Debugging on the Intel® Xeon Phi™ Coprocessor

Dr.-Ing. Michael Klemm
Software and Services Group
Intel Corporation
(michael.Klemm@intel.com)
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY
ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS
DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS
OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR
WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR
INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components
and reflect the approximate performance of Intel products as measured by those tests. Any
difference in system hardware or software design or configuration may affect actual performance.
Buyers should consult other sources of information to evaluate the performance of systems or
components they are considering purchasing. For more information on performance tests and on

Copyright © 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, Phi,
VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries. *Other names
and brands may be claimed as the property of others.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 SSSE3 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.</td>
</tr>
</tbody>
</table>

Notice revision #20110804
Agenda

→ Overview

Installation

Command Line Debugger
• Debugging a Coprocessor Native Application
• Debugging Offloaded Code
• The GNU* Project Debugger (GDB*) & Intel® Debugger (IDB)

Eclipse* CDT Integration

GDB* Enabling
Motivation

Intel® Xeon Phi™ Coprocessor relies on new programming models and debug communication models.

Intel® Debugger provides cross-debug solution to debug on Intel® Xeon Phi™ Coprocessor based coprocessor cards.
Intel® Debugger for Intel® Xeon Phi™ Coprocessor

Command Line Debugger
Eclipse* IDE Integration
Linux* hosted
C/C++ & Fortran
Offload and Native Code Debug
Debugger for Intel® Xeon Phi™ Coprocessor
Basic Setup

Debugger

Offload

Debug agent
Debugger Installation

Intel® Debugger for Intel® Xeon Phi™ Coprocessor is part of the Intel® Composer XE for Linux* Including Intel® Xeon Phi™ Coprocessor.

Gets installed automatically running the install.sh install script from the l_ccompxe_2013.0.xxx.tgz package.

Setting up the Intel® C++ Compiler environment via

$ source /opt/intel/composer_xe_2013/bin/compilervars.sh intel64

will also set up the environment for the Intel® Debugger.
Agenda

Overview

Installation

Command Line Debugger

• Debugging a Coprocessor Native Application
• Debugging Offloaded Code
• The GNU* Project Debugger (GDB*) & Intel® Debugger (IDB)

Eclipse* CDT Integration

GDB* Enabling
Command Line Debugging

Debugger Executables

After installation is complete you will find the debugger executables at

/opt/intel/composer_xe_2013/bin/intel64_mic

idbc is the command line debugger driver for the host.

idbc_mic is the command line debugger driver for the Intel® Xeon Phi™ Coprocessor based card.

idbc and idbc_mic are the debugger executables
Launching the Debugger

Launch the target debugger by using the following command:

```
idbc_mic -tco -rconnect=tcpip:coprocessor-ip-address:port
```

or

```
idbc_mic -tco -rconnect=tcpip:mic[n]:port
```

when using the default IP addresses.

The default port number is 2000.

For example, enter

```
idbc_mic -tco -rconnect=tcpip:mic0:2000
```

for the first Intel® Xeon Phi™ Coprocessor device using the default port.
Starting Application from within Debugger

If you are debugging an application to run natively on Intel® Xeon Phi™ Coprocessor you can start the application using the debugger:

1. Specify the remote executable file:
   (idb) idb file-remote target-bin-path

2. Specify the file containing debug info on host:
   (idb) file host-bin-path

3. After setting breakpoints and whatever else you want to do before starting the application, start the Application to reach breakpoint:
   (idb) run

Note:
The application has to be uploaded to the coprocessor outside of the debugger prior to launching it.
Attaching to Application with Debugger

Attach

If you are debugging an application on the coprocessor target, run the application and attach the debugger to it:

Enter the following command:

(idb) attach <pid> <image_file>

<pid> The pid of the process to attach to.
<image_file> The image path and file name on the host

Target Process List

To identify the process to attach to the following command can be used

(idb) idb show process-list ["proc-name"]

Semantics:

(idb) show process-list prints all processes running on the target and variable $lsproc will hold the number of processes found as a negative number, i.e. If there are 5 processes running, $lsproc will be -5

(idb) show process-list "proc-name" will get the process list and search for a process named proc-name. If found, $lsproc will contain the process id or 0 if no applicable process has been found.
Native Debug Considerations (1)

1. New command:

Setting remote working directory on the coprocessor:

(idb) idb remote-working-dir

2. MPSS supports standard Linux user account configurations

idbserver_mic and the native application to be debugged need to be launched as the same user.

This means both the native app on the target and idc_mic need to either be launched as root or with a user account of the same name.
Native Debug Considerations (2)

When debugging native coprocessor applications on the command line, the shared library libmyodbl-service.so, needs to be uploaded manually moving forward.

Solution:

Create an overlay, so the file is uploaded at boot time. Follow the instructions on how to use overlays in the MPSS readme.txt.

Specific Steps:

a. Target: Create /etc/sysconfig/mic/conf.d/myo.conf containing

```
# MYO download files
Overlay / /opt/intel/mic/myo/config/myo.filelist
```

b. Host: Create /opt/intel/mic/myo/config/myo.filelist containing

```
dir /lib64 755 0 0
file /lib64/libmyodbl-service.so
opt/intel/mic/myo/lib/libmyodbl-service.so 755 0 0
```
Using an Endless Worker Loop (1)

It may be useful to use an endless worker loop to be able to attach to an application at a defined location.

To ensure the endless worker loop in your native or coprocessor-side application is executed even with aggressive compiler optimizations enabled it is recommended to implement it as follows:

```c
void attach_idb() {
    volatile int loop = 1;
    do {
        volatile int a = 1;
    } while (loop);
}
```

Call this function at a location suitable for attaching the debugger.
Using an Endless Worker Loop (2)

If you added an endless worker loop to your offload code, the following method may be used to start debugging just outside of the loop.

1. Using the following commands:

```
(idb) list <filename>
```

2. Now you have a line number for `volatile int a =1`. Set a breakpoint on that line and continue.

```
(idb) p loop=0
(idb) break <line number>
```

3. Continue

```
(idb) continue
```

You can identify the source line directly after the endless worker loop and start debugging from that source line.

**Note:** After that, set a breakpoint using the break command at any code line or address of interest and issue a continue command to run to it.
Shared Object Debugging

**Using LD_LIBRARY_PATH**

If the target-side application is dynamically linked against shared objects that are not part of the compiler or driver, and that need `LD_LIBRARY_PATH` to be set in order to find them,

1. Set a breakpoint before the libraries are loaded.
2. When the debugger stops at that location, use the command
   (idb) set environment LD_LIBRARY_PATH path
3. Do the same for other environment variables that the application might need.

**Specifying Additional Symbol Info Search Paths**

To tell the debugger where to search for the debug information specific to your application:

(idb) set solib-search-path path[:path]

* e.g.
(idb) set solib-search-path /usr/linux-k1om-4.7/linux-k1om/lib64/lib:/usr/linux-k1om-4.7/x86_64-k1om-linux/lib64

To ensure correct target Linux* runtime library pick-up by the debugger

(idb) show solib-search-path

Provides listing of all directories in search path
Agenda

Overview

Installation

Command Line Debugger

• Debugging a Coprocessor Native Application

• Debugging Offloaded Code

• The GNU* Project Debugger (GDB*) & Intel® Debugger (IDB)

Eclipse* CDT Integration

GDB* Enabling
Simultaneous Debug Host and Coprocessor (1)

No Debug Synchronization between Host and Coprocessor

For command line debug there is no active synchronization between host and coprocessor debugged code.

1. Set your host side breakpoint where a target workload already exists.

2. After attach with idbc_mic, set a breakpoint there where you want to start debugging.

If you are debugging a heterogeneous application and intend to debug host and coprocessor code simultaneously,

run idbc for the host

run idbc_mic for the coprocessor targeted codebase.

→ Two terminal windows or for a remote debug setup two ssh sessions will be necessary.
Simultaneous Debug Host and Coprocessor (2)

**Host Debug:**

idbc <application>

Start the app on the host side through the debugger and stopped at a breakpoint somewhere after creating target offload process.

**Target Debug:**

idbc_mic -tco -rconnect=tcpip:<cardip>;<port>

(idb) attach <pid> /opt/intel/composerxe/lib/mic/offload_main

The actual location of the `offload_main` binary may differ depending on the tools version used.
Target Debug:

IDB will attach to the process and read debug info from debuggee process and loaded libraries. If the libraries are located at a different location than at compile time, you can set up library search paths using the debugger command.

(idb) set solib-search-path <path-to-so>[:<path-to-so>]

Note:

The location the debugger stops on the target is random, typically in the scheduler or libpthread:
• set host side breakpoint where target workload already exists
• after attach with idbc_mic set a breakpoint there where you want to start debugging.

• You may want to consider introducing infinite worker loop in offload code do define connection point.
Agenda

Overview

Installation

Command Line Debugger

• Debugging a Coprocessor Native Application
• Debugging Offloaded Code

• The GNU* Project Debugger (GDB*) & Intel® Debugger (IDB)

Eclipse* CDT Integration

GDB* Enabling
Intel® Debugger (IDB) and The GNU* Project Debugger (GDB*)

IDB is the Intel debugger. It has a GDB-style command line interface.

Basic commands and behavior are the same as GDB.

In addition it features,

- Enhanced Fortran 90/95 support
- Support for dynamic arrays in Fortran
- Integration into Eclipse* CDT offers enhanced threading support
- Enhanced Parallelism and Threading Support (next slide)

Debugger Online Help:
http://software.intel.com/sites/products/documentation/hpc/comp
Intel® Debugger unique commands

- idb directory
- idb freeze
- idb info barrier
- idb info lock
- idb info openmp thread tree
- idb info task
- idb info taskwait
- idb info team
- idb info thread
- idb process
- idb reentrancy
- idb session restore
- idb session save
- idb set cilk-serialization
- idb set openmp-serialization
- idb set solib-path-substitute
- idb sharing
- idb sharing event expend
- idb sharing event list
- idb sharing filter add file
- idb sharing filter add function
- idb sharing filter add range
- idb sharing filter add variable
- idb sharing filter delete
- idb sharing filter disable
- idb sharing filter enable
- idb sharing filter list
- idb sharing filter toggle
- idb sharing reset
- idb sharing status
- idb sharing stop
- idb show cilk-serialization
- idb show openmp-serialization
- idb show solib-path-substitute
- idb stopping threads
- idb synchronize
- idb target threads
- idb thaw
- idb uninterrupt
- idb unset solib-path-substitute

Start with “idb”

Cover thread specific run-control
• Define thread groups, freeze, thaw

Intel® Cilk™ Plus and OpenMP execution serialization

Data sharing event detection

Thread filtering

OpenMP* thread info:
• Locks, barriers, teams, tasks, thread tree

idb show process-list “<image-name>”
• Displays process ID of image name.

IDB provides advanced thread run-control and awareness
Agenda

Overview

Installation

Command Line Debugger
- Debugging a Coprocessor Native Application
- Debugging Offloaded Code
- The GNU* Project Debugger (GDB*) & Intel® Debugger (IDB)

Eclipse® CDT Integration

GDB® Enabling
Debugging Heterogeneous Applications in Eclipse* IDE integration

- Offload process gets launched on Intel® Xeon Phi™ Coprocessor. This process executes offloaded code segment.
- Debugger remotely attaches to offload process offload_main
  - Automatic debug agent download on offload
  - Protocol: Virtual TCP/IP
  - Automatic attach to offload process
- Once launching host process is finished, offload process is automatically removed on coprocessor.
- Debugger captures process removal and detaches.

```c
main() {
    #pragma offload target (mic)
    #pragma omp parallel for reduction(:pi)
    for (i=0; i<count; i++)
    {
        float t = (float)((i+0.5)/count);
        pi += 4.0/(1.0+t*t);
    }
    pi /= count
}
```
Debugging Intel® Xeon Phi™ Coprocessor applications with Eclipse* CDT

- Eclipse* IDE integrated debugger with integrated thread view and source view for coprocessor code execution.
- Multi-card offload debug. Single card offload and direct/native mode support.
Single Eclipse* GUI based Debug Solution

Standard Eclipse IDE Debugger with integrated Cross-Debug for heterogeneous applications

Install Integration Add-on for Eclipse* >
Adding the Compiler and Debugger to Eclipse*

1. Start Eclipse.

2. Select or create a workspace. For example, select or create a makefile project with already existing code.

3. Select Help > Install New Software.

4. Next to the Work with field, Click the Add button. The Add Site dialog opens.

5. Click the Local button and browse to the appropriate Intel CDT integration directory:

   \( \text{install\_dir/eclipse\_support/cdt8.0/eclipse} \)

6. Click OK.

7. Make sure Group items by category is not checked.

8. Select the options beginning with Intel, including the Intel Debugger (IDB), should you choose to use it, and click Next.
Start Offload Debug Session in Eclipse* IDE

Launch Eclipse*:

```bash
$ ./eclipse &
```

![Image of Eclipse IDE interface](image.png)

Copyright © 2013, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.
Define Debug Configuration – Eclipse* Style

1. Select **Run > Debug Configurations....**
2. Select the debug configuration type, for example, **C/C++ Application.**
3. Click the **New button.**
4. Enter a name for your configuration.
Define Debug Configuration – Eclipse* Style

1. Switch to the **Debugger tab**.

2. Under **Debugger Options** select the **Main tab**. Make sure that the path to the debugger executable is specified correctly in the field **IDB Debugger**.

3. By default, **IDB debugger contains the correct start script** `idb_mpm`.
Select Offload Process Launcher for Debug Session

1. On the **Main tab** locate the process launcher information and click Select other....
2. The **Select Preferred Launcher** dialog box appears.
3. Check the **Use configuration specific settings** checkbox.
4. Select **IDB-MIC (DSF) Create Process Launcher**.
Select Application to Debug

At the **Main** tab of the **Debug Configurations** dialog, enter the path to the application you wish to debug in the field **C/C++ Application**.
Select Application to Debug

At the **Main** tab of the **Debug Configurations** dialog, enter the path to the application you wish to debug in the field **C/C++ Application**.
Add Search Directories for Shared Objects
Debug Offload Code from within Eclipse* Debug Perspective

Select Run > Debug from the menu bar or click the Debug button.
View Offload Threads
Debugging Only on the Coprocessor

1. Select **Run > Debug Configurations**....
2. Select the debug configuration type **C/C++ Attach to Application**.
3. Click the **New** button.
4. Enter a name for your configuration.
5. On the **Main** tab locate the process launcher information and click **Select other**....
The **Select Preferred Launcher** dialog box appears.
6. Check the **Use configuration specific settings** checkbox.
7. Select **IDB-MIC (DSF) Attach to Process Launcher**.
8. Click **OK**.
The **Select Preferred Launcher** dialog box is closed.
9. Switch to the **Debugger** tab.
10. Under **Options** check the **Attach to Intel® Xeon Phi™** checkbox.
11. Choose the desired coprocessor from the **Card** pull down menu.
12. At **File Location** browse for the symbol file related to the process you want to attach to.
13. Click **Debug**.
The debugger starts and the **Select a Process** dialog box opens displaying a list of running processes.
14. Select the process you want to attach to and click **OK**.
Agenda

Overview

Installation

Command Line Debugger
- Debugging a Coprocessor Native Application
- Debugging Offloaded Code
- The GNU* Project Debugger (GDB*) & Intel® Debugger (IDB)

Eclipse* CDT Integration

➔ GDB* Enabling
The GNU* Project Debugger and Intel® Xeon Phi™ Coprocessor

GDB* native-only debugger released.

The GNU* Project Debugger and Intel® Xeon Phi™ Coprocessor

Modified and rebuilt GDB* 7.4

• Unpack

    tar xzf gdb-intel-mic-2.1.xxxx.tgz

• Copy to target

    export CARD=172.xxx.x.x

    scp gdb root@$CARD:/usr/bin

• Use GDB* locally in target processor terminal to attach and launch native process as you would on standard Linux*

• Sources to rebuild and modify GDB* and patches are provided.
More questions?