Intel® Xeon Phi™ Coprocessor
Lab Instructions
C ++ Version

Introduction

Goal

This document will help you get started writing code and running applications on a development platform (host) that includes the Intel® Xeon Phi™ coprocessor through a series of simple, self-guided labs. It demonstrates basic build and run procedures, and covers a few basic optimization techniques.

Before you Start

- Please read through the "Intel® Xeon Phi™ coprocessor Quick Start Developer's Guide".
- This document assumes that the development machine is Tornado cluster, available by tornado.hpc.susu.ac.ru via ssh.
- Please read **Cluster Notes** before start.
- The labs in this document require the following Intel® Software to be installed on the Development Platform:
    - o Intel® Manycore Platform Software Stack. (MPSS)
    - o Intel® C++ Composer XE 2013 or higher.
    - o Intel® Vtune™ Amplifier XE 2013 or higher.

Useful References

For more information on Programming and Compiling for Intel® Many Integrated Core (Intel® MIC) architecture, please refer to:

http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture

.

Lesson 1: Converting Code for Offload

Goal

You will learn how to convert pure host code into a heterogeneous form that runs partially on the host and partially on the Intel® Xeon Phi™ coprocessor using the explicit offload model.

Useful References

- Compiler reference manual :
  C/C++:
  https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/

- Example code showing various subtleties of the offload syntax:
  C/C++: /opt/software/intel/composerxe/Samples/en_US/C++/mic_samples

Lab

Let's Revise Offload using Explicit Copies

	C/C++ Syntax	Semantics
Offload pragma	#pragma offload	Allow next statement to execute on Intel® Xeon Phi™ coprocessor or host CPU
Keyword for variable & function definitions	__attribute__((target(mic)))	Compile function for, or allocate variable on, both CPU and Intel® Xeon Phi™ coprocessor
Entire blocks of code	#pragma offload_attribute(push, target(mic)) . . . #pragma offload_attribute(pop)	Mark entire files or large blocks of code for generation on both host CPU and Intel® Xeon Phi™ coprocessor
Inputs	in(var-list modifiers)	Copy from host to coprocessor
Outputs	out(var-list modifiers)	Copy from coprocessor to host

Let's get some simple matrix multiply code running on the Intel® Xeon Phi™ coprocessor and see what happens.

Set up the compiler environment:

```
#source /opt/software/intel/composerxe/bin/compilervars.sh intel64
```

Copy omp_offload_start.cpp to omp_offload.cpp:

```
#cp omp_offload_start omp_offload.cpp
```

Add code to offload the OpenMP section and to offload the test for whether or not the code is running on the coprocessor. Check the references above in case you forget the syntax.

Compare omp_offload.cpp to omp_offload_ours.cpp to make sure you got everything.

```
#diff omp_offload.cpp omp_offload_ours.cpp
```

Make sure the number of OpenMP threads is unconstrained:

```
# unset[env] OMP_NUM_THREADS
```

Build the result for host-only and note the vectorization messages:

```
# icc -vec-report3 -openmp -no-offload omp_offload.cpp main.cpp
```

Build the result for offload and note how the vectorization messages change:

```
# icc –vec-report3 –openmp omp_offload.cpp main.cpp
```

You probably saw the number of messages increase. This is because your single compile command is actually causing two compilations to occur under the covers: one for the host system and one for the coprocessor. Each produces its own messages and each may reach different optimization decisions. All messages containing *MIC* are caused during the compilation for the coprocessor.

Run the result with different numbers of threads on the coprocessor so that you can see the scaling:

```
# export  MP_NUM_THREADS=<number>
#./a.out 2048
```

Number of threads	Runtime (seconds)
1	

2	
16	
32	
64	
93	
128	
204	

What sort of scaling do you see?

Now let's try a slightly more advanced example of offloading.

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 65 and code 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
```

If you get a message complaining that omp_get_max_threads() is not defined for offload, look up the **#pragma offload_attribute(push,target(mic))** pragma in the offload compiler guide and modify your code to make those warnings go away.

Now run.

```
# ./a.out
```

Compare your result to **mCarlo_offload_ours**.cpp

Bonus

If there is time, experiment with the **H_TIME** and **H_TRACE** environment variables and try to interpret their output.

What we learned

- How to use **#pragma offload target(mic) in(var:elements)** to send data to the card

- How to use **__attribute__((target(mic))** to mark a function for compilation on the Intel® Xeon Phi™ coprocessor as well as the host (or to define a variable on both architectures)
- How to use the **__MIC__** preprocessor variable to mark code as executing only on the host or only on the coprocessor.
- How to use **#pragma offload_attribute(push,target(mic))** and **#pragma offload_attribute(pop)** to mark large bodies of code for offload
- (optional) How to monitor what is happening during the offload process using **H_TRACE** and **H_TIME**.

Lesson 2: Building and Running a "Native" Intel® Xeon Phi™ Coprocessor Application

Goal

You will learn how to build and run applications that are intended purely for use on the Intel® Xeon Phi™ coprocessor.

Useful References

- Intel® Xeon Phi™ coprocessor Quick Start Developer's Guide (found on http://software.intel.com/mic-developer).
- Read Cluster Notes before.

Lab

"Native" Intel® Xeon Phi™ 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.

Build our sample application with the **–mmic** flag (a single-file version of the matrix multiply code – you are welcome to inspect it first):

```
#icc –mmic –vec-report3 –openmp omp_offload_native.cpp
```

Note that your home folder is a NFS shared folder and copy binaries files to the coproccessor not required. Just connect to the coprocessor by ssh.

```
#ssh 'hostname'-mic0
#cd ./<folder with binaries>
# ./a.out 2048 64
```

If you see error message about missing OpenMP runtime library, add path to OpenMP runtime libraries:

```
#export
LD_LIBRARY_PATH=/opt/software/intel/composer_xe_2013.1.117/compiler
/lib/mic:$LD_LIBRARY_PATH
```

Now try to run again on the coprocessor (in the ssh window).

```
~# ./a.out 2048 64
```

Notes:

- You can use sftp, ssh and scp if you want. See Intel® Xeon Phi™ coprocessor Quick Start Developer's Guide
- You can also use the micnativeloadex utility which will automatically copy dependent files and run the program on the coprocessor.
- If your program requires data to run, you will have to copy it to the coprocessor as well, just like you did with the OpenMP* library. And likewise, any data you generate would need to be copied back to the host manually to be used there.
- Any time when you want to run native programm on Xeon Phi™, you can modify $LD_LIBRARY_PATH for set path to dependencies.
- Be sure to clean up any binaries or data you copied to the card or that were generated when your program ran. We have seen numerous cases in which an offloaded application that is close to the system memory limit fails because someone forgot to clean up files copied to the coprocessor when doing native programming.

So, why would you use the "Native" programming model? As you will notice after programming the Intel® Xeon Phi™ coprocessor for a while, using the offload compiler model can introduce a lot of overhead into your runtime if you aren't careful about it. It also hides a lot of the complexity of getting code and data to the coprocessor. But what if you just want to see how fast/slow this coprocessor is without all that overhead or want to have much more control over data movement during optimization? This is when the native programming model might appeal, since it gets you "down to the metal" on the coprocessor with no intermediate layers eating up application time.
In its current implementation all "communication" to and from the coprocessor is via ssh and scp utility, which happens in a human-scale timeframe. There is no way for the program on the coprocessor to communicatıo.e with the host system directly in computer-scale timeframes when doing native programming.

Bonus

Make a “native” version of the Monte Carlo program. You will have to worry about getting the MKL library over as well.

Build on host

```
 #icc -mmic -openmp -mkl mCarlo_myoffload.cpp
```

Connect to mic and run:

```
#  source /opt/software/intel/composerxe/mkl/bin/mklvars.sh mic

 # ./a.out
```

What we learned

How to cross-compile an application for the Intel® Xeon Phi™ coprocessor using the –mmic flag
~~How to transfer this application to coprocessor using scp and run it there from ssh~~

Lesson 3: Data Persistence

Goal

You will become familiar with the offload programming pattern needed to separate data transfer from computation and the reuse of data from one offload call to another.

Useful References

Compiler reference manual :

- C/C++: http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm

- Read Cluster Notes before

- Example code showing various subtleties of the offload syntax:
  C/C++: /opt/intel/composerxe/Samples/en_US/C++/mic_samples

Lab

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.

Revies – Offload using Explicit Copies – Modifiers

Clauses / Modifiers	Syntax	Semantics
Non-copied data	nocopy(var-list modifiers)	Data is local to target
Specify pointer length	length(element-count-expr)	Copy N elements of the pointer's type
Control pointer memory allocation	alloc_if (condition)	Allocate memory to hold data referenced by pointer if condition is TRUE
Control freeing of pointer memory	free_if (condition)	Free memory used by pointer if condition is TRUE
Control target data alignment	align (expression)	Specify minimum memory alignment on target

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 omp_3stageoffload_nopersist.cpp -o mmul_nopersist
#./mmul_nopersist 1024
```

You will see an error message.
Now compare **omp_3stageoffload_nopersist.cpp** to **omp_3stageoffload_persist.cpp**

```
diff omp_3stageoffload_nopersist.cpp  omp_3stageoffload_persist.cpp
```

Build and run **omp_3stageoffload_persist.cpp**:

```
# icc -O3 omp_3stageoffload_persist.cpp -o mmul_persist
#./mmul_persist 1024
```

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.

Bonus
Implement a similar data transfer pattern in another small application

What we learned

How the **`alloc_if`**, **`free_if`**, and **`nocopy`** qualifiers are used to control the allocation and freeing of buffers used on offload statements.

Lesson 4: Asynchronous data transfers

Goal

You will become familiar with the use of asynchronous data transfers needed to overlap the data transfer and computation on the coprocessor.

Useful References

- Compiler reference manual :
  C/C++:
  http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm

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.

Compare **async_start.cpp** to **async_ours.cpp** to make sure you got everything.

Build and run the program.

```
# icc -o async.out async_start.cpp
# ./async.out
```

Notice that **do_async** function is faster compared to **do_offload** 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.

You may have noticed that the above program provided only a small improvement in performance. To get substantial performance improvements, there should be a larger overlap between the data transfers and the computation.

Take a look at **async_advanced.cpp**. Notice how the arrays have broken down into smaller blocks and then processed. Breaking down the array into smaller blocks allows more for over overlap between the data transfers and computation.

Also, observe how only a small portion of the array is transferred over to the coprocessor by using the array notations.
Build and run the code, and observe the performance.

```
# icc -o async_advanced.out async_advanced.cpp
# ./async_advanced.out
```

What we learned

- How to use **#pragma offload_transfer** to start a non-block transfer to the coprocessor.
- How to use **signal** and **wait** qualifiers with the offload statement to ensure completion of data transfers before a compute.
- How to use array notations to transfer a portion of the array to the coprocessor.

Lesson 6: Getting Code to Vectorize

Goal

You will become familiar using and interpreting the vectorization and optimization reports produced by the compiler.

Useful References

- Compiler documentation: C/C++: http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm
- Read Cluster Notes before.

Lab

One important skill to master when using compiler-based auto-vectorization is how to listen to the compiler. This involves using some compiler options that let the compiler tell you about the decisions it makes and the reasons it makes them.

Code that doesn't vectorize

Inspect **serial.cpp**. Now run the following command:

```
#icc -mmic -vec-report3 serial.cpp main.cpp
```

- Did any of the loops vectorize?
- Why not?

There are two variants of this diagnostic message:
1. vector dependence: assumed ANTI dependence between xxxx line <val> and xxxx line <val>
2. vector dependence: assumed FLOW dependence between xxxx line <val> and xxxx line <val>

Cause:
1. ANTI dependence stands for "READ before WRITE" scenario when considering the vector version of a loop. In the below, when the compiler auto-vectorizer tries to vectorize for SSE2 architecture by default, it chooses a vector length of 4 (since data type it operates on is int). But when considering a vector operand instead of scalar operands for this loop, there is an overlap between the input vector and output vector. The overlap is such those overlapped locations are read before latest value is written into them.

2. FLOW dependence stands for "WRITE before READ" scenario when considering the vector version of a loop. In the below, when the compiler auto-vectorizer tries to vectorize for SSE2 architecture by default, it chooses a vector length of 4 (since data type it operates on is int). But when considering a vector operand instead of scalar operands for this loop, there is

an overlap between the input vector and output vector. The overlap is such those overlapped locations are written into before their initial value is read for the computation.

Note: If you like to see lots of diagnostic information, build your entire project with this option. For more terse output, we recommend manually compiling just the files you are trying to optimize with the **vec-report** flag switched on while you try to improve vectorization.

Getting vectorization by following compiler advice

Run the following command:

```
# icc –mmic –guide-vec serial.cpp
```

Look at the messages and advice given by the compiler (look only at the "ALTERNATIVE" suggestion). Do you understand what it is telling you?

Inspect **restrict.cpp**. Does it implement the compiler's suggestions properly? How does it compare to serial.cpp?

Definition: By qualifying a pointer with the restrict keyword, you assert that an object accessed by the pointer is accessed by only that pointer in the given scope

Run the following command:

```
# icc –mmic -restrict -vec-report3 restrict.cpp
```

Did any of the loops vectorize this time?

Run the following command (see -opt-report):

```
# icc –mmic -restrict -opt-report restrict.cpp
```

Can you tell what the compiler did to vectorize the loop. (Hint: Look at the High Level Optimizer Report)?

Changing the compiler's vectorization decisions using pragmas

a) By its nature, the compiler has to be conservative about what it can vectorize. In this case, since serial.cpp and main.cpp are separate source files, it can't tell whether the input arrays are different arrays, overlap, or point to the same arrays, so we need to help it.

- Inspect **pragma.cpp**.How does it compare to **serial.cpp**?

Definition: #pragma ivdep instructs the compiler to ignore assumed vector dependencies. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization.

Now issue the following command:

```
# icc –mmic -vec-report3 pragma.cpp
```

- Did the code vectorize?
- Why?

b) The previous pragmas told the compiler not to make some assumptions that would prevent vectorization of the inner-most loop . The **#pragma simd** directive is quite different in that it tells the compiler that you know this loop will vectorize under all inputs. This is strong stuff, so use it with care.

- Inspect **psimd.cpp**. How does it compare to **serial.cpp**?
-

Definition: The pragma simd enforces vectorization of innermost loops.
Now issue the following command:

```
# icc –mmic -vec-report3 psimd.cpp
```

- Did the code vectorize?
- Why?

Bonus

Apply the same triage procedure to another trivial loop of your choosing

What we Learned

- How to use the **–vec-report** compiler switch to determine which loops and functions are vectorizing
- How to use the **–opt-report** compiler switch to understand some of the ways the compiler transforms your code when compiling it

- How to use the **–guide-vec** compiler switch to get advice from the compiler on how to transform your code so that it will vectorize
-

Lesson 7: Finding Good Offload Candidates

Goal

- Using Loop Profiler, code inspection, and a little math, you will figure out which, if any, of the three provided serial workloads are good candidates for offloading to the Intel® Xeon Phi™ coprocessor.
- Read Cluster Notes before
- Read Intel tools installation instructions

Lab

We need to discover the hot functions and loops in the sample code, and understand the data that are passed to/from those hot functions and loops. Rather than talk about VTune™ Amplifier XE at the moment, we'll use the compiler to do this for us.

Build each example at the –O1 optimization level with compiler profiling turned on.

```
# icc -O1 -profile-functions -profile-loops=all
-profile-loops-report=2 -liomp5 -lpthread common.cpp
lifeserial.cpp -o life

#icc -O1 -profile-functions -profile-loops=all
-profile-loops-report=2 -mkl mCarlo.cpp -o mCarlo

#icc -O1 -no-offload -profile-functions -profile-loops=all
-profile-loops-report=2 main.cpp serial.cpp -o serial
```

Run each program

- ./life virus.dat

Will run for about 30 seconds

- ./mCarlo

Will run for about 20 seconds

- ./serial 1024

Will run for about 5 seconds

When we ran the programs, the compiler generated profiling information for every function and loop it encountered. Let us look at these data.

Look at the resulting xml files, copy files from cluster to your notebook/desktop and open files by **loopprofileviewer** located at **(C:\Program Files (x86)\Intel\Composer XE 2013\bin\)**

Now open each xml file using File/Open in the tool that launched when running previous command. Record the following information from loop profiler and inspection of the code.

Record data for only the loops or functions with the largest overall runtimes (the graph is sorted by self-time, which should correlate to the largest times without resorting).

life

Function	Time	%Self Time	Loop entries	Min Iterations	Avg Iterations	Max Iterations

mCarlo

Function	Time	%Self Time	Loop entries	Min Iterations	Avg Iterations	Max Iterations

serial

Function	Time	%Self Time	Loop entries	Min Iterations	Avg Iterations	Max Iterations

Total Runtime:

- Which functions or loops in each program would be good candidates for offloading to a coprocessor?
  - Life:
  - MonteCarlo:
  - MMul:

Assuming you offloaded the chosen function/loop, how much data would be transferred each time you offloaded work to the coprocessor?

  - Life:
  - MonteCarlo:
  - MMul:

- Assuming you offloaded the chosen function/loop, would it make sense to keep some data on the coprocessor between offload calls?
  - Life:

- MonteCarlo:
- MMul:

- So which of these programs would ***make sense*** to run with a portion offloaded?
- Life:
- MonteCarlo:
- MMul:

What we Learned

- How to gather loop and function information using the –profile-loops compiler option
- How to view/interpret the output from Loop Profiler
- How to reason about what code makes sense to offload to the coprocessor