Intel® Xeon Phi™ Coprocessor
Lab Instructions
Fortran Version

Table of Contents

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", found at http://software.intel.com/mic-developer.

This document assumes that the development platform has been setup and is ready to use. For information on setting up the development platform, please refer to the "Intel® Xeon Phi™ coprocessor Quick Start Developer's Guide"

The labs in this document require the following Intel® Software to be installed on the Development Platform:

Intel® Manycore Platform Software Stack (MPSS)

Intel® Fortran Composer XE 2013 or higher

Intel® C++ Composer XE 2013 or higher

Intel® Vtune™ Amplifier XE 2013 or higher

Useful References

For more information on Programming and Compiling for the Intel® Xeon Phi™ coprocessor, 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:

  Fortran:

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

  C/C++:

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

- Example code showing various subtleties of the offload syntax:

  Fortran:

  `/opt/intel/composerxe/Samples/en_US/Fortran/mic_samples`

  C/C++:

  `/opt/intel/composerxe/Samples/en_US/C++/mic_samples`

Lab

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/intel/composerxe/bin/compilervars.[c]sh intel64
  ```

- Copy `omp_offload_start.F90` to `omp_offload.F90`.
- Open `omp_offload.F90` in your editor of choice.
- 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.F90` to `omp_offload_ours.F90` to make sure you got everything.
- 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:

  ```
  ifort -vec-report3 -openmp -no-offload omp_offload.F90 main.F90
  ```

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

  ```
  ifort -vec-report3 -openmp  omp_offload.F90 main.F90
  ```

- 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:

  csh: `setenv OMP_NUM_THREADS <number>`

  bash: `export OMP_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.F90`:

  ```
  cp mCarlo_offload_start.F90 mCarlo_myoffload.F90
  ```

- Add code to offload the subroutine call at line 61 and code 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
  ```

- If you get a message complaining that the various VSL functions are not defined for offload, look up the `!dec$ options` directive in the Fortran compiler reference. Using this directive, modify your code to declare the contents of mkl.fi as offloadable.
- Now run.
- Compare your result to `mCarlo_offload_ours.F90`

<u>Bonus</u>

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 `!dec$ offload target(mic) in(var:elements)` to send data to the card
- How to use `!dec$ attributes offload:mic :: func_name` to mark a function for compilation for 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 `!dec$ options/offload_attribute_target=mic` and `!dec$ end options` to mark a body of code for offload
- (optional) How to monitor what is happening during the offload process using `H_TRACE` and `H_TIME`.

Lesson 2: Monitoring the Coprocessor

Goal

You will learn how to connect to the coprocessor and monitor its health from within, as well as how to monitor its activity on a per-core basis.

Useful References

- `Use of ssh` and `micsmc` are briefly described in the Intel® Xeon Phi™ Coprocessor Quick Start Developer's Guide, found on http://software.intel.com/mic-developer

Lab

You now have some code running on the coprocessor. Let's play with some of the ways to find out what is happening when your code runs there.

- Gather information about the coprocessor installed on your system:
  - On the host, log in as root
  - Type the following:

    ```
    /opt/intel/mic/bin/micinfo
    ```

  Note: The micctrl utility can be used to tell whether or not the coprocessor needs to be rebooted/started. Check in the *Quick Start Developer's Guide* and use this utility to shut down and then restart your coprocessor, observing what `micinfo` reports in both cases.
- Look at what is running on the system with and without your program running:
  - Open a new terminal window.
  - On the host, log in as root.
  - Run `ssh mic0`
  - Type `top` on the coprocessor in the `ssh` window.
  - In your original host window (NOT the `coprocessor` window), compile `malloc_test.F90`:

    ```
    ifort malloc_test.F90 -o malloc_test
    ```

  - Run the program, varying `numGB` from 1 to 10:

    ```
    ./malloc <numGB>
    ```

  - Observe the memory usage shown by top as you increase numGB.
  - Watch the `host` window for errors when you increase `numGB`.
  - Look at the source code to help you understand this code.
  - Exit from `ssh`.
  - Log out from root.

- Now let us look at an alternative tool we can use to monitor these systems.
  - On the host, log in as root.
  - Start the system management and configuration tool and get familiar with the information it reports:

    ```
    /opt/intel/mic/bin/micsmc &
    ```

        - Now build and run the matrix multiplication code from the last example and watch how `micsmc` responds.
        - Log out from root.
    - **Note**: In some situations we have found that `micsmc` can slow down code running on the coprocessor. Make sure you shut it down before doing any benchmarking or other performance-critical measurements such as running Intel® Vtune™.

<u>What we learned</u>

- How to gather high-level statistics about the coprocessor(s) installed on your host using `micinfo`
- How to connect to the coprocessor and poke around in it using `ssh`
- How to monitor core utilization on a coprocessor using `micsmc`

Lesson 3: 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).

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. (If you do not have an account on the coprocessor, you may need to run this lab as root, putting the files you copy over into /tmp.)

- 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-report3 -openmp omp_offload_native.F90
  ```

- Now upload the result (a.out) to the coprocessor

  ```
  scp a.out mic0:a.out
  ```

- Connect with `ssh` and run `a.out`:

  ```
  > ssh mic0
  ~ # ./a.out 2048
  ```

- As you noted from the error message, we are missing the OpenMP* runtime library needed to run this application. So copy it over to the coprocessor from the host, using the original console window:

  ```
  > scp
     /opt/intel/composerxe/lib/mic/libiomp5.so \
     mic0:libiomp5.so
  ```

- Now go to the `ssh` window and try to run again on the coprocessor

  ```
  ./a.out 2048
  ```

- That's right; it still isn't running because the image activator doesn't have the local directory in its search path by default. So help it out:

  ```
  export LD_LIBRARY_PATH=$ LD_LIBRARY_PATH:~
  ```

- Now try to run again.
- Success!

Notes:

- Use `ssh` to log onto the coprocessor. Use `scp` to copy files to or from 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.
- **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.
- The program /opt/intel/mic/bin/micnativeloadex will attempt to find all the library dependencies for you, then copy your executable and the required libraries over to the coprocessor and run the executable.

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.

Bonus

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

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 4: 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:

  Fortran:

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

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

- Example code showing various subtleties of the offload syntax:

  Fortran:

  `/opt/intel/composerxe/Samples/en_US/Fortran/mic_samples`

  `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.

- 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.

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 5: 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 :
  Fortran:
  http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/fortran-lin/index.htm

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

Lab

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 `omp_ours.cpp` to make sure you got everything.
- 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.

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.F90. Notice how the arrays have broken down into smaller blocks and then processed. Breaking down the array into smaller blocks allows for more 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.

```
ifort –o async_advanced.out async_advanced.F90

./async_advanced.out
```

<u>What we learned</u>

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

Lesson 6: Simultaneous Computation

Goal

You will become familiar with the programming pattern currently needed to implement computation on both the host and the Intel® Xeon Phi™ coprocessor at the same time.

Useful References

- Compiler reference manual :

  Fortran:

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

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

Lab

At present, offload directives `(!dec$ offload)` block on the host until the offloaded code completes on the coprocessor. This causes the host to idle while waiting for the coprocessor--not a good use of resources.

There are scenarios where you may want both the host and coprocessor to be active at once, such as

- Wanting the host to run parallel or different calculations while the coprocessor is in use;
- Wanting to pass input to and receive output from a long-running task on the coprocessor (for example, processing streaming video);
- Dispatching multiple units of work to the coprocessor on behalf of remote machines or dispatching work to other active coprocessors.

A trivial example of the second use case can be found in `simultcompute.F90`, which you can compile using :

```
ifort -openmp simultcompute.F90
```

Study this program to see how it works, and then run it.

Remove the OpenMP tasks to make these tasks run one after another rather than in parallel, and observe any changes in runtime.

Bonus

- See if you can modify the program to do additional work on either the host or the coprocessor, and/or to implement a more interesting (or simple) communication or locking model. For example, take the Monte Carlo or matrix multiply code from the previous lessons and see if you can get the host and coprocessor working at once on different data sets.
- Try re-implementing this sample using Pthreads.

- Experiment with spawning off an independent thread on the coprocessor inside an offload statement, allowing the offload to return, and then reconnecting to that independent thread using a variable marked `!dec$ attributes offload : mic` in a subsequent offload statement on the same host thread.

<u>What we learned</u>

- How to use multi-threaded code and multiple offload statements to work around the fact that any given offload statement will block the host thread until the offloaded code completes.

Lesson 7: 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:

  Fortran:
  http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/fortran-lin/index.htm

  C/C++:
  http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm
- Appendix B in this document: *Information on Loop Independence from the Intel Compiler Guide*

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.

C code that doesn't vectorize while the Fortran does

- Inspect `serial`.F90
- Now run the following command:

```
ifort -mmic -vec-report3 serial.F90
```

- Did any of the loops vectorize?
- Inspect serial.cpp
- Run the following command:

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

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

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

- In the preceding example, the way in which arrays are handled in Fortran subroutines and functions allowed the compiler to rearrange the order of the loops and vectorize the code. However, the C version does not vectorize.
- 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 (This is the defined behavior in Fortran. Therefore, there is no equivalent to the restrict keyword in Fortran.)*
- Run the following command:

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

- Did any of the loops vectorize this time?
- Run the following command:

```
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*.)
- Run the following command:

```
ifort -mmic -opt-report serial.F90
```

- Compare the output to that from compiling restrict.cpp.

Manually duplicating how the compiler vectorized the code

- Continuing from the last section, now inspect `reorder.F90`, in which we manually reordered the loop to mirror what we think the compiler did automatically. How does it compare to `serial.F90`?
- Does this implement the reordering you think the compiler did from reading the output of **`-opt-report`** in the last example?
- Now issue the following commands:

```
ifort -mmic -vec-report3 reorder.F90
ifort -mmic -opt-report reorder.F90
```

- Did the code vectorize?
- How does the output from `-vec-report3` compare to what you get building `serial.F90`? Look at the vectorization message–it doesn't say "*permuted loop was vectorized*" like it did when we built `serial.F90`.
- Does the optimization report suggest that we matched the compiler's technique correctly?

Changing the compiler's vectorization decisions using !dec$

a) In the previous example, the compiler chose to vectorize over the loop index i, even though this required the compiler to reorder the loops. It did this so that the vector looped over arrays C and A with a stride of 1. What if we knew for some reason, possibly based on input data, that a loop should not be vectorized?
   - Inspect cdir1.F90. How does it compare to serial.F90?
   - ***Definition: !dec$ novector** instructs the compiler to not vectorize a specific loop.*
   - Now issue the following command:

   ```
   ifort -mmic -vec-report3 cdir1.F90
   ```

   - Did the code vectorize?
   - How does the vectorization differ from serial.F90?

b) By its nature, the compiler has to be conservative about what it can vectorize. The Fortran compiler, by definition, is able to treat the arrays passed to subroutines as non-overlapping. However, what if we are accessing the same array with different indexes?
   - Inspect `poss_dep.F90`. How does it compare to `serial.F90?`
   - Issue the command:

```
ifort –mmic -vec-report3 poss_dep.F90
```

- Did the code vectorize?
- Why?
- Inspect cdir2.F90. How does it compare to poss_dep.F90?
- ***Definition**: **!dec$ 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:

```
ifort –mmic -vec-report3 cdir2.F90
```

- Did the code vectorize?
- Why?

c) The previous !dec$ told the compiler not to make some assumptions that would prevent vectorization of the inner-most loop . The `!dec$ 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.F90`. How does it compare to `poss_dep.F90`?
  - ***Definition:!dec$ simd** enforces vectorization of the loop it immediately precedes.*
  - Now issue the following command:

```
ifort –mmic -vec-report3 cdir3.F90
```

  - 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 8: 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.

Useful References

- Compiler documentation http://software.intel.com/en-us/intel-software-technical-documentation
- Appendix A in this document: *Our Results for Lesson 7*

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.

  ```
  ifort -O1 -profile-functions -profile-loops=all -profile-loops-report=2 -liomp5 common.F90 lifeserial.F90 -o life
  ```

  ```
  ifort -O1 -profile-functions -profile-loops=all -profile-loops-report=2 -mkl mCarlo.F90 -o mCarlo
  ```

  ```
  ifort -O1 -profile-functions -profile-loops=all -profile-loops-report=2 mat_mul.F90 -o mat_mul
  ```

- Run each program
  - `./life virus.dat`
    - Will run for about 30 seconds
  - `./mCarlo`
    - Will run for about 20 seconds
  - `./mat_mul 1024`

- When we ran the programs, the compiler generated profiling information for every function and loop it encountered. Let us look at these data.
- Do an "`ls -la *.xml`" and note the times on the files.
- Look at the resulting xml files :

  ```
  source
  /opt/intel/composerxe/bin/intel64/loopprofileviewer.[c]sh
  ```

- Now open each xml file using File/Open in the tool that launched when the previous command was run. 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).
- To get some of this information, you will need to look at the source of the programs.

Life–Size of Grid = 582,000 bytes:

Time per call = Time / Number times called

Function	Number times called (call count)	Time per call	Data size passed per call (bytes)

Loop entries per function call = Loop entries / Number Times Function Called

Self Time per iteration = Loop Self time / (Loop entries * Average iterations)

Loop	Average iterations	Loop entries	Loop entries per function call	Time per iteration

Total Runtime:

MonteCarlo:

Function	Number times called (call count)	Time per call

Loop	Average iterations	Loop entries	Loop entries per function call	Time per iteration

Total Runtime:

MMul (1024x1024):

Function	Number times called (call count)	Time per call	Data size passed per call (bytes)

Loop	Average iterations	Loop entries	Loop entries per function call	Time per iteration

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, how frequently would you offload the work to the coprocessor (e.g., is a lot of work or a little work done on the host before the next offload call)?
    - 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:
- Assuming you offloaded the chosen function/loop, how long would the offloaded code run before it returned to the host?
    - 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

Lesson 9: Basic Performance Analysis

Goal

Gain a basic understanding for how to collect and analyze performance data on the Intel® Xeon Phi™ coprocessor

Useful References

- Intel® VTune™ Amplifier XE 2011 Documentation at http://software.intel.com/en-us/intel-software-technical-documentation

Lab

A few lessons ago we ported some matrix multiply code to this platform. It seemed to run OK, but how well is it really running?

- Let's look at performance optimization starting with the reordered version of the matrix multiply example, `omp_offload_ours.F90`.
- Build a version of this code for offload with optimization enabled:

```
ifort -O3 omp_offload_ours.F90 main.F90 \
     -o mmul_offload
```

- Run this version (`./mmul_offload 2048`) and time how long it takes to run:

  ____

- Our first step is to collect system performance data on the result. For this analysis we will sample the counts of each event observed while the program runs:

```
/opt/intel/vtune_amplifier_xe/bin64/amplxe-cl -collect-with
runsa-knc -knob event-
config="CPU_CLK_UNHALTED,INSTRUCTIONS_EXECUTED" -r mmul1 --
./mmul_offload 2048

/opt/intel/vtune_amplifier_xe/bin64/amplxe-cl -collect-with
runsa-knc -knob event-
config="DATA_READ_OR_WRITE,DATA_READ_MISS_OR_WRITE_MISS" -r
mmul2 -- ./mmul_offload 2048
```

- Open up the Intel® VTune™ Amplifier XE GUI and check the summary tab to find out the hardware event counts.

  /opt/intel/vtune_amplifier_xe/bin64/amplxe-gui mmul1/mmul1.amplxe

- Look at the data for core 2 in mmul1.txt and mmul2.txt.
  - What is the ratio of clock ticks to instructions executed? ______________
  - What fractions of reads or writes result in a miss? _______________
  - Is this program running well?

If you recall, our offload example `mCarlo_offload_ours.F90` did not perform very well compared to the host. Let's see if we can discover some of the reasons.

- Rebuild the application to run with optimization and with debug symbols:

```
ifort -g -O2 -mkl -openmp mCarlo_offload_ours.F90
```

- Let's start with some basic triage just to see where all the time is adding up. Rather than counting total numbers of events we will observe the program's over-time behavior:

```
/opt/intel/vtune_amplifier_xe/bin64/amplxe-cl -collect-with
runsa-knc -knob event-
config="CPU_CLK_UNHALTED,INSTRUCTIONS_EXECUTED,DATA_READ_OR_WR
ITE,DATA_READ_MISS_OR_WRITE_MISS" -r hs0001 -- ./a.out
```

- Fire up the Intel® VTune™ Amplifier XE GUI and use that to identify the hotspot in this program:

  ```
  /opt/intel/vtune_amplifier_xe/bin64/amplxe-gui \
  hs0001/hs0001.amplxe
  ```

  Note: Pay attention to what event is selected in the lower right for display on the timeline. Make sure it is set to `CPU_CLK_UNHALTED`.

- Where are the hotspots? Are they in code we can change?
- What is happening with memory accesses? Are we hitting cache? Why or why not?
- Right click on functions in the top pane or on the threads in the lower pane. Select a time range in the bottom pane. Note how you use all three of these elements to filter the displayed data. Also note how you can adjust the zoom on the timeline automatically when filtering. Finally, find the controls that allow you to undo the zooming or filtering.

Bonus

- Do the analysis on a different sample of offload code.

What we Learned

- How to collect over-time performance data while a program is executing and then how to view that data in Intel® VTune™ Amplifier XE 2013
- How to filter the amount of data you see in Amplifier by time, function, or thread

Appendix A: Our Results for Lesson 7:

Loop entries per function call = Loop entries / Number Times Function Called

Self Time per iteration = Loop Self time / (Loop entries * Average iterations)

Self Time per call = Self time / Number times called

Life–Size of Grid = 582,000 bytes:

Function	Number times called (call count)	Time per call	Data size passed per call (bytes)
ClearMap	144000000		582000
NeighborCount	144000000		582000+8

Loop	Average iterations	Loop entries	Loop entries per function call	Time per iteration
ClearMap:261	602	242,242	242	
NeighborCount:96	3	144000000	1	

MonteCarlo:

Function	Number times called (call count)	Time per call
Main	1	

Loop	Average iterations	Loop entries	Loop entries per function call	Time per iteration
Main:53	1000000	1	1	
Main:66	1000	1000000	1000000	

MMul (1024x1024):

Function	Number times called (call count)	Time per call	Data size passed per call (bytes)
Mmul	1		12582912
Main	1		12582912

Loop	Average iterations	Loop entries	Loop entries per function call	Time per iteration
Mmul:6	1024	1048576	1048576	
Mmul:5	1024	1024	1024	

Appendix B: Information on Loop Independence from the Intel Compiler Guide

Loop Independence

Loop independence is important since loops that are independent can be parallelized. Independent loops can be parallelized in a number of ways, from the course-grained parallelism of OpenMP* to the fine-grained Instruction Level Parallelism (ILP) of vectorization and software pipelining.

Loops are considered independent when the computation of iteration Y of a loop can be done independently of the computation of iteration X. In other words, if iteration 1 of a loop can be computed and iteration 2 simultaneously could be computed without using any result from iteration 1, then the loops are independent.

Occasionally, you can determine if a loop is independent by comparing results from the output of the loop with results from the same loop written with a decrementing index counter.

For example, the loop shown in loop_indep1 might be independent if the code in loop_indep2 generates the same result.

Example

```
#define MAX 1024

subroutine loop_indep1(a, b)
  integer, dimension(MAX) :: a, b
  do j=1,MAX
    a(j) = b(j)
  end do
end subroutine loop_indep1

subroutine loop_indep2(a, b)
  integer, dimension(MAX) :: a, b
  do j=MAX, 1, -1
    a(j) = b(j)
  end do
end subroutine loop_indep2
```

When loops are dependent, improving loop performance becomes much more difficult. Loops can be dependent in several general ways.

- Flow Dependency
- Anti Dependency
- Output Dependency
- Reductions

The following sections illustrate the different loop dependencies.

Flow Dependency--Read After Write

Cross-iteration flow dependence is created when variables are written then read in different iterations, as shown in the following example:

Example

```
subroutine flow_dep(A)
  real(kind=8),dimension(*)::A
  do j=2, MAX
    A(j)=A(j-1)
  end do
end subroutine flow_dep
```

The above example is equivalent to the following lines for the first few iterations:

Sample Iterations

```
A(2)=A(1)
A(3)=A(2)
```

Recurrence relations feed information forward from one iteration to the next:

Example

```
subroutine time_stepping_loops(a,b)
  real(kind=8),dimension(*):: a, b
  do j=2, MAX
    a(j) = a(j-1) + b(j)
  end do
end subroutine time_stepping_loops
```

Most recurrences cannot be made fully parallel. Instead, look for a loop further out or further in to parallelize. You might be able to get more performance gains through unrolling.

Anti Dependency--Write After Read

Cross-iteration anti-dependence is created when variables are read then written in different iterations, as shown in the following example:

Example

```
subroutine anti_dep1(A)
  real(kind=8),dimension(*):: A
  do j=1, MAX-1
    A(j)=A(j+1)
  end do
end subroutine anti_dep1
```

The above example is equivalent to the following lines for the first few iterations:

Sample Iterations

```
A(1)=A(2)
A(2)=A(3)
```

Output Dependency--Write After Write

Cross-iteration output dependence is when variables are written then rewritten in a different iteration. The following example illustrates this type of dependency:

Example

```
subroutine anti_dep2( A, B, C)
  real(kind=8),dimension(*):: A, B, C
  do j=1, MAX-1
    A(j)=B(j)
    A(j+1)=C(j)
  end do
end subroutine anti_dep2
```

The above example is equivalent to the following lines for the first few iterations:

Sample Iterations

```
A(1)=B(1)
A(2)=C(1)
A(2)=B(2)
A(3)=C(2)
```

Reductions

The Intel® compiler can successfully vectorize or software pipeline (SWP) most loops containing reductions on simple math operators like multiplication (*), addition (+), subtraction (-), and division (/). Reductions collapse array data to scalar data by using associative operations:

Example

```
subroutine reduction(sum, c)
  real(kind=8):: sum
  real(kind=8),dimension(*):: c
  do j=1, MAX
    sum = sum + c(j)
  end do
end subroutine reduction
```

The compiler might occasionally misidentify a reduction and report flow-, anti-, or output-dependencies, or sometimes loop-carried memory-dependency-edges; in such cases, the compiler will not vectorize or SWP the loop. When you recognize that the programming construct is simply a reduction, direct the compiler through the use of directives (such as !dec$ ivdep) to vectorize or SWP the loop.

Legal Notice