PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

The Hands-On Workshop (HOW) Series

Session 2

Colfax International — *colfaxresearch.com*

September 2016

WELCOME

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.

COURSE ROADMAP

- HOW to Program Intel Architecture
  - 01. Parallelism, specialization, guided tour – Sep 26
  - 02. Programming Intel Xeon Phi (KNC, KNL) – Sep 27
- HOW to Express Parallelism
  - 03. Automatic vectorization – Sep 28
  - 04. Multi-threading with OpenMP – Sep 29
- HOW to Get Performance
  - 05. Comprehensive demo – Sep 30
  - 06. Scalar & vectorization tuning – Oct 3
  - 07. Multi-threading: common issues – Oct 4
  - 08. Multi-threading: memory aspect – Oct 5
  - 09. Memory traffic – Oct 6
- HOW to Scale
  - 10. Distributed Computing: MPI – Oct 7

September 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	

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					

— Webinar+remote access

HOW ONLINE

Course page: colfaxresearch.com/how-16-09

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

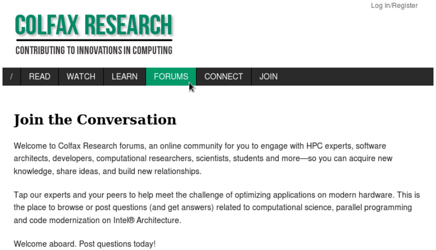
Additional resources:

- ▹ More workshops like this one: colfaxresearch.com/training
- ▹ Video courses: colfaxresearch.com/video-courses

GET YOUR QUESTIONS ANSWERED

Chat (current):
colfaxresearch.com/how-16-09

Forums (technical):
colfaxresearch.com/discussion

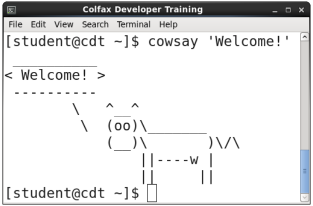


Email (organizational):

`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

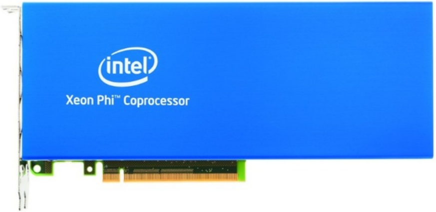


- Not among the 96? Stay tuned: follow along with instructor, use own system, or wait for a seat
- Use it or lose it: if you do not log in for a while, remote access token goes to next student on the list

§2. PROGRAMMING COPROCESSORS

OFFLOAD AND NATIVE MODELS

COMPUTING PLATFORMS

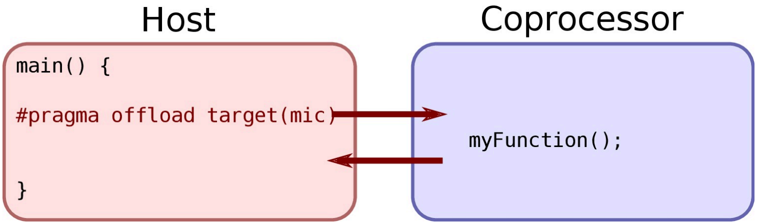
Intel Xeon Processor	Intel Xeon Phi Coprocessor, 1st generation	Intel Xeon Phi Processor, 2nd generation*
		
		* socket and coprocessor versions
Current: Broadwell Upcoming: Skylake	Knights Corner (KNC)	Knights Landing (KNL)

Multi-Core Architecture

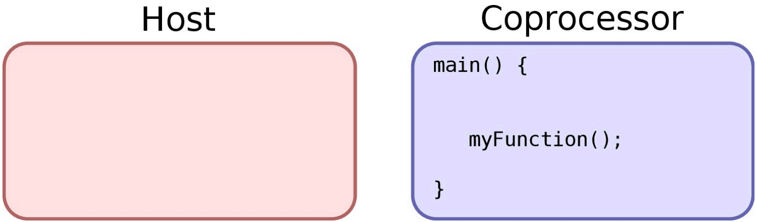
Intel Many Integrated Core (MIC) Architecture

OFFLOAD AND NATIVE MODELS

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



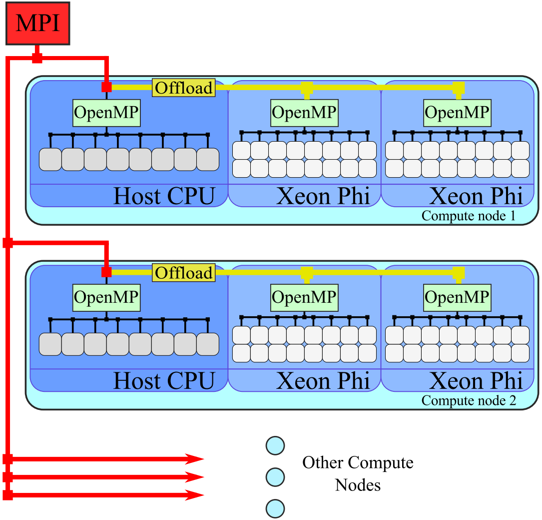
▷ Native model (standalone application/MPI process):



HETEROGENEOUS DISTRIBUTED COMPUTING WITH XEON PHI

Option 1: MPI+OpenMP with Offload.

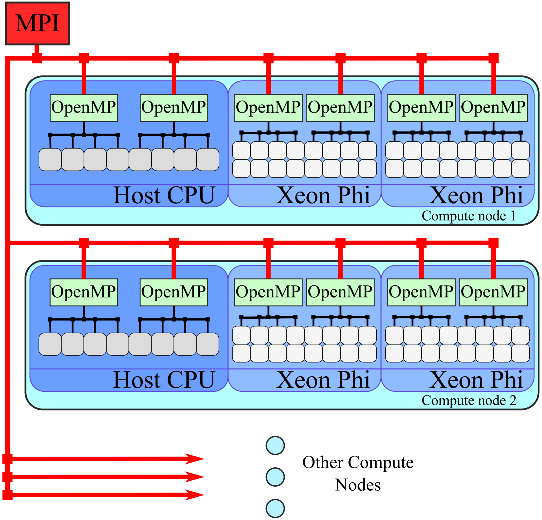
- ▷ MPI processes are multi-threaded with OpenMP.
- ▷ MPI runs only on CPUs.
- ▷ MPI processes offload to coprocessor(s).
- ▷ OpenMP in offload regions.



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

Native	**Offload**
$\leq$ 16 GiB	> 16 GiB
All parallel	Parallel + serial phases
Complex data structures	Bitwise-copyable data
Any arithmetic intensity	(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:

```
#include <cstdio>
int main() {
    printf("Hello World from host!\n");
#pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n"); fflush(stdout);
    }
    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.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

```
__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 processors.\n",
  sysconf(_SC_NPROCESSORS_ONLN )); fflush(stdout);
}
```

```
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:

```
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 OFFLOADS USING HOST THREADS

```
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

DYNAMIC WORK DISTRIBUTION

```
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`

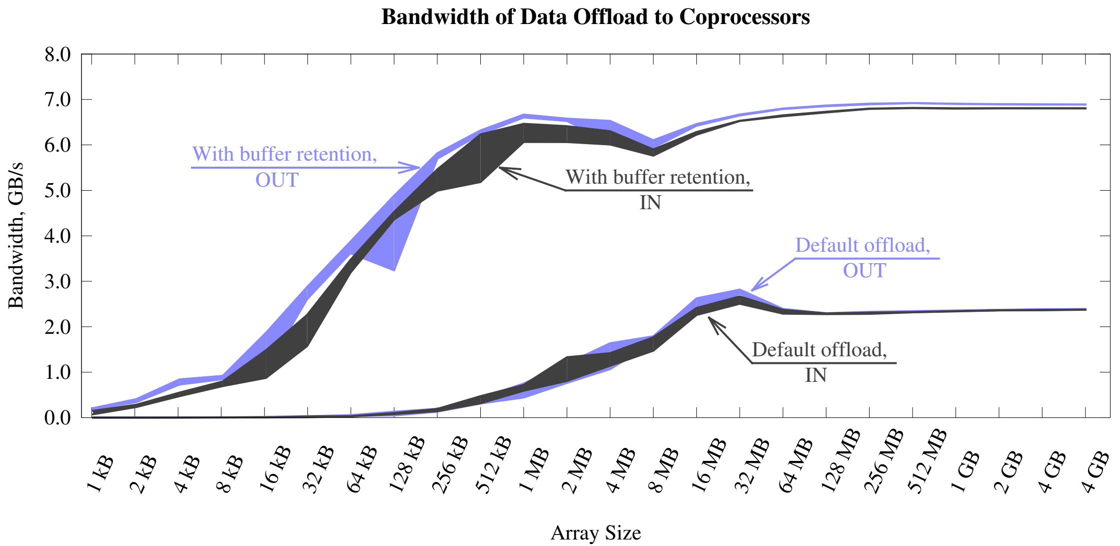
MEMORY ALLOCATION CONTROL

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

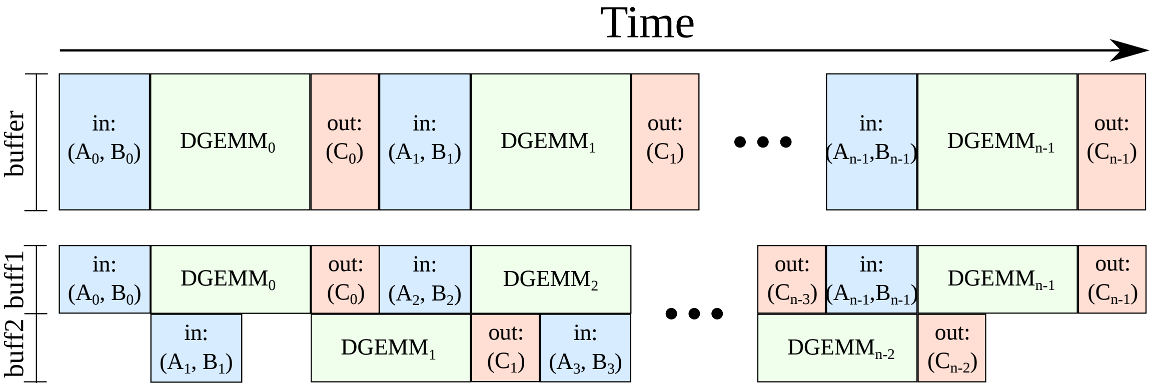
- By default, `#pragma offload` blocks until offload completes
- Use clause “`signal`” with any pointer to avoid blocking
- Use `#pragma offload_wait` to block where needed

```
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.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`

```
#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 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

```
_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

- 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

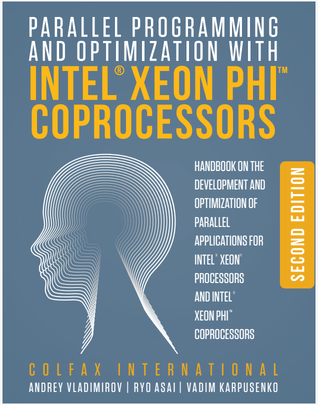
colfaxresearch.com/how-knl/

TEXTBOOK

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

Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

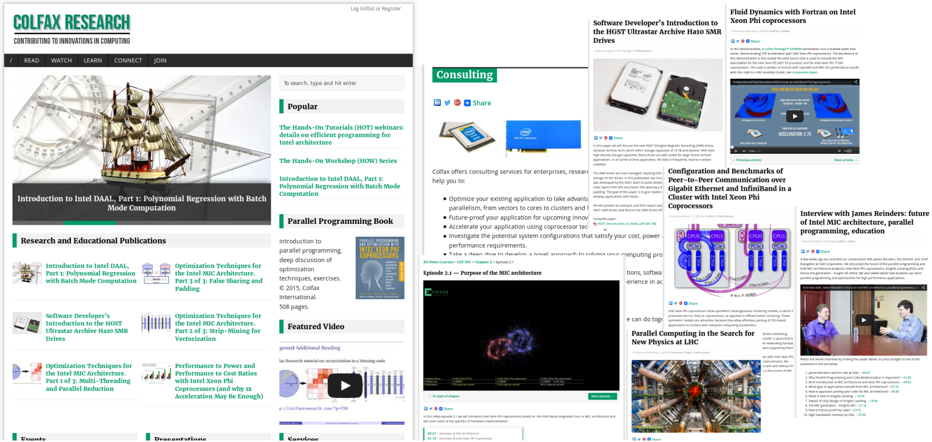
Handbook on the Development and Optimization of Parallel Applications for Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors



http://xeonphi.com/book

LEARN MORE

COLFAX RESEARCH



http://colfaxresearch.com/

LEARN MORE