White Paper

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

Version 1.0

Contents

Introduction

This document will help you get started writing code and running applications on a development platform (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. For now, the developer will have to cut/paste the examples provided in the document to their system.

Goals

This document does:

1. Walk you through the system registration and Beta stack installation.
2. Introduce the build environment for Intel MIC Architecture software.
3. Give an example of how to write code for Intel® Xeon Phi™ coprocessor and build using Intel® Composer XE.
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 an Intel® Xeon Phi™ 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. The operating systems (OS) supported on the host are Red Hat* Enterprise Linux* 6.0, Red Hat* Enterprise Linux* 6.1, Red Hat* Enterprise Linux* 6.2, SUSE* Linux* Enterprise Desktop 11.1 and SUSE* Linux* Enterprise Desktop 11.2. The user will have to install the OS.

Target – The Intel® Xeon Phi™ coprocessor and corresponding runtime environment installed inside the coprocessor.

uOS – Micro Operating System – the Linux*-based operating system and tools running on the Intel® Xeon Phi™ coprocessor.

ISA – Instruction Set Architecture – 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).[1]

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

NAcc – Native Acceleration – a mode or form of Intel® MKL in which the data being processed and the MKL function processing the data reside on the Intel® Xeon Phi™ coprocessor.

[1] Intel acronyms dictionary, 8/6/2009, http://library.intel.com/Dictionary/Details.aspx?id=5600

Offload Compilers – The Intel® C/C++ Compiler XE 2013 and Intel® Fortran Compiler XE 2013 compilers, which 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 Intel® Xeon Phi™ coprocessor, or paired binaries that run on both the host and the Intel® Xeon Phi™ coprocessor and communicate with each other.

SDP – Software Development Platform – the combination of the host platform, the Intel® Xeon Phi™ coprocessor.

KNC – an abbreviation for Intel® Xeon Phi™ Coprocessor (codename: **Knights Corner**), the first Intel® Xeon Phi™ product.

MPSS – Intel **M**IC Architecture **P**latform **S**ystem **S**oftware – the user- and system-level software that allows programs to run on and communicate with the Intel® Xeon Phi™ coprocessor.

SCIF - Symmetric Communications Interface – the mechanism for inter-node communication within a single platform, where an node is a Intel® Xeon Phi™ coprocessor or an Intel® Xeon® processor-based host processor complex. In particular, SCIF 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-provided SDP consisting of 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® MIC Software

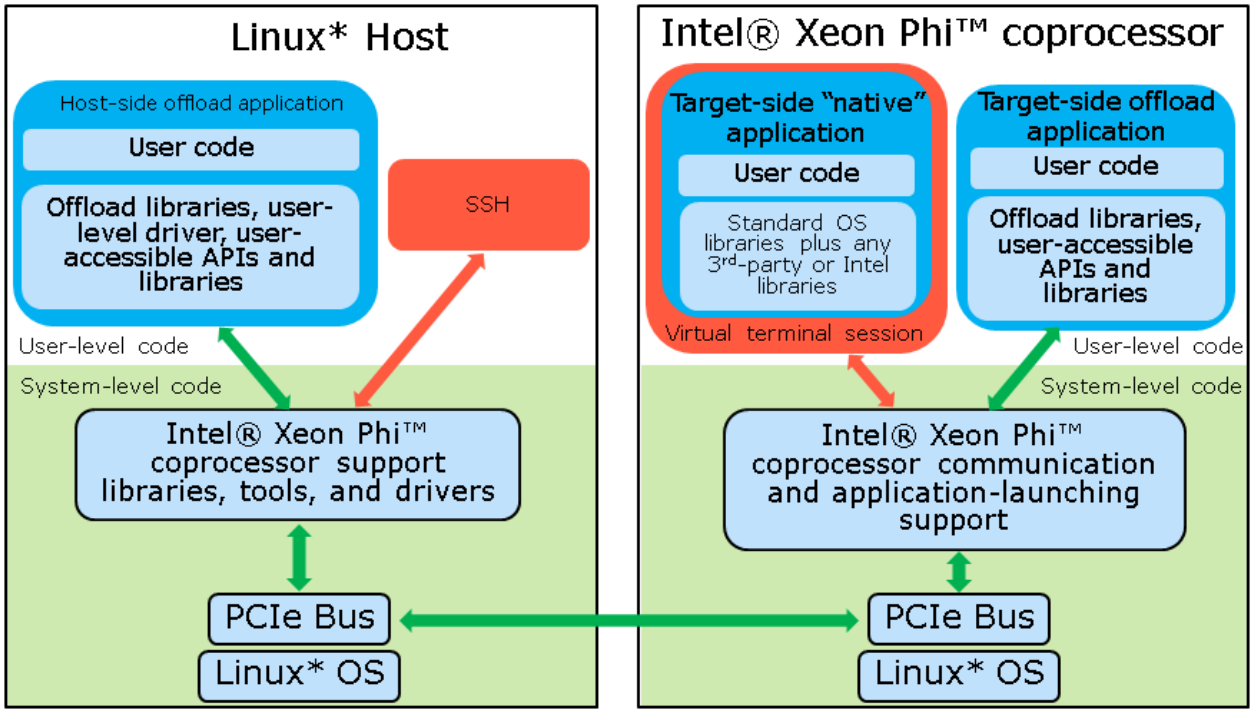

Figure 1: Software Stack

The Intel® Xeon Phi coprocessor 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 Intel® Xeon Phi™ 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 /opt/intel/mic/bin/**micinfo** for querying system information, /opt/intel/mic/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 SCIF library interface can be found at http://software.intel.com/en-us/blogs/2012/06/05/knights-corner-open-source-software-stack/. Some of the other low level interfaces (COI, MYO) are used only by Intel tools and are currently not available for general use. These low level interfaces may be deprecated or exposed in the future.

Intel® Many Integrated Core Architecture Overview

The Intel® Xeon Phi™ coprocessor has more than 50 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 4 hardware threads. In addition to the cores, there are multiple on-die memory controllers and other components.

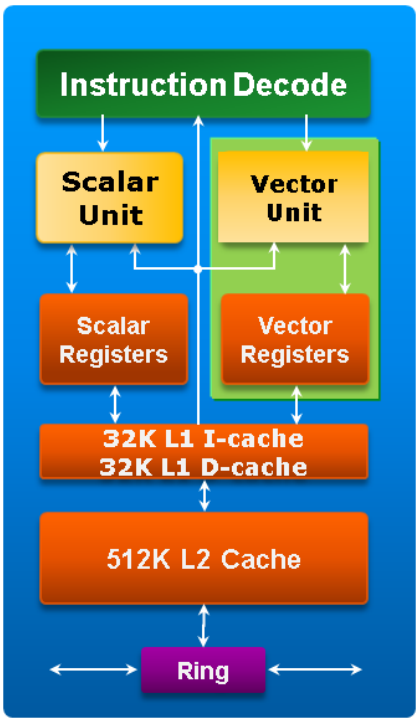

Figure 2: Architecture overview of an Intel MIC Architecture core

Each core includes a newly-designed Vector Processing Unit (VPU). Each vector unit 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 vector unit is critical for best Intel 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 machine architecture, please refer to the *Knights Corner Software Developers Guide* posted at https://mic-dev.intel.com/.

Administrative Tasks

Register your system if you received an SDP (Software Development Platform)

Before you start to use your software development platform, please go to http://support.intel.com/support/go/makc1sdp and register it using the serial number of your workstation. In you received only an Intel® Xeon Phi™ coprocessor, use the serial number of your old MAKF1SDP workstation to register. Once your system is registered, you will see new products appear under your account at https://premier.intel.com:

- **Intel(R) SDP MAKC1 Family**– use this product to file bugs and feature requests against the physical development platform hardware (installation issues, power issues, over-heating, bad disk, etc.) and if you have problems with the driver for the Intel® Xeon Phi™ coprocessor. Under this product, you will also be able to download the latest hardware drivers and release notes for the platform (the release notes include driver and firmware installation instructions).
- **Knights Ferry/Knights Corner SW Development –** use this product if you have proprietary or sensitive questions about porting or optimization that are not specific to a software component.

Register your card if you received a Intel® Xeon Phi™ Coprocessor (not a system)

If you just received the Intel® Xeon Phi™ Coprocessor only and never received an SDP before, please go to http://support.intel.com/support/go/KNCSDV and register it using the card serial number.. Once your card is registered, you will see new products appear under your account at https://premier.intel.com:

- **Intel(R) SDP HW/MPSS (Knights Corner)**– use this product to file bugs and feature requests against the physical development platform hardware (installation issues, power issues, over-heating, bad disk, etc.) and if you have problems with the driver for the Intel® Xeon Phi™ coprocessor. Under this product, you will also be able to download the latest hardware drivers and release notes for the platform (the release notes include driver and firmware installation instructions).

- **Knights Ferry/Knights Corner SW Development –** use this product if you have proprietary or sensitive questions about porting or optimization that are not specific to a software component.

Details of this registration process is explained in https://mic-dev.intel.com/article/getting-support

All the software tools are now provided by the Intel® Registration Center (IRC). If you have an existing valid license for all the Linux Intel tools, you may use that license. If you don't have a valid license, you need to get a new license from the IRC. This process is described in https://mic-dev.intel.com/page/license-migration-beta-license-2012-2013-enabling-license

If you completed the original Statement of Work (SOW), you likely also have access to https://mic-dev.intel.com website. If not, contact your Intel Account Manager to get this access set up. Use the forum on the web site for general "how-to" questions that do not involve sensitive intellectual property and to download whitepapers discussing the Intel® Xeon Phi™ coprocessor.

Refer to https://mic-dev.intel.com/article/getting-support for step-by-step information about submitting Intel® Premier Support issues under Intel® Xeon Phi™ coprocessor (codename: Knights Corner)-specific products.

Preparing Your System for First Use

Note: if you received just an Intel® Xeon Phi™ Coprocessor (codename: Knights Corner) and are installing it in a system that used to house an Intel® Xeon Phi™ Coprocessor Software Development Vehicle (codename: Knights Ferry), please review the next section, **Updating an Existing System**, before starting this section.

You should have received a big, heavy workstation containing the Intel® Xeon Phi™ coprocessor. Unpack and collect all the documentation.

Steps to install the driver and start the card

1. Find in your box the "Readme file for the MPSS Beta release", or download it from https://premier.intel.com under the "**Intel(R) SDP MAKC1 Family**" product. (Look for *readme.txt*). Also download the release notes (releaseNotes.txt)
2. Log in as root.
3. Install Red Hat* Enterprise Linux* 64-bit 6.0 kernel 2.6.32-71, Red Hat* Enterprise Linux* 64-bit 6.1 kernel 2.6.32-131, Red Hat* Enterprise Linux* 6.2 64-bit kernel 2.6.32-220, SUSE* Linux* Enterprise Desktop 11 SP1 kernel 2.6.32.12-0.7-default or SUSE* Linux* Enterprise Desktop 11 SP2 kernel 3.0.13-0.27-default (Sections 4.1-4.2 in *_readme.txt*). Be sure to install ssh, which is used to log in to the card's uOS.

   WARNING: On installing Red Hat, it may automatically update you to 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 7 of *readme.txt* for instructions on building a MPSS host driver for a specific Linux kernel.

4. Download the Beta release drivers appropriate for your operating system from https://premier.intel.com under the "**Intel(R) SDP MAKC1 Family**" product (*KNC_beta_isv-2.1.3653-8-rhel-6.0.tgz, KNC_beta_isv-2.1.3653-8-rhel-6.1.tgz, KNC_beta_isv-2.1.3653-8-rhel-6.2.tgz,* or *KNC_beta_isv-2.1.3653-8-suse-11.1.tgz or KNC_beta_isv-2.1.3653-8-suse-11.2.tgz*).
5. Install the host driver RPMs as detailed in section 4.3-4.6 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 4.7 of *readme.txt.*
7. Reboot the system.
8. 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
sudo /opt/intel/mic/bin/micinfo
```

   - Verify that the driver version is 3653-8.
   - Verify that the MPSS version is 2.1.3653-8.

- Verify that the "Flash Version" is 2.1.01.0372.

Steps to install the Software Development tools

If you have an existing valid license for all the Linux Intel tools, you may use that license. If you don't have a valid license, you need to get a new license from the Intel® Registration Center (IRC). This process is described in https://mic-dev.intel.com/page/license-migration-beta-license-2012-2013-enabling-license . To get a new license, click on "GET MY NEW SERIAL NUMBER" in that page, and then click "Continue". This brings you to the page "Intel® Cluster Studio XE for Linux*". Save the product serial number as you will need it to install the products. Select the product "Intel® C++ Composer XE for Linux*" and download the tar file l_ccompxe_intel64_2003.x.x.tgz if you want the C/C++ compiler. Select the product "Intel® Fortran XE for Linux*" and download the tar file l_fcompxe_intel64_2003.x.x.tgz if you want the Fortran compiler.

1. Download the development tools that you wish to install. You can install one of the two "complete" Composer XE 2013 packages, or download and install some or all of the "ADDON" tools available after you install the Composer XE 2013 "BASE" package, as well as separately downloading and installing Intel® VTune™ Amplifier XE 2013 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 (*release-notes-c-2013-l-en.pdf, release-notes-f-2013-l-en.pdf*) carefully.
    - Untar the product file (tar -xvf l_ccompxe_intel64_2003.x.x.tgz).

2. Install the software tools using the above serial number.
3. Verify that the card is working by running the sample program again with "`setenv H_TRACE 2`" or "`export H_TRACE=2`" to display the dialog between the Host and Intel® Xeon Phi™ coprocessor (messages from the processor will be prefixed with "MIC:"). If you do see 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 2013:
    a) Load the data collection driver after starting the coprocessor by going to /opt/intel/vtune_amplifier_xe/bin64/k1om/ and running:

```
sudo sep_micboot_create.sh
sudo sep_micboot_install.sh
```

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

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

    The coprocessor has successfully restarted when `micctrl -w` reports "mic*x*: online"

    c) The sampling driver will now start every time the coprocessor is restarted

d) If you ever need to reinstall the sampling driver, it can be done as follows:

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

Updating an Existing System

Updating a system to an Intel® Xeon Phi™ coprocessor (codename: Knights Corner)

If you received a Intel® Xeon Phi™ Coprocessor (codename: Knights Corner) to put in your "**Intel(R) SDP MAKF1 Family**" system, there are a number of things you need to do:

1. Register your system. To do this, go to http://support.intel.com/support/go/makc1sdp, and register using the serial number of your MAKF1 system (not the serial number of the card you received). This will give you access to the "**Intel(R) SDP MAKC1 Family**" at https://premier.intel.com, where you can download the drivers and tools for the Intel® Xeon Phi™ Coprocessor (codename: Knights Corner). The drivers and tools installed in your system for Knights Ferry will not work with Knights Corner.
2. Download the document *Knights_Corner_ShadyCove_Installation_BKC.pdf* and *BIOS59.zip* from the "**Intel(R) SDP MAKF1 Family**" product and follow the directions. Do not install the Intel® Xeon Phi™ coprocessor (codename: Knights Corner) until you have followed these directions. When asked to install the Knights Corner SDV drivers and firmware, refer to the instructions in sections 4.3-4.6 of *readme.txt.*
3. Once you remove the Knights Ferry coprocessor from your system, put the Intel® Xeon Phi™ Coprocessor (codename: Knights Corner) in, paying special attention to the power connectors. Some do not lock or may allow you to insert them backwards.
4. Check if the Intel® Xeon Phi™ coprocessor (codename: Knights Corner) is detected:

```
# $ /sbin/lspci | grep "Co-processor"
03:00.0 Co-processor: Intel Corporation Device 225x
```

5. If the coprocessor was not identified, check that both PCIE auxiliary power connectors are fully connected to the card. The fact that there are blue lights blinking on the card does not necessary imply that all the connectors are fully connected – check the connectors and then repeat the previous step.
6. Update the flash on your card(s) as detailed in section 4.7 of *readme.txt.*
7. Reboot the system.
9. 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
sudo /opt/intel/mic/bin/micinfo
```

- *V*erify that the driver version is3653-8.

- Verify that the MPSS version is 2.1.3653-8.
- Verify that the "Flash Version" is 2.1.01.0372 .

Once the operating system software on both the host and coprocessor have been updated, you will need to remove the Alpha Intel software development tools you installed for Knights Ferry, and install the software development tools for the Intel® Xeon Phi™ coprocessor (codename: Knights Corner). Run the `uninstall.sh` scripts you will find in the top-level or `bin` directory of each product, and then install the software development tools as noted in the previous section.

Updating a system that already has an Intel® Xeon Phi™ Coprocessor (codename: Knights Corner)

1. Update the system software using the instructions in sections 4.3-4.6 of *readme.txt*
2. Continue with step 6 in the previous section

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

The Intel® Xeon Phi™ coprocessor will not start when the host system reboots. So you will need to manually start the Intel® Xeon Phi™ coprocessor, 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
sudo /opt/intel/mic/bin/micinfo
```

Note: It is possible to make the coprocessor automatically start on reboot and preload desired files. See sections 4.3-4.6 of *readme.txt* for details

Restarting the Intel® Xeon Phi™ coprocessor If It Hangs

If a process running on the Intel® Xeon Phi™ coprocessor hangs, but the coprocessor is otherwise responsive via `ssh`, log into the card 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 MPSS service is still functioning properly, you can try to restart the coprocessor without affecting any attached coprocessors as follows:

```
sudo micctrl --reset <micx>
sudo micctrl --boot <micx>
sudo micctrl -w
sudo /opt/intel/mic/bin/micinfo
```

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

```
sudo service mpss stop
sudo service mpss unload
sudo service mpss start
sudo micctrl -w
sudo /opt/intel/mic/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 5.3 of *readme.txt* for details:

1. Execute the monitor
   `/opt/intel/mic/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 Intel® Xeon Phi™ coprocessor and execute it using the "`micnativeloadex`" utility. See section 5.5 of *readme.txt* 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 at `172.31.1.254`. The second is called "`mic1`" and is located at `172.31.2.1`, seeing the host at `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 Intel® Xeon Phi™ coprocessor, etc., please see section 6 *readme.txt*.

Useful Administrative Tools

This product ships with the following administrative tools, found in the "`/opt/intel/mic/bin`" directory. Root, and users needing to use these tools, should add this directory 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 - a tool designed to ease the burden of monitoring and managing Intel® Xeon Phi™ coprocessors.
- miccheck – a utility for verifying the configuration of an Intel® Xeon Phi™ coprocessor by running various diagnostic tests.
- micnativeloadex – a utility that will copy an Intel MIC Architecture native binary to a specified Intel® Xeon Phi™ coprocessor r and execute it
- micctrl – a tool to help the system administrator configure and restart the coprocessor

Please see section 6 in *readme.txt* for details on these tools and their arguments.

Getting Started/Developing Intel® MIC 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 Intel® Xeon Phi™ 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 SW development Tools / Environments

You can start programming for 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 Intel® Xeon Phi™ 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 best performance from the Intel® Xeon Phi™ coprocessor, it is important to understand the Intel MIC Architecture.

Development Environment: Available Compilers and Libraries

- **Compilers**
  - Intel® C++ Composer XE 2013 for building applications that run on Intel 64 and Intel MIC architectures

  - Intel® Fortran Composer XE 2013 for building applications that run on Intel 64 and Intel MIC architectures
- **Libraries** packaged with the compilers include:
  - Intel® Math Kernel Library (Intel® MKL) optimized for the Intel MIC Architecture
  - Intel® Threading Building Blocks (Intel® TBB)
- **Libraries** packaged separately include:
  - Intel® MPI for Linux* OS including Intel® Many Integrated Core (Intel® MIC) Architecture

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 Intel® Xeon Phi™ coprocessor.

- **Debugger**
  - Intel® Debugger for applications running on the Intel 64 and Intel MIC architectures
  - Intel® C++ Eclipse* Product Extension including Debugging
- **Profiling**
  - Intel® VTune™ Amplifier XE 2013 for Linux*, which is used on the host Linux* OS to collect and view performance data collected on the Intel® Xeon Phi™ coprocessor.

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 2013**: `/opt/intel/composerxe/bin/compilervars.csh` or `compilervars.sh` script with `intel64` as the argument.

  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 by themselves. (Among other things, there are ordering issues that might result in unpredictable behavior.)
  - **Intel debugger:** `/opt/intel/composerxe/pkg_bin/idbvars.csh` or `idbvars.sh` script with `intel64` as the argument.
  - **Intel TBB:** `/opt/intel/composerxe/tbb/bin/tbbvars.csh` or `tbbvars.sh` with `intel64` as the argument.
  - **Intel MKL:** `/opt/intel/composerxe/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/composerxe/Documentation/en_US/` including:
  - **`compiler_c/main_cls/index.htm`** and **`compiler_f/main_cls/index.htm`** - complete documentation for Intel® C++ Compiler XE 13.0 and the Intel® Fortran Compiler XE 2013.

    - Most information on how to build for the Intel MIC Architecture can be found in the "Key Features" 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-*-2013-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. You'll find Intel MIC Architecture-specific information primarily in sections 3.
    - **Note**: For various reasons, this document can miss some last-minute updates. The *release-notes-*-2013-l-en.pdf* documents from the Fortran compiler can be downloaded from the IRC always the most recent version of this document (see Section "Steps to install the Software Development tools").
  - **`debugger/debugger_documentation.htm`** – Information on how to use the Intel Debugger. You will find information specific to debugging Intel MIC Architecture applications under the "Debugging with the Intel® Debugger on Eclipse*" and "Debugging on the Command Line" sections
- Other documentation that includes sections on using the Intel® Xeon Phi™ coprocessor:
  - The Intel MKL User's Guide, which can be accessed via **`mkl_documentation.htm`** found in `/opt/intel/composerxe/Documentation/en_US/mkl`, 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" of Intel MKL functions.
  - Information on collecting performance data on the Intel® Xeon Phi™ coprocessor using VTune Amplifier XE 2013 for Linux* can be found in **`mic-data-collection.pdf`**, located in `/opt/intel/vtune_amplifier_xe_2013/documentation/en`.
- Useful documentation on the Web:
  - On the NDA-only website https://mic-dev.intel.com you will find a wide range of documentation that can be downloaded, most notably the ***Knights Corner Software Developers Guide*** and updated versions of this document. Much of the Knights Ferry information found on this website also applies to Knights Corner. Here you will also find user forums, the Tuning and Optimization Wiki, and videos/slides from our developer workshops.
  - Navigating down from http://software.intel.com/en-us/blogs/2012/06/05/knights-corner-open-source-software-stack/, you will find the source code for the Intel® Xeon Phi™ coprocessor's uOS, a native Intel MIC Architecture version of gdb, and documentation including the ***Knights Corner Instruction Set Reference Manual***, ***ABI document System V Application Binary Interface K1OM Architecture Processor Supplement***, and ***Knights Corner Performance Monitor Units*** under the "Resources" link.
- Some sample offload code using the explicit memory copy model can be found in:
  - **Intel C++**: `/opt/intel/composerxe/Samples/en_US/C++/mic_samples/intro_sampleC/`
  - **Intel Fortran**: `/opt/intel/composerxe/Samples/en_US/Fortran/mic_samples/`
  - **Intel MKL**: `/opt/intel/composerxe/mkl/examples/mic_samples`
    - For examples of Intel MKL automated offload: `/opt/intel/composerxe/mkl/examples/mic_samples/ao_sgemm` and `…/ao_sgemm_f`

    - The rest of the samples demonstrate use of MKL via compiler-assisted offload.
- Some sample offload code using the implicit memory copy model can be found in:
    - C: `/opt/intel/composerxe/Samples/en_US/C++/mic_samples/celo_sampleC` and `…/LEO_tutorial`
    - C++: `/opt/intel/composerxe/Samples/en_US/C++/mic_samples/celo_sampleCPP`

Build-Related Information

- The offload compiler produces "fat" binaries and `.so` files that contain code for both host and the Intel® Xeon Phi™ coprocessor.
- The offload compiler produces code that examines the runtime execution environment for the presence of an Intel® Xeon Phi™ coprocessor. If one is not present, it falls back to executing the "offloaded" code on the host. In other words, 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 *release-notes-*-2013-l-en.pdf*.

Compiler Switches and Makefiles

When building applications that offload some of their code to the Intel® Xeon Phi™ 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 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 potions 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.

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 Intel® Xeon Phi™ 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 Math Kernel Library (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 Intel® Xeon Phi™ coprocessor. At this time, 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, the user may specify the region of code to run on the target, there is no guarantee of execution on the Intel® Xeon Phi™ coprocessor. Depending on the presence of the target hardware or the availability of resources on the Intel® Xeon Phi™ coprocessor when execution reaches the region of code marked for offload, the code can run on the Intel® Xeon Phi™ coprocessor or may fall back to executing on the host.

The following code samples show several versions of porting reduction code to the Intel® Xeon Phi™ coprocessor using the offload pragma directive.

Reduction

The operation refers to computing the expression:

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

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++

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

Creating the Offload Version

Serial Reduction with Offload

The programmer uses **#pragma offload target(mic)** (as shown in the example below) to mark statements (offload constructs) that should execute on the Intel® Xeon Phi™ 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, the programmer 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 Intel® Xeon Phi™ coprocessor is available. For details see the "About Asynchronous Computation" and "About Asynchronous Data Transfer" sections in the Intel® C++ Compiler XE 2013 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/composerxe /Samples/en_US/C++/mic_samples/intro_sampleC/ sampleC13.c`

Using the Offload Compiler – Implicit Memory Copy Model

Intel® C++ Composer XE 2013 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);
```

It should be noted that this is not actually "shared memory": there is no hardware that maps some portion of the memory on the Intel® Xeon Phi™ coprocessor 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 at 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__
    int nThreads = 32;
    omp_set_num_threads(nThreads);
```

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

    #else
    printf("MIC 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/composerxe/Samples/en_US/C++/mic_samples/shrd_sampleC` and `…/LEO_tutorial`

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

Native Compilation

Applications can also be run natively on the Intel® Xeon Phi™ 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 Intel® Xeon Phi™ coprocessor's filesystem (or make them visible over there via NFS).

Example:

1. Copy `openmp_sample.c` from `/opt/intel/composerxe/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* runtime library:

```
scp /opt/intel/composerxe/compiler/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* runtime library):

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

6. 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 the host systems are available for the Intel® Xeon Phi™ coprocessor. These include the following:

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

The following sections will discuss the use of these parallel programming models in code using the offload extensions. Code that runs natively on the Intel® Xeon Phi™ 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 Intel® Xeon Phi™ 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 Intel® Xeon Phi™ coprocessor. Since the entire OpenMP* construct is executed on the Intel® Xeon Phi™ coprocessor, within the construct the usual OpenMP* semantics of shared and private data apply.

Multiple host CPU threads can offload to the Intel® Xeon Phi™ coprocessor at any time. If a CPU thread attempts to offload to the Intel® Xeon Phi™ coprocessor and resources are not available on the coprocessor, 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

Intel® Xeon Phi™ 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 Intel® Xeon Phi™ 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()** 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 Open MP 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 (float 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 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 Intel® Xeon Phi™ 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 method join 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 Intel® Xeon Phi™ coprocessor by __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++

Using Intel® MKL

For offload users, Intel MKL is most commonly used in Native Acceleration (NAcc) mode on the Intel® Xeon Phi™ coprocessor. In NAcc, all data and binaries reside on the Intel® Xeon Phi™ coprocessor. Data is transferred by the programmer 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 Native Acceleration Mode can also be used in native Intel MIC Architecture code – in this case the Intel MKL shared libraries must be copied to the Intel® Xeon Phi™ coprocessor before execution.

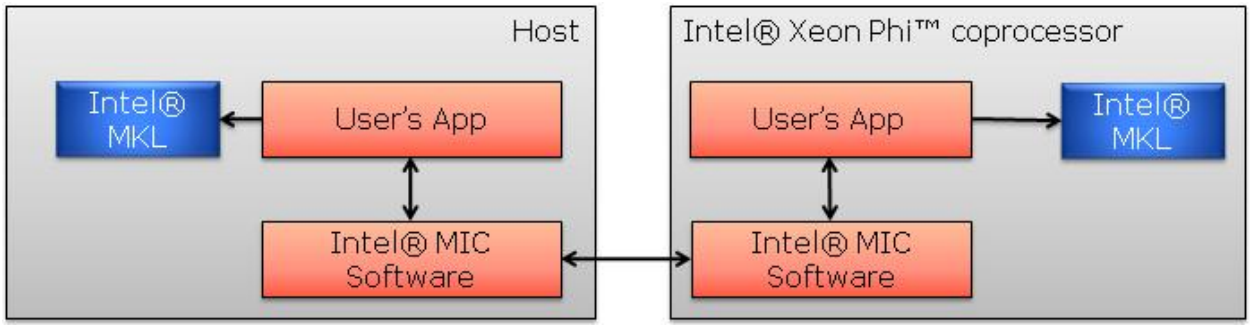

Figure 3.1: Using MKL Native Acceleration with Offload

SGEMM Sample

Using SGEMM routine from 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 Intel® Xeon Phi™ coprocessor using #`pragma offload`. In this example, the `free_if(0)` qualifier is used to make the data persistent on the Intel® Xeon Phi™ coprocessor.

```
#pragma offload target(mic) \
    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 Intel® Xeon Phi™ coprocessor. The `nocopy()` qualifier causes the data copied to the card in step 2 to be reused.

```
#pragma offload target(mic)
    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) \
        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 the MKL function within the offloaded code.

```
#pragma offload target(mic) \
    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 Intel® Xeon Phi™ 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 Intel® Xeon Phi™ 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 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/composerxe/mkl/examples/mic_samples/ao_sgemm/`.

Debugging on the Intel® Xeon Phi™ coprocessor

You will find information specific to debugging Intel MIC Architecture applications under the "Debugging with the Intel® Debugger on Eclipse*" and "Debugging on the Command Line" sections of `/opt/intel/composerxe/Documentation/en_US/debugger/debugger_documentation.htm`

Performance Analysis on the Intel® Xeon Phi™ coprocessor

Information on collecting performance data on the Intel® Xeon Phi™ coprocessor using VTune Amplifier XE 2013 for Linux* can be found in `mic-data-collection.pdf`, located in `/opt/intel/vtune_amplifier_xe_2013/documentation/en`.

Information on performance tuning can be found at the https://mic-dev.intel.com/ website in case studies, white papers, and our Tuning and Optimization Wiki.

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 Masters degree in Computer Engineering from Clemson University in May 2010. She joined Intel in 2010 and been working as an enabling Application Engineer, focusing on optimizing applications for ISV's and developing collateral for Intel® Many Integrated Core Architecture.

Charles Congdon is a senior software engineer with Intel® Corporation'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.

Notices

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