Lab MIC Experiments 6/16/13
TACC

#	pg.	Subject	Purpose	directory
1	3 5	Offload, Begin (C) (F90)	Compile and Run (CPU, MIC, Offload)	offload_hello
2	7	Offload, Data	Optimize Offload Data Transfers	offload_transfer
3	11	Offload, Async	OMP Concurrent CPU+MIC Execution	offload_stencil
4	12	MPI	MPI CPU+MIC Execution	mpi
5	14	Performance	Measurement of Single Prec. FLOPS/sec	flops
6	17 19	Vectors Alignment	Vector/novector and alignment	vector align
7	20	Threads	OMP team creation and barrier overheads	thread_overhead

Before you begin:

login & untar: **ssh <*my_login_name*>@stampede.tacc.utexas.edu**
cd
tar –xvf ~train00/mic_offload.tar

For accessing a compute node interactively through 1 or 2-different window(s) use these instructions:

From one terminal window interactively access a compute node, execute:

ssh <my_login_name>@stampede.tacc.utexas.edu

then execute:

```
idev                    #we will call this an idev window
...
                        #you will get a new prompt
```

Your new prompt is your interactive compute node name==idev_node

e.g. prompt may be: `c557-001$` idev_node is c557-001 in this case.

You use this window for executing MPI code with ibrun—it has the right environment. Also, use it for editing, compiling, etc.
The compute node will be dedicated to you for your use (1 hour).

From another terminal window execute:

ssh <my_login_name>@stampede.tacc.utexas.edu

then execute

```
ssh <idev_node>        # gets you to the compute node
```

then execute:

```
ssh mic0               # we will call this the 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. Open 2 windows on your laptop and login to stampede in both. Execute **idev** in one window to get a compute node (idev window). Once you get an interactive compute node, in the other window ssh to the compute node (idev_node), and then ssh to the MIC (mic0)for executing commands directly on the MIC (native execution)—this is your MIC window.

2. 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

3. Compile and Run Cases:

INSTRUCTION DETAILS

1.) If you don't have an interactive session on a compute node, create one with idev and go to the offload_hello directory. Also create a mic window for executing on the MIC in part 4.

idev #in one window	ssh <*idev_node*> #in other window
...	ssh mic0
cd mic/offload_hello/C	cd /mic/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 (idev 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:

```
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 (idev window1) execute:

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

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 (mic window) execute:
(No need for MIC_ prefix on MIC when executing natively!)

```
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.) Change the 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. Open 2 windows on your laptop and login to stampede in both. Execute **idev** in one window to get a compute node (idev window). Once you get an interactive compute node, in the other window ssh to the compute node (idev_node), and then ssh to the MIC (mic0 for executing commands directly on the MIC (native execution)—this is your MIC window. Go to the offload_hello/C (F90) directory.

2. 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

3. Compile and Run Cases:

INSTRUCTION DETAILS

1.) If you don't have an interactive session on a compute node, create one with idev and go to the offload_hello directory. Also create an mic window for executing on the MIC in part 4.

idev #in one window	ssh <*idev_node*> #in other window
...	ssh mic0
cd mic/offload_hello/F90	cd /mic/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 (idev 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 (idev 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 (window2) execute:
(No need for MIC_ prefix on MIC when executing natively!)

```
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.) Change the number of threads on the host and the mic.

2 README offload_transfer offload transfer 2

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. Open 2 windows on your laptop and login to stampede in both. Execute **idev** in one window to get a compute node (idev window). Once you get an interactive compute node, in the other window ssh to the compute node (idev_node), and then ssh to the MIC (mic0 for executing commands directly on the MIC (native execution)—this is your MIC window Go to the offload_transfer directory: mic/offload_transfer/C or ...F90.

2. 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 **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.)

3.) 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.

4.) Look over the makefile. Note, only the –offload-attribute-target=mic and –mkl loader flags are needed for offloading MKL routines to the MIC!

INSTRUCTION DETAILS

1.) If you don't have an interactive session on a compute node, create one with idev and go to the offload_transfer/C or …/F90 directory. Also create a mic window for executing on the MIC in part 5.

```
idev #in one window          ssh <idev_node> #in other window
...                          ssh mic0
cd mic/offload_transfer      cd /mic/offload_transfer  #/C or /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 source.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**.
Use the ***in*** data_specifier clause.

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

Record the time for this **optimization:** _____ (sec.)

By using the ***in*** clause you should have reduced the time by about .5 seconds. You avoided transferring 2 x 12800*12800*8B/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 quotes) and all hardware threads will appear. In the idev window execute mxm, and watch the hardware thread (cpu) occupation.

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

```
Edit source.affinity   (MIC_OMP_NUM_THREADS – 120 and/or 60)
                       (MIC_KMP_AFFINITY – balanced, scatter, compact)
Loop over {
  edit ...
  source  source.affinity
  ./mxm
}
```

See: http://www.prace-ri.eu/Best-Practice-Guide-Intel-Xeon-Phi

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 all cases.

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.

!!!* Make sure you execute module swap at the end ***!!!**

1.) 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.

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

```
module swap intel intel/13.0.1.117    #just once for ex. 3
ifort -openmp sten.F90
./doit
```

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.

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 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.
module swap intel/13.0.1.117 intel  #swap back to default
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.

In this lab you will execute a simple MPI reduction
on a single Stampede node:

Natively on the E5 (Sandy Bridge) CPUs
Natively on the Phi(MIC) coprocessor
Across the CPU and the MIC.

We do not include a makefile so that you can exercise the commands yourself. The code is not optimized nor compiled for performance--it is meant to be instructional.

*****!!! Make sure you execute source unsource.mpi at the end !!! *****

1 If you don't have an interactive session on a compute node, create one with idev and go to the mpi directory. Also create a mic window for executing on the MIC in part 7.

```
idev #in one window        ssh <idev_node> #in other window
...                        ssh mic0
cd mic/mpi                 cd /mic/mpi
```

3.) Source a source.mpi file that changes to the Intel MPI (IMPI) and sets up all of the env. vars.

```
source source.mpi  #changes to impi MPI and set env. vars.
```

4.) From the single source, compile an E5 and Phi version of the program:

```
    mpif90 -mmic mpi_reduction.f90 -o a.out.mic
or
    mpicc  -mmic mpi_reduction.c   -o a.out.mic

    mpif90       mpi_reduction.f90 -o a.out.cpu
or
    mpicc        mpi_reduction.c   -o a.out.cpu
```

5.) Run the mpi code with 4 tasks on the CPUs:

```
    mpiexec.hydra -n 4 -host localhost ./a.out.cpu
```

6.) Run the mpi code with 8 tasks on the MIC:

```
mpiexec.hydra  -n 8 -host mic0 ./a.out.mic
```

7.) IF YOU HAVE TIME, try this on the MIC
in your mic window execute:

```
cd mic/mpi
source source.mpi        #very important: sets MIC path
mpiexec.hydra -n 2 ./a.out.mic
```

8.) Now launch an execution on the CPUs and the MIC (E5 and the PHi) **in your idev window** with the command:

```
mpiexec.hydra -n 4 -host localhost ./a.out.cpu : \
              -n 8 -host mic0      ./a.out.mic
```

9.) OR try the ibrun.sym command. You must execute these commands in your idev window, it has the right MPI environment for ibrun.

For symmetric run you need to set env. vars. for MIC and the host.

When you accessed this compute node interactively, you requested 16 tasks for the host (CPU) with the "-n 16" option in the idev command. We will use this value for the number of tasks on the host for this part of the exercise. (You could change to a smaller value by setting the MY_NSLOTS environment variable.)

For ibrun.sym (which you should use in job scripts when you submit from the login nodes) you specify the number of MIC MPI tasks by setting the MIC_PPN environment variable. Below we launch 16 tasks on the host and 12 on the MIC

```
#export     OMP_NUM_THREADS=<#>  #Only if you have omp code
#export MIC_OMP_NUM_THREADS=<##> #Only if you have omp code


export MIC_PPN=12
ibrun.symm -m ./a.out.mic -c  ./a.out.cpu
```

These commands execute 16 tasks on CPUS 12 on the MIC.

Experiment with changing the number of tasks. NOTE: if the reduction values are not identical, it is because the partial sums are used and roundoff will affect the results.

```
source unsource.mpi   #very important reset back to mvapich2
                      #or just logout
```

5 README flops flops 5

What you will do and learn: (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. Open 2 windows on the login node.
(You can edit/compile on a compute node; no compiling on MIC)
Execute **idev** to acquire an interactive compute node and go to the
flops directory. Create a mic window for the 5th part.

2. Look over the flops codes (C and/or Fortran versions):

```
    serial_nopt.c  *.f90 (not optimized)
     serial_opt.c  *.f90 (optimized)
   threaded_opt.c  *.f90 (Openmp parallelized)
```

You will run the compiled executables directly on the MIC (i.e.,
you will run it natively on the MIC).

Measure the floating point performance for a loop containing:

```
fa[k] = a * fa[k] + fb[k]
```

* The floating point operations use data from the L2 cache.
* Since the reals are 4-byte, one can obtain 32floats/Cycle,
* twice the number for 8-byte reals!

3. Compile and Run Cases, and report the performance:

INSTRUCTION DETAILS

1.) If you don't have an interactive session on a compute node, create one with idev and go to the flops directory. Also create a mic window for executing on the MIC in part 5.

```
idev #in one window      ssh <idev_node> #in other window
...                      ssh mic0
cd mic/flops             cd /mic/flops
```

2.) Look over the compute loop in the code, it is executed many times to acquire an average value with a simple timer. Compile the code for the MIC using the -mmic option. (The default optimization is -O2.)

```
icc      -mmic serial_noopt.c    -o a.out_noopt.mic
icc -O3  -mmic serial_opt.c      -o a.out_opt.mic
icc -O3  -mmic threaded_opt.c    -o a.out_omp.mic -openmp
```

or

```
ifort     -mmic serial_noopt.f90 -o a.out_noopt.mic
ifort -O3 -mmic serial_opt.f90   -o a.out_opt.mic
ifort -O3 -mmic threaded_opt.f90 -o a.out_omp.mic -openmp
```

In a mic window (on MIC) execute:

```
./a.out_noopt.mic    ****This will take ~35 seconds ******
```

What is the Peak Performance of a MIC core for 4-byte data?
hint: Processor Speed (GHz) x FLOPs/Clock_period x cores,
1.1GHz, 16 FLOPs/CP add and 16 FLOPs/CP multiplies

_____TFLOPS (Peak)

What is the performance for the code?

_____TFLOPS (Single-core, non-optimized flops code.)

3.) Review the serial_opt and threaded_opt code (either C or F90). Note the additional optimization directives (and openmp directives).
Next, run the executables:

In the MIC window -- on the MIC:

Execute the optimized serial version of the code natively:

```
./a.out_opt.mic
```

On the MIC (in mic window) execute the OpenMP version natively:

```
export OMP_NUM_THREADS=122
export KMP_AFFINITY=scatter
./a.out_omp.mic
```

What is the performance for this verions?

_____TFLOPS (Serial Version -- a.out_opt.mic)
_____TFLOPS (OpenMP Version -- a.out_omp.mic)

4.) Try the calculations with 61 and 183 threads. Try using -O2.

In the OMP code an outer loop is added, with an offset to the arrays, to provide more work on additional data. This outer loop is parallelized with openmp. Also, data alignment of the data (in the C code), and loop unrolling and loop-count directives allow the compiler to significantly optimize the performance of the loop by a factor greater than 10.

Data alignment often only contributes enhancements up to 15%. For the C code, it allowed the compiler to aggregate the whole inner loop into a set of instructions. Likewise, for the Fortran code the Loop Count & unroll directives provided the same information, in a different form that allows the compiler to create the same compact set of instructions. See the instructor for more details on how to see this with the assembly print options (-S).

Because each iteration of the computation loop is independent, openmp directive are used to execute the code in parallel.

General:

The vector code calculates a simple Riemann sum on a unit cube. It consists of a triply-nested loop. The grid is meshed equally in each direction, so that a single array can be used to hold incremental values and remain in the L1 caches.

When porting to a Sandy Bridge (SNB) and MIC it is always good to determine how well a code vectorizes, to get an estimate of what to expect when running on a system with larger vector units. Remember, the Sandy Bridgge can perform 8 FLOPS/CP (4 Mults + 4 ADDS), while the MIC can do double that!

The function "f" (in f.h) uses by default the default integrand: x*x + y*y + z*z

If you don't have an interactive session on a compute node, create one with idev and go to the vector directory. Also create a mic window for executing on the MIC in part 9.

```
idev #in one window        ssh <idev_node> #in other window
...                        ssh mic0
cd mic/vector              cd /mic/vector
```

1.) Create the executable with the make command (on the host):

```
make
#Note the vectorization.
```

2.) Run the code:

```
./vector
```

3.) You will get the total number of cycles, called clock periods, CPs, required to execute your code.

record the CP for the vector code

The output will look like this
(clock periods): 18864693
To determine the time, divide the clock periods (CPs) by the speed of the core:
2.7x10**9 CPs/Sec on our Sandy Bridges.

4.) Remake the vector executable with
vectorization turned off:

```
make clean
make VEC_OPT=-no-vec
./vector
            record the CPs for the no vector code.
```

5.) Determine the speedup (CPs non-vec/CPs vec).

6.) What is the speedup? _____

7.) Change the integrand (in the f.h file),
and perform the same vec/no-vec calculations
in 1-6 above.

8.) What is the speedup when sin and pow
are used in the integrand? _____

If you don't see any difference in the vectorized and non-vectorized version of an applications, then you should expect any gain in a system (E5/MIC) with has larger vector functional units. It would also be a good time to consider re-thinking an algorithmic strategy for using vector units in the application.

9.) Make a vector and novector executable for the MIC, then ssh to mic0 and execute them natively. Do this on a compute node.

```
make clean
make -f Makefile_mic VEC_OPT=-no-vec
mv vector novector

make clean
make -f Makefile_mic
```

In a mic window:

```
./vector
./novector
```

Note that the ratio between novector/vector is only about as good as the E5. This is because it is necessary to have 2 threads running on a MIC core to have a new instruction issued every clock period. Multi-threading is required for optimal floating point performance on the MIC.

General:

For subroutines and functions, compilers have limited information, and cannot make adjustments for mis-aligned data, because the routines may be in other programming units and/or cannot be inlined. Because this is the usual case, our experimental code is within a subroutine in another file.

In a multi-dimensioned Fortran array if the first dimension is not a multiple of the alignment, the performance is diminished.

e.g.for a real*8 a(15,16,16) array 15*8bytes is not a multiple of 16 bytes for Westmere and 32 bytes for Sandy Bridge.

The stencil updates arrays (in align_sub.F90) of dimensions of 15x16x16. By padding the first dimension to 16, addition performance can be obtained. In the experiment we make sure the arrays are within the cache for the timings.

The timer captures time in machine clock periods from the rdtsc instruction. The way it is used here, it is accurate within +/- 80 CPs. But timings may have a wider range for conditions we cannot control.

You will measure the performance with/without alignment in the first dimension of a fortran multi-dimensional array. You will also see how inter-procedural optimization can be as effective as padding.

Instructions: cd to mic/align directory after creating an idev window.

1.) Build the code (execute "make"),
and run the code (./align) with
15 as the leading dimension and
record the time.

2.) Change the leading dimension to from 15 to 16,
rebuild and record the time.
By what percent has the performance increased?

3.) Try putting -ipo in the FFLAGS.
rebuild and record the time.
What has happened????

4.) Advanced users should try building a mic version
(compile with –mmic) and execute it natively on the MIC.

General:

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. Open 2 windows on the login node.
(You can edit/compile on a compute node; no compiling on MIC)
Execute **idev** to acquire an interactive compute node and go to the thread_overhead directory. Create a mic window for the 5th part

2. Look over the overhead.c code, and the scan scripts:

overhead.c	(Times new and 2nd parallel, and barrier regions)
scan.cpu	(Runs 1-16 threads, and prints timings.)
scan.mic	(Runs 1-244 threads, and prints timings.)
Makefile	Makes mic and cpu executables (MAC=cpu or MAC=mic)

3. Make a CPU and MIC version of overhead.c.

4. Run scan.cpu on the compute node.
5. Run scan.mic on the MIC

INSTRUCTION DETAILS

1.) If you don't have an interactive session on a compute node, create one with idev and go to the thread_overhead directory. Also create a mic window for executing on the MIC in part 4.

idev #in one window	ssh <*idev_node*> #in other window
...	ssh mic0
cd mic/thread_overhead	cd /mic/thread_overhead

2.) Create overhead.cpu and overhead.mic in the **idev window**:

```
make clean; make MAC=cpu
make clean; make MAC=mic
```

3.) In **idev window** (on the host) execute:

```
C559_001$ ./scan.cpu

1-16 threads:
Thread Creation: ____ to ____ milliseconds
Thread Revival : ____ to ____ microseconds
Thread Barrier : ____ to ____ microseconds
```

4.) In **mic window** (on the MIC) execute:

```
./scan.mic    (ON MIC -- prompt is "~/mic/thread_overhead $ "

1-240 threads:
Thread Creation: ____ to ____ milliseconds
Thread Revival : ____ to ____ microseconds
Thread Barrier : ____ to ____ microseconds
```

Experiment with different KMP_AFFINITY settings.