Lab: MIC Native Execution

Summary

- What you will learn about
  - Evaluating and Analyzing vector performance.
  - Controlling Affinity, visualizing thread distribution.

- What you will do:
  - Determine the extent of total code vectorization:
    1st run with vectorize options,
    2nd run with vectorization turned off.
  - Use the vector reports to analyze loop dependencies (reasons for not vectorizing) and alignments.
  - Learn how to set affinity and observe location of threads.

Getting started

Connect to Stampede:

```
$ ssh username@stampede.tacc.utexas.edu
```

Extract the lab to your account:

```
$ tar xvf ~train00/mic_native.tar
```

Change to the lab directory:

```
$ cd ./native
```

Obtain an interactive session in Stampede:

```
$ idev
```

The **idev** command will start an interactive session on a compute node and export the X11 display.

Exercise 1 – Vector Performance

Compile **vector.c** as a native MIC application:

```
$ icc -openmp -O3 -mmic ./vector.c -o vec.mic
```

And also as a MIC application but disabling vectorization:

```
$ icc -openmp -O3 -mmic -no-vec ./vector.c -o novec.mic
```

Run both executables from the host (they will be automatically executed on the MIC) and take note of the timing difference.

How much speedup comes from the vectorization?

Does this make sense given what you have learned about the MIC architecture?

Exercise 2 – Vectorization Report

Let's get some information about the vectorization in this example code.

Compile the code again, but add the **-vec-report5** option to the compilation line.

There will be some lines in the code which are not vectorizable, but in some cases (line 34) it is not clear why. Can you use a higher vector report level to find out more?

```
$ icc -O3 -openmp -vec-report5 -mmic ./vector.c -o vec.mic
./vector.c(33): (col. 2) remark: loop was not vectorized: loop was transformed to memset or memcpy.
./vector.c(34): (col. 33) remark: loop was not vectorized: statement cannot be vectorized.
./vector.c(35): (col. 33) remark: loop was not vectorized: statement cannot be vectorized.
./vector.c(37): (col. 9) remark: loop was not vectorized: existence of vector dependence.
./vector.c(37): (col. 35) remark: vector dependence: assumed FLOW dependence between y line 37
      and y line 37.
./vector.c(44): (col. 2) remark: loop was not vectorized: not inner loop.
```

Exercise 2 – Vectorization Report

The **–vec-report6** option provides lots of additional information, including alignment:

```
$ icc -O3 -openmp -vec-report6 -mmic ./vector.c -o vec.mic
```

./vector.c(33): (col. 2) remark: loop was not vectorized: loop was transformed to memset or memcpy.

./vector.c(34): (col. 33) remark: loop was not vectorized: statement cannot be vectorized.

./vector.c(34): (col. 33) remark: vectorization support: **call to function rand cannot be vectorized**.

…

./vector.c(37): (col. 9) remark: loop was not vectorized: existence of vector dependence.

./vector.c(37): (col. 35) remark: vector dependence: assumed FLOW dependence between y line 37 and y line 37.

…

./vector.c(47): (col. 4) remark: vectorization support: **reference z has aligned access**.

./vector.c(46): (col. 3) remark: vectorization support: unroll factor set to 8.

./vector.c(46): (col. 3) remark: LOOP WAS VECTORIZED.

./vector.c(44): (col. 2) remark: loop was not vectorized: not inner loop.

This indicates that the reason the loop in line 34 of the code can't be vectorized is because it calls an external function, in this case **rand**.

Exercise 3 – Affinity Settings

Compile a native version of the **vector_omp.c** code to use in this example:

```
$ icc -openmp -O3 -mmic ./vector_omp.c -o vec_omp.mic
```

Run the **vec_omp.mic** executable using 4 OpenMP threads and different affinity settings.

Use the KMP_AFFINITY variable and the **compact/scatter/balanced** and **verbose** settings as described in the lectures. For example, to run the case using compact affinity use:

```
$ export MIC_ENV_PREFIX=MIC
$ export MIC_OMP_NUM_THREADS=4
$ export MIC_KMP_AFFINITY=compact,granularity=fine,verbose
$ ./vec_omp.mic
```

Write down the timings and the processor number to which each thread is bound.

Do the results make sense given what you have learned?

Exercise 3 – Affinity Settings

Your outputs should look similar to the following:

```
$ export MIC_OMP_NUM_THREADS=4
$ export MIC_KMP_AFFINITY=compact,granularity=fine,verbose
$ ./vec_omp.mic
.... <lots of KMP_AFFINITY information output>…
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {1}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {2}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {3}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {4}

$ export MIC_KMP_AFFINITY=balanced,granularity=fine,verbose
$ ./vec_omp.mic
... <lots of KMP_AFFINITY information output>...
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {1}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {5}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {9}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {13}
```

Note that the **scatter** and the **balanced** settings should provide the same thread distribution until more than one thread per core is used.

Exercise 4 – Thread Location (`top`)

- You can observe the location of threads on the MIC with the standard Unix **`top`** utility or the Intel **`micsmc`** utility.
- For **`top`**, you will need to open up another window directly on the MIC:

```
$ ssh <username>@stampede.tacc.utexas.edu
$ ssh <compute_node>
$ ssh mic0
$ extend window, reduce font size & execute:  resize
$ ssh mic0
$ top       # press the 1 key to see the threads
```

In your original idev window create a MIC *`load`* binary, ssh over to the MIC and execute with different affinity options. Watch top in the other window.

```
$ icc -O3 -openmp -mmic load.c -o load.mic
$ ssh mic0
$ cd native
$ export OMP_NUM_THREADS=60
$ export KMP_AFFINITY=compact
$ ./load.mic        #watch top, execute ^C to quit
$ export KMP_AFFINITY=scatter
```

/load.mic #watch top, execute ^C to quit

Part 4 – Thread Location (micsmc)

- Unlike top, **micsmc** is run on the compute node (it is not available on the MIC). It has command line (text) and GUI interfaces.
- Linux/Mac laptops will use their default X11 display manager; windows systems must run an X display manager (e.g. Xming).
- You will need to use X forwarding through all of your connections, so you should logout and login using the –Y option.

```
$ ssh -Y <username>@stampede.tacc.utexas.edu
$ idev      #the session will "X forward" to the node
            # when connected, run micsmc in the background.
c559-100$ /opt/intel/mic/bin/micsmc-gui
```

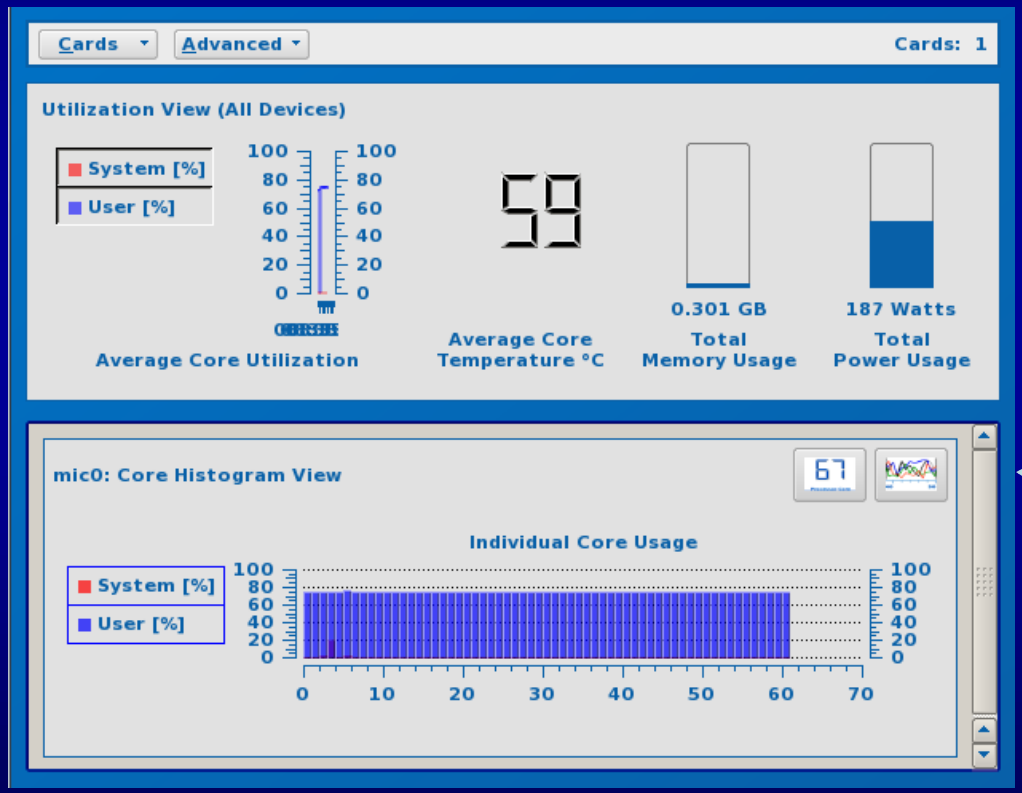
If the display is too unresponsive (through wireless), use the command line:

```
c559-100$ watch -n 3 /opt/intel/mic/bin/micsmc -c
```

This is the compute node IP, ssh to this node in another window, see next page. It is different for each idev session.

Part 4 – Thread Location (micsmc)

- When the smc display appears only 1 panel will be visible. To observe the core usage histogram, select “show all” under the Cards tab (Cards→Show all) and a 2nd panel will appear; next click on the the “core histogram” icon in the new panel if it isn’t already displayed.



Core Histogram Icon

Part 4 – Thread Location (micsmc)

- In another window login to Stampede , ssh to the compute node, and then ssh into the MIC. Execute the *load.mic* binary with different Affinities, and watch the behavior in the **micsmc** display.
-

```
$ ssh <username>@stampede.tacc.utexas.edu
$ ssh <compute_node>     #access to the compute node
$ ssh mic0               #access to MIC coprocessor
$ cd native

$ export OMP_NUM_THREADS=60
$ export KMP_AFFINITY=compact
$ ./load.mic     #watch micsmc, execute ^C to quit

$ export KMP_AFFINITY=scatter
$ ./load.mic     #watch micsmc, execute ^C to quit
                  # Why is the load only 25%/core?
```