Offload Computing on Stampede

Kent Milfeld
milfeld@tacc.utexas.edu

IEEE Cluster 2013
Sept. 23, 2013
MIC Information

- **mic-developer** (programming & training labs): http://software.intel.com/mic-developer

- **Intel Programming & Compiler for MIC**

- Intel Compiler Manuals: **C/C++  Fortran**
  (Key Features → Intel ® MIC Architecture)

- **example code**: /opt/apps/intel/13/composer_xe_2013.2.146/
  Samples

- Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, James Reinders. Intel Xeon Phi Coprocessor High Performance Computing, Jim Jeffers & James Reinders

- **Stampede User Guide**: http://www.tacc.utexas.edu/ (User Services → UserGuides → Stampede)
Outline

Offloading

• Offloading: Basic Concepts
  – Basics
  – Directive Syntax
  – Automatic Offloading (AO)
  – Compiler Assisted Offloading (CAO)
    • Directives (Code Blocks – Targets)
    • Preparation and Offload Process Steps (mechanism)
    • Data Transfers
    • Declaration for Functions and Globals, Pointer Data
    • Persistent Data
    • Asynchronous Offloading

• Offloading inside an OMP parallel region.
• Offloading: Basic Concepts
  – Basics
  – Directive Syntax
  – Automatic Offloading (AO)
  – Compiler Assisted Offloading (CAO)
    • Directives (Code Blocks – Targets)
    • Preparation and Offload Process Steps (mechanism)
    • Data Transfers
    • Declaration for Functions and Globals, Pointer Data
    • Persistent Data
    • Asynchronous Offloading

• Offloading inside an OMP parallel region.
Definition of a Node

A “node” contains a host and a MIC component

- **host** – refers to the Sandy Bridge component
- **MIC** – refers to Intel Xeon Phi co-processor cards

### NODE on Stampede

**host**
- 2x Intel 2.7 GHz E5-2680
- 16 cores
- 32 GB Memory

**MIC**
- 1 or 2 Intel Xeon PHI SE10P
- 61 cores/244 HW threads
- 8GB Memory
Offloading Strategy

• Think threads
  – (Whether working on a MIC, GPU, ARM, etc.)

• Options:
  – Have the MIC do all of the work
    • May be viable for low-performance-CPU – MIC solution
  – Share the work -- host and MIC
    • More reasonable for HPC system with MICs

• Great time to venture into many-core architectures
  1.) Try offloading compute-intensive section
      If it isn’t threaded, make it threaded
  2.) Optimize data transfers
  3.) Split calculation & use asynchronous mechanisms
Basics: What is Offloading

- Send block of code to be executed on coprocessor (MIC).
  - Must have a binary of the code (code block or function).
  - Compiler makes the binary and stores it in the executable (a.out).

- During execution on the CPU, the “runtime” is contacted to begin executing the MIC binary at an offload point.
  - When the coprocessor is finished, the CPU resumes executing the CPU part of the code.
Models

- **Non-Shared memory**
  - Host and MIC have separate memory sub systems—think distributed memory and bit-wise data copy between platforms.

- **Virtual-Shared Memory**
  - C/C++; complex data structures (pointer based structures, classes, etc.) can be shared; coherency overhead.

**Best:** When compute complexity is $O(N^{i+1})$ and data complexity is $O(N^i)$

Code is non-IO intensive

Offload can be done asynchronously
Basics: Directives

- Directives can be inserted before code blocks and functions to run the code on the Xeon Phi Coprocessor (the “MIC”).
  - No recoding required. (Optimization may require some changes.)
  - Directives are simple, but more “details” (specifiers) can be used for optimal performance.
  - Data must be moved to the MIC
    - For large amounts of data:
      Amortize with large amounts of work.
      Keep data resident (“persistent”).
      Move data asynchronously.
Basics: Simple Example

```c
int main(){
  float a[10]; int i;
  #pragma offload target(mic)
  { for(i=0,i<10;i++)
    a[i]=(float) i;
  }
  #pragma offload target(mic)
  foo(a);
  printf(" %f \n",a[10]);
}
```

- Insert Offload Directive:
- Compile with Intel Compiler:
- How to turn off offloading:

```
#pragma offload target(mic) C/C++
!dir$ offload begin target(mic)
do i=1,10
  a(i)=i; end do
!dir$ end offload
!dir$ offload target(mic)
call foo(a)
print*, a(10)
end program
```

use `–no-offload` option
OpenMP regions can be offloaded directly.

OpenMP parallel regions can exist in offloaded code blocks or functions.
Compile & Run

- Compile on login node (as shown), or on compute node interactively (see `idev` in lab exercise).

- Run on compute node (or in batch script).

- Use KMP_AFFINITY when thread count < 4*core count.

```
login2$  icc -openmp -xhost -O3 omp_prog.c
login2$  ifort -openmp -xhost -O3 omp_prog.f90
login2$  idev ...  
```

```
c559-001$  export MIC_PREFIX=MIC
```

```
c559-001$  export OMP_NUM_THREADS=16
```

```
c559-001$  export MIC_OMP_NUM_THREADS=240
```

```
c559-001$  ./a.out
```

Tell runtime to find MIC_ prefixed variables, strip off MIC_ and use them on MIC.

“C559-001$” is the shell prompt for a compute node (host+mic) after executing `idev`
• Offloading: Basic Concepts
  – Basics
  – Directive Syntax
  – Automatic Offloading (AO)
  – Compiler Assisted Offloading (CAO)
    • Directives (Code Blocks – Targets)
    • Preparation and Offload Process Steps (mechanism)
    • Data Transfers
    • Declaration for Functions and Globals, Pointer Data
    • Persistent Data
    • Asynchronous Offloading

• Offloading inside an OMP parallel region.
#pragma offload specifier [ [,] specifier ]

C/C++

Fortran

specifier:

- **target**( targ-name [:dev#] )
- **if**( if-specifier ) or **mandatory**
- **signal**( tag ) or **wait**( tag )
- **data_specifier**(...)

Intel calls this an “offload-parameter”. For this training module I named it something more reasonable.

Often called “clauses”.
**Offload Directive**

**data_specifier:**

- `in` (identifier [,]identifier... [: modifier [,]modifier... ] )
- `out("")`
- `inout("")`
- `nocopy("")`

Variables, arrays...

**For explicit data transfers.**

- `length()`
- `alloc_if()`
- `free_if()`
- `align`

Storage handlers
Offload Directives

C/C++ starts with: #pragma
Fortran starts with: !dir$

offload*
offload_attribute
offload_transfer
offload_wait

Stand Alone directives (no offload code)

Specifies MIC vars & functions
Data Host ➔ MIC
Wait for async. offload

* Fortran uses offload begin ... end offload, C/C++ uses {...}

__attribute__ and __declspec "decorations" can be used in lieu of offload_attribute in C/C++. Use !dir$ attributes list in Fortran.
• Offloading: Basic Concepts
  – Basics
  – Directives
  – Automatic Offloading (AO)
  – Compiler Assisted Offloading (CAO)
    • Directives (Code Blocks – Targets)
    • Preparation and Offload Process Steps (mechanism)
    • Data Transfers
    • Declaration for Functions and Globals, Pointer Data
    • Persistent Data
    • Asynchronous Offloading
• Offloading inside an OMP parallel region.
Automatic Offload

- Offloads some MKL routines automatically
  - No coding change
  - No recompiling
- Makes sense with BLAS-3 type routines
  - Minimal Data $O(n^2)$, Maximal Compute $O(n^3)$
- Supported Routines (more to come)

<table>
<thead>
<tr>
<th>Type</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-3 BLAS</td>
<td>$x$GEMM, $x$TRSM, STRMM</td>
</tr>
<tr>
<td>LAPACK 3 amigos</td>
<td>LU, QR, Cholesky</td>
</tr>
<tr>
<td>Eigen Solver</td>
<td></td>
</tr>
</tbody>
</table>
Automatic Offload

- Compile as usual, use new –mkl
  - Works with serial, OpenMP and MPI codes.
- Enable with MKL_MIC_ENABLE variable

```sh
login1$ ifort -mkl -xhost -O2 app_has_MKLdgemm.f90
login1$ icc -mkl -xhost -O2 app_has_MKLdgemm.c
...
c559-001$ export OMP_NUM_THREADS=16
c559-001$ export MKL_MIC_ENABLE=1
c599-001$ ./a.out
```

See MKL_MIC_WORKDIVISION environment variable to set (force) a relative work load.
• Offloading: Basic Concepts
  – Basics
  – Directives
  – Automatic Offloading (AO)
  – Compiler Assisted Offloading (CAO)
    • Directives (Code Blocks – Targets)
    • Preparation and Offload Process Steps (mechanism)
    • Data Transfers
    • Declaration for Functions and Globals, Pointer Data
    • Persistent Data
    • Asynchronous Offloading

• Offloading inside an OMP parallel region.
Compiler Assisted Offload

• Compiler looks for offload directive everywhere:
  – Before blocks, functions (subroutines), statements
  – For global variables and function declarations
  – As stand-alone directives for data transfer and waits

• Target( mic : dev_id )

  target(mic) : Execute on runtime selected MIC, on cpu if error or not available
  target(mic:-1) : Execute on runtime selected MIC, fail otherwise
  target(mic:0-n) : Execute on device id=mod(#, no. of coprocs), fail otherwise

With more than 1 MIC, use dev-id with: offload, offload_transfer, offload_wait
Compiler Assisted Offload

int main(){
    ...
    #pragma offload target(mic:0)
    {
        #pragma omp parallel for
        for (i=0; i<N;i+){
            a[i]=sin(b[i])+cos(c[i]);
        }
    }
    ...
    }
    program main
    ...
    !$omp parallel do
    do i = 1,N
    a(i)=sin(b(i))+cos(c(i))
    end do
    !$omp end parallel do
    ...
    end program

• Data (a, b, and c) within lexical scope are moved implicitly.

• C/C++ use {...} (curly braces) to mark a block
• Fortran use begin and !dir$ end offload to mark block
The Offload Preparation

- Code is instrumented with directives.
- Compiler creates a CPU binary and a MIC binary for offloaded code block.
- Loader places both binaries in a single file. (→ a.out)
- During CPU execution of the application an encountered offload code block is executed on a coprocessor (through runtime), subject to the constraints of the target specifier...
The basic operations of an offload rely on interaction with the runtime to:

- Detect a target phi coprocessor
- Allocate memory space on the coprocessor
- Transfer data from the host to the coprocessor
- Execute offload binary on coprocessor
- Transfer data from the coprocessor back to the host
- Deallocate space on coprocessor

Binaries are moved on first offload.
Data Transfers

• If you know the intent of data usage, minimize unnecessary transfers with in/out/inout data specifiers.

```c
#pragma offload target(mic [:dev_id]) data_specifier(identifier_list)//syntax

#pragma offload target(mic) in( b,c ) // Only copy b and c into MIC
#pragma offload target(mic) out(a ) // Only return a
#pragma offload target(mic) inout(d ) // Default, copy into and out of
```
Data Transfers

```
int main(){
    ...
    #pragma offload target(mic) \ 
in(b,c) out(a)

    {
        #pragma omp parallel for
        for (i=0; i<N;i+){
            a[i]=sin(b[i])+cos(c[i]);
        }
    }
    ...
}

program main
    ...
    !dir$ offload begin target(mic) & 
in(b,c) out(a)

    !$omp parallel do
    do i = 1,N
        a(i)=sin(b(i))+cos(c(i))
    end do
    !dir$ end offload
    ...
end program
```
Offload Functions, Globals & Pointer Data

- “Decorate” all functions **used** in offloads with a target “attribute”.
- Likewise with globals

```plaintext
offsetof_attribute_ ( ( target(mic) ) ) <followed by function/global declaration>  C/C++
offsetof_declspec    ( target(mic) ) <followed by function/global declaration>
!
dir$ attributes offload:mic :: <function/subroutine name or variables>  F90
```
### Offload Functions, Globals & Pointer Data

**C/C++**

```c
__declspec(target(mic))
int global = 0;

__declspec(target(mic))
int foo()
{
    return ++global;
}

main()
{
    int i;
    #pragma offload target(mic) inout(global)
    {
        i = foo();
    }
    printf("global:i=%d:%d both=1\n", global, i);
}
```

**F90**

```f90
module mydat
!dir$ attributes offload:mic :: global
    integer :: global = 0
end module mydat

!dir$ attributes offload:mic :: foo
    integer function foo
        use mydat
        global = global + 1
        foo = global
    end function foo

program main
    use mydat
    integer i
    integer, external :: foo
    !dir$ attributes offload:mic :: foo
    !dir$ offload target(mic:0) inout(global)
    i = foo()
    print *, "global:i=", global, i, "(both=1)"
end program main
```
Offload Functions, Globals & Pointer Data

- Offload attributes can be applied to an entire file through a compiler option:

```
icpc | icc | ifort  -c -offload-attribute-target=mic      my_fun.cpp | c | f90
icpc | icc | ifort      my_fun.o                      my_app.cpp | c | f90
```

- **C/C++** has file scoping, **FORTRAN** does not:

```
#pragma offload_attribute(push, target(mic))
void fun1(int i) {i=i+1;}
void fun2(int j) {j=j+2;}
#pragma offload_attribute(pop)
```

```
module my_globs
!dir$ options /offload_attribute_target=mic
real, allocatable :: in1(:,), in2(:,), out1(:,), out2(:)
!dir$ end options
end module
```
Offload Functions, Globals & Pointer Data

- C pointer to contiguous data requires **length modifier**—(default copy is 1 element).
- Not required for Fortran allocated arrays.

```c
... a=(double *) malloc(N * sizeof(double));
b=(double *) malloc(N * sizeof(double));
c=(double *) malloc(N * sizeof(double));
d=(double *) malloc(M * sizeof(double));
e=(double *) malloc(N*2*sizeof(double));
...
```

```c
#pragma offload target(mic:0) in( a,b,c : length( N ) ) // pointers a, b & c, length N
#pragma offload target(mic:0) out( d : length( M ) ) // pointer d has length M
#pragma offload target(mic) inout( e : length(2*N) ) // pointer e has length of N*2
```

Alignment might be important
Persistent Data

- Default implicit and explicit behavior: allocate space for all data before offload, and deallocate (free) on offload completion.

```c
alloc_if( logic_expression )  // if true allocate space at begin
free_if( logic_expression )   // if true free space at end
```

- The `offload_transfer` directive allows data management (data specifiers) without a code block. It is a stand-alone directive.
Persistent Data

- Fortran and C/C++ syntaxes are identical, except:
  - Sentinels are different: #pragma versus !dir$
  - Truth variables: Fortran: logical .true./.false.  C/C++ int 1/0

```plaintext
#pragma offload data_specifier( identifier(s): alloc_if(TorF) free_if(TorF) )
```

```plaintext
#pragma offload ... in( a : alloc_if(1) free_if(0) )  //allocate space, don’t free at end
{...}
#pragma offload ... inout( a : alloc_if(0) free_if(0) )  //don’t allocate, don’t free at end
{...}
#pragma offload ... out( a : alloc_if(0) free_if(1) )  //don’t allocate, free at end
{...}
#pragma offload_transfer... in( a : alloc_if(1) free_if(0) )  //allocate space, don’t free at end
... ...
#pragma offload_transfer... out( a : alloc_if(0) free_if(1) )  //don’t allocate, free space at end
```

... == target(mic)
## Alloc/Free Truth Table

<table>
<thead>
<tr>
<th>Allocation Operation</th>
<th>Deallocation (Free) Operation</th>
<th>Operations Performed (Use Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc_if(true)</td>
<td>free_if(true)</td>
<td>This is the default when no storage operations are specified. Allocate space at beginning, free at end.</td>
</tr>
<tr>
<td>alloc_if(true)</td>
<td>free_if(false)</td>
<td>Allocate space, don’t free (make space available on device, and retain for future use).</td>
</tr>
<tr>
<td>alloc_if(false)</td>
<td>free_if(true)</td>
<td>don’t allocate, but free (reuse device storage, but will not need later)</td>
</tr>
<tr>
<td>alloc_if(false)</td>
<td>free_if(false)</td>
<td>don’t allocate, don’t free (reuse device storage, and leave for future use)</td>
</tr>
</tbody>
</table>
Asynchronous Offloading

- Default behavior: CPU process waits for offload to complete.
- **Signal and wait specifiers** allow CPU to continue executing after the offload code block, once the runtime is notified to perform the offload (i.e. offload becomes asynchronous).
- **Offload_wait** is a stand-alone directive (no code block).

Syntax:

```
#pragma offload target(mic[:#id]) … signal(tag)
#pragma offload target(mic[:#id]) … wait(tag)
#pragma offload_wait … wait(tag)

!dir$ offload target(mic[:#id]) … signal(tag)
!dir$ offload target(mic[:#id]) … wait(tag)
!dir$ offload_wait … wait(tag)
```

(Only one tag. A list of tags is no longer supported.)
Asynchronous Offloading

- Offload events are identified by unique value of tag.
  - F90: `signal(var)
  - C/C++: `signal(&var)
- Wait/signal can have only a single tag.
- Directives can have wait and signal specifiers.

```c
#define N 10000
__attribute__((target(mic:0))) void work(int, int, int, int *);

int main()
{
    int sig1=1, i, knt=1, *a, NSm, NEm, NSc, NEC;
    a=(int*)malloc(N*sizeof(int));
    do{
        NSm=0; NEm=N/2;
        #pragma offload target(mic:0) signal(&sig1) \
        inout(a:length(NEm-1))
        work(knt,NSm,NEm, a);
        NSc=N/2; NEC=N;
        work(knt,NSc,NEC, a);
        #pragma offload_wait target(mic:0) wait(&sig1)
        knt=knt+1;
    }while (knt < 10);
} // CPU & MIC work on different parts of a
```
## Offload Thread Placement

Controlled through environment variable: `KMP_AFFINITY=<type>`

<table>
<thead>
<tr>
<th>Type</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>compact</td>
<td>pack threads close to each other</td>
</tr>
<tr>
<td>scatter</td>
<td>Round-Robin threads to cores</td>
</tr>
<tr>
<td>balanced</td>
<td>keep OMP thread ids consecutive (MIC only)</td>
</tr>
<tr>
<td>explicit</td>
<td>use the proclist modifier to pin threads</td>
</tr>
<tr>
<td>none</td>
<td>does not pin threads</td>
</tr>
</tbody>
</table>

### System with only 4 MIC CORES

<table>
<thead>
<tr>
<th>compact</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
<tr>
<td>4 5 6 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>scatter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 4</td>
</tr>
<tr>
<td>1 5</td>
</tr>
<tr>
<td>2 6</td>
</tr>
<tr>
<td>3 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
</tr>
<tr>
<td>2 3</td>
</tr>
<tr>
<td>4 5</td>
</tr>
<tr>
<td>6 7</td>
</tr>
</tbody>
</table>

Explicit, proclist=[1,2,3,4,9,10,11,12]

<table>
<thead>
<tr>
<th>CPU id (4/core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3</td>
</tr>
<tr>
<td>8, 9, 10, 11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 241, 242, 243</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software thread id</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 OpenMP threads</td>
</tr>
</tbody>
</table>

Offload automatically avoids last core (HW threads 0,241,242,243), and with scatter/compact.

Be careful if you pin threads with `explicit`, offload communication/transfers occur on last core.
• Offloading: Basic Concepts
  – Basics
  – Directives
  – Automatic Offloading (AO)
  – Compiler Assisted Offloading (CAO)
    • Directives (Code Blocks – Targets)
    • Preparation and Offload Process Steps (mechanism)
    • Data Transfers
    • Declaration for Functions and Globals, Pointer Data
    • Persistent Data
    • Asynchronous Offloading
  – Offloading inside an OMP parallel region.
Heterogeneous Computing, Concurrent

**C/C++**

```c
#pragma omp parallel
{
    #pragma omp single nowait
    #pragma offload target(mic)
    { foo(); }

    #pragma omp for schedule(dynamic)
    for(i=0; i<N; i++) {...}
}
```

**F90**

```f90
!$omp parallel
!$omp single
!$omp target(mic)
    call foo();
!$omp end single nowait

!$omp do schedule(dynamic)
    do i=1,N; ...
    end do
!$omp end parallel
```
#include <omp.h>
#include <stdio.h>

int main() {
    const int N=100000000;
    int i, nt, N_mic, N_cpu;
    float *a;

    a = (float *) malloc(N*sizeof(float));
    for(i=0;i<N;i++)a[i]=-1.0; a[0]=1.0;

    N_mic = N/2; N_cpu = N/2;
    nt = 16; omp_set_num_threads(nt);

    #pragma omp parallel
    {
        #pragma omp single nowait
        {
            #pragma offload target(MIC:0) out(a:length(N_MIC))
            #pragma omp parallel for
            for(i=0;i<N_mic;i++) { a[i]=(float)i; }
        }
        #pragma omp for schedule(dynamic,N/nt)
        for(i=N_cpu;i<N;i++) { a[i]=(float)i; }
    }

    printf("a[0],a[N-1] %f %f\n",a[0],a[N-1]);
}
OpenMP 3.0 supports nested parallelism, older implementations may ignore the nesting and serialize inner parallel regions.

A nested parallel region can specify any number of threads to be used for the thread team, new id's are assigned. Scheduling: static, etc.
omp_set_nested(1);
omp_set_max_active_levels(2);
omp_set_num_threads(2);

#pragma omp parallel
{
    printf("reporting in from %d\n", \
            omp_get_thread_num());

#pragma omp sections
{
    #pragma omp section
    {
        #pragma offload target(mic)
        bar(1);
    }
    #pragma omp section
    {
        #pragma omp parallel for num_threads(3)
        for(i=2;i<5;i++) {bar(i);} 
    }
}

Sections allows 1 generating thread in each section.

Nested level re-defines a thread team with new thread ids.
(Worksharing team is no longer dependent upon original parallel region team size.) Scheduling can be static!


## Compiler Options and Env Vars

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-no-offload</td>
<td>Ignore offload directives</td>
</tr>
<tr>
<td>-offload-attribute-target=mic</td>
<td>Flag every global data object and routine with the offload attribute</td>
</tr>
<tr>
<td>-opt-report-phase=offload</td>
<td>Optimization phase report for offload</td>
</tr>
<tr>
<td>-offload-option,mic,compiler,&quot;option list&quot;</td>
<td>Compiler options for MIC</td>
</tr>
<tr>
<td>-offload-option, ld,compiler,&quot;option list&quot;</td>
<td>Loader options for MIC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC_ENV_PREFIX</td>
<td>(usually =MIC) Controls variables passed to MIC.</td>
</tr>
<tr>
<td>OFFLOAD_REPORT</td>
<td>(=1</td>
</tr>
<tr>
<td>MIC_STACKSIZE</td>
<td>Specifies the stack size of the main thread for the offload. (default =12M)</td>
</tr>
<tr>
<td>MKL_MIC_ENABLE</td>
<td>(=1) Sets automatic offloading on.</td>
</tr>
<tr>
<td>MKL_MIC_WORKDIVISION MKL_HOST_WORKDIVISION</td>
<td>Sets fraction of automatic offload work for MIC/HOST.</td>
</tr>
</tbody>
</table>
References

In these Compiler User Guides for offload details GO TO:
Key Features→Intel MIC Architecture→Programming for Intel MIC Architecture

• [Link to documentation]

Intel MIC Programming and Computing

• [Link to documentation]

Developer’s Guide

• [Link to documentation]

MKL

• [Link to documentation]
• [Link to webinar]

Questions?

www.tacc.utexas.edu
Offload Lab

Lab instructions at:
tacc.utexas.edu/user-services/training/course-materials

• Exercise 1
  – Simple Offload Examples: Compilation/Execution, etc.

• Exercise 2
  – Data Transfer Optimization

• Exercise 3
  – Concurrent and Asynchronous Offloads