Intel® Xeon Phi™ Programming Environment

Abstract

We can look at the Intel® Xeon Phi™ as a black box and understand its architecture by looking at the response produced by this hardware with the impulse we provide it. In our case this will be the software instructions executing on the coprocessor. The objective of this book is to introduce the reader to Intel® Xeon Phi™ architecture that affects software performance through programming. I believe one of the best ways to understand and learn about a new architecture is to see its behavior with respect to how it performs against the requests made to it.

In this article we shall look at the tools and development environment that is available to the developers as they explore and develops software for this coprocessor. Having this knowledge will allow us in subsequent chapter to write the code to evaluate various architectural features implemented on the coprocessor.

Article

Intel® Xeon Phi™ cores are Pentium cores and works as coprocessor to host processor. Due to the fact that it is based on Pentium core allowed developers to port many of the tools and development environment from Intel® Xeon® based processor to the Xeon Phi coprocessor. In fact the software designer opted for running a complete micro OS based on Linux kernel rather than the driver based model often used for PCI express based attached cards like Graphics cards on a system.

As such there are various execution models that can be used to design and execute an application on Intel® Xeon Phi™ Coprocessor in association with the host processor. The programming models supported for the coprocessor may vary between the Windows OS and Linux OS used on the host system. Currently we support only Linux* and Windows* operating environment. As of this writing Windows* operating environment is in the development and will support a subset of the execution models available for Linux* operating system. Although the compiler syntax for running on the Windows* environment will be very close to Linux* environment; to simplify the discussion in this book I would be focusing on Linux* based platform only.

Intel® Xeon Phi™ Execution Models

The most common execution models are depicted in Figure 1 below and can be broadly categorized as follows:

1. Offload Execution Mode: Also known as heterogeneous programming mode, here the host system offloads part or all of the computation from one or multiple processes or threads running on host. The application starts execution on the host. As the computation proceeds it can decide to send data to the coprocessor and let that work on it and the host and the coprocessor may or may not work in parallel. This is the common execution model in other coprocessor operating environments. As of this writing, there is an OpenMP 4.0 TR being proposed and implemented in Intel® Composer XE to provide directives to perform offload computations. Composer XE also provides some custom directives to perform offload operations.
   This mode of operation will be available on both Linux and Windows.
2. Coprocessor Native Execution Mode: As I described earlier, an Intel® Xeon Phi™ hosts a Linux micro OS in it and can appear as another machine connected to the host like another node in a cluster. This execution environment allows the users to view the coprocessor as another compute node. In order to run natively, an application has to be cross compiled for Xeon Phi operating environment. Intel® Composer XE provides simple switch to generate cross compiled code.
3. Symmetric Execution: In this case the application processes run on both the host and the Intel® Xeon Phi™ coprocessor. They usually communicate through some sort of message passing interface like MPI. This execution environment treats Xeon Phi card as another node in a cluster in a heterogeneous cluster environment.

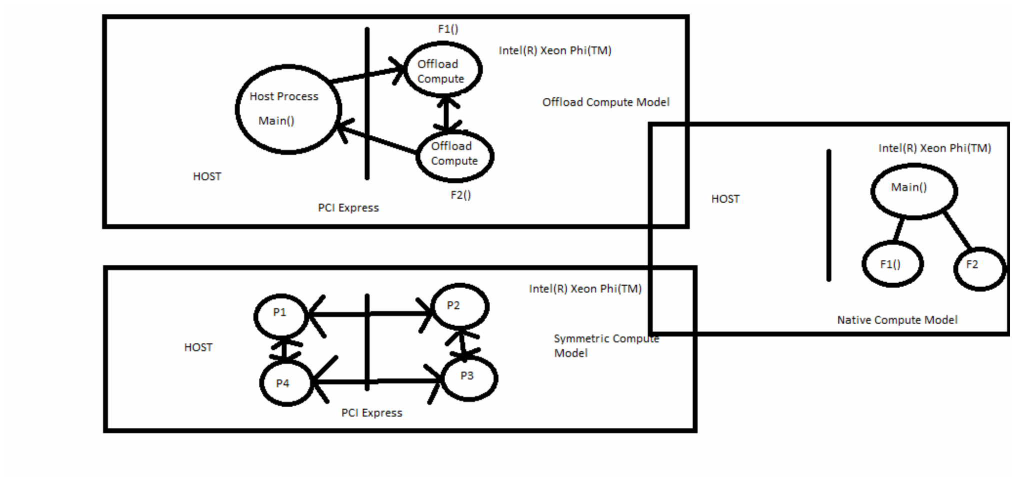

Figure 1: Intel® Xeon Phi™ Execution Models

Development Tools for Intel® Xeon Phi™ Architecture

Figure 2 shows various tools that are developed by Intel to ease developing, and tuning applications for Intel® Xeon Phi™ coprocessor. There are also other excellent tools developed by third party vendors that will not be covered in this section.

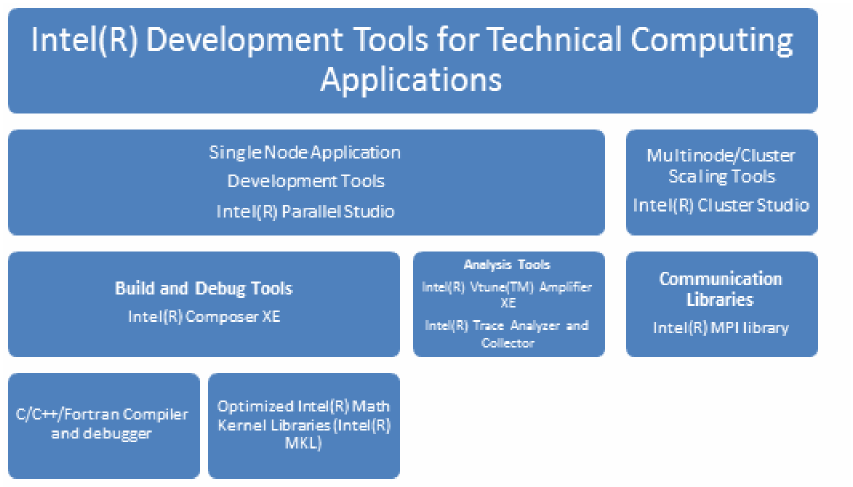
Intel(R) Development Tools for Technical Computing Applications
Single Node Application Development Tools Intel(R) Parallel Studio
Multinode/Cluster Scaling Tools Intel(R) Cluster Studio
Build and Debug Tools Intel(R) Composer XE
Analysis Tools Intel(R) Vtune(TM) Amplifier XE Intel(R) Trace Analyzer and Collector
Communication Libraries Intel(R) MPI library
C/C++/Fortran Compiler and debugger
Optimized Intel(R) Math Kernel Libraries (Intel(R) MKL)

Figure 2 Software Development Tools for Intel® Xeon Phi™

Intel® Composer XE

The Intel® Composer XE is the key development tool and SDK suite available for developing on Intel® Xeon Phi™. The suite includes C/C++ and Fortran compiler and related runtime libraries like OpenMP, thread etc., debugging tool and math kernel library (MKL).
Together they give necessary tool to build you application for Intel® Xeon® compatible processors and Intel® Xeon Phi™ coprocessors. You can also use the same compiler for cross compilation to Intel® Xeon Phi™.
Assuming you have access to an Intel® Xeon Phi™ based development environment, we shall walk through how we can write a simple application to run on various execution modes described above. Once you learn to do so, we can progress to following chapter on Xeon Phi™ architecture to get better understand through experimentation.

We shall also cover the run time environment and system level details in this chapter for a complete understanding of how the software architecture is created to work in conjunction with host processor to complement host processor computations.
The C/C++/Fortran tools contained in Intel® Composer XE supports various parallel programming models for Intel® Xeon Phi™ such as Intel Cilk Plus, Intel® threading building block (TBB), OpenMP and pthread. The composer XE also contains

Intel® Math Kernel Library (MKL) which contains common routines like BLAS, FFT and standard interfaces for technical computing applications.

Getting the Tools

All of the tools described in this chapter for developing for Intel® Xeon Phi™ are available from Intel® web site. If you do not have the tool, you can get evaluation version of the tool from http://software.intel.com/en-us/intel-software-evaluation-center/

Using the Compilers

The C++/Fortran compiler included as part of the Intel® Composer Xe package generates both offload and cross compiled code for Intel® Xeon Phi™. The compiler can be used in command line or Eclipse development environment. We shall follow the command line options for our work in this book. If you are interested in Eclipse development environment, please refer to the user's guide provided as part of the compiler documentation.

I shall be covering Xeon Phi specific and related features on 64 bit Redhat* Linux* 6.3 for Intel® Xeon® processor, in this book. I shall also assume 'bash' script command syntax for the examples. The compiler contains a lot of features that will not be covered in this book. The command prompt will be indicated by ***">"*** symbol.

In order to invoke the compiler you need to set some environment variables so that the compiler is in the path and runtime libraries are available. To do this you need to invoke a batchfile called compilervars.sh included with the compiler. If you have installed in the default path chosen by the compiler, the batch file to set environment can be found at /opt/intel/composerxe/bin/compilervars.sh. To set the path invoke

"> source /opt/intel/composerxe/bin/compilervars.sh intel64".

The compiler is invoked and linked with *"icc"* for building "C" source files and *"icpc"* for building and linking "C++" source files. For Fortran sources, you need to use the *"ifort"* command for both compiler and link. Make sure you link with appropriate command as these commands link in proper libraries to produce the executable.

Of course you can use the make utility to build multiple objects. If you are porting a make file from using gcc, you need to make with Intel® Compiler, set the 'CC' environment variables to use the Intel® compilers and modify the command line switches appropriately. Since it is hard to remember all the switches, you can always invoke Intel® compilers like icc with *">icc help"* to figure out the appropriate options for you compiler builds. In most cases if not asked specifically for compiling only, an 'icc' or 'icpc' command will invoke both the compiler and the linker. In fact these commands like icc, icpc or ifort are driver program that in turn parses the command line arguments and processes accordingly with compiler or the linker as necessary. The driver program processes the input file and calls the linker with the object files created as well as necessary library files to generate final executable or libraries. That is why it is important to use proper compiler, for example ifort to build Fortran or a mix of Fortran and C files so that appropriate libraries can be linked.

The Intel® compiler uses file extensions to interpret the type of each input file. The file extension determines whether the file passed to compiler or linker. A file with .c, .cc, .CC, .cpp, .cxx is recognized by the C/C++ compiler. A Fortran compiler, *'ifort'* recognizes .f90, .for, .f, .fpp, .i, .i90, .ftn extensions. Fortran compiler assumes files with .f90 or .i90 extensions are free form Fortran source files. The compiler assumes, .f, .for, .ftn, .i as fixed form Fortran files. Files with extensions .a, .so, .i, .o and .s are passed on to the linker.Table 2.1 describes the action by the compiler depending on the file extensions.

File extensions	Interpretation	Execution
.c	C source file	C/C++ compiler
.C, .CC,.cc,.cpp,.cxx	C++ source file	C++ compiler
.f, .for, .ftn, .i, .fpp, .FPP, .F, .FOR, .FTN	Fixed form Fortran	Fortran compiler
.f90,.i90, .F90	Free form Fortran	Fortran compiler
.a, .so, .o	Library, object files	Linker
.s	Assembly file	assembler

Table 1: File extensions and their interpretation by Intel® Compiler.

The Intel® compiler can be invoked as follows:

<compiler name> [options] file1 [file2…]

Where:

<compiler name> is one of the compiler names like icc, icpc, ifort described above.

[options] are options that are passed to compiler and can control code-generation, optimization and output file names, type and path. If no *optionsa*re specified, the compiler invokes some default options like –O2 for default optimization. If you want to modify the default option for compilation, you will need to modify the corresponding configuration file found in the installed <compiler install path>bin/intel64_mic or similar folders and named as icc.cfg, icpc.cfg etc. Please refer to compiler manual for details.

Compiler options play a significant roles in tuning for Intel® MIC architecture as you can control various aspects of code generation like loop unrolling, prefetch generation etc. Often, one might find useful to look at the assembly code generated by the compiler. You can use the –S option to generate assembly file to see the assembly coded output of the compiler.

Setting up an Intel® Xeon Phi™ System

We shall assume you have access to a host with one or more Intel® Xeon Phi™ cards installed in it and one of the supported Linux OS on the host. For Windows* please refer to corresponding users guide. There are two high level packages the drivers (also known as MPSS package) and the development tools and libraries (distributed as Intel® Cluster Studio or a single node version Intel® Composer XE) packages to get a system where you can develop applications for Intel® Xeon Phi™.

Install MPSS stack:

1. Goto Intel® Developer Zone page http://software.intel.com/mic-developer , go to tab "*Tools & Downloads*" and select "Intel® Many Integrated Core Architecture (Intel® MIC Architecture) Platform Software Stack". Download appropriate version of the MPSS to match your host OS and also download the readme.txt from the same location.
2. You will need super user privilege to install the MPSS stack.
3. Communication with the Linux* micro-os (operating system) running on the Intel(R) Xeon Phi(TM) Coprocessor is provided by a standard network interface. The interface uses a virtual network driver over the PCIe bus. The Intel(R) Xeon Phi(TM) coprocessor Linux* OS supports network access for all users using SSH keys. A valid SSH key is required for you to access the card. Most users will have it on the machine. If you have connected to other machine form this host through ssh you most probably have it. If you do not have a 'ssh' key do the following:

To generate the SSH key, execute

```
user_prompt> ssh-keygen
user_prompt> sudo service mpss stop
user_prompt> sudo micctrl --resetconfig
user_prompt> sudo service mpss start
```

4. Make sure you have downloaded the correct version of the MPSS stack that matches your host operating system where you installed the Intel® Xeon Phi™ card. If not, the MPSS source is provided to build for some of the supported Linux OS versions.
5. These packages are distributed as gzipped Linux* tar files with extension .tgz. Untar the *.tgz package and go to untarred location.
6. Install the rpms in the untarred directory with appropriate rpm install command. For example, on Red Hat* Enterprise Linux* you can use the following command on the command line as a root user.
   - *commnad_prompt> yum install --nopgpcheck --noplugins --disablerepo=* *.rpm*
7. Reset the driver using:
   - *command_prompt>micctrl -r*
8. Update the system flash if necessary. To see whether you need to update, please run the *command_prompt>/opt/intel/mic/bin/micinfo* which will printout the Intel® Xeon Phi™ related information including the flash file. *Note: the default MPSS install puts most of the Intel Xeon Phi™ related drivers, flash and utilities in /opt/intel/mic and /usr/bin default path. If your system admin put it somewhere else, you need to check with her/him.I shall assume everything is installed in the default folder as of this writing.*

- The flash files can be found in the folder /opt/intel/mic/flash and should match with that printed out as part of the micinfo. If the installed version is older than the one available with new MPSS you are installing, update the flash with 'micflash' utility. Please refer to readme.txt provided with the documentation to select the proper flash file. Once you determined the proper flash file for the revision of card on your system use following command to flash:*command_prompt>/opt/intel/mic/bin/micflash –Update\ /opt/intel/mic/flash/<your flash file name>*

9. Once the flash is updated, reboot the machine for the new flash to affect. MPSS is installed as a Linux* service and can be started/stopped by *service mpss start|stop|restart* commands.
10. Note that you need to have proper driver configuration to get the card started. For a machine which had an old driver stop, a mismatched card configuration form previous install could prevent the card from booting. I would strongly suggest you read up on MPSS configuration section and "micctrl" utility provided in readme.txt, if you face any issue starting the card.

Install the Development Tools
Install the Intel® C/C++ compiler by obtaining Intel® composer XE package or a superset of this package like Intel® Cluster Studio XE for Linux. As of this writing the latest version was Intel® composer XE 2013. Follow the instructions provided in the link above or refer to appendix on how to get access to the tools for step by step methods to install. Since this is a step is same as installing the Intel® tools on any Intel® Xeon® processor based hosts, I am not going to cover this step.

Code Generation for Intel® Xeon Phi™ Architecture

The traditional Intel® compiler has been modified to support Intel® Xeon Phi™ code generation. In order to do that, the compiler engineers had to make changes at various levels.
The first step was adding new language features that allow one to describe the offload syntax for a code fragment that can be sent to the coprocessor. As of this writing, these language features are available as OpenMP 4.0 Technical Report as well as Intel® proprietary C/C++/Fortran extensions.
Secondly, we needed some new compiler options to account for Intel® MIC architecture specific code generations and optimizations. These include providing new intrinsics corresponding to Intel® Xeon Phi™ specific ISA extensions which we shall see in subsequent chapter. These intrinsics will help us explore various architectural features of the hardware.
Finally, we need to provide support for new environment variables and libraries to allow execution of the offload programs on the hardware.
There are two prominent programming models for Intel® Xeon Phi™. The Native execution mode where you cross-compile the code written for Intel® Xeon® Processors and run them on the Xeon Phi micro-os. The other is heterogeneous or hybrid code where you start the main code on the host and the code executes on coprocessor or both the host and the coprocessor.

Native Execution Mode

This book will mainly deal with native execution mode as this is the simplest way to run something on the Xeon Phi™ hardware. Since we are more interested in understanding the architecture, this programming mode is well suited for the coding described in this book.
In native execution mode, you can run any C/C++/Fortran source that can be compiled for Intel® Xeon, can be cross compiler for Intel® Xeon Phi™ architecture. The code is then transferred to Xeon Phi micro-os operating environment using familiar networking tools like 'scp' and executed in the Xeon Phis native execution environment.
In this section I shall assume you already have access to a system with Intel® Xeon Phi™ coprocessor installed in it. I shall also assume you have downloaded the appropriate driver and composer XE on your machine form Intel® registration center.

Hello World Example:
Let us try running a simple 'hello world' application on Intel® Xeon Phi™ processor in native mode. Say you have a simple code segment as follows in a source file test.c;

```
//Content of test.c
```

```
#include <stdio.h>
int main()
{
   printf("Hello world from Intel® Xeon Phi™\n");
}
```

To build the code you need to compile and link this files with '-mmic' switch as follows:

```
command_prompt>icc -mmic test.c -o test.out
```

This will create an output file test.out on the same folder as your source.Now copy the source file to first Intel® Xeon Phi™ 'mic0' as follows:

```
command_prompt>scp test.out mic0:
```

This will copy the test.out file to your home directory on the coprocessor environment. At this phase you can login to the coprocessor using ssh command as follows:

```
command_prompt>ssh mic0
[command_prompt-mic0]$ ls
test.out
```

The listing now shows the files test.out on the native coprocessor environment. If you run it on the card, it will printout:

```
commnad_prompt-mic0>./test.out
Hello world
commnad_prompt-mic0>
```

Language extensions to Support Offload Computation on Intel® Xeon Phi™

Heterogeneous Computing Model and Offload Pragmas

There are two supported ways to deal with non-shared memory between Intel® Xeon Phi™ coprocessor and host processor. One is data marshaling where the compiler runtime generates and sends data buffer over to coprocessors by marshaling the parameters provided by the application. The second option is virtual shared memory model which is depends on a third system level runtime support to maintain data coherency between the host and coprocessor shared memory address space. As of this writing, the non-shared memory model is supported by OpenMP 4.0 TR.
We should also note that Intel® compiler had proprietary language extensions that were implemented to support non-shared programming model before the OpenMP 4.0 TR was published, however we shall stick to OpenMP 4.0 TR syntax in this article for non-shared programming model and is supported by Intel® compilers.
In the non-shared programming model, the main program starts on the host and the data and computation can be sent to Intel® Xeon Phi™ through 'offload' pragmas when the data exchange between the host and the coprocessor are bitwise copy-able. The bitwise copy-able includes scalars, arrays and structures without indirections. Since the Coprocessor memory space is separate from the host memory space, the compiler runtime is in charge of copying data back and forth between the host and the coprocessor around the offload block indicated by the pragmas.
The data selected for offload may be implicitly copied if used inside the offload code block provided the variables are in the lexical scope of the code block performing offload and variable listed explicitly as part of the pragma. The requirement is that the data must have a flat structure. However, the data which is used only within the offload block can be arbitrarily complex with multiple indirections but can be passed between host and the coprocessor. If this is needed, you have to marshall the data into a flat data structure or buffer to move data back and forth. The Intel® compiler does support transparent marshaling of data using a "shared virtual memory constructs" described later which can handle more complex data structures.
In order to provide seamless execution so that the same code can run on host processor with or without a coprocessor, the compiler runtime determines whether the coprocessor is present on the system or not. So if the coprocessor is not available or inactive during the offload call, the code fragment inside the offload code block may execute on the host as well.

Language Extensions and Execution Model
In order to understand the new language extensions for non-shared memory programming, we should understand the terminologies introduced in OpenMP 4.0 Technical Report (TR) (www.openmp.org/mp-documents/TR1_167.pdf) to describe the language support and is described below. The description in this book is not comprehensive; please refer to TR for details.
In order to understand the language extensions defined in Open4.0 to deal with coprocessor, we need to understand some terminologies associated with the specification to explain the coprocessor interaction.

Terminologies
Device: The coprocessor with its own memory:A *device* could have one or more co-processors or host.A host device is the device executing the main thread.A target device executes the offloaded code segment.
Offload: The process of sending computation from host to target.
Data Environment: The variables associated with a given execution environment.
Device data environment A *data environment* associated with a **target data** or **target** construct.
Mapped variable Mapping of a *variable* in a data environment to a *variable* in a device data environment. The original and corresponding variable may share storage.
Mappable type: A valid 'data type' for a *mapped variable*.

Offload Function and Data Declaration Constructs
Target Data Declarations: Declares device data environment for the scope of the target code block. This allows enables creation of versions of specified function or data that can be used inside a target region executing on the coprocessor.
Syntax:
C/C++
#pragma omp declare target *new-line*
[function-definition-or-declaration]
#pragma omp end declare target *new-line*
Fortran
!$omp declare target*[(proc-name-list | list)]new-line*

Function Offload and Execution Constructs:
In order to execute on a coprocessor, a 'target' device must be available on the host system running the code. For non-shared memory model, the host application executes the initial program which spawns the master execution thread called "initial device thread". The OpenMP pragma 'target' provides the capability to offload computations to coprocessor(s).
- A **target** region begins as a single thread of execution and executes sequentially, as if enclosed in an implicit task region, called the initial device task region.
- When a **target** construct is encountered, the **target** region is executed by the implicit device task.
- The task that encounters the **target** construct waits at the end of the construct until execution of the region completes. If a coprocessor does not exist, or is not supported by the implementation, or cannot execute the **target** construct then the **target** region is executed by the host device.
- The data environment is created at the time the construct is encountered if needed. Whether a construct creates a data environment is defined in the description of the construct.

Syntax:
C/C++
#pragma omp target *[clause[[,] clause],...] new-line*
parallel-loop-construct | parallel-sections-construct
Fortran
!$omp target *[clause[[,] clause],...]*
parallel-loop-construct | parallel-sections-construct
!$omp end target

Where clauses are:

- **device(*scalar-integer-expression*)**
  - The integer expression must be a positive number to differentiate various coprocessor available on a host. If no device is specified, the default device is determined by internal control variable (ICV) named num-var.
- **map(alloc | to | from | tofrom: *list*), Scratch(list):**
  – These are data motion clauses that allow copying and mapping of variables or common block to/from host scope to target device scope.
- **if(*scalar-expr*):** If the scalar-expression evaluates to false, the new device environment is not created.
- **Num_threads(** *list* **)**

The 'target' directive creates a device data environment and executes the code block on the target device. The target region binds to the enclosing parallel or task region. It provides a superset of *"target data"* constructs described later and describes data as well as the code block to be executed on the target device. The master task waits for the coprocessor to complete the target region at the end of the constructs.
When an 'if' clause is present and if the logical expression inside the if clause evaluates to false, the target region is not executed by the device. When a *num_threads* clause is present, the nthreads-var in the coprocessor environment is assigned the value list.

Restrictions:

- If a target, target update, or target dataconstruct appears within a target region then the construct is ignored.
- At most one device clause may appear on the directive. The device expression must evaluate to a positive integer value.
- At most one 'if' clause can appear on the directive.
- The result of an **omp_set_default_device**, **omp_get_default_device**, or **omp_get_num_devices** routine called within a target region is unspecified.
- The effect of an access to a **threadprivate** variable in a target region is unspecified.
- A variable referenced in a **target construct** that is not declared in the construct is implicitly treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.
- A variable referenced in a target region but not the target construct that is not declared in the target region, must appear in a **declare target** directive.
- C/C++ Specific: A throw executed inside a **target** region must cause execution to resume within the same **target** region, and the same thread that threw the exception must catch it.

The extension also provides routines to set and get runtime environment settings (referred to as internal control variables ICV in the OpenMP spec). These are:

ICV name	Routines
device-num-var	**omp_set_device_num(),** **omp_get_device_num();**

The syntax for expressing data transfers between host and coprocessors is expressed by target data and update constructs discussed next.

Target Data Constructs

Target Data Constructs: Creates the device data environment for the scope of the target code block. If there are no 'device' clause, the default device is determined by device-num-var
Syntax:
C/C++

#pragma omp target data*[clause[[,] clause],...] new-line*
structured-block

Fortran

!$omp target data *[clause[[,] clause],...]*
structured-block
!$omp end target data

Where Clauses are:
device(*scalar-integer-expression*)
The integer expression must be a positive number to differentiate various coprocessor available on a host. If no device is specified, the default device is determined by internal control variable (ICV) named num-var.
map(alloc | to | from | tofrom: *list*),Scratch(list):
– These are data motion clauses that allow copying and mapping of variables or common block to/from host scope to target device scope.
if(*scalar-expr*): If the scalar-expression evaluates to false, the new device environment is not created.
Structured-block: No branching in and out of the block of code is allowed.

Target Update Constructs

Target Update Constructs: This construct synchronizes the list items in the device data environment consistent with their corresponding original list items.
Syntax:
C/C++

#pragma omp target update*[clause[[,] clause],...] new-line*

Fortran

!$omp target update*[clause[[,] clause],...]*

Where Clauses are:
device(*scalar-integer-expression*): The integer expression must be a positive number to differentiate various coprocessor available on a host. If no device is specified, the default device is determined by internal control variable (ICV) named num-var.
map(to | from: *list*): These are data motion clauses that allow copying and mapping of variables or common block to/from host scope to target device scope.
if(*scalar-expr*): If the scalar-expression evaluates to false, the update clause is ignored.

Offload Example

The example is the text box below shows a simple example of how the constructs discussed above for OpenMP 4.0 can be used to offload computation to Intel® Xeon Phi™ coprocessor.
This example shows a sum reduction operation being run on a Xeon Phi processor. The offload computation “reduce” is first declared with *“#pragma omp declare target”* clause. This causes the routine or its component to be executable on the coprocessor.
On line #13 of the listing, you will notice the offload pragma,
“*`#pragma omp target map(inarray[0:SIZE]) map(sum)`*“
that causes specific code block line 14-17 to be sent to coprocessor, for computing. In this case it is computing the reduction of an array of numbers and returning the computing value through the “sum” variable to the host. The “inarray” and the “sum” is copied in and out of the coprocessor before and after the computation.
The main routine calls the “reduce” function in line#32 as shown in the code block. The code in turn offloads part of the computation to Intel® Xeon Phi™ to compute the sum reduction. Once the reduction is done the results received from the coprocessor is returned to main and compared against the reduction done on host to validate the results.
You can compile this code as you would on a Intel® Xeon™ machine using the following command:
command_prompt>icpc –openmp reduction.cpp –o test.out
And you can run the compiled test.out on the host environment as shown in the text block 2 below using

command_prompt>./test.out

Example Test Code:

```
// Sample code reduction.cpp
// Example showing use of OpenMP 4.0 pragmas for offload calculation
// Tis code was compiled with Intel(R) Composer XE 2013

#include <stdio.h>

#define SIZE 1000
#pragma omp declare target
int reduce(int *inarray)
{

   int   sum=0;
   #pragma omp target map(inarray[0:SIZE]) map(sum)
   {
      for(int i=0;i<SIZE;i++)
        sum += inarray[i];
   }
  return sum;
}

int main()
{
   int inarray[SIZE], sum, validSum;

   validSum=0;
   for(int i=0; i<SIZE; i++){
     inarray[i]=i;
     validSum+=i;
   }

   sum=0;
   sum = reduce(inarray);

   printf("sum reduction = %d, validSum=%d\n",sum, validSum);
}
```

You can see the data movement by the runtime engine if you set H_TRACE=1 in the host environment. The output with H_TRACE set to 1 during runtime is shown in text block 2. Here you see that there were 4000 Bytes (1000 integer elements of the 'inarray' are sent back and forth between host and the target).

Data Transfer Log between Coprocessor and Host:

```
command_prompt>export H_TRACE=1
command_prompt>./test.out
HOST:  Offload function
__offload_entry_reduction_cpp_10_Z6reducePi,
is_empty=0, #varDescs=3, #waits=0, signal=(nil)
HOST:  Total pointer data sent to target: [4000] bytes
HOST:  Total copyin data sent to target: [12] bytes
HOST:  Total pointer data received from target: [4000]
bytes
MIC0:  Total copyin data received from host: [12]
bytes
MIC0:  Total copyout data sent to host: [4] bytes
HOST:  Total copyout data received from target: [4]
bytes
sum reduction = 499500, validSum=499500
command_prompt>
```

This article is based on material found in the book *Intel® Xeon Phi™ Coprocesser Micro Architecture and Tools.* Visit the Intel Press web site to learn more about this book:
http://noggin.intel.com/intelpress/categories/books/intel%C2%AE-xeon-phi%E2%84%A2-coprocesser-micro-architecture-and-tools

Also see our Recommended Reading List for related topics: www.intel.com/technology/rr

About the Author
Reza Rahman is a Sr. Staff Engineer at Intel Software and Services Group. Reza lead the worldwide technical enabling team for Intel Xeon Phi™ product through Intel software engineering team. He played a key role during the inception of Intel MIC product line by presenting value of such architecture for technical computing and leading a team of software engineers to work with 100s of customers outside Intel Corporation to optimize code on Intel® Xeon Phi™. He worked internally with hardware architects and Intel compiler and tools team to optimize and add features to improve performance of Intel MIC software and hardware components to meet the need of technical computing customers. He has been with Intel for 19 years. During his first seven years he was at Intel Labs working on developing audio/video technologies and device drivers. Rest of the time at Intel he has been involved in optimizing technical computing applications as part of software enabling team. He is also believes in standardization process allowing software and hardware solutions to interoperate and has been involved in various industry standardization group like World Wide Web consortium (W3C).

Reza holds a Masters in computer Science from Texas A&M university and Bachelors in Electrical Engineering from Bangladesh University of engineering and Technology.

=========================