COLFAX
Customized Solutions

Programming and Optimization for Intel® Architecture

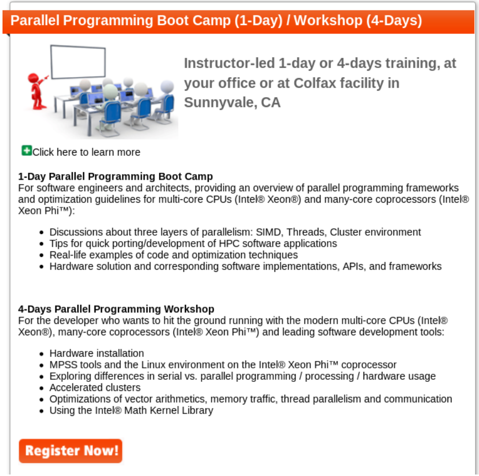
The Hands-On Workshop (HOW) Series

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

April 2016 , Rev. 02b

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.



Parallel Programming Boot Camp (1-Day) / Workshop (4-Days)

Instructor-led 1-day or 4-days training, at your office or at Colfax facility in Sunnyvale, CA

Click here to learn more

1-Day Parallel Programming Boot Camp
For software engineers and architects, providing an overview of parallel programming frameworks and optimization guidelines for multi-core CPUs (Intel® Xeon®) and many-core coprocessors (Intel® Xeon Phi™):

- Discussions about three layers of parallelism: SIMD, Threads, Cluster environment
- Tips for quick porting/development of HPC software applications
- Real-life examples of code and optimization techniques
- Hardware solution and corresponding software implementations, APIs, and frameworks

4-Days Parallel Programming Workshop
For the developer who wants to hit the ground running with the modern multi-core CPUs (Intel® Xeon®), many-core coprocessors (Intel® Xeon Phi™) and leading software development tools:

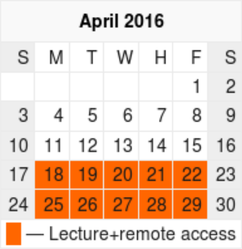
- Hardware installation
- MPSS tools and the Linux environment on the Intel® Xeon Phi™ coprocessor
- Exploring differences in serial vs. parallel programming / processing / hardware usage
- Accelerated clusters
- Optimizations of vector arithmetics, memory traffic, thread parallelism and communication
- Using the Intel® Math Kernel Library

colfaxresearch.com/how-series

Disclaimer

Course Roadmap

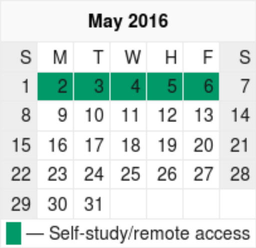
1. Why Intel Parallel Architectures?
   - Parallelism and specialization – April 18
   - Programming model continuity – April 18
2. Programming models for Xeon Phi coprocessors
   - Native programming – April 18
   - Offload programming – April 19
3. Expressing Parallelism
   - Introduction to vectorization – April 20
   - Crash-course on OpenMP – April 21
4. Optimization – intro on April 22
   - Vectorization tuning – April 25
   - Multi-threading – April 26, 27
   - Memory traffic – April 28
5. Distributed Computing: MPI – April 29



April 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

🟧 — Lecture+remote access



May 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				

🟩 — Self-study/remote access

HOW Online

Course page: colfaxresearch.com/how-16-04

- 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

Developer’s Guide to Knights Landing

colfaxresearch.com/knl-webinar/

§2. Programming Coprocessors

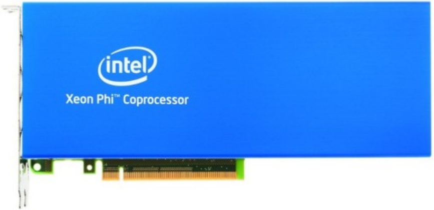
Offload and Native Models

Computing Platforms

Intel Xeon
Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi
Corocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi
Processor, 2nd generation*

* socket and coprocessor versions

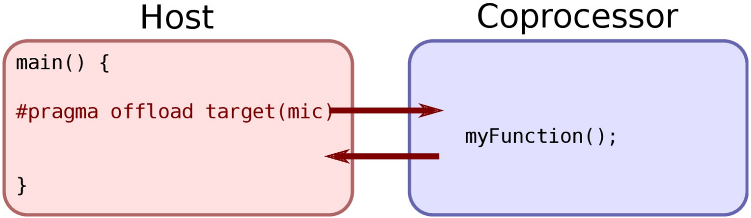
Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Offload and Native Models

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



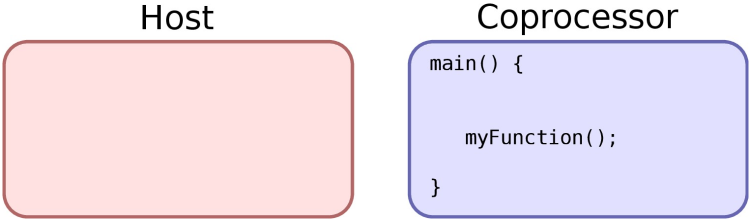
Host

```
main() {

#pragma offload target(mic)

}
```

Coprocessor

```
myFunction();
```

- Native model (standalone application/MPI process):



Host

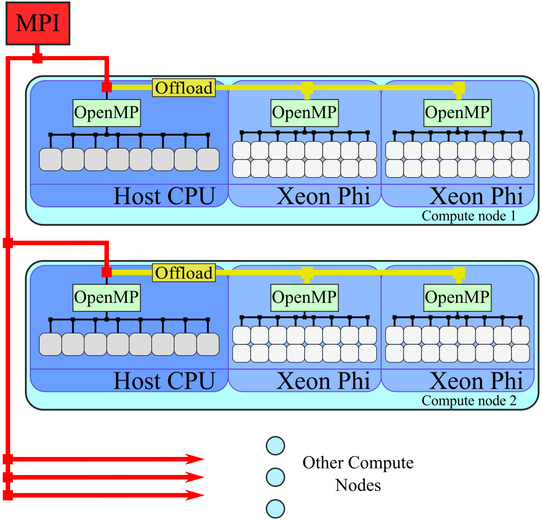
Coprocessor

```
main() {

    myFunction();

}
```

Heterogeneous Distributed Computing with Xeon Phi

Option 1: MPI+OpenMP with Offload.

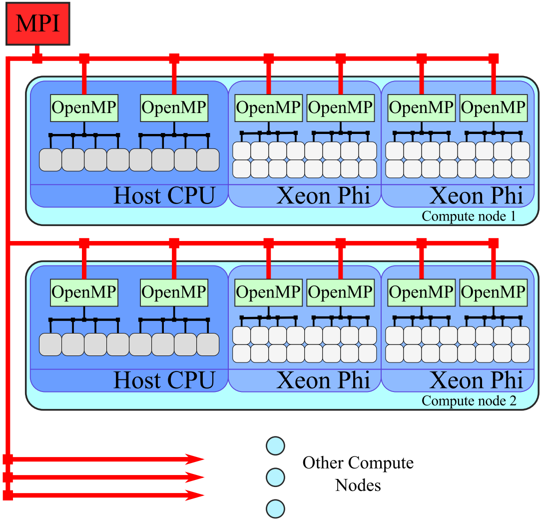
- MPI processes are multi-threaded with OpenMP.
- MPI runs only on CPUs.
- MPI processes offload to coprocessor(s).
- OpenMP in offload regions.



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.



Explicit Offload (LEO)

Explicit Offload: Pragma-based approach

“Hello World” in the explicit offload model:

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

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.

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

Offloading Functions and Data

Offloading Functions

```
__attribute__((target(mic))) void MyFunction() {
    // ... implement function as usual
}

int main(int argc, char * argv[] ) {
#pragma offload target(mic)
    {
        MyFunction();
    }
}
```

- 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)

Offloading Multiple Functions

```
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
// ... implement function as usual
}

void MyFunctionTwo() {
// ... implement function as usual
}
#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

```
void MyFunction() {
    const int N = 1000;
    int data[N];
#pragma offload target(mic)
    {
        for (int i = 0; i < N; i++)
          data[i] = 0;
    }
```

- 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

```
int* __attribute__((target(mic))) data;

void MyFunction() {
    static int __attribute__((target(mic))) N;
    // ...
}

int main() {
    // ...
}
```

- 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

```
double *p1=(double*)malloc(sizeof(double)*N);
double *p2=(double*)malloc(sizeof(double)*N);

#pragma offload target(mic) in(p1 : length(N)) out(p2 : length(N))
{
    // ... perform operations on p1[] and p2[]
}
```

- `#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

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

```
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.
```

Multiple Coprocessors with Explicit Offload

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);
}
```

Multiple Blocking Offloads Using Host Threads (Explicit Offload)

```
const int nDevices = _Offload_number_of_devices();
#pragma omp parallel num_threads(nDevices)
  {
    const int i = omp_get_thread_num();
#pragma offload target(mic: i)
      {
        MyFunction(/*...*/);
      }
  }
}
```

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

Blocking Explicit Offloads Using Threads: Dynamic Work Distribution Across Coprocessors

```
const int nDevices = _Offload_number_of_devices();
omp_set_num_threads(nDevices);
#pragma omp parallel for schedule(dynamic, 1)
  for (int i = 0; i < nWorkItems; i++) {
    const int iDevice = omp_get_thread_num();
#pragma offload target(mic: iDevice)
    {
      MyFunction(i);
    }
  }
```

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

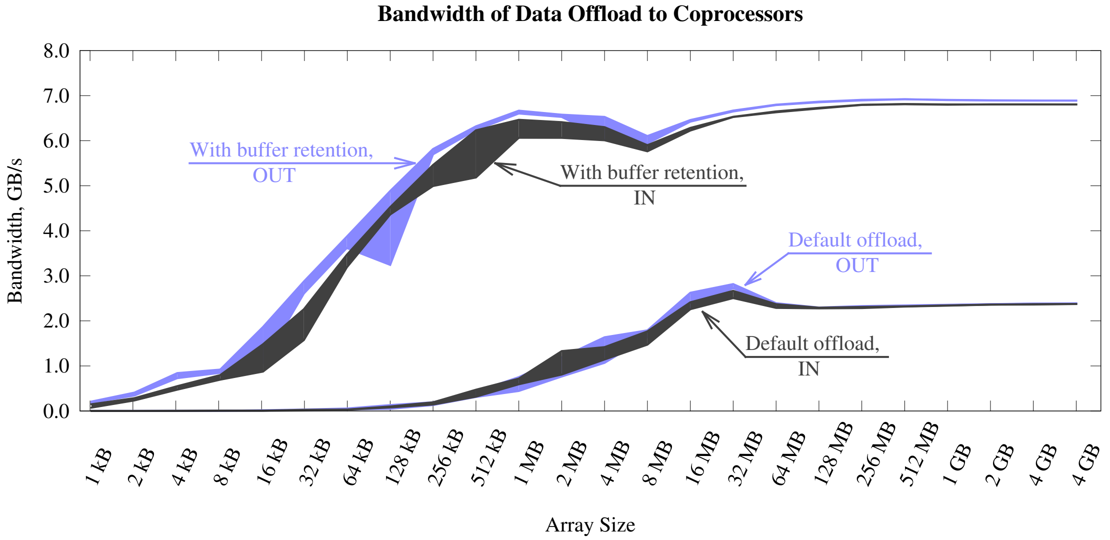
Memory Allocation Control

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 ≈ 7 GB/s
- To allocate memory on the coprocessor – 0.5-2.0 GB/s

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

#pragma offload target(mic:0) in(p : length(N) alloc_if(0) free_if(0) )
{ /* re-use previously allocated memory buffer on coprocessor */ }

#pragma offload target(mic:0) in(p : length(0) alloc_if(0) free_if(0) )
{ /* re-use previously transferred data on coprocessor */ }

#pragma offload target(mic:0) out(p : length(N) alloc_if(0) free_if(1) )
{ /* 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.

```
#pragma offload target(mic:0) in(p : length(N) alloc_if(1) free_if(0) )
{ /* 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

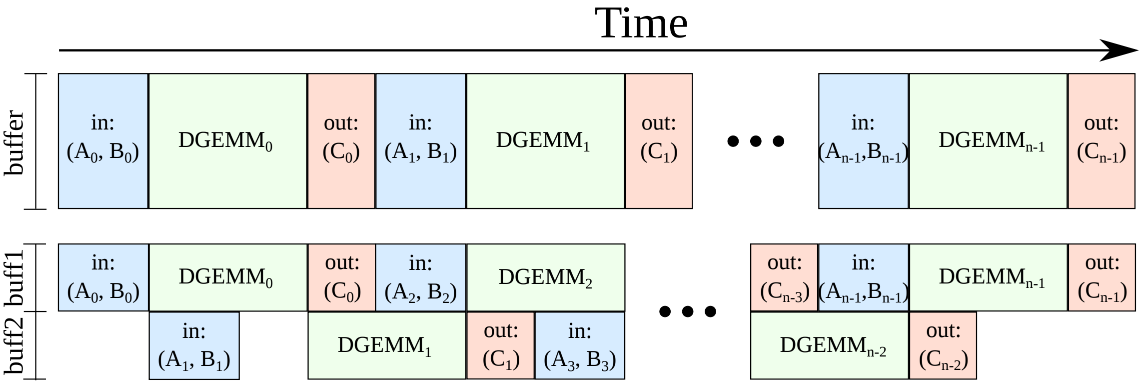
- By default, `#pragma offload` blocks until offload completes
- Use clause “`signal`” with any pointer to avoid blocking
- Use `#pragma offload_wait` to block where needed

```
float* offload0 = &data[0]; // Any unique pointer value as signal
#pragma offload target(mic:0) signal(offload0) in(data : length(N))
{ /* ... will not block code execution because of clause "signal" */ }

DoSomethingElse();

/* Now block until offload signalled by pointer "offload0" completes */
#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

```
for(int i = 1; i < nMatrices-1; i++) {
  double* A_T = MatrixA[i]; // Dataset to send next; ...same for B and C

#pragma offload target(mic:0)  signal(A_buff_C)  \
  in(A_buff_C: length(0) alloc_if(0) free_if(0)) ...(same for B and C)
  { cblas_dgemm(..., A_buff_C, ...); } // Asynchronous offload (COMPUTATION)

// Send next data set, retrieve previous results (COMMUNICATION):
#pragma offload_transfer target(mic:0) in(A_T[0:n*n]: into (A_buff_T[0:n*n]))...
#pragma offload_transfer target(mic:0) out(C_buff_T[0:n*n]: into (C_T[0:n*n]))
// Wait for asynchronous offload (SYNCHRONIZATION):
#pragma offload_wait target(mic:0) wait(A_buff_C)

if(i%2==1) // Swap Buffers
  { A_buff_T=A_buff2; A_buff_C =A_buff1; /* ...same for B and C */ }
else
  { 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

```
__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
}
```

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

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.
```

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`

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

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

```
#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:

```
#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.

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.

Offloading functions wtih `#pragma omp target`

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

```
#pragma omp declare target
void myinit(int* data, int size){
#pragma omp parallel for
  for(int i=0; i<size; i++) data[i] = 0;
}
#pragma omp end declare target

int main(int argv, char** argc){
...
#pragma omp target map(tofrom:data) map(to:size)
    myinit(data, size);
}
```

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 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.

Shared Virtual Memory Offload Model

Shared Virtual Memory Model

```
_Cilk_shared int arr[N]; // This is a virtual-shared array

_Cilk_shared void Compute() { // This function may be offloaded
    // ... function uses array arr[]
}

int main() {
    // arr[] can be initialized on the host
    _Cilk_offload Compute(); // and used on coprocessor
    // and the values are returned to the host
}
```

- 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

```
int* _Cilk_shared data; // Pointer to a virtual-shared array

int main() {
    // Working with pointer-based data is illustrated below:
    data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));
    _Offload_shared_free(data);
}
```

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

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.