Intel Xeon Phi Lab - Fortran

Preliminary:

- Don't forget to load the intel module and (eventually) the Intel MPI module too.
```
module load intel
module load mkl   (if required)
source $INTEL_HOME/bin/compilervars.sh intel64
```

Exercise 1

- Copy `omp_offload_start.F90` to `omp_offload.F90`
- Edit `omp_offload.F90` and add code to offload the OpenMP section and to offload the test to check whether or not the code is running on the coprocessor
- Compare `omp_offload.F90` to `omp_offload_ours.F90` to make sure you got everything
- Make sure that the number of threads is unconstrained (unset OMP_NUM_THREADS)
- Build the result for host-only and check the vectorization messages
```
   ifort  -vec-report=3  -openmp -no-offload omp_offload.F90
   main.F90
```
- Build the result for offload and note compare the vectorization messages with the case of the host compilation
```
   ifort -vec-report=3 -openmp omp_offload.F90 main.F90
```
- Run the result with different numbers of threads on the coprocessor so that you can see the scaling
- What sort of scaling do you see?

Exercise 2

- Make a copy of `mCarlo_offload_start.F90`:
```
cp mCarlo_offload_start.F90 mCarlo_myoffload.F90
```
- Add code to offload the "`do_calculation`" subroutine and write code in order to test whether or not the code is running on the coprocessor. The code to be offloaded was placed in a subroutine to simplify the creation of the streams on the coprocessor rather than on the processor.
- Build the result:
```
ifort -mkl  -openmp mCarlo_myoffload.F90
```
- It could happen that for the last recent of the compiler, the VSL_METHOD_DGAUSSIAN_BOXMULLER2

is no longer present. You can replace it with the new method name: VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2
- Compare your result to `mCarlo_offload_ours.F90`

Exercise 3

“Native” Intel® Xeon PhiTM coprocessor applications treat the coprocessor as a standalone multicore computer. Once the binary is built on your host system, it is copied to the “filesystem” on the coprocessor along with any other binaries and data it requires. The program is then run from the ssh console. (On Cineca machines you can reach a MIC card typing ssh $HOSTNAME-mic0 or $HOSTNAME-mic1)

- Build our sample application with the –mmic flag. The sample code is a single-file version of the matrix multiply code we previously worked with:
  ```
  ifort –mmic –vec-report=3 –openmp omp_offload_native.F90
  ```
- Now upload the result (a.out) to the coprocessor
  ```
  scp a.out $HOSTNAME-mic0:a.out
  ```
- Connect with ssh and run a.out:
  ```
  > ssh $HOSTNAME-mic0
  ~ # ./a.out 2048
  ```
- As you noted from the error message, we are missing the OpenMP runtime library needed to run this application. So, when you're logged on the Xeon Phi, export the proper library directory:
  ```
  export LD_LIBRARY_PATH= \
  /cineca/prod/compilers/intel/cs-xe-2013/binary/composerxe/lib/mic/
  ```
- Try to run again.

Exercise 4

Code of any complexity tends to do things in stages. This can complicate things when multiple stages need to execute on a coprocessor, and you need the results from one stage to persist until the next call. In this section, we will explore how this is done.

- Take a look at `omp_offload_ours.F90` and note how the data transfer and work happen in a single offload call. Let us artificially change this into three stages and observe what happens.
- Start with `omp_3stageoffload_nopersist.F90`. Build it and observe what happens when it runs:
  ```
  ifort -O3 omp_3stageoffload_nopersist.F90 -o mmul_nopersist
  ./mmul_nopersist 2048
  ```
- You will see an error message.

- Now compare `omp_3stageoffload_nopersist.F90` to `omp_3stageoffload_persist.F90`
- Build and run `omp_3stageoffload_persist.F90`:
```
ifort -O3 omp_3stageoffload_persist.F90 -o mmul_persist
./mmul_persist 2048
```
- Did you get the expected result?
- Make sure you understand how the alloc_if, free_if, and nocopy qualifiers are used in the offload statement. Refer to the compiler reference manual.

Exercise 5

Codes often operate on blocks of data which require the data block to be moved to the coprocessor at the start of the computation and back to the host at the end. Such codes benefit by the use of asynchronous data transfers where the coprocessor computes one block of data while another block is being transferred from the host. Asynchronous transfers can also improve performance for codes requiring multiple data transfers between the host and the coprocessor.

- Take a look at do_offload subroutine in `async_start.F90` and notice how the two arrays are processed one after the other using offload statements.
- Change this code so that you transfer one array while the other one is computing. Modify the `do_async` function to use asynchronous data transfers.
- Build and run the program.
```
ifort –o async.out async_start.F90
./async.out
```
- Notice that the `do_async` function is faster compared to the `do_offloads` function.
- Make sure you understand how the signal and wait qualifiers are used in the offload statements. Refer to the compiler reference manual for more details.

Intel Xeon Phi Lab - C

Preliminary:

- Don't forget to load the intel module and (eventually) the Intel MPI module too.
```
module load intel
module load mkl   (if required)
source $INTEL_HOME/bin/compilervars.sh intel64
```

Exercise 1

- Copy `omp_offload_start.cpp` to `omp_offload.cpp`
- Edit `omp_offload.cpp` and add code to offload the OpenMP section and to offload the test for whether or not the code is running on the coprocessor
- Compare `omp_offload.cpp` to `omp_offload_ours.cpp` to make sure you got everything
- Make sure that the number of threads is unconstrained (unset OMP_NUM_THREADS)
- Build the result for host-only and check the vectorization messages
```
icc  -qopt-report-phase=vec  -openmp -qno-offload omp_offload.cpp
main.cpp
```
- Build the result for offload and note how the vectorization message change
```
ifort -qopt-report-phase=vec -openmp omp_offload.cpp main.cpp
```
- Check and understand the different optimization reports.
- Run the result with different numbers of threads on the coprocessor so that you can see the scaling
- What sort of scaling do you see?

Exercise 2

- Make a copy of `mCarlo_offload_start.cpp`:
```
cp mCarlo_offload_start.cpp mCarlo_myoffload.cpp
```
- Add code to offload the OpenMP section at line 64 and write code in order to test whether or not the code is running on the coprocessor. Note how we had to move the VSLStreamStatePtr definitions within the offload statement block (compare to `mCarlo_offload_ours.cpp`).
- Build the result:
```
icc: -mkl -openmp mCarlo_myoffload.cpp
```
It could happen that for the last recent of the compiler, the VSL_METHOD_DGAUSSIAN_BOXMULLER2

is no longer present. You can replace it with the new method name: VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2

- Compare your result to `mCarlo_offload_ours.cpp`

Exercise 3

“Native” Intel® Xeon PhiTM coprocessor applications treat the coprocessor as a standalone multicore computer. Once the binary is built on your host system, it is copied to the “filesystem” on the coprocessor along with any other binaries and data it requires. The program is then run from the ssh console. (On Cineca machines you can reach a MIC card typing ssh $HOSTNAME-mic0 or $HOSTNAME-mic1)

- Build our sample application with the –mmic flag. The sample code is a single-file version of the matrix multiply code we previously worked with:
```
icc –mmic –vec-report=3 –openmp omp_offload_native.cpp
```
- Now upload the result (a.out) to the coprocessor
```
scp a.out $HOSTNAME-mic0:a.out
```
- -Connect with ssh and run a.out:
```
> ssh $HOSTNAME-mic0
~ # ./a.out 2048 1
```
- As you noted from the error message, we are missing the OpenMP runtime library needed to run this application. So, when you're logged on the Xeon Phi, export the proper library directory:
```
export LD_LIBRARY_PATH= \
/cineca/prod/compilers/intel/cs-xe-2013/binary/composerxe/lib/mic/
```

- Now go to the ssh window and try to run again on the coprocessor
```
./a.out 2048 1
```

Exercise 4

Code of any complexity tends to do things in stages. This can complicate things when multiple stages need to execute on a coprocessor, and you need the results from one stage to persist until the next call. In this section, we will explore how this is done.

- Take a look at `omp_offload_ours.cpp` and note how the data transfer and work happen in a single offload call. Let us artificially change this into three stages and observe what happens.

- Start with `omp_3stageoffload_nopersist.cpp`. Build it and observe what happens when it runs:
  ```
  icc -O3 -openmp omp_3stageoffload_nopersist.cpp -o mmul_nopersist
  ./mmul_nopersist 2048
  ```
- You will see an error message.
- Now compare `omp_3stageoffload_nopersist.cpp` to omp_3stageoffload_persist.F90
- Build and run `omp_3stageoffload_persist.cpp`:
  ```
  icc -O3 -openmp omp_3stageoffload_persist.cpp -o mmul_persist
  ./mmul_persist 2048
  ```
- Did you get the expected result?
- Make sure you understand how the alloc_if, free_if, and nocopy qualifiers are used in the offload statement. Refer to the compiler reference manual.

Exercise 5

Codes often operate on blocks of data which require the data block to be moved to the coprocessor at the start of the computation and back to the host at the end. Such codes benefit by the use of asynchronous data transfers where the coprocessor computes one block of data while another block is being transferred from the host. Asynchronous transfers can also improve performance for codes requiring multiple data transfers between the host and the coprocessor.

- Take a look at `do_offload` function in `async_start.cpp` and notice how the two arrays are processed one after the other using offload statements.
- Change this code so that you transfer one array while the other one is computing. Modify the `do_async` function to use asynchronous data transfers.
- Build and run the program.
  ```
  icc –o async.out async_start.cpp
  ./async.out
  ```
- Notice that the `do_async` function is faster compared to the `do_offloads` function.
- Make sure you understand how the signal and wait qualifiers are used in the offload statements. Refer to the compiler reference manual for more details.