Snapdragon™ OpenCL Debugger

User Guide

80-ND791-3 A

August 11, 2014

Submit technical questions at:
https://developer.qualcomm.com/

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Tables

Revision history

Revision	Date	Description
A	Sep 2013	Initial release

1 Introduction

1.1 Purpose

This document provides a guide for installing and using the OpenCL Debugger for the Snapdragon™ processor with integrated Adreno™ 4X series GPU and an integrated Qualcomm Technologies, Inc. (QTI) Enhanced ARM® CPU.

1.2 Scope

This document is intended for those developing OpenCL applications for a Snapdragon device.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

Code variables appear in angle brackets, e.g., `<number>`.

Commands to be entered appear in a different font, e.g., **`copy a:*.* b:`**.

Button and key names appear in bold font, e.g., click **Save** or press **Enter**.

1.4 References

Reference documents are listed in Table 1-1. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm Technologies		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Resources		
R1	*Java SE Downloads*	http://www.oracle.com/technetwork/java/javase/downloads/index.html
R2	*Eclipse IDE for Java Developers*	http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/junosr1
R3	*Khronos OpenCL API Registry*	http://www.khronos.org/registry/cl/

1.5 Technical assistance

For assistance or clarification on information in this document, submit a case to QTI at https://support.cdmatech.com/.

If you do not have access to the CDMATech Support Service website, register for access or send email to support.cdmatech@qti.qualcomm.com.

1.6 Acronyms

For definitions of terms and abbreviations, see [Q1].

2 Tools and Resources

This chapter describes developer tools provided with the OpenCL Debugger from QTI, a fully functional debugger for source-level debugging of OpenCL kernel code. The targets supported are Snapdragon CPU and Adreno 4X series GPUs. The capabilities of each device can impact how the user interacts with the debugger. These differences are explained in this document.

The debugger can also debug ARM-compiled host code for facilitating debugging of OpenCL kernels. Host code refers to the ARM-compiled code that invokes the OpenCL API to manage and run kernels.

The debugger is based on LLDB and supports most of the standard LLDB commands, in addition to new ones to observe and evaluate OpenCL execution.

2.1 Debugserver

Debugserver is an on-target native Android™ application that must be installed and run in conjunction with the OpenCL application being debugged. This application provides a TCP interface for all other debugger tools. It must be launched for every debugging session. See Section 5.6.2 for instructions on starting debugserver.

2.2 LLDB for command line debugging

Command line debugging from the PC is made possible with the LLDB application. This software is supported on both Windows and Linux environments.

See Section 5.6.3 for more information.

2.3 LLDBMI

This application is similar to LLDB but is used with the Eclipse plug-in to communicate with the GDB backend.

2.4 Eclipse plug-in for graphical OpenCL debugging

The OpenCL Debugger Eclipse plug-in is an extension of the CDT Eclipse plug-in. It provides GDB debugging for C and C++ applications on the Snapdragon CPU, as well as LLDB OpenCL debugging on the Snapdragon CPU and integrated Adreno GPU devices. See Chapter 6 for more information.

3 Installation

3.1 Minimum System Requirements

The software prerequisites for installing the OpenCL Debugger tools are as follows:

- Microsoft Windows 7 or Ubuntu 12.04

For certain features of the software, Java version 1.6 or greater is required.

3.2 LLDB and LLDBMI

The LLDB and LLDBMI applications are installed in the path specified by the installation application. The default location for these applications is C:\AdrenoSDK\Bin\Debugger\Windows\. For Linux systems, it is ~/AdrenoSDK/Bin/Debugger/Linux/.

NOTE: The remainder of this document assumes these default paths are used.

NOTE: For non-default paths, you may need to adjust the path indicating where the AdrenoSDK directory is located.

3.3 OpenCL Debugger Eclipse plug-in

This section describes installation and configuration of the Eclipse plug-in. For usage, see Chapter 6.

3.3.1 Installing Eclipse

Eclipse requires a Java Runtime Environment (JRE). Download and install the appropriate JRE from [R1].

http://www.oracle.com/technetwork/java/javase/downloads/index.html

NOTE: For Ubuntu Linux, installing Java may be as simple as typing sudo apt-get install default-jre

Find the Eclipse IDE for Java Developers – Juno SR1 package ([R2]) that corresponds to your operating system at http://www.eclipse.org/downloads/packages/release/juno/sr1. For example:

- For 64-bit Windows – http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/juno/SR1/eclipse-java-juno-SR1-win32-x86_64.zip
- For 64-bit Linux – http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/juno/SR1/eclipse-java-juno-SR1-linux-gtk-x86_64.tar.gz

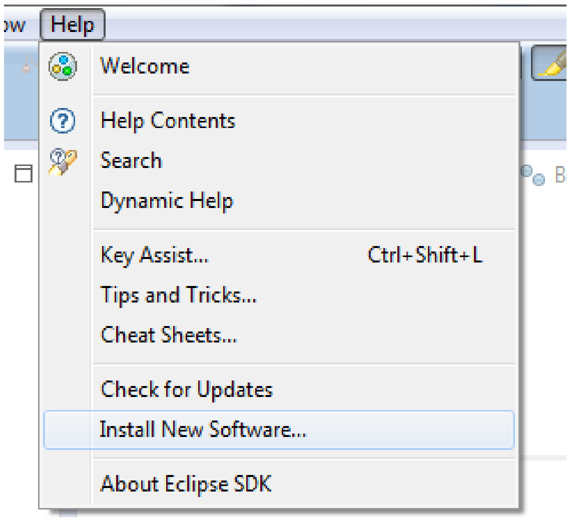
Follow typical installation instructions.

WARNING: Other versions of Eclipse have not been verified to work correctly and may not be supported as of this release.

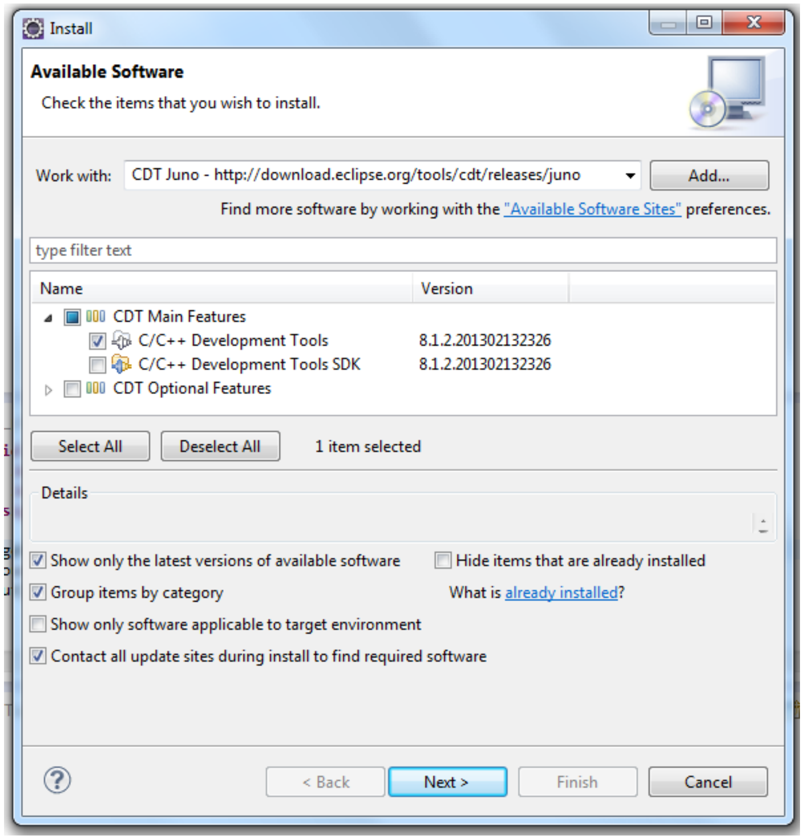
3.3.2 Installing Eclipse plug-in

You can install additional plug-ins for the Eclipse IDE to support OpenCL application debugging:

1. Start the Eclipse IDE.
2. Go to the Help menu and select **Install New Software**.



3. In the Work with text field, enter **CDT Juno - http://download.eclipse.org/tools/cdt/releases/juno** and press **Enter**.
4. Wait for the Available Software window to refresh.


Install

Available Software

Check the items that you wish to install.

Work with: CDT Juno - http://download.eclipse.org/tools/cdt/releases/juno

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
▣ CDT Main Features	
✓ C/C++ Development Tools	8.1.2.201302132326
☐ C/C++ Development Tools SDK	8.1.2.201302132326
☐ CDT Optional Features	

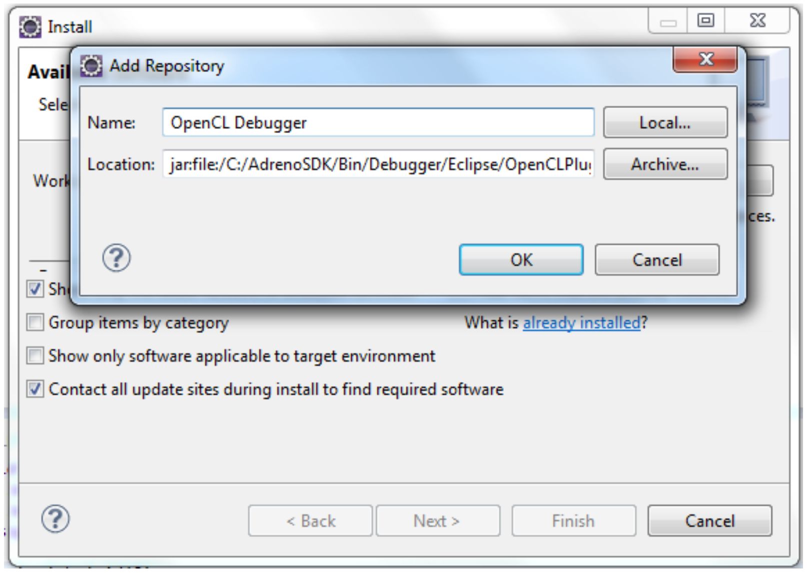
1 item selected

Details

✓ Show only the latest versions of available software
☐ Hide items that are already installed
✓ Group items by category
What is already installed?
☐ Show only software applicable to target environment
✓ Contact all update sites during install to find required software

5. Expand CDT Main Features**.**
6. Select C/C++ Development Tools and click **Next**.
7. In the ensuing window, Click **Next** to install the selected plug-in.
8. Accept the Licensing Agreement and Terms of Services. Click **Finish**.
9. When the install is finished, a dialog prompts you to restart the IDE; do so.
10. When the IDE has reinitialized, repeat Step 2 to navigate to the Install New Software window via the Help menu.
11. Uncheck the Group items by category checkbox.
12. Click **Add**.

13. When the Add Repository window appears, click **Archive…**.



Name: OpenCL Debugger
Location: jar:file:/C:/AdrenoSDK/Bin/Debugger/Eclipse/OpenCLPlu

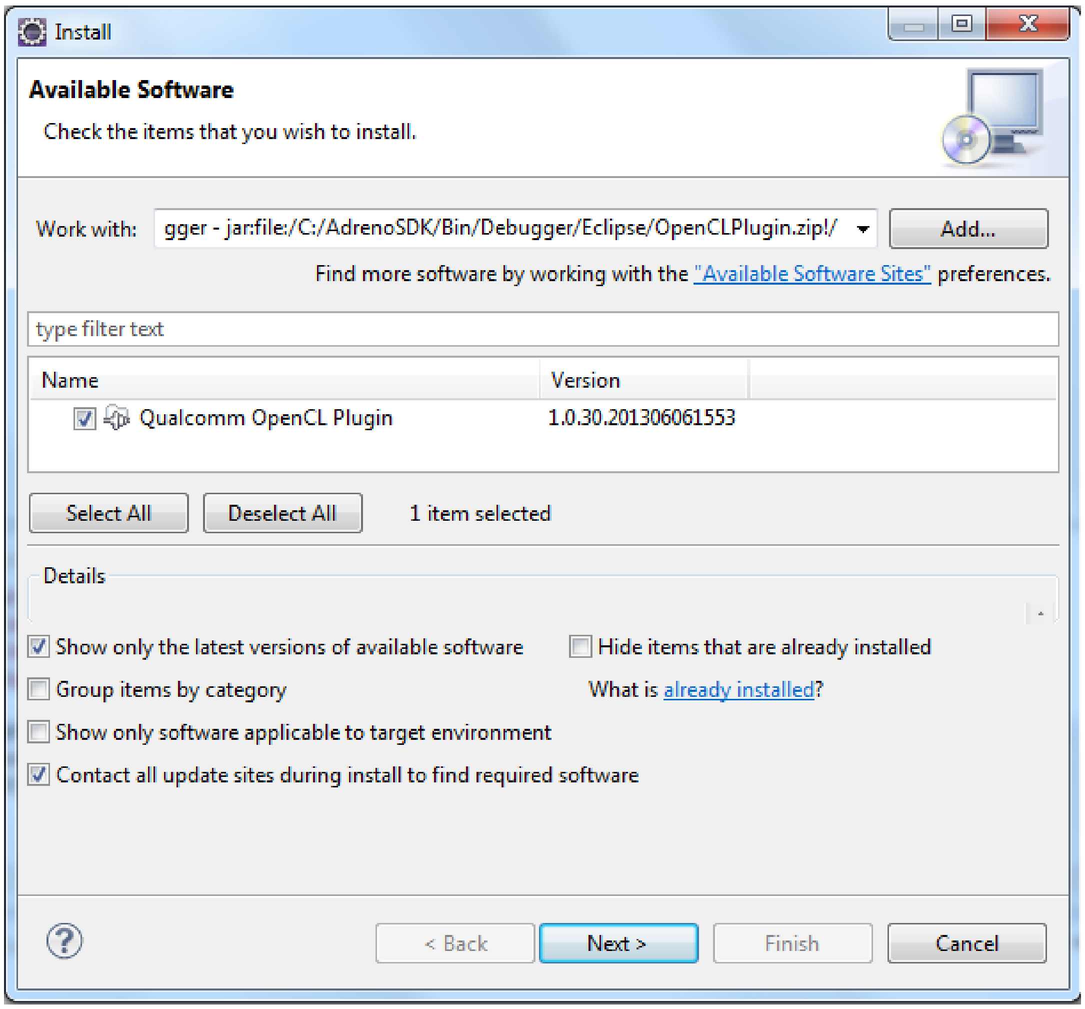
☐ Group items by category
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

14. Navigate to the directory containing the OpenCLPlugin.zip file and select it.

NOTE: For Windows systems, the default location of this file is C:\AdrenoSDK\Bin\Debugger\Eclipse\OpenCLPlugin.zip. For Linux systems, the default location is AdrenoSDK/Bin/Debugger/Eclipse/OpenCLPlugin.zip.

15. Click **OK**.

16. When the Install window has refreshed, select the checkbox next to Qualcomm OpenCL Plugin and click **Next**.



Install
Available Software
Check the items that you wish to install.
Work with: gger - jar:file:/C:/AdrenoSDK/Bin/Debugger/Eclipse/OpenCLPlugin.zip!/
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Select All
Deselect All
Details
What is already installed?
< Back
Next >
Finish
Cancel

Name	Version
☑ Qualcomm OpenCL Plugin	1.0.30.201306061553

1 item selected

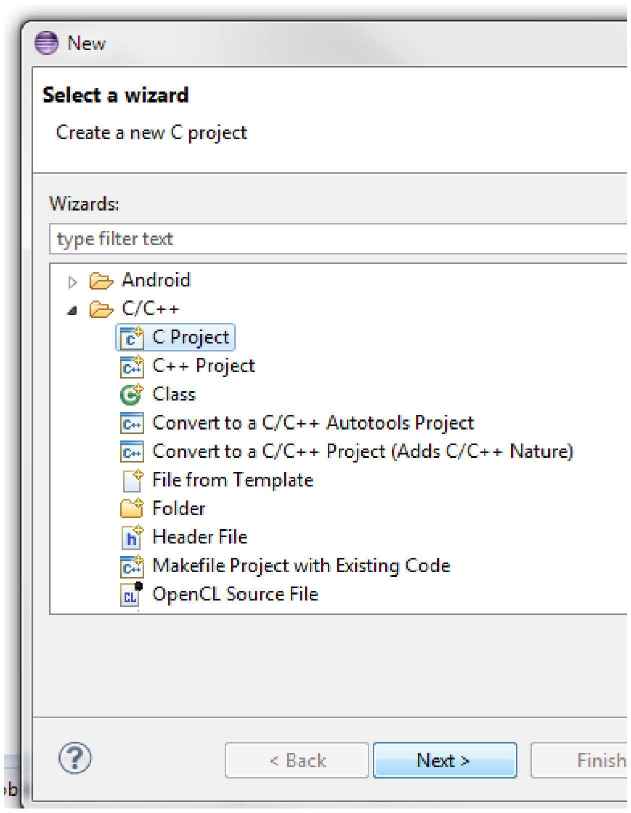
- ☑ Show only the latest versions of available software
- ☐ Hide items that are already installed
- ☐ Group items by category
- ☐ Show only software applicable to target environment
- ☑ Contact all update sites during install to find required software

17. A new window appears. Click **Next** again.
18. Accept the Licensing Agreement and Terms of Services. Click **Finish**.
19. Restart the IDE when prompted.

3.3.3 Configuring Eclipse plug-in installation

To configure the OpenCL Debugger plug-in for the Eclipse IDE:

1. Go to the File menu and select **New→Project**.
2. In the ensuing window, expand C/C++ and select C Project.

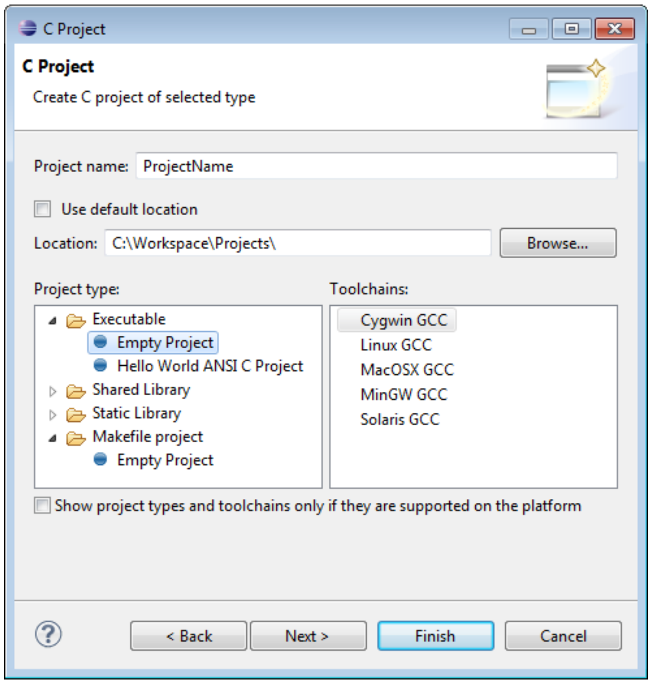


Select a wizard

Create a new C project

3. Enter a project name and select **Executable→Empty Project**.

NOTE: If **Executable→Empty Project** is not visible, uncheck Show project types and toolchains only if they are supported on the platform.



C Project

C Project

Create C project of selected type

Project name: ProjectName

☐ Use default location

Location: C:\Workspace\Projects\

Project type:
- Executable
  - Empty Project
  - Hello World ANSI C Project
- Shared Library
- Static Library
- Makefile project
  - Empty Project

Toolchains:
- Cygwin GCC
- Linux GCC
- MacOSX GCC
- MinGW GCC
- Solaris GCC

☐ Show project types and toolchains only if they are supported on the platform

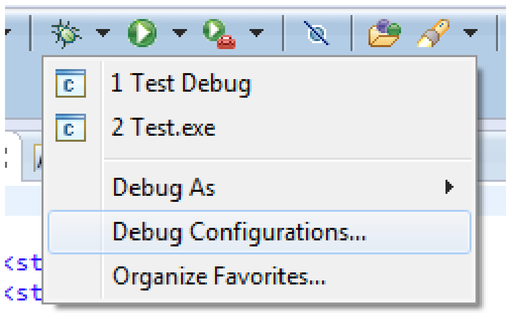
4. Click **Finish.**

NOTE: If you have existing source code, you can import it into your project by right-clicking and selecting **Import→File System** and then navigating to your source code.

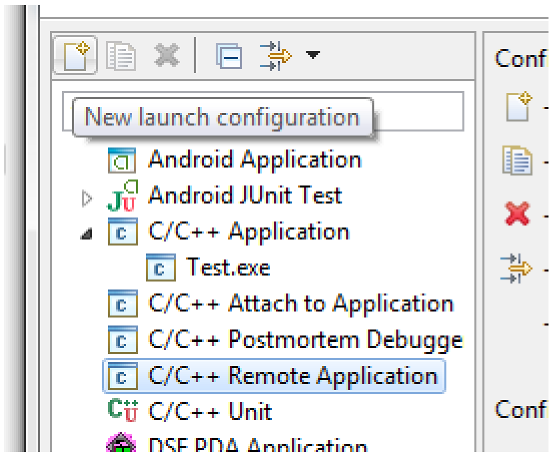
Typical files may include OpenCL files, C/C++ source files, and varying directory structures.

Steps 5 through 19 describe how to set a new Debug Configuration to enable OpenCL debugging.

5. Click the small arrow next to the debug icon and select **Debug Configurations**.



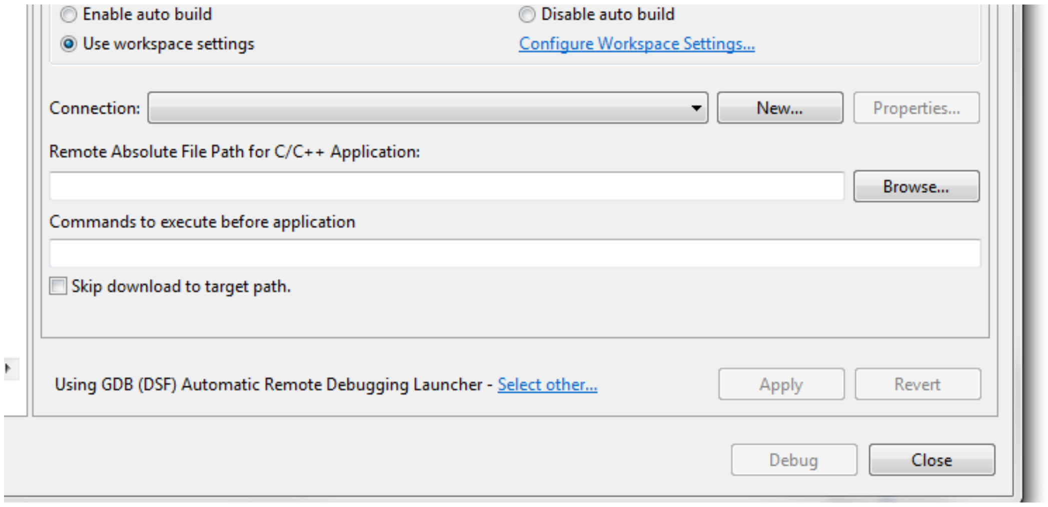
6. Select the C/C++ Remote Application and click the New Launch Configuration icon that appears toward the top of the window.



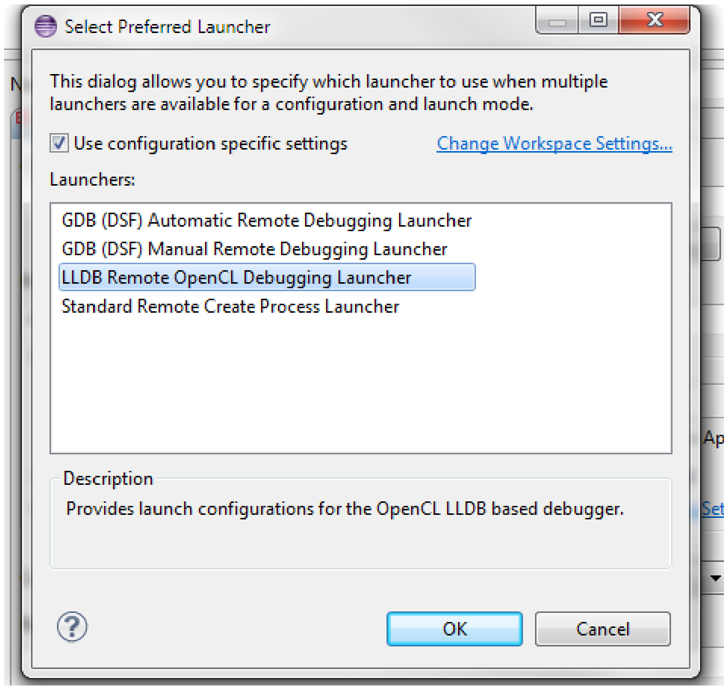
7. At the bottom of the window that appears after you have selected New Launch Configuration, you will see – next to the **Apply** button – a label reading Multiple launchers available. Click **Select one...**.

NOTE: In some instances, a launcher may already be selected. In this case, the label contains the text "Using <currently selected debugging launcher> - Select other…".

WARNING: Do not select Automatic Remote Debugging Launcher.


Enable auto build
Disable auto build
Use workspace settings
Configure Workspace Settings...
Connection:
New...
Properties...
Remote Absolute File Path for C/C++ Application:
Browse...
Commands to execute before application
Skip download to target path.
Using GDB (DSF) Automatic Remote Debugging Launcher - Select other...
Apply
Revert
Debug
Close

8. Select the Use configuration specific settings checkbox.



Select Preferred Launcher

This dialog allows you to specify which launcher to use when multiple launchers are available for a configuration and launch mode.

[✓] Use configuration specific settings — Change Workspace Settings...

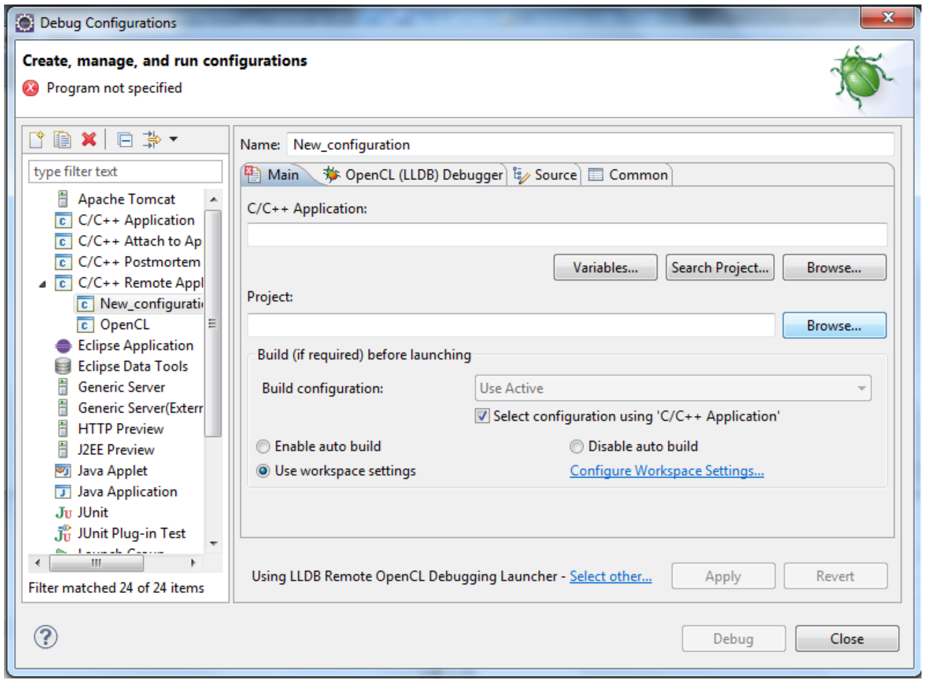
Launchers:

- GDB (DSF) Automatic Remote Debugging Launcher
- GDB (DSF) Manual Remote Debugging Launcher
- LLDB Remote OpenCL Debugging Launcher
- Standard Remote Create Process Launcher

Description

Provides launch configurations for the OpenCL LLDB based debugger.

9. Select LLDB Remote OpenCL Debugging Launcher and click **OK**.
10. In the Main tab of the Launch Configuration window, there is a Project text field. Click **Browse** and choose the project you wish to debug.



Debug Configurations

Create, manage, and run configurations

Program not specified

Name: New_configuration

C/C++ Application:

Project:

Build (if required) before launching

Build configuration: Use Active

[✓] Select configuration using 'C/C++ Application'

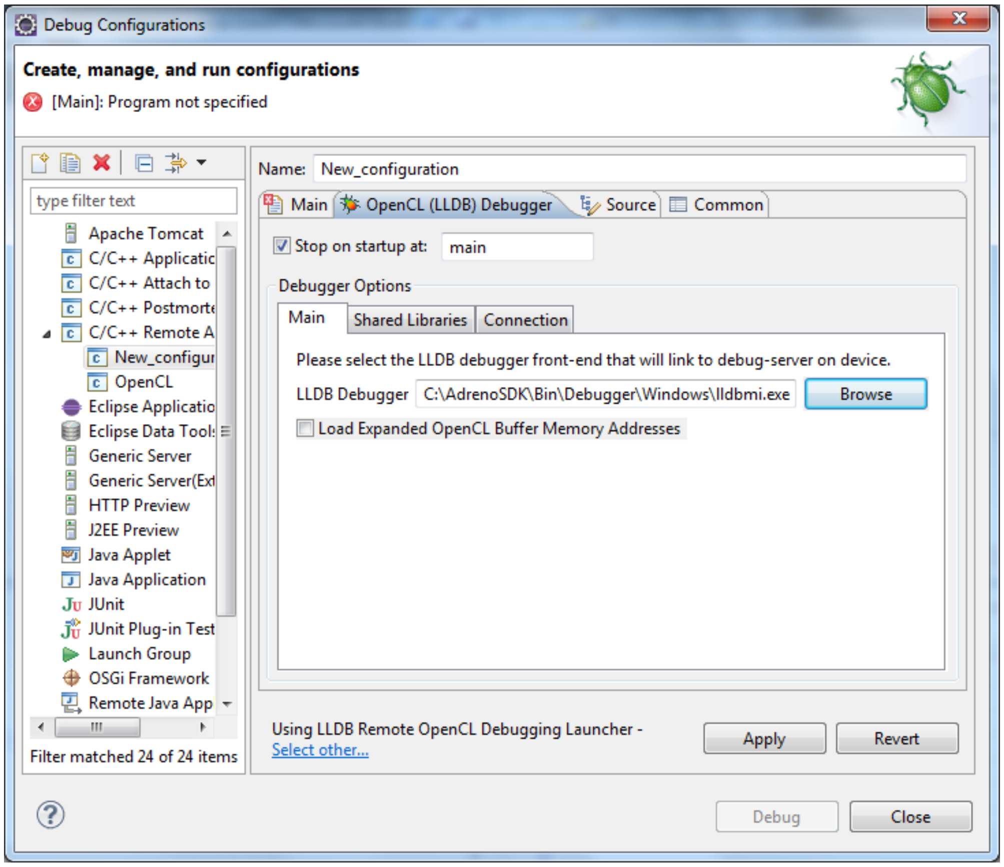
( ) Enable auto build ( ) Disable auto build

(●) Use workspace settings — Configure Workspace Settings...

Using LLDB Remote OpenCL Debugging Launcher - Select other...

11. Select your newly created project and click **OK**.
12. Click **Search project…** or **Browse…** and select the binary to debug

13. Click the OpenCL (LLDB) Debugger tab.


Debug Configurations

Create, manage, and run configurations

[Main]: Program not specified

Name: New_configuration

Tabs: Main | OpenCL (LLDB) Debugger | Source | Common

[✓] Stop on startup at: main

Debugger Options

Main | Shared Libraries | Connection

Please select the LLDB debugger front-end that will link to debug-server on device.

LLDB Debugger: C:\AdrenoSDK\Bin\Debugger\Windows\lldbmi.exe [Browse]

[ ] Load Expanded OpenCL Buffer Memory Addresses

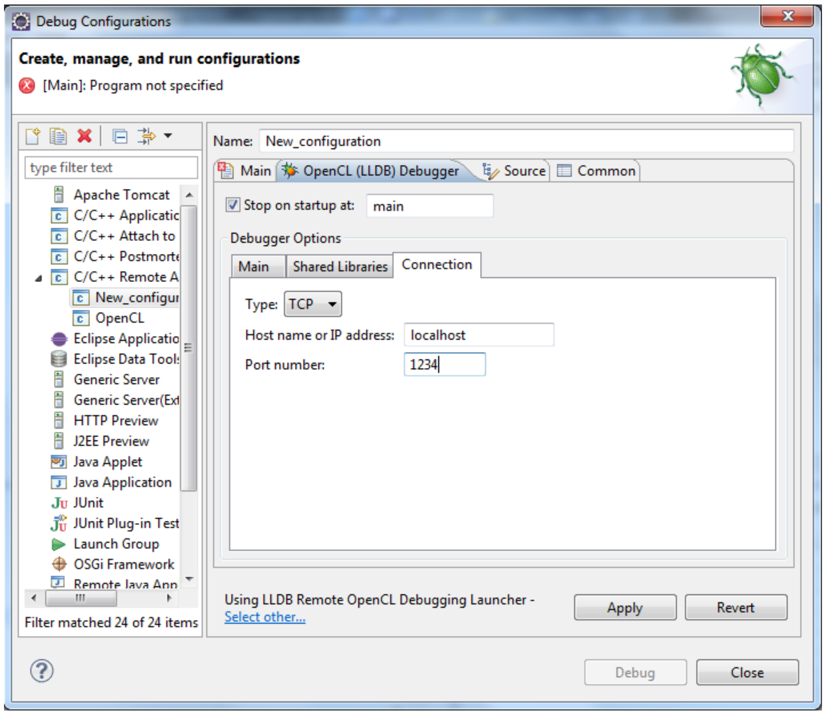
Using LLDB Remote OpenCL Debugging Launcher - Select other...

Filter matched 24 of 24 items

14. Click **Browse** to specify the lldbmi executable.

NOTE: For Windows systems, the default location of this file is C:\AdrenoSDK\Bin\Debugger\Windows\lldbmi.exe. For Linux systems, the default location is AdrenoSDK/Bin/Debugger/Linux/lldbmi.

15. Click the Connection tab.



Debug Configurations

Create, manage, and run configurations

[Main]: Program not specified

type filter text

Apache Tomcat
C/C++ Applicatic
C/C++ Attach to
C/C++ Postmorte
C/C++ Remote A
New_configur
OpenCL
Eclipse Applicatio
Eclipse Data Tool:
Generic Server
Generic Server(Ext
HTTP Preview
J2EE Preview
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Launch Group
OSGi Framework
Remote Java App

Filter matched 24 of 24 items

Name: New_configuration

Main | OpenCL (LLDB) Debugger | Source | Common

Stop on startup at: main

Debugger Options

Main | Shared Libraries | Connection

Type: TCP

Host name or IP address: localhost

Port number: 1234

Using LLDB Remote OpenCL Debugging Launcher - Select other...

16. Make sure localhost is entered in the Host name or IP address field.
17. Enter port 1234 in the Port number field.
18. Click **Apply**.
19. Click **Debug** to attach to debugserver if already running. If it is not running, see Chapter 5 for instructions on using the command line debugger to launch debugserver.

4 General OpenCL Debugging

4.1 Breakpoints

When debugging a process that loads a debug-enabled version of the OpenCL driver, various breakpoint options are available that are useful in many debugging scenarios.

4.1.1 Kernel entry breakpoint

A kernel entry breakpoint acts much like a breakpoint on a function on many other debuggers, in that it stops at the entry to the kernel. It is a pending breakpoint and offers the ability to specify what type of device you want to stop on, be it the CPU, GPU, or both. The fact that it is pending is very useful, as this makes it possible to set the breakpoint before the OpenCL subsystem has initialized and before the kernel itself has been compiled.

Kernel entry breakpoints stop on every work item possible, one at a time. All work items in the work group are visible at a kernel entry breakpoint, with each work item shown as a separate thread. In both cases of the CPU and GPU, multiple work items – the number of work items per work group – are visible at a kernel entry breakpoint.

4.1.2 Conditional kernel entry breakpoint

This breakpoint is conditional on a work-item specifier such as 12, or 12,0,0. It is the same as a kernel entry breakpoint in that it is pending, so it can be set before the kernel is loaded, but does not have as rich a feature set; e.g., you cannot specify having a conditional kernel entry breakpoint only on GPU kernels.

4.1.3 Program line breakpoint

You can set breakpoints on a per-line basis of OpenCL program source. Unlike the kernel entry breakpoints, this is not pending, and the program must be loaded into the system for it to work correctly. The usual workflow is to conditionally breakpoint on the item required and then toggle the breakpoint flag of the OpenCL program line.

4.1.4 Event breakpoint

Event breakpoints can halt execution of the program when an OpenCL command event occurs. OpenCL Events include:

- CL_QUEUED – Command has been enqueued.
- CL_SUBMITTED – Command has been submitted to the compute device.
- CL_RUNNING – Compute device is executing the command.
- CL_COMPLETE – Command has completed.

4.2 Debugging GPU details

When debugging kernels running on the Adreno GPU, there are various differences in feature set and debugger use than would be experienced on the Snapdragon CPU. On the GPU you may see a great number of new threads in the debugging process. Each of these new threads is a work item executing on the GPU. Debugging work items as threads allows use of the existing thread switch and other commands such as frame variable to examine work item state for an entire work-group.

- GPU kernels are restricted to the local maximum number of work items per work group. This is handled automatically in the Adreno GPU driver.
- When running GPU kernels, breakpoints trigger all work items in the work group to suspend at the specified breakpoint.
- When single-stepping or continuing execution after being stopped at a GPU breakpoint, all work items in the work group step or restart.
- When single-stepping on a single GPU thread, all other GPU threads in the same work group step at the same time.
- On GPU local information is limited to the topmost frame of the stack.
- On GPU locals of user-defined types cannot be examined with correct type information.
- On GPU you cannot view registers.
- On GPU you cannot view disassembly.
- On GPU you cannot edit the value of local memory.

Although there are some differences between GPU debugging and CPU debugging, both provide a rich set of features, and the developer’s experience should be similar for each.

5 Using the Command Line Debugger

This chapter discusses the command line debugger, LLDB. For Eclipse plug-in usage, see Chapter 6.

5.1 Current capabilities

The debugger can run simple kernels via the ND Range OpenCL calls. The debugger can debug both host ARM code and driver-built OpenCL kernels.

Implemented features

- Setting breakpoints on OpenCL kernel entry points
- Setting general code breakpoints (symbol, PC location)
- Single-stepping
- Reading/writing memory and register locations
- Kernel source-level debugging
- Cross-platform Linux and Windows hosts

Planned features

- OpenCL expression evaluation

5.2 Limitations

The debug driver limits OpenCL execution to a single worker thread. This can increase execution time drastically. In addition, the debugger is fetching data for each kernel, which may also slow execution.

5.3 Installing device files

Assuming the device is already set up as a developer to use the OpenCL libraries, use the Android Debug Bridge (ADB) to transfer files to the device as shown below.

- debugserver (required)
- hello-cl-cpu (example)
- hello-cl-gpu (example)

```
DEVTARGET=/data/local/qdebugger
adb shell mkdir $DEVTARGET

# Copy executables to device
for FILE in debugserver hello-cl-cpu hello-cl-gpu; do
    adb push PATH/TO/EXECUTABLES/$FILE $DEVTARGET
done
```

5.4 Before debugging – Caching files

The debugserver and LLDB instances require the binaries of the debugger processes to operate correctly. This includes all dynamic libraries. Debugserver sends LLDB these, if not found locally in its cache. The cache, located at /tmp/lldb-device/ on Linux and C:\tmp\lldb-device on Windows, mirrors the device file system. To prevent very slow operation, the dynamic libraries that are known, e.g., libOpenCL.so and libllvm-arm-debug.so, should be manually copied to mirror their position on the device.

5.5 Mapping compiled source paths to local LLDB paths

A command is available to map directories to others such that a binary compiled on Linux can be debugged effectively on Windows and vice versa.

```
(lldb) settings set target.source-map /home/user/cl_app
D:\workspace\cl_app\
```

This maps the first directory to the second. Therefore, if a source file compiled was:

```
/home/user/cl_app/main.c
```

It would look for the source file instead at:

```
D:\workspace\cl_app\main.c
```

5.6 Command line debugging with LLDB

This section describes the steps needed to successfully debug on the device.

5.6.1 Forwarding ports

Ports must be forwarded via ADB to set up debugserver-to-LLDB communications, e.g., using port 1234:

```
$ adb forward tcp:1234 tcp:1234
```

5.6.2 Launching on target device

To make sure everything is in order, run the program to be debugged without debugger first:

```
$ adb shell
root@android:/data/local/qdebugger # ./hello-cl-cpu
num_platforms: 1, platforms[0] = de763ed3
Selected device is: CPU
Device id: 40176f40
clCreateContext passed
clCreateCommandQueue passed
clCreateProgramWithSource passed
clBuildProgram passed
Computed '1024/1024' correct values!
```

To debug, you must execute the following command on the device:

```
./debugserver localhost:1234 <program> <progam_args>
```

NOTE: You can use another port other than 1234, if required. <program_args> is optional.

You are presented with the following output on successful debugserver launch:

```
./debugserver localhost:1234 ./hello-cl-cpu
debugserver-193 for armv7.
Listening to port 1234...
```

Once a debugging connection from LLDB is established, the additional text is displayed:

```
Got a connection, waiting for debugger instructions.
```

5.6.3 Launching LLDB

Launch LLDB via the Windows command prompt using C:\AdrenoSDK\Bin\Debugger\Windows\lldb.exe, or via a Linux terminal, using ~/AdrenoSDK/Bin/Debugger/Linux/lldb.

Upon launch, LLDB prompts for commands.

```
(lldb)
```

Create a new target for the application to be debugged:

```
(lldb) target create --platform remote-android <path/to/local/exeutable>
```

Connect to the remote target, specifying the port that was forwarded in previous steps, in this case 1234:

```
(lldb) process connect connect://localhost:1234 --plugin gdb-remote
```

On successful connection, the target process is spawned and stopped, awaiting user input:

```
Process 9457 stopped
* thread #1: tid = 0x24f1, 0xb0001000, stop reason = trace
    frame #0: 0xb0001000
-> 0xb0001000:  mov    r0, sp
   0xb0001004:  mov    r1, #0
   0xb0001008:  blx    0xffffffffb0004578
   0xb000100c:  mov    pc, r0
(lldb)
```

You can then enter LLDB commands as normal to debug the process. To use OpenCL commands, the libOpenCL.so library must have been loaded by the process. The easiest way to be sure of this is to set a breakpoint on main and continue until it is reached.

```
(lldb) breakpoint set -n main
Breakpoint 1: where = hello-cl-cpu`main at hello.c:199, address =
0x00008b1c
(lldb) continue
Process 15192 resuming
Process 15192 stopped
* thread #1: tid = 0x3b58, 0x00008b1c hello-cl-cpu`main(argc=1,
argv=0xbef36be4) at hello.c:199, stop reason = breakpoint 1.1
    frame #0: 0x00008b1c hello-cl-cpu`main(argc=1, argv=0xbef36be4) at
hello.c:199
…
```

For a full list of commands, type **help** or **apropos opencl** in the LLDB prompt, or see Appendix A.

6 Using the OpenCL Debugger Plug-in

This chapter describes usage of the OpenCL Debugger plug-in for Eclipse. See Section 3.3 for installation and configuration. See Section 5.6.2 for launching the device applications.

All views described in this chapter are part of the OpenCL Debug perspective.

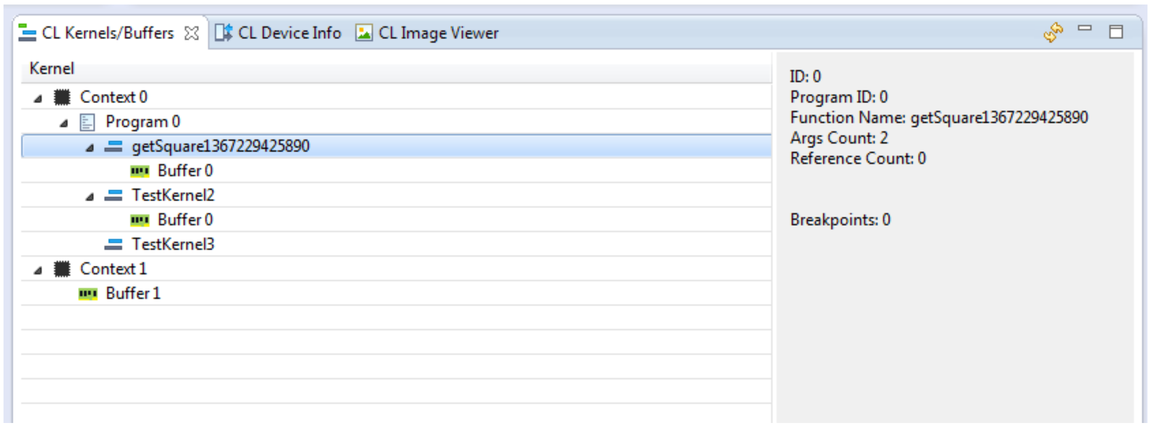
NOTE: Eclipse may ask if you want to switch to the Debug perspective. Ensure that you are using the correct OpenCL Debug perspective and not the regular Debug perspective.

6.1 Kernel view

The kernel view is designed to allow you to quickly view the context-program-kernel hierarchy on your debuggable device. There can be a number of different kernels per program and a number of different programs per context as outlined in [R3]. You can also view attached buffers that specific kernels may address.

6.1.1 Viewing item-specific information

You can view more information about each item in the view's hierarchy by clicking the item. When you do so, the right side of the view containing a text view populates; the Kernel view is shown here.

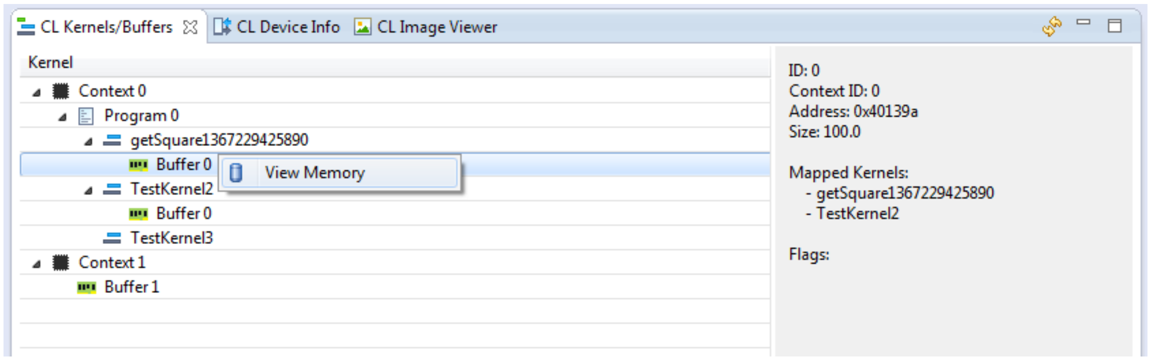


ID: 0
Program ID: 0
Function Name: getSquare1367229425890
Args Count: 2
Reference Count: 0

Breakpoints: 0

6.1.2 Viewing a buffer memory trace

The kernel view allows you to easily and quickly view the memory trace of a buffer. To do this, right-click a buffer in the kernel view and select View Memory.

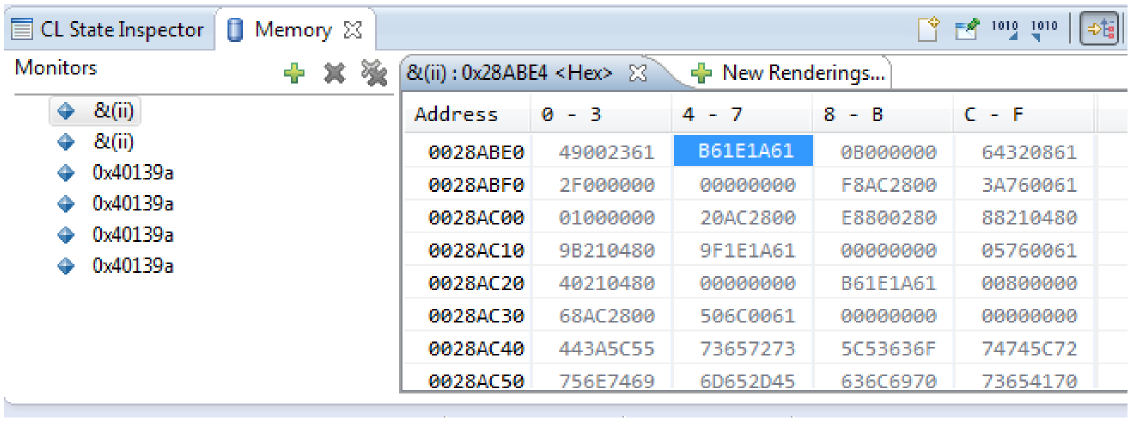


ID: 0
Context ID: 0
Address: 0x40139a
Size: 100.0

Mapped Kernels:
- getSquare1367229425890
- TestKernel2

Flags:

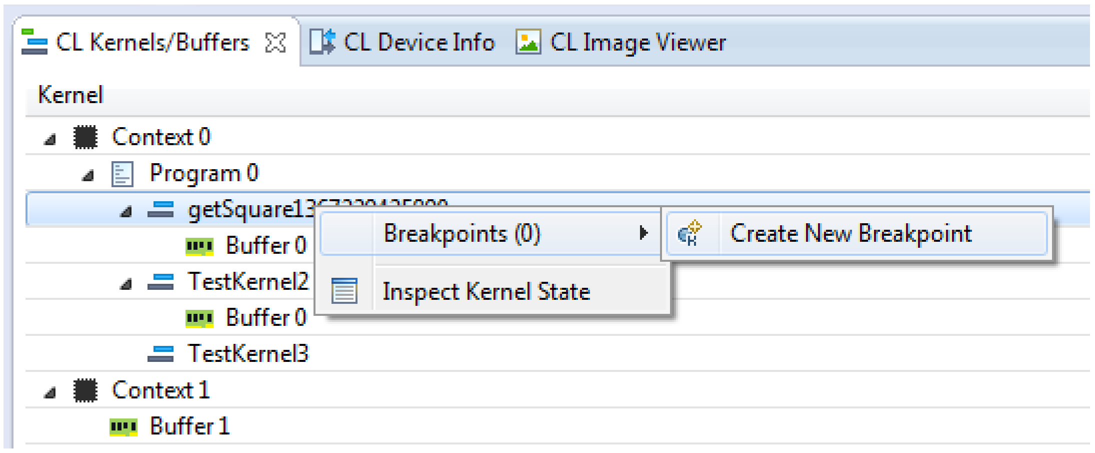
The memory view then autoinitializes and allows you to view the buffer. If the Select Rendering window appears, select the first option and click **OK**.



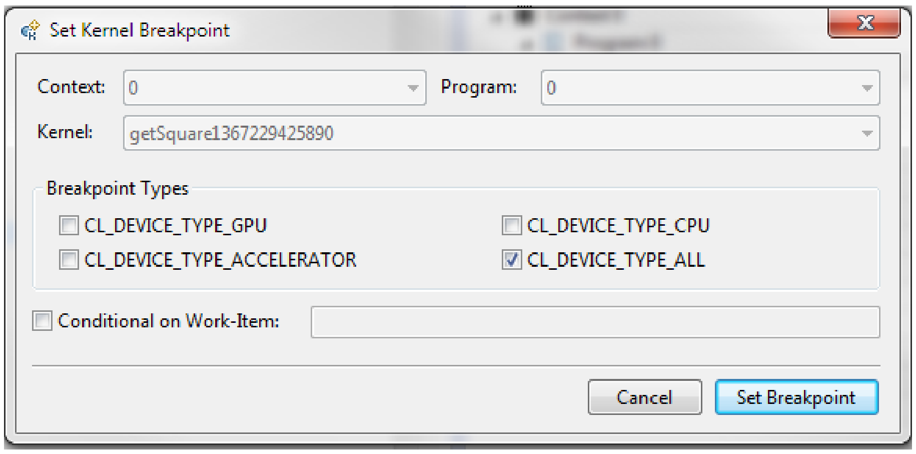
```
Address   0 - 3     4 - 7     8 - B     C - F
0028ABE0  49002361  B61E1A61  0B000000  64320861
0028ABF0  2F000000  00000000  F8AC2800  3A760061
0028AC00  01000000  20AC2800  E8800280  88210480
0028AC10  9B210480  9F1E1A61  00000000  05760061
0028AC20  40210480  00000000  B61E1A61  00800000
0028AC30  68AC2800  506C0061  00000000  00000000
0028AC40  443A5C55  73657273  5C53636F  74745C72
0028AC50  756E7469  6D652D45  636C6970  73654170
```

6.1.3 Setting a kernel breakpoint

The kernel view allows you to easily set a kernel breakpoint. To do this, right-click a kernel in the kernel window and select Create New Breakpoint.



The CL breakpoint window gains focus and a pop-up breakpoint creator window appears, allowing you to set all the specific breakpoint settings.



Context: 0
Program: 0
Kernel: getSquare1367229425890

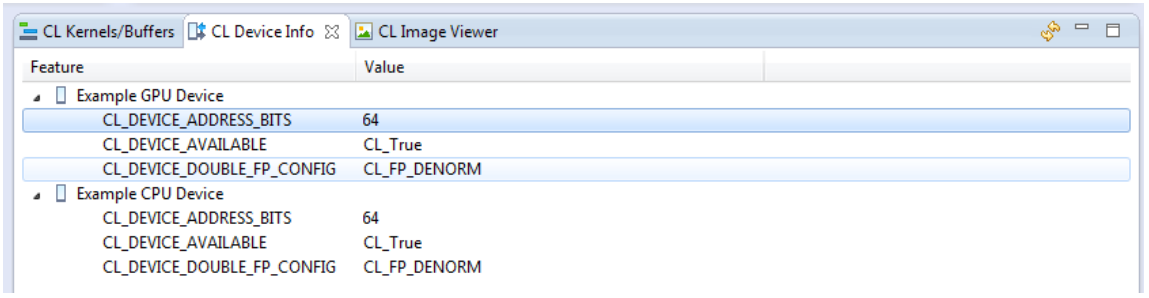
Breakpoint Types

- [ ] CL_DEVICE_TYPE_GPU
- [ ] CL_DEVICE_TYPE_CPU
- [ ] CL_DEVICE_TYPE_ACCELERATOR
- [x] CL_DEVICE_TYPE_ALL

- [ ] Conditional on Work-Item:

6.2 Device info view

The device info view is designed to allow you to quickly see the OpenCL supported hardware. Each hardware item can contain a number of specific settings represented by a set of key/value pairs. The device info view contains no interactive features but allows you to quickly analyze the setup of your device and the settings it utilizes.



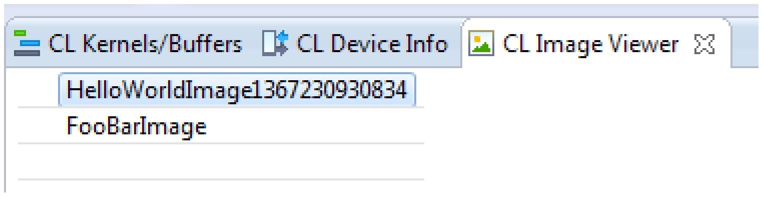
Feature	Value
Example GPU Device	
CL_DEVICE_ADDRESS_BITS	64
CL_DEVICE_AVAILABLE	CL_True
CL_DEVICE_DOUBLE_FP_CONFIG	CL_FP_DENORM
Example CPU Device	
CL_DEVICE_ADDRESS_BITS	64
CL_DEVICE_AVAILABLE	CL_True
CL_DEVICE_DOUBLE_FP_CONFIG	CL_FP_DENORM

6.3 Image viewer

The image viewer allows you to quickly view the current in-context images and then view them on demand. Each image can then be saved directly to the host file system. This view also automatically updates when the image changes on-device while debugging.

6.3.1 Image list

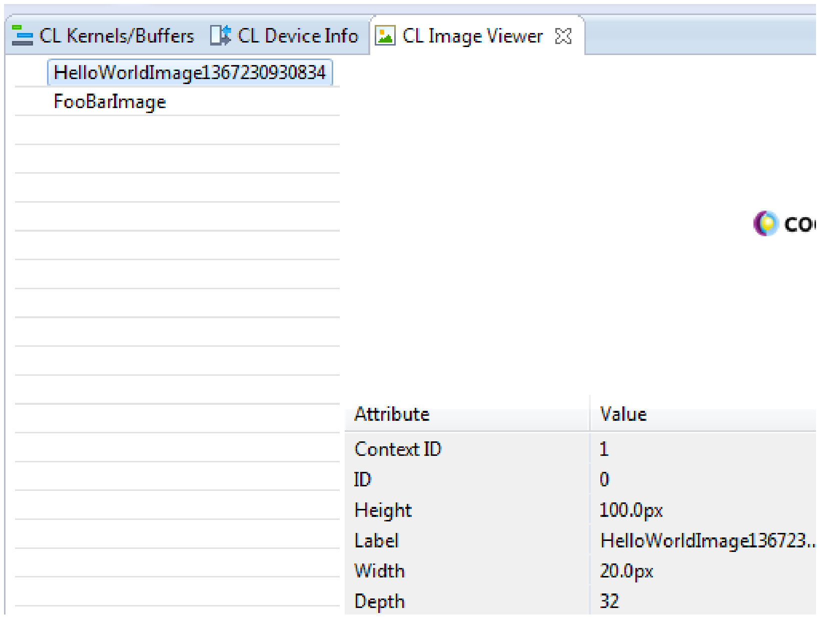
The image list represents the current list of in-context images. Clicking each image dynamically loads the image and its details into the right side of the image view.



HelloWorldImage1367230930834
FooBarImage

6.3.2 Image details

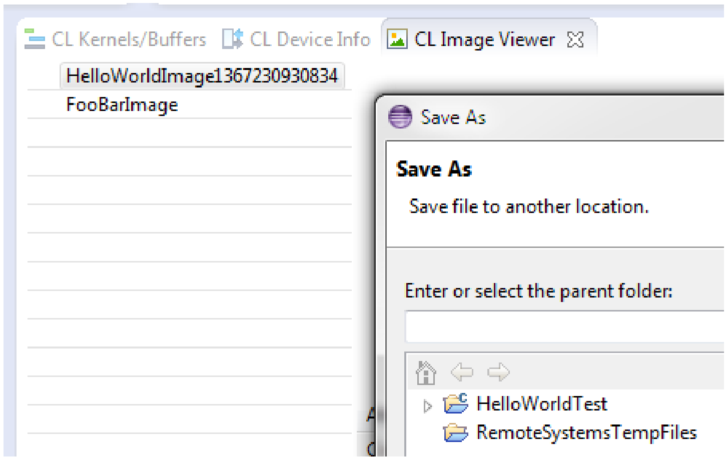
Each image also contains a list of image details such as width, height, and depth. These can be found under the image in the image view.



Attribute	Value
Context ID	1
ID	0
Height	100.0px
Label	HelloWorldImage136723..
Width	20.0px
Depth	32

6.3.3 Saving an image

You can save a local copy of an image from the device via the **Save** icon.

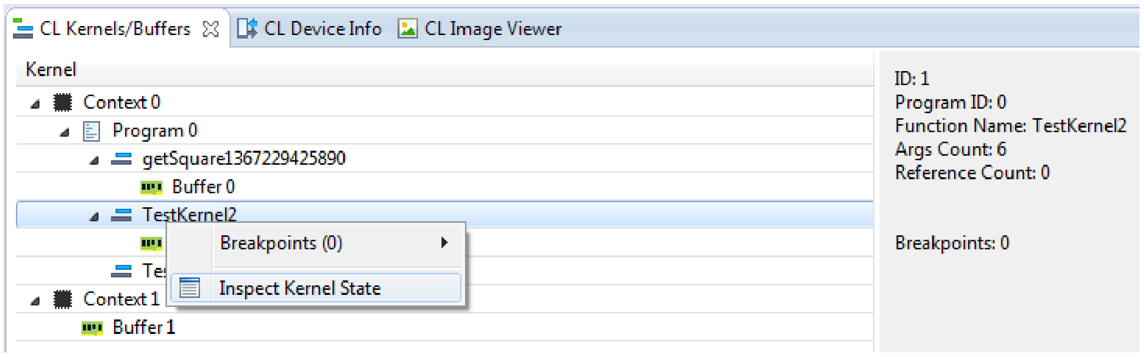


6.4 State inspector view

The state inspector view allows you to view information about a kernel’s state and automatically retrieve updates when the debugging state changes. The Eclipse plug-in can track a number of different kernel states during the debugging process.

6.4.1 Adding a new state to track

To add a new state to track, click the kernel view. Right-click the kernel you wish to track and select Inspect Kernel State.

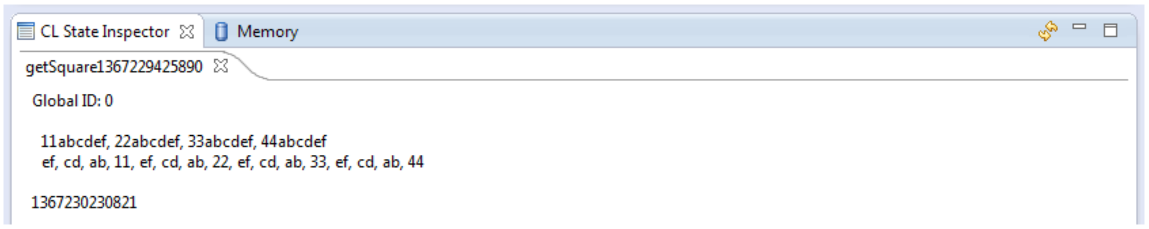


ID: 1
Program ID: 0
Function Name: TestKernel2
Args Count: 6
Reference Count: 0

Breakpoints: 0

6.4.2 Viewing kernel states

You can inspect the state of several different kernels while debugging. Shown below is an example of some information you may expect. The information provided changes based on back-end support for state information.



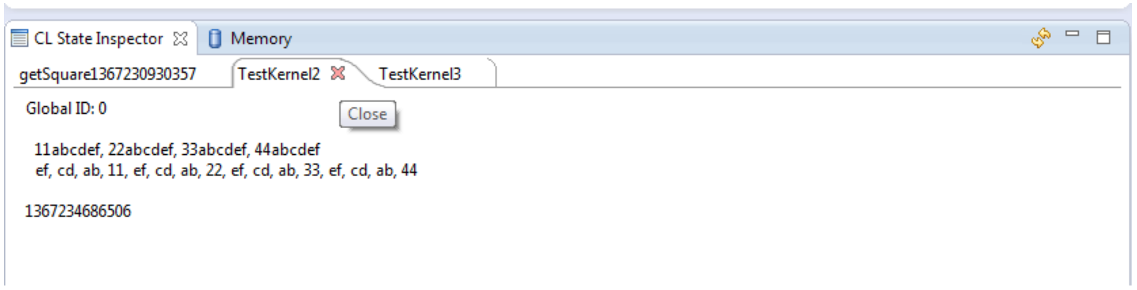
```
Global ID: 0

 11abcdef, 22abcdef, 33abcdef, 44abcdef
 ef, cd, ab, 11, ef, cd, ab, 22, ef, cd, ab, 33, ef, cd, ab, 44

1367230230821
```

6.4.3 Removing a kernel state tracker

You can easily remove a kernel from the state tracker by clicking the kernel's state tab and clicking the **Close** icon to the right of the kernel name.



```
Global ID: 0

 11abcdef, 22abcdef, 33abcdef, 44abcdef
 ef, cd, ab, 11, ef, cd, ab, 22, ef, cd, ab, 33, ef, cd, ab, 44

1367234686506
```

6.5 Breakpoint window

The breakpoint window allows you to set and manage a number of breakpoints. Breakpoints can be either pending (a breakpoint on a kernel that is currently not in scope) or live (set a breakpoint on a current kernel) and can have a number of different parameters such as conditions and types.

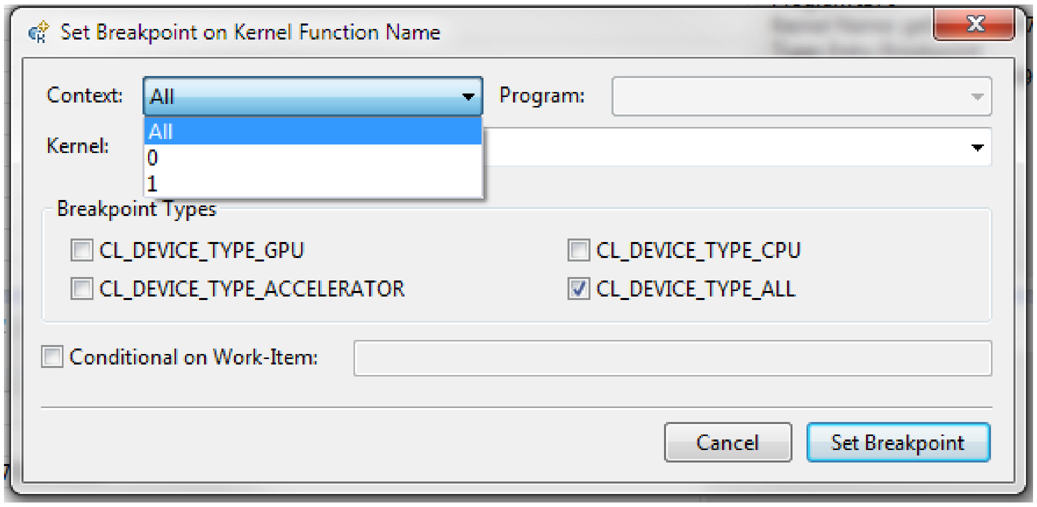
6.5.1 Breakpoint window overview

The breakpoint window consists of a list containing created breakpoints, information view with information about each breakpoint, and an action bar allowing you to create new breakpoints and refresh the view. Clicking each breakpoint in the breakpoint view populates the information view with specific information about the breakpoint, e.g., any attached condition, type of breakpoint, e.g., GPU, CPU, Accelerator, etc., and also information about the kernel to which it is set.

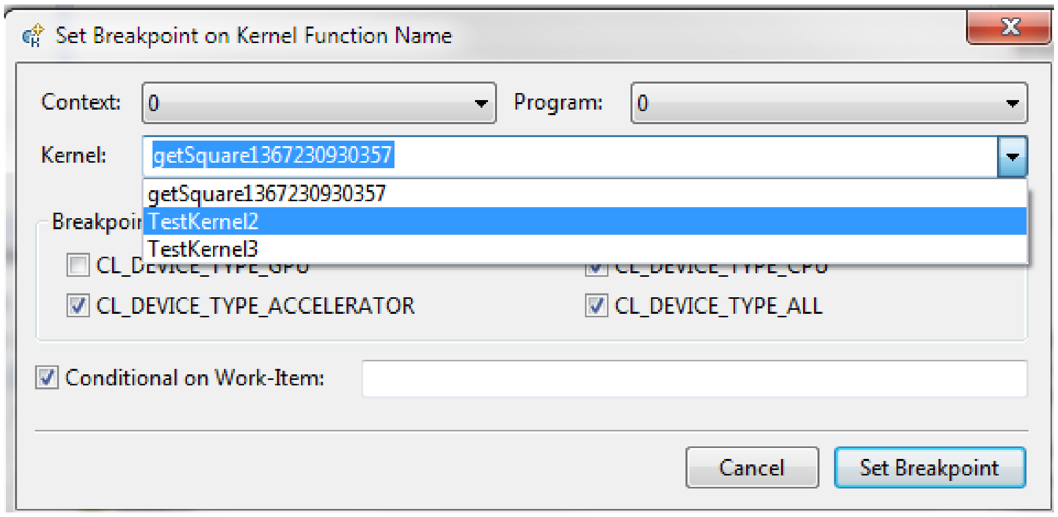
6.5.2 Setting a breakpoint on an existing kernel

To create a breakpoint on an existing kernel:

1. Click **Add New Breakpoint** at the top right of the view.
2. A window pops up, allowing you to enter breakpoint-specific information. To set the breakpoint on a kernel, you must select the kernel's parent context and program using the provided drop-down menus:



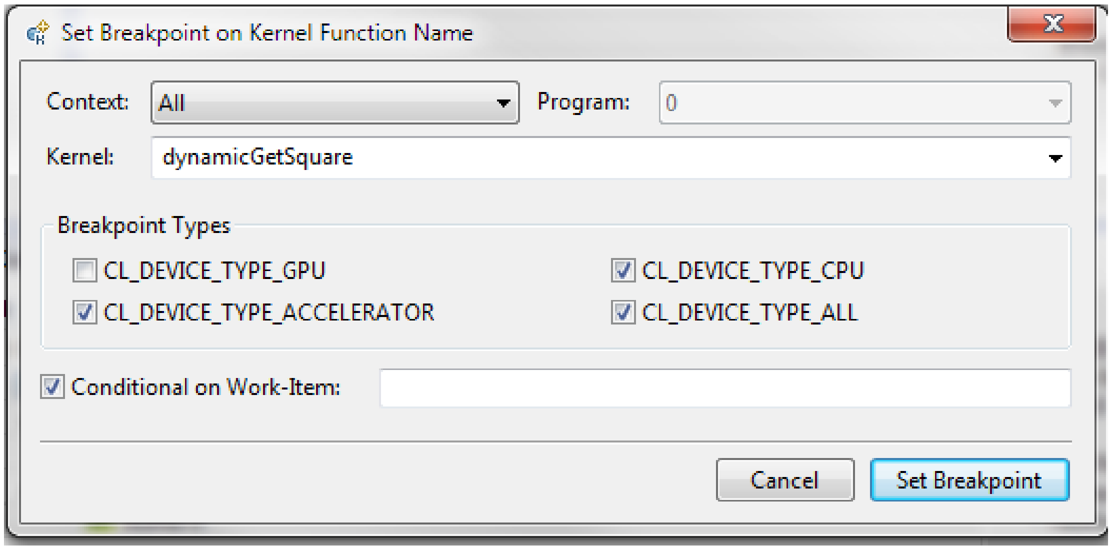
3. Select the kernel name from the Kernel drop-down menu, and click **Set Breakpoint** to set the breakpoint.



6.5.3 Setting a pending breakpoint

You can also create a breakpoint on a kernel that is currently not in scope or as a global breakpoint that will hit when any kernel with the same name and conditions entered occurs. To create a pending breakpoint:

1. Click **Add New Breakpoint** at the top right of the breakpoint view. A pop-up appears and you can begin creating your pending breakpoint.
2. Select **All** from the Context drop-down menu. The Program drop-down menu becomes gray.
3. Click the Kernel drop-down field and type the name of the kernel you wish to set your pending breakpoint on as shown below.


Set Breakpoint on Kernel Function Name

Context: All

Program: 0

Kernel: dynamicGetSquare

Breakpoint Types

- [ ] CL_DEVICE_TYPE_GPU
- [✓] CL_DEVICE_TYPE_CPU
- [✓] CL_DEVICE_TYPE_ACCELERATOR
- [✓] CL_DEVICE_TYPE_ALL

[✓] Conditional on Work-Item:

4. Click **Set Breakpoint** to set the breakpoint.

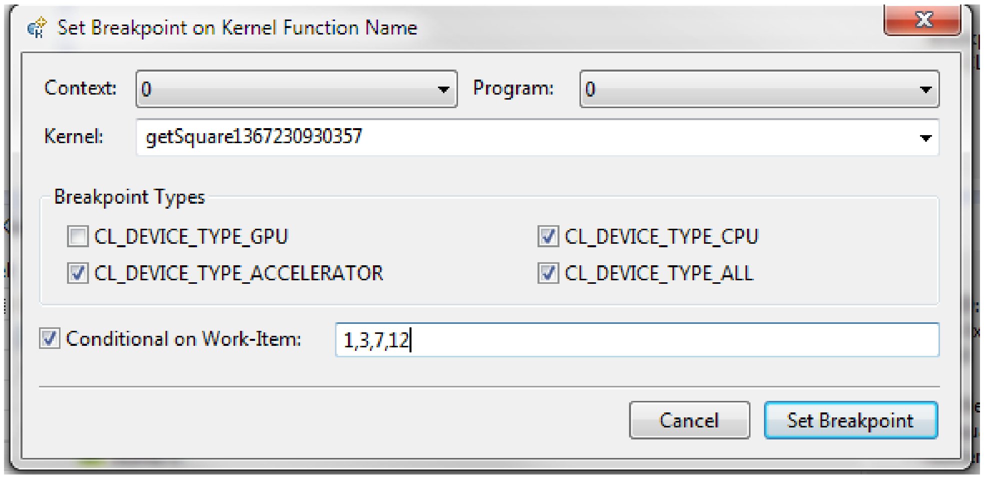
6.5.4 Breakpoint types

Breakpoint types can be configured to ensure that the breakpoint is only hit if the breakpoint occurs on specific platforms, e.g., you can create a breakpoint for a specific kernel that only hits if the current device is of type GPU or Accelerator. To do this, check the CL_DEVICE_TYPE_GPU and CL_DEVICE_TYPE_ACCELERATOR checkboxes. By default, CL_DEVICE_TYPE_ALL is selected, meaning the breakpoint hits on all devices.

6.5.5 Breakpoint conditions

Breakpoint conditions allow you to specify that a breakpoint should only be hit if a certain condition occurs; e.g., you can set a breakpoint on a specific work item number. To set a condition:

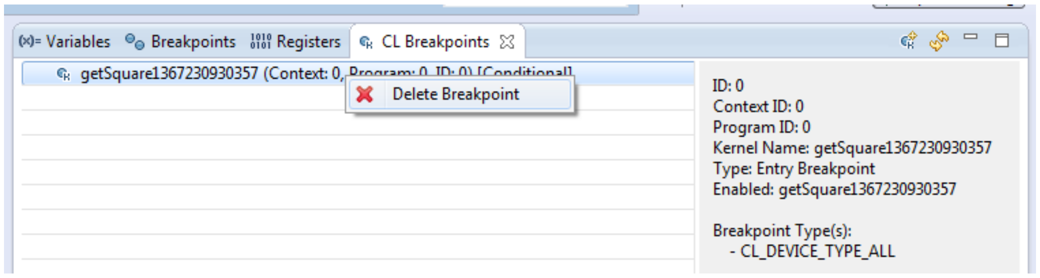
1. Click the checkbox next to Conditional on Work-Item.
2. In the text box to the right, enter your condition, e.g., 1,2 or 1,2,5, etc.
3. Click **Set Breakpoint** to set a conditional breakpoint.



Context: 0
Program: 0
Kernel: getSquare1367230930357
[ ] CL_DEVICE_TYPE_GPU
[✓] CL_DEVICE_TYPE_CPU
[✓] CL_DEVICE_TYPE_ACCELERATOR
[✓] CL_DEVICE_TYPE_ALL
[✓] Conditional on Work-Item: 1,3,7,12

6.5.6 Removing a breakpoint

To remove a breakpoint, right-click a breakpoint in the breakpoint window and click **Delete Breakpoint**.



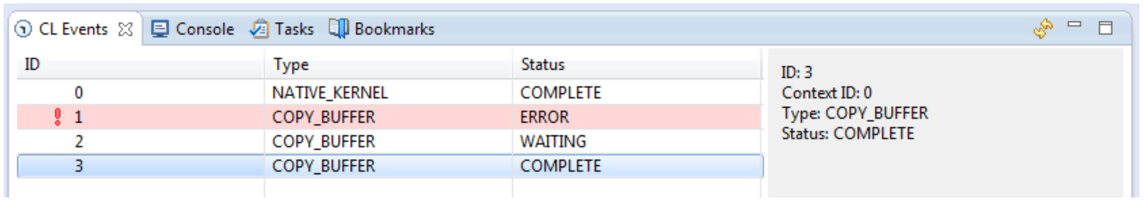
getSquare1367230930357 (Context: 0, Program: 0, ID: 0) [Conditional]

ID: 0
Context ID: 0
Program ID: 0
Kernel Name: getSquare1367230930357
Type: Entry Breakpoint
Enabled: getSquare1367230930357

Breakpoint Type(s):
- CL_DEVICE_TYPE_ALL

6.6 Events view

You can also track and view CL events using the built-in CL event viewer view. The view consists of a list of events and an information dialog that you can use to view more information about the specific event. To view more information about a specific event, click the desired event in the event list.

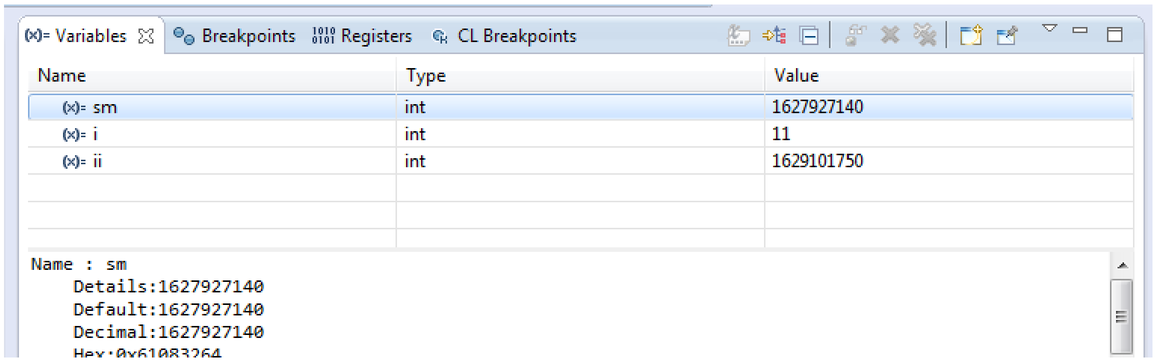


ID	Type	Status
0	NATIVE_KERNEL	COMPLETE
1	COPY_BUFFER	ERROR
2	COPY_BUFFER	WAITING
3	COPY_BUFFER	COMPLETE

```
ID: 3
Context ID: 0
Type: COPY_BUFFER
Status: COMPLETE
```

6.7 Variables view/variable hover

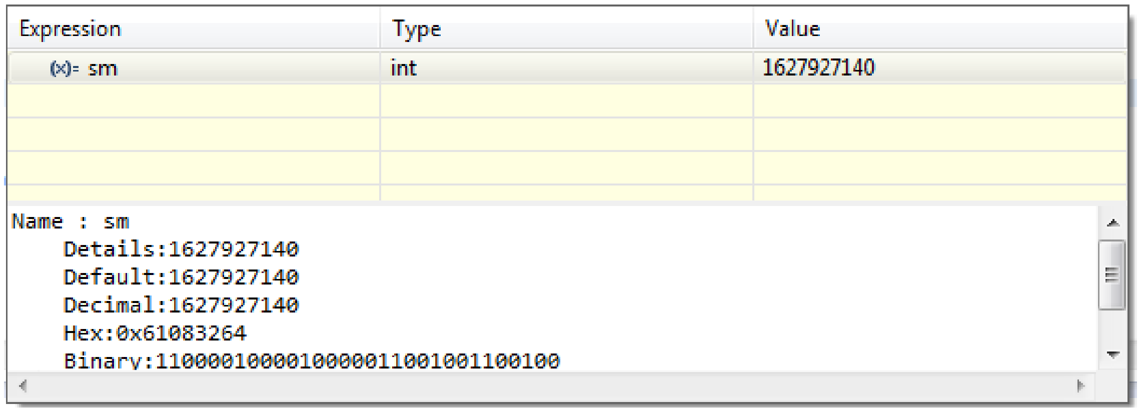
The variables view allows you to quickly view the in-scope variables in your current debugging session. You can change these values or just use them as a reference.



Name	Type	Value
sm	int	1627927140
i	int	11
ii	int	1629101750

```
Name : sm
    Details:1627927140
    Default:1627927140
    Decimal:1627927140
    Hex:0x61083264
```

Eclipse also supports the ability to quickly view information about a variable by hovering over it in the source editor. When doing so, a pop-up presents itself with detailed variable information.



Expression	Type	Value
sm	int	1627927140

```
Name : sm
    Details:1627927140
    Default:1627927140
    Decimal:1627927140
    Hex:0x61083264
    Binary:1100001000010000011001001100100
```

NOTE: In some cases, you need to focus the variables view at least once in your active debugging session to force Eclipse to poll for variable information. To do this, click the Variables tab. If it is not in your current perspective, select it from **Window→Show View→Variables View**.

6.8 Perspectives

The Eclipse plug-in has two ways of managing your debug session, the standard debug integration recommended by the Eclipse plug-in debugger design team and a perspective that contains everything set up.

Choose the standard debug perspective if you are using your Eclipse IDE for multiple different purposes. Choose the customized OpenCL Debug perspective if you are only using the IDE for debugging via LLDB.

6.8.1 Selecting the OpenCL Debug perspective

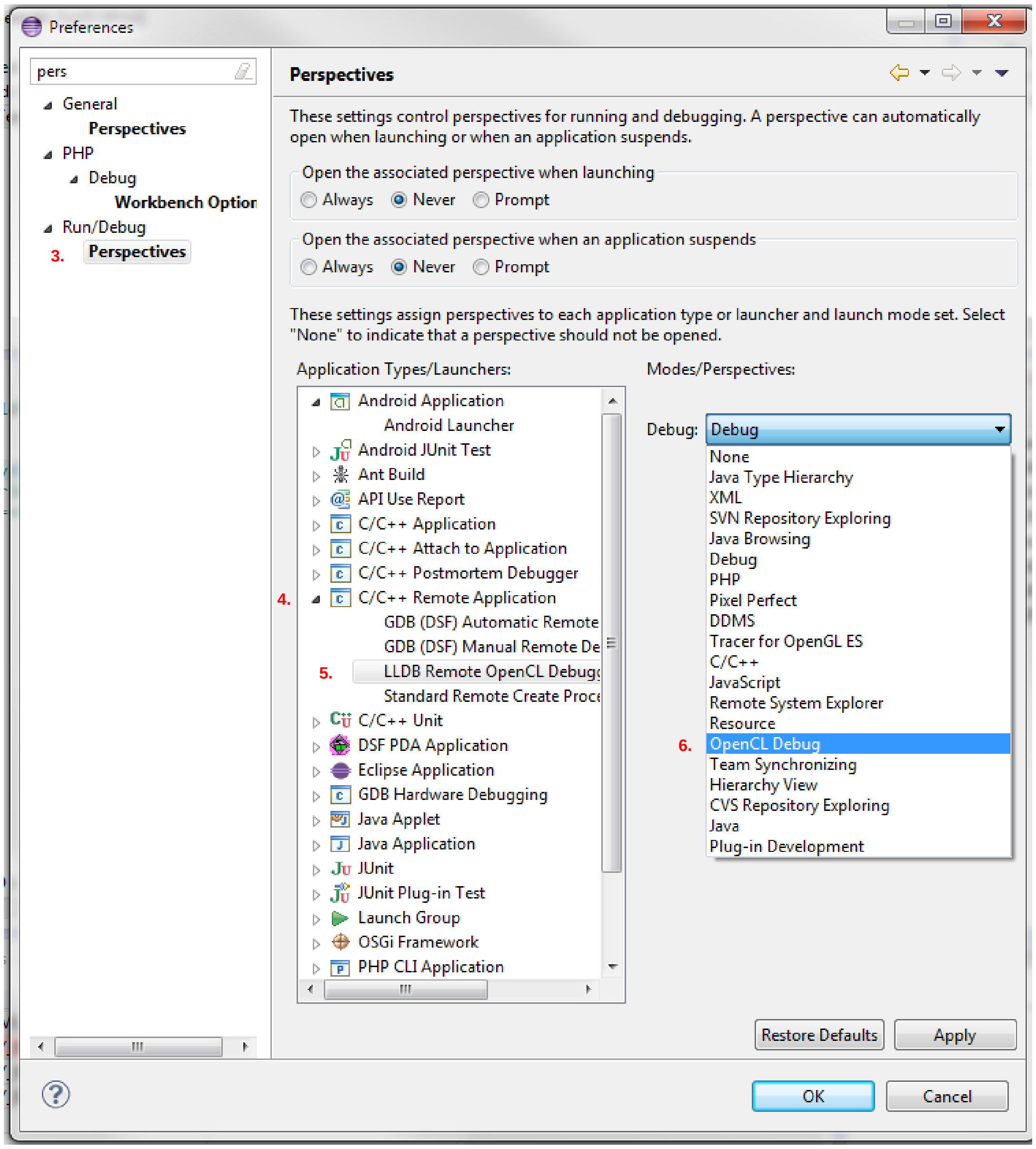
To select the OpenCL Debug perspective:

1. From the Window menu, click **Open Perspective**.
2. Click **Other**. A window appears.
3. Click **OpenCL Debug**. Once done, you will have all standard supported debug views and all OpenCL-specific debug views set up inside your perspective.

6.8.2 Setting Default Debug perspective

Each time you create a new debugging session, step, or resume, Eclipse returns to the standard Debug perspective. You must change the default perspective to use:

1. From the Window menu, click **Preferences**.
2. Type “perspective” in the filter box (usually at the top left of preferences dialog).
3. Select Perspectives shown under Run/Debug to the left. This gives the following layout:



Perspectives

These settings control perspectives for running and debugging. A perspective can automatically open when launching or when an application suspends.

Open the associated perspective when launching
Always [✓ Never] Prompt

Open the associated perspective when an application suspends
Always [✓ Never] Prompt

These settings assign perspectives to each application type or launcher and launch mode set. Select "None" to indicate that a perspective should not be opened.

Application Types/Launchers:

- Android Application
  - Android Launcher
- Android JUnit Test
- Ant Build
- API Use Report
- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- C/C++ Remote Application
  - GDB (DSF) Automatic Remote
  - GDB (DSF) Manual Remote De
  - LLDB Remote OpenCL Debugg
  - Standard Remote Create Proce
- C/C++ Unit
- DSF PDA Application
- Eclipse Application
- GDB Hardware Debugging
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Launch Group
- OSGi Framework
- PHP CLI Application

Modes/Perspectives:

Debug: Debug

None
Java Type Hierarchy
XML
SVN Repository Exploring
Java Browsing
Debug
PHP
Pixel Perfect
DDMS
Tracer for OpenGL ES
C/C++
JavaScript
Remote System Explorer
Resource
OpenCL Debug
Team Synchronizing
Hierarchy View
CVS Repository Exploring
Java
Plug-in Development

4. Click **C/C++ Remote Application** in the Application Types/Launchers selection box.
5. Click **LLDB Remote OpenCL Debugger**.
6. Click **OpenCL Debug** in the Debug drop-down menu.

A Debugger Commands

A.1 General LLDB commands

```

(lldb) help
The following is a list of built-in, permanent debugger commands:
_regexp-attach    -- Attach to a process id if in decimal, otherwise treat the
                     argument as a process name to attach to.
_regexp-break     -- Set a breakpoint using a regular expression to specify the
                     location, where <linenum> is in decimal and <address> is
                     in hex.
_regexp-bt        -- Show a backtrace.  An optional argument is accepted; if
                     that argument is a number, it specifies the number of
                     frames to display.  If that argument is 'all', full
                     backtraces of all threads are displayed.
_regexp-display   -- Add an expression evaluation stop-hook.
_regexp-down      -- Go down "n" frames in the stack (1 frame by default).
_regexp-env       -- Implements a shortcut to viewing and setting environment
                     variables.
_regexp-list      -- Implements the GDB 'list' command in all of its forms
                     except FILE:FUNCTION and maps them to the appropriate
                     'source list' commands.
_regexp-tbreak    -- Set a one shot breakpoint using a regular expression to
                     specify the location, where <linenum> is in decimal and
                     <address> is in hex.
_regexp-undisplay -- Remove an expression evaluation stop-hook.
_regexp-up        -- Go up "n" frames in the stack (1 frame by default).
apropos           -- Find a list of debugger commands related to a particular
                     word/subject.
breakpoint        -- A set of commands for operating on breakpoints. Also see
                     _regexp-break.
command           -- A set of commands for managing or customizing the debugger
                     commands.
disassemble       -- Disassemble bytes in the current function, or elsewhere in
                     the executable program as specified by the user.
expression        -- Evaluate a C/ObjC/C++ expression in the current program
                     context, using user defined variables and variables
                     currently in scope.
```

```
frame             -- A set of commands for operating on the current thread's
                     frames.
gdb-remote        -- Connect to a remote GDB server.  If no hostname is
                     provided, localhost is assumed.
help              -- Show a list of all debugger commands, or give details
                     about specific commands.
kdp-remote        -- Connect to a remote KDP server.  udp port 41139 is the
                     default port number.
log               -- A set of commands for operating on logs.
memory            -- A set of commands for operating on memory.
opencl            -- A set of commands for operating on OpenCL state.
platform          -- A set of commands to manage and create platforms.
plugin            -- A set of commands for managing or customizing plugin
                     commands.
process           -- A set of commands for operating on a process.
quit              -- Quit out of the LLDB debugger.
register          -- A set of commands to access thread registers.
script            -- Pass an expression to the script interpreter for
                     evaluation and return the results. Drop into the
                     interactive interpreter if no expression is given.
settings          -- A set of commands for manipulating internal settable
                     debugger variables.
source            -- A set of commands for accessing source file information
target            -- A set of commands for operating on debugger targets.
thread            -- A set of commands for operating on one or more threads
                     within a running process.
type              -- A set of commands for operating on the type system
version           -- Show version of LLDB debugger.
watchpoint        -- A set of commands for operating on watchpoints.
```

For more information on any particular command, try 'help <command-name>'.OpenCL specific commands.

A.2 OpenCL-specific commands

```
(lldb) apropos opencl
The following commands may relate to 'opencl':
opencl                           -- A set of commands for operating on
                                    OpenCL state.
opencl breakpoint                -- A set of commands for manipulating
                                    breakpoints on OpenCL events. You can
                                    set breakpoints on program lines using
                                    the 'opencl program break' command.
opencl breakpoint event          -- A set of commands that allow stopping
```

```
                                          the execution of the program when the
                                          status of OpenCL events changes.
opencl breakpoint event clear          -- Reset all OpenCL event breakpoints.
opencl breakpoint event disable        -- Disable stopping execution when the
                                          status of OpenCL events changes.
opencl breakpoint event enable         -- Enable stopping executions when the
                                          status of OpenCL events changes.
opencl breakpoint event show           -- Show whether execution should be
                                          stopped when the status of OpenCL
                                          events change.
opencl breakpoint kernel               -- A set of commands for manipulating
                                          breakpoints on OpenCL Kernel entries.
opencl breakpoint kernel delete        -- Delete a breakpoint entry to the
                                          specified kernel name.
opencl breakpoint kernel list          -- List all the kernel entry breakpoints
                                          active in the debugger
opencl breakpoint kernel set           -- Set a breakpoint on entry to the
                                          specified kernel name.
opencl breakpoint programline delete -- Delete a breakpoint on an OpenCL
                                          program line
opencl breakpoint programline list     -- List the breakpoints for an OpenCL
                                          program
opencl breakpoint programline set      -- Set a breakpoint on an OpenCL program
                                          line
opencl breakpoint work-item            -- A set of commands for manipulating
                                          breakpoints on specific work-items of
                                          OpenCL N-D ranges.
opencl breakpoint work-item delete     -- Delete one or more work-item
                                          breakpoints.
opencl breakpoint work-item list       -- List work-item breakpoints for a
                                          specific kernel.
opencl breakpoint work-item set        -- Create a breakpoint that is hit before
                                          a specific work-item is processed.
opencl context                         -- A set of commands for operating on
                                          OpenCL contexts.
opencl context buffers                 -- Show a summary of all created OpenCL
                                          buffers in a context.
opencl context image                   -- A set of commands for operating on
                                          OpenCL images.
opencl context image dump              -- Dump an OpenCL image to Bitmap file.
opencl context image list              -- Show a summary of all created OpenCL
                                          images in a context.
opencl context list                    -- Show a summary of all loaded OpenCL
                                          contexts in a process.
```

```
opencl context select              -- Select the OpenCL context to use for
                                      other OpenCL commands.
opencl kernel                      -- A set of commands for operating on
                                      OpenCL kernels.
opencl kernel list                 -- Show a summary of all loaded OpenCL
                                      kernels in a process.
opencl kernel status               -- Show the status of current stopped
                                      OpenCL kernel.
opencl options                     -- A set of commands for manipulating
                                      OpenCL debugging options.
opencl options build               -- Define how OpenCL programs should be
                                      built.
opencl program                     -- A set of commands for operating on
                                      OpenCL programs.
opencl program assemdump           -- Dump the assembly of current stopped
                                      OpenCL program to a file.
opencl program list                -- Show a summary of all loaded OpenCL
                                      programs in a context.
opencl program source              -- Show the source compiled for this
                                      OpenCL program.
```

B Example LLDB Session

This appendix provides some example LLDB commands and output for debugging sessions to introduce features and functionality.

B.1 Features in example

Some actions performed in the provided example debugging session include:

- Source debugging of OpenCL kernels
- Setting and hitting breakpoints on an OpenCL kernel name, conditionally on work-item
- Single-stepping inside of an OpenCL kernel
- Debugging GPU OpenCL kernel
- Listing of OpenCL constructs (kernels, programs, contexts)
- Viewing local variables within OpenCL kernels
- Reading and writing memory
- Reading and writing registers

B.2 Example LLDB session

Attaching to the debugged process and stopping at main.

```
(lldb) target create --platform remote-android /tmp/lldb-
device/data/local/qdebugger/hello-cl-cpu
Current executable set to '/tmp/lldb-device/data/local/qdebugger/hello-cl-
cpu' (arm).
(lldb) process connect connect://localhost:1234 --plugin gdb-remote
Process 16679 stopped
* thread #1: tid = 0x3b58, 0xb0001000, stop reason = trace
    frame #0: 0xb0001000
-> 0xb0001000:  mov    r0, sp
   0xb0001004:  mov    r1, #0
   0xb0001008:  blx    0xb0004578
   0xb000100c:  mov    pc, r0
(lldb) b main
Breakpoint 1: where = hello-cl-cpu`main at hello.c:199, address =
0x00008b1c
(lldb) continue
Process 16679 resuming
Process 16679 stopped
* thread #1: tid = 0x3b58, 0x00008b1c hello-cl-cpu`main(argc=1,
argv=0xbef36be4) at hello.c:199, stop reason = breakpoint 1.1
    frame #0: 0x00008b1c hello-cl-cpu`main(argc=1, argv=0xbef36be4) at
hello.c:199
```

B.2.1 Setting breakpoints on a CPU OpenCL kernel

Set a breakpoint on any function called getsquare (this includes OpenCL kernel functions). Note that the warning below only means that no such function was found at the time of creating the breakpoint. Any library loaded by the process at runtime, such as OpenCL programs, is scanned for functions with this name to set a breakpoint on (as the message “x location added to breakpoint y” shows).

```
(lldb) opencl breakpoint kernel set -k getsquare
Breakpoint created
(lldb) continue
Process 16679 resuming
num_platforms: 1, platforms[0] = de763ed3
Selected device is: CPU
Device id: 400a3f40
clCreateContext passed
clCreateCommandQueue passed
clCreateProgramWithSource passed
1 location added to breakpoint 2
clBuildProgram passed
OpenCL Kernel Entry Breakpoint resolved to kernel getsquare[0] within
program [0] of context [0] OpenCL Kernel Entry Breakpoint resolved to
kernel getsquare[1] within program [0] of context [0] OpenCL Kernel Entry
Breakpoint with kernel getsquare[1] was resolved in error, disabling.
Process 16679 stopped
* thread #2: tid = 0x416c, 0x400c90dc cl_program_00001.so`getsquare + 4 at
cl_program_00002.cl:6, stop reason = breakpoint 2.1
    frame #0: 0x400c90dc cl_program_00001.so`getsquare + 4 at
cl_program_00002.cl:6
   3 __global int *input,
   4 __global int *output,
   5 const unsigned int count)
-> 6 {
   7 int i = get_global_id(0);
   8 if(i < count)
   9 output[i] = input[i] * input[i];
State for kernel 'getsquare', N-D range a7c9d0, chunk 0
   global_id = <0,0,0>
   global_size = <1024,1,1>
```

At this point, the process is stopped inside of the getsquare kernel. Any time this happens, the current kernel state is printed by the debugger (see global_id and global_size above). The same information can also be shown at any point later using the opencl kernel status command:

```
(lldb) opencl kernel status
State for kernel 'getsquare', N-D range a7c9d0, chunk 0
   global_id = <0,0,0>
   global_size = <1024,1,1>
```

As breakpoints are effective as long as they are not deleted, continuing the process leads to stopping at the beginning of the second work item, i.e., global_id is <1, 0, 0>.

```
(lldb) continue
Process 16679 resuming
Process 16679 stopped
* thread #2: tid = 0x416c, 0x400c90dc cl_program_00001.so`getsquare + 4 at
cl_program_00002.cl:6, stop reason = breakpoint 2.1
frame #0: 0x400c90dc cl_program_00001.so`getsquare + 4 at
cl_program_00002.cl:6
   3 __global int *input,
   4 __global int *output,
   5 const unsigned int count)
-> 6 {
   7 int i = get_global_id(0);
   8 if(i < count)
   9 output[i] = input[i] * input[i];
State for kernel 'getsquare', N-D range a7c9d0, chunk 0
   global_id = <1,0,0>
   global_size = <1024,1,1>
```

Similarly, continuing the process leads to stopping at the beginning of the third work item.

```
(lldb) continue
Process 16679 resuming
Process 16679 stopped
* thread #2: tid = 0x416c, 0x400c90dc cl_program_00001.so`getsquare + 4 at
cl_program_00002.cl:6, stop reason = breakpoint 2.1
frame #0: 0x400c90dc cl_program_00001.so`getsquare + 4 at
cl_program_00002.cl:6
   3 __global int *input,
   4 __global int *output,
   5 const unsigned int count)
-> 6 {
   7 int i = get_global_id(0);
   8 if(i < count)
   9 output[i] = input[i] * input[i];
State for kernel 'getsquare', N-D range a7c9d0, chunk 0
   global_id = <2,0,0>
   global_size = <1024,1,1>
```

To avoid stopping at the beginning of each work item, the breakpoint must be disabled or deleted. Continuing the execution will then allow the test program to finish normally without stopping.

```
(lldb) opencl breakpoint kernel delete –k getsquare
Breakpoints deleted
(lldb) continue
Process 16679 resuming
Computed '1024/1024' correct values!
Process 16679 exited with status = 0 (0x00000000)
```

B.2.2 Setting breakpoints on an OpenCL kernel name for certain work item

Continuing many times until the relevant work item is reached can be long and cumbersome. In this situation, you can use a work item breakpoint, which stops the execution at the beginning of the kernel before a given work item is executed.

Note that this only allows stopping at the entry of the kernel. To work around this limitation, you can create a conditional breakpoint and wait until it is reached. Then, you can set a program line breakpoint using the OpenCL program break.

The following extract shows the use of work-item breakpoints.

```
(lldb) opencl breakpoint work-item set -k getsquare -i 42
Added a breakpoint that will stop execution before work-item <42,0,0> is
processed by kernel 'getsquare'.
(lldb) continue
(…)
Process 15192 stopped
* thread #2: tid = 0x3bde, 0x400da0d8 cl_program_00001.so`getsquare at
cl_program_00002.cl:6, stop reason = breakpoint
    frame #0: 0x400da0d8 cl_program_00001.so`getsquare at
cl_program_00002.cl:6
   3 __global int *input,
   4 __global int *output,
   5 const unsigned int count)
-> 6 {
   7 int i = get_global_id(0);
   8 if(i < count)
   9 output[i] = input[i] * input[i];
State for kernel 'getsquare', N-D range 5589d0, chunk 0
   global_id = <42,0,0>
   global_size = <1024,1,1>
```

```
(lldb) opencl program break -p0 -l9
Breakpoint on line 9
(lldb) continue
Process 15192 stopped
* thread #2: tid = 0x3bde, 0x400da0d8 cl_program_00001.so`getsquare at
cl_program_00002.cl:6, stop reason = breakpoint
    frame #0: 0x400da100 cl_program_00001.so`getsquare at
cl_program_00002.cl:9
   7 int i = get_global_id(0);
   8 if(i < count)
-> 9 output[i] = input[i] * input[i];
   10 }
   11
State for kernel 'getsquare', N-D range 5589d0, chunk 0
   global_id = <42,0,0>
   global_size = <1024,1,1>
```

B.2.3 Debugging GPU OpenCL kernel

The entry breakpoint commands operate on all devices by default, so the same command triggers breakpoints on GPU:

```
(lldb) opencl breakpoint kernel set -k getsquare
Breakpoint created
(lldb) c
Process 2762 resuming
num_platforms: 1, platforms[0] = de763ed3
Selected device is: GPU
Device id: 409b0470(lldb)
clCreateContext passed
clCreateCommandQueue passed
clCreateProgramWithSource passed(lldb)
clBuildProgram passed(lldb)
OpenCL Kernel Entry Breakpoint resolved to kernel getsquare[0] within
program [0] of context [0]
OpenCL Kernel Entry Breakpoint resolved to kernel getsquare[1] within
program [0] of context [0]
OpenCL Kernel Entry Breakpoint with kernel getsquare[1] was resolved in
error, disabling.
* thread #3: tid = 0x7000000, 0x00000001
libOpenCL.so`getsquare(input=0x0000000040ec5000, output=0x0000000040ec7000,
count=1024) at cl_program_00309.cl:7, stop reason = trace
    frame #0: 0x00000001 libOpenCL.so`getsquare(input=0x0000000040ec5000,
output=0x0000000040ec7000, count=1024) at cl_program_00309.cl:7
   4 __global int *output,
   5 const unsigned int count)
   6 {
-> 7 int i = get_global_id(0);
   8 if(i < count)
   9 output[i] = input[i] * input[i];
   10 }
OpenCL GPU Execution Mode:
      GPU Group (0, 0, 0), Item (0, 0, 0): Kernel 'getsquare'
      local_id = <0,0,0>
      group_id = <0,0,0>
      global_size = <1024,0,0>
      global_offset = <0,0,0>
      num_groups = <4,0,0>
```

Standard commands can operate in GPU mode.

```
(lldb) bt
* thread #3: tid = 0x7000000, 0x00000001
libOpenCL.so`getsquare(input=0x0000000040ec5000, output=0x0000000040ec7000,
count=1024) at cl_program_00309.cl:7, stop reason = trace
    frame #0: 0x00000001 libOpenCL.so`getsquare(input=0x0000000040ec5000,
output=0x0000000040ec7000, count=1024) at cl_program_00309.cl:7
    frame #1: 0x00000001 libOpenCL.so`GPU Group (0, 0, 0), Item (0, 0, 0):
Kernel 'getsquare' + 1 at cl_program_00309.cl:1
(lldb) frame variable
(int) i = 0
(int*) input = 0x0000000040ec5000
(int*) output = 0x0000000040ec7000
(const unsigned int) count = 1024
```

When setting a conditional breakpoint while already stopped in a GPU kernel, the conditional work item must exist in the next work group, due to the fact it executes one work group at a time.

```
(lldb) opencl breakpoint kernel delete -i0
Breakpoint deleted
(lldb) opencl breakpoint work-item set -k getsquare -i 322
Added a breakpoint that will stop execution before work-item <322,0,0> is
processed by kernel 'getsquare'.
(lldb) c
Process 2762 resuming
* thread #69: tid = 0x7000042, 0x00000001
libOpenCL.so`getsquare(input=0x0000000040ec5000, output=0x0000000040ec7000,
count=1024) at cl_program_00309.cl:7, stop reason = trace
    frame #0: 0x00000001 libOpenCL.so`getsquare(input=0x0000000040ec5000,
output=0x0000000040ec7000, count=1024) at cl_program_00309.cl:7
   4 __global int *output,
   5 const unsigned int count)
   6 {
-> 7 int i = get_global_id(0);
   8 if(i < count)
   9 output[i] = input[i] * input[i];
   10 }
OpenCL GPU Execution Mode:
      GPU Group (1, 0, 0), Item (66, 0, 0): Kernel 'getsquare'
      local_id = <66,0,0>
      group_id = <1,0,0>
      global_size = <1024,0,0>
      global_offset = <0,0,0>
      num_groups = <4,0,0>
```

Single-step over the assignment to i.

```
(lldb) s
* thread #69: tid = 0x7000042, 0x00000002
libOpenCL.so`getsquare(input=0x0000000040ec5000, output=0x0000000040ec7000,
count=1024) at cl_program_00309.cl:8, stop reason = trace
    frame #0: 0x00000002 libOpenCL.so`getsquare(input=0x0000000040ec5000,
output=0x0000000040ec7000, count=1024) at cl_program_00309.cl:8
   5 const unsigned int count)
   6 {
   7 int i = get_global_id(0);
-> 8 if(i < count)
   9 output[i] = input[i] * input[i];
   10 }
   11
OpenCL GPU Execution Mode:
      GPU Group (1, 0, 0), Item (66, 0, 0): Kernel 'getsquare'
      local_id = <66,0,0>
      group_id = <1,0,0>
      global_size = <1024,0,0>
      global_offset = <0,0,0>
      num_groups = <4,0,0>
(lldb) frame variable
(int) i = 322
(int*) input = 0x0000000040ec5000
(int*) output = 0x0000000040ec7000
(const unsigned int) count = 1024
```

The whole GPU work group is stopped. Selecting the next thread moves the work group, and it can be seen that the global id assigned to i was incremented.

```
(lldb)thread select 70
* thread #70: tid = 0x7000043, 0x00000002
libOpenCL.so`getsquare(input=0x0000000040ec5000, output=0x0000000040ec7000,
count=1024) at cl_program_00309.cl:8, stop reason = trace
    frame #0: 0x00000002 libOpenCL.so`getsquare(input=0x0000000040ec5000,
output=0x0000000040ec7000, count=1024) at cl_program_00309.cl:8
   5 const unsigned int count)
   6 {
   7 int i = get_global_id(0);
-> 8 if(i < count)
   9 output[i] = input[i] * input[i];
   10 }
   11
(lldb) frame variable
(int) i = 323
(int*) input = 0x0000000040ec5000
(int*) output = 0x0000000040ec7000
(const unsigned int) count = 1024
```

B.2.4 Listing of OpenCL constructs

Below are examples for the LLDB OpenCL commands to display various constructs.

```
(lldb) opencl breakpoint work-item set -k SimpleIndexing -i 88
Added a breakpoint that will stop execution before work-item <88,0,0> is
processed by kernel 'SimpleIndexing'.
(lldb) opencl breakpoint work-item list -k SimpleIndexing
Kernel ‘SimpleIndexing’ work-item breakpoints:
   0: <42,0,0>
   1: <88,0,0>

(lldb) opencl breakpoint work-item delete -k SimpleIndexing -i 42
Remove breakpoint on work-item <42,0,0> of kernel ‘SimpleIndexing’
(lldb) opencl breakpoint work-item list -k SimpleIndexing
Kernel ‘SimpleIndexing’ work-item breakpoints:
   0: <88,0,0>
```

Listing event breakpoints:

```
(lldb) opencl breakpoint event show
Global stop settings:
   Stop when ‘complete’: 0
   Stop when ‘running’: 0
   Stop when ‘submitted’: 0
   Stop when ‘queued’: 0
(lldb) opencl context
The following subcommands are supported:
    buffers -- Show a summary of all created OpenCL buffers in a context.
    image   -- A set of commands for operating on OpenCL images.
    list    -- Show a summary of all loaded OpenCL contexts in a process.
    select  -- Select the OpenCL context to use for other OpenCL commands.
(lldb) opencl context list
OpenCL Contexts:
   0: cl_context 0x41d6f600 – 4 programs (*)
   1 context entries
(lldb) opencl context images
OpenCL images for context 0x41d6f600:
   0 image entries
(lldb) opencl context buffers
OpenCL Buffers for context 0x41df600:
   0: cl_buffer 0x412c6000 – 4096 bytes
   1: cl_buffer 0x412c7000 – 4096 bytes
   2 buffer entries
(lldb) opencl kernel
The following subcommands are supported:
    list   -- Show a summary of all loaded OpenCL kernels in a process.
    status -- Show the status of current stopped OpenCL kernel.
For more help on any particular subcommand, type 'help <command>
<subcommand>'.
(lldb) opencl kernel list
OpenCL Kernels:
cl_program 0x41d6fb08
   0: SimpleIndexing
   1 kernel entries
(lldb) opencl kernel status
State for kernel ‘SimpleIndexing’, N-D range 43054f8, chunk 0
   global_id    = <88,0,0>
   global_size  = <1024,1,1>

(lldb) opencl program
The following subcommands are supported:
    list   -- Show a summary of all loaded OpenCL programs in a context.
    source -- Show the source compiled for this OpenCL program.
For more help on any particular subcommand, type 'help <command>
<subcommand>'.
(lldb) opencl program list
OpenCL Programs for context 0x41d6f600:
   0: cl_program 0x41d6fb08 – 2 kernels
   1 program entries
```

B.2.5 Viewing local variables

Below are the tests for checking local variables using the frame variable command.

```
(lldb) frame variable
(int) i = 0
(int*) input = 0x00000000b625c000
(int*) output = 0x00000000b625a000
(const unsigned int) count = 1024


(lldb) memory read 0xb625c000
0xb625c000: 12 00 00 00 57 00 00 00 5b 00 00 00 17 00 00 00
....W...[.......
0xb625c010: 11 00 00 00 41 00 00 00 0e 00 00 00 61 00 00 00
....A.......a...
```

B.2.6 Reading and writing memory

Below are examples of reading and writing memory location.

```
(lldb) memory read 0xb625c000
0xb625c000: 12 00 00 00 57 00 00 00 5b 00 00 00 17 00 00 00
....W...[.......
0xb625c010: 11 00 00 00 41 00 00 00 0e 00 00 00 61 00 00 00
....A.......a...
 (lldb) memory write 0xb625c000 ff
0xb625c000: ff 00 00 00 57 00 00 00 5b 00 00 00 17 00 00 00
....W...[.......
0xb625c010: 11 00 00 00 41 00 00 00 0e 00 00 00 61 00 00 00
....A.......a...
```

B.2.7 Reading and writing registers

The register command can display and modify register values.

```
(lldb) register read
General Purpose Registers:
        r0 = 0x00000001
        r1 = 0xbeddca14
        r2 = 0xbeddca1c
        r3 = 0x000025be
        r4 = 0x400f79b5  hello-cl-cpu`main + 1 at hello.c:87
        r5 = 0xbeddca14
        r6 = 0x00000001
        r7 = 0xbeddca1c
```

```
              r8 = 0x00000000
              r9 = 0x00000000
             r10 = 0x00000000
             r11 = 0xbeddca0c
             r12 = 0x401d281c
              sp = 0xbeddc9e0
              lr = 0x40329741  libc.so`__libc_init + 41
              pc = 0x400f79b6  hello-cl-cpu`main + 2 at hello.c:126
            cpsr = 0x60000030
(lldb) register write r0 0xff
(lldb) register read
General Purpose Registers:
              r0 = 0x000000ff
              r1 = 0xbeddca14
          …
```