CHAPTER 2

Programming Xeon Phi

Viewing the Intel Xeon Phi as a black box, you can infer its architecture from its responses to the impulses you provide it: namely, the software instructions you execute on the coprocessor. The objective of this book is to introduce you to Intel Xeon Phi architecture in as much as it affects software performance through programming. I believe one of the best ways to understand and learn about a new architecture is to observe its behavior with respect to how it performs in relation to the requests made of it.

This chapter looks at the tools and development environment that are available to developers as they explore and develop software for this coprocessor. The knowledge in this chapter provides the foundation for writing code to evaluate various architectural features implemented on the coprocessor, which will be covered in Chapters 3 through 6.

Intel Xeon Phi Execution Models

Intel Xeon Phi cores are Pentium cores and work as coprocessors to the host processor. Pentium core adoption allowed developers to port many of the tools and much of the development environment from the Intel Xeon-based processor to the Xeon Phi coprocessor. In fact, the software designer opted for running a complete micro OS based on the Linux kernel rather than the driver-based model often used for PCIe-based attached cards, comparable to graphics cards on a system.

There are various execution models that can be used to design and execute an application on the Intel Xeon Phi coprocessor in association with the host processor. The programming models supported for the coprocessor vary between the Windows OS and Linux OS used on the host system. For example, the native programming model is only available on Linux but not on Windows. Intel Xeon Phi supports only Linux and Windows operating environments. The compiler syntax for running on the Windows environment is very close to that for the Linux environment. To simplify the presentation, I focus in this book on the Linux-based platform only.

The most common execution models can be broadly categorized as follows (Figure 2-1):

Offload execution mode. Also known as the *heterogeneous programming mode*, the host system in this mode offloads part or all of the computation from one or multiple processes or threads running on the host. The application starts execution on the host. As the computation proceeds, it can decide to send data to the coprocessor and let the coprocessor work on it. The host and the coprocessor may or may not work in parallel in the offload execution model. 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 is available on both Linux and Windows.

Coprocessor native execution mode. An Intel Xeon Phi has a Linux micro OS running 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 the Xeon Phi operating environment. Intel Composer XE provides a simple switch to generate cross-compiled code.

Symmetric execution. In this mode the application processes run on both the host and the Intel Xeon Phi coprocessor. They usually communicate through some sort of message-passing interface such as Message Passing Interface (MPI). This execution environment treats the Xeon Phi card as another node in a cluster in a heterogeneous cluster environment.

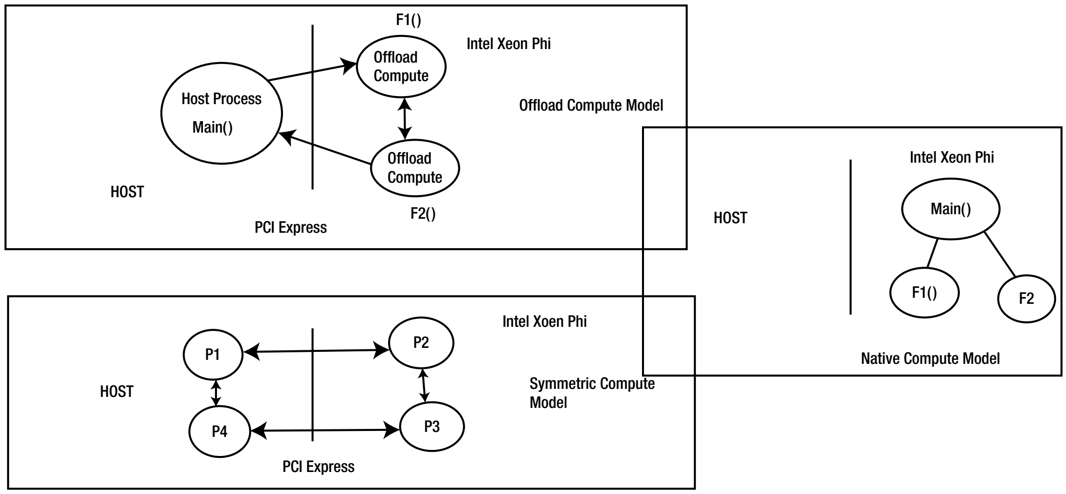


Figure 2-1. *Intel Xeon Phi execution models. Here P indicates processes and F() indicates function calls in various execution modes. The arrows indicate the function invocation, message passing, and data communication directions between the processes and functions*

Development Tools for Intel Xeon Phi Architecture

Various tools (Figure 2-2) developed by Intel ease developing and tuning applications for the Intel Xeon Phi coprocessor. Various excellent tools developed by third-party vendors will not be covered in this section.

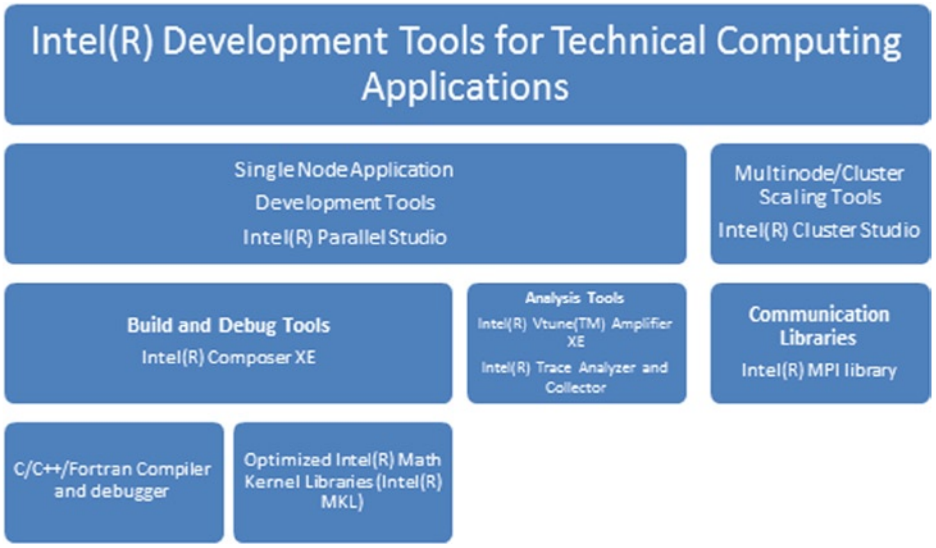

Intel(R) Development Tools for Technical Computing Applications
Single Node Application Development Tools Intel(R) Parallel Studio
Multinode/Cluster Scaling Tools Intel(R) Cluster Studio
Build and Debug Tools Intel(R) Composer XE
Analysis Tools Intel(R) Vtune(TM) Amplifier XE Intel(R) Trace Analyzer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

Figure 2-2. *Software development tools for Intel Xeon Phi*

Intel Composer XE

Intel Composer XE is the key development tool and Software Development Kit (SDK) suite available for developing on Intel Xeon Phi. The suite includes C/C++ and Fortran compiler and related runtime libraries and tools such as OpenMP, threading building block, Cilk Plus, a debugging tool, and the math kernel library (MKL). Together they give you the necessary tools to build your 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.

On the assumption that you have access to an Intel Xeon Phi–based development environment, I will walk you through how you can write a simple application to run on the various execution modes described above. Once you learn to do that, you can progress to Chapter 3 on Xeon Phi architecture to get a better understanding through experimentation.

The present chapter also covers the runtime environment and system-level details to give you a complete understanding of how the software architecture is created to work in conjunction with the host processor to complement its computations.

The C/C++/Fortran tools contained in the Intel Composer XE support various parallel programming models for Intel Xeon Phi, such as Intel Cilk Plus, Intel threading building blocks (TBB), OpenMP, and POSIX threads (pthread). The Composer XE also contains the Intel MKL, which contains common routines for technical and high-performance computing applications, including Basic Linear Algebra Subroutines (BLAS), Fast Fourier Transform (FFT), and standard interfaces for technical computing applications.

Getting the Tools

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

Using the Compilers

The C++/Fortran compilers included as part of the Intel Composer XE package generate both offload and cross-compiled code for Intel Xeon Phi. The compiler can be used in a command-line or Eclipse development environment. This book will follow the command-line options. If you are interested in the Eclipse development environment, refer to the user's guide provided as part of the compiler documentation.

This book covers Xeon Phi-specific and -related features on 64-bit Redhat Linux 6.3 for the Intel Xeon processor. I will assume "bash" script command syntax in the examples. The compiler contains a lot of features that will not be covered in this book. The command prompt will be indicated by the ➤ symbol.

In order to invoke the compiler, you need to set some environment variables so the compiler is in the path and runtime libraries are available. To do this, you need to invoke a batch file called `compilervars.sh` included with the compiler. If you have installed this in the default path chosen by the compiler, the batch file to set the 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 the appropriate command, as these commands link to the proper libraries to produce the executable.

Of course, you can use the make utility to build multiple objects. If you are porting an application built with GNU C Compiler (gcc) to Xeon Phi, you will need to change the build script to use the Intel compiler and to modify the command line switches appropriately. Because 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 your 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 the commands `icc`, `icpc`, and `ifort` are driver programs that in turn parse the command-line arguments and processes in accordance with the 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 the library files necessary to generate final executables or libraries. That is why it is important to use the proper compiler so that the 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 is passed to the compiler or linker. A file with .c, .cc, .CC, .cpp, or .cxx is recognized by the C/C++ compiler. A Fortran compiler recognizes .f90, .for, .f, .fpp .i90, and .ftn extensions. A Fortran compiler assumes that files with .f90 or .i90 extensions are free-form Fortran source files. The compiler assumes .f, .for, and .ftn as fixed-form Fortran files. Files with extensions .a, .so, .o, and .s are passed on to the linker. Table 2-1 describes the action by the compiler depending on the file extensions.

Table 2-1. *File Extensions and Their Interpretation by the Intel Compiler*

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

The Intel compiler can be invoked as follows:

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

where

`<compiler name>` is one of the compiler names such as icc, icpc, ifort;

`[options]` are options that are passed to the compiler and can control code generation, optimization, and output file names, type, and path.

If no `[options]` are specified, the compiler invokes some default options, such as –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`, and so forth. Please refer to the compiler manual for details.

Compiler options play a significant role in tuning Intel MIC architecture, as you can control various aspects of code generation such as loop unrolling and prefetch generation.

You might find it useful to look at the assembly code generated by the compiler. You can use the –S option to generate the assembly file to see the assembly-coded output of the compiler.

Setting Up an Intel Xeon Phi System

I will assume you have access to a host with one or more Intel Xeon Phi cards installed in it and one of the supported Linux OSs on the host. (For Windows please refer to the corresponding user's guide.) There are two high-level packages: the drivers (also known as the Manycore Platform Software Stack (MPSS) package) and the development tools and libraries packages (distributed as Intel Cluster Studio or a single-node version of Intel Composer XE) from which to build a system and in which you can develop applications for Intel Xeon Phi.

Install the MPSS Stack

Install the MPSS stack by the following steps:

1. Go to the Intel Developer Zone web page (`http://software.intel.com/mic-developer`), go to the tab Tools & Downloads, and select "Intel Many Integrated Core Architecture (Intel MIC Architecture) Platform Software Stack." Download the 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 running on the Intel Xeon Phi coprocessor is provided by a standard network interface. The interface uses a virtual network driver over the PCIe bus. The Intel Xeon Phi coprocessor's 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 their machine. If you have connected to the other machine from this host through ssh you most probably have it. If you do not have an ssh key, execute the following code to generate the ssh key:

   ```
   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 an 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:

   ```
   command 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 `command_prompt>/opt/intel/mic/bin/micinfo`, which will print out the Intel Xeon Phi–related information including the flash file.[1]

   The flash files can be found in the folder `/opt/intel/mic/flash` and should match with those printed out as part of the micinfo. If the installed version is older than the one available with the new MPSS you are installing, update the flash with the micflash utility. Please refer to the `readme.txt` provided with the documentation to select the proper flash file. Once you have determined the proper flash file for the revision of the card on your system, use the 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 take effect. The MPSS is installed as a Linux service and can be started or stopped by `service mpss start|stop|restart` commands.

10. Note that you need to have the proper driver configuration to get the card started. A mismatched card configuration from a previous install could prevent the card from booting. I would strongly suggest you read up on MPSS configuration and micctrl utility in the `readme.txt` if you encounter any issue starting the card.

Install the Development Tools

Install the Intel C/C++ compiler by obtaining the Intel Composer XE package or a superset of this package such as Intel Cluster Studio XE for Linux. Follow the instructions provided in the link in the "Install the MPSS Stack" section on how to get access to the tools and for step-by-step methods to install. Since these steps are the same as for installing the Intel tools on any Intel Xeon processor-based hosts, I am not going to cover them here.

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, compiler engineers had to make changes at various levels.

The first step was adding new language features that allow you to describe the offload syntax for a code fragment that can be sent to the coprocessor. These language features are introduced by the new OpenMP 4.0 Technical Report as well as the Intel proprietary C/CFortran extensions.

[1] The default MPSS install puts most of the Intel Xeon Phi–related drivers, flash, and utilities in the `/opt/intel/mic` and `/usr/sbin` default paths. If your system administrator put it somewhere else, you need to check with that person. I shall assume everything is installed in the default folder.

Second, Intel 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, as discussed in Chapter 3. These intrinsics will help us explore various architectural features of the hardware. Finally, we need to provide support for the new environment variables and libraries to allow execution of the offload programs on the hardware.

There are two predominant programming models for Intel Xeon Phi. One is the *native execution mode* by which you cross-compile the code written for Intel Xeon processors and run the resulting executable on the Xeon Phi micro OS. The other is the *heterogeneous* or *hybrid mode*, by which you start the main code on the host and the code executes on the coprocessor or both the host and the coprocessor.

Native Execution Mode

Being chiefly concerned with architecture, this book mainly deals with the native execution mode, as this is the simplest way to run something on the Xeon Phi hardware. In native execution mode, you can run any C/C++/Fortran source that can be compiled for Intel Xeon or can be cross-compiled for Intel Xeon Phi architecture. The code is then transferred to the Xeon Phi coprocessor OS environment using familiar networking tools such as *secure copy* (scp) and executed in Xeon Phi's native execution environment.

In this section I shall assume you already have access to a system with an Intel Xeon Phi coprocessor installed in it. I shall also assume you have downloaded the appropriate driver and Composer XE on your machine from the Intel registration center.

Hello World Example

Let's try running a simple "Hello world" application on the 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 these files with a `-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 the 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.

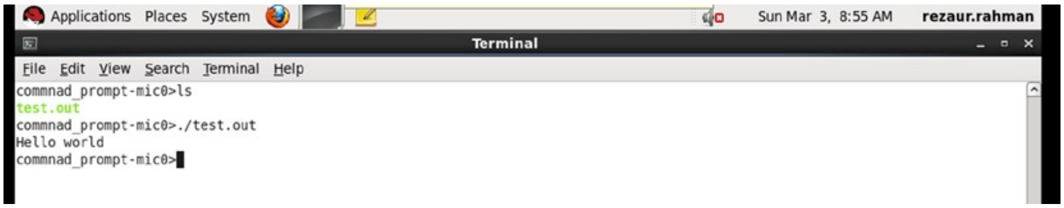
At this phase you can log in to the coprocessor using the `ssh` command as follows:

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

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

```
command_prompt-mic0>./test.out
Hello world   //printed from Intel Xeon Phi
command_prompt-mic0>
```

I have also captured a screenshot of this execution, as shown in Figure 2-3.


Applications Places System
Sun Mar 3, 8:55 AM
rezaur.rahman
Terminal
File Edit View Search Terminal Help

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

Figure 2-3. *Executing a Hello world program inside Intel Xeon Phi micro OS command prompt. The printout is from the code running on a Xeon Phi coprocessor*

Language Extensions to Support Offload Computation on Intel Xeon Phi

Intel has extended the C/C++ and Fortran language to support programming the Intel Xeon Phi coprocessor. OpenMP 4.0 standard also includes extensions to target similar coprocessor computing models.

Heterogeneous Computing Model and Offload Pragmas

There are two supported ways to deal with nonshared memory between an Intel Xeon Phi coprocessor and the host processor. One is *data marshaling*, in which the compiler runtime generates and sends data buffers over to coprocessors by marshaling the parameters provided by the application. The second option is the *virtual shared memory model*, which depends on system-level runtime support to maintain data coherency between the host's and coprocessor's virtual shared memory address space. Because the first model is supported by OpenMP 4.0 specifications and is more prevalent than the second model, the first model will be the working model in all subsequent chapters.

The Intel compiler had proprietary language extensions that were implemented to support the nonshared programming model before the OpenMP 4.0 specifications were published.

In the nonshared virtual memory space 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 copyable. The bitwise copyable data structures include 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 added to the source code.

The data selected for offload may be implicitly copied if used inside the offload code block and provided the variables are in the lexical scope of the code block performing the offload and the variables are listed explicitly as part of the pragmas. The requirement is that the data must have a flat structure. However, the data that are used only within the offload block can be arbitrarily complex with multiple indirections but cannot be passed between host and the coprocessor. If this is needed, you have to marshal the data into a flat data structure or buffer the data to move them back and forth. The Intel compiler does support transparent marshaling of data using shared virtual memory constructs (described in Chapter 8), which can handle more complex data structures.

In order to provide seamless execution so the same code can run on the host processor with or without a coprocessor, the compiler runtime determines whether or not the coprocessor is present on the system. 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 nonshared memory programming, we should understand the terminology introduced in OpenMP 4.0 specifications to describe the language support.[2] The treatment of the terminology and directives in this section is not comprehensive but rather covers those parts of the specifications that are relevant to the Intel Xeon Phi offload programming model.

Terminology

device. A *device* may have one or more co-processors with their own memories or a host. A *host device* is the device executing the main thread. A *target device* executes the offloaded code segment.

offload. The process of sending a computation from host to target.

data environment. The variables associated with a given execution environment.

device data environment. A data environment associated with *target data* or a *target construct*.

mapped variable. Either variable when a variable in a data environment is mapped to a variable in a device data environment. The original and corresponding variables may share storage.

mappable type. A valid *data type* for a mapped variable.

Offload Function and Data Declaration Directives

These directives are used to declare functions and variables so that these are available on the coprocessor.

declare target Directives

`declare target` directives declare data, functions, and subroutines that should be available in a target (coprocessor) execution environment. They allow the 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
 declaration-definition-sequence
#pragma omp end declare target new-line
```

[2] http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf

Fortran

`!$omp declare target` *`(list) new-line`*

In C/C++/Fortran, variables and function declarations that appear between `declare` and `end declare target` or in the `list` argument are created in the device context and can be used or executed in the target region.

Restrictions

- Threadprivate variables cannot be in a `declare target` directive.
- Variables declared in a `declare target` directive must have a mappable type.
- In C/C++ the variables declared in a `declare target` directive must be at file or namespace scope.

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 the nonshared memory model, the host application executes the initial program, which spawns the master execution thread called the *initial device thread*. The OpenMP `pragma target` and `pragma target data` provide the capability to offload computations to a coprocessor(s).

Target Data Directive

This directive causes a new device data environment to be created and, encountering the task, executes the target data region.

Syntax

C/C++

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

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 coprocessors available on a host. If no device is specified, the default device is determined by internal control variable (ICV) named `default-device-var` (`OMP_DEFAULT_DEVICE openmp` environment variable). The default data environment is constructed from the enclosing device environment, the data environment of the enclosing task, and the data mapping clauses in the construct.

- map([map-type:]*list*)
    - These are data motion clauses that allow copying and mapping of variables or common block to or from the host scope to the target device scope. The map type:
    - `alloc` indicates the data are allocated on the device and have an undefined initial value.
    - `to` declares that on entering the region, each new data in the list will be initialized to original list item value.
    - `from` declares that the data elements are “out” type and copied from the device data to host data on exit from the region.
    - `tofrom(Default)` declares that data elements are in or out type and values are copied to and from the data elements in the device corresponding to data elements on the host.
    - If the list is an array element, it must be a contiguous region.
- if(scalar-expr)
    - if the scalar-expression evaluates to false, the device is a host.

Restrictions

- 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.

Target Directive

This directive provides a superset of functionality and restriction of the target data directive, as discussed in the previous subsection.

- 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, 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
structured-block
```

Fortran

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

where clauses are:

- device(scalar-integer-expression)
  - The integer expression must be a positive number to differentiate various coprocessors available on a host. If no device is specified, the default device is determined by the ICV named `default-device-var` (`OMP_DEFAULT_DEVICE openmp` environment variable). The default data environment is constructed from the enclosing device environment, the data environment of the enclosing task, and the data mapping clauses in the construct.
- map([map-type:]*list*)
  - These are data motion clauses that allow copying and mapping of variables or common block to or from the host scope to the target device scope. The map type:
  - `alloc` indicates the data are allocated on the device and have an undefined initial value.
  - `to` declares that on entering the region, each new data in the list will be initialized to original list item value.
  - `from` declares that the data elements is "out" type and copied from the device data to the host data on exit from the region.
  - `tofrom(Default)` declares that data elements are in or out type and values are copied to and from the data elements in the device corresponding to data elements on the host.
  - If the list is an array element, it must be contiguous region.
- if(scalar-expr)
  - if the scalar-expression evaluates to false, the device is a host.

The target directive creates a device data environment and executes the code block on the target device. The target region binds to the enclosing structured block code region. It provides a superset of *target data constructs* 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 the logical expression inside the `if` clause evaluates to false, the target region is not executed by the device but executed on the host.

Restrictions

- If a target, target update, or target data construct appears within a target region, then the behavior is undefined.
- 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 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 declared in the target construct 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 syntax for expressing data transfers between host and coprocessors is expressed by target data and update constructs, as discussed in the next section.

Target Update Directive

Target update directive synchronizes the list items in the device data environment consistent with their corresponding original list items according to the map clause.

Syntax

C/C++

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

Fortran

```
!$omp target update motion-clause [clause[[,] clause],...]
```

Where `motion-clause` is one of the following:

```
to(list)
from(list)
```

Each list item in the `to` or `from` clause corresponds to a device item and a host list item. The `from` clause corresponds to `out` data from the device to the host and the `to` clause corresponds to `in` data from the host to the device.

Clauses are:

- device(scalar-integer-expression)
  - The integer expression must be a positive number to differentiate the various coprocessors available on a host. If no device is specified, the default device is determined by an ICV named `default-device-var`.
- if(scalar-expr): If the scalar expression evaluates to false, the `update` clause is ignored.

Runtime Library Routines

The OpenMP 4.0 specifications also provide routines to set and get runtime environment settings (referred to as ICVs in the OpenMP specs) and to query the number of available coprocessors using the following APIs:

```
void omp_set_default_device(int device_num),
int omp_get_default_device();
```

Description: Set default device for offload. This gets or sets the ICV `default-device-var`. The corresponding environment variable is `OMP_DEFAULT_DEVICE`.

```
int omp_get_num_devices();
```

Description: Query number of coprocessors in the system.

Offload Example

Listing 2-1 gives a simple example of how the constructs discussed for OpenMP 4.0 can be used to offload computation to an Intel Xeon Phi coprocessor.

Listing 2-1. Sample Test Code

```
// Sample code reduction.cpp
// Example showing use of OpenMP 4.0 pragmas for offload calculation
// This 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);
}
```

On line 13 of the listing, you will notice the offload pragma #pragma omp target map(inarray[0:SIZE]) map(sum), which causes specific code block lines (14 to 17) to be sent to the coprocessor for computing. In this case it is computing the reduction of an array of numbers and returning the computed value through the sum variable to the host. The inarray and the sum are copied in and out of the coprocessor before and after the computation.

The main routine calls the reduce function in line 32 in the code block. The code in turn offloads part of the computation to the Intel Xeon Phi to compute the sum reduction. Once the reduction is done, the results received from the coprocessor are returned to the main function (line 32) and compared against the reduction done on the 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 Listing 2-2 using:

```
command_prompt>./test.out
```

You can see the data movement by the runtime engine if you set OFFLOAD_REPORT=2 in the host environment. The output with OFFLOAD_REPORT set to 2 during runtime is shown in Listing 2-2. Here you see that there were 4012 bytes (1000 integer elements + 4 bytes sum + 8 bytes pointer of the inarray are sent from host to the target). Also because we used the map clause, the same array was sent back from coprocessor to host. Thus you can see in the report that 4004 (1000 integer + sum) bytes are returned.

Listing 2-2.

```
[Offload] [MIC 1] [File]            reduction.cpp
[Offload] [MIC 1] [Line]            13
[Offload] [MIC 1] [Tag]             Tag0
[Offload] [MIC 1] [CPU Time]        0.000000 (seconds)
[Offload] [MIC 1] [CPU->MIC Data]   4012 (bytes)
[Offload] [MIC 1] [MIC Time]        0.000177 (seconds)
[Offload] [MIC 1] [MIC->CPU Data]   4004 (bytes)

sum reduction = 499500, validSum=499500
```

Now suppose you don't want to return the inarray back to the host because it is not modified, thus saving precious bandwidth. You can do so by changing the map clause to map(to:inarray[0:SIZE]) in line 13. The output of such a modification is shown in Listing 2-3 below:

Listing 2-3.

```
[Offload] [MIC 1] [File]            reduction.cpp
[Offload] [MIC 1] [Line]            13
[Offload] [MIC 1] [Tag]             Tag0
[Offload] [MIC 1] [CPU Time]        0.000000 (seconds)
[Offload] [MIC 1] [CPU->MIC Data]   4004 (bytes)
[Offload] [MIC 1] [MIC Time]        0.000156 (seconds)
[Offload] [MIC 1] [MIC->CPU Data]   4 (bytes)
```

Here you can see only 4 bytes are transferred back to host for `[MIC->CPU data]` value.

Summary

This chapter has walked you through the set up of a Xeon Phi processor. You have also looked at programming this processor with OpenMP 4.0 specifications provided for coprocessor programming. You will use this knowledge in the next chapter to explore the Xeon Phi hardware features. Chapter 8 will discuss in depth the other programming languages and tools provided by Intel for developing the Xeon Phi coprocessor.