<table>
<thead>
<tr>
<th>#</th>
<th>pg.</th>
<th>Subject</th>
<th>Purpose</th>
<th>directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>Offload, Begin (C) (F90)</td>
<td>Compile and Run (CPU, MIC, Offload)</td>
<td>offload_hello</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>Offload, Data</td>
<td>Optimize Offload Data Transfers</td>
<td>offload_transfer</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>Offload, Async</td>
<td>OMP Concurrent CPU+MIC Execution</td>
<td>offload_stencil</td>
</tr>
</tbody>
</table>
Before you begin, create 2 or 3 windows on a login node using:

\[
\text{login: } \texttt{ssh -Y <my_login_name>@stampede.tacc.utexas.edu}
\]

If you don’t have an idev session already, create one with the following command. ONLY USE ONE IDEV SESSION!!!!

\[
\text{idev session: login4\% idev}
\]

\[
\text{...}
\]

\[
\text{C558-100\% } \leftarrow \text{this is your compute-node ip*}
\]

\[
\text{If you exit this window your idev session is aborted.}
\]

Do all of your work on a compute node (your own development node!). In the non-idev windows ssh over to the compute node.

\[
\text{other windows: login4\% ssh -Y c558-100 } \leftarrow \text{use your node ip}
\]

\[
\text{...}
\]

\[
\text{c558-100\%}
\]

Select one of your windows for doing the exercises, and use the other for running \textit{top} and other utilities in some exercises. We won’t show the prompt in commands from now on—assume you are on a compute node unless otherwise stated.

Untar files into \$HOME directory:

\[
\text{get files: } \texttt{tar -xvf ~train00/offload_lab.tar}
\]

\[
\text{cd offload_lab}
\]

* \textbf{Your new prompt from idev is your interactive compute-node ip.}

\textit{e.g.} if your prompt is: \texttt{c557-001\%} \textit{Your compute-node ip is c557-001.}

You can use this window for executing MPI code with ibrun (or any window in which you ssh’d to this node)—it has the right MPI environment. Also, use it for editing, compiling, etc.

The compute node will be dedicated to you for your use (30 min).

When you need to access the MIC from a compute node, execute:

\[
\text{mic access: ssh mic0 } \leftarrow \text{we will call this a mic window}
\]

Do the exercises in the order of the listing on the previous page.

Instructions follow:
1 README offload_hello for C offload hello C 1

What you will do: (See next section for INSTRUCTIONS.)

Rather than putting the compile commands in makefiles, we have you type out the commands so that you can see how simple it is to compile for all (most) cases.

1. Look over the code for the cases:

   a.)
   hello.c     i.) Reduction, run on host
   hello.c     ii.) Reduction, run natively on mic
   hello_off.c iii.) Reduction, run on host and offload to mic

   b.)
   hello_omp.c i.) OMP Reduction, run on host
   hello_omp.c ii.) OMP Reduction, run natively on mic
   hello_omp_off.c iii.) OMP Reduction, run on host & offload to mic

2. Compile and Run Cases:

INSTRUCTION DETAILS

1.) Make sure you have 2 windows open on a compute node on your laptop and login to stampede in both. (See instruction on 1st page). In one of the windows ssh to the MIC (ssh mic0) for executing commands directly on the MIC (native execution)—this is your MIC window. In the compute node window and the mic window go to the offload_hello/C directory:

   cd offload_lab/offload_hello/C

2.) The “hello” toy codes do a simple reduction. Compile them.
   To run natively on the MIC you must compile with -mmic. No options are required for offloading.

   icc     hello.c    -o a.out.cpu
   icc -mmic hello.c   -o a.out.mic
   icc     hello_off.c -o a.out.off

   On the host (compute node window) execute:
./a.out.cpu

./a.out.off

./a.out.mic (this will run on the MIC!)

Or, in the MIC window execute the MIC binary:
./a.out.mic

3.) The omp “toy” codes do a simple OpenMP reduction. Compile them:

```bash
icc -openmp hello_omp.c -o a.out.omp_cpu
icc -mmic -openmp hello_omp.c -o a.out.omp_mic
icc -openmp hello_omp_off.c -o a.out.omp_off
```

On the host (compute node window) execute:

```bash
export OMP_NUM_THREADS=16
./a.out.omp_cpu #Run code on CPU -- faster
```

```bash
export MIC_PREFIX=MIC #Set up MIC env with MIC_ prefixed
export MIC_OMP_NUM_THREADS=240 #variables.
./a.out.omp_off #Run offloads on MIC
```

4.) On the MIC (in the mic window*) execute:
(On the MIC the prompt is your present working directory.)
(No need for MIC_ prefix on MIC when executing natively!)

**ON THE MIC* execute:**

```bash
export OMP_NUM_THREADS=244
./a.out.omp_mic
```

5.) While you are on the MIC, kick the tires on BusyBox.
`cat /proc/cpuinfo, etc.`

6.) Rerun cases above with a different number of threads on the host and the MIC.
1 README offload_hello for F90  offload hello F90 1

What you will do:  (See next section for INSTRUCTIONS.)

Rather than putting the compile commands in makefiles, we have you
type out the commands so that you can see how simple it is to
compile for all cases.

1. Look over the Code for the cases:

   a.)
   hello.F90                        i.) Reduction, run on host
   hello.F90                        ii.) Reduction, run natively on mic
   hello_off.F90                    iii.) Reduction, run on host and offload to mic

   b.)
   hello_omp.F90                    i.) OMP Reduction, run on host
   hello_omp.F90                    ii.) OMP Reduction, run natively on mic
   hello_omp_off.F90                iii.) OMP Reduction, run on host & offload to mic

2. Compile and Run Cases:

----------------------------------------------------------------------------

INSTRUCTION DETAILS

1.) Make sure you have 2 windows open on a compute node on your laptop
and login to stampede in both. (See instruction on 1st page). In
one of the windows ssh to the MIC (ssh mic0) for executing commands
directly on the MIC (native execution)—this is your MIC window.
In the compute node window and the mic window go to the
offload_hello/F90 directory:

   cd offload_lab/offload_hello/F90

2.) The “hello” toy codes do a simple reduction. Compile them.
To run natively on the MIC you must compile with -mmic. No
options are required for offloading.

   ifort       hello.F90  -o a.out.cpu
   ifort -mmic hello.F90  -o a.out.mic
   ifort       hello_off.F90  -o a.out.off
On the host (compute node window) execute:

./a.out.cpu
./a.out.off
./a.out.mic  #this will run on the MIC!

Or, in the mic window execute the MIC binary:

./a.out.mic

3.) The omp “toy” codes do a simple OpenMP reduction. Compile them:

ifort -openmp hello_omp.F90 -o a.out.omp_cpu
ifort -mmic -openmp hello_omp.F90 -o a.out.omp_mic
ifort -openmp hello_omp_off.F90 -o a.out.omp_off

On the host (in compute window) execute:

export OMP_NUM_THREADS=16
./a.out.omp_cpu  #Run code on CPU

export MIC_PREFIX=MIC  #Set up MIC env with MIC_ prefixed
export MIC_OMP_NUM_THREADS=240  #variables.
./a.out.omp_off  #Run offloads on MIC

4.) On the MIC (in the mic window*) execute:
(On the MIC the prompt is your present working directory.)
(No need for MIC_ prefix on MIC when executing natively!)

ON THE MIC* execute:

export OMP_NUM_THREADS=244
./a.out.omp_mic

5.) While you are on the MIC, kick the tires on BusyBox.
   cat /proc/cpuinfo, etc.

6.) Rerun cases above with a different number of threads on the host
   and the MIC.
What you will do: (See next section for INSTRUCTION DETAILS.)

You will learn how to use data transfer clauses in the offload directive to minimize data transfer; how to have the compiler report data transfers; and how to instruct the runtime to report data transfers while the code is executing. You will also see how to set KMP_Affinity environment variables for the MIC.

1. Look over the dgemm matrix multiply code (mxm.c or mxm.F90).
   Note, it is only necessary to declare a function as offloadable with the attribute declaration statement and use the offload directive to offload the MKL dgemm routine call. Because we use an `dev_id` for the mic in the target clause, `target(mic:0)`, we force the function to be executed on the MIC. (Since we use pointers in the C code, the storage behind the pointer must be specified in what we call a “data specifiers” -- `inout` here.)

2. Look over the source.affinity file. Note that the number of threads and affinity for the execution is set with the `MIC_OMP_NUM_THREADS` and `MIC_KMP_AFFINITY` variables, respectively.

3. Look over the makefile. Note, only the -mkl loader flags is needed for offloading MKL routines to the MIC! (The -offload-attribute-target=mic is unnecessary, but could be used to automatically make MIC offloadable binaries of all functions in any source file.)

INSTRUCTION DETAILS

1.) Make sure you have 2 windows open on a compute node on your laptop and login to stampede in both. (See instruction on 1st page). In one of the windows ssh to the MIC (ssh mic0) for executing commands directly on the MIC (native execution) – this is your MIC window. In the compute node window and the mic window go to the offload_transfer directory:

   cd offload_lab/offload_transfer/C
   or cd offload_lab/offload_transfer/F90
2.) Once you have looked over the code, make the mxm executable, set the affinity and number of threads (by sourcing the source.affinity file), and run the mxm on the host:

```
make clean
make
source sourceme.affinity
./mxm  #takes 30 seconds.
```

Record the time for the 12,800x12,800 matrix normal execution: __________ (sec.)

Now, change the code so that the a and b matrices are only copied to the MIC and c is only copied back. Use the in and out data_specifier clause.

```
make clean; make
./mxm
```

Record the time for this optimization: __________ (sec.)

By using the in/out clause you should have reduced the time by about .5 seconds. You avoided transferring 3 x 12800*12800*(8 B/word)Bytes.

Determine the Bandwidth between the MIC and the host by dividing the number of bytes by the time.

```
Report Bandwidth: __________ (GB/sec)
```

3.) Look over what data the compiler is moving between the host and the MIC by uncommenting the -opt-report-phase=offload option in the makefile. Clean and remake:

```
make clean
make
```

4.) Now, watch the data traffic to the MIC by setting the OFFLOAD_REPORT environment variable and rerunning the code:

```
export OFFLOAD_REPORT=2
./mxm
```

```
unset OFFLOAD_REPORT  #turn reporting off when finished here
```
5.) In the mic window (the window you ssh’d into the mic0 from)
execute the `top` command and type “1” (not the quotes) and all
hardware threads will appear. In the compute-node window execute
`.mxm`, and watch the hardware thread (cpu) occupation.

What is the binding pattern for the 120 threads? Fill in the dots.

6.) Experiment with changing the number of threads and affinity to
see how threading and affinity affect the execution time and location
(watch with top):

```bash
Edit sourceme.affinity
CHANGE: MIC_OMP_NUM_THREADS = 120 and/or 60)
        MIC_KMP_AFFINITY = balanced, scatter, compact

Loop over {
    edit sourceme.affinity
    source source.affinity
    ./mxm
}
```

See: www.prace-project.eu/IMG/pdf/Best-Practice-Guide-Intel-Xeon-Phi.pdf
3 README offloadStencil

General:
Rather than putting the compile commands in makefiles, we have you type out the commands so that you can see how simple it is to compile for different modes of computing.

Please review the code sten.F90. It runs the same routine on the host and the mic concurrently with OMP.

Note: The same code is used for both architectures. Developers may apply different optimizations to MIC and host code. One can use use #ifdef's with __MIC__ if different “bits” of code are needed for the host and MIC.

A "signal" clause is used to allow asynchronous offload execution. It uses a variable as a handle for an event.

No optimization (compiler and code) are performed here-- it is just a simple stencil for illustrating the concurrency mechanism.

sten.F90 is a simple code that shows how to compute on the MIC and host concurrently.

The code shows several features of offloading:
 a.) offloading a routine
 b.) persistent data (data stays on the MIC between offloads)
 c.) asynchronous offloading (for host-mic concurrent computing)

The "doit" script shows how to set up the execution environment.

-----------------------------------------------------------------

INSTRUCTION DETAILS

2.) You don't even need a makefile for this:
   (Script doit runs a.out with OPENMP env. vars for host & MIC.)

    ifort -openmp sten.F90
    ./doit offload
the output will report the time of the concurrent execution.
Change the value of "L" (between 1 and 4,000) to change the
distribution of work for the CPU and MIC. See code. (change back)

3.) Even though the code has been programmed for offloading, you can
force the compiler to ignore the offload directives and only run
the code on the host. It is that simple, just use the
"-no-offload" option. See environment details for host execution
in the doit script. (Script “doit host” runs a.out with OpenMP
env. vars for the E5.)

ifort -openmp -no-offload sten.F90
./doit host

4.) When compiling you can have the compiler report on offload
statements and MIC vectorization with the following option:

ifort -openmp -offload-option,mic,compiler,"-vec-report3 -O2" \
-opt-report-phase:offload sten.F90

5.) If you want to see what is going on in the offload regions during
execution, set the OFFLOAD_REPORT to a level of verbosity {1-5).
E.g.

ifort -openmp sten.F90
export OFFLOAD_REPORT=2
./doit

#IMPORTANT when finished with Ex. 3.

unset OFFLOAD_REPORT #turn of reporting

6.) Having fun:

Try adjusting the number of MIC threads using the “mic” option
with the doit script. For this UNOPTIMIZED code, what is the
sweet spot for the thread count (balanced affinity)—total time.

Try a native execution:
How would you compile the sten.F90 code? (Hint:
can you combine the -no-offload and -mmic?) Compile the code
for native execution and run it directly on the MIC.