White Paper

Intel® Xeon Phi™ Coprocessor
DEVELOPER'S QUICK START GUIDE

Version 1.8

Contents

Introduction

This document will help you get started writing code and running applications on a system (host) that includes the Intel® Xeon Phi™ coprocessor based on the Intel® Many Integrated Core Architecture (Intel® MIC Architecture). It describes the available tools and includes simple examples to show how to get C/C++ and Fortran-based programs up and running.

This document is available at http://software.intel.com/mic-developer under the "*Overview*" tab.

Goals

This document does:

1. Walk you through the Intel® Manycore Platform Software Stack (Intel® MPSS) installation.
2. Introduce the build environment for software enabled to run on Intel Xeon Phi coprocessor.
3. Give an example of how to write code for Intel Xeon Phi coprocessor and build using Intel® Parallel Studio XE 2015.
4. Demonstrate the use of Intel libraries like the Intel® Math Kernel Library (Intel® MKL).
5. Point you to information on how to debug and profile programs running on a coprocessor.
6. Share some best known methods (BKMs) developed by users at Intel.

This document does not:

1. Cover each tool in detail. Please refer to the user guides for the individual tools.
2. Provide in-depth training.

Terminology

Host – The Intel® Xeon® platform containing the Intel Xeon Phi coprocessor installed in a PCIe* slot. As of Intel MPSS 3.4, the operating systems (OS) supported on the host are Red Hat Enterprise Linux* 6.3, Red Hat Enterprise Linux* 6.4, Red Hat Enterprise Linux* 6.5, Red Hat Enterprise Linux* 6.6, Red Hat Enterprise Linux* 7.0, SUSE Linux* Enterprise Server SLES 11 SP2, and SUSE Linux* Enterprise Server SLES 11 SP3. The user will have to install the OS.

Target – The Intel Xeon Phi coprocessor and corresponding run-time environment installed inside the coprocessor.

Micro Operating System (uOS) – The Linux-based operating system and tools running on the Intel Xeon Phi coprocessor.

Instruction Set Architecture (ISA) – The part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O (Input/Output)

Vector Processing Unit (VPU) – The portion of a CPU responsible for the execution of SIMD (single instruction, multiple data) instructions.

Native Acceleration (NAcc) – A mode or form of Intel MKL in which the data being processed and the Intel MKL function processing the data reside on the coprocessor.

Offload Compilers – The Intel® C/C++ Compiler and Intel® Fortran Compiler can generate binaries for both the host system and the Intel Xeon Phi coprocessor. The offload compilers can generate binaries that will run only

on the host, only on the coprocessor, or paired binaries that run on both the host and the coprocessor and communicate with each other.

Intel MPSS –the user- and system-level software that allows programs to run on and communicate with the Intel Xeon Phi coprocessor.

Symmetric Communications Interface (SCI) – The mechanism for inter-node communication within a single platform, where an node is an Intel Xeon Phi coprocessor or an Intel Xeon processor-based host processor complex. In particular, SCI abstracts the details of communicating over the PCIe bus (and controlling related Intel Xeon Phi coprocessor hardware) while providing an API that is symmetric between all types of nodes

System Configuration

The configuration assumed in this document is an Intel workstation containing two Intel Xeon processors, one or two Intel Xeon Phi coprocessors attached to a PCIe x16 bus, and a GPU for graphics display.

Intel® Xeon Phi™ Software

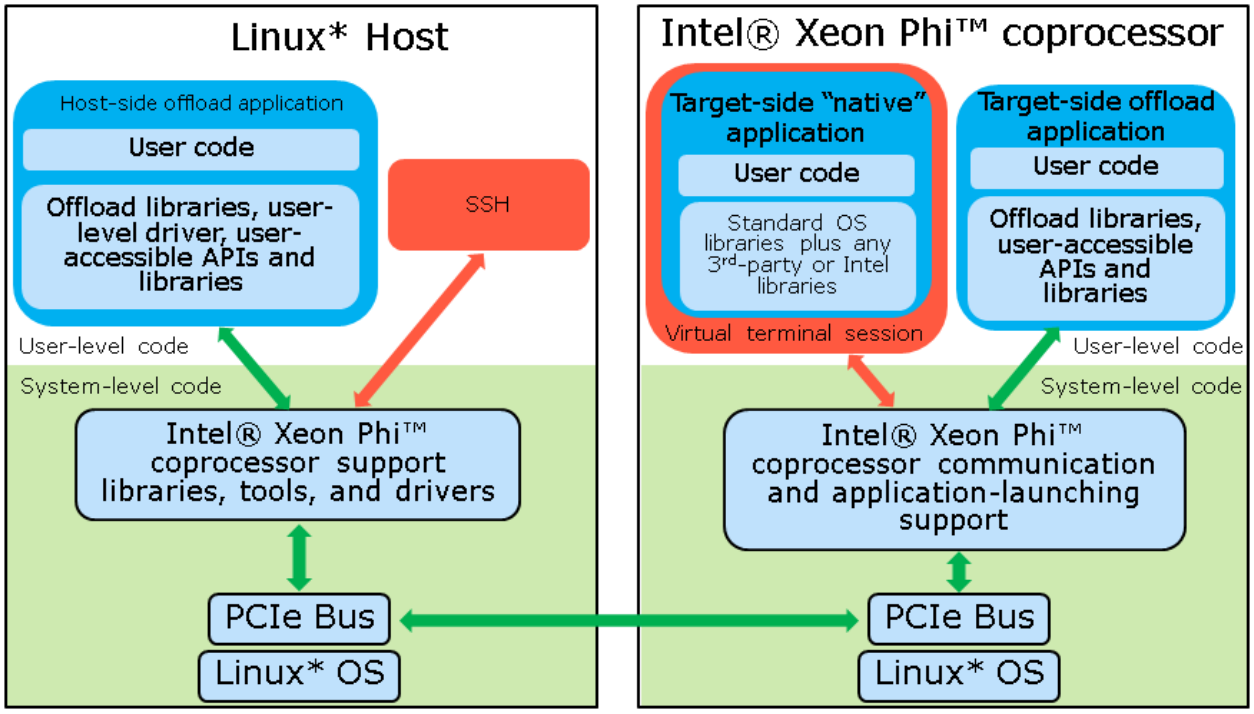


Figure 1: Software Stack

The Intel Xeon Phi coprocessor's software stack consists of layered software architecture as noted below and depicted in Figure 1.

Driver Stack:

The Linux software for the Intel Xeon Phi coprocessor consists of a number of components:

- **Device Driver:** At the bottom of the software stack in kernel space is the Intel Xeon Phi coprocessor device driver. The device driver is responsible for managing device initialization and communication between the host and target devices.

- **Libraries:** The libraries live on top of the device driver in user and system space. The libraries provide basic card management capabilities such as enumeration of cards in a system, buffer management, and host-to-card communication. The libraries also provide higher-level functionality such as loading and unloading executables onto the coprocessor, invoking functions from the executables on the card, and providing a two-way notification mechanism between host and card. The libraries are responsible for buffer management and communication over the PCIe bus.
- **Tools:** Various tools that help maintain the software stack. Examples include /usr/bin/**micinfo** for querying system information, /usr/bin/**micflash** for updating the card's flash, /usr/sbin/**micctrl** to help administrators configure the card, etc.
- **Card OS (uOS)**: The Linux-based operating system running on the Intel Xeon Phi coprocessor.

NOTE: Source for relatively recent versions of the uOS, the device driver, and the low-level SCI library interface can be found at http://software.intel.com/mic-developer .

Intel® Many Integrated Core Architecture Overview

The Intel Xeon Phi coprocessor has up to 61 in-order Intel MIC Architecture processor cores running at 1GHz (up to 1.3GHz). The Intel MIC Architecture is based on the x86 ISA, extended with 64-bit addressing and new 512-bit wide SIMD vector instructions and registers. Each core supports four hardware threads. In addition to the cores, there are multiple on-die memory controllers and other components.

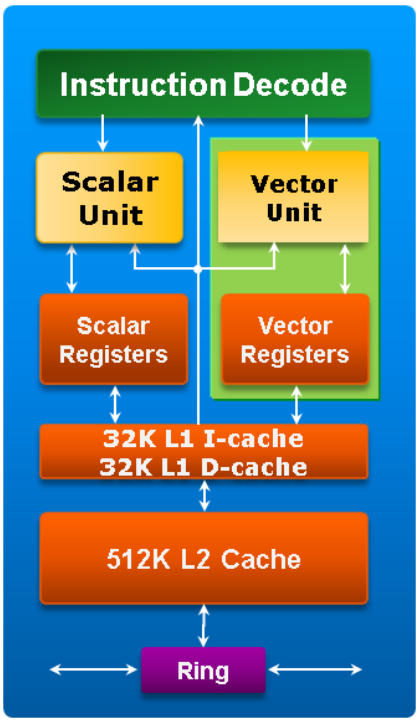


Figure 2: Architecture overview of an Intel® MIC Architecture core

Each core includes a newly designed vector processing unit (VPU). Each VPU contains 32 512-bit vector registers. To support the new vector processing model, a new 512-bit SIMD ISA was introduced. The VPU is a key feature of the Intel MIC Architecture-based cores. Fully utilizing the VPU is critical for best Intel Xeon Phi coprocessor performance. It is important to note that Intel MIC Architecture cores do not support other SIMD ISAs (such as MMX™, Intel® SSE, or Intel® AVX).

Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a 512KB L2 cache. The L2 caches of all cores are interconnected with each other and the memory controllers via a bidirectional ring bus, effectively creating a shared last-level cache of up to 32MB. The design of each core includes a short in-order pipeline. There is no latency in executing scalar operations and low latency in executing vector operations. Due to the short in-order pipeline, the overhead for branch misprediction is low.

For more details on the architecture, please refer to the *Intel® Xeon Phi™ Coprocessor Software Developer's Guide* posted at http://software.intel.com/mic-developer *under "TOOLS & DOWNLOADS" tab.*

Administrative Tasks

If you purchased the Intel Xeon Phi coprocessor from an equipment manufacturer, please go to the Intel® Developer Zone (IDZ) page http://software.intel.com/mic-developer and click the *"TOOLS & DOWNLOADS"* tab, then select "Intel® Manycore Architecture Platform Software Stack (Intel® MPSS)." This brings you to a page where you can download the latest hardware drivers and release notes for the platform.

Preparing Your System for First Use

Steps to install the driver and start the card

1. From the http://software.intel.com/mic-developer page, click the *"TOOLS & DOWNLOADS"* tab, then select "Intel® Manycore Platform Software Stack (Intel® MPSS)." Navigate to the latest version of the Intel MPSS release for Linux and download "Readme for Linux (English)" (*readme.txt*). Also download the release notes *(releaseNotes-linux.txt)* and the User's Guide for Intel MPSS.
2. You may install your system with Red Hat Enterprise Linux 6.3 64-bit kernel 2.6.32-279, Red Hat Enterprise Linux 6.4 64-bit kernel 2.6.32-358, Red Hat Enterprise Linux 6.5 64-bit kernel 2.6.32-431, Red Hat Enterprise Linux 6.6 64-bit kernel 2.6.32-504, Red Hat Enterprise Linux 7.0 64-bit kernel 3.10.0-123, SUSE Linux Enterprise Server SLES 11 SP2 kernel 3.0.13-0.27-default, or SUSE Linux Enterprise Server SLES 11 SP3 kernel 3.0.76-0.11-default (Section 2.1 in *readme.txt*). Be sure to install ssh, which is used to log in to the card's uOS.

   WARNING: When installing Red Hat, it may automatically update a new version of the Linux kernel. If this happens, you will not be able to use the pre-built host driver, but will need to rebuild it manually for the new kernel version. Please see section 2.1 in *readme.txt* for instructions on building an Intel MPSS host driver for a specific Linux kernel.
3. Log in as root.
4. Download the appropriate release driver for your operating system in step 1 (<mpss-version>-linux.tar) where <*mpss-version*> is mpss-3.4 at the time this document was updated.
5. Install the host driver RPMs as detailed in section 2.2 of *readme.txt.* Don't skip the creation of configuration files for your coprocessor.
6. Update the flash on your coprocessor(s) as detailed in section 2.4 of *readme.txt.*
7. Reboot the system.
8. Start the coprocessor (while you can set up the card to start with the host system, it will not do so by default), and then run "`micinfo`" to verify that it is set up properly:

```
sudo service mpss start[1]
sudo micctrl –w
sudo /usr/bin/micinfo
```

   - Verify that the driver version, Intel MPSS version, and flash version are correct according to the following table:

[1] For RHEL 7.0, use "`sudo systemctl start mpss`" instead.

Table 1: Corresponding Driver Version, Intel® MPSS Version, and Flash Version found in each Intel MPSS release.

Intel® MPSS stack installed	Driver Version	Intel MPSS Version	Flash Version
mpss-3.4	3.4-xx	3.4	2.1.02.0390
mpss-3.3	3.3-xx	3.2	2.1.02.0390
mpss-3.2	3.2-xx	3.2	2.1.03.0386
mpss-3.1	3.1-xx	3.1	2.1.03.0386
mpss_gold_update_3-2.1.6720-13	6720-13	2.1.6720-13	2.1.02.0386
KNC_gold_update_2-2.1.5889-16	5889-16	2.1.5889-16	2.1.05.0385
KNC_gold_update_1-2.1.4982-15	4982-15	2.1.4982-15	2.1.05.0375
KNC_gold-2.1.4346-xx	4346-xx	2.1.4346-xx	2.1.01.0375

Steps to install the Software Development tools

You can purchase software development tools at http://software.intel.com/en-us/linux-tool-suites. Select the tool(s) that fit(s) your need (e.g., "Intel® Parallel Studio XE 2015 Cluster Edition" or the Intel® Parallel Studio XE Professional Edition). After selecting the tools you need and completing the purchasing process, you will receive a serial number. Alternatively, visit http://software.intel.com/en-us/mic-developer, under "Tools and Downloads" select the "Intel® Software Development Products" to find the latest list of supported tools for the Intel Xeon Phi coprocessor.

If you acquired a serial number for Intel tools, go to the Intel® Registration Center (IRC) at http://registrationcenter.intel.com to register and download the products. Clicking the button "Register Product" will bring you to the download page of the tool(s) you purchased. The following example shows buying the Intel Parallel Studio XE 2015 Cluster Edition: from http://software.intel.com/en-us/intel-cluster-studio-xe/. Under the Documentation tab, you can get the Install Guide, Getting Started Guide, and Release Notes.

1. Follow the instructions in the Install Guide to install the Intel Parallel Studio XE Cluster Edition for Linux. If you bought the Intel Parallel Studio XE Composer Edition for Linux, read the corresponding Install Guide to install these packages, as well as separately installing Intel® VTune™ Amplifier XE for Linux.
    - For first-time installations, be sure to get the product license number described above that is required to activate the product, and then provide the license number during installation. Subsequent installations can select the "Use existing license" option.
    - Read the release notes of the product (*ipsxe2015-cluster-edition-release-notes.pdf* if you bought the Intel Cluster Studio XE for Linux, or *intel-parallel-studio-xe-2015-composer-edition-release-notes.pdf* if you bought the Intel Parallel Studio XE Composer Edition for Linux.
    - Untar the product file.

- `tar -xvzf parallel_studio_xe_2015.<update>.<package_num>.tgz`, or
- `tar -xvf l_composer_2015.<update>.<package_num>.tgz`, or

2. Install the software tools using the previously acquired serial number.
3. Verify that the card is working by running a sample program (located in `/opt/intel/composer_xe_2015.*.*/Samples/en_US/C++/mic_sample` for C/C++ code or in `/opt/intel/composer_xe_2015.*.*/Samples/en_US/Fortran/mic_sample` for Fortran code) with "`setenv H_TRACE 2`" or "`export H_TRACE=2`" to display the dialog between the host and the coprocessor (messages from the processor will be prefixed with "MIC:"). If you do see a dialog, then everything is running fine and the system is ready for general use.
4. If you intend to collect performance data on this system using Intel VTune Amplifier XE 2015:
   a) After Intel MPSS gets started, it loads the data collection driver automatically. But for some reason, if it fails to load the data collection driver, you can manually load the driver by going to /opt/intel/vtune_amplifier_xe/bin64/k1om/ and running:

```
sudo sep_micboot_install.sh
```

   b) Start (or restart) the Intel MPSS service (this also starts the sampling driver once the files are copied in the previous step):

```
sudo service mpss restart
sudo micctrl -r
sudo micctrl -w
```

   The coprocessor has successfully restarted when `micctrl -w` reports "`micx: online`".
   c) The sampling driver will now start every time the coprocessor is restarted.
   d) If you ever need to reinstall the sampling driver, do the following:

```
sudo service mpss stop
sudo sep_micboot_uninstall.sh
sudo service mpss restart
sudo micctrl -w
```

Updating an Existing System

Updating a system that already has an Intel® Xeon Phi™ Coprocessor

1. From the http://software.intel.com/mic-developer page, click the "*TOOLS & DOWNLOADS*" tab, then select "Software Drivers: Intel® Manycore Platform Software Stack (Intel® MPSS)". Download "Readme file for the Intel® MPSS release" (*readmetxt*). Also download the release notes *(releaseNotes-linux.txt).*

2. Uninstall the previous version of the Intel MPSS and install the new one using the instructions in sections 2.3 and 2.2 of *readme.txt* .
3. Update the flash on your card(s) as detailed in section 2.4 of *readme.txt.*
4. Reboot the system.
5. Start the Intel Xeon Phi coprocessor (while you can set up the card to start with the host system, it will not do so by default), and then run "`micinfo`" to verify that it is set up properly:

```
sudo service mpss start
sudo micctrl -w
/usr/bin/micinfo
```

- Verify that the driver version, Intel MPSS version, and flash version are correct according to table 1 in the previous section.

Regaining Access to the Intel® Xeon Phi™ Coprocessor after Reboot

The coprocessor will not start when the host system reboots. You will need to manually start it, and then run "`micinfo`" to verify that it started properly. You may need to add `/usr/sbin` and `/sbin` to your path to do this successfully as a non-root user via `sudo`:

```
sudo service mpss start
sudo micctrl -w
/usr/bin/micinfo
```

Note: It is possible to make the coprocessor uOS automatically start on reboot and preload desired files. See section 20.12 of the *Intel MPSS User's Guide* for details

Restarting the Intel® Xeon Phi™ Coprocessor If It Hangs

If a process running on the coprocessor hangs, but it is otherwise responsive via `ssh`, log onto the coprocessor and kill the process like any other Linux process.

When a coprocessor hangs and is inaccessible or unresponsive via `ssh`, there are two ways to restart it. But first, see if you can tell what is happening:

```
sudo micctrl --status <micx>
```

Assuming that the Intel MPSS service is still functioning properly, you can try to restart the coprocessor without affecting any other attached coprocessors as follows:

```
sudo micctrl --reset <micx>
```

```
sudo micctrl --boot <micx>
sudo micctrl -w
/usr/bin/micinfo
```

If the Intel MPSS service is not running properly, then you need to restart the driver and all connected coprocessors:

```
sudo service mpss stop
sudo service mpss unload
sudo service mpss start
sudo micctrl -w
/usr/bin/micinfo
```

Monitoring the Intel® Xeon Phi™ Coprocessor

If you want to monitor the load on your coprocessor, its temperature, etc., run the System Management and Configuration (SMC) utility. See section 8.3 of the *Intel MPSS User's Guide* for details:

Execute the monitor

```
/usr/bin/micsmc &
```

When started with no arguments, `micsmc` will run in GUI-mode. When invoked with arguments, it will run in character-mode.

Running an Intel® Xeon Phi™ Coprocessor program from the host system

It is possible to copy an Intel MIC Architecture native binary to a specified coprocessor and execute it using the "`micnativeloadex`" utility. This utility conveniently copies library dependencies to the coprocessor. See section 8.5 of the *Intel MPSS User's Guide* for details.

Working directly with the uOS Environment Intel® Xeon Phi™ Coprocessor

Since the coprocessor is running Linux and is effectively a separate network node, root, or non-root users can log into it via "`ssh`" and issue many common Linux commands. Files are transferred to/from the coprocessor using "`scp`" or other means.

The default IP address for the coprocessor as seen from the host is `172.31.<coprocessor>.1`, while the coprocessor sees the host at `172.31.<coprocessor>.254` by default. The coprocessor can also be referred to from the host by the alias `mic<coprocessor>`. For example, the first coprocessor you install in your

system is called "`mic0`" and is located at `172.31.1.1`. It sees the host as `172.31.1.254`. If a second coprocessor were installed, it would be called "`mic1`" and located at `172.31.2.1`, and it would see the host as `172.31.2.254`.

For detailed information on setting up the card for non-root users, adjusting the network configuration, mounting an NFS file system exported by the host for use on the coprocessor, etc., please see the *Intel MPSS User's Guide*.

Useful Administrative Tools

This product ships with the following administrative tools, found in the "`/usr/bin`" directory. Be sure root and users needing to use these tools have this directory added to their default path:

- **micinfo** - provides information about host and coprocessor system configuration.
- **micflash** - updates the flash on the coprocessor; saves and retrieves the version and other information for each section of the flash.
- **micsmc** - eases the burden of monitoring and managing Intel Xeon Phi coprocessors.
- **miccheck** –verifies a coprocessor's configuration by running various diagnostic tests.
- **micnativeloadex** – copies an Intel® MIC Architecture native binary to a specified coprocessor and executes it.
- **micctrl** – helps the system administrator configure and restart the coprocessor.
- **micrasd** – runs on the host and handles and logs hardware errors.
- **mpssflash** – the POSIX version of *micflash*.
- **mpssinfo** – the POSIX version of *micinfo*.

Please see section 8 in the *Intel MPSS User's Guide* for details on these tools and their arguments.

Getting Started/Developing Intel® Xeon Phi™ Software

You develop applications for the Intel MIC Architecture using your existing knowledge of multi-core and SIMD programming. The offload language extensions allow you to port sections of your code (written in C/C++ or Fortran) to run on the coprocessor, or you can port your entire application to the Intel MIC Architecture. Best performance will only be attained with highly parallel applications that also use SIMD operations (generated by the compiler or using compiler intrinsics) for most of their execution.

Available Software Development Tools / Environments

You can start programming the Intel Xeon Phi coprocessor using your existing parallel programming knowledge and the same techniques you use to develop parallel applications on the host. New tools were not created to support development directly on the coprocessor; rather, the familiar host-based Intel tools have been extended to add support for the Intel MIC Architecture via a few additions to standard languages and APIs. However, to make best use of the development tools and to get the best performance from the coprocessor, it is important to understand the Intel MIC Architecture.

Development Environment: Available Compilers and Libraries

- **Compilers**

  - Intel C++ Composer XE 2015 for building applications that run on Intel® 64 architecture and Intel MIC Architecture
  - Intel Fortran Composer XE 2015 for building applications that run on Intel 64 architecture and Intel MIC Architecture
- **Libraries** packaged with the compilers include:
  - Intel MKL optimized for the Intel MIC Architecture
  - Intel® Threading Building Blocks (Intel® TBB)
  - Intel® Integrated Performance Primitive (Intel® IPP)
- **Libraries** packaged separately include:
  - Intel® MPI Library for Linux OS including Intel MIC Architecture
  - Intel® Trace Collector and Analyzer
  - Intel® SDK for OpenCL™ Applications 2014 available at: http://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe

Development Environment: Available Tools

In addition to the standard compilers and Intel libraries, the following tools are available to help you debug and optimize software running on the coprocessor.

- **Debugger**
  - Intel® Debugger for applications running on the Intel 64 architecture and Intel MIC Architecture
  - Intel® C++ Eclipse* Product Extension including Debugging
- **Profiling**
  - Intel VTune Amplifier XE 2015 for Linux, which is used on the host Linux OS to collect and view performance data collected on the coprocessor.
  - Intel® Inspector 2015, which is used to detect memory and threading errors for serial and parallel applications.
  - Intel® Advisor 2015, which is used to assist developers to design threads.

General Development Information

Development Environment Setup

- To set up your development environment for use with the Intel tools, you need to source the following script (the default install locations are assumed):
  - **Intel C++ and Fortran Composer XE 2015**: `/opt/intel/composerxe/bin/compilervars.csh` or `compilervars.sh` script with `intel64` as the argument, e.g.
    ```
    source /opt/intel/composer_xe_2015/bin/compilervars.sh intel64
    ```

  The following scripts are run as a result of calling the `compilervars` script. To get your environment properly initialized, it is advisable not to run them individually (among other things, there are ordering issues that might result in unpredictable behavior).
  - **Intel TBB:** `/opt/intel/composer_xe_2015/tbb/bin/tbbvars.csh` or `tbbvars.sh` with `intel64` as the argument.
  - **Intel MKL:** `/opt/intel/composer_xe_2015/mkl/bin/mklvars.csh` or `mklvars.sh` with `intel64` as the argument.

Documentation and Sample Code

- The most useful documentation can be found in `/opt/intel/composer_xe_2015/Documentation/en_US/` including:
  - **`compiler_c/index.htm`** and **`compiler_f /index.htm`** - complete documentation for Intel C++ Compiler XE 2015 and the Intel Fortran Compiler XE 2015.
    - Most information on how to build for the Intel MIC Architecture can be found in the "Key Features/Intel® MIC Architecture" section under "Programming for the Intel® MIC Architecture".
    - Information on Intel MIC Architecture intrinsics can be found in the "Compiler Reference/Intrinsics" section under "Intrinsics for Intel® MIC Architecture".
  - **`Release_Notes_*_2015_L_EN.pdf`** - please read these carefully for known issues and their workarounds, plus installation instructions, for all the tools with Intel MIC Architecture support.
    - **Note**: For various reasons, this document can miss some last-minute updates. The most recent *Release_NOTES_*_2015_L_EN.pdf* documents can be downloaded from the Intel Registration Center (see the "Steps to install the Software Development tools" section).
  - **`debugger/debugger_documentation.htm`** – Information on how to use the Debugger. You will find information specific to debugging Intel MIC Architecture applications under the "Starting GDB for Intel® Phi™ Coprocessor Applications" section of the document gdb_quickstart_lin.pdf .
- Other documentation that includes sections on using the coprocessor:
  - The Intel MKL User's Guide, which can be accessed via **`mkl_documentation.htm`** found in `/opt/intel/composer_xe_2015/Documentation/en_US/mkl/mkl_userguide/index.htm`, contains a section called "Using the Intel® Math Kernel Library on Intel® MIC Core Architecture Coprocessors" which describes both "Automatic Offload" and "Compiler Assisted Offload" for Intel MKL functions.
  - Information on collecting performance data on the coprocessor using the VTune Amplifier XE 2015 for Linux can be found in `/opt/intel/vtune_amplifier_xe_2015/documentation/en/tutorials/find_lw_hotspots/C++/index.htm`
- Useful documentation on the Web:
  - On the http://software.intel.com/mic-developer website you will find a wide range of documentation that can be downloaded, most notably the ***Intel® Xeon Phi™ Coprocessor Software Developers Guide*** under ***"TOOLS & DOWNLOAD"*** tab, as well as including the ***Intel® Xeon Phi™ Performance Monitor Units*** *(*under ***"PROGRAMMING"*** tab*), and* the ***Intel® Xeon Phi™ Coprocessor Instruction Set Reference Manual*** (under the ***"OVERVIEW"*** tab) . From this site you will also be able to find a community forum to ask questions, links to other available tools, code samples, and case studies.
  - http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture contains a wealth of information on compilers.
- Some sample offload code using the explicit memory copy model can be found in:
  - **C++**: `/opt/intel/composer_xe_2015/Samples/en_US/C++/mic_samples/intro_sampleC/`

- **Fortran**: `/opt/intel/composer_xe_2015/Samples/en_US/Fortran/mic_samples/`
- **Intel MKL**: `/opt/intel/composer_xe_2015/mkl/examples/mic*`
  - For examples of Intel MKL automated offload: `/opt/intel/composer_xe_2015/mkl/examples/mic_ao/blasc` and `…/mic_ao/blasf`
  - The rest of the samples demonstrate use of Intel MKL via compiler-assisted offload (`/opt/intel/composer_xe_2015/mkl/examples/mic_offload`).

- Some sample offload code using the implicit memory copy model can be found in:
  - **C**: `/opt/intel/composer_xe_2015/Samples/en_US/C++/mic_samples/shrd_sampleC` and `…/LEO_tutorial`
  - **C++**: `/opt/intel/composer_xe_2015/Samples/en_US/C++/mic_samples/shrd_sampleCPP`

Build-Related Information

- The offload compiler produces "fat" binaries and `.so` files that contain code for both the host and coprocessor.
- The offload compiler produces code that examines the run-time execution environment for the presence of a coprocessor. The offload compiler will create both host and Intel MIC Architecture versions of all code marked for offload.
- A number of workarounds and hints can be found in *releaseNotes-linux.txt*.

Compiler Switches and Makefiles

When building applications that offload some of their code to the coprocessor, it is possible to cause the offloaded code to be built with different compiler options from the host code. The method of passing these options to the compiler is documented in the compiler documentation under the "Compiler Reference/Compiler Options/Compiler Option Categories and Descriptions" section. Look for the `-offload-option` compiler switch. In that same section, also look up the `-offload-attribute-target` compiler switch, which provides an alternative to editing your source files in some situations (applies to the pragma-based offload methods). Finally, `-no-offload` provides a way to make the compiler ignore the `_Cilk_offload` and `#pragma_offload` constructs (which cause it by default to build a heterogeneous binary).

Debugging During Runtime

To debug offload activity, the following environment variables are available:

- To learn whether offload portions of the program are running on the host or coprocessor

  For csh - `setenv H_TRACE 1`

  For sh - `export H_TRACE=1`

- For more complete debug information

  For csh - `setenv H_TRACE 2`

  For sh - `export H_TRACE=2`

- To print the compiler's internal offload timers, a value of 1 reports just the time the offload took measured by the host and the amount of computation time done by the coprocessor. A value of 2 adds information on how much data was transferred in either direction.

  For csh - `setenv OFFLOAD_REPORT <1 or 2>`

  For sh - `export OFFLOAD_REPORT=<1 or 2>`

Details can be found in the compiler documentation in the "Compilation/Setting Environment Variables" section.

Where to Get More Help

If you have any questions, you can post them on the Forum at: http://software.intel.com/en-us/forums/intel-many-integrated-core.

Using the Offload Compiler – Explicit Memory Copy Model

In this section, a reduction is used as an example to show a step-by-step approach for developing applications for the Intel Xeon Phi coprocessor using the offload compiler. The offload compiler is a heterogeneous[2] compiler, with both host CPU and target compilation environments. Code for both the host CPU and coprocessor is compiled within the host environment, and offloaded code is automatically run within the target environment. The offload behavior is controlled by compiler directives: pragmas in C/C++ and directives in Fortran.

Some common libraries, such as the Intel MKL, are available in host versions as well as target versions. When an application executes its first offload and the target is available, the runtime loads the target executable onto the coprocessor. It also initializes the libraries linked with the target code. The loaded target executable remains in the target memory until the host program terminates. Thus, any global state maintained by the library is maintained across offload instances.

Note: Although you can specify the region of code to run on the target, there is no guarantee of execution on the coprocessor. Depending on the presence of the target hardware or the availability of resources on the coprocessor when execution reaches the region of code marked for offload, the code may or may not run on the coprocessor.

The following code samples show several versions of porting reduction code to the coprocessor using the offload pragma directive.

Reduction

The operation refers to computing the expression:

```
ans = a[0] + a[1] + … + a[n-1]
```

[2] http://dictionary.reference.com/browse/heterogeneous

Host Version:

The following sample code shows the C code to implement this version of the reduction.

```
float reduction(float *data, int size)
{
    float ret = 0.f;
    for (int i=0; i<size; ++i)
    {
        ret += data[i];
    }
    return ret;
}
```

Code Example 1: Implementing Reduction Code in C/C++

Creating the Offload Version

Serial Reduction with Offload

You can use **#pragma offload target(mic)** (as shown in the example below) to mark statements (offload constructs) that should execute on the coprocessor. The offloaded region is defined as the offload construct plus the additional regions of code that run on the target as the result of function calls. Execution of the statements on the host will resume once the statements on the target have executed and the results are available on the host (i.e., the offload will block, although there is a version of this pragma that allows asynchronous execution). The **in**, **out**, and **inout** clauses specify the direction of data to be transferred between the host and the target.

Variables used within an offloaded construct that are declared outside the scope of the construct (including the file-scope) are copied (by default) to the target before execution on the target begins and copied back to the host on completion.

For example, in the code below, the variable **ret** is automatically copied to the target before execution on the target and copied back to the host on completion. The offloaded code below is executed by a single thread on a single Intel MIC Architecture core.

```
float reduction(float *data, int size)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(size))
    for (int i=0; i<size; ++i)
    {
        ret += data[i];
    }
    return ret;
}
```

Code Example 2: Serial Reduction with Offload

Vector Reduction with Offload

Each core on the Intel Xeon Phi coprocessor has a VPU. The auto vectorization option is enabled by default on the offload compiler. Alternately, as seen in the example below, you can use the Intel® Cilk™ Plus Extended Array Notation to maximize vectorization and take advantage of the Intel MIC Architecture core's 32 512-bit registers. The offloaded code is executed by a single thread on a single core. The thread uses the built-in reduction function *__sec_reduce_add()* to use the core's 32 512-bit vector registers to reduce the elements in the array sixteen at a time.

```
float reduction(float *data, int size)
{
    float ret = 0;
    #pragma offload target(mic) in(data:length(size))
    ret = __sec_reduce_add(data[0:size]); //Intel® Cilk™ Plus
                                          //Extended Array Notation
    return ret;
}
```

Code Example 3: Vector Reduction with Offload in C/C++

Asynchronous Offload and Data Transfer

Asynchronous offload and data transfer between the host and the coprocessor is available. For details see the "About Asynchronous Computation" and "About Asynchronous Data Transfer" sections in the Intel® C++ Compiler User and Reference Guide (under "Key Features/Programming for the Intel® MIC Architecture").

For an example showing the use of asynchronous offload and transfer, refer to
`/opt/intel/composer_xe_2015`
`/Samples/en_US/C++/mic_samples/intro_sampleC/sampleC13.c`

Note that when using the Explicit Memory Copy Model in C/C++, arrays are supported, provided the array element type is scalar or a bitwise copyable struct or class. So arrays of pointers are not supported. For C/C++ complex data structure, use the Implicit Memory Copy Model. Please consult the section "Restrictions on Offload Code Using a Pragma" in the "Intel C++ Compiler User and Reference Guide" for more information.

Using the Offload Compiler – Implicit Memory Copy Model

Intel Composer XE 2015 includes two additional keyword extensions for C and C++ (but not Fortran) that provide a "shared memory" offload programming model appropriate for dealing with complex, pointer-based data structures such as linked lists, binary trees, and the like (`_Cilk_shared` and `_Cilk_offload`). This model places variables to be shared between the host and coprocessor (marked with the `_Cilk_shared` keyword) at the same virtual addresses on both machines and synchronizes their values at the beginning and end of offload function calls marked with the `_Cilk_offload` keyword. Data to be synchronized can also be dynamically allocated using special allocation and free calls that ensure the allocated memory exists at the same virtual addresses on both machines.

APIs for dynamic shared memory allocation:

```
void *_Offload_shared_malloc(size_t size);
_Offload_shared_free(void *p);
```

APIs for dynamic aligned shared memory allocation:

```
void *_Offload_shared_aligned_malloc(size_t size, size_t alignment);
_Offload_shared_aligned_free(void *p);
```

Note that this is not actually "shared memory": there is no hardware that maps some portion of the coprocessor's memory to the host system. The memory subsystems on the coprocessor and host are completely independent, and this programming model is just a different way of copying data between these memory subsystems at well-defined synchronization points. The copying is implicit, in that these synchronization points (offload calls marked with `_Cilk_offload`) do not specify what data to copy. Rather, the runtime determines what data has changed between the host and coprocessor and copies only the deltas at the beginning and end of the offload function call.

The following code sample demonstrates the use of the `_Cilk_shared` and `_Cilk_offload` keywords and the dynamic allocation of "shared" memory.

```
float * _Cilk_shared data; //pointer to "shared" memory

_Cilk_shared float MIC_OMPReduction(int size)
{
    #ifdef __MIC__
    float Result;
    int nThreads = 32;
    omp_set_num_threads(nThreads);

    #pragma omp parallel for reduction(+:Result)
    for (int i=0; i<size; ++i)
    {
        Result += data[i];
    }
    return Result;


    #else
    printf("Intel(R) Xeon Phi(TM) Coprocessor not available\n");
    #endif
    return 0.0f;
}

int main()
{
    size_t size = 1*1e6;
    int n_bytes = size*sizeof(float);
    data = (_Cilk_shared float *)_Offload_shared_malloc (n_bytes);
    for (int i=0; i<size; ++i)
    {
        data[i] = i%10;
    }

    _Cilk_offload MIC_OMPReduction(size);

    _Offload_shared_free(data);
    return 0;
}
```

Code Example 4: Using the "_Cilk_shared" and "_Cilk_offload" Keywords with Dynamic Allocation in C/C++

Note: For more examples on using the implicit memory copy model, see:

C: `/opt/intel/composer_xe_2015/Samples/en_US/C++/mic_samples/shrd_sampleC` and `…/LEO_tutorial`

C++: `/opt/intel/composer_xe_2015/Samples/en_US/C++/mic_samples/shrd_sampleCPP`

For more information, read the Intel C++ Compiler User and Reference Guides and/or the Intel Fortran Compiler User and Reference Guides.

The "Restrictions on Offload Using Shared Virtual Memory" section in the "Intel C++ Compiler User and Reference Guide" explains the restrictions of using this programming model.

Native Compilation

Applications can also be run natively on the coprocessor, in which case the coprocessor will be treated as a standalone multicore computer. Once the binary is built on the host system, copy the binary and other related binaries or data to the coprocessor's filesystem (or make them visible there via NFS).

Example:

1. Copy `openmp_sample.c` from `/opt/intel/composer_xe_2015/Samples/en_US/C++/openmp_samples/` to your home directory
2. Build the application with the `-mmic` flag:

```
icc -mmic -vec-report3 -openmp openmp_sample.c
```

3. Upload the binary to the coprocessor:

```
scp a.out mic0:/tmp/a.out
```

4. Copy over any shared libraries required by your application, in this case the OpenMP* run-time library:

```
scp /opt/intel/composer_xe_2015/lib/mic/libiomp5.so mic0:/tmp/libiomp5.so
```

5. Connect to the coprocessor with `ssh` and export the local directory so that the application can find any shared libraries it uses (in this case the OpenMP run-time library):

```
ssh mic0
export LD_LIBRARY_PATH=/tmp
```

6. This application may generate a segmentation fault if the stack size is not set correctly. To modify the stack size use:

```
ulimit –s unlimited
```

7. Go to `/tmp` and run `a.out`:

```
cd /tmp
./a.out
```

Parallel Programming Options on the Intel® Xeon Phi™ Coprocessor

Most of the parallel programming options available on host systems are also available on the coprocessor. These include the following:

1. Intel Threading Building Blocks (Intel TBB)
2. OpenMP
3. Intel Cilk Plus
4. pthreads

The following sections discuss the use of these parallel programming models in code using the offload extensions. Code that runs natively on the coprocessor can use these parallel programming models just as they would on the host, with no unusual complications beyond the larger number of threads.

Parallel Programming on the Intel® Xeon Phi™ Coprocessor: OpenMP*

There is no correspondence between OpenMP threads on the host CPU and on the coprocessor. Because an OpenMP parallel region within an offload/pragma is offloaded as a unit, the offload compiler creates a team of threads based on the available resources on the coprocessor. Since the entire OpenMP construct is executed on the coprocessor, within the construct the usual OpenMP semantics of shared and private data apply.

Multiple host CPU threads can offload to the coprocessor at any time. If a CPU thread attempts to offload to the coprocessor and it doesn't have any resources available, the code meant to be offloaded may be executed on the host. When a thread on the coprocessor reaches the "omp parallel" directive, it creates a team of threads based on the resources available on the coprocessor. The theoretical maximum number of hardware threads that can be created is 4 times the number of cores in your coprocessor. The practical limit is four less than this (for offloaded code) because the first core is reserved for the uOS and its services.

The code shown below is an example of a single host CPU thread attempting to offload the reduction code to the coprocessor using OpenMP in the offload construct.

```
float OMP_reduction(float *data, int size)
{
    float ret = 0;
    #pragma offload target(mic) in(size) in(data:length(size))
    {
        #pragma omp parallel for reduction(+:ret)
        for (int i=0; i<size; ++i)
        {
            ret += data[i];
```

```
        }
    }
    return ret;
}
```

Code Example 5: C/C++: Using OpenMP* in Offloaded Reduction Code

```
real function FTNReductionOMP(data, size)
    implicit none
    integer :: size
    real, dimension(size) :: data
    real :: ret = 0.0

!dir$ omp offload target(mic) in(size) in(data:length(size))
!$omp parallel do reduction(+:ret)
    do i=1,size
        ret = ret + data(i)
    enddo
!$omp end parallel do

    FTNReductionOMP = ret
    return
end function FTNReductionOMP
```

Code Example 6: Fortran: Using OpenMP* in Offloaded Reduction Code

Parallel Programming on the Intel® Xeon Phi™ Coprocessor: OpenMP* + Intel® Cilk™ Plus Extended Array Notation

The following code sample further extends the OpenMP example to use Intel Cilk Plus Extended Array Notation. In the following code sample, each thread uses the Intel Cilk Plus Extended Array Notation **__sec_reduce_add()**, a built-in reduction function to use all 32 of the Intel MIC Architecture's 512-bit vector registers to reduce the elements in the array.

```
float OMPnthreads_CilkPlusEAN_reduction(float *data, int size)
{
    float ret=0;
    #pragma offload target(mic) in(data:length(size))
    {
        int nthreads = omp_get_max_threads();
        int ElementsPerThread = size/nthreads;
        #pragma omp parallel for reduction(+:ret)
        for(int i=0;i<nthreads;i++)
        {
            ret =_sec_reduce_add(
                   data[i*ElementsPerThread:ElementsPerThread]);
        }
        //rest of the array
        for(int i=nthreads*ElementsPerThread; i<size; i++)
        {
            ret+=data[i];
        }
```

```
    }
    return ret;
}
```

Code Example 7: Array Reduction Using OpenMP* and Intel® Cilk™ Plus in C/C++

Parallel Programming on the Intel® Xeon Phi™ Coprocessor: Intel® Cilk™ Plus

Intel Cilk Plus header files are not available on the target environment by default. To make the header files available to an application built for the Intel MIC Architecture using Intel Cilk Plus, wrap the header files with *#pragma offload_attribute(push,target(mic))* and *#pragma offload_attribute(pop)* as follows:

```
#pragma offload_attribute(push,target(mic))
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>
#pragma offload_attribute(pop)
```

Code Example 8: Wrapping the Header Files in C/C++

In the following example, the compiler converts the *cilk_for* loop into a recursively called function using an efficient divide-and-conquer strategy.

```
float ReduceCilk(float*data, int size)
{
    float ret = 0;
    #pragma offload target(mic) in(data:length(size))
    {
        cilk::reducer_opadd<int> total;
        cilk_for (int i=0; i<size; ++i)
        {
            total += data[i];
        }
        ret = total.get_value();
    }
    return ret;
}
```

Code Example 9: Creating a Recursively Called Function by Converting the "*cilk_for*" Loop

Parallel Programming on Intel® Xeon Phi™ Coprocessor: Intel® Threading Building Blocks (Intel® TBB)

Like Intel Cilk Plus, the Intel TBB header files are not available on the target environment by default. They are made available to the Intel MIC Architecture target environment using similar techniques:

```
#pragma offload_attribute (push,target(mic))
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
#include "tbb/task.h"
#pragma offload_attribute (pop)

using namespace tbb;
```

Code Example 10: Wrapping the Intel® TBB Header Files in C/C++

Functions called from within the offloaded construct and global data required on the coprocessor should be appended by the special function attribute `__attribute__((target(mic)))`.

As an example, *parallel_reduce* recursively splits an array into subranges for each thread to work on. The *parallel_reduce* uses a splitting constructor to make one or more copies for each thread. For each split, the join method is invoked to accumulate the results.

1. Prefix the class by the macro `__MIC__` and the class name by `__attribute__((target(mic)))` if you want them to be generated for the coprocessor.

```
#ifdef __MIC__
class __attribute__((target(mic))) ReduceTBB
{
private:
    float *my_data;
public:
    float sum;

    void operator()( const blocked_range<size_t>& r )
    {
        float *data = my_data;
        for( size_t i=r.begin(); i!=r.end(); ++i)
        {
            sum += data[i];
        }
    }

      ReduceTBB( ReduceTBB& x, split) : my_data(x.my_data), sum(0) {}

      void join( const ReduceTBB& y) { sum += y.sum; }

      ReduceTBB( float data[] ) : my data(data), sum(0) {}
};
#endif
```

Code Example 11: Prefixing an Intel® TBB Class for Intel® MIC Architecture code generation in C/C++

2. Prefix the function to be offloaded to the coprocessor with `__attribute__((target(mic)))`.

```
__attribute__((target(mic)))
float MICReductionTBB(float *data, int size)
{
    ReduceTBB redc(data);
    // initializing the library
    task_scheduler_init init;
    parallel_reduce(blocked_range<size_t>(0, size), redc);
    return redc.sum;
}
```

Code Example 12: Prefixing an Intel® TBB Function for Intel® MIC Architecture code generation in C/C++

3. Use #pragma offload target(mic) to offload the parallel code using Intel TBB to the coprocessor.

```
float MICReductionTBB(float *data, int size)
{
    float ret(0.f);
    #pragma offload target(mic) in(size) in(data:length(size)) out(ret)
    ret = _MICReductionTBB(data, size);
    return ret;
}
```

Code Example 13: Offloading Intel® TBB Code to the coprocessor in C/C++

NOTE: Codes using Intel TBB with an offload should be compiled with -tbb flag instead of -ltbb.

Using Intel® MKL

For offload users, Intel MKL is most commonly used in native acceleration (NAcc) mode on the coprocessor. In NAcc, all data and binaries reside on the coprocessor. You transfer data through offload compiler pragmas and semantics to be used by Intel MKL calls within an offloaded region or function. NAcc functionality contains BLAS, LAPACK, FFT, VML, VSL, Sparse Matrix Vector, and required Intel MKL service functions. Please see the Intel MKL release documents for details on which functions are optimized and which are not supported.

The NAcc mode can also be used in native Intel MIC Architecture code. In this case the Intel MKL-shared libraries must be copied to the coprocessor before execution.

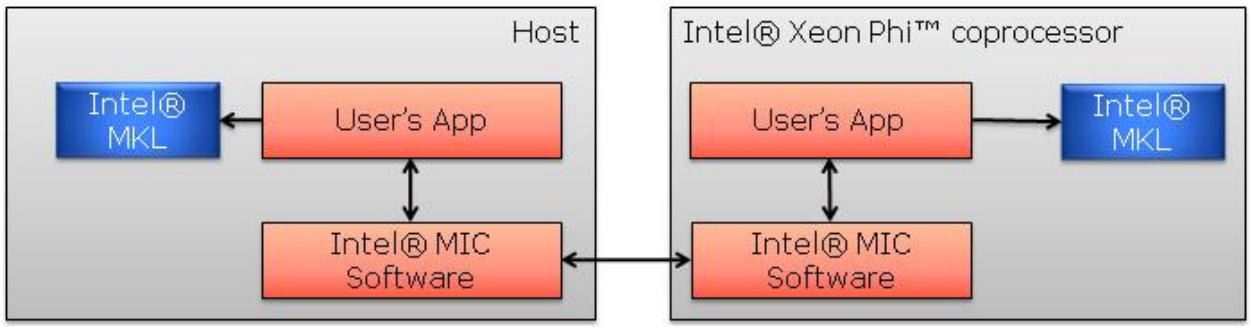


Figure 3.1: Using Intel® MKL Native Acceleration with Offload

SGEMM Sample

Below are the steps for using SGEMM routine from the BLAS library.

Sample Code – sgemm

Step 1: Initialize the matrices, which in this example need to be global variables to make use of data persistence.

Step 2: Send the data over to the coprocessor using #`pragma offload`. In this example, the `free_if(0)` qualifier is used to make the data persistent on the coprocessor.

```
#define PHI_DEV 0
#pragma offload target(mic:PHI_DEV) \
    in(A:length(matrix_elements) free_if(0)) \
    in(B:length(matrix_elements) free_if(0)) \
    in(C:length(matrix_elements) free_if(0))
    {
    }
```

Code Example 14: Sending the Data to the Intel® Xeon Phi™ Coprocessor

Step 3: Call sgemm inside the offload section to use the "Native Acceleration" version of Intel MKL on the coprocessor. The `nocopy()` qualifier causes the data copied to the card in step 2 to be reused.

```
#pragma offload target(mic:PHI_DEV) \
    in(transa, transb, N, alpha, beta) \
    nocopy(A: alloc_if(0) free_if(0)) nocopy(B: alloc_if(0) free_if(0)) \
    out(C:length(matrix_elements) alloc_if(0) free_if(0)) // output data
    {
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
            &beta, C, &N);
    }
```

Code Example 15: Calling sgemm Inside the Offload Section

Step 4: Free the memory you copied to the card in step 2. The `alloc_if(0)` qualifier is used to reuse the data on the card on entering the offload section, and the `free_if(1)` qualifier is used to free the data on the card on exit.

```
#pragma offload target(mic:PHI_DEV) \
        in(A:length(matrix_elements) alloc_if(0) free_if(1)) \
        in(B:length(matrix_elements) alloc_if(0) free_if(1)) \
        in(C:length(matrix_elements) alloc_if(0) free_if(1))
        {
        }
```

Code Example 16: Set the Copied Memory Free

As with Intel MKL on any platform, it is possible to limit the number of threads it uses by setting the number of allowed OpenMP threads before executing any functions within the offloaded code.

```
#pragma offload target(mic:PHIDEV) \
    in(transa, transb, N, alpha, beta) \
    nocopy(A: alloc_if(0) free_if(0)) nocopy(B: alloc_if(0) free_if(0))
    out(C:length(matrix_elements) alloc_if(0) free_if(0)) // output data
    {
        omp_set_num_threads(64); // set num threads in openmp
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
```

```
                &beta, C, &N);
    }
```

Code Example 17: Controlling Threads on the Intel® Xeon Phi™ Coprocessor Using omp_set_num_threads()

Intel® MKL Automatic Offload Model

A few of the host Intel® MKL functions are automatic offload-aware—you call them as you normally would on the host. However, if you have preceded the library call with a call to `mkl_mic_enable()`, Intel MKL will automatically decide at runtime whether some or all of the work required to complete the call should be divided between the host and the coprocessor. It bases this decision on problem size, the load on both processors, and other metrics. Turn this functionality off with `mkl_mic_disable()`.

Automatic offload applies only to select host Intel MKL library calls made ***outside*** of code run on the coprocessor via `_Cilk_offload` or `#pragma offload`. As a result, you should be careful to minimize transferring the same data both in automatic offload calls and in code run on the coprocessor by `_Cilk_offload` or `#pragma offload`. At present, there is no way to keep common data on the coprocessor between automatic Intel MKL offloads and explicit programmer-controlled offloads (via `_Cilk_offload` or `#pragma offload`).

An example that demonstrates how to control automatic offload can be found at `/opt/intel/composer_xe_2015/mkl/examples/mic_ao/blasc` for C code, and at `/opt/intel/composer_xe_2015/mkl/examples/mic_ao/blasf` for Fortran code.

Debugging on the Intel® Xeon Phi™ Coprocessor

You will find information specific to debugging Intel MIC Architecture applications in the document "Debugging Intel® Xeon Phi™ Application on Linux*" on this http://software.intel.com/mic-developer page, under the "PROGRAMMING" tab.

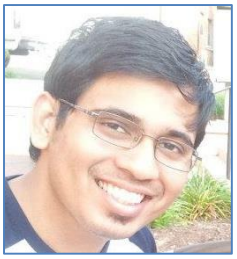
Performance Analysis on the Intel® Xeon Phi™ Coprocessor

Information on collecting performance data on the coprocessor using VTune Amplifier XE for Linux can be found in Getting Started -> Intel Xeon Phi Coprocessor Analysis Workflow section, located in `/opt/intel/vtune_amplifier_xe_2015documentation/en/help/index.htm`

About the Authors

Sudha Udanapalli Thiagarajan received a Bachelor's degree in Computer Science and Engineering from Anna University Chennai, India in 2008 and a Master's degree in Computer Engineering from Clemson University in May 2010. She joined Intel in 2010 and has been working as an enabling Application Engineer, focusing on optimizing applications for ISVs and developing collateral for Intel® Many Integrated Core Architecture.

Charles Congdon is a senior software engineer with Intel's Software and Services Group. He specializes in improving application performance and scalability and has written software and documentation for projects inside and outside Intel. Before joining Intel, Charles was a consulting software engineer for Oracle Corporation, where he concentrated on parallelism and 64-bit support in Windows* NT and OpenVMS® versions of the Oracle RDBMS on the Digital Alpha processors.



Sumedh Naik received a Bachelor's degree in Electronics Engineering from Mumbai University, India in 2009 and a Master's degree in Computer Engineering from Clemson University in December 2012. He joined Intel in 2012 and has been working as an Software Engineer, focusing on developing collateral for Intel® Xeon Phi™ coprocessor.

Loc Q Nguyen received an MBA from the University of Dallas, a master's degree in Electrical Engineering from McGill University, and a bachelor's degree in Electrical Engineering from École Polytechnique de Montréal. He is currently a software engineer with Intel's Software and Services Group. His areas of interest include computer networking, computer graphics, and parallel processing.

Notices

Performance Notice

For more complete information about performance and benchmark results, visit www.intel.com/benchmarks

Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804