EPCC Training Day 1: Offload

James Briggs

[1]COSMOS DiRAC

April 29, 2015

Session Plan

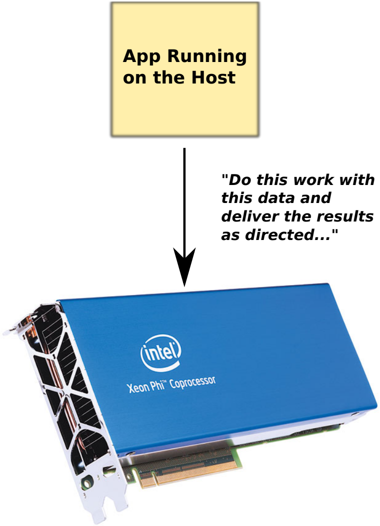
1. Concepts
2. Offloading with Intel LEO
3. Data Movement in Intel LEO
4. Asynchronous Execution
5. Compiling and Running

Section 1

Concepts

Offloading – Accelerator Mode

- A program runs on the host and “offloads” work by specifying that the Xeon Phi executes a block of code.
- The host also directs the movement of data between the host and the co-processor.
- Similar data model to GPGPU.



Offload Models

- Explicit
  - Programmer explicitly directs data movement and code execution.
  - This is achievable with Intel LEO, OpenMP 4.0, or with low level API.
- Implicit Offload
  - Virtual shared memory provided by Cilk Plus.
  - Programmer marks some data as “shared” in the virtual sense.
  - Runtime automatically synchronizes values between host and co-processor.
- Offload Enabled Library
  - Library manages offloading and data movement internally.
  - Examples: Intel MKL, MAGMA.

Section 2

Offloading with Intel LEO

Offload with Intel LEO

- LEO - Language Extensions for Offload.
- Add pragmas and new keywords to working code to make sections run on the co-processor.
- Heterogeneous compiler
  $\Rightarrow$ generates code for both the processor and co-processor architecture.

Intel LEO – Offload Syntax

- Designate a block of code to be ran on the coprocessor.
- C/C++:

```
#pragma offload target(mic[:target-number]) [,clause...]
{...}
```

- Fortran:

```
!dir$ offload target(mic[:target-number]) [,clause...]
...
!dir$ end offload
```

- `target-number` allows you to specify which logical Phi number if there are multiple.

Intel LEO – Offloading Functions

- Declare that a function or global variable should be compiled for *both* host and coprocessor using attribute keyword.
- C/C++

```
__attribute__((target(mic))) int g_size;

__attribute__((target(mic)))
double myfunc(double* a, double* b)
{ ... }
```

- Fortran:

```
!dir$ attributes offload : mic :: g_size
integer :: g_size;

!dir$ attributes offload : mic :: my_func
function myfunc(a, b)
```

Intel LEO – Offloading Functions

- C/C++ – entire blocks of code:

```
#pragma offload_attribute(push,target(mic))
int g_size;

double myfunc(double* a, double* b)
{ ... }
#pragma offload_attribute(pop)
```

- Fortran – can only do variables:

```
!dir$ options /offload_attribute_target=mic
integer :: g_size
real :: x
!dir$ end options
```

Section 3

Data Movement in Intel LEO

Data Movement

- *Memory on host and coprocessors are* **separate** *both physically and virtually.*
- With LEO programmer must copy in/out *explicitly*:
  - Programmer designates variables that need to be copied between host and card in the offload pragma/directive.
  - Provide additional clauses to the offload pragma.

Data Movement Clauses

- `in(var1 [,...])`:
  Copy from host to coprocessor.
- `out(var1 [,...])`:
  Copy from coprocessor to host.
- `inout(var1 [,...](`:
  Copy from host to coprocessor and back to host at end.
- `nocopy(var1 [,...])`:
  Don't copy selected variables.

Data Movement Example

```
double a[100000], b[100000], c[100000], d[100000];
...
#pragma offload target(mic) \
 in(a), out(c,d), inout(b)

 #pragma omp parallel for
 for (i=0; i<100000; i++) {
   c[i] = a[i] + b[i];
   d[i] = a[i] - b[i];
   b[i] = -b[i];
}
```

Dynamically Allocated Data

- Dynamically allocated data needs also to be allocated and freed on the coprocessor.
- Add additional clauses to in/out/inout:
  - `length(element-count-expr)`:
    Copy N elements of the pointer's type
  - `alloc_if (condition)`:
    Allocate memory to hold data referenced by pointer if condition is TRUE.
  - `free_if (condition)`:
    ree memory used by pointer if condition is TRUE

Example

```
int N = 5000000;

double *a, *b;
a = (double*) _mm_malloc(N*sizeof(double),64);
b = (double*) _mm_malloc(N*sizeof(double),64);
...
#pragma offload target(mic) \
  in(a : length(N) alloc_if(1) free_if(1)), \
  out(b : length(N) alloc_if(1) free_if(0))

  #pragma omp parallel for
  for (i=0; i<N; i++) {
    b[i] = 2.0*a[i];
  }
```

Example – Useful Macros

- More convenient and readable to use the following macros:

```
#define alloc_if(1) ALLOC
#define alloc_if(0) REUSE
#define free_if(1) FREE
#define free_if(0) RETAIN
```

Example – with Macros

```
int N = 5000000;

double *a, *b;
a = (double*) _mm_malloc(N*sizeof(double),64);
b = (double*) _mm_malloc(N*sizeof(double),64);
...
#pragma offload target(mic) \
  in(a : length(N) ALLOC FREE), \
  out(b : length(N) ALLOC RETAIN)

  #pragma omp parallel for
  for (i=0; i<N; i++) {
    b[i] = 2.0*a[i];
  }
```

Offload Transfer

- Can also do a **data-only** offload, that only moves data and doesn't execute code on the coprocessor.
- Syntax C/C++:

```
#pragma offload_transfer target(mic[:target-number]) [,clause...]
```

- Fortran:

```
!dir$ offload_transfer target(mic[:target-number]) [,clause...]
```

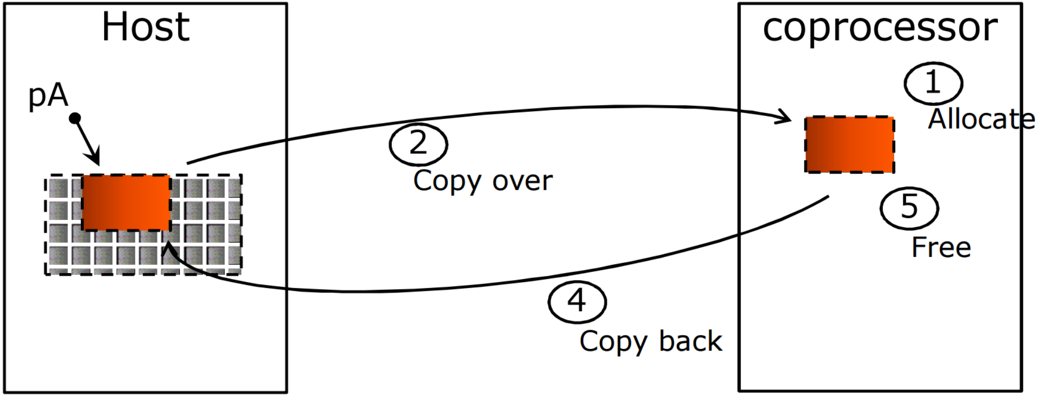
- All the clauses from the `offload` pragma also apply to `offload_transfer`.

Example

```
#pragma offload_transfer target(mic:0) \
 in(a : length(N) ALLOC RETAIN), \
 nocopy(b : length(N) ALLOC RETAIN)
```

- a – the space is allocated on Phi and data is copied over.
- b – the space is allocated on Phi, but no data is transferred.

Offload Dynamic Data Life-cycle



- 3. `#pragma offload inout(pA:length(n)) {...}`

Section 4

Asynchronous Execution

Intel LEO – Offload Clauses

- `if(stmt)`
  Allow a test at execution time for whether or not the executable should try to offload the statement. If `true` then execute on the coprocessor.
- `signal(tag)`
  If clause is included then the offload section occurs *asynchronously*. This allows for concurrent host / coprocessor usage.
- `wait(tag)`
  Include it to specify a wait for completion of a previously initiated asynchronous data transfer or asynchronous computation.

Intel LEO – Offload Clauses

- There is also a wait-only pragma
- C/C++ Syntax:

```
#pragma offload_wait target(mic[:target-number]) wait(s)
```

- Fortran Syntax:

```
!dir$ offload_wait target(mic[:target-number]) wait(s)
```

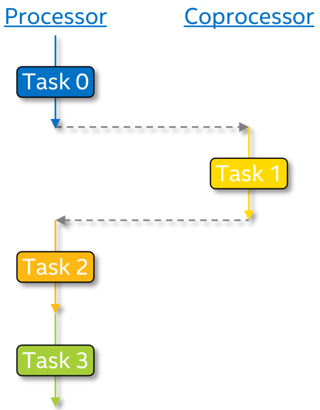
Intel LEO – Usage Models

- There are at least three different usage models for offload:
  1. Host *offloads and waits* for the coprocessor to finish the task.
  2. Host offloads and works on a *different task*.
  3. Host offloads and works on a *part of the same task*.
- Possible within MPI tasks and with multiple coprocessors.
- Reverse offloading (coprocessor $\longrightarrow$ host) possible in theory, but *not implemented*.

Usage Model – Offload / Wait

- Most common offload model.
- Host execution waits until coprocessor has finished.

```
Task0();
#pragma offload target(mic:0)
{
    Task1(0, N);
}
Task2();
Task3();
```

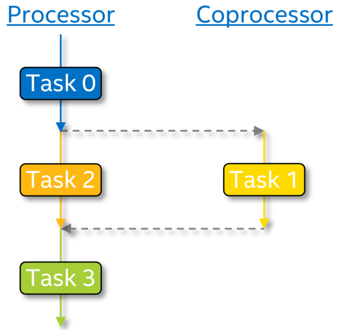


Courtesy: John Pennycook (Intel Corp.)

Usage Model – Concurrent

- Common offload model.
- Host intiates asynchronous offload of one task, and then executes a different task simultaneously.

```
Task0();
int s=0;
#pragma offload target(mic:0) signal(s)
{
    Task1(0, N);
}
Task2();
#pragma offload_wait target(mic:0) wait(s)
Task3();
```

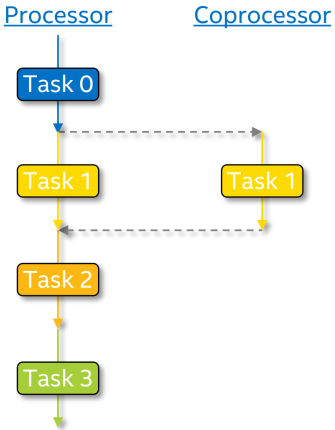


Courtesy: John Pennycook (Intel Corp.)

Usage Model – Worksharing

- Least common offload model and hardest to do right.
- Host and coprocessor work on different domains of the same problem in parallel.

```
int s=0;
Task0();
#pragma offload target(mic:0) signal(s)
{
    Task1(0,3*N/4);
}
Task1(3*N/4,N);
Task2();
#pragma offload_wait target(mic:0) wait(s)
Task3();
```



Courtesy: John Pennycook (Intel Corp.)

Section 5

Compiling and Running

Compiling and Running

- Compiling:
  - To compile code that has offload sections **no** additional flags are needed by the Intel compiler (MPSS install is required however).
- Running:
  - Controlled via environment variables:

```
export OFFLOAD_DEVICES=0
export MIC_ENV_PREFIX=MIC
export MIC_OMP_NUM_THREADS=236
export MIC_KMP_AFFINITY=compact,granularity=fine
```

Compiling and Running

- STDOUT/STDERR are piped back to the host STDOUT/STDERR so print statements can be seen in offload code.
- Remember to flush:

```
printf("Hello\n");
fflush(0);
```

- Useful environment variables:

```
#ifdef __MIC__    // if code is compiled for MIC
#ifdef __INTEL_OFFLOAD // if code is offload code
```

Summary

Intel LEO Main Pragmas:

```
#pragma offload target(mic)
#pragma offload_transfer target(mic)
#pragma offload_wait target(mic)
```

Intel LEO Data Transfer Clauses:

```
in/out/inout/nocopy
alloc_if(), free_if()
length, into
```

Intel LEO Asynchronous Clauses:

```
signal(s), wait(s)
```