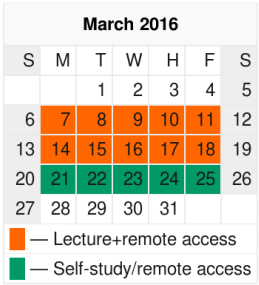
Disclaimer

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.

© Colfax International, 2013–2016

Course Roadmap

1. Why Intel Parallel Architectures?
   - Parallelism and specialization – March 7
   - Programming model continuity – March 7
2. Programming models for Xeon Phi coprocessors
   - Native programming – March 7
   - Offload programming – March 8
3. Expressing Parallelism
   - Introduction to vectorization – March 9
   - Crash-course on OpenMP – March 10
4. Optimization – intro on March 11
   - Vectorization tuning – March 14
   - Multi-threading – March 15, 16
   - Memory traffic – March 17
5. Tools: MKL and VTune – March 18



 © Colfax International, 2013–2016

HOW Online

Course page: colfaxresearch.com/how-series

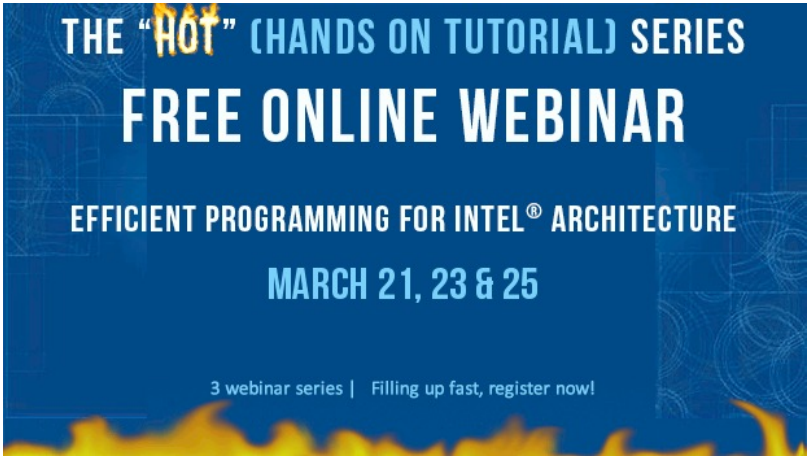
- Slides (including this one), code downloads
- Video of recorded sessions
- Chat (during webinars or offline)

Additional resources:

- More workshops like this one: colfaxresearch.com/how-series
- Video courses: colfaxresearch.com/video-courses
- Intel Many Integrated Core Architecture Forum

© Colfax International, 2013–2016

HOT Series



colfaxresearch.com/hot-16-03/

 © Colfax International, 2013–2016

§2. Programming Coprocessors

 © Colfax International, 2013–2016

Offload and Native Models

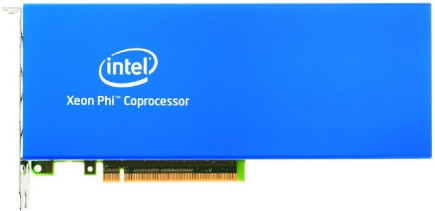
© Colfax International, 2013–2016

Computing Platforms

Intel Xeon Processor

Current: Haswell
Upcoming: Broadwell

Intel Xeon Phi Corocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

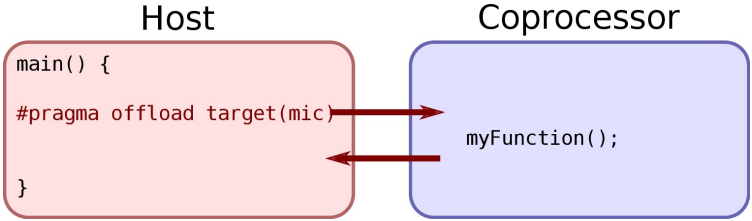
Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

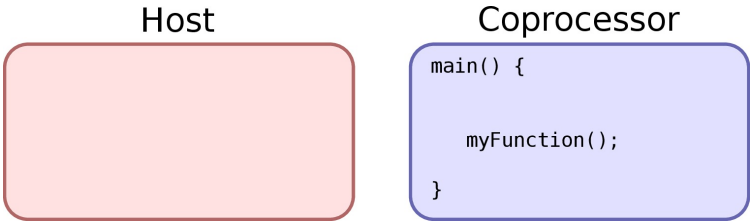
 © Colfax International, 2013–2016

Offload and Native Models

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



- Native model (standalone application/MPI process):

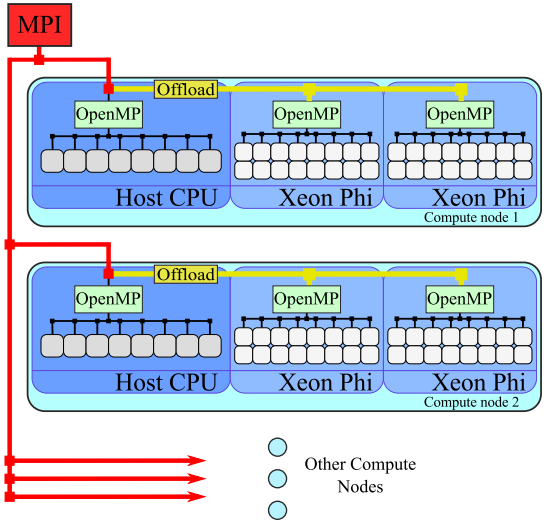


 © Colfax International, 2013–2016

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.

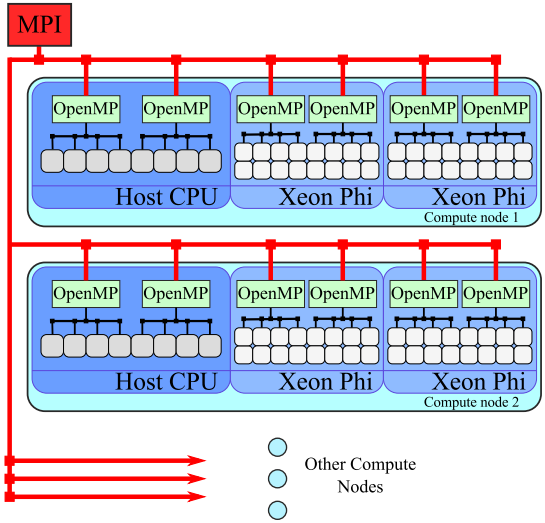


 © Colfax International, 2013–2016

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.



 © Colfax International, 2013–2016

Explicit Offload (LEO)

© Colfax International, 2013–2016

Explicit Offload: Pragma-based approach

"Hello World" in the explicit offload model:

```
1 #include <stdio.h>
2 int main(int argc, char * argv[]) {
3     printf("Hello World from host!\n");
4 #pragma offload target(mic)
5     {
6         printf("Hello World from coprocessor!\n"); fflush(0);
7     }
8     printf("Bye\n");
9 }
```

Application runs on the host, but some parts of code and date are moved ("offloaded") the coprocessor.
Detailed syntax in the Intel C++ Compiler Reference.

 © Colfax International, 2013–2016

Compiling and Running an Offload Application

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

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- Code inside of `#pragma offload` is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside `#pragma offload` runs on the host system

 © Colfax International, 2013–2016

To Offload or Not To Offload

© Colfax International, 2013–2016

Offload Game 1: Find the Maximum

© Colfax International, 2013–2016

Offload Game 2: Build Words

© Colfax International, 2013–2016

PCIe Bandwidth Considerations

Does offload pay off?

- PCIe bandwidth ≈7 GB/s
- 1 TFLOP/s in double precision is 8 TB/s

Conclusion: offload if algorithm performs $\gg 1000$ operations per transferred word.

Good for offload:	**Bad for offload:**
Strong scaling (e.g., $O(n^2)$)	Weak scaling (e.g., $O(n)$, $O(n \log n)$)
Large problems ($n \gg 1000$)	Small problems ($n \lesssim 1000$)

© Colfax International, 2013–2016

Hands-On Part

We are now switching to the practical part. Slides in the rest of this section may help you with self-study.

 © Colfax International, 2013–2016

Offloading Functions and Data

© Colfax International, 2013–2016

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 coprocesor version of such functions (to enable fall-back to host)

 © Colfax International, 2013–2016

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)`

 © Colfax International, 2013–2016

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

© Colfax International, 2013–2016

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

© Colfax International, 2013–2016

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

© Colfax International, 2013–2016

Optional Offload, Fall-Back to Host

```
1 #pragma offload target(mic) optional
2 {
3   printf("Hello World! I have %d logical cores.\n",
4   sysconf(_SC_NPROCESSORS_ONLN )); fflush(0);
5 }
```

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

 © Colfax International, 2013–2016

Multiple Coprocessors with Explicit Offload

© Colfax International, 2013–2016

Multiple Coprocessors with Explicit Offload

- Querying the number of coprocessors:

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

- Specifying offload target:

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

- Query the device number from within Offload:

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

 © Colfax International, 2013–2016

Multiple Blocking Offloads Using Host Threads (Explicit Offload)

```
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

 © Colfax International, 2013–2016

Blocking Explicit Offloads Using Threads: Dynamic Work Distribution Across Coprocessors

```
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`

 © Colfax International, 2013–2016

Memory Allocation Control

© Colfax International, 2013–2016

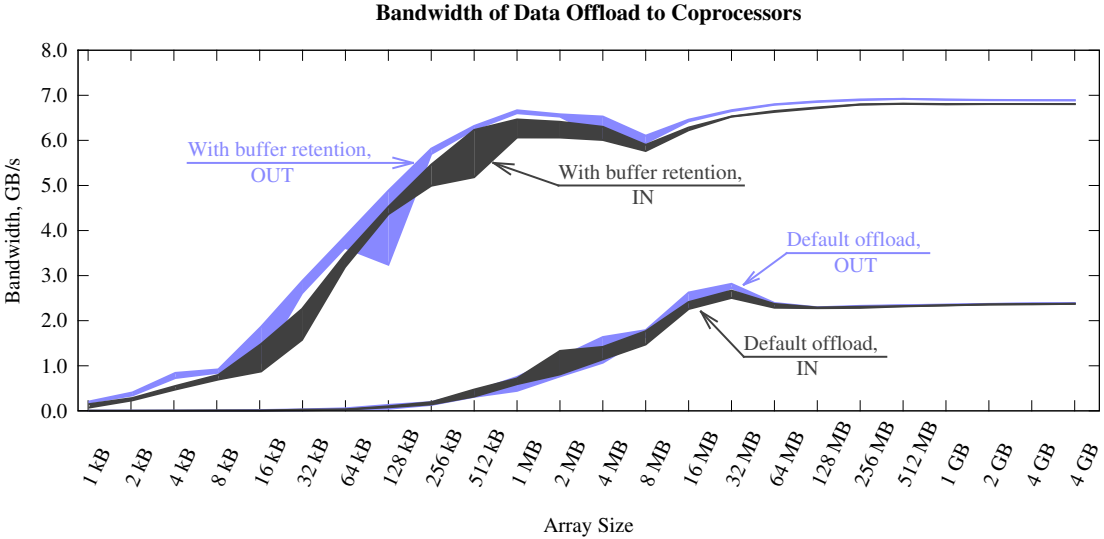
Memory retention and data persistence on coprocessor

- 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 ≈ 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 */ }
```

 © Colfax International, 2013–2016

Offload Latency With and Without Memory/Data Retention



 © Colfax International, 2013–2016

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.

 © Colfax International, 2013–2016

Overlapping Communication and Computation

© Colfax International, 2013–2016

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

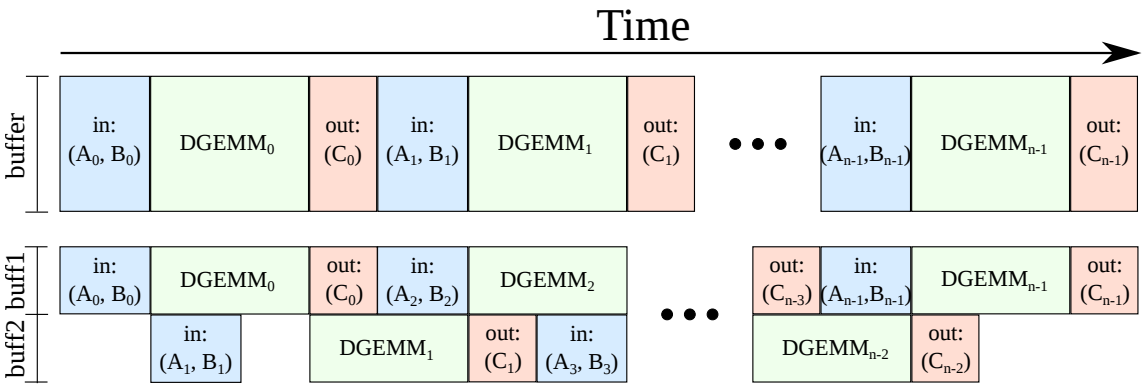
```
1 float* offload0 = &data[0]; // Any unique pointer value as signal
2 #pragma offload target(mic:0) signal(offload0) in(data : length(N))
3 { /* ... will not block code execution because of clause "signal" */ }
4
5 DoSomethingElse();
6
7 /* Now block until offload signalled by pointer "offload0"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

- Use the target number to avoid hanging

 © Colfax International, 2013–2016

Overlapping Communication and Computation

Usage example: double buffering to mask communication latency



 © Colfax International, 2013–2016

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 */ }
```


© Colfax International, 2013–2016

Additional Offload Controls

© Colfax International, 2013–2016

Target-Specific Code

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

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

 © Colfax International, 2013–2016

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.cpp
[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

© Colfax International, 2013–2016

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 cores.
vega@lyra% export OFFLOAD_DEVICES=none # Coprocessors disabled
vega@lyra% ./Offload-Fallback
Hello from offload on CPU with 48 logical cores.
```


© Colfax International, 2013–2016

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
```

 © Colfax International, 2013–2016

Offload in OpenMP 4.0

 © Colfax International, 2013–2016

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.

 © Colfax International, 2013–2016

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.

 © Colfax International, 2013–2016

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:

```
#pragma omp target data map(from:data)
  {
#pragma omp target
#pragma omp parallel for
    for(int i=0; i<size; i++) data[i] = 0;

#pragma omp target
#pragma omp parallel for
    for(int i=0; i<size; i++) data[i] += 1;
  }
```

`data` array copied back from coprocessor only once at the end.
Link to reference manual.

 © Colfax International, 2013–2016

Movement of Persistent Data

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

```
#pragma omp target data map(from:data)
{
#pragma omp target
  { ... }

#pragma omp target update from(data)

#pragma omp target
  { ... }
}
```

`data` array copied from coprocessor between offloads, and at the end. Link to reference manual.

 © Colfax International, 2013–2016

Offloading functions wtih `#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.

 © Colfax International, 2013–2016

`#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 of 2015, 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 (may change in Parallel Studio 2016).
4. `#pragma offload` is a more flexible model and will continue to be supported (see Intel's communication).

Additional information: webinar.

 © Colfax International, 2013–2016

Shared Virtual Memory Offload Model

© Colfax International, 2013–2016

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

© Colfax International, 2013–2016

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

 © Colfax International, 2013–2016

Review and What's Next

- 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
- Offload = application is launched on host, some functions run on coprocessor

Next session: expressing data parallelism, vectorization.

© Colfax International, 2013–2016