# OFFLOAD MODE PROGRAMMING

Adrian Jackson

adrianj@epcc.ed.ac.uk

@adrianjhpc

# Overview

- Offloading with Intel LEO
- Data Movement in Intel LEO
- Asynchronous Execution
- Compiling and Running

# Offloading model

- Similar data model to GPGPU.
  - *Kernels* of work run on co-processors, main program on host
- 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
  - Data loaded on to the co-processor, and results copied off
- Reduces user interaction with co-processor

# Programming models

- Three different ways offload can be programmed
  - Explicit
  - Implicit
  - Library
- Explicit
  - Programmer explicitly directs data movement and code execution
  - This is achievable with Intel LEO, OpenMP 4.0, or with low level API
- Implicit
  - Virtual shared memory provided by Cilk Plus
  - Programmer marks some data as shared
  - Runtime automatically synchronizes values between host and co-processor
- Library
  - Some libraries have offload kernels implemented in them, i.e. Intel MKL
  - Library manages offloading and data movement internally.

# Intel LEO

- LEO – Language Extensions for Offload
- Compiler can generate code for host and co-processors
- LEO adds:
  - pragmas and keywords make sections run on the Xeon Phi
- C/C++:

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

- Fortran:

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

`target-number:`

  - Optional, can be used to specific a specific Xeon Phi

# LEO offload attribute

- Can mark entire function or global variable for offloading
  - Will compile/create for both host and co-processor

- C/C++

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

- Fortran

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

# Offloading blocks of code

- Also possible to offload a whole section of code:

```
#pragma offload_attribute(push, target(mic))
int gsize;
double myfunc (double* a, double* b)
{...}
#pragma offload_attribute(pop)
```

- Fortran: Only possible for variables

```
!dir$ options /offload_attribute_target=mic
integer :: mydata
real :: rsize
!dir$ end options
```

# Data movement

- Co-processor and host have different memory and memory spaces
- LEO requires explicit data movement
  - Data movement directives
  - Offloading directives can also include information about data
- Data clauses for offload directives
  - Copy from host to Xeon Phi

```
in(var1 [,...])
```

  - Copy from coprocessor to host.

```
out(var1 [,...])
```

  - Copy from host to coprocessor and back to host at end.

```
inout(var1 [,...])
```

  - Don't copy selected variables.

```
nocopy(var1 [,...])
```

# Movement examples

- C:

```
double data1[1000], data2[2000], data3[500], outputdata[2000]
#pragma offload target(mic) in(data2), out(outputdata), inout(data1,data3)

#pragma omp parallel for
for(i=0;i<500;i++){
  data1[i] = data2[i] + data3[i];
  data3[i] = data1[i]*data1[i];
  outputdata[i] = data1[i] + data3[i];
}
```

- Fortran

```
real, dimension(1000) :: data1
real, dimension(2000) :: data2
real, dimension(500) :: data3
real, dimension(2000) :: outputdata
!dir$ offload target(mic) in(data2), out(outputdata), inout(data1,data3)

!omp$ parallel do
do i=1,500
  data1(i) = data2(i) + data3(i)
  data3(i) = data1(i) * data1(i)
  outputdata(i) = data1(i) + data3(i)
end do
```

# Dynamic data

- Dynamically allocated data needs to be managed on the Xeon Phi
- 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 on co-processor if condition is true

  `free_if(condition)`

  - free memory used by pointer on co-processor if condition is true

# Dynamic data examples

- C:

```
double *data1, *data2, *data3, *outputdata;
data1 = (double *) malloc(1000*sizeof(double));
data2 = (double *) malloc(2000*sizeof(double));
data3 = (double *) malloc(500*sizeof(double));
outputdata = (double *) malloc(2000*sizeof(double));
#pragma offload target(mic) in(data2: length(2000) alloc_if(1)
free_if(0)), out(outputdata: length(2000) alloc_if(1) free_if(1)),
inout(data1: length(1000) alloc_if(1) free_if(1)), inout(data3:
length(500) alloc_if(1) free_if(1))
```

- Fortran

```
real, allocatable, dimension(:) :: data1, data2, data3, outputdata
allocate(data1(1000))
allocate(data2(2000))
allocate(data3(500))
allocate(outputdata(2000))
!dir$ offload target(mic) in(data2: length(2000) alloc_if(1)
free_if(0)), out(outputdata: length(2000) alloc_if(1) free_if(1)),
inout(data1: length(1000) alloc_if(1) free_if(1)), inout(data3:
length(500) alloc_if(1) free_if(1))
```

# Data only transfer

- Move data without code execution on co-processors
  - `offload_transfer`

- Fortran

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

- C/C++

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

# Data only transfer example

- Fortran:

```
!dir$ offload_transfer target(mic:0)
in(a:length(N) alloc_if(1) free_if(0))
nocopy(b:length(N) alloc_if(1) free_if(0))
```

- C:

```
#pragma  offload_transfer target(mic:0)
in(a:length(N) alloc_if(1) free_if(0))
nocopy(b:length(N) alloc_if(1) free_if(0))
```

# Asynchronous execution

- Previous examples driven by host code
  - Host code blocking whilst accelerator executes
- Asynchronous execution allows host to also execute whilst co-processor is working
  - `if(stmt)`
    - If `stmt` is true then code is executed on the co-processor, if not executed on the host
  - `signal(tag)`
    - Triggers asynchronous execution of offload section.
  - `wait(tag)`
    - Wait for previous asynchronous execution of data transfer to complete. Matches with `tag` in previous `signal` statement

# Wait

- Can do a wait by itself (without data transfer or code execution)
- Fortran

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

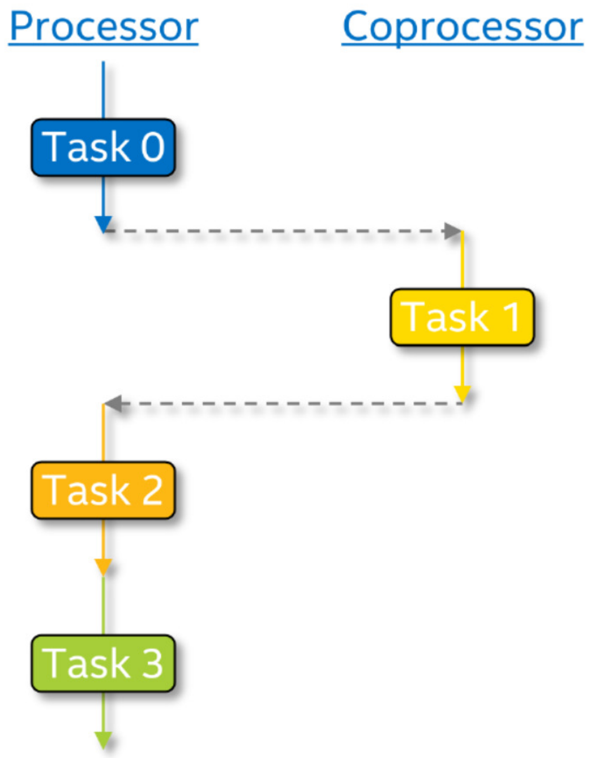
- C/C++

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

# Offload modes: Offload and wait

- Execute on co-processor, host waits

```
work1();
#pragma offload target(mic)
{
  work2();
}
work3();
…
```

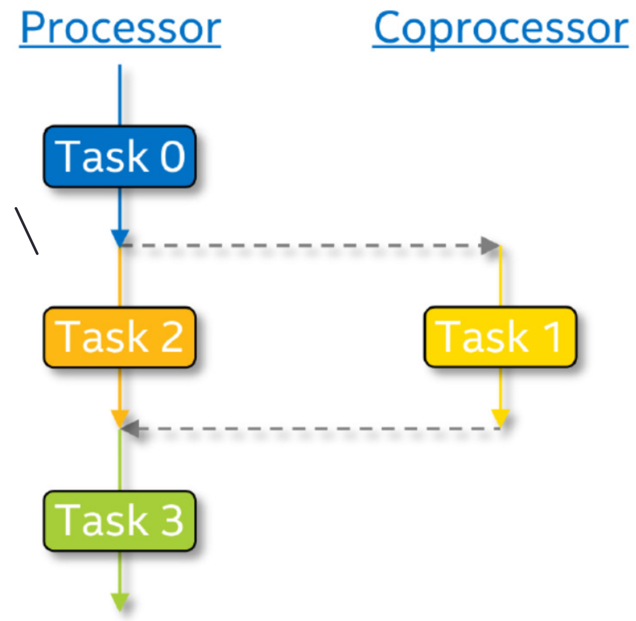


Courtesy: John Pennycook (Intel Corp.)

# Offload modes: Concurrent

- Execute on co-processor and host, same thing, different parts

```
int sig=0;
work1();
#pragma offload target(mic)\
signal(sig)
{
  work2();
}
work3();
#pragma offload_wait \
target(mic) wait(sig)
…
```

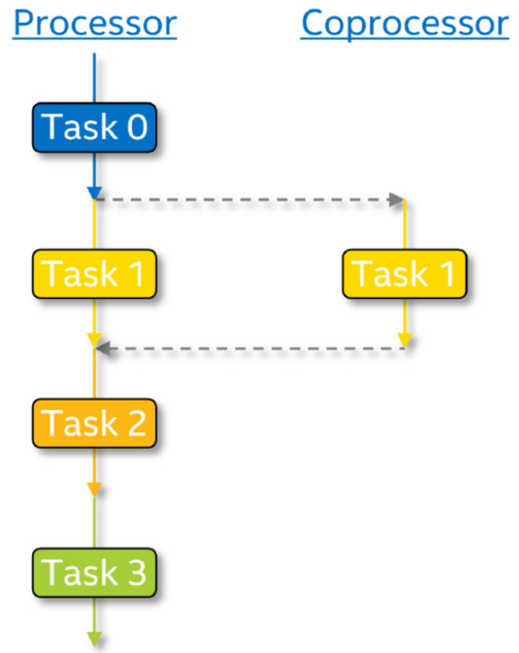


Courtesy: John Pennycook (Intel Corp.)

# Offload modes: Symmetric

- Execute on co-processor and host, doing different things

```
int sig=0;
work1();
#pragma offload target(mic)\
signal(sig)
{
  work2(N/4);
}
work2(3N/4);
#pragma offload_wait \
target(mic) wait(sig)
work3()
…
```



Courtesy: John Pennycook (Intel Corp.)

# Running offload

- Compilation is same as normal code
  - No special flags or libraries needed
  - MPSS install is required

- Running offload code uses environment variables

```
export OFFLOAD_DEVICES=1

export MIC_ENV_PREFIX=MIC

export MIC_KMP_AFFINITY=compact,granularity=fine

export MIC_OMP_NUM_THREADS=236
```

# Output and conditional compilation

- Output is returned to host
  - `fflush` (C/C++) or `flush` (Fortran) may be required to get output to appear real time

- Can use pre-defined pre-processor macros in code

```
#ifdef __MIC__

#ifdef __INTEL_OFFLOAD__
```