Welcome to CodeXL!

CodeXL is a comprehensive tool suite that allows developers to harness the benefits of AMD CPUs, GPUs, and APUs. Its capabilities include:

- Powerful Host and GPU debugging
- Comprehensive GPU, CPU and Power profiling
- DirectX® 12 Graphics Frame Analysis
- Static OpenCL™ kernel analysis capabilities
- Static OpenGL®, Vulkan® and DirectX® shader analysis capabilities

This tool suite enhances accessibility and lets software developers take maximum advantage of heterogeneous computing. CodeXL is available as a Visual Studio® extension, as well as a stand-alone user interface application for Windows® operating system and Linux® operating system.

This document is intended for software developers. The chapters on GPU Debugging and Profiling assume a working knowledge of OpenCL and/or OpenGL. The chapter on CPU Profiling assumes an understanding of the concepts of threads and processes, as well as familiarity with CPU architecture. The chapter on Kernel Analysis assumes knowledge of OpenCL and GPU architecture.

Latest Version of This Document

For the latest and greatest version of the documentation, go to the GPUOpen Website.
Contact Information

GPUOpen Website
CodeXL GitHub Issues Page
Getting Started

This section contains information about system requirements, installation, known issues, and support.

- System Requirements
- Installation Instructions
- Using CodeXL From a Remote Station
- Known Issues
- Support
System Requirements

Operating Systems

- Microsoft Windows 7-64 bit
- Microsoft Windows 8.1-64 bit
- Microsoft Windows 10 64-bit
- Linux 64-bit (Red Hat, Ubuntu, SUSE)

For detailed system requirements see the CodeXL Release Notes in the CodeXL installation folder or on the Documentation section of the CodeXL web page.

CodeXL Visual Studio Extension


Profiling OpenCL™ Applications

- [GPU device] AMD Catalyst driver with OpenCL™ GPU support
- [GPU device] AMD Radeon™ HD 5000 series or newer
- AMD APP SDK (requirements)
Power Profiling

- Kaveri, Mullins, Temash or Carrizo APU

For detailed system requirements see the CodeXL Release Notes in the CodeXL installation folder or on the Documentation section of the CodeXL web page.
Installation Instructions

1. Ensure you have the required components specified in the System Requirements.
2. Install CodeXL using one of the following methods:

   **Windows**
   - Run CodeXL_Win_x.x.x.exe

   **Linux - plain tar**
   - Extract CodeXL tarball using: `tar -xvzf CodeXL_Linux_x86_64_x.x.x.tar.gz`

   On Linux systems, the CodeXL Debian and RPM packages perform the driver installation automatically. However, if you’ve downloaded the CodeXL tar archive, you have to install the Power Profiler’s Linux driver manually. This includes a simple step of running `<codexl-install-dir>/AMDTPwrProfDriverInstall.run` with su credentials. Example:
   ```
   sudo ./AMDTPwrProfDriverInstall.run
   ```

   **Red Hat Linux (including CentOS, Fedora)**
   - Install the CodeXL RPM package using: `rpm -Uvh CodeXL_Linux.x.x.x-0.x86_64.rpm`

   **Ubuntu and other Debian based Linux distributions**
   - Install the CodeXL Debian package using:
     ```
     sudo dpkg -i codexl_x.x.x_64.deb
     ```
     followed by
     ```
     sudo apt-get -f install
     ```

**Confirm Installation**
Assuming you used the default install location, confirm the CodeXL binaries installation with the following steps.

**Windows**

- The C:\Program Files (C:\Program Files (x86) on 64-bit machines) folder should have a new sub-folder named “CodeXL” (the full path of the CodeXL folder should be: C:\Program Files\CodeXL, or C:\Program Files (x86)\CodeXL on 64-bit machines)
- A CodeXL shortcut has been created (unless you chose not to have shortcut).
- The Control Panel shows CodeXL in the list of installed Programs.

**Linux using tarball**

1. Navigate to `<CodeXL Directory>`
2. Launch CodeXL using ./CodeXL.
   A GUI appears.

**Linux RPM / Debian**

1. Navigate to /opt/AMD/CodeXL_X.Y-ZZZZ/
2. Launch CodeXL using ./CodeXL.
   A GUI appears.

**Confirm Visual Studio Extension**

For Windows, to confirm that the Visual Studio Extension has been installed successfully:

2. Click on Help > About Microsoft Visual Studio from the Visual Studio main menu bar.
3. Under Installed products:, find CodeXL x.x.
Using CodeXL From a Remote Station

CodeXL can be used remotely using the following methods:

1. Desktop sharing
2. CodeXL Remote Agent

These methods are detailed below.

Desktop Sharing

Run CodeXL graphic client application on the target platform via desktop sharing such as Windows Remote Desktop, VNC, SSH and X forwarding, etc.

Limitations:

1. OpenGL applications that are run for debugging/profiling may not recognize the shared desktop as supporting their GL requirements. If this happens run CodeXL locally on the target platform without desktop sharing or use the CodeXL Remote Agent (see below).
2. When CodeXL is run remotely using Linux SSH access, CodeXL requires X streaming such as provided by applications like Xming, etc.

CodeXL Remote Agent

Run the CodeXL Remote Agent on the target platform, and run the CodeXL graphic client application on a local station. For more details see Remote GPU Profiling, Power Profiling and GPU Debugging.
Known Issues

Check known issues in the CodeXL Release Notes that are found in CodeXL installation directory, and at the CodeXL Forum.
Support

- CodeXL Project in github
- OpenCL Zone
- AMD Accelerated Parallel Processing OpenCL Programming Guide


© 2015 Advanced Micro Devices. All rights reserved.
Using CodeXL

CodeXL contains the following modules:

- General GUI Controls
- Frame Analysis
- GPU Debugger
- CPU Profiler
- GPU Profiler
- Static Analyzer
- Power Profiler
- Remote Frame Analysis, GPU Profiling, Power Profiling and GPU Debugging

© 2015 Advanced Micro Devices. All rights reserved.
General GUI Controls

Several CodeXL views and controls are used by the debugger, the Profiler, and the static analyzer.

- CodeXL Welcome Page
- Getting Started Dialog
- Project Settings
- Execution Toolbar
- Properties View
- CodeXL Explorer
- Global Settings dialog
- System Information Dialog
- CodeXL Search Toolbar
CodeXL Welcome Page

CodeXL Welcome Page lets you access or create projects. It also contains links to common tasks you can be interested in.

The Welcome Page is opened in CodeXL startup. To access the Welcome Page, on the View menu, click Start Page.

The Welcome Page is divided into three main sections:

**Modes menu:** Browse CodeXL modes, and select a task for quick project creation:

<p>| <strong>Create New Project for Debugging</strong> | Use this option to create a new CodeXL project. After choosing this option, CodeXL will select “Debug Mode” in the execution mode toolbar, and will prepare the project for a debugging session. |
| <strong>Create New Project for Profiling</strong> | Use this option to create a new CodeXL project. After choosing this option, CodeXL will select “Profile Mode” in the execution mode toolbar, and will prepare the project for a profiling session. |
| <strong>Attach to Process</strong> | Select this option when you want to perform a CPU profiling session on an existing process. |
| <strong>System-wide Profiling</strong> | Use this option when you want to perform a CPU or power system-wide profiling session on the currently running processes on your machine. |
| <strong>Create a new</strong> | Selecting this option will create an empty new OpenCL kernel file, with a default name. After editing the new |</p>
<table>
<thead>
<tr>
<th><strong>OpenCL file for Analysis</strong></th>
<th>create kernel name, CodeXL will allow you to paste code or edit the kernel code and then build and analyze this code on selected devices.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add an existing OpenCL file for Analysis</strong></td>
<td>Select this to add an existing OpenCL kernel file and use CodeXL to build and analyze the kernel code on selected devices.</td>
</tr>
</tbody>
</table>

**Recent projects:** The list displays projects you have worked on recently. Clicking on one of the links will open the project for work.

**Samples:** Click on CodeXL Teapot sample link, to open the sample project. The sample can be used for getting to know CodeXL capabilities.
Another way to get start with CodeXL is using the startup dialog. Click the “Start” button (see the following screenshot). A dialog specifying the different options for getting started with CodeXL opens.
Create New Project

Use this option to create a new CodeXL project. The project settings dialog will open and guide you through the steps to create a project with your desired settings.

Create New Option

Use this option to create a new CodeXL project. After choosing this option,
<table>
<thead>
<tr>
<th><strong>Project for Debugging</strong></th>
<th>CodeXL will select “Debug Mode” in the execution mode toolbar, and will prepare the project for a debugging session.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create New Project for Profiling</strong></td>
<td>Use this option to create a new CodeXL project. After choosing this option, CodeXL will select “Profile Mode” in the execution mode toolbar, and will prepare the project for a profiling session.</td>
</tr>
<tr>
<td><strong>Attach to Process</strong></td>
<td>Select this option when you want to perform a CPU profiling session on an existing process.</td>
</tr>
<tr>
<td><strong>System-wide Profiling</strong></td>
<td>Use this option when you want to perform a CPU or power system-wide profiling session on the currently running processes on your machine.</td>
</tr>
<tr>
<td><strong>Create a new OpenCL file for Analysis</strong></td>
<td>Selecting this option will create an empty new OpenCL kernel file, with a default name. After editing the new create kernel name, CodeXL will allow you to paste code or edit the kernel code and then build and analyze this code on selected devices.</td>
</tr>
<tr>
<td><strong>Add an existing OpenCL file for Analysis</strong></td>
<td>Select this to add an existing OpenCL kernel file and use CodeXL to build and analyze the kernel code on selected devices.</td>
</tr>
</tbody>
</table>
Project Settings

To start debugging or profiling an application:

1. Create a CodeXL project.
   A CodeXL project consists of general information for the debugged / profiled application, such as command-line arguments, environment variables. The project also configures debugging and profiling specific configurations.

2. Use the File > New Project menu to open the new project dialog.
   The same dialog can be used later to configure the project settings during the debugging / profiling. (Debug > Debug Settings and Profile > Profile Settings menus)
General Project Settings
<table>
<thead>
<tr>
<th><strong>CodeXL Project Name</strong></th>
<th>The name of the current project used to identify a project in the Explorer views.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target Host</strong></td>
<td>The host on which the executable file will be debugged or profiled (Local / Remote).</td>
</tr>
<tr>
<td><strong>Remote Host Address</strong></td>
<td>The remote host address when the “Remote Host” option is selected.</td>
</tr>
<tr>
<td><strong>Port</strong></td>
<td>The port number on the remote host when the “Remote Host” option is selected.</td>
</tr>
<tr>
<td><strong>Test Connection</strong></td>
<td>Click to test the current connection settings.</td>
</tr>
<tr>
<td><strong>Application Type</strong></td>
<td>The type of application that will be debugged or profiled. The application can be either a desktop application or a Windows Store application.</td>
</tr>
<tr>
<td><strong>Executable Path</strong></td>
<td>The path to the executable / Windows Store application to be debugged or profiled. Use the Browse button for quick selection.</td>
</tr>
<tr>
<td><strong>Working Directory</strong></td>
<td>The directory in which the executable is to be debugged or profiled. Use the Browse button for quick selection.</td>
</tr>
<tr>
<td><strong>Command Line Arguments</strong></td>
<td>Arguments to be passed to the executable.</td>
</tr>
<tr>
<td><strong>Environment Variables</strong></td>
<td>A list of environment variables and their values to be set in the environment for the executable.</td>
</tr>
<tr>
<td><strong>Source Files Directory</strong></td>
<td>List of directories with the source code for the CPU profiled applications or the OpenCL™ kernels. Use the Browse</td>
</tr>
</tbody>
</table>
Specific Project Settings

- GPU Debugging Settings
- CPU Profiling Settings
- GPU Profiling Settings
Execution Toolbar

Once a CodeXL project is started, the initial interface is displayed, as shown in the following screenshot.
**CodeXL Initial Layout when a project is loaded**

The CodeXL toolbar allows quick access to CodeXL views and main controls.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug Mode</td>
<td>Switch CodeXL to Debug mode. This is the default mode.</td>
</tr>
<tr>
<td>Profile Mode</td>
<td>Switch CodeXL to Profile mode. Use the Profile menu to switch between the various profile modes.</td>
</tr>
<tr>
<td>Frame Analysis Mode</td>
<td>Switch CodeXL to Frame Analysis mode.</td>
</tr>
<tr>
<td>Analyze Mode</td>
<td>Switch CodeXL to Analyze mode.</td>
</tr>
<tr>
<td>Start Debugging / Profiling</td>
<td>Start the startup project with CodeXL in the selected mode.</td>
</tr>
<tr>
<td>Pause / Stop Debugging / Profiling</td>
<td>Pause / Stop the startup project with CodeXL in the selected mode.</td>
</tr>
<tr>
<td>Debug Steps</td>
<td>API step / Draw Step / Frame Step / Step In / Step Over / Step Out the debugged application.</td>
</tr>
<tr>
<td>CodeXL Explorer</td>
<td>Show the CodeXL Explorer tree view.</td>
</tr>
<tr>
<td>Properties View</td>
<td>Show the Properties view.</td>
</tr>
<tr>
<td>Output View</td>
<td>Show the Output view</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Function Calls History View</td>
<td>Show the Function Calls History view.</td>
</tr>
<tr>
<td>Debugged Process Events View</td>
<td>Show the Debugged Process Events view.</td>
</tr>
<tr>
<td>Call Stack View</td>
<td>Show the Debugger Call Stack view.</td>
</tr>
<tr>
<td>Locals View</td>
<td>Show the Debugger Locals view.</td>
</tr>
<tr>
<td>Debugger Watch View</td>
<td>Show the Debugger Watch view.</td>
</tr>
<tr>
<td>Debugger OpenGL State Variables</td>
<td>Show the Debugger OpenGL State Variables view.</td>
</tr>
<tr>
<td>OpenCL Debugger Multiwatch View 1, 2, 3</td>
<td>Show the OpenCL Debugger Multiwatch views.</td>
</tr>
<tr>
<td>Debugger Breakpoints View</td>
<td>Show the Debugger Breakpoints view.</td>
</tr>
<tr>
<td>Debugger Memory View</td>
<td>Show the Debugger Memory view.</td>
</tr>
<tr>
<td>Debugger Statistics View</td>
<td>Show the Debugger Statistics view.</td>
</tr>
</tbody>
</table>
Properties View

The CodeXL properties view displays detailed information about items appearing in other CodeXL views.

OpenCL Image Properties

1. Debug an OpenCL sample.
2. Break after clCreateContext.
3. Click the OpenCL Context object in the CodeXL Explorer to see the properties view content.

As shown in the image, the properties view specifies the image type, the image handle, dimensions and format.

Clicking on the image thumbnail in the properties view will open an image view.

OpenCL Context Properties

1. Debug an OpenCL sample with images
2. Break the application.
3. Click on an OpenCL context in the CodeXL Explorer to
The properties of an OpenCL context contains the context run time properties.

By clicking the context platform or devices link, the System Information Dialog is displayed. This shows platform / devices details.

Profile Session

1. Set CodeXL execution mode to profile.
2. Click on one of the sessions in the CodeXL Explorer to see the properties view content.

Session properties view displays the session type, the session file path and working folder, and few of the session properties.
<table>
<thead>
<tr>
<th><strong>CPU: Profile Session</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Profile Type</strong></td>
</tr>
<tr>
<td><strong>Session Name</strong></td>
</tr>
<tr>
<td>Executable Path:</td>
</tr>
<tr>
<td>Working Directory:</td>
</tr>
<tr>
<td>Profile Scope:</td>
</tr>
<tr>
<td><strong>Profile Start Time</strong></td>
</tr>
<tr>
<td><strong>Profile End Time</strong></td>
</tr>
<tr>
<td>CPU Details:</td>
</tr>
<tr>
<td>Profile Duration:</td>
</tr>
<tr>
<td>CPU Affinity:</td>
</tr>
<tr>
<td>Call Stack Information:</td>
</tr>
</tbody>
</table>
The CodeXL Explorer tree lets you navigate while debugging, profiling, or analyzing.

The tree structure is different for each of the execution modes. In Debug mode, the tree contains the debugged application details only when the debugged application is paused. Only one debug tree can be viewed while debugging. In Profile mode, on the other hand, the tree contains all the profiled sessions for the current project. You can view multiple sessions. When switching to Analyze mode, the tree will display all the OpenCL kernel files added to the current project, with build and analyze results for each of them.
OpenCL / OpenGL Debugged Objects Tree

While debugging, the CodeXL Explorer tree view lets you navigate and access all OpenCL and OpenGL objects during an application run. The Objects tree displays all the debugged application's allocated objects in the hierarchy: Application -> Context -> (Object Type) -> (Specific Object). Contexts marked with the sharing icon have their memory allocated objects shared by at least one other context. See cl_khr_gl_sharing in the OpenCL Specification, and wglShareLists in the OpenGL Specification, for more details about context resource sharing. Note that sharing contexts can have memory sizes even after being deleted, as long as there is at least one other context sharing their objects.
In Profile mode, the CodeXL Explorer lists the profiling sessions done for the current project; it indicates the type of profile for each session. When a CodeXL project is opened, the Profiler Session Explorer automatically displays all previous sessions associated with that project. Double-click on a session to open the session data view. Right-click on a session to rename it, delete it or to open the folder that contains the session output file. You can also import a previously-generated Profiler session, or delete all sessions using the context menu in the Profiling Session Explorer. A previously-generated Profiler session can be imported by dragging and dropping the session file from the system file browser to the Session Explorer.
CodeXL Analyze Tree

In Analyze mode, the tree displays 2 sub trees:

1. Tree of programs created by the user, with the output build results.
2. Tree of OpenCL / DirectX / OpenGL / Vulkan shader and kernel files added by the user and used while building the programs.
· Double clicking on a source file will open the source file in CodeXL
· Double clicking on an output build results will open a view that will show the source and IL and/or ISA if applicable for the specific device. The build results are grouped by kernel and then by device families.
· Double-clicking the “Statistics” and “Analysis” nodes will open the appropriate information view.
· You can drag a source file from the tree, in order to place it on a program.

The Tree also allows quick activation of the “Create new source file” and “Add existing source File” commands via double-clicking the two nodes at the bottom of the tree.

**Back / Forward Buttons**

The two tree navigation buttons are located in CodeXL Explorer top panel. Use these buttons to navigate to the previously viewed objects in the tree. The navigation history resets when opening another project or a debug session is terminated.
CodeXL Explorer context menu

Use the tree context menu to navigate and manage the current displayed project data.

For example, see the following context menus:

- Context menu on a CL buffer item when the debugged process is paused. The menu will contain the option to display the buffer in an MDI window and to view the buffer memory analysis and statistics.

- Context menu on a Time-Based CPU profile session. The menu will contain the options to open the session in an
MDI window, rename or delete the session, and open the folder that contains the session.

- Context menu on multiple sessions in profile mode. The menu will enable the deletion of multiple sessions.

- Context menu on source file. This menu enables one to open the source file, rename it, remove it from the CodeXL project, and open the containing folder.

- Context menu on a program. This menu enables one to run the build program command, to rename, remove or change the source of a program.
Global Settings

This dialog box lets you specify global settings for the CodeXL environment.

General Global Settings
| **Debug Log Level** | The debug log level is an internal mechanism that helps fix CodeXL problems. There are three levels.  
· Error: Log errors that occur while running CodeXL.  
· Info: Log errors and CodeXL internal information seen while running CodeXL (default level).  
· Debug: Log CodeXL debugging information, errors, and other internal information that is generated while running CodeXL. (This can be used by the CodeXL support team to locate a problem inside CodeXL).  
· Extensive: Log extensive CodeXL debugging information, errors, and other internal information that is generated while running CodeXL. (This can be used by the CodeXL support team to locate a problem inside CodeXL). |
<p>| <strong>Log Files and Images Directory</strong> | The directory in which the API call log files and temporary image and source files are saved. |
| <strong>I am using an HTTP proxy server</strong> | Check this option if your computer connects to the Internet through an HTTP proxy. If you are experiencing problems with CodeXL's online features (check for updates and send error report), changing the proxy settings might be the solution. |
| <strong>Proxy server / Port number</strong> | If you are using a proxy server, please input its information here. The server name could be a DNS address or an IP address. If you are unsure of what your HTTP proxy server or port are, contact your network administrator or copy the settings from your web browser. CodeXL does not currently support the |</p>
<table>
<thead>
<tr>
<th><strong>Remote Debugging Ports</strong></th>
<th>Specifies the port numbers that are being used for remote debugging sessions. On the client machine, all four ports should be able to receive incoming connections, and on the remote machine all four ports should not be blocked for outgoing connections.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Floating-point Precision</strong></td>
<td>The display precision for displaying floating-point numbers.</td>
</tr>
<tr>
<td><strong>Alert when a source file is missing</strong></td>
<td>Alert the user when a source file is not found during the profile session source code display.</td>
</tr>
<tr>
<td><strong>Restore Default Settings</strong></td>
<td>Click this button to restore all settings on all pages to their default values.</td>
</tr>
</tbody>
</table>
The system information dialog displays computer configuration details and OpenCL / OpenGL implementation details. These include operating system, memory, graphic card and driver details; available OpenCL platforms and devices; monitor details, available pixel formats, and available OpenGL extensions.

### System Information Tabs

<table>
<thead>
<tr>
<th>System</th>
<th>Collects and displays information about the system configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Collects and displays information about</td>
</tr>
<tr>
<td><strong>Display</strong></td>
<td>the display system configuration.</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td><strong>OpenGL Renderer</strong></td>
<td>Collects and displays information about the graphics accelerator configuration.</td>
</tr>
<tr>
<td><strong>OpenGL Pixel Formats</strong></td>
<td>Collects and displays information about the system’s supported pixel format.</td>
</tr>
<tr>
<td><strong>OpenGL Extensions</strong></td>
<td>Collects and displays information about the system’s supported OpenGL extensions.</td>
</tr>
<tr>
<td><strong>OpenCL Platforms</strong></td>
<td>Collects and displays information about the supported OpenCL platforms.</td>
</tr>
<tr>
<td><strong>OpenCL Devices</strong></td>
<td>Collects and displays information about the supported OpenCL devices and their capabilities and limits.</td>
</tr>
<tr>
<td><strong>Save Button</strong></td>
<td>Exports the system information data from all views to a .csv file.</td>
</tr>
</tbody>
</table>

**OpenCL Devices View**

The OpenCL Devices View displays the OpenCL devices available in the 32-bit and 64-bit OpenCL runtimes that are installed on the local station. The information is collected using the modules from the system path and may differ from runtimes installed elsewhere, such as OpenCL modules installed in the AMD APP SDK folder.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Device 1 (32-bit, GPU)</th>
<th>Device 2 (32-bit, CPU)</th>
<th>Device 1 (64-bit, GPU)</th>
<th>Device 2 (64-bit, CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform ID</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Device Type</td>
<td>GPU</td>
<td>CPU</td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td>Device Name</td>
<td>Hawaii</td>
<td>AMD FX(tm)-8150 Ei..</td>
<td>Hawaii</td>
<td>AMD FX(tm)-8150 Ei..</td>
</tr>
<tr>
<td>Vendor</td>
<td>Advanced Micro De..</td>
<td>AuthenticAMD</td>
<td>Advanced Micro De..</td>
<td>AuthenticAMD</td>
</tr>
<tr>
<td>Command Queue Properties</td>
<td>Queue profiling</td>
<td>Queue profiling</td>
<td>Queue profiling</td>
<td>Queue profiling</td>
</tr>
<tr>
<td>Max Global Variable Size</td>
<td>0 bytes</td>
<td>1,024 MB</td>
<td>2,592 MB</td>
<td>1,792 MB</td>
</tr>
<tr>
<td>Global Variable Preferred Total Size</td>
<td>0 bytes</td>
<td>1,024 MB</td>
<td>4,096 MB</td>
<td>1,792 MB</td>
</tr>
<tr>
<td>Queue On Device Properties</td>
<td>N/A</td>
<td>N/A</td>
<td>Out-of-order execut...</td>
<td>N/A</td>
</tr>
<tr>
<td>Queue On-Device Preferred Size</td>
<td>0 bytes</td>
<td>0 bytes</td>
<td>256 KB</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Queue On-Device Max Size</td>
<td>0 bytes</td>
<td>0 bytes</td>
<td>8 MB</td>
<td>0 bytes</td>
</tr>
</tbody>
</table>

**Description:**
OpenCL devices view collects and displays information about the OpenCL devices available on all OpenCL platforms installed on your system. This information is collected using the OpenCL modules in your system path. If the AMD APP SDK is installed it may provide more advanced device capabilities.
CodeXL Search Toolbar

Search toolbar appears when a view with searchable text is in focus and the Ctrl+F shortcut is used or the Find command is clicked in the Edit menu:
The user can navigate through a view’s text content using the Previous and Next buttons. Use the Match Case toggle button to control the case sensitivity of the search.
Frame Analysis

CodeXL Frame Analysis allows the developer to generate an API Trace with CPU side timing data and a GPU Trace with GPU side timing data. Both sets of data are displayed in a unified timeline which quickly allows the user to identify expensive GPU executions and the CPU side API calls which generate them. CodeXL allows you capture one or more frames from your running application and the captures are automatically saved to disk. The captured frames can be inspected offline using the CodeXL client (your application does not need to be running).

There are 4 stages to viewing the API and GPU trace data from your application.

1. Creating a New Project
2. Starting Your Application
3. Capturing Frames
4. Viewing Capture Data

© 2015 Advanced Micro Devices. All rights reserved.
Creating a New Project

The CodeXL project allows you to define your application executable, its working directory, and any command line arguments it requires. CodeXL will automatically save your project settings so the next time you want to analyze your application you simply load the project to get started.

1. Click “File -> New Project” to open the “Create a new CodeXL Project” dialog.
2. Browse to the executable you wish to analyze by clicking on the folder to the right of the “Executable Path:” data entry field.
3. The project will automatically be given the same name as the chosen executable. If you wish to use a different project name, enter a new name under the “CodeXL Project Name:” field after selecting the executable.
4. Enter any command line arguments, with which to run the executable, in the “Command Line Arguments:” data entry field.
Frame Analysis settings

1) In the ‘CodeXL Project Settings’ dialog or the ‘Create a new CodeXL Project’ dialog, click on the ‘Frame Analysis’ tree node to display the ‘Frame Analysis Settings’.
2) By default the “Automatically connect to first active API” option is turned on. When checked, this setting makes CodeXL automatically connect to the first DX12 application it finds when launching an analysis session. There are a couple circumstances where this behavior is undesired:
   a. If the executable launches multiple DX12 processes and you want to be able to choose which one to connect to.
   b. If you have trouble connecting to the DX12 application because CodeXL times out before the program has fully launched.

NOTE: If you have auto-connect disabled, the API selection window will remain open until you select an API context to connect to. Frame capture cannot begin until a connection has been made to a DX12 application.
Starting Your Application

1) Switch to Frame Analysis Mode either by clicking the Frame Analysis button in the toolbar, or by selecting ‘Frame Analysis’ -> ‘Switch to Frame Analysis Mode’ from the menu bar.

2) Begin a frame capture session either by:
   a. Clicking the Play button in the toolbar, or
   b. Selecting ‘Frame Analysis’ -> ‘Start Frame Analysis’ from the menu bar, or
   c. Pressing F5.
3) At this point, your application will launch and a selection box will appear with a list of all the API contexts found so far.

The above image shows the connection dialog with auto-connect disabled. Auto connect will be enabled by default.

Once a connection is established to a DX12 application, focus will switch from the launched program to CodeXL, and you will see a central preview image of the frames being rendered.

© 2015 Advanced Micro Devices. All rights reserved.
Capturing Frames

1) Once a frame capture session is running, press the Capture button 📸 either underneath the central preview image or in the toolbar to capture a frame. Once a frame capture has been initiated, it may take a few seconds to complete. The capture is complete when the capture button returns to an enabled state and a small preview image of the captured frame appears in the margin on the right hand side of the window.

2) Continue to capture as many frames as desired, waiting for a previous capture to finish before initiating a new one. The right hand margin will provide a running count of the captured frames, as well as a small preview image and basic data for each.

3) When stopping the capture session, press the Stop button ⏹️ either underneath the central preview image or in the toolbar.
Running Frame

Frame #4,147 Elapsed: 01:11 59 FPS

Captured Frames (3 Frames)

Frame #116
02/06/16
Duration: 00:00:01
FPS: 15
AP Calls: 3,329
Draw Calls: 2,053

Frame #178
02/06/16
Duration: 00:00:01
FPS: 15
AP Calls: 3,329
Draw Calls: 2,053

Frame #308
02/06/16
Duration: 00:00:01
FPS: 15
AP Calls: 3,329
Draw Calls: 2,053

© 2015 Advanced Micro Devices. All rights reserved.
Viewing Capture Data

1) After stopping a capture session, the captured frames will appear in the left hand pane CodeXL Explorer Tree under “(Project name here) | Frame Analysis Mode -> Frame Analysis”.
2) Expand the desired capture session node (capture sessions are named with the date and time of capture).
3) Select the desired frame sub-node. There should be a Timeline and Image item for each frame.
4) Double click the Image item to bring up a full resolution image of the captured frame.
5) Double click the timeline item to bring up GPU and CPU trace data.
The timeline view displays the collected frame data. See:

- The Frame Timeline View
- Navigating the Frame Timeline View
The Frame Timeline View
The frame timeline view contains the following elements (from bottom to top). Each of the elements’ numbers is in the screenshot above.

1. A navigation chart that allows you to zoom in and out, view the whole frame timeline or focus on a fragment of it, and display API calls duration, count and concurrency.
2. A collapsed detailed view of the API calls and their durations/count/concurrency.
3. A timeline chart visualizing the API calls over the frame timeline.
4. An API calls table for each of the CPU threads.
5. A commands table for each of the GPU command queues.
6. Summary tables that summarize the time consumption for each of the API types in the frame.
7. A Top 20 calls table for the currently selected API in the summary table.
View and analyze data in API Summary tables

The bottom pane of the trace view contain hotspot Summary tables. The table are summarizing the API call, and display each of the API types significant times. These tables can be used for quick identification of performance bottlenecks in the analyzed frame.

The table below specified each of the API table columns.

<table>
<thead>
<tr>
<th>CPU \ GPU properties:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td><strong>Call</strong></td>
</tr>
<tr>
<td><strong>Max Time</strong></td>
</tr>
<tr>
<td><strong>Min Time</strong></td>
</tr>
<tr>
<td><strong>Average Time</strong></td>
</tr>
<tr>
<td><strong>Cumulative Time</strong></td>
</tr>
<tr>
<td><strong>% of total</strong></td>
</tr>
<tr>
<td>Time</td>
</tr>
<tr>
<td>------</td>
</tr>
</tbody>
</table>

CodeXL 2.1 supports both DX12 and Vulkan API. The hotspot summary table shows DX12 Command list \\ Vulkan command buffer properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command List\Buffer</td>
<td>The command list \ buffer name</td>
</tr>
<tr>
<td>Execution Time</td>
<td>The duration for this command.</td>
</tr>
<tr>
<td>Start Time</td>
<td>The start time of this command list \ buffer. Clicking on the link in this column will select the specific command list \ buffer.</td>
</tr>
<tr>
<td>End Time</td>
<td>The end time of this command list \ buffer. Clicking on the link in this column will select the specific command list \ buffer.</td>
</tr>
<tr>
<td># of commands</td>
<td>The number of commands in this command list</td>
</tr>
<tr>
<td>GPU Queue</td>
<td>The GPU queue associated with this command list</td>
</tr>
<tr>
<td>Handle</td>
<td>The address of this command list \ buffer.</td>
</tr>
</tbody>
</table>

- Use timeline selection scope: when checked, the summary tables will reflect the selected timeline frame (which is painted within the red boundaries above). When un-checked, the summary tables will display the whole frame time range.

Selection of a line in the API summary table, will fill the top 20 calls, sorted by the time.
The screenshot below shows the API calls table, with “DrawIndexedInstanced” selected. The Top 20 table will display the top 20 time consuming calls to DrawIndexedInstanced.

<table>
<thead>
<tr>
<th>Name</th>
<th>The number of the call (1 for the top time consuming)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Id</td>
<td>The thread on which the call was executed</td>
</tr>
<tr>
<td>Index</td>
<td>The index within the thread</td>
</tr>
<tr>
<td>Time</td>
<td>The call time in ms. Clicking on the link in this</td>
</tr>
</tbody>
</table>
View data in CPU API Tables

The CPU and GPU API table monitor each of the frames API calls. The table below specified each of the API table columns.

<table>
<thead>
<tr>
<th>Index</th>
<th>Interface</th>
<th>Call</th>
<th>Parameters</th>
<th>CPU Time</th>
<th>Result</th>
<th>Start Time</th>
<th>End Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ID3D12GraphicsComm</td>
<td>DrawIndexedInstanced</td>
<td>1519, 1, 0, 0</td>
<td>4.406 ms</td>
<td>void</td>
<td>5.233 ms</td>
<td>5.289 ms</td>
</tr>
<tr>
<td>12</td>
<td>ID3D12GraphicsComm</td>
<td>DrawIndexedInstanced</td>
<td>300, 1, 15936, 6225, 9</td>
<td>978 ms</td>
<td>void</td>
<td>5.923 ms</td>
<td>5.936 ms</td>
</tr>
<tr>
<td>14</td>
<td>ID3D12GraphicsComm</td>
<td>DrawIndexedInstanced</td>
<td>488, 1, 16974, 6460, 9</td>
<td>978 ms</td>
<td>void</td>
<td>5.239 ms</td>
<td>5.240 ms</td>
</tr>
</tbody>
</table>

* Notice: BeginEvent, EndEvent, and SetMarker API calls are
marked in yellow for easy navigation.
The CPU and GPU API table monitor each of the frames API calls. The table below specified each of the API table columns.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>The API call index for this thread.</td>
</tr>
<tr>
<td>Cmd List #</td>
<td>The index of the command list \ buffer.</td>
</tr>
<tr>
<td>Call</td>
<td>The API call name.</td>
</tr>
<tr>
<td>Parameters</td>
<td>A string describing the parameters for this call</td>
</tr>
<tr>
<td>GPU Time</td>
<td>The duration of this API call on the GPU</td>
</tr>
<tr>
<td>Result</td>
<td>The API return value</td>
</tr>
<tr>
<td>Start Time</td>
<td>The time (from the frame start time in ms) the API call started</td>
</tr>
<tr>
<td>End Time</td>
<td>The time (from the frame start time in ms) the API call ended</td>
</tr>
</tbody>
</table>

© 2015 Advanced Micro Devices. All rights reserved.
Navigating the Frame Timeline View

A graphics frame can be very busy and contains tens of thousands of API calls. The following buttons and UI elements are useful in navigating the frame timeline and highlighting API calls of interest:

**Focused Timeline Fragment**

Use the left and right handles to expand/reduce the focused timeline fragment in and out:

**Navigation Bar Visualizations**

The navigation bar drop-list contains 3 visualization aids:

- Duration
- Count
- Concurrency

**API Calls Duration**

Select “Duration” in the top left combo box, to view the duration of API calls:
API Calls Count

Select “Count” in the top left combo box, to view the API calls count for each time fragment:

API Calls Concurrency

Select “Concurrency” in the top left combo box, to view the max / average busy threads concurrency over the frame timeline:
Timeline chart and API Calls tables

Double click a CPU API table item, and the timeline chart will zoom to the corresponding timeline item and highlight it. If there is a linked GPU API item, it will also be highlighted in the GPU API table.
The bottom ribbon of the timeline view contains the Hotspot Summary tables. These tables display aggregated data for API calls and command lists \ buffers, showing statistics for each type of API and the top individual calls to that API type.

The Max Time and Min Time columns display the execution
time of the longest and shortest API call of the selected API type. These are also direct links – clicking them causes the API tables and timeline chart to display the individual call item.

The ‘Top 20’ table is automatically populated with the 20 longest calls of the API type selected in the API summary table.

Performance tip: The longest GPU command in the frame, is always the first API call in the Top 20 table when the timeline view is opened.

For command lists \ buffers the hotspot table displays the command lists \ buffers which were executed during the captured frame.

The Execution Time column displays the execution time of the longest and shortest command lists \ buffer. The start and end time are also direct links – clicking them causes the timeline chart to display the individual command list \ buffer.

The ‘Top 20’ table is automatically populated with the 20 longest GPU call which belongs to the command list \ buffer which is selected in the hotspot summary table.

Performance tip: The longest GPU command in the frame, is always the first API call in the Top 20 table when the timeline view is opened.
The CodeXL GPU Debugging module traces application activity that makes use of OpenCL and OpenGL to provide application behavior information necessary for finding bugs and optimizing application performance.

With CodeXL, you can look inside your OpenCL and OpenGL API usage to see the effect individual commands have on application behavior.

CodeXL also lets you debug your OpenCL kernels at runtime, inspect variable values across different work item and work groups, inspect the kernel call stack, and more.

There are different ways to use the analytic capabilities of the CodeXL GPU Debugging Module: from locating bugs through removing redundant calls and errors to performing regression tests.

Whether you want to shorten debugging time, improve application quality, or optimize application performance, CodeXL displays the information you want.

The GPU debugging controls include:

- **Toolbars**
- **Views**
- **Dialogs**

© 2015 Advanced Micro Devices. All rights reserved.
GPU Debugging Toolbars

There are two types of GPU debugging toolbars:

- Images and Buffers
- Current Work Item
The Images and Buffers toolbar provides control over the CodeXL **object views**.

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Select/pan" /></td>
<td><strong>Select / pan</strong> Change the mouse click operation from selecting pixels to panning the image view.</td>
</tr>
<tr>
<td><img src="image" alt="Zoom controls" /></td>
<td><strong>Zoom controls</strong> Control the zoom level of the image view.</td>
</tr>
<tr>
<td><img src="image" alt="Rotation controls" /></td>
<td><strong>Rotation controls</strong> Rotate the image and data views clockwise or counterclockwise.</td>
</tr>
<tr>
<td><img src="image" alt="Channel Selection controls" /></td>
<td><strong>Channel Selection controls</strong> For multi-channeled images and textures, select which of the RGBA channels to show in the image and data views.</td>
</tr>
<tr>
<td><img src="image" alt="Invert" /></td>
<td><strong>Invert</strong> Invert the image view's displayed colors.</td>
</tr>
<tr>
<td><img src="image" alt="Grayscale" /></td>
<td><strong>Grayscale</strong> Desaturate the image view's displayed image.</td>
</tr>
</tbody>
</table>
Current Work Item toolbar

The Current Work Item toolbar is shown during debugging. Use this toolbar to select the active thread.

When N-Dimensional kernel debugging is taking place (clEnqueueNDRangeKernel), use this toolbar to select the current work item.

- When looking at variables (watch view, locals view, hovering over the variable name), the values shown will be those relevant to this work item.
- When stepping through code, code locations where this work item is not valid will be skipped.
- For coordinates that have more than 8 work items, the last item in the drop down list will show the range of available work items (in the above example, there are 1048576 work items, with indexes ranging between 0 and 1048575).
- To reach a work item whose index is larger than 7, you need to type the index manually and press enter (or switch to another combo box). For example, if you want the X coordinate to be set on the index is 8192, you should type the string 8192 in the X combo box and press enter:
Thread / Wavefront Selection Combo-box

This combo-box contains a list of the host threads currently active in the debugged application. Selecting a thread shows its call stack in the Call Stack view.

During kernel debugging, the active wavefronts are also shown in this combo-box. Selecting a wavefront displays its kernel source call stack.

Note that when debugging with breakpoint emulation, all work-items are gathered in a single virtual wavefront, which does not represent the actual work-item distribution on the hardware.

X, Y, Z Combo-boxes

These combo-boxes are filled with valid values for the work-item coordinates (based on the global work offset and size). If the work dimension is too low, the combo-box is disabled (for example: Z in a 2-dimensional work).
GPU Debugging Views

There are six GPU debugging views.

- API Function Calls History
- Memory view
- OpenCL Multi-Watch
- Visual Studio native debugging views
- Statistics view
- Object views
API Function Calls History View

This view displays a log of OpenCL, OpenGL, OpenGL extensions, as well as WGL and glX function calls executed in each context.

Function Calls List

The function calls are displayed as a list, ordered by the time in which they were called. Each list line represents a single function call; it contains:

- the called function name,
- its arguments values, and
- the function type (as an icon).

The function types are marked as shown in the following screenshot.

The icons have the following meanings
<table>
<thead>
<tr>
<th>C</th>
<th>OpenCL function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>📖</td>
<td>OpenCL buffer and image function.</td>
</tr>
<tr>
<td>📖</td>
<td>OpenCL queue function.</td>
</tr>
<tr>
<td>G</td>
<td>OpenGL function.</td>
</tr>
<tr>
<td>🏆</td>
<td>OpenGL extension function.</td>
</tr>
<tr>
<td>W</td>
<td>A WGL function.</td>
</tr>
<tr>
<td>X</td>
<td>A GLX function.</td>
</tr>
<tr>
<td>📖</td>
<td>OpenGL program and shader function.</td>
</tr>
<tr>
<td>📖</td>
<td>OpenGL texture function.</td>
</tr>
<tr>
<td>📖</td>
<td>OpenGL buffer function.</td>
</tr>
<tr>
<td>📖</td>
<td>GL_string_marker_GREMEDY function.</td>
</tr>
</tbody>
</table>

**The Viewed Render Context**

The list title bar shows the viewed context and the number of functions executed in it. Use the CodeXL Debugging Objects Explorer tree view to change the displayed context.

**Breaked-On Function**

A yellow arrow indicates the function called when the debugged process was suspended. Note that the process is suspended by CodeXL before the suspending function is executed. This lets you use the Step (F10) command to observe the effect its execution has on the API and objects.

**Frame Terminators**

When frame terminators are specified in the project settings, then during Frame Terminator execution the list in the context is cleared, and the function calls count is reset to 0.
Displaying Function Call Properties

Selecting a list line presents the appropriate function call in the Properties view.

HTML Log File

You can save the OpenCL / OpenGL function calls history in an HTML log file using the Record button. After pressing the Record button, a log file is created for each active context. This log file contains the details of each API function call, program and shader sources and image files for image and texture objects (if enabled in the Options dialog). To stop recording, press the record button again. To view the log file gathered so far, press the Open Current HTML Log File button.
Memory view

This view gives you information about your compute and graphic memory consumption and usage, as well as detected graphic or compute memory leaks.

Graphic Object Details View

The Graphic Object Details view lists all the graphic memory allocated objects of the selected type in the appropriate context (for example, Context 1 textures). The columns in the list vary depending on the selected item type, displaying information relevant to the current items' memory size.
Object Creation Calls Stack View

This view displays the calls stack when the currently selected item was created. Select a stack frame to view its information. If your application has the appropriate debug information and source code, double-clicking a stack frame opens the Visual Studio Source Code editor, highlighting the appropriate line.

Graph View

This view contains details of the graphic memory consumed by the objects displayed in the Graphic Objects Details view.
OpenCL Multi-Watch Views

This view lets you compare the values of an OpenCL Kernel variable across work-items and work-groups.

**Variable Values**

The Multiwatch main window displays either a graphical (image view) or spreadsheet (data view) visualization of the selected kernel variable across the various work-items and work-groups. This view operates very similarly to the Object views. The two following screenshots display a multi watch view for the kernel variable sum.s2. The upper screenshot shows an image view, the lower shows a data view.
Variable Name Combo-box

Select a variable name from the list, or type a watch expression. If the expression can be parse and evaluated, the values are updated in the main view.

Kernel Work Geometry and Selected / Hovered Details

Selecting, or hovering with the cursor over, work-items displays
the kernel work geometry (local and global work size and offset) for the current N-dimensional kernel execution, as well as their location and value.

**Value Range Slider**

Shows the value range, and allows marking the valid value range. Values above and below these lines are colored to show they are not in the selected range.
Visual Studio Native Debugging Views

Many Visual Studio native views have information from the CodeXL GPU Debugging Engine.

Source Code Window

- During API debugging, CodeXL displays the C/C++ source that led to the API function call in the source code window.
- During OpenCL kernel debugging, CodeXL displays the kernel source code in the source code window. This is done from a project file, if available, or from a temporary file for applications that generate the kernel source at runtime.
- Double-clicking an OpenCL kernel or program, or an OpenGL shader in the CodeXL Debugging Object Explorer, displays its source code in the source code window.
- Double-clicking an object allocation call stack frame in the Memory view displays the source location associated with that stack frame in the source code window.

Breakpoints View

- CodeXL API function breakpoints are displayed as C/C++ function breakpoints.
- CodeXL kernel function name breakpoints are displayed as function breakpoints with the prefix Kernel:.
- CodeXL Error / Warning breakpoints appear as function breakpoints, with their descriptive string as the breakpoint.
- Kernel source code breakpoints are displayed as
breakpoints in the kernel source file. Note that if a temporary kernel source file is created at runtime, the breakpoints set in it do not work in future runs of the application. To associate a kernel with a source file, include the source file in the debugged project, with a .cl file extension.

Watch View

During OpenCL kernel debugging, enter variable names in the Watch view to see them update during debugging or when switching between work-items in the Work-Item toolbar.

Autos View

During OpenCL kernel debugging, the Autos view displays the values of variables near the program counter, if possible.

Locals View

During OpenCL kernel debugging, the Locals view displays all the available kernel variables in the current scope. An additional variable, KernelDispatchDetails, which indicates the kernel’s work size and the selected work item ID and workgroup, is also displayed.

Call Stack View

During API debugging, the current API function's call stack is displayed in the Call Stack view. During OpenCL kernel debugging, the kernel's call stack is displayed in the Call Stack view. Note that since the kernel is debugged at execution time, the call stack does not contain the clEnqueueNDRangeKernel or clEnqueueTask API function call that started the debugging.
Statistics View

This view lets you view statistical information about your OpenCL and OpenGL APIs usage.

Context Selection

Select an OpenCL or OpenGL context in the CodeXL Debugging Object Explorer to update the statistics to that context's information.

Graph View

This view contains a graphical representation of the information in the Function Types, Function Calls Statistics, State Change Statistics, Deprecated Function Statistics, and Vertex Batch Statistics views.
Properties Box

This box, located in the lower right-hand side of the Statistics view, displays the selected object properties. The properties include information about the object, as well as an explanation of any warnings (represented by exclamation point icons next to the items).

Statistics Tabs

Statistics tabs include:
- **Function Type Statistics** - Displays details of OpenCL / OpenGL function calls to categories. Note that a function can belong to multiple or none of the categories. To see which categories a function belongs to, find it in the Function Calls Statistics view.

- **Function Call Statistics** - Displays a breakdown of all the OpenCL and OpenGL functions used, as well as useful tips and information about unrecommended functions.

- **Deprecated Function Statistics** - Displays a breakdown of the usage of functions deprecated by any OpenCL or OpenGL version. In Analyze Mode, this view also displays details about usage of deprecated features in partially deprecated functions.

- **Calls History** - Displays the history of API calls executed for last frame on the current displayed context.

- **Vertex Batch Statistics** - Displays a breakdown of vertex drawing OpenGL function calls (or vertex batches) by the number of vertices drawn in each.
Function Type Statistics view

The Function Types Statistics view displays a breakdown of OpenCL / OpenGL function calls to categories.

Note that a function can belong to multiple or none of the categories. If you wish to know which categories does a function belong to, find it in the Function Calls Statistics view.
Graph and Properties Views

Graph View

When in the Function Types Statistics view, the Graph view displays a bar graph showing the part each function type takes. The grid lines represent 25, 50, and 75 percent of all function calls. Selecting a function type in the list highlights it in the graph. Redundant State Change functions (when available) are marked in red; the Get functions are marked in orange.
Properties Box

When in the Function Types Statistics view, the Properties box shows information about the currently selected function type.
Function Call Statistics View

The Function Calls Statistics view shows the number of times each OpenCL / OpenGL function call was executed in the previously rendered frame, as well as its percentage of the total functions execution.
Detailed Function Calls

For most of the functions, the calls to that function are displayed using a single entry regardless of the arguments passed to the function.

For the following functions, CodeXL provides a separation based on one of the arguments used in the function call to provide more precise information, i.e. they are listed by their enumerators.

- `glBegin`
Exporting Function Calls Statistics Data Into a file
The Function Calls Statistics data can be exported to a file (.csv) using the right-click context menu "Export Function Calls Statistics" command. Exporting the function calls statistics data can help you compare the function calls statistics of different frames. It also allows you to perform regression tests by comparing the function calls statistics data of two versions of your application.

Functions Not Recommended

The Usage of certain OpenCL and OpenGL functions is unrecommended, mostly for performance taxing reasons. These functions are noted as such in this view in varying degrees - mildly unrecommended (yellow warning sign), intermediately unrecommended (orange warning sign) and highly unrecommended (red warning sign). Click on an unrecommended function to display information about it and a better alternative to using it in the Properties view.

Graph and Properties Views

Graph View

When in the Function Calls Statistics view, the Statistics view Graph view will display a pie chart of the OpenCL / OpenGL function calls. Each pie "slice" is one API function (or in some cases, a combination of an OpenGL function and enumerator). Selecting a function or functions in the list causes their respective slice in the graph to be highlighted. The graph can be rotated by clicking on it and dragging.

Properties Box

When in the Function Calls Statistics view, the Statistics view Properties Box will display the function name. If the function is an unrecommended one, the Properties Box will also display an explanation of why using this function is unrecommended, as well as a recommendation for an alternative to using this
function.
Deprecation Function Statistics view

The Deprecation Function Statistics view displays information about your usage of functions deprecated by any OpenCL or OpenGL version.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Deprecation Reason</th>
<th># of Calls</th>
<th>% of Calls</th>
<th>Deprecated at Ver.</th>
<th>Removed at Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>glTexCoord2f</td>
<td>Completely Deprecated Func...</td>
<td>2,104</td>
<td>12.26</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glVertex3f</td>
<td>Completely Deprecated Func...</td>
<td>2,104</td>
<td>12.26</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glMatrixMode</td>
<td>Completely Deprecated Func...</td>
<td>1,509</td>
<td>0.20</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glLoadIdentity</td>
<td>Completely Deprecated Func...</td>
<td>1,053</td>
<td>6.14</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glBegin</td>
<td>Completely Deprecated Func...</td>
<td>526</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glEnd</td>
<td>Completely Deprecated Func...</td>
<td>526</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glOrtho</td>
<td>Completely Deprecated Func...</td>
<td>526</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glPopMatrix</td>
<td>Completely Deprecated Func...</td>
<td>526</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glPushMatrix</td>
<td>Completely Deprecated Func...</td>
<td>526</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glMultMatrix</td>
<td>Completely Deprecated Func...</td>
<td>1</td>
<td>0.01</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td><strong>Total (10 Items)</strong></td>
<td></td>
<td><strong>9,471</strong></td>
<td><strong>55.21</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**glLoadIdentity**

**Deprecated Feature:** Fixed管线Vertex Processing

The function `glLoadIdentity` was marked as deprecated in OpenGL version OpenGL 3.0 and was removed from OpenGL at version OpenGL 3.1.

**Warning:** It is recommended to avoid using deprecated functions, as their compatibility with future OpenGL versions is not assured.
Each of the lines in the list displays the usage of one Deprecated function or a combination of a partially deprecated function and a deprecated feature. The number of calls to this function or deprecated uses of this function and feature combination is displayed (as well as a percentage of the total API function calls). Finally, the OpenCL / OpenGL version when this feature was deprecated and the version it was removed (if any) are displayed.

**Graph and Properties Views**

**Graph View**

When in the Deprecated Function Statistics view, the Statistics view Graph view displays a pie chart, representing a breakdown of the deprecated function calls per combination of deprecated function and deprecated feature.

**Properties View**

When in the Deprecated Function Statistics view, the Statistics view Properties view displays the currently selected deprecated function's name, as well as information about the deprecated feature it is a part of, and other functions and behaviors belonging to the same deprecated feature. A forward-compatible alternative to this feature is supplied in bold font.
API Function Calls History View

This view displays a log of OpenCL, OpenGL, OpenGL extensions, as well as WGL and glX function calls executed in each context.

**Function Calls List**

The function calls are displayed as a list, ordered by the time in which they were called. Each list line represents a single function call; it contains:

- the called function name,
- its arguments values, and
- the function type (as an icon).

The function types are marked as shown in the following screenshot.
The icons have the following meanings

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>OpenCL function.</td>
</tr>
<tr>
<td>🏛️</td>
<td>OpenCL buffer and image function.</td>
</tr>
<tr>
<td>🗺️</td>
<td>OpenCL queue function.</td>
</tr>
<tr>
<td>G</td>
<td>OpenGL function.</td>
</tr>
<tr>
<td>🎨</td>
<td>OpenGL extension function.</td>
</tr>
<tr>
<td>🌐</td>
<td>A WGL function.</td>
</tr>
<tr>
<td>🗑️</td>
<td>A GLX function.</td>
</tr>
<tr>
<td>🎨</td>
<td>OpenGL program and shader function.</td>
</tr>
<tr>
<td>📖</td>
<td>OpenGL texture function.</td>
</tr>
<tr>
<td>OpenGL buffer function.</td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td></td>
</tr>
<tr>
<td>GL_string_marker_GREMEDY function.</td>
<td></td>
</tr>
</tbody>
</table>
The Vertex Batch Statistics view displays information about your usage of OpenGL vertex drawing functions or vertex batches, divided by the batch size (i.e. how many vertices were drawn with a single OpenGL function call).

### Statistics - GL Context 2

<table>
<thead>
<tr>
<th>Vertices per Batch</th>
<th># of Batches</th>
<th>% of Batches</th>
<th># of Vertices</th>
<th>% of Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15,368</td>
<td>26.57%</td>
<td>15,368</td>
<td>0.03%</td>
</tr>
<tr>
<td>6</td>
<td>123</td>
<td>0.20%</td>
<td>738</td>
<td>0.00%</td>
</tr>
<tr>
<td>24</td>
<td>8,061</td>
<td>13.13%</td>
<td>193,464</td>
<td>0.38%</td>
</tr>
<tr>
<td>72</td>
<td>4,052</td>
<td>6.67%</td>
<td>294,624</td>
<td>0.50%</td>
</tr>
<tr>
<td>1,536</td>
<td>32,736</td>
<td>53.33%</td>
<td>50,282,495</td>
<td>99.01%</td>
</tr>
<tr>
<td>Total</td>
<td>61,380</td>
<td>100%</td>
<td>50,787,590</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Warning:**

*72 Vertex Batch*

You draw 0.5801% of your vertices with 6.67% of your batches

- It is good practice to draw as many vertices using as few batches as possible. Doing so will usually improve the application’s rendering performance.
Each of the lines in the view displays statistics for a range of vertex batch sizes (normal view) or a specific batch size (detailed view). The number and percentage of batches of this size represents the "cost" of drawing with this batch size. The number and percentage of vertices represent the "benefit" gained from drawing with this batch size. Batches which are small in relation to the application (have a higher percentage of batches than is expected from their percentage of vertices) are noted with a warning icon.

**Show Detailed Batch Statistics**

Toggles between the Vertex Batch Statistics view's normal (gather batch sizes into ranges by amount) and detailed (show each vertex batch size separately) views.

**Graph and Properties Views**

**Graph View**

When in the Vertex Batch Statistics view, the Statistics view Graph view displays a bar chart, in which each bar is a range of batch sizes. The height of the bar is the number of batches drawn in this range, and the bar's color represents the range's position in the application's distribution (Red bars are the smallest batches and green bars are the largest ones). The grid lines represent 25, 50 and 75 percent of vertex drawing function calls. When the Vertex Batch Statistics shows the detailed view, this graph becomes a histogram of the vertex batch sizes.

**Properties View**

When in the Vertex Batch Statistics view, the Statistics view Properties view displays the currently selected batch size range, the percentage statistics for this range and a short explanation about vertex batches.
Object Views

The Object views - Image view and Data view - allow you to view all OpenCL buffers and images, and OpenGL textures, static buffers, VBOs (vertex buffer objects), FBOs (frame buffer objects), render buffers and pbuffers (pixel buffers).

Each object can be viewed both as an image in the Image view (except OpenCL Buffers and OpenGL VBOs) and as a spreadsheet containing the object raw data in the Data view. The object properties and parameters will appear in the properties view.

Image View

Data View

Information Panel

The information panel helps you make fast inquiries and adjustments to the currently viewed object pixels/texels. The information panel is composed of a few main elements (which are only shown when relevant):

Current and Previous Pixel

When hovering with the mouse over the Image view, the currently highlighted pixel data is displayed as the current pixel; Both RGBA values and raw data value are displayed. If you wish to compare between two different pixels, you can save a specific pixel value by clicking on it with the left mouse button. The pixel data will be stored as the "Selected Pixel", allowing you to make fast comparison to another pixel.
**Single Channel Range Adjustment Slider**

When viewing an object that has a single component data format ("Depth" / "Luminance" / "Intensity" / etc.) the image view translates these values to grayscale values. Use the single channel range adjustment slider to adjust the data translation to grayscale values.

**3D Layer Slider**

When viewing 3D images or textures, you can scroll through the 2D images that make up the 3D image using the 3D layer slider located in the information panel.

**Texture Array Layer Slider**

When viewing texture arrays, you can scroll through the
textures that make up the array using the Texture array layer slider located in the information panel.

**Texture Mipmap Slider**

When viewing a texture which has automatic or manual mipmaps defined, you can scroll through the different texture levels using the Mipmap slider located inside the information panel.

**OpenCL buffer / OpenGL VBO Format Controls**

When viewing an OpenCL buffer or an OpenGL VBO, the information panel will contain several items that allow choosing how the VBO data is displayed:

- Data combo-box - choose which kind of data the buffer contains. Each option in this combo-box matches either the OpenCL C types or an OpenGL gl*Pointer function ("Interleaved" matches glInterleavedArrays). Choose "All" to display all formats. An exception to this is the "Index" option which also matches the glDrawElements (and variants) function's indices parameter and not only glIndexPointer.
- Format combo-box - choose the format the data is stored in, matching the OpenCL C type the kernel uses (__global XXXX*) size and type parameters of the gl*Pointer function or the format parameter of glInterleavedArrays. The formats are named as data-components-format, for example C4UB stands for color data stored in 4 components of unsigned byte type.
- Offset spin control - choose the offset from the beginning of the VBO to the start of the data. This should match the pointer parameter of the gl*Pointer function.
- Stride spin control - choose the stride (space in bytes between the end of one vertex's data and the next one's) to match the stride parameter of the gl*Pointer function.
<table>
<thead>
<tr>
<th>i</th>
<th>i</th>
<th>i</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>81</td>
<td>92</td>
<td>113</td>
<td>255</td>
</tr>
<tr>
<td>113</td>
<td>173</td>
<td>210</td>
<td>510</td>
</tr>
<tr>
<td>173</td>
<td>260</td>
<td>308</td>
<td>765</td>
</tr>
<tr>
<td>242</td>
<td>350</td>
<td>415</td>
<td>1030</td>
</tr>
<tr>
<td>515</td>
<td>441</td>
<td>535</td>
<td>1275</td>
</tr>
<tr>
<td>365</td>
<td>536</td>
<td>659</td>
<td>1530</td>
</tr>
<tr>
<td>451</td>
<td>651</td>
<td>778</td>
<td>1785</td>
</tr>
<tr>
<td>506</td>
<td>719</td>
<td>851</td>
<td>2040</td>
</tr>
<tr>
<td>591</td>
<td>813</td>
<td>994</td>
<td>2295</td>
</tr>
<tr>
<td>669</td>
<td>869</td>
<td>1099</td>
<td>2550</td>
</tr>
<tr>
<td>729</td>
<td>994</td>
<td>1213</td>
<td>2805</td>
</tr>
<tr>
<td>830</td>
<td>1098</td>
<td>1336</td>
<td>3060</td>
</tr>
<tr>
<td>870</td>
<td>1106</td>
<td>1443</td>
<td>3315</td>
</tr>
<tr>
<td>959</td>
<td>1270</td>
<td>1650</td>
<td>3810</td>
</tr>
<tr>
<td>1042</td>
<td>1370</td>
<td>1681</td>
<td>3925</td>
</tr>
<tr>
<td>1099</td>
<td>1479</td>
<td>1704</td>
<td>4080</td>
</tr>
<tr>
<td>1182</td>
<td>1584</td>
<td>1901</td>
<td>4385</td>
</tr>
<tr>
<td>1269</td>
<td>1679</td>
<td>2070</td>
<td>4690</td>
</tr>
<tr>
<td>1359</td>
<td>1796</td>
<td>2134</td>
<td>4845</td>
</tr>
<tr>
<td>1419</td>
<td>1887</td>
<td>2366</td>
<td>5100</td>
</tr>
<tr>
<td>1500</td>
<td>1988</td>
<td>2381</td>
<td>5355</td>
</tr>
<tr>
<td>1560</td>
<td>2100</td>
<td>2510</td>
<td>5510</td>
</tr>
<tr>
<td>1655</td>
<td>2218</td>
<td>2654</td>
<td>5865</td>
</tr>
<tr>
<td>1770</td>
<td>2352</td>
<td>2787</td>
<td>6120</td>
</tr>
<tr>
<td>1874</td>
<td>2482</td>
<td>2944</td>
<td>6375</td>
</tr>
</tbody>
</table>
Object Image view

You can view an image, a buffer or a texture object loaded image using the Image view.
Images and Buffers toolbar
Use the Images and Buffers toolbar to control the Image view

Click on the data tab to see the image data
Object Data view

You can view a buffer, an image or a texture object raw data using the Data view.
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>92</td>
<td>113</td>
<td>255</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>113</td>
<td>373</td>
<td>210</td>
<td>510</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>273</td>
<td>608</td>
<td>308</td>
<td>765</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>242</td>
<td>390</td>
<td>415</td>
<td>1020</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>232</td>
<td>349</td>
<td>420</td>
<td>1020</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>272</td>
<td>363</td>
<td>449</td>
<td>1020</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>279</td>
<td>371</td>
<td>470</td>
<td>1020</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>266</td>
<td>360</td>
<td>466</td>
<td>1020</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>268</td>
<td>372</td>
<td>481</td>
<td>1020</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>261</td>
<td>363</td>
<td>440</td>
<td>1020</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>268</td>
<td>360</td>
<td>435</td>
<td>1020</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>295</td>
<td>379</td>
<td>455</td>
<td>1020</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>289</td>
<td>376</td>
<td>449</td>
<td>1020</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>293</td>
<td>377</td>
<td>451</td>
<td>1020</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>292</td>
<td>375</td>
<td>448</td>
<td>1020</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>296</td>
<td>381</td>
<td>448</td>
<td>1020</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>310</td>
<td>398</td>
<td>458</td>
<td>1020</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>314</td>
<td>403</td>
<td>460</td>
<td>1020</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>327</td>
<td>415</td>
<td>473</td>
<td>1020</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>320</td>
<td>408</td>
<td>472</td>
<td>1020</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>320</td>
<td>404</td>
<td>480</td>
<td>1020</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>317</td>
<td>404</td>
<td>486</td>
<td>1020</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>344</td>
<td>432</td>
<td>520</td>
<td>1020</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>351</td>
<td>448</td>
<td>531</td>
<td>1020</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>374</td>
<td>474</td>
<td>563</td>
<td>1020</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>391</td>
<td>486</td>
<td>580</td>
<td>1020</td>
<td>26</td>
</tr>
<tr>
<td>26</td>
<td>410</td>
<td>506</td>
<td>603</td>
<td>1020</td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td>432</td>
<td>521</td>
<td>623</td>
<td>1020</td>
<td>28</td>
</tr>
<tr>
<td>28</td>
<td>444</td>
<td>528</td>
<td>624</td>
<td>1020</td>
<td>29</td>
</tr>
<tr>
<td>29</td>
<td>442</td>
<td>540</td>
<td>626</td>
<td>1020</td>
<td>30</td>
</tr>
</tbody>
</table>
Grid Zoom

The grid zoom slider allows you to set the zoom level of the displayed raw data cells.

Link Image and Data Views

When enabled, clicking on a pixel at the image view will select this pixel at the data view grid.

Show Normalized Values

The values showed in the grid are the original data values of the object as held by OpenCL and OpenGL. These values may be in various data formats such as OpenCL C types, or GL_FLOAT, GL_INT, GL_SHORT and GL_BYTE. When enabled, the data view will display the values in the grid normalized to the GL_BYTE type ([0..255] range), regardless of the original data type.

Viewing OpenCL buffers and OpenGL Vertex Buffer Objects (VBOs)

These buffers can only be shown in the Data View. After choosing the VBO's data, format, offset and stride; each vertex's data will be shown as a separate line in the view. The column headers represent the meaning of each value - X, Y, Z, W for vertex position; Nx, Ny, Nz for normal direction; R, G, B, A for vertex color and S, T, R, Q for texture coordinates.
This section provides information on:

- Adding / removing breakpoints
- GPU debugging project settings
- GPU debugging global settings
Add / Remove CodeXL Breakpoints Dialog

The Breakpoint dialog lets you choose OpenCL and OpenGL API function breakpoints, as well as kernel function name breakpoints.

Breakpoints added this way are added as C/C++ function breakpoints to the Visual Studio Breakpoints view. Adding an OpenCL or OpenGL function as a Visual Studio breakpoint (Debug > Breakpoints > New Breakpoint...) adds it to this dialog.

The debugged process execution is suspended when a debugged process thread calls an API function marked as a breakpoint. The process run is suspended before the function is executed; this lets you observe the effect the breakpoint function has on the application behavior.

The debugged process execution is suspended when a kernel with a function name marked as a breakpoint starts executing. This lets you start kernel debugging without having to use the clEnqueueNDRangeKernel function call.
### Breakpoints

<table>
<thead>
<tr>
<th><strong>API Functions</strong></th>
<th>Contains OpenCL and OpenGL API functions supported by CodeXL.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel Functions</strong></td>
<td>When an OpenCL application is debugged, these are special breakpoints that can be set:</td>
</tr>
<tr>
<td></td>
<td>· Break on OpenGL error - This is hit when OpenGL encounters an error code. Note that if your application uses glGetError for validation, its flow might be interrupted.</td>
</tr>
<tr>
<td></td>
<td>· Break on OpenCL error – This is hit when OpenCL encounters an error.</td>
</tr>
</tbody>
</table>
| Error / Warning | error code.  
· Break on Detected error - The CodeXL specifications. This breakpoint is hit when a break on error is detected.  
· Break on Deprecated function - OpenGL marked as deprecated and must be removed. |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Filter</td>
<td>Enables filtering the Breakpoints list. Entries containing all the strings.</td>
</tr>
<tr>
<td>Active Breakpoints</td>
<td>Contains the currently selected API and checking / unchecking the box next to the breakpoint list.</td>
</tr>
<tr>
<td>Add Breakpoint</td>
<td>To add a breakpoint, select one or more APIs from the Breakpoints list by double-clicking on it, or by pressing the Remove Breakpoint button.</td>
</tr>
<tr>
<td>Remove Breakpoint</td>
<td>To remove a breakpoint, select the function and press the Remove button.</td>
</tr>
<tr>
<td>Remove all Breakpoints</td>
<td>Press this button to remove all of the active breakpoints.</td>
</tr>
</tbody>
</table>

**Settings**

**Enable / Disable all Function Breakpoints**

This section describes how to change the enabled status of all active breakpoints.
GPU Debugging Project settings

Use the File > Project Settings menu (Ctrl-P) to open the project settings dialog. Navigate to the “Debug” node in the project settings tree displayed in the left panel of the dialog.
OpenCL frame terminators are the functions that end your application computation
**OpenCL Frame Terminators**

Frame. They let you view the log of a single computation frame, not the entire calls log. Available Frame Terminators are: `cl_gremedy_computation_frame`, `clFlush`, `clFinish`, and `clWaitForEvents`.

| **OpenCL Frame Terminators** | OpenGL frame terminators are the functions that end your application render frame. They let you view the log of a single render frame, not the entire calls log. Available Frame Terminators are: `glClear`, `glFlush`, `glFinish`, `wglSwapBuffers`, `wglMakeCurrent`, `wglSwapLayerBuffers`, and `glFrameTerminatorGREMEDY`. For example:

```
  glFlush is usually chosen for single-buffered applications.
  wglSwapBuffers is usually chosen for double-buffered applications.
```

You must select at least one OpenGL Frame Terminator. See [Frames and Frame Terminators](#) for more details. |
| **Add / Remove Breakpoints** | Click this button to open the Add / Remove CodeXL Breakpoints dialog.

© 2015 Advanced Micro Devices. All rights reserved.
The GPU Debug global settings are set when the project is initially created, and affect every session.

To set the options:

1. In the CodeXL menu, click **CodeXL Options**. The Edit CodeXL Global Settings dialog box is displayed.
2. Select the GPU Debug tab. See the descriptions below for each of the global debug settings.
3. Click **OK** to activate the new options and close the dialog box, or click **Restore Default Settings** to reset the dialog box selections to the system default, or click **Cancel** to close without changes.
Call Stack

When specifying additional source code directories, if the source code file is not found at its debug information path, it will be searched, by file name only, in the additional source code directories. The search is performed according to the order in which the directories are specified. The additional source code directories are semicolon separated.

When specifying a source code root location,
the root location is prefixed to each source code file path. Only one source code root location can be specified. Example: If the source code root is "D:\Dir1\" and the debug information source code path is "\Dir2\Dir3\MyFile.cpp", the Source Code editor input path will be "D:\Dir1\Dir2\Dir3\MyFile.cpp".

Deselect this checkbox if you don't want CodeXL to collect the graphic memory allocated objects creation calls stacks (which are displayed in the Memory View). This can improve Debug Mode performance in some cases when many graphic memory objects are allocated.

When this box is checked, textures data is saved and displayed in the calls log file.

You can select the format by choosing the appropriate radio button. Available formats are: JPG, BMP, PNG, and TIFF. 3D textures are stored as tiff images.

Set the logging limit for OpenGL API function calls and OpenCL
API function calls. If this limit is exceeded, the log is cleared. Define frame terminators in the project settings to avoid this automatic clearing.

**Advanced**

<table>
<thead>
<tr>
<th>Floating-Point Precision</th>
<th>The maximum number of significant digits that will be displayed in the Object and Multi-Watch views.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush Log File After Every Monitored Function Call</td>
<td>When the Flush log file after every monitored function call check box is checked, CodeXL will flush the OpenCL / OpenGL calls history log file after every API function call instead of not use memory cached batches. This feature can help tracking the function call that led to a debugged application crash. Using this feature dramatically slows down the debugged application performance and therefore it is not recommended for regular use.</td>
</tr>
<tr>
<td>Restore Default Settings</td>
<td>Click this button to restore all settings on all pages to their default values.</td>
</tr>
</tbody>
</table>
GPU Debugging API Support

CodeXL GPU debugging supports OpenCL up to and including version 2.0, OpenGL up to and including version 4.5 (including compatibility profiles) and a large variety of OpenCL and OpenGL extensions.

Kernel debugging is not supported for OpenCL 2.0 kernels.

The supported extensions include:

### OpenCL Extensions

<table>
<thead>
<tr>
<th>Extension name</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_khr_3d_image_writes</td>
</tr>
<tr>
<td>cl_khr_byte_addressable_store</td>
</tr>
<tr>
<td>cl_khr_dx9_media_sharing</td>
</tr>
<tr>
<td>cl_khr_d3d10_sharing</td>
</tr>
<tr>
<td>cl_khr_d3d11_sharing</td>
</tr>
<tr>
<td>cl_khr_fp64</td>
</tr>
<tr>
<td>cl_khr_gl_sharing</td>
</tr>
<tr>
<td>cl_khr_icd</td>
</tr>
<tr>
<td>cl_khr_global_int32_base_atomics</td>
</tr>
<tr>
<td>cl_khr_global_int32_extended_atomics</td>
</tr>
<tr>
<td>cl_khr_local_int32_base_atomics</td>
</tr>
<tr>
<td>cl_khr_local_int32_extended_atomics</td>
</tr>
<tr>
<td>cl_ext_atomic_counters_32</td>
</tr>
<tr>
<td>cl_ext_cl20_atomics</td>
</tr>
<tr>
<td>cl_ext_cl20_svm</td>
</tr>
<tr>
<td>cl_ext_device_fission</td>
</tr>
<tr>
<td>cl_amd_c11_atomics</td>
</tr>
</tbody>
</table>
cl_amd_device_attribute_query
cl_amd_fp64
cl_amd_media_ops
cl_amd_popcnt
cl_amd_printf
cl_amd_svm
cl_amd_vec3
cl_nv_d3d9_sharing
cl_nv_d3d10_sharing
cl_nv_d3d11_sharing

* Using this extension in a kernel might prevent the kernel from

**OpenGL extensions**

<table>
<thead>
<tr>
<th>Extension name</th>
<th>Support level</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_ARB_arrays_of_arrays</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_base_instance</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_blend_func_extended</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_clear_buffer_object</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_color_buffer_float</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_compatibility</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_compressed_texture_pixel_storage</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_compute_shader</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_conservative_depth</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_copy_buffer</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_copy_image</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_debug_output</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_debug_output</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_depth_buffer_float</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_depth_clamp</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_draw_buffers</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_draw_buffers_blend</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_draw_elements_base_vertex</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_draw_elements_base_vertex</td>
<td>Full</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>GL_ARB_draw_indirect</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_draw_instanced</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_ES2_compatibility</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_ES3_compatibility</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_explicit_attrib_location</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_explicit_uniform_location</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_fragment_coord_conventions</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_fragment_coord_conventions</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_fragment_layer_viewport</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_fragment_program</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_fragment_program_shadow</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_fragment_shader</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_framebuffer_no_attachments</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_framebuffer_object</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_framebuffer_sRGB</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_geometry_shader4</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_get_program_binary</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_gpu_shader_fp64</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_gpu_shader5</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_half_float_pixel</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_half_float_vertex</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_instanced_arrays</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_instanced_arrays</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_internalformat_query</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_internalformat_query2</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_invalidate_subdata</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_map_buffer_alignment</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_map_buffer_range</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_matrix_palette</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_multi_draw_indirect</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_multisample</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_multitexture</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_occlusion_query</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_occlusion_query2</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_pixel_buffer_object</td>
<td>Standard</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>GL_ARB_point_parameters</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_point_sprite</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_program_interface_query</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_provoking_vertex</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_provoking_vertex</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_robust_buffer_access_behavior</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_sample_shading</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_sampler_objects</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_seamless_cube_map</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_seamless_cube_map</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_separate_shader_objects</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shader_atomic_counters</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_shader_bit_encoding</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shader_image_load_store</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shader_image_size</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shader_objects</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_shader_precision</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shader_storage_buffer_object</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shader_subroutine</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shading_language_100</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_shading_language_420pack</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shading_language_packing</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_shadow</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_shadow_ambient</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_stencil_texturing</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_sync</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_sync</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_tessellation_shader</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_border_clamp</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_buffer_object</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_buffer_object_rgb32</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_buffer_range</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_compression</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_compression_rgtc</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_cube_map</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_cube_map_array</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_env_add</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_env_combine</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_env_crossbar</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_env_dot3</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_float</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_gather</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_mirrored_repeat</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_multisample</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_non_power_of_two</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_query_levels</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_rectangle</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_texture_rg</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_rgb10_a2ui</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_storage</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_storage_multisample</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_swizzle</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_texture_view</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_timer_query</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_transform_feedback_instanced</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_transform_feedback2</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_transform_feedback3</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_transpose_matrix</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_uniform_buffer_object</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_vertex_array_bgra</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_vertex_array_bgra</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_vertex_array_object</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_vertex_attrib_64bit</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_vertex_attrib_binding</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_vertex_blend</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ARB_vertex_buffer_object</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_vertex_program</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_vertex_shader</td>
<td>Full</td>
</tr>
<tr>
<td>OpenGL Extension Name</td>
<td>Status</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>GL_ARB_vertex_type_2_10_10_10_rev</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_viewport_array</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ARB_window_pos</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_bgra</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_bindable_uniform</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_blend_logic_op</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_blend_minmax</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_blend_subtract</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_compiled_vertex_array</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_direct_state_access</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_draw_instanced</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_framebuffer_blit</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_framebuffer_multisample</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_framebuffer_object</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_geometry_shader4</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_multi_draw_arrays</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_packed_pixels</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_stencil_two_side</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_texture</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_texture_array</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_texture_buffer_object</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_texture_integer</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_texture_rectangle</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_texture_shared_exponent</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_texture3D</td>
<td>Full</td>
</tr>
<tr>
<td>GL_EXT_vertex_shader</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_AMD_debug_output</td>
<td>Full</td>
</tr>
<tr>
<td>GL_AMDX_debug_output</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_aux_depth_stencil</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_client_storage</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_element_array</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_fence</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_float_pixels</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_flush_buffer_range</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_flush_render</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_object_purgeable</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_packed_pixels</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_pixel_buffer</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_specular_vector</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_texture_range</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_transform_hint</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_vertex_array_object</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_vertex_array_range</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_APPLE_vertex_program_evaluators</td>
<td>Full</td>
</tr>
<tr>
<td>GL_APPLE_ycbcr_422</td>
<td>Full</td>
</tr>
<tr>
<td>GL_ATI_draw_buffers</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ATI_fragment_shader</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_ATI_text_fragment_shader</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_GREMEDY_frame_terminator</td>
<td>Full</td>
</tr>
<tr>
<td>GL_GREMEDY_string_marker</td>
<td>Full</td>
</tr>
<tr>
<td>GL_HP_occlusion_test</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_KHR_debug</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_fragment_program</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_fragment_program_option</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_fragment_program2</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_geometry_shader4</td>
<td>Full</td>
</tr>
<tr>
<td>GL_NV_occlusion_query</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_primitive_restart</td>
<td>Full</td>
</tr>
<tr>
<td>GL_NV_texgen_reflection</td>
<td>Full</td>
</tr>
<tr>
<td>GL_NV_texture_rectangle</td>
<td>Full</td>
</tr>
<tr>
<td>GL_NV_texture_shader</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_texture_shader3</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_vertex_program</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_vertex_program1_1</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_vertex_program2</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_vertex_program2_option</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_vertex_program2</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_NV_vertex_program3</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_OES_draw_texture</td>
<td>Standard</td>
</tr>
<tr>
<td>GL_SGIS_generate_mipmap</td>
<td>Full</td>
</tr>
<tr>
<td>GL_SGIS_texture_border_clamp</td>
<td>Full</td>
</tr>
<tr>
<td>Extension</td>
<td>Support Level</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>GL_SGIS_texture_edge_clamp</td>
<td>Full</td>
</tr>
<tr>
<td>GL_SGIS_texture_lod</td>
<td>Full</td>
</tr>
<tr>
<td>GL_SGIS_texture_select</td>
<td>Full</td>
</tr>
<tr>
<td>GL_SGI_X_depth_texture</td>
<td>Full</td>
</tr>
<tr>
<td>GL_SGI_X_interlace</td>
<td>Full</td>
</tr>
<tr>
<td>GL_SGI_X_shadow</td>
<td>Full</td>
</tr>
<tr>
<td>GL_SGI_X_shadow_ambient</td>
<td>Full</td>
</tr>
<tr>
<td>GLX_ARB_create_context</td>
<td>Full</td>
</tr>
<tr>
<td>GLX_ARB_create_context_profile</td>
<td>Full</td>
</tr>
<tr>
<td>GLX_ARB_fbconfig_float</td>
<td>Standard</td>
</tr>
<tr>
<td>GLX_ARB_framebuffer_sRGB</td>
<td>Full</td>
</tr>
<tr>
<td>GLX_ARB_get_proc_address</td>
<td>Full</td>
</tr>
<tr>
<td>GLX_ARB_multisample</td>
<td>Standard</td>
</tr>
<tr>
<td>GLX_SGI_video_sync</td>
<td>Standard</td>
</tr>
<tr>
<td>GLX_SGI_X_fbconfig</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_AMD_gpu_association</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_ARB_buffer_region</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_ARB_create_context</td>
<td>Full</td>
</tr>
<tr>
<td>WGL_ARB_create_context_profile</td>
<td>Full</td>
</tr>
<tr>
<td>WGL_ARB_extensions_string</td>
<td>Full</td>
</tr>
<tr>
<td>WGL_ARB_framebuffer_sRGB</td>
<td>Full</td>
</tr>
<tr>
<td>WGL_ARB_make_current_read</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_ARB_multisample</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_ARB_pbuffer</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_ARB_pixel_format</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_ARB_pixel_format_float</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_ARB_render_texture</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_I3D_genlock</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_NV_gpu_affinity</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_NV_present_video</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_NV_swap_group</td>
<td>Standard</td>
</tr>
<tr>
<td>WGL_NV_video_out</td>
<td>Standard</td>
</tr>
</tbody>
</table>

**Standard Extension Support Level**
The Standard Extension Support Level enables one to log the calls and arguments of the extension functions, set breakpoints at the extension functions, and watch the values of the extension states variables.

**Full Extension Support Level**

In addition to supporting the Standard Extension Support Level features, the Full Extension Support Level enables one to view the extension-related data in the corresponding CodeXL views.

© 2015 Advanced Micro Devices. All rights reserved.
GPU Kernel Debugging Support

CodeXL GPU debugging supports OpenCL kernel debugging. In order to step into a kernel, and start debugging, use one of the following methods:

1. Breakpoint in clEnqueueNDRangeKernel
2. Break in a kernel

**Breakpoint in clEnqueueNDRangeKernel**

1. Put a breakpoint in clEnqueueNDRangeKernel using the [breakpoints](#) dialog.

   2. Once the breakpoint hits, click F11, or step-in button

   3. The kernel source file is opened, and you can use step-over, step-in and step-out buttons.

**Break in a kernel**

1. Run your application.

2. After the kernel is created, break into the debugger.

3. Open the [breakpoints](#) dialog.

4. Navigate to the “Kernel Functions” tab

5. Select the kernel in which you want to break, and click the “Add button”.

6. Click Ok.
7. Resume the debugged application.

8. Next time this kernel is executed, the kernel source file will open, and you will be able to use step-over, step-in and step-out buttons.

**Kernel Debugging**

Once CodeXL goes into kernel debugging mode, the kernel source code is opened. If the kernel was created with a string, a temporary file with the kernel source is created.
While debugging the kernel you can:
1. Put a breakpoint in any of the kernel source code lines.

2. Step-in, Step-out and Step-over the kernel source code lines.

3. See the values of variables in various views:
   
   **Locals view**

   The locals view displays a tree of the current local variables.

   ![Locals view](image)

   - **Watch view**

   Type an expression, or drag an expression from the source code view to see the current variable value.

   ![Watch view](image)

   Use the work item toolbar in order to change the currently viewed work item both in Watch view and Locals view.
- **Multi-watch view**

In order to see multiple work item view of a variable, open the Multi-watch view and use the visual and numeric data representation to see the variable’s values.

- **Use the mouse to hover a variable and watch its value in a tooltip**

As seen in the attached screenshot, once CodeXL is in Kernel Debugging mode, when the mouse cursor hovers over the variable name, a tooltip with the variable value is displayed.
CPU Profiler

CodeXL's CPU Profiler is used for performance analysis and tuning of applications running on CPU.

The CPU Profiler lets you identify main performance bottlenecks of the profiled application or the entire system.

Features

- Three profile modes:
  - Time-Based Profile (TBP) can identify the “hot-spots” in the profiled applications. (Hot-spots are code areas that use significantly more time compared to other areas in the code.)
  - Event-Based Profile (EBP) can identify CPU and memory related performance issues in the profiled applications.
  - Instruction-Based Sampling (IBS) can record and count the instructions that trigger HW events, as well as calculate various metrics, such as data cache latency.
- Per-Process mode and System-Wide mode profiling:
  - **Per-process mode** profiles a process and its children.
  - **System-wide mode** profiles the entire system.
- Attach to process for profiling.
- User-mode profiling and Kernel-mode profiling (Windows only).
- Native applications profiling (C, C++ and Fortran).
- Profiling C++ inline functions.
- CLR/.NET applications profiling (only on Windows).
- Java applications profiling.
- Call Stack Sampling (CSS) for all profile modes.
- Aggregation of the collected samples at various levels:
Process/Modules/Functions/Source and Disassembly.
· HW events counter multiplexing.
· Debugging Data Formats Supported:
  – CodeXL supports symbol information for unmanaged Executables compiled by MS Visual Studio or GCC (under Linux or other Unix-like systems (like Cygwin and MinGW)). That includes the following debugging data formats: PDB, COFF, DWARF, STABS.
  – For managed Executables, CodeXL supports Java and .NET applications’ debug information.
· Time-Based Profile (TBP) and Event-Based Profile (EBP) are supported in guest OS running on VMware Workstation 11.0 or later.
· Time-Based Profile (TBP) and Event-Based Profile (EBP) are supported on Microsoft Hyper-V.
· Time-Based Profile (TBP) is supported on Xen Project hypervisor.
· Time-Based Profile (TBP) is supported on Linux KVM hypervisor.
· Controlling CPU Profiling i.e. pause and resume profiling, from target application to limit profiling scope.
· Aggregated instruction based (IMIX) report generation from CPU Profiler CLI.

Limitations
· CPU Profiler expects the profiled application executable binaries must not be compressed or obfuscated by any software protector tools, e.g. VMProtect.

Using the CPU Profiler

This section explains various key concepts related to CPU Profiling. It consists of the following subsections.
· CPU Profile Key Concepts
· CPU Profile Configurations
- **CPU Profile Session**
- **CPU Profile Data Analysis**
- **CPU Profile Command Line Interface**
- **CPU Profile C/C++ Inline Functions**
- **CPU Profile PLT Relocations**
- **CPU Profile on Virtual Machine**
- **CPU Profile Control APIs**
- **CPU Profile IMIX report generation**
CPU Profile Key Concepts

This section explains various key concepts related to CPU Profiling.

CPU Profiling in CodeXL

The CodeXL CPU Profiler follows a statistical sampling-based approach to gather the profile data periodically. It uses a variety of SW and HW resources available in AMD x86 based processor families. CPU Profiler uses the SW timer, HW Performance Monitor Counters (PMC), and HW IBS feature. The most time-consuming parts of a program have a larger number of samples; this is because they have a higher probability of being executed while samples are being taken by the CPU Profiler.

Sampling Interval

The time between the collection of every two samples is the Sampling Interval. For example, in TBP, if the time interval is 1 millisecond, then roughly 1,000 TBP samples are being collected every second for each processor core.

HW Performance Monitor Counters (PMC)

AMD's x86-based processors have Performance Monitor Counters (PMC) that let them monitor various micro-architectural events in a CPU core. The PMC counters are used in two modes:

- In counting mode, these counters are used to count the specific events that occur in a CPU core.
In sampling mode, these counters are programmed to count a specific number of events; once the count is reached the appropriate number of times (called sampling interval), an interrupt is triggered. During the interrupt handling, the CPU Profiler collects profile data.

The number of hardware performance event counters available in each processor is implementation-dependent (see the BIOS and Kernel Developer's Guide [BKDG] of the specific processor for the exact number of hardware performance counters). The operating system and/or BIOS can reserve one or more counters for internal use. Thus, the actual number of available hardware counters may be less than the number of hardware counters. The CPU Profiler uses all available counters for profiling.

**Time-Based Profile (TBP)**

In this profile mode, the profile data is periodically collected based on the specified timer interval. It is used to identify the hot-spots of the profiled applications.

**Event-Based Profile (EBP)**

In this mode, the CPU Profiler uses the PMCs to monitor the various micro-architectural events supported by the AMD x86-based processor. It helps to identify the CPU and memory related performance issues in profiled applications. CodeXL provides a number of predefined EBP profile configurations. To analyze a particular aspect of the profiled application (or system), a specific set of relevant events are grouped and monitored together. The CPU Profiler provides a list of predefined event configurations, such as Assess Performance and Investigate Branching, etc. You can select any of these predefine configurations to profile and analyze the runtime characteristics of your application. You also can create their custom configurations of events to profile.

This profile mode is supported on the various AMD processor
models, such as Family 10h, Family 11h, Family 12h, Family 14h, Family 15h models 00h-0Fh, 10-1Fh, 30-3Fh and Family 16h models 00-0Fh.

In this profile mode, a delay called skid occurs between the time at which the sampling interrupt occurs and the time at which the sampled instruction address is collected. This skid distributes the samples in the neighborhood near the actual instruction that triggered a sampling interrupt. This produces an inaccurate distribution of samples and events are often attributed to the wrong instructions.

**Instruction-Based Sampling (IBS)**

In this mode, the CPU Profiler uses the IBS HW supported by the AMD x86-based processor to observe the effect of instructions on the processor and on the memory subsystem. In IBS, HW events are linked with the instruction that caused them. Also, HW events are being used by the CPU Profiler to derive various metrics, such as data cache latency.

IBS is supported starting from the AMD processor family 10h.

**Event-Counter Multiplexing**

If the number of monitored PMC events is less than, or equal to, the number of available performance counters, then each event can be assigned to a counter, and each event can be monitored 100% of the time. In a single-profile measurement, if the number of monitored events is larger than the number of available counters, the CPU Profiler time-shares the available HW PMC counters. (This is called event counter multiplexing.) It helps monitor more events and decreases the actual number of samples for each event, thus reducing data accuracy. The CPU Profiler auto-scales the sample counts to compensate for this event counter multiplexing. For example, if an event is monitored 50% of the time, the CPU Profiler scales the number of event samples by factor of 2.
CPU Profile Configurations

CodeXL handles all of the details and mechanics of data collection and profile formation.

CodeXL uses statistical sampling to collect and build a program profile. Because all profiles rely on statistical sampling, it is important for the CPU Profiler to take enough samples. The number of samples collected during a session depends on the following:

- how frequently samples are taken (the sampling period or interval)
- the measurement length; that is, the amount of time which takes for the CPU Profiler to take a specific sample
- whether the profiling is system-wide

CodeXL provides the following profile sessions types:

- Time-Based Profiling
- Assess Performance
- Instruction-Based Sampling
- Investigate Branching
- Investigate Data Access
- Investigate Instruction Access
- Investigate L2 Cache Access
- Cache Line Utilization
- Custom Profile

The type and frequency of the profiled events can indicate the presence of a pipeline bottleneck, poor memory access pattern,
poorly predicted conditional branches, or some other performance issues. Once hot-spots are found through time-based profiling, EBP and IBS are used identify opportunities for optimization. AMD processors provide a wide range of hardware events that can be monitored and measured. The number of counters and the hardware events that can be measured are processor-dependent.

When compiling your application to use for profiling, consider generating debug information as well. If present, the debug information can help identify profiled areas in the code; however, debug information is not required. Performance data can be collected for an application program that was compiled without debug information; in this case the results displayed by CodeXL are less descriptive. For example, CodeXL displays assembly code rather than function names or source code.

The CPU Profiler measures CPU execution time of a program. Through these measurements it helps you optimize the program. The Profiler records which functions were opened and how long it took to execute each one. CodeXL does this by taking samples. To take a sample, CodeXL interrupts the program’s execution at specified intervals, and logs the state of the program’s call chain.

CodeXL samples all of the various performance monitoring registers to obtain detailed information about the running application or the entire system. While all work can be done through the GUI, profiles also can be collected and analyzed through Visual Studio by using the CodeXL Visual Studio extension.

System profiling identifies a hot-spot anywhere in the system or an application under test. Any software component (an executable image, dynamically loaded library, device driver, or even the operating system kernel) that executes during the measurement period can be sampled. Any child processes spawned by a profiled process are profiled automatically. During the profiling process, the application to be analyzed is run at full speed on the same machine that is running CodeXL.
Time-based samples (collected at 1 ms intervals on each core) can be used to identify possible bottlenecks, execution penalties, or optimization opportunities. The TBP feature can be used on both AMD and non-AMD processors with an Advanced Programmable Interrupt Controller (APIC) timer. Event-based samples and instruction-based samples can be used to help determine the cause of hot-spots or optimization opportunities. The sampling intervals are weighted so the types of view analysis are valid. The EBP and IBS features are only available on AMD processors.

Call chain sampling collects function call information, including caller-to-called relationships between functions. It is used in conjunction with the selected profile. When call chain sampling is enabled, CodeXL collects information from the run-time call stack of a monitored application process (and child processes) whenever a regular profile sample is taken for the process. When compared to other techniques, such as instrumentation, call chain sampling is a relatively low-overhead approach to the collection of function call information. However, call chain sampling results are subject to statistical variation.
Time-Based Profiling

When time-based profiling is enabled and started, CodeXL configures a timer that periodically interrupts the program executing on a processor core.

When a timer interrupt occurs, a sample is created and saved for post-processing. Post-processing builds up a type of histogram, which describes what the system and its software components were doing. The most time-consuming parts of a program have the most samples because, most likely, the program is executing in those regions when a timer interrupt is generated and a sample is taken.

The frequency of sample taking is controlled by the timer interval. This sometimes is called the "sampling period." The timer interval is 1 millisecond: roughly 1,000 TBP samples are taken each second for each processor core.

The second factor is the length of time during which the samples are taken. The measurement period depends on the overall execution time of the workload and the way in which CodeXL data collection is configured. Using the CPU Profile Options, CodeXL can be configured to collect samples for all, or part, of the time that the test workload is executing. If program run-time is short (less than 15 seconds), it may be necessary to increase program run-time by using a larger data set or more loop iterations to obtain a statistically useful result.

Deciding how many samples are enough requires a working knowledge about the characteristics of the workload under test. Scientific applications often have tight inner loops that are executed several times. In these situations, samples are being aggregated rapidly within the inner loops, and even a fairly
short run-time yields a statistically useful number of samples. Other workloads, like transaction processing, usually have just a few inner loops, and the profiles are relatively "flat." For flat workloads, a longer measurement period is required to aggregate samples in code regions of interest.
Assess Performance

This pre-defined configuration is intended to get an overall assessment of hardware performance and to give an idea of the possible causes of performance issues.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

- [0C0] Retired Instructions
- [076] CPU clock cycles not halted
- [0C2] Retired branch instructions
- [0C3] Retired mispredicted branch instructions
- [040] Data cache accesses
- [041] Data cache misses
- [046] L1 DTLB and L2 DTLB misses
- [047] Misaligned accesses

This profile configuration measures eight different events and requires event counter multiplexing. Each event is sampled approximately half of the time (a 50% duty cycle) when four hardware performance counters are available. When using a profile configuration that requires event counter multiplexing, ensure run time is long enough to build up a statistically accurate picture of program behavior.

The available views for profiles with this data are:

- All Data
- Branch assessment
- DTLB assessment
- Data access assessment
- IPC assessment
- Misaligned access assessment
· Overall assessment
Instruction-Based Sampling (IBS) identifies and diagnoses performance issues in program hot-spots. It collects data on how instructions behave on the processor and in the memory subsystem; it also provides a range of measurable data for each sample. When running IBS,

- hardware events are linked with the instructions that caused them.
- it produces a wealth of event data in a single test run.
- latency is measured for key performance factors such as data cache miss latency.

IBS provides the most common types of information needed for program performance analysis. It uses a hardware sampling technique to generate event information similar to that produced by event-based profiling. Event-based profiling, however, offers a wider range of events that can be monitored, such as those related to HyperTransport™ links.

Processor pipeline stages can be categorized into two main phases: instruction fetch and execution. Each instruction fetch operation produces a block of instruction data that is passed to the decode stages in the pipeline. The decoder identifies AMD64 instructions in the fetch block. These AMD64 instructions are translated to one or more macro-operations, called "macro-ops" or "ops," that are executed in the execution phase.

**Note:** For more information about instruction-based sampling, see the following documents which are available at AMD’s [Developer Guides & Manuals page](https://developer.amd.com) :

- [Software Optimization Guide for AMD Family 16h](https://developer.amd.com)
How IBS Works

IBS provides separate means to sample fetch operations and macro-ops. IBS fetch sampling and IBS op sampling can be enabled and collected separately or together.

**IBS Fetch Sampling**

This is a statistical sampling method. IBS fetch sampling counts the completed fetch operations. When the number of completed fetch operations reaches the maximum fetch count (the sampling period), IBS tags the fetch operation and monitors that fetch operation until it either completes or aborts.

When a tagged fetch completes or aborts, a sampling interrupt is generated, and an IBS fetch sample is taken. An IBS fetch sample contains a timestamp, the identifier of the interrupted process, the virtual fetch address, and several event flags and values that describe what happened during the fetch operation. Similar to time-based profiling and event-based profiling, CodeXL uses the IBS sample data and information from the executable images, debug information, and source to build a profile IBS for software components executed on the system. IBS is also available in system-wide profiling.

The event data reported in an IBS sample includes the
following:

- If the fetch completed or aborted.
- If the address translation initially missed in the level one (L1) or level two (L2) instruction translation lookaside buffer (ITLB).
- The page size of the L1 ITLB address translation (4K, 2M).
- Whether the fetch initially missed in the instruction cache (IC).
- Fetch latency (number of processor cycles from when the fetch was initiated to when the fetch completed or aborted).

Event-based profiling requires several counters to collect as much information as IBS. The fetch address precisely identifies the fetch operation associated with the hardware events. The IBS fetch address may be the address of a fetch block, the target of a branch, or the address of an instruction that is the fall-through of a conditional branch. A fetch block does not always start with a complete, valid AMD64 instruction; this occurs when an AMD64 instruction straddles two fetch blocks. In this case, CodeXL associates the IBS fetch sample with the AMD64 instruction in the preceding fetch block.

A fetch can be abandoned before it delivers data to the decoder, or due to a control flow redirection; this can happen at any time during the fetch process. A fetch abandoned before initial access to the ITLB (before address translation) is not regarded as useful for analysis. These early abandoned fetches are called **killed** fetches.

CodeXL identifies killed fetches. The fetch operations remaining after killed fetches are removed from consideration are called **attempted** fetches: these fetches represent valid attempts to obtain instruction bytes.

A **completed** fetch is an attempted fetch that successfully delivered instruction data to the decoder. An **aborted** fetch is an attempted fetch that did not complete.
Note: Instruction fetch is an aggressive, speculative activity, and even instruction data produced by a completed fetch may not be used.

**IbsOps IBS Op Sampling**

IBS op sampling operates like fetch sampling. It provides two methods for op selection:

- **Cycles mode** – IBS hardware counts processor cycles. When reaching the maximum cycle count (the sampling period), IBS tags an available valid op.
- **Dispatched op mode** – IBS hardware counts ops as they are issued into the pipeline. When the number of dispatched ops reaches the maximum op count (the sampling period), IBS tags the op. Dispatched op mode is preferred because Cycles mode selection is susceptible to delay induced sampling bias.

Note: Some processors do not support dispatched op mode. For more details, see the [BKDG for the AMD processor](#) for your platform. The execution stages of the pipeline monitor the tagged macro-op. When the tagged macro-op retires, a sampling interrupt is generated, and an IBS op sample is taken. An IBS op sample contains:

- a timestamp,
- the identifier of the interrupted process,
- the virtual address of the AMD64 instruction from which the op was issued, and
- several event flags and values that describe what happened when the macro-op executed.

CodeXL uses this and other information to build an IBS profile.

Cycle-based op sampling can be susceptible to timing bias: it can cause ops from some instructions to be selected more often than other instructions. Dispatched op-based sampling is the preferred IBS operating mode because it is not biased by timing.
IBS op samples are taken only for ops that retire. Thus, IBS op event information does not measure speculative execution activity. The cycles-based tagging scheme can introduce statistical bias due to stalls at the decoding stage of the pipeline. If a macro-op is not available for tagging when the maximum op count is reached, the hardware tags a macro-op and starts counting again from a small, pseudo-random initial count.

IBS op sampling reports the following values for all ops:
- Virtual address of the parent AMD64 instruction from which the tagged op was issued.
- Tag-to-retire time (the number of processor cycles from when the op was tagged to when the op retired).
- Completion-to-retire time (the number of processor cycles from when the op completed to when the op was retired).

Attribution of event information is precise because the IBS hardware reports the address of the AMD64 instruction causing the events. For example, branch mispredictions are attributed to the mispredicted branch, and cache misses are attributed to the AMD64 instruction that caused the cache miss. IBS makes it easier to identify the performance-degrading instructions.

Some ops implement branch semantics. Branches include unconditional and conditional branches, subroutine calls, and subroutine returns.

Event information reported for branch ops include whether the branch was mispredicted or was taken.

IBS also indicates whether a branch operation was a subroutine return, and if the return was mispredicted. Some ops can perform a load (memory read), store (memory write), or a load and a store to the same memory address, as in the case of a read-op-write sequence.

When an op performs a load and/or store, event information includes the following:
- Whether a load was performed.
Whether a store was performed.
- Whether address translation initially missed in the L1 and/or L2 data translation lookaside buffer (DTLB).
- Whether the load or store initially missed in the data cache (DC).
- Virtual data address for the memory operation.
- Latency when a load misses the DC.

Requests made through the Northbridge produce additional event information:
- Whether the access was local or remote.
- Data source that fulfilled the request.

A full list of IBS op event information appears in the section on IBS-Derive events below. For hardware-level details, see the BIOS and Kernel Developer's Guide (BKDG) for the AMD processor for your platform.

**IBS-Derived Events**

CodeXL translates the IBS information produced by the hardware into **derived event** sample counts that resemble EBP sample counts. All IBS-derived events have "IBS" in the event name and abbreviation. Although IBS-derived events and sample counts look similar to EBP events and sample counts, the source and sampling basis for the IBS event information are different.

Arithmetic should **never** be performed between IBS derived event sample counts and EBP event sample counts. It is not meaningful to directly compare the number of samples taken for events that represent the same hardware condition. For example, fewer IBS DC miss samples is not necessarily better than a larger quantity of EBP DC miss samples.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All IBS fetch samples</td>
<td>The number of all IBS fetch samples. This derived event counts the number of all IBS fetch samples that were</td>
</tr>
<tr>
<td>Event Description</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IBS fetch killed</td>
<td>The number of IBS sampled fetches that were killed fetches. A fetch operation is killed if the fetch did not reach ITLB or IC access. The number of killed fetch samples is not generally useful for analysis and are filtered out in other derived IBS fetch events (except Event Select 0xF000 which counts all IBS fetch samples including IBS killed fetch samples.)</td>
</tr>
<tr>
<td>IBS fetch attempted</td>
<td>The number of IBS sampled fetches that were not killed fetch attempts. This derived event measures the number of useful fetch attempts and does not include the number of IBS killed fetch samples. This event should be used to compute ratios such as the ratio of IBS fetch IC misses to attempted fetches. The number of attempted fetches should equal the sum of the number of completed fetches and the number of aborted fetches.</td>
</tr>
<tr>
<td>IBS fetch completed</td>
<td>The number of IBS sampled fetches that completed. A fetch is completed if the attempted fetch delivers instruction data to the instruction decoder. Although the instruction data was delivered, it may still not be used (e.g., the instruction data may have been on the &quot;wrong path&quot; of an incorrectly predicted branch.)</td>
</tr>
<tr>
<td>IBS fetch aborted</td>
<td>The number of IBS sampled fetches that aborted. An attempted fetch is aborted.</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>IBS fetch aborted</td>
<td>if it did not complete and deliver instruction data to the decoder. An attempted fetch may abort at any point in the process of fetching instruction data. An abort may be due to a branch redirection as the result of a mispredicted branch. The number of IBS aborted fetch samples is a lower bound on the amount of unsuccessful, speculative fetch activity. It is a lower bound since the instruction data delivered by completed fetches may not be used.</td>
</tr>
<tr>
<td>IBS ITLB hit</td>
<td>The number of IBS attempted fetch samples where the fetch operation initially hit in the L1 ITLB (Instruction Translation Lookaside Buffer).</td>
</tr>
<tr>
<td>IBS L1 ITLB misses (and L2 ITLB hits)</td>
<td>The number of IBS attempted fetch samples where the fetch operation initially missed in the L1 ITLB and hit in the L2 ITLB.</td>
</tr>
<tr>
<td>IBS L1 L2 ITLB miss</td>
<td>The number of IBS attempted fetch samples where the fetch operation initially missed in both the L1 ITLB and the L2 ITLB.</td>
</tr>
<tr>
<td>IBS instruction cache misses</td>
<td>The number of IBS attempted fetch samples where the fetch operation initially missed in the IC (instruction cache).</td>
</tr>
<tr>
<td>IBS instruction cache hit</td>
<td>The number of IBS attempted fetch samples where the fetch operation initially hit in the IC.</td>
</tr>
<tr>
<td><strong>IBS 4K page translation</strong></td>
<td>samples where the fetch operation produced a valid physical address (i.e., address translation completed successfully) and used a 4-KByte page entry in the L1 ITLB.</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>IBS 2M page translation</strong></td>
<td>The number of IBS attempted fetch samples where the fetch operation produced a valid physical address (i.e., address translation completed successfully) and used a 2-MByte page entry in the L1 ITLB.</td>
</tr>
<tr>
<td><strong>IBS fetch latency</strong></td>
<td>The total latency of all IBS attempted fetch samples. Divide the total IBS fetch latency by the number of IBS attempted fetch samples to obtain the average latency of the attempted fetches that were sampled.</td>
</tr>
<tr>
<td><strong>IBS fetch L2 cache miss</strong></td>
<td>The instruction fetch missed in the L2 Cache.</td>
</tr>
<tr>
<td><strong>IBS ITLB refill latency</strong></td>
<td>The number of cycles when the fetch engine is stalled for an ITLB reload for the sampled fetch. If there is no reload, the latency will be 0.</td>
</tr>
<tr>
<td><strong>All IBS op samples</strong></td>
<td>The number of all IBS op samples that were collected. These op samples may be branch ops, resync ops, ops that perform load/store operations, or undifferentiated ops (e.g., those ops that perform arithmetic operations, logical operations, etc.). IBS collects data for retired ops. No data is collected for ops that are aborted due to pipeline flushes, etc. Thus, all sampled ops are architecturally significant and</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IBS tag-to-retire cycles</td>
<td>The total number of tag-to-retire cycles across all IBS op samples. The tag-to-retire time of an op is the number of cycles from when the op was tagged (selected for sampling) to when the op retired.</td>
</tr>
<tr>
<td>IBS completion-to-retire cycles</td>
<td>The total number of completion-to-retire cycles across all IBS op samples. The completion-to-retire time of an op is the number of cycles from when the op completed to when the op retired.</td>
</tr>
<tr>
<td>IBS branch op</td>
<td>The number of IBS retired branch op samples. A branch operation is a change in program control flow and includes unconditional and conditional branches, subroutine calls and subroutine returns. Branch ops are used to implement AMD64 branch semantics.</td>
</tr>
<tr>
<td>IBS mispredicted branch op</td>
<td>The number of IBS samples for retired branch operations that were mispredicted. This event should be used to compute the ratio of mispredicted branch operations to all branch operations.</td>
</tr>
<tr>
<td>IBS taken branch op</td>
<td>The number of IBS samples for retired branch operations that were taken branches.</td>
</tr>
<tr>
<td>IBS mispredicted taken branch op</td>
<td>The number of IBS samples for retired branch operations that were mispredicted taken branches.</td>
</tr>
</tbody>
</table>

Contribute to the successful forward progress of executing programs.
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBS return op</td>
<td>The number of IBS retired branch op samples where the operation was a subroutine return. These samples are a subset of all IBS retired branch op samples.</td>
</tr>
<tr>
<td>IBS mispredicted return op</td>
<td>The number of IBS retired branch op samples where the operation was a mispredicted subroutine return. This event should be used to compute the ratio of mispredicted returns to all subroutine returns.</td>
</tr>
<tr>
<td>IBS resync op</td>
<td>The number of IBS resync op samples. A resync op is only found in certain microcoded AMD64 instructions and causes a complete pipeline flush.</td>
</tr>
<tr>
<td>IBS all load store ops</td>
<td>The number of IBS op samples for ops that perform either a load and/or store operation. An AMD64 instruction may be translated into one (&quot;single fastpath&quot;), two (&quot;double fastpath&quot;), or several (&quot;vector path&quot;) ops. Each op may perform a load operation, a store operation or both a load and store operation (each to the same address). Some op samples attributed to an AMD64 instruction may perform a load/store operation while other op samples attributed to the same instruction may not. Further, some branch instructions perform load/store operations. Thus, a mix of op sample types may be attributed to a single AMD64 instruction depending upon the ops that are issued from the AMD64 instruction and the op types.</td>
</tr>
<tr>
<td>IBS load ops</td>
<td>The number of IBS op samples for ops that perform a load operation.</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>IBS store ops</td>
<td>The number of IBS op samples for ops that perform a store operation.</td>
</tr>
<tr>
<td>IBS L1 DTLB hit</td>
<td>The number of IBS op samples where either a load or store operation initially hit in the L1 DTLB (data translation lookaside buffer).</td>
</tr>
<tr>
<td>IBS L1 DTLB misses L2 hits</td>
<td>The number of IBS op samples where either a load or store operation initially missed in the L1 DTLB and hit in the L2 DTLB.</td>
</tr>
<tr>
<td>IBS L1 and L2 DTLB misses</td>
<td>The number of IBS op samples where either a load or store operation initially missed in both the L1 DTLB and the L2 DTLB.</td>
</tr>
<tr>
<td>IBS data cache misses</td>
<td>The number of IBS op samples where either a load or store operation initially missed in the data cache (DC).</td>
</tr>
<tr>
<td>IBS data cache hits</td>
<td>The number of IBS op samples where either a load or store operation initially hit in the data cache (DC).</td>
</tr>
<tr>
<td>IBS misaligned data access</td>
<td>The number of IBS op samples where either a load or store operation caused a misaligned access (i.e., the load or store operation crossed a 128-bit boundary).</td>
</tr>
<tr>
<td>IBS bank conflict on load op</td>
<td>The number of IBS op samples where either a load or store operation caused a bank conflict with a load operation.</td>
</tr>
<tr>
<td>IBS bank conflict on</td>
<td>The number of IBS op samples where</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>store op</td>
<td>Either a load or store operation caused a bank conflict with a store operation.</td>
</tr>
<tr>
<td>IBS store-to-load forwarded</td>
<td>The number of IBS op samples where data for a load operation was forwarded from a store operation.</td>
</tr>
<tr>
<td>IBS store-to-load cancelled</td>
<td>The number of IBS op samples where data forwarding to a load operation from a store was cancelled.</td>
</tr>
<tr>
<td>IBS UC memory access</td>
<td>The number of IBS op samples where a load or store operation accessed uncacheable (UC) memory.</td>
</tr>
<tr>
<td>IBS WC memory access</td>
<td>The number of IBS op samples where a load or store operation accessed write combining (WC) memory.</td>
</tr>
<tr>
<td>IBS locked operation</td>
<td>The number of IBS op samples where a load or store operation was a locked operation.</td>
</tr>
<tr>
<td>IBS MAB hit</td>
<td>The number of IBS op samples where a load or store operation hit an already allocated entry in the Miss Address Buffer (MAB).</td>
</tr>
<tr>
<td>IBS L1 DTLB 4K page</td>
<td>The number of IBS op samples where a load or store operation produced a valid linear (virtual) address and a 4-KByte page entry in the L1 DTLB was used for address translation.</td>
</tr>
<tr>
<td>IBS L1 DTLB 2M page</td>
<td>The number of IBS op samples where a load or store operation produced a valid linear (virtual) address and a 2-MByte page entry in the L1 DTLB was used for address translation.</td>
</tr>
<tr>
<td>IBS L1 DTLB 1G page</td>
<td>The number of IBS op samples where a load or store operation produced a valid linear (virtual) address and a 1-GByte page entry in the L1 DTLB was used for address translation.</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IBS L2 DTLB 4K page</td>
<td>The number of IBS op samples where a load or store operation produced a valid linear (virtual) address, hit the L2 DTLB, and used a 4 KByte page entry for address translation.</td>
</tr>
<tr>
<td>IBS L2 DTLB 2M page</td>
<td>The number of IBS op samples where a load or store operation produced a valid linear (virtual) address, hit the L2 DTLB, and used a 2-MByte page entry for address translation.</td>
</tr>
<tr>
<td>IBS L2 DTLB 1G page</td>
<td>The number of IBS op samples where a load or store operation produced a valid linear (virtual) address, hit the L2 DTLB, and used a 1-GByte page entry for address translation.</td>
</tr>
<tr>
<td>IBS data cache miss load latency</td>
<td>The total DC miss load latency (in processor cycles) across all IBS op samples that performed a load operation and missed in the data cache. The miss latency is the number of clock cycles from when the data cache miss was detected to when data was delivered to the core. Divide the total DC miss load latency by the number of data cache misses to obtain the average DC miss load latency.</td>
</tr>
<tr>
<td>IBS load resync</td>
<td>Load Resync.</td>
</tr>
<tr>
<td></td>
<td>The number of IBS op samples where a</td>
</tr>
<tr>
<td>IBS Northbridge local</td>
<td>load operation was serviced from the local processor. Northbridge IBS data is only valid for load operations that miss in both the L1 data cache and the L2 data cache. If a load operation crosses a cache line boundary, then the IBS data reflects the access to the lower cache line.</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IBS Northbridge remote</td>
<td>The number of IBS op samples where a load operation was serviced from a remote processor.</td>
</tr>
<tr>
<td>IBS Northbridge local L3</td>
<td>The number of IBS op samples where a load operation was serviced by the local L3 cache.</td>
</tr>
<tr>
<td>IBS Northbridge local core L1 or L2 cache</td>
<td>The number of IBS op samples where a load operation was serviced by a cache (L1 data cache or L2 cache) belonging to a local core which is a sibling of the core making the memory request.</td>
</tr>
<tr>
<td>IBS Northbridge local core L1, L2, L3 cache</td>
<td>The number of IBS op samples where a load operation was serviced by a remote L1 data cache, L2 cache or L3 cache after traversing one or more coherent HyperTransport links.</td>
</tr>
<tr>
<td>IBS Northbridge local DRAM</td>
<td>The number of IBS op samples where a load operation was serviced by local system memory (local DRAM via the memory controller).</td>
</tr>
<tr>
<td>IBS Northbridge remote DRAM</td>
<td>The number of IBS op samples where a load operation was serviced by remote system memory (after traversing one or more coherent HyperTransport links and through a remote memory)</td>
</tr>
<tr>
<td>IBS Northbridge local APIC MMIO Config PCI</td>
<td>The number of IBS op samples where a load operation was serviced from local MMIO, configuration or PCI space, or from the local APIC.</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IBS Northbridge remote APIC MMIO Config PCI</td>
<td>The number of IBS op samples where a load operation was serviced from remote MMIO, configuration or PCI space.</td>
</tr>
<tr>
<td>IBS Northbridge cache modified state</td>
<td>The number of IBS op samples where a load operation was serviced from local or remote cache, and the cache hit state was the Modified (M) state.</td>
</tr>
<tr>
<td>IBS Northbridge cache owned state</td>
<td>The number of IBS op samples where a load operation was serviced from local or remote cache, and the cache hit state was the Owned (O) state.</td>
</tr>
<tr>
<td>IBS Northbridge local cache latency</td>
<td>The total data cache miss latency (in processor cycles) for load operations that were serviced by the local processor.</td>
</tr>
<tr>
<td>IBS Northbridge remote cache latency</td>
<td>The total data cache miss latency (in processor cycles) for load operations that were serviced by a remote processor.</td>
</tr>
</tbody>
</table>
Investigate Branching

This pre-defined configuration is intended to assist an investigation into branching and near-return performance.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

- [0C0] Retired Instructions
- [0C2] Retired branch instructions
- [0C3] Retired mispredicted branch instructions
- [0C4] Retired taken branch instructions
- [0C8] Retired near returns
- [0C9] Retired mispredicted near returns
- [0CA] Retired mispredicted indirect branches

The available views for profiles with this data are:

- All Data
- Branch assessment
- Near return report
- Taken branch report
Investigate Data Access

This pre-defined configuration helps investigate data locality and poor DTLB behavior.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

- [0C0] Retired Instructions
- [040] Data cache accesses
- [041] Data cache misses
- [042] Data cache refills from L2 or Northbridge
- [045] L1 DTLB miss and L2 DTLB hit
- [046] L1 DTLB and L2 DTLB misses
- [047] Misaligned accesses

The available views for profiles with this data are:

- All Data
- DTLB assessment
- DTLB report
- Data access assessment
- Data access report
- Misaligned access assessment
Investigate Instruction Access

This pre-defined configuration helps investigate instruction fetches with poor L1 locality and poor ITLB behavior.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

- [0C0] Retired Instructions
- [080] Instruction Cache fetches
- [081] Instruction Cache misses
- [084] L1 ITLB miss and L2 ITLB hits
- [085] L1 ITLB miss and L2 ITLB miss

The available views for profiles with this data are:

- All Data
- ITLB report
- Instruction Cache Report

© 2015 Advanced Micro Devices. All rights reserved.
Investigate L2 Cache Access

This pre-defined configuration helps investigate memory access operations with poor L2 cache locality.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

- [0C0] Retired Instructions
- [07D] Requests to L2 cache
- [07E] L2 cache misses
- [07F] L2 fill/writeback

The available views for profiles with this data are:

- All Data
- L2 Access Report

© 2015 Advanced Micro Devices. All rights reserved.
Cache Line Utilization

This feature is a first step towards providing data-centric application profiling capabilities. This feature models the behavior of the processor L1 data cache and uses the Load and Store Instruction-Based Sampling (IBS) records to provide a measure of how efficiently an application utilizes the L1 data cache.

A cache is a relatively small amount of on-chip memory which is extremely fast compare to main memory. When the processor needs to access a location in main memory, it first checks whether a copy of that data is in the cache. If the data is present in cache, it is called a cache hit and the processor immediately accesses the data from cache. If the data is not in the cache, it is called a cache miss. In this case processor has to wait for the data to be fetched from main memory before it can continue to execute. All of the data required by all of the processes running on a processor cannot simultaneously fit in the cache, so the processor removes, or evicts, data from the cache when new data is needed and the cache is full. Data is transferred between memory and cache in blocks of fixed size, called cache lines.

The cache misses directly influence the performance of the application. Having the data in the cache when the processor needs it is one way to optimize performance of an application. Additionally, because cache size is small, it is desirable to fill the cache with data that will be used before it is evicted from the cache.

AMD processors have a separate instruction and data cache per core (L1 (Level 1) instruction and L1 data caches) as well as a unified L2 (per module) cache and L3 (per-chip) cache.
However, CLU models the L1 data cache only. CLU measures how much of a cache line is used (read or written) before it is evicted from the cache. The percent cache line utilization is defined as percentage of number of bytes in the cache line had been accessed before the cache line has been evicted.

A low CLU value implies that the cache is being filled with data that is never or less accessed before it is evicted, implying cache capacity pressure, as well as main memory bandwidth pressure (reading data from main memory that is not accessed before being evicted).

A high usage percentage (CLU) means that the application is properly exploiting spatial and temporal locality of its data. Ideally, one would like to have 100 percent CLU. Practically speaking however, a good CLU is about 20 to 30 percent, primarily due to the sampling nature of the core in its collection of the load and store data.

Note: See the BIOS and Kernel Developer's Guide for AMD Family 10h Processors (order #31116) for detailed information about caches.

The following table describes the data that can be shown for each module, function, source and disassembly

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Line Utilization</td>
<td>The cache line utilization percentage for all cache lines on all cores accessed by this instruction / function / module.</td>
</tr>
<tr>
<td>Utilization Percentage</td>
<td></td>
</tr>
<tr>
<td>Line Boundary Crossings</td>
<td>The number of accesses to the cache line that spanned two cache lines. This happens when an unaligned access is made that causes two cache lines to be touched.</td>
</tr>
<tr>
<td>Bytes/L1 Eviction</td>
<td>The number of bytes accessed between cache line evictions.</td>
</tr>
<tr>
<td></td>
<td>The number of accesses (loads plus</td>
</tr>
<tr>
<td>Accesses/L1 Eviction</td>
<td>stores) to a cache line between evictions.</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>L1 Evictions</td>
<td>The number of times a cache line was evicted where this instruction depended on the data in the cache line.</td>
</tr>
<tr>
<td>Accesses</td>
<td>The total number of loads and stores samples for this instruction / function / module.</td>
</tr>
<tr>
<td>Bytes Accessed</td>
<td>The total number of bytes accessed by this instruction / function / module.</td>
</tr>
</tbody>
</table>
Custom Profile

This configuration is intended for advanced users who know what hardware events or combinations of events are important for specific analysis.

It allows a combination of hardware events, timer samples, and instruction-based sampling events. At least one event must be monitored for CodeXL to make the profiling.

Use of a smaller sampling period increases data collection overhead. Since data collection must be performed on the same platform as the test workload, more frequent sampling increases the intrusiveness of event-based profiling, and the sampling process adversely affects shared hardware resources such as instruction and data caches, translation lookaside buffers, and branch history tables. Extremely small sampling periods also can cause system instability. Start off conservatively and slowly decrease the sampling period for an event until the appropriate volume of samples is generated.

**Warning** You can demand too much sampling, causing the system to hang or crash. If this occurs, the easiest solution is to increase the sample intervals for the most popular events.

A factor when choosing the sampling period for an event is the workload behavior. Some workloads are CPU-intensive; other workloads are memory-intensive. Some workloads can be CPU-intensive and require high memory bandwidth to stream data into the CPU. For example, a CPU-intensive application that does not accesses memory very often causes relatively few data-cache miss events. The characteristics of the workload can even vary by phase, where the phase setting up a
computation has a different behavior from the computation phase itself. Thus, the workload behavior determines the frequency of certain kinds of events, requiring changes to the sample period.

The available views for the profile data depend on the events selected. The events are selected through the _Ref371333729 Dialog.
**CPU Profile Session**

A CodeXL CPU profile session collects profile data of a single execution of the profiled application. The profile session results can be viewed and analyzed using CodeXL tabbed views. After making the recommended changes to the profiled application, you can execute the same application and analyze the new bottlenecks after the change.

To create and run a CPU profile session in CodeXL requires **Executing CPU Profile Session** Other options can be configured in the project and global settings:

1. [Changing CPU Profiling Global Settings](#)
2. [Changing CPU Profiling Project Settings](#)
3. [Changing CPU Profiling Configuration Events](#)
Execute a CPU Profile Session

To run a CPU Profile session:

1. Open or create a CodeXL project

2. Select a CPU profile type. See detailed information for each of CodeXL supported CPU Profile Types. In the Active Mode toolbar, click the Profile Mode toolbar button to change the mode to Profiling (Profile > Profile Mode). Select any CodeXL CPU type of profile (Profile > CPU: Assess Performance for example).

3. Use the CPU Profile Setting dialog to configure the profile session parameters.

4. Click the Start CodeXL Profiling toolbar button, ▶, to start profiling.

5. Optional: Pause / Stop the data collection of the profiled
application using the execution toolbar buttons: 

Stopping/pausing the profile session is optional. You can let the application run to its end and then the profiling session will automatically end. For long running applications, pausing allows you to control when the profiling data collection occurs so it matches the stage in your application execution that you want to profile.

6. When the profiled application execution is over, CodeXL processes the data collected, and a profile session window is opened. See the View and Analyze the Profile Session Data Analysis section.

© 2015 Advanced Micro Devices. All rights reserved.
Attach to Process

To start a CPU Profile session of an already running process:

1. Select a CPU profile type. See detailed information for each of CodeXL supported CPU Profile Types. In the Active Mode toolbar, click the Profile Mode toolbar button to change the mode to Profiling (Profile > Profile Mode). Select any CodeXL CPU type of profile (Profile > CPU: Assess Performance, for example).

2. Use the CPU Profile Setting dialog to configure the profile session parameters. Note that if a CodeXL project is not opened when changing the profile settings, then a generic new project is automatically created.

3. Open the “Attach to Process...” dialog from the menu bar (Profile > Attach to Process...). Note that if a CodeXL project is not opened (or created) when the profiling session starts, then a generic new project is automatically created.
4. Select the desired process from the list of attachable processes and click the “Attach” button to start profiling.

5. Optional: Pause / Stop the data collection of the profiled application using the execution toolbar buttons: ■ ■. Stopping/pausing the profile session is optional. You can let the application run to its end and then the profiling session will automatically end. For long running applications, pausing allows you to control when the profiling data collection occurs so that you can match the stage in your application execution that you want to profile.

6. When the profiled application execution is over, CodeXL processes the data collected, and a profile session window is opened. See the View and Analyze the Profile Session Data Analysis section.

Attach to Process dialog
The dialog shows all the processes in the system, which are divided into 2 groups:

1. **Attachable processes**: These are the processes which the current user may attach to. Note that a user may not attach to other users’ processes without having administrative privileges.

2. **Non-attachable processes**: These processes cannot be attached to due to certain privilege protections. These processes are not viewed by default, and are viewed only if the “Show non-attachable processes” checkbox is checked. The list of processes is static, and may be refreshed by clicking the “Refresh” button (or reopening the dialog).
The CPU Profile global settings are set when the project is initially created. These settings affect every session and control how AMD CodeXL displays profile data.

To set the options:

1. In the CodeXL menu, click **CodeXL Options**. The Edit CodeXL Global Settings dialog box is displayed.
2. Select the CPU Profile tab. The CPU Profile tab controls the display of source files and symbolic information. See the description for each of the global profile settings below.
3. Do one of the following:
   - Click **OK** to activate the new options and close the dialog box,
   - Click **Restore Default Settings** to reset the dialog box selections to the system default, or
   - Click **Cancel** to close without committing any changes.
### CodeXL Options

#### General Options

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Additional Debug Symbol Paths</strong></td>
<td>Select least one symbol server.</td>
</tr>
<tr>
<td><strong>Symbol Server Directories</strong></td>
<td>Select directories for the symbol server.</td>
</tr>
<tr>
<td><strong>Block size in bytes to fetch when scrolling through disassembly</strong></td>
<td>The size of the data block that is being automatically fetched when scrolling through disassembly.</td>
</tr>
</tbody>
</table>
Use the CodeXL Project Settings dialog to configure the current project CPU profile settings:

**General Settings Page**

To configure a specific application for profiling, enter the path to the application executable in the ‘Executable Path’ field.

This field can be left empty if you intend to perform System Wide profiling.

To profile a Java application:

- Enter the path of the Java runtime executable in the ‘Executable Path’ field, e.g. “C:\Program Files\Java\jre7\bin\java.exe”
- Enter the path of the java classes top folder in the “Working Directory” field, e.g. “C:\Scimark2”
- Enter the name of the main java class in the command line arguments, e.g. “jnt.scimark2.commandline”

The screenshot below shows an example of configuring a Java application for profiling.

**Profile Settings Page**

Click File -> Project Settings and select the Profile tree node.

The “Profile Type” settings page will help you set the profile type of the current project and the profile scope:
Displays the currently selected **Profile Type**.
### Profile Session Type

The combo box contains both CPU and GPU profile types. Use this to select which type of data you want to collect while running the next profile session.

Use the text below the profile type combo box to get a description of each of the profile types.

### Profile Scope

Defines the scope that the next profile session will monitor.

- **Single Application** – the profile session will collect data for only the profiled application (defined in the “General” settings page).
- **System-Wide Profile** – the profile session will collect data for each of the running processes in the system.
- **System-Wide Profile with focus on application** – the profile session will collect data for each of the running processes in the system. The profile session will also collect call stack details for the focused application (defined in the “General” settings page).

### CPU Profile Settings Page

Click Profile -> Profile Settings to open this settings page.

The CPU Profile settings page contains CPU profile specific configurations.
Checking this option will set CodeXL to
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collect call stack details</strong></td>
<td>Collect call stack details while profiling.</td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>Call stack collection has higher overhead compared to sessions without call stack sampling, because stack unwinding is done whenever a sample is taken for the target process. The unwind operation, combined with the larger amount of data that must be written to the trace file, creates the higher overhead. Also, because of OS limits, the complete call stack might not be available. Set the appropriate “Call stack collection depth” and “Call stack every” options, to balance between performance overhead and statistical accuracy.</td>
</tr>
<tr>
<td><strong>Collect for code executed in</strong></td>
<td>Limits the collection of call stacks only for code running in User space, Kernel space or both (User space and Kernel space).</td>
</tr>
<tr>
<td><strong>Collect call stack every</strong></td>
<td>This parameter will define the frequency of call stack collection.</td>
</tr>
<tr>
<td><strong>Call Stack Collection Depth</strong></td>
<td>Defines the level of depth for the collected call stack. A higher depth will require performance overhead, but the data of the collected call stack will be more accurate. Options are: Minimal / Low / Medium / High / Maximal. Selecting one of these options will sample the call stack with an up to the depth of 2 / 32 / 64 / 128 / 392 levels. <strong>Note:</strong> This option is set separately for Time-Based Sampling and other CPU Profiling session types.</td>
</tr>
</tbody>
</table>
| Reproduce missing call stack info | Perform additional analysis to overcome frame-pointer omission (FPO) in 32-bit apps and lack of unwind info in 64-bit. The profiler will store additional data during the profile session and require more time during post-session processing.  
**Note:** This option is set separately for Time-Based Sampling and other CPU Profiling session types. |
|---|---|
| Collection Schedule | Configures the schedule of collecting data during the profile session execution.  
Throughout entire duration – collect data throughout entire profile session execution  
Start profile with collection paused – the profile session will start with no data collection. Use the “Pause” button to resume data collection on the profiled application.  
Scheduled – set the specific timing for data collection during the execution of the profile session. |
<p>| Start data collection after | Start data collection with a delay of ‘X’ seconds from the start of the profile session |
| End data collection after | When selected, the profile session data collection will end after ‘X’ seconds. |
| Then, terminate the process | When checked, after ‘X’ seconds selected to end the data collection after, the profiled process will be terminated by CodeXL. |</p>
<table>
<thead>
<tr>
<th><strong>Profile hardware scope</strong></th>
<th>Use the tree structure of the existing hardware cores, or the affinity mask, to define the cores on which the profiled data will be collected.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Restore Default Settings</strong></td>
<td>Restore the original default settings.</td>
</tr>
</tbody>
</table>

**CPU Profile Custom Settings Page**

Click Profile -> Profile Settings to open this settings page. Select “Profile -> CPU Profile -> Custom” tree node to go to the custom settings page.
This settings page configures the list of events collected while running a session of type “Custom Profile”. 
Available Events lists all the available events.
Monitored Events lists the events which are selected for monitoring.
Use Add, Remove, Remove All buttons to add the selected event(s), remove the selected event(s), or remove all the selected events.

1. Some events have a checkbox:
   o Usr Enables the collection of user-level samples for an event.
   o Os Enables the collection of operating system-level samples for an event.
   o Edge Enables the edge- and level- detection that control the way an event signal is sensed; this affects the way an event is counted in a performance counter.

The available events depend on the CPU hardware in your system. Note that only one Timer event, IBS all op samples event, or IBS fetch sample event can be monitored at a time.

<table>
<thead>
<tr>
<th>Available Events</th>
<th>Lists the available hardware events for profiling, nested within the configurations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add / Remove buttons</td>
<td>Adds to, or removes from the Monitored Events table the selected available event or configuration.</td>
</tr>
<tr>
<td></td>
<td>A list containing the selected events for data collection in the “Custom Profile” sessions.</td>
</tr>
<tr>
<td></td>
<td>A monitored event item in this list will contain:</td>
</tr>
<tr>
<td></td>
<td>Name - The name of the monitored event.</td>
</tr>
<tr>
<td></td>
<td>Interval - the period of how often a sample is to be taken (for counting occurring events).</td>
</tr>
<tr>
<td></td>
<td>Unit Masks - Used to specify the unit mask setting for the selected event. Each bit set has a different meaning. The Event Settings pane gives you a description of the current setting.</td>
</tr>
</tbody>
</table>
**Monitored Events**

**Usr** - Enables collection of user-level samples for an event.

**Os** - Enables collection of operating system-level samples for an event.

**Edge** - Edge detection and level detection control the way an event signal is sensed; it affects the way an event is accumulated as a count in a performance counter. The occurrence of an event (a hardware condition) is asserted as a physical hardware signal. An event has a duration that can be as short as a single CPU clock cycle or it can be several cycles long.

When **OK** is clicked and the Edge checkbox is checked, the "Custom Profile" configuration is selected automatically, and profiling can start immediately.

---

**Notable Available Events**

<table>
<thead>
<tr>
<th><strong>[E000] Timer event</strong></th>
<th>Hardware APIC timer event. The default is 1ms. The minimum interval is 0.1ms.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[F000] IBS fetch samples</strong></td>
<td>Determines how often an IBS fetch sample is taken. IBS fetch sampling counts completed fetches to determine when the next IBS fetch sample is taken.</td>
</tr>
<tr>
<td></td>
<td>Determines how often an IBS op sample is taken. When the Unit Mask is 0x0 (Count clock cycles), IBS op sampling counts processor cycles to determine</td>
</tr>
<tr>
<td>[F100] IBS all op samples</td>
<td>when the next IBS op sample is taken. When the Unit Mask is 1 (Count ops dispatched), IBS op sampling counts dispatched ops to determine when the next IBS op sample is taken. Dispatched op counting is the preferred mode because profiles produced through cycle counting can be biased by instruction timing.</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Events by Hardware Source</td>
<td>Performance monitoring counter events that vary according to the system's hardware. The individual descriptions are displayed when the event is selected.</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>A hardware event can be added multiple times, but the unit mask, Usr, or Os settings must be different. When an available event or configuration is selected, a description is shown below the list. Most event details can be directly edited within the table by clicking on the detail to change. When an event is selected, the unit mask details are also shown below the Monitored Events table. The presence of a hardware condition is asserted when the event signal is high. Absence of the condition is asserted when the event signal is low. When edge detection is used, each low-</td>
<td></td>
</tr>
</tbody>
</table>
to-high transition of the event signal is counted as a single event; that is, the performance counter is incremented by one. When level detection is used, the level is sensed during each clock cycle, and the performance counter is incremented by one for each cycle during which the event signal is asserted.

The typical setting for an event is level-detect (the Edge box is unchecked). For example, if the event signal represents the "NOT HALTED" CPU state, then the performance counter counts the number of CPU clocks that the CPU spent in the "NOT HALTED" state. This configuration corresponds to the conventional "CPU Clocks Not Halted" event.

Performance counters count either specific processor events or the duration of events. The "Dispatch Stalls" counter event, for example, measures the number of processor cycles when the instruction decoder has stalled for any reason. Edge detection and level detection can be used to determine the average number of cycles per stall by:

- Counting the number of dispatch stall cycles (level detection).
- Counting the number of dispatch stalls (edge detection).
- Dividing the number of stall cycles by the number of stalls.
CPU Profile Data Analysis

The data and source results of a profile session are displayed in the form of tabulated information and annotated source code. Use the CodeXL Explorer to navigate between tabs of the same sessions, and between different sessions.

Profile session results contain several pages that can be accessed through the CodeXL Explorer and by using the context menus for any of the profile session data tables.

The following views can be displayed for each session:

- **CPU Profile Session Explorer**
- **Overview Page**
- **Profile Session Modules View**
- **Profile Session Functions View**
- **Profile Session Call Graph View**
- **Profile Session Source or Disassembly View**
- **Profile Session Display Settings**
- **Import a Profile Session**
- **Save CPU Profile Data**

© 2015 Advanced Micro Devices. All rights reserved.
The CPU Profiling project consists of the CodeXL Explorer pane, the data pane, menu and toolbar commands, and subsequent tabs and views that are displayed during the profile sessions.

The CodeXL Explorer pane lists the profile sessions collected for the current project, organized by profile session type.

The CodeXL Explorer lets you:

- Double-click on a session node in the tree, to open it in the data pane.
- Right-click on a tree node to delete, rename, or import sessions.
- Right-click on a session tree node to browse the folder that contains the session. In the screenshot below you can see the CodeXL explorer for “classic” project. The session in bold is the currently selected profile session, and the profile type in bold font (CPU: Assess Performance) is the profile type for the currently selected session.
The 5 hottest functions usually indicate the most significant performance bottlenecks of your application. Select a different Hotspot Indicator to display hot functions based on different criteria.
Session Views Right-Click Menus

Another way to navigate between the profile session views is to use the session views' right-click menus. For example, in the screenshot below, the function "multiply_matrices" is marked as a hot-spot in the overview page. Right-click on the function to see the following context menu commands:

- **Opening Source Code** opens the source code for "multiply_matrices" to show how the data samples are distributed within the selected function lines.
- **Display in Call Graph View** opens the call graph view to show the "multiply_matrices" call paths.
- **Display in Functions View** opens the functions view to see how the data is distributed within the functions in this module (classic.exe).

![Context Menu](image)

After clicking the "Display in Call Graph" command, the call graph view is opened, with the function "multiply_matrices" selected. When right-clicking on the function, a context menu is opened with the option to either display the function in source code or in functions view.
CodeXL Explorer Navigation Arrows

The CodeXL Explorer arrows record the navigation to each of the session views that were opened. Use these arrows to go back and forward between the displayed session views.
Profile Session Overview Page

The CPU Profile Session Overview Page describes an overview of the data collected during the profile session.

The top section of the overview page contains the following data tables: Hot Spot Functions table, Hot Spot Modules table and Hot Spot Processes table. Note that the Processes table is only displayed when running a system-wide profile session.

The overview data tables can get a zoom-out image of the performance of the executed application. Use these tables to review the summary of hot spot locations in your application. The tables display the data organized by a 'hot spot indicator'.

The bottom section of the overview page contains general information for this session.
Hot Spot Indicator
The screenshot above displays an Assess Performance session overview. The combo box at the right side of the view is a hot-spot indicator combo box. Selecting another field in this combo box causes the data in the view to be re-calculated. The samples column contains the hot spot functions/modules/processes calculated with the new indicator. The screenshot below displays the same session overview, after selecting "CPU clocks not halted (cycles)" in the hot spot indicator combo box. The tables are updated to display hottest functions / modules/ processes according to each of the monitored event / data field listed in the combo box. The table is updated with the CPU clocks sample count and percent, and the "Samples" column tooltip now specifies the CPU Clocks event.

Session Properties Section

The bottom section of the overview page contains the profiled session properties.
Contains the session execution properties:

- Target Path, Working Folder, Data Folder, Command Line Arguments and Environment Variables. These session settings are general CodeXL project settings. Use the CodeXL Project Settings Dialog to configure them.
- CPU Affinity, Call Stack Sampling, System-Wide Profile, Call Stack Information -these settings are CPU profile project settings. Use the CodeXL CPU Profile Project Settings Dialog to configure them.

Profile Details
Contains the following information of the profile session: Profile session type (See CPU Profile Types for details), Profile start-end time, Duration, CPU Family and Total Amount of Processes and Threads. This section also contains a list of monitored events collected for this session.
Navigating to other views

The Overview page describes a zoomed-out image of the profiled session. For more information about bottle necks in your application, use Modules View, Functions View, and Call Graph View. You can also open a Source-Code View to see a source-line level performance data. These views can be opened using the CodeXL Explorer Tree. Each of the tables in the overview page implements a right-click menu that lets you navigate between session views.

Per-Process vs System-Wide Profile Session

A CPU Profile session can be:
· Per-process – Only the target application is profiled. This is the default.
· System-Wide – Samples are taken for all running processes in the system. Use the CodeXL CPU Profile Project Settings Dialog to check / un-check the system-wide profile option.
Profile Session Modules View

This displays a module-by-module detailing of performance data. Use this view to see the distribution of hot spots among the modules of your application. The data displayed in this view is collected for the session profile type. The screenshot below displays an Assess Performance profile session, with a single process (classic.exe). Modules with a high sample count usually indicate performance bottlenecks. Sort the modules table according to a specific counter to highlight potential bottlenecks.
The modules view displays two tables: processes and modules. The following screenshot shows the "classic" project in a
system-wide session. The combo-box on the right side of the view lets you select display processes or modules in the top table.

**Display Modules View by Processes**

The top table displays processes. Select one or more processes in the top table, and the bottom table displays only the modules used by the selected processes.

**Display by Modules**
The top table displays modules. Select one or more modules in the top table, and the bottom table displays only the processes used by the selected modules.

**Modules View Context Menus**

The following two screenshots display the context menus for the modules and processes tables. The context menus lets you:

- Display in Functions View – Displays the selected module/process in the functions view.
- Select "ModuleName.dll" in modules table – Switches to
"Display by Modules" and selects ModuleName.dll in the modules table.

Modules View Display Settings

Use the display filter link to open the Display Settings Dialog and change the properties of the currently displayed session.
Profile Session Functions View

Function view displays list of functions called during profiling of the current session in a table.
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>BHDrv64.sys!psDriver!77738E1</td>
<td>BHDrv64.sys</td>
<td>1</td>
</tr>
<tr>
<td>classic_multiply_matrices(VOID)</td>
<td>AVDTClsMathMul.exe</td>
<td>496</td>
</tr>
<tr>
<td>CreateThread</td>
<td>CComhost.DLL</td>
<td>1</td>
</tr>
<tr>
<td>DEC_SHARE.ECF_CNT</td>
<td>win32base.sys</td>
<td>1</td>
</tr>
<tr>
<td>EnterCriticalSection</td>
<td>win32base.sys</td>
<td>1</td>
</tr>
<tr>
<td>GetAcquireResourceLockExclusiveEx</td>
<td>ntkomm.lee</td>
<td>2</td>
</tr>
<tr>
<td>GetAcquireResourceLockCacheAwareEx</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>GetAllocateProcWithQuotaTag</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>GetAllocateProcWithTag</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>GetFreeProcWithTag</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>GetReleaseResourceLockExclusiveEx</td>
<td>ntkomm.lee</td>
<td>2</td>
</tr>
<tr>
<td>GetReleaseResourceLockCacheAwareEx</td>
<td>ntkomm.lee</td>
<td>2</td>
</tr>
<tr>
<td>GetThreadFunc</td>
<td>sysdir.dll</td>
<td>1</td>
</tr>
<tr>
<td>GetThreadValue</td>
<td>KERNELBASE.dll</td>
<td>2</td>
</tr>
<tr>
<td>GetQueryInformationFile</td>
<td>FILMGR.SYS</td>
<td>1</td>
</tr>
<tr>
<td>GetMultipleHandles</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>GetNativeObjectAttributesForPrivateNamespace</td>
<td>KERNELBASE.dll</td>
<td>1</td>
</tr>
<tr>
<td>GetLastError</td>
<td>KERNELBASE.dll</td>
<td>4</td>
</tr>
<tr>
<td>GetMontiorInfo</td>
<td>use32.dll</td>
<td>1</td>
</tr>
<tr>
<td>HalFlushCommonBuffer</td>
<td>hal.dll</td>
<td>6</td>
</tr>
<tr>
<td>GetSecurityBitmap</td>
<td>win32base.sys</td>
<td>1</td>
</tr>
<tr>
<td>GetSecurityOwners</td>
<td>win32base.sys</td>
<td>1</td>
</tr>
<tr>
<td>intialize_matrices(VOID)</td>
<td>AVDTClsMathMul.exe</td>
<td>1</td>
</tr>
<tr>
<td>intialize_nano_environment</td>
<td>ucrtbase.dll</td>
<td>2</td>
</tr>
<tr>
<td>intialize_stacklimits</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>intialize_threadQueueLockedSpinlock</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>intializeThreadQueueLockedSpinlock</td>
<td>ntkomm.lee</td>
<td>1</td>
</tr>
<tr>
<td>intializeThreadQueuePriorityThread</td>
<td>ntkomm.lee</td>
<td>3</td>
</tr>
</tbody>
</table>

Functions with a high sample count usually indicate performance bottlenecks. Sort the table according to a specific metric to highlight potential bottleneck functions.
This view can be opened in any of the following ways.

- Double-click the **Functions** node of a session. It opens the functions view with Process selected as **All Processes**. If the view was already there, it just opens the view tab.

  ![CodeXL Explorer](image)

- Double-click the process table in the Profile Session Overview Page. This opens the Function view selecting that process. All the functions from that process are listed. This way of opening the Function view is possible only when multiple processes exist.

- Double-click the Module table in Profile Session Overview Page. This opens the Function view selecting only that module.

- Double-click the process table in Profile Session Modules View. This opens the Function view selecting that process.

- Double-click the Module table in Profile Session Modules View. This opens the Function view selecting only that module.

- Selecting the "Display in Functions View" option from context menu of any of the four tables of the Call Graph View. This opens the Function view selecting the function that was selected in Call Graph view on the current table.

**Columns**

This table shows two columns:
· **Function** – Contains the name of the function, if available, or a **NO_SYMBOL** string if the function name could not be determined from the available symbol. There is an icon before each function name in this column indicating if the function is a system function or a function from a user module. It also indicates if the corresponding library is 32-bit or 64-bit.

· **Module** – This column lists the corresponding module of the function. It lists the module name only, without path. Full path information is visible on the tooltips of the corresponding module.

Other optional columns are displayed only if the corresponding events are profiled and those columns are selected for display. For TBP, the only available variable column is timer. Multiple views are possible for EBP or IBS.

Each of the views has a corresponding list of selectable columns for display in the functions view. The combo-box at the beginning of **Column** section of **Display Settings** dialog lists the available views for the selected profile type of the current session.

Below the view combo-box is a list of checkboxes, each of which controls the visibility of a certain column. When the combo-box is checked, the corresponding column is visible; when the combo-box is unchecked, the corresponding column is hidden (see **CodeXL CPU Profile Display Settings Dialog**). For a full list of views and available columns, see the **CodeXL CPU Profile Types** section.

**Filters**

Functions table data can be filtered with one of the three options.

· **Display Settings dialog** – This is opened from the hyperlink above the table, labeled **Display**. This dialog lets you select the view, the displayed column(s) from them. It also let you specify whether to display system modules,
split the data based on core or NUMA nodes (splitting data based on NUMA nodes is available on Windows only), and display the absolute value of data or some percentage with a bar. (For details, see the CodeXL CPU Profile Display Settings Dialog.)

- Module Filter – This is opened from the hyperlink above the table. It shows the number of visible and hidden modules. Clicking on this hyperlink opens the Module Filter dialog. It contains a table with two columns: Modules and Path. Modules column lists the modules name (the icons have the same meaning as described in the function table), and Path contains the path of that module. Both columns are sortable by clicking on the respective heading.

The following screenshot shows an example result after clicking the hyperlink shown in the red boundary above.
This column contains one checkbox per cell which indicates whether to display functions of this module in functions view. There are two checkboxes below the table. **The Select all modules** checkbox is used to select/un-select all the modules currently displayed in the table. This checkbox is always enabled. 

**The Display System modules in Modules Filter** checkbox is enabled only if the Display system modules checkbox is checked (in the Display Settings dialog). When enabled, it shows / hides the system modules in the table. 

On selecting some modules in this table with the checkboxes and clicking the OK button, the functions view is updated. It shows the function only from selected modules, and the link above the table shows the updated count of shown and hidden modules.
**Processes**

Process combo-box lists all the processes in **process-name (process-id)** format. It also has one option called **All Processes**. If a process is selected in this combo-box, all the functions from that process are listed in the table. If **All Processes** is selected, all the functions from all the modules of the current session are displayed in the current table.

**Context Menu**

The Function table provides a context menu. Open it by right clicking on the table. The context menu has four items separated into two groups. The following list describes each of the four items.

- Opening Source Code – Opens the source code or
disassembly view for the currently selected function in the function table.

- Display in Call Graph View – Opens the Call Graph view for the currently selected function in the function table.
- Copy –Copies the selected column(s) in the function table, along with the headers of the column in buffer, for pasting into another application such as Notepad, Microsoft word, Microsoft Excel, etc. See the common context menu section.
- Select All –Selects all the rows and all the columns of the function table. See the common context menu section.
Profile Session Call Graph View

The Call Graph view displays a list of functions with their Call Graph information, including caller-to-called relationship. This can be enabled or disabled from the Call stack collection checkbox of the CPU Profile Project Options for a C++ based session. Call Graph is not supported for a CLR- or Java-based session.
Open Call Graph View
Open this view in one of the following ways.

- Double click the **Call Graph** node of a session. If the Call Graph view was already there, this opens the Call Graph view tab.

- Click the context menu item **Display in Call Graph View** of the functions table in the Profile Session Overview.

- Click the context menu item **Open Call Graph** of **Call Graph** node of a session Page. This opens the Call Graph view with that process selected. All the functions from that process are listed. This way of invoking Function view is possible only when multiple processes exist.
· Click the context menu item **Display in Call Graph View** of the functions table in the Profile Session Function View.

### Tables in Call Graph View

There are four tables in the Call Graph View.

- **Function Table** – Appears at the top of the page and contains the list of all functions (of the selected process or all processes if All Processes option is selected).
This table has the following columns.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function name that appeared in the call stack when the chosen monitored event triggered a sample collection. Based on the configuration of <strong>Display system modules</strong> in the <strong>Display Settings</strong> dialog, this includes the function from system modules.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self Samples</td>
<td>This shows the number of times this function was on the top of call stack when the chosen monitored event triggered a sample collection.</td>
</tr>
<tr>
<td>Deep Samples</td>
<td>Number of times this function was present in the call stack when the chosen monitored event triggered a sample collection.</td>
</tr>
<tr>
<td>% of Deep Samples</td>
<td>Percentage of deep samples out of the total count of the samples gathered [100 \times (\text{Deep Sample}) / (\text{Total Sample collected})]</td>
</tr>
<tr>
<td>No. of Paths</td>
<td>Number of unique paths containing this function.</td>
</tr>
<tr>
<td>Source File</td>
<td>The source files containing the function. The line number is shown in parentheses.</td>
</tr>
<tr>
<td>Module</td>
<td>The name of the module that contains this function. Full path is available on tooltips text of the corresponding cell of the table.</td>
</tr>
</tbody>
</table>
· **Parent Table** – Appears on the left side of the two tables in the middle of the page. It shows the parents (the functions that call the current function) of the selected function in the function table. This table is updated when changing the selection on function table.

<table>
<thead>
<tr>
<th>Parents</th>
<th>Samples</th>
<th>% of samples</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>mainCRTStartup</td>
<td>10160</td>
<td>100.00%</td>
<td>classic.exe</td>
</tr>
</tbody>
</table>

This table has the following columns:

- **Parents**: Lists all the function names that called the current function. In the call stack samples, those parents (direct ancestors) are immediately below the currently selected function.
- **Samples**: The distribution of the currently selected function’s deep-samples between its parents.
- **% of Samples**: Percentage of the paths samples from the total deep samples of the currently selected function.

· **Children Table** – Appears in the right side of two tables in the middle of the page. It shows the Children (the functions called by the current function) of the selected function in the function table. This table is updated on changing the selection on function table.

<table>
<thead>
<tr>
<th>Self + children</th>
<th>Samples</th>
<th>% of samples</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiply_matrices(void)</td>
<td>10150</td>
<td>99.90%</td>
<td>classic.exe</td>
</tr>
<tr>
<td>initialize_matrices(void)</td>
<td>10</td>
<td>0.10%</td>
<td>classic.exe</td>
</tr>
<tr>
<td>[self]</td>
<td>0</td>
<td>0.00%</td>
<td>classic.exe</td>
</tr>
</tbody>
</table>

This table has the following columns:

- **Self + children**: Lists all of the function names called by the
Self + children: current function. In the call stack samples, those children (direct descendants) are immediately above the current function. It also lists one entry called \textit{[self]}. This entry is on the selected function in the function table.

Samples: The distribution of the currently selected function’s deep-samples between its children. For the \textit{[self]} function it is the self-samples (as there are no descendants to the function when it has been sampled itself).

\% of Samples: Percentage of the paths samples from the total deep samples of the currently selected function.

- **Path Graph Table** – Represents the call chain in the form of a tree. In the tree, a child node represents a function called by the function represented by parent node.

<table>
<thead>
<tr>
<th>Function</th>
<th>Self samples</th>
<th>Downstream samples</th>
<th>% of Downstream samples</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>RethandleExceptionChain</td>
<td>10160</td>
<td>10160</td>
<td>100.00%</td>
<td>ntddi.dll</td>
</tr>
<tr>
<td>BaseThreadInitThunk</td>
<td>10160</td>
<td>10160</td>
<td>100.00%</td>
<td>kernal32.dll</td>
</tr>
<tr>
<td>mainCRTStartup</td>
<td>10160</td>
<td>10160</td>
<td>100.00%</td>
<td>classic.exe</td>
</tr>
<tr>
<td>main</td>
<td>10160</td>
<td>10160</td>
<td>100.00%</td>
<td>classic.exe</td>
</tr>
</tbody>
</table>

This table has the following columns:

- **Function**: The names of functions represented in the call chain tree.
- **Self samples**: Number of samples in this function.
- **Downstream-samples**: From this point down in the path, not including the function samples.
- **\% of Downstream-samples**: Percentage of downstream-samples out of all samples for the specific call path.
Show Call Graph selection path – This checkbox controls whether to show a line that connects all the functions in the sub-tree that form a unique path, ending in the selected function. The small number at the end of the path’s line is the length of the path (starting with 0).

The path of the selected function is colored in yellow, while the path of the function on which the cursor is currently hovering above is colored in red.

Filters

- **Processes** – List all the processes in process-name(process-id) format

It also has the All Processes option. If a process is selected in this combo-box, all the functions from that process are listed in the table. If All Processes is selected,
then all the functions from all the modules of the current session are displayed in the current table.

- **Monitored event** – Combo-box at the top right

This lists the set of monitored events observed during current profiling. Only the samples, that were collected when the event selected in this combo-box was triggered, are listed.

- **Display Settings dialog** – Opened from the hyperlink-labeled Display above the table

Only one checkbox, **Display system modules**, is enabled when opened from **Call Graph view**, which is used to show/hide the system modules in the view.

**Context Menu**
The context menu provides the same information for each table discussed above, except Display Call Graph details, which are not available in the Function Table.

Context menu items consist of the following.

<table>
<thead>
<tr>
<th><strong>Open Source Code</strong></th>
<th>Opens the source code or disassembly view for the currently selected function in the current table.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Display in Call Graph</strong></td>
<td>Opens the Call Graph view for the currently selected function in the current table. This menu item is not available for <strong>Function table</strong>.</td>
</tr>
<tr>
<td><strong>Display in Functions View</strong></td>
<td>Opens the Function view for the currently selected function in the current table. This menu item is not enabled for functions that are not present in <strong>Functions View</strong> (this can happen if the function was back-traced from another function (that may have been actually sampled), but not sampled - the monitored event was never triggered directly for that function).</td>
</tr>
<tr>
<td><strong>Copy</strong></td>
<td>Copies the selected column(s) in the current table, along with the headers to the clipboard. See the common context menu section.</td>
</tr>
<tr>
<td><strong>Expand All</strong></td>
<td>Expands all the nodes below the selected function’s node in the tree.</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Collapse All</strong></td>
<td>Collapses all the nodes below the selected function’s node in the tree.</td>
</tr>
</tbody>
</table>
Profile Session Source or Disassembly View

The Source Code / Disassembly view shows the source lines annotated with assembly instructions and sample count for a selected function.

Opening the Source Code View
This view can be opened from one of the following ways:

- **Overview Page**: Double-click on a function in the

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Source Code</th>
<th>Code Bytes</th>
<th>Hotspot Samples</th>
<th>% of Hotspot Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>Ox12312</td>
<td>void classic_multiply_matrices()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>Ox12312</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td></td>
<td>// Multiply the two matrices</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td></td>
<td>for (int i = 0; i &lt; ROWS; i++)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>97</td>
<td></td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td></td>
<td>for (int j = 0; j &lt; COLUMNS; j++)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Ox12312</td>
<td>float sum = 0.0;</td>
<td></td>
<td>3.00%</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>for (int k = 0; k &lt; COLUMNS; k++)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
<td>sum = sum + matrix_a[i][k] * matrix_b[k][j];</td>
<td>475</td>
<td>95.77%</td>
<td>475</td>
</tr>
<tr>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td></td>
<td>matrix_d[i][j] = sum;</td>
<td></td>
<td>3.33%</td>
<td>18</td>
</tr>
<tr>
<td>107</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td></td>
<td>void improved_multiply_matrices()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>for (int i = 0; i &lt; ROWS; i++)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>for (int j = 0; j &lt; COLUMNS; j++)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Select a function from the drop-down list to display its source/disassembly lines with sample counts.
functions table, or right-click on a function, and select **Open Source Code**.

- **Functions View:** Double-click on a function in the functions table, or right-click on a function, and select **Open Source Code**.

- **Call-Graph View:** Right-click on a function in the functions, and select **Open Source Code**.

![Open Source Code View from Overview Page](image1)

![Open Source Code View from Call Graph View](image2)

After selecting a function and clicking **Open Source Code**, a source code view will be created.

A separate instance of source code view is opened for each module. When the source code is created, a node is added in the CodeXL Explorer, called **Source Code**. Under this node, a node will be created for the module’s source code view.
If CodeXL cannot locate the source file, a file selection dialog is opened, and the user will be asked to locate the source file. If the source file will not be located, the source code view will display only disassembly.

On the top of the source code view, a combo-box is created with the list of the module’s functions. The function that was right-clicked will be selected in the functions combo-box, and will be highlighted in the code.
Source Code View opened for "multiply_matrices" function

Data Displayed in Source Code View

The source code consists of a table displaying the following data for each source code line / disassembly line in the current displayed source file:

<table>
<thead>
<tr>
<th>Line</th>
<th>The source code line number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Memory address where each instruction is located.</td>
</tr>
<tr>
<td>Source Code</td>
<td>The source code for the current line.</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Code Bytes</td>
<td>Byte representations of the actual machine instructions.</td>
</tr>
<tr>
<td><strong>Hotspot Samples</strong></td>
<td>The amount of samples collected for the current hotspot indicator event / metric. The hotspot indicator is selected in the hotspot indicator combo-box.</td>
</tr>
<tr>
<td>% of Hotspot Samples</td>
<td>The percentage of samples collected for the current hotspot indicator event / metric. The percentage is calculated relatively to the currently display function (all the function source code lines will sum up to 100%). The hotspot indicator is selected in the hotspot indicator combo-box.</td>
</tr>
<tr>
<td><strong>Samples count for each of the collected / calculated data of the current session</strong></td>
<td>The following columns will contain the amount of samples collected for the monitored events and calculated metrics. Use the Display Settings Dialog to select which of the metrics will be displayed currently in the table.</td>
</tr>
</tbody>
</table>

**Functions combo-box**

The functions combo-box contain the list of the current module’s functions. Selecting a function will highlight the function in the displayed source code, or will display disassembly code for the requested function.
Process IDs combo-box

Thread IDs combo-box

Process ID / Thread ID combo-boxes
The process IDs / Thread IDs combo-boxes are displayed only in cases where the displayed module is multi-process or multi-threaded. Use the process IDs and thread IDs combo-boxes to display the selected process / thread ID collected samples.

**Navigating through the presented data**

The navigation can be done using the mouse wheel, the Page Down\Up keyboard keys or the Down\Up Arrow keyboard keys. While you navigate through a large disassembly, CodeXL will fetch another block of disassembly data as you scroll down and reach the bottom. The size of the data block that is being automatically fetched can be configured in the CPU Profiling global settings. To open the CPU Profiling global settings, go to the CodeXL menu bar and click on Tools->Options. Then, choose the CPU Profile tab, and edit the value in the text box to set the data block size in bytes:
Changing the size of the data block which is being automatically fetched when scrolling through disassembly

Source Code View Display Settings

In order to configure the source code view display settings, click the display settings link on the top of the source code view.
The display settings dialog can be used to select which columns will be displayed in the source code view table, to display the data separately for each core / NUMA node etc’. See more details on the display settings in **Profile Session Display Settings**.

The function "initialize_matrices" is highlighted after being selected in the functions combo-box

**Hotspot Indicator combo-box**

The hotspot indicator combo-box contains the list of monitored events / metrics that is currently displayed in the source code view. Each of these events / metrics can be selected. The selection of an event / metric will update the ‘**Hotspot Samples**’ and ‘**% of Hotspot Samples**’ columns to contain the data collected for this event / metric. Use the hotspot indicator
combo-box to look for the most significant performance bottlenecks of your application.

**Context Menu**

The source code context menu can be opened by right-clicking on one of the items in the table.

The context menu contain the following commands:

- Copy
- Select All
- Expand All
- Collapse All
- Show Code Bytes
- Show Address

Copy the selected row/s in the source code
<table>
<thead>
<tr>
<th><strong>Copy</strong></th>
<th>view table.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Select All</strong></td>
<td>Select all the rows in the source code view table.</td>
</tr>
<tr>
<td><strong>Expand All</strong></td>
<td>Expand all the items in the source code table (show all the disassembly lines)</td>
</tr>
<tr>
<td><strong>Collapse All</strong></td>
<td>Collapse all the items in the source code table (hide all the disassembly lines)</td>
</tr>
<tr>
<td><strong>Show Code Bytes</strong></td>
<td>Show / hide the code bytes column in the source code view table</td>
</tr>
<tr>
<td><strong>Show Address</strong></td>
<td>Show / hide the address column in the source code view table</td>
</tr>
</tbody>
</table>
Profile Session Display Settings

This dialog can be used to configure the display of the CPU profile for the currently opened sessions.
Click on the display link, and open the display settings dialog.
Columns

This section lets you select the group of columns displayed by
the current profiled session. Each of the groups in this combo-
box is a set of data columns for the current displayed session. 
The list of groups in this check box depends on the current 
session profile type.

**Display System Modules**

Display or hide the data collected from system modules. This 
option is global and affects the content of each of the opened 
views.

**Show Percentage Bars**

When this option is unchecked, the profile session data is 
displayed as values. When checked, the data is displayed as 
percentages, and percentage bars are displayed within the 
tables.

**CPU Cores**

Select the list of cores for which the results are to be displayed.

**Separate Data Per Core / NUMA**

Display a separate column for each core / NUMA. 
After setting the options in this dialog, click OK. CodeXL 
updates each of the opened sessions with the global options 
(percentage and system dll display); it also updates the current 
view with the local settings. The display filter link on the top of 
the views contains the current display settings. For example: 
The following Functions view display settings are: "Branch 
Assessment, System Modules Hidden, Percentages, 2 Cores"
This means that the list of columns displayed for the current 
session are the Branch Assessment data columns. In the 
modules table, below the displayed columns, are the columns 
related to branch assessment. The data is displayed in 
percentages, and only two cores are selected.
Modules Filter Dialog

The below screenshot contain the display settings link for the Functions View. The right section of the string contain the amount hidden and shown modules in the functions view. Click on the right link to open the modules filter dialog.
Display System modules in Modules Filter

When this box is checked, the table above it includes the system modules. When un-checked, these modules are not shown. This option is enabled when system modules are hidden in all views.

The modules table icons represent the modules type.
<table>
<thead>
<tr>
<th></th>
<th>32/64 bit user module</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32/64 bit system module</td>
</tr>
</tbody>
</table>
Importing Profile Data

The profile data in a CodeXL EBP file can be imported into a CodeXL project. A new session is created for the profile data.

To import profile data:

1. If a CodeXL project is not already opened, create a new CodeXL project or open an existing CodeXL project.
2. Right-click the project name in the CodeXL Explorer pane. The shortcut menu is displayed.
3. Click **Import Session**...
4. Select the CodeXL EBP file to be imported into the
5. A new session is created and displayed for the imported project.
**Profile Data**

### 5 Hottest Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Samples</th>
<th>% of Hotspot Samples</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_transform internal</td>
<td>9,102</td>
<td>27.68%</td>
<td>ScimarkStable.exe</td>
</tr>
<tr>
<td>LU_factor</td>
<td>8,742</td>
<td>20.00%</td>
<td>ScimarkStable.exe</td>
</tr>
<tr>
<td>random_normalDouble</td>
<td>7,323</td>
<td>17.44%</td>
<td>ScimarkStable.exe</td>
</tr>
<tr>
<td>SOR_execute</td>
<td>6,480</td>
<td>15.34%</td>
<td>ScimarkStable.exe</td>
</tr>
<tr>
<td>SparseComPaXmatmul</td>
<td>6,980</td>
<td>14.95%</td>
<td>ScimarkStable.exe</td>
</tr>
</tbody>
</table>

### 5 Hottest Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Samples</th>
<th>% of Hotspot Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScimarkStable.exe</td>
<td>41,969</td>
<td>96.86%</td>
</tr>
<tr>
<td>ucrtbase.dll</td>
<td>42</td>
<td>0.10%</td>
</tr>
<tr>
<td>ntdll.dll</td>
<td>13</td>
<td>0.03%</td>
</tr>
<tr>
<td>stdlib.dll</td>
<td>2</td>
<td>0.00%</td>
</tr>
<tr>
<td>libmsvtss.s</td>
<td>1</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

### Execution

- **Target Path**: C:\Codex\Testing\CPUProfiling\Apps\ScimarkStable.exe\Release\ScimarkStable.exe
- **Working Directory**: C:\Codex\Testing\CPUProfiling\Apps\ScimarkStable.exe\Release
- **Profile Session Directory**: C:\Users\pedrag\AppData\Roaming\AMD\Codel\Codel\Matrix\Behind\Profile\Output\Mar-29-2016_12-01-48
- **Command Line Arguments**:
  - CPU Affinity: 0
  - Profile Scope: Single Application
  - Call Stack Sampling: True
  - Call Stack Information: User mode, Depth - 128, Frequency - per 1 samples, FPO - On

### Profile Details

- **Profile Session Type**: Time-based Sampling
- **Profile Start Time**: Mar-28-2016_12-01-48

*The 5 Hottest Functions usually indicate the most significant performance bottlenecks of your application. Select a different Hotspot Indicator to display hottest functions.*
Saving Profile Data

You can save the profile data in the tables of different profile session views for later analysis. This data is save in the CSV files.

To save profile data:

1. Right click the table having the profile data to be saved.
   A context-menu is displayed.
2. Click ‘Select All’
3. Right-click and click ‘Copy’
4. Open notepad, and paste the copied profile data.
5. Save the file as a CSV file (with .csv extension).

6. To view the profile data, open the CSV file in a spreadsheet program, such as Microsoft Excel or OpenOffice.org Calc.
<table>
<thead>
<tr>
<th>Process</th>
<th>PID</th>
<th>Ret Inst</th>
<th>Ret branch</th>
<th>Ret misp</th>
<th>Branch rate</th>
<th>Mispredict rate</th>
<th>Mispredict ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>classic.exe</td>
<td>6924</td>
<td>12178</td>
<td>5044</td>
<td>52</td>
<td>2.39765263</td>
<td>0.01945573</td>
<td>0.081158215</td>
</tr>
<tr>
<td>devenv.exe</td>
<td>6752</td>
<td>2482</td>
<td>6662</td>
<td>322</td>
<td>8.65322495</td>
<td>0.73136961</td>
<td>9.21796038</td>
</tr>
<tr>
<td>devenv.exe</td>
<td>6496</td>
<td>2194</td>
<td>5668</td>
<td>302</td>
<td>6.40619469</td>
<td>0.89532214</td>
<td>7.6614604</td>
</tr>
<tr>
<td>devenv.exe</td>
<td>11864</td>
<td>1944</td>
<td>5036</td>
<td>238</td>
<td>8.22721672</td>
<td>0.47905275</td>
<td>2.03038311</td>
</tr>
<tr>
<td>System</td>
<td>4</td>
<td>908</td>
<td>708</td>
<td>22</td>
<td>2.10854959</td>
<td>0.05504386</td>
<td>0.70373464</td>
</tr>
<tr>
<td>Explorer.EXE</td>
<td>2276</td>
<td>602</td>
<td>1098</td>
<td>72</td>
<td>3.63061166</td>
<td>0.22400974</td>
<td>1.74279916</td>
</tr>
<tr>
<td>devenv.exe</td>
<td>4896</td>
<td>270</td>
<td>678</td>
<td>82</td>
<td>4.48354912</td>
<td>0.62786824</td>
<td>2.6894455</td>
</tr>
<tr>
<td>CodeXL.exe</td>
<td>1752</td>
<td>224</td>
<td>446</td>
<td>38</td>
<td>1.18774331</td>
<td>0.11679089</td>
<td>1.60826468</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>436</td>
<td>190</td>
<td>466</td>
<td>30</td>
<td>0.52030238</td>
<td>0.02903076</td>
<td>0.12373409</td>
</tr>
<tr>
<td>chrome.exe</td>
<td>10660</td>
<td>182</td>
<td>396</td>
<td>54</td>
<td>1.84686875</td>
<td>0.25797981</td>
<td>0.8790139</td>
</tr>
<tr>
<td>SMC.exe</td>
<td>1548</td>
<td>164</td>
<td>374</td>
<td>32</td>
<td>1.78216124</td>
<td>0.11065934</td>
<td>0.35541886</td>
</tr>
<tr>
<td>conhost.exe</td>
<td>3492</td>
<td>142</td>
<td>346</td>
<td>20</td>
<td>1.56762242</td>
<td>0.15875152</td>
<td>0.44774246</td>
</tr>
<tr>
<td>optray.exe</td>
<td>4544</td>
<td>134</td>
<td>240</td>
<td>56</td>
<td>0.9517542</td>
<td>0.35684213</td>
<td>2.2316463</td>
</tr>
<tr>
<td>System Idle</td>
<td>0</td>
<td>132</td>
<td>236</td>
<td>100</td>
<td>0.37261903</td>
<td>0.19880953</td>
<td>3.43956235</td>
</tr>
<tr>
<td>my_data</td>
<td>292</td>
<td>102</td>
<td>224</td>
<td>12</td>
<td>0.94220124</td>
<td>0.05918552</td>
<td>0.19184719</td>
</tr>
</tbody>
</table>
CodeXL provides a command line interface utility for users who prefer to use command interpreters like `cmd.exe` on Windows and `bash` on Linux. This CLI utility will be used to collect and analyze the profile data. It can also be used from a batch file or a test script.

**Usage:** CodeXLCpuProfiler.exe command <options> [InputApplication] [InputApplication’s command line arguments]

Following commands are supported:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect</td>
<td>Run the given input application and collect cpu profile samples.</td>
</tr>
<tr>
<td>report</td>
<td>Process the given cpu profile data file and generate a cpu profile report in CSV format.</td>
</tr>
</tbody>
</table>

Following options are supported with `collect` command:

- `-m <profile type>`
  Predefined profile type to be used to collect samples. Supported profile types are:
  - `tbp`: Time-based Sampling
  - `assess`: Assess Performance
  - `branch`: Investigate Branching
  - `data_access`: Investigate Data Access
  - `ibs`: Instruction-based Sampling
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>inst_access</strong>: Investigate Instruction Access</td>
<td></td>
</tr>
<tr>
<td><strong>l2_access</strong>: Investigate L2 Cache Access</td>
<td></td>
</tr>
<tr>
<td><strong>clu</strong>: Cache Line Utilization (Windows only)</td>
<td></td>
</tr>
<tr>
<td><strong>-T &lt;n&gt;</strong></td>
<td>Sampling interval for custom time-based profiling (tbp). Sampling interval &lt;n&gt; is specified in milliseconds.</td>
</tr>
<tr>
<td><strong>-d &lt;n&gt;</strong></td>
<td>Profile duration in seconds.</td>
</tr>
<tr>
<td><strong>-o &lt;file name&gt;</strong></td>
<td>Base name of the output file. If this option is specified, default path will be used. The default path will be %Temp%\Codexl-CpuProfile-&lt;timestamp&gt; on Windows and /tmp/Codexl-CpuProfile-&lt;timestamp&gt; on Linux.</td>
</tr>
<tr>
<td><strong>-p &lt;PID,PID,..&gt;</strong></td>
<td>Profile existing processes (processes to attach to). Process IDs are separated by comma.</td>
</tr>
<tr>
<td><strong>-a</strong></td>
<td>System Wide Profile (SWP). If this flag is not set then the command line tool will profile only the launched application or the PIDs attached.</td>
</tr>
</tbody>
</table>
| **-G** | Enable callstack sampling with default Callstack collection sampling interval and Unwind Depth. The default values are:
| Callstack Collection | Sampling Interval: 1  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unwind Depth: 128</td>
<td></td>
</tr>
</tbody>
</table>

Enable callstack sampling with user defined callstack collection sampling interval, unwind depth, scope and FPO values. Scope and FPO are Windows only options.

**Windows only**

Scope should contain one of these options:

- **user**: Collect callstacks only for code executed in user space.
- **kernel**: Collect callstacks only for code executed in kernel space.
- **all**: Collect callstacks for code executed in user space and kernel space.

To collect missing frames due to omission of frame pointers by compiler.

- **fpo**: Collect missing callstack frames.
- **nofpo**: Do not collect missing callstack frames. (default)

<table>
<thead>
<tr>
<th>Core Affinity Mask</th>
<th>Core Affinity Mask. Default affinity is all the available cores. In System-wide profiling, samples are collected only from these cores. In Per-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Process profile, processor affinity is set for the launched application.</td>
<td></td>
</tr>
<tr>
<td><code>-f</code></td>
<td>Profile the children of the launched application (i.e. processes launched by the profiled application).</td>
</tr>
<tr>
<td><code>-b</code></td>
<td>Terminate the launched application after profile collection.</td>
</tr>
<tr>
<td><code>-s &lt;n&gt;</code></td>
<td>Start profiling after the specified delay duration <code>&lt;n&gt;</code> in seconds.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Print version string.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>Specify the working directory. Default will be the path of the launched application.</td>
</tr>
<tr>
<td><code>-h</code></td>
<td>Displays this help information.</td>
</tr>
<tr>
<td><code>-C &lt;Custom profile&gt;</code></td>
<td>Path to the custom profile XML file.</td>
</tr>
</tbody>
</table>

Following options are supported with **report** command.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-i &lt;file name&gt;</code></td>
<td>Input file name. Either the raw profile data file (<strong>.prd</strong> on Windows and <strong>.caperf</strong> on Linux) or the processed data file (<strong>.ebp</strong>) can be specified.</td>
</tr>
</tbody>
</table>
| `-o <output dir>` | Output directory in which the processed data file (ebp and imd) will be created. The default path will be `%Temp%\<base-name-of-input-file>` on Windows and `/tmp/\<base-`
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V &lt;view xml&gt;</td>
<td>Specify the View configuration XML file. All the raw data will be reported.</td>
</tr>
<tr>
<td>-R &lt;section,..&gt;</td>
<td>Specify the report sections to be generated. Supported report sections are: all: Report all the sections. overview: Report Overview section. process: Report process details. module: Report module details. callgraph: Report callgraph. Options process and module are mutually exclusive. Options module and callgraph are mutually exclusive.</td>
</tr>
<tr>
<td>-e</td>
<td>Specify the event index for which callgraph will be generated. This event is also used to find the hot functions in the Overview section.</td>
</tr>
<tr>
<td>-I</td>
<td>Ignore samples from System Modules.</td>
</tr>
<tr>
<td>-P</td>
<td>Show Percentage.</td>
</tr>
</tbody>
</table>

**Examples**

- Launch the application classic.exe and collect Time-based profile (TBP) samples:
  ```
  CodeXLCpuProfiler.exe collect -m tbp -o c:\Temp\cpuprof-tbp classic.exe
  ```
- Launch the application classic.exe and collect **assess performance** profile samples for the duration of 10 seconds:
  ```
  CodeXLCpuProfiler.exe collect -m assess -o c:\Temp\cpuprof-assess -d 10 classic
  ```
- Launch the application classic.exe and collect Instruction Based Sampling (IBS) samples in **System wide profile**
(SWP) mode:

CodeXL_CpuProfiler.exe collect -m ibs -a -o c:\Temp\cpuprof-ibs-swp classic

- Collect Time-based profile samples in System wide profile mode for the duration of 10 seconds:
  CodeXL_CpuProfiler.exe collect -m tbp -a -o c:\Temp\cpuprof-TBP-swp -d 10

- Launch the application classic.exe and collect Time-based profile (TBP) samples. Also enable collecting callstack samples whenever the TBP samples are collected:
  CodeXL_CpuProfiler.exe collect -m tbp -G -o c:\Temp\cpuprof-tbp classic.exe

- Print help:
  CodeXL_CpuProfiler.exe -v

- Print version string:
  CodeXL_CpuProfiler.exe -h

- Once the raw cpu profile data file is generated, CodeXL_CpuProfiler report command can be used to generate CSV report from that raw data file:
  CodeXL_CpuProfiler.exe report -i c:\Temp\cpuprof-tbp.prd -o c:\Temp\cpuprof-tbp-out

Once the raw CPU profile data file is generated using the command line utility, the results can be viewed within CodeXL using the **Import Session** command in the **CodeXL Explorer**.
Profile Configuration File Format

This section describes the XML configuration file passed to –C option. This data collection configuration file describes how CodeXL CPU Profiler is to be configured for data collection. Pre-defined configurations are provided with CodeXL. Advanced users can create their own data collection configuration by writing an XML file. A data configuration XML file contains only one configuration.

The <dc_configuration> and </dc_configuration> tags mark the beginning and end of configuration information within a data collection configuration file.

```
<dc_configuration>
...
</dc_configuration>
```

A collection configuration contains <tbp>, <ebp>, <ibs> or <clu> elements. Each element describes a data collection configuration of type indicated by its element name. A collection configuration should contain one or more non duplicate elements.
The `<tbp>` and `</tbp>` tags mark the beginning and end of a time-based profile data collection configuration. This element has the following attributes:

<table>
<thead>
<tr>
<th>name</th>
<th>Configuration name (string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>Sampling interval in milliseconds (float)</td>
</tr>
</tbody>
</table>

Sample TBP configuration:

```xml
<dc_configuration>
    <tbp name="time based profile" interval="10.0">
        <tool_tip> Find program hotspots
        <description> Configuration to identify where an application is spending its time
        </description>
    </tbp>
</dc_configuration>
```
EBP Collection Configuration

The `<ebp>` and `</ebp>` tags mark the beginning and end of an event-based profile data collection configuration. This element has the following attributes:

<table>
<thead>
<tr>
<th>name</th>
<th>Configuration name (string)</th>
</tr>
</thead>
</table>

The sampling events are specified using `<event>` element. One or more events can be specified. The tag `<event> </event>` mark the beginning and end of an event element. It describes how an individual event counter is configured for data collection.

An event has the following attributes:

<table>
<thead>
<tr>
<th>select</th>
<th>Event select value (integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mask</td>
<td>Unit mask value (integer)</td>
</tr>
<tr>
<td>os</td>
<td>Enables OS sampling (Boolean)</td>
</tr>
<tr>
<td>user</td>
<td>Enables user level sampling (Boolean)</td>
</tr>
<tr>
<td>count</td>
<td>Sampling period (integer)</td>
</tr>
<tr>
<td>edge_detect</td>
<td>Enables edge detect when counting even (Boolean). This is optional.</td>
</tr>
<tr>
<td>host</td>
<td>Enables host mode event counting (Boolean). This is optional.</td>
</tr>
<tr>
<td>guest</td>
<td>Enables guest mode event counting (Boolean). This is optional.</td>
</tr>
</tbody>
</table>
The values must be validated against the events and specific capabilities supported by the measurement platform. The maximum number of events depends upon the number of counters supported by the platform on which the measurements are taken.

Sample EBP configuration:

```xml
<dc_configuration>
  <ebp name="event based profile">
    <event select="C0" mask="00" os="T" user="T" count="250000"/>
    <event select="76" mask="00" os="T" user="T" count="250000"/>
  </ebp>
  <tool_tip> HW PMC events based profiling </tool_tip>
  <description> Configuration to find potential issues for investigation </description>
</dc_configuration>
```
IBS Collection Configuration

The `<ibs>` and `</ibs>` tags mark the beginning and end of an instruction-based sampling (IBS) data collection configuration. This element has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Configuration name (string)</td>
</tr>
<tr>
<td>fetch_sampling</td>
<td>Enables IBS fetch sampling (Boolean)</td>
</tr>
<tr>
<td>fetch_max_count</td>
<td>Maximum periodic fetch count/sampling period (integer)</td>
</tr>
<tr>
<td>op_sampling</td>
<td>Enables IBS Op sampling (Boolean)</td>
</tr>
<tr>
<td>op_max_count</td>
<td>Maximum periodic op count/sampling period (integer)</td>
</tr>
<tr>
<td>op_cycle_count</td>
<td>Count clock cycles (Boolean).</td>
</tr>
</tbody>
</table>

Sample IBS configuration:

```
<dc_configuration>
  <ibs name="instruction based sampling"
    fetch_sampling="T"
    op_sampling="T"
    fetch_max_count="250000"
    op_max_count="250000"
  </ibs>
  <tool_tip> Collect data using IBS
</tool_tip>
```
<description> Configuration to attribute samples to instructions precisely </description>
</ibs>
</dc_configuration>
CLU Collection Configuration

The `<clu>` and `</clu>` tags mark the beginning and end of a Cache line utilization (CLU) data collection configuration. This element has the following attributes:

<table>
<thead>
<tr>
<th>name</th>
<th>Configuration name (string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clu_sampling</td>
<td>Enables CLU sampling (Boolean)</td>
</tr>
<tr>
<td>clu_max_count</td>
<td>Maximum sampling period (integer)</td>
</tr>
</tbody>
</table>

Sample CLU configuration:

```xml
<dc_configuration>
  <clu name="cache line utilization"
        clu_sampling="T"
        clu_max_count="250000"
  
  <tool_tip> measure of cache line utilization of L1 data cache </tool_tip>
  
  <description> Configuration to find potential issues related to data locality and data access pattern. </description>

  </clu>
</dc_configuration>
```
Miscellaneous tags

The tags `<tool_tip>` and `</tool_tip>` mark the beginning and end of a short **tool tip** description of a configuration. The text between the tags is the tool tip description. It is usually only a few key words with no line breaks.

The tags `<description>` and `</description>` mark the beginning and end of a short description of a configuration. The text between the tags is the description. It is usually only a few sentences long and may contain line breaks. Line breaks will be replaced by spaces and runs of spaces will be replaced by single space character.

The valid values for a Boolean attribute is one of the strings “T” or “F” which denotes TRUE and FALSE respectively. The default value for Boolean attribute is “F”
CPU Profile C/C++ Inline Functions

CodeXL CPU Profiler reports functions that are inlined and attributes samples which belong to the code region of the instances of inlined functions. In the CodeXL session views the inlined functions are displayed with the word “[inlined]” prefixed to the function name. CPU Profiler can identify the inlined functions only if the target application binary includes the information about the instances of inlined functions. If the target application binary does not contain the information about the inlined functions, then CodeXL CPU Profiler would attribute the samples to the non-inlined caller function.

Overview, Functions View, Call Graph View and Source Code view display information about inlined functions.
On Linux platforms, CodeXL CPU Profiler attributes and reports the samples that belong to Procedure Linkage Table (PLT) section. PLT information is generated by compiler, which is used by dynamic linker/loader to link the application with its dependent dynamic libraries. Samples that are attributed to PLT instructions are reported against “[PLT] <function-name>” symbol. For example, an application calling a library function rand() will have an corresponding entry for this function in the PLT section. Samples attributed to this PLT entry will be shown as “[PLT] rand” and samples due to actual rand() function will be attributed to “rand()” function entry in the session views.
VMware Workstation

CPU Profiler supports TBP and EBP on Guest OS running on VMware Workstation 11.0 or later. It is always recommended to use latest version of VMware Workstation. Recent AMD Carrizo processor is not yet supported by VMware Workstation 11.1.x.

To run TBP within guest OS, no additional configuration needed in host OS or guest OS.

To run EBP within guest OS, please ensure the following settings are done:

- Enable Virtualization or SVM (AMD-V) in BIOS settings before booting the host OS.
- Enable AMD-V in guest OS VM settings. Edit virtual machine settings > Hardware > Processors> Virtualization engine > Enable “Virtualize AMD-V/RVI”
- Enable vPMC in guest OS VM settings. Edit virtual machine settings > Hardware > Processors> Virtualization engine > Enable “Virtualize CPU performance counters”

Known Issues on Windows 7 Host OS:

- When CPU Profiler EBP is running on Windows 7 host OS and a Linux guest OS is launched, crash is observed on Windows 7 due to VMware driver.
- If EBP is performed on Windows 7 host OS and EBP is performed on Linux guest OS simultaneously, then crash is observed on Windows 7 due to VMware driver.

These scenarios work fine when host OS is Windows 8, 8.1 and 10.
Microsoft Hyper-V

CPU Profiler supports TBP on Windows Host OS, Windows/Linux Guest OS running on Hyper-V.

CPU Profiler supports EBP only on Windows 10 Host and Windows 10 Guest OS (running on Windows 10 Host OS). Please enable Virtualization or SVM (AMD-V) in BIOS settings before booting the host OS.
CPU Profiler supports only TBP on Windows/Linux OS running on Xen hypervisor.
Linux KVM

CPU Profiler supports only TBP on Windows/Linux OS running on KVM hypervisor.
CPU Profiler control APIs allow user to limit the profiling scope to a specific portion of the code within the target application. Usually, when the profiling done, it captures the samples for the complete application, i.e. start of execution till end of the application execution. The control APIs can be used to enable the profiler only for a specific part of application, e.g. a CPU intensive loop, a hot function, etc. The target application need to be recompiled after adding the control APIs within the application.

The control APIs:

// To pause CPU profiling, call one of the below two APIs.
int amdtStopProfiling(amdtProfilingControlMode); // Set mode to AMDT_CPU_PROFILING
int amdtStopProfilingEx(void);

// To resume CPU profiling, call one of the below two APIs.
int amdtResumeProfiling(amdtProfilingControlMode); // Set mode to AMDT_CPU_PROFILING
int amdtResumeProfilingEx(void);

CPU Profiler only profiles the code within each Resume, Stop APIs pair. Refer “CPU Profiler Tutorial” on how to use these APIs, compile your target application and profile only the desired part of code.
CPU Profile IMIX report generation

If you are interested in the hot instructions for a target application then in such IMIX report will be useful. IMIX report generates report on hotspot instructions summary.

Sample IMIX report summary:

<table>
<thead>
<tr>
<th>Disassembly</th>
<th>Samples Percentage</th>
<th>Samples Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [rsp+08h],rcx</td>
<td>5.54</td>
<td>111</td>
</tr>
<tr>
<td>retnq</td>
<td>3.54</td>
<td>71</td>
</tr>
<tr>
<td>mov rax,[rsp+08h]</td>
<td>3.09</td>
<td>62</td>
</tr>
<tr>
<td>sub rsp,28h</td>
<td>2.79</td>
<td>56</td>
</tr>
<tr>
<td>mov [rsp+18h],r8d</td>
<td>2.44</td>
<td>49</td>
</tr>
<tr>
<td>mov [rsp+10h],edx</td>
<td>2.34</td>
<td>47</td>
</tr>
</tbody>
</table>

Only CPU Profiler CLI interface supports IMIX report generation. Use option ‘-R imix’ in CLI during report generation to generate IMIX information. Refer CodeXLCpuProfiler help (-h) option to get more details.
The GPU Profiler component in CodeXL is a performance analysis tool that gathers data from the API run-time and GPU for OpenCL™, HSA or DirectCompute applications. This information can be used by developers to discover bottlenecks in the application and find ways to optimize the application's performance. The GPU Profiler can also be used as a command-line tool.

Features of the GPU Profiler include:

- Measure the execution time of an OpenCL™, HSA or DirectCompute kernel.
- Query the hardware performance counters on an AMD Radeon™ graphics card.
- Use the AMDTActivityLogger API to trace and measure the execution of segments in the program.
- Compare multiple runs (sessions) of the same or different programs.
- Store the profile data for each run in a text file.
- Display the IL/HSAIL and ISA (hardware disassembly) code of the kernel for OpenCL™ kernels and DXASM code for DirectCompute kernels.
- Show a timeline (including data transfer and kernel dispatch) and an API trace for OpenCL™ or HSA programs.
- Calculate and display kernel occupancy info, which estimates the number of in-flight wavefronts on a compute unit as a percentage of the theoretical maximum number of wavefronts that the compute unit can support.

The following screenshots display the results of the Application Timeline Trace and Performance Counters sessions for an
OpenCL™ program.
Using the GPU Profiler

The GPU Profiler provides two modes:

| Application Timeline Trace | This mode provides a high-level overview of:  
| 1. An API Trace, showing all OpenCL™ of HSA APIs called by the application.  
| 2. A timeline showing the call sequence and duration of all OpenCL™ or HSA APIs as data transfers (OpenCL™ only) and kernels.  
| 3. A set of Summary Pages, providing a set of statistics for the application.  
| For more information, see:  
| Application Timeline Trace Session  
| Application Timeline Trace Summary Pages |

| Performance Counters | This mode collects performance counters from each device by either and OpenCL™, HSA or DirectCompute application. It also displays statistics from the shader compiler. The performance counters and statistics can be used to discover bottlenecks. This mode also can display the kernel source code, the generated IL code, for an OpenCL™, DirectCompute kernel dispatch.  
| For more information, see:  
| GPU Profiler Performance Counters Session  
| GPU Profiler Code Viewer |

For OpenCL™ programs, both profiling modes can also generate Kernel Occupancy information for each kernel dispatched to a GPU. For HSA applications Kernel Occupancy information is only available in Performance Counter mode. For more information, see GPU Profiler Kernel Occupancy Viewer and GPU Profiler Kernel Occupancy Viewer

To use the GPU Profiler:

1. Create a new project, or open an existing project.
2. Switch to Profile Mode in CodeXL.
   You can switch to Profile Mode using the CodeXL Toolbar or the CodeXL menu.

3. In Profile Mode, use the menu to select one of the above two modes.

4. Start the profile session using the "Start CodeXL Profiling" toolbar or menu item. Profiling results are gathered while the application is running.

Once the application terminates, a new session is added to the CodeXL Explorer. The results of the profile also are displayed by CodeXL.

The following links provide more information on the features available in the GPU Profiler.

- Application Timeline Trace Session
- GPU Profiler Performance Counters Session
- GPU Profiler Summary Pages
- GPU Profiler Code Viewer
- GPU Profiler Kernel Occupancy Viewer
- GPU Profiler Kernel Occupancy
- GPU Profile Project Settings
- Description of Output Files
- Description of Configuration Files
- Using the Command-Line Interface
- AMDTActivityLogger Library

© 2015 Advanced Micro Devices. All rights reserved.
The following screenshot shows the timeline and API trace data for a profile session. To get the .atp file of the result, right-click the session in the CodeXL Explorer, and select "Open Containing Folder" from the menu. See Description of Output Files for a detailed description of the format of this file.
From the application trace data, you can:

- Discover the high-level structure of the application with
the Timeline View. For OpenCL™ programs, you can use this view to determine the number of OpenCL™ contexts and command queues created, as well as the relationships between these items in the application.

- Determine if an OpenCL™ application is bound by kernel execution or data transfer operations.
- View and debug the input parameters and output results for all API calls made by the application with the API Trace View.
- View and analyze the performance for sections in the program, using AMDTActivityLogger.

The panel is divided into two sections. The upper section shows the application timeline, the lower section shows the API trace.

**Application Timeline Trace**

The application timeline provides a visual representation of the execution of the application. Along the top of the timeline is the time grid, which shows the total elapsed time, in milliseconds, of the application. Timing begins when the first OpenCL™ or HSA call is made by the application; it ends when the final OpenCL™ or HSA call is made.

Directly below the time grid, each host (OS) thread that made at least one OpenCL™ or HSA call is listed. For each host thread, the API calls are plotted along the time grid, showing the start time and duration of each call. Below the host threads, an API-specific tree shows device-specific information. For OpenCL™, the tree shows all contexts and queues created by the application, along with data transfer operations and kernel execution operations for each queue. For HSA, the tree shows all kernels dispatched to a particular device.

The Timeline View can be useful for debugging your OpenCL™ application. Using the data displayed in the timeline, you can:

- Easily confirm that the high-level structure of your application is correct. By examining the timeline, you can
verify that the number of queues and contexts created matches your expectations for the application.

- Confirm that synchronization has been performed properly in the application. For example, if kernel A execution is dependent on a buffer operation and outputs from kernel B execution, then kernel A execution appears after the completion of the buffer execution and kernel B execution in the time grid. It can be hard to find this type of synchronization error using traditional debugging techniques.

- Confirm that the application has been using the hardware efficiently. For example, the timeline shows that non-dependent kernel executions and data transfer operations occur simultaneously.

**Navigating the Application Timeline**

The application timeline provides many ways to view and analyze the profile result: through zooming, navigating, and expanding/collapsing.

**Zooming**

When first opened, the timeline view is fully zoomed out: the entire application timeline is visible in the timeline. It can be useful to zoom in to specific parts of the timeline in order to better understand the profiling data. As you zoom in and out, the time grid at the top changes to display the timestamp of the currently displayed timeline subsection.

1. **Manual zoom** – Use the mouse wheel to manually zoom in and out. Roll the mouse wheel up to zoom in, and down to zoom out. If using a mouse not equipped with a mouse wheel, or if you prefer to use the keyboard, you can use the plus key to zoom in, and the minus key to zoom out. The current zoom pivot point (displayed as a vertical line over the entire timeline) represents the point in the timeline into which the view is zoomed. The zoom pivot
point tracks the mouse cursor as it moves over the timeline. The current timestamp represented by the zoom pivot is displayed as a hint in the grid displayed at the top of the timeline.

2. **Zoom into specific API call** – To zoom into a particular API call, double-click the API call in the API Trace list.

3. **Zoom into specified region** – To zoom into a specific region of the timeline, hold down the Control key and drag the mouse to highlight a specific region. When you release the mouse button, the timeline is zoomed into the highlighted region. While you are dragging, hints are displayed in the grid at the top of the timeline, showing the start and end timestamps for the selected region, as well as the duration of the selected region.

**Navigation**

When the timeline is zoomed in, you can navigate to different parts of the timeline. You can use either the horizontal scrollbar (located along the bottom of the timeline), or you can click and drag the mouse to pan the timeline within the viewable area. You also can use the left or right arrow keys on the keyboard to pan the timeline within the viewable area.

**Expanding and Collapsing the timeline tree**

When the timeline is first displayed, its tree is fully expanded. You can collapse parts of the tree in order to limit the amount of data shown. Use the tree view controls within the timeline to collapse or expand parts of the timeline tree. When a branch of the tree is collapsed, timeline items from the collapsed sub-branches are displayed in the parent branch.

**Viewing timeline item details**

There are several ways to view more information about items
shown in the timeline view.

1. **Tooltip hints** – Hover the mouse over a block shown in the timeline, and a tooltip hint appears. It gives additional details about that block.

2. **Navigating to the API trace** – Click an API block in a "Host Thread" row, and that block is selected in the API Trace. There, additional details for that particular API call are shown. Click an item in the "Data Transfer" or "Kernel Execution" row, and the enqueue API that enqueued the data transfer or kernel execution is selected in the API Trace.

**API Trace**

The API trace is a list of all the OpenCL™ or HSA API calls made by the application. Each host thread that makes at least one API call is listed in a separate tab. Each tab contains a list of all the API calls made by that particular thread. For each call, the list displays:

- the index of the call (representing execution order),
- the name of the API function,
- a semi-colon delimited list of parameters passed to the function, and
- the value returned by the function.

When displaying parameters, the Profiler tries to dereference pointers and decode enumeration values; this is in order to give as much information as possible about the data being passed in, or returned from, the function. Double-clicking an item in the API Trace list displays and zooms into that API call in the Host Thread row in the Application Timeline.

For OpenCL™ Enqueue API calls that result in either a kernel execution or a data transfer operation, there is a clickable entry in the "Device Block" column. Clicking this entry zooms into the corresponding timeline block under the OpenCL™ tree
in the timeline.
For OpenCL™ Enqueue API calls that result in a kernel execution on a GPU, there is a clickable value in the "Kernel Occupancy" column. Clicking this entry opens the **GPU Profiler Kernel Occupancy Viewer**, which provides more information about the **kernel occupancy**.

If the option to Enable navigation to source code is checked on the **Application Timeline Trace page**, you can right-click any item in the API trace and choose **Go to source code** from the context menu. This uses the symbol information generated during the trace to navigate to the source code location of the API call. Note that this feature only works if the profiled application was built with debugging information.

The API Trace lets you analyze and debug the input parameters and output results for each API call. For example, you can easily check that all the API calls are returning CL_SUCCESS (OpenCL™) or HSA_STATUS_SUCCESS (HSA), or that all the OpenCL™ buffers are created with the correct flags. You also can identify redundant API calls using the API Trace.

**Colors**

Colors are used in both the application timeline and the API Trace to help distinguish data transfer and kernel dispatch. Green is used on kernel dispatch items for both the OpenCL™ Enqueue calls from the host and the kernels themselves on the device. Shades of blue are used to color OpenCL™ data transfer items, with slight color variations for read, write, and copy calls.

**Note: Special case when the OpenCL™ trace may be incomplete**

When the setting to write trace data in intervals during program execution is enabled on the **Application Timeline Trace page** (which is the default and only supported mode on
Linux), the application trace might not include the full trace of all APIs called by the application. This is because any APIs called after the final interval in the application's lifetime might be omitted. To limit the number of APIs omitted in this scenario, the Profiler also writes all queued-up trace data when the `clReleaseContext` API is called. However, if an application does not call `clReleaseContext` to clean up any OpenCL™ contexts it has created, or if it calls any OpenCL™ APIs after the final `clReleaseContext` call, then the trace might not contain all APIs called.
The following panel shows the GPU performance counters for an OpenCL™ profile session. To get the .csv file of the result, right-click the session in the CodeXL Explorer, and select "Open Containing Folder" from the menu. See Description of Output Files for a detailed description of the format of this file.

At the top of the panel, there is a ""Show Zero Columns"" checkbox. When checked, the session table shows all columns. When unchecked, any column that has a zero or empty value for every row is hidden.

The first several columns in the session are always displayed, even if no performance counters are selected for the profile. A description of these columns for OpenCL™ applications is given in the following table.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>The kernel name (appended with __k[KernelID][DeviceName][DeviceID] to differentiate unique execution order)</td>
</tr>
<tr>
<td>ExecutionOrder</td>
<td>The order of execution for the kernel dispatch operations in the program.</td>
</tr>
<tr>
<td>ThreadID</td>
<td>The thread ID of the host thread that made the OpenCL™ API call that initiated the kernel dispatch.</td>
</tr>
<tr>
<td>CallIndex</td>
<td>The call index of the OpenCL™ API call that initiated the kernel dispatch.</td>
</tr>
<tr>
<td>GlobalWorkSize</td>
<td>The global work-item size of the kernel.</td>
</tr>
<tr>
<td>WorkGroupSize</td>
<td>The work-group size of the kernel.</td>
</tr>
<tr>
<td>Time</td>
<td>The time spent (in milliseconds) executing the kernel.</td>
</tr>
<tr>
<td>LocalMemSize</td>
<td>The amount of local memory (LDS for GPU) used by the kernel.</td>
</tr>
<tr>
<td>VGPRs</td>
<td>The number of general-purpose vector registers used by the kernel.</td>
</tr>
<tr>
<td>SGPRs</td>
<td>The number of general-purpose scalar registers used by the kernel.</td>
</tr>
<tr>
<td>ScratchRegs</td>
<td>The number of scratch registers used by the kernel.</td>
</tr>
<tr>
<td>KernelOccupancy</td>
<td>The kernel occupancy (valid only for GPU devices) as a percentage of the theoretical maximum.</td>
</tr>
</tbody>
</table>

The following table gives a description of these columns for an HSA application.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>The kernel name (appended with the Device Name).</td>
</tr>
<tr>
<td>ExecutionOrder</td>
<td>The order of execution for the kernel dispatch operations in the program.</td>
</tr>
<tr>
<td>ThreadID</td>
<td>The thread ID of the host thread that made the HSA API call that initiated the kernel dispatch.</td>
</tr>
<tr>
<td>GlobalWorkSize</td>
<td>The global work-item size of the kernel.</td>
</tr>
<tr>
<td>WorkGroupSize</td>
<td>The work-group size of the kernel.</td>
</tr>
<tr>
<td>LocalMemSize</td>
<td>The amount of local memory (LDS for GPU) used by the kernel.</td>
</tr>
<tr>
<td>VGPRs</td>
<td>The number of general-purpose vector registers used by the kernel.</td>
</tr>
<tr>
<td>SGPRs</td>
<td>The number of general-purpose scalar registers used by the kernel.</td>
</tr>
<tr>
<td>KernelOccupancy</td>
<td>The kernel occupancy. This is an estimate of the theoretical maximum number of wavefronts.</td>
</tr>
</tbody>
</table>
The following table gives a description of these columns for a DirectCompute application.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>The kernel name (appended with a pointer value that is unique for each kernel instance) or the data transfer operation name.</td>
</tr>
<tr>
<td>ExecutionOrder</td>
<td>The order of execution for the kernel and data transfer operation program.</td>
</tr>
<tr>
<td>ThreadGroup</td>
<td>The thread group size of the kernel.</td>
</tr>
<tr>
<td>WorkGroupSize</td>
<td>The work-group size of the kernel.</td>
</tr>
<tr>
<td>Time</td>
<td>For a kernel dispatch operation: time spent (in milliseconds) executing the kernel; does not include the kernel setup time. For a data transfer operation, time spent (in milliseconds) transferring data.</td>
</tr>
</tbody>
</table>

The **GPU Profile: Performance Counters page** of the **GPU Profiling Project Settings** contains the description of the performance counters. This description is also shown if you hover the mouse cursor over the counter name in the Session panel.

To show the source, IL, or ISA code of an OpenCL™ kernel, or the DXASM code of a DirectCompute kernel, click on the kernel name in the first column to open the **GPU Profiler Code Viewer**.

For OpenCL™ applications, if a kernel is run on a CPU device, only the global work size, work group size, local memory, and the execution time for the kernel is available.

Using the performance counters lets you:

- Find the number of resources (general-purpose registers, local memory size, and flow control stack size) allocated for the kernel. These resources affect the possible number of in-flight wavefronts in the GPU. A higher number can hide data latency better.
- Determine the number of ALU, global, and local memory instructions executed by the GPU.
- Determine the number of bytes fetched from, and written to, the global memory.
- Determine the use of the SIMD engines and memory units in the system.
- View the efficiency of the shader compiler in packing ALU instructions into the VLIW instructions used by AMD GPUs.
- View any local memory (local data share - LDS) bank conflicts.
- View **Kernel occupancy percentage**, which estimates the number of in-flight wavefronts on a compute unit as a percentage of the theoretical maximum number of wavefronts that the compute unit can support.

To view more information about the **kernel occupancy** figure for an OpenCL™ kernel, click on the percentage value in the **Kernel Occupancy** column to open the **GPU Profiler Kernel Occupancy Viewer**.

**Note: Special case when other workloads are using the GPU while profiling**

When collecting performance counters, it is strongly recommended that no other workloads (i.e. graphics workloads) are running on the GPU. Performance counters on AMD Radeon™ GPUs are global in nature, meaning that graphics workloads running on the GPU concurrently with a compute workload that is being profiled can affect the counter values reported. It is recommended that all other applications are closed before profiling. Note: The Windows user interface itself uses the GPU for rendering and it may not be possible to disable this. Because of this, there may be some rare occurrences where the counters for a particular kernel dispatch may be incorrect.

**Note: Special case when an OpenCL™ kernel uses printf**

When profiling an OpenCL™ kernel that contains one or more **printf** calls, the Performance Counter results will show values as if the kernel was dispatched with a single wavefront (regardless of how many actual wavefronts are launched). This is due to the way **printf** is implemented in the OpenCL
runtime. When a kernel contains `printf`, internally, the runtime dispatches each wavefront separately. It is recommended that you remove all `printf` statements from a kernel before you attempt to profile it.
The GPU Profiler Summary Pages show the statistics for your OpenCL™ or HSA application. They can provide you with a general idea of the location of the application's bottlenecks. They also provide information such as the number of buffers and number of images created on each context (for OpenCL™), the most expensive kernel call, etc. One Summary Page, the "Warning(s)/Error(s)" page, shows the result of a rule-based analysis of the API trace and timeline data. You can sort each column in a summary page by clicking the table header. You also can rearrange the columns by dragging them to a new location. By default, the Summary Pages are generated when performing an Application Trace profile from CodeXL. You can control whether the Summary Pages are generated by changing the settings on the Application Timeline Trace page. To generate summary pages from the command line, see Using the Command Line Interface. You can find summary pages under the same directory as the .atp file. You can view each summary page in your default web browser because all summary pages are in html format.
API Summary Page

The API Summary page shows statistics for all OpenCL™ or HSA API calls made by the application. This page can help to identify any API hotspots.

<table>
<thead>
<tr>
<th>API Name</th>
<th># of Calls</th>
<th>Total Time(ms)</th>
<th>Ave Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clSetKernelArg</td>
<td>60884</td>
<td>106.42239</td>
<td>0.00175</td>
</tr>
<tr>
<td>clGetKernelInfo</td>
<td>4332</td>
<td>8.29252</td>
<td>0.00191</td>
</tr>
<tr>
<td>clEnqueueNDRangeKernel</td>
<td>4332</td>
<td>178.09443</td>
<td>0.04111</td>
</tr>
<tr>
<td>clReleaseMemObject</td>
<td>4265</td>
<td>537.25810</td>
<td>0.12597</td>
</tr>
<tr>
<td>clCreateBuffer</td>
<td>4265</td>
<td>13.41322</td>
<td>0.00314</td>
</tr>
<tr>
<td>clEnqueueReadBuffer</td>
<td>938</td>
<td>2403.70320</td>
<td>2.56258</td>
</tr>
<tr>
<td>clEnqueueWriteBuffer</td>
<td>231</td>
<td>318.01591</td>
<td>1.37669</td>
</tr>
<tr>
<td>clEnqueueWriteBufferRect</td>
<td>63</td>
<td>1286.34994</td>
<td>20.41825</td>
</tr>
<tr>
<td>clEnqueueReadBufferRect</td>
<td>63</td>
<td>4288.33441</td>
<td>68.06880</td>
</tr>
<tr>
<td>clReleaseKernel</td>
<td>13</td>
<td>0.03764</td>
<td>0.00290</td>
</tr>
</tbody>
</table>
### OpenCL™ Context Summary Page

The Context summary page shows the statistics for all the OpenCL™ kernel dispatch and data transfer operations for each context. It also shows the number of buffers and images created for each context.

<table>
<thead>
<tr>
<th>Context ID</th>
<th># of Buffers</th>
<th># of Images</th>
<th># of Kernel Dispatch - CPU_Device</th>
<th>Total Kernel Time(ms) - CPU_Device</th>
<th># of Kernel Dispatch - Juniper</th>
<th>Total Kernel Time(ms) - Juniper</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>69.07170</td>
<td>1</td>
<td>4.10271</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>35.53000</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>3.41856</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>35.73143</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>1.38896</td>
</tr>
<tr>
<td>Total</td>
<td>10</td>
<td>0</td>
<td>3</td>
<td>140.33313</td>
<td>3</td>
<td>8.91023</td>
</tr>
</tbody>
</table>
Kernel Summary Page

The Kernel summary page shows statistics for all the kernels that are dispatched by the application.

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Device Name</th>
<th># of Calls</th>
<th>Total Time (ms)</th>
<th>Avg Time (ms)</th>
<th>Max Time (ms)</th>
<th>Min Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiDeviceKernel</td>
<td>CPU_Device</td>
<td>3</td>
<td>140.33313</td>
<td>46.77771</td>
<td>69.07170</td>
<td>35.53000</td>
</tr>
<tr>
<td>multiDeviceKernel</td>
<td>Juniper</td>
<td>3</td>
<td>8.91023</td>
<td>2.97008</td>
<td>4.10271</td>
<td>1.38896</td>
</tr>
</tbody>
</table>

OpenCL™ Top 10 Data Transfer Summary Page

The Top 10 Data transfer summary page shows a sorted list of the ten most time-consuming OpenCL™ data transfers operations. Clicking on a hyperlink takes you to the corresponding item in the Timeline view. Since data transfer operations can have a great impact on application performance, ensuring that kernel execution operations and data transfer operations overlap can lead to better overall performance.

<table>
<thead>
<tr>
<th>Command Type</th>
<th>Context ID</th>
<th>Command Queue ID</th>
<th>Duration (ms)</th>
<th>Transfer Size</th>
<th>Transfer Rate (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_BUFFER</td>
<td>4</td>
<td>5</td>
<td>1.45957</td>
<td>256.00 KB</td>
<td>171.284</td>
</tr>
<tr>
<td>WRITE_BUFFER</td>
<td>0</td>
<td>1</td>
<td>0.94618</td>
<td>256.00 KB</td>
<td>264.220</td>
</tr>
<tr>
<td>WRITE_BUFFER</td>
<td>2</td>
<td>3</td>
<td>0.68449</td>
<td>256.00 KB</td>
<td>365.237</td>
</tr>
<tr>
<td>WRITE_BUFFER</td>
<td>1</td>
<td>2</td>
<td>0.21951</td>
<td>256.00 KB</td>
<td>1138.895</td>
</tr>
<tr>
<td>WRITE_BUFFER</td>
<td>3</td>
<td>4</td>
<td>0.17307</td>
<td>256.00 KB</td>
<td>1444.527</td>
</tr>
<tr>
<td>WRITE_BUFFER</td>
<td>0</td>
<td>0</td>
<td>0.13885</td>
<td>256.00 KB</td>
<td>1800.569</td>
</tr>
</tbody>
</table>

Top 10 Kernel Summary Page

The Top 10 kernel summary page shows a sorted list of the 10
most time-consuming kernel execution operations. Clicking on a hyperlink takes you to the corresponding item in Timeline view.

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Context ID</th>
<th>Command Queue ID</th>
<th>Device Name</th>
<th>Duration(ms)</th>
<th>Global Work Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiDeviceKernel</td>
<td>0</td>
<td>0</td>
<td>CPU_Device</td>
<td>69.07170</td>
<td>{65536}</td>
</tr>
<tr>
<td>multiDeviceKernel</td>
<td>3</td>
<td>4</td>
<td>CPU_Device</td>
<td>35.73143</td>
<td>{65536}</td>
</tr>
<tr>
<td>multiDeviceKernel</td>
<td>1</td>
<td>2</td>
<td>CPU_Device</td>
<td>35.53000</td>
<td>{65536}</td>
</tr>
<tr>
<td>multiDeviceKernel</td>
<td>0</td>
<td>1</td>
<td>Juniper</td>
<td>4.10271</td>
<td>{65536}</td>
</tr>
<tr>
<td>multiDeviceKernel</td>
<td>2</td>
<td>3</td>
<td>Juniper</td>
<td>3.41856</td>
<td>{65536}</td>
</tr>
<tr>
<td>multiDeviceKernel</td>
<td>4</td>
<td>5</td>
<td>Juniper</td>
<td>1.38896</td>
<td>{65536}</td>
</tr>
</tbody>
</table>

**Warning(s)/Error(s) Page**

The Warning(s)/Error(s) Page shows potential problems in your OpenCL™ or HSA application. It can detect unreleased resources, API failures, and it can provide suggestions for better performance. Clicking on a hyperlink takes you to the corresponding API.

<table>
<thead>
<tr>
<th>Index</th>
<th>Call Index</th>
<th>Thread ID</th>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>542</td>
<td>2268</td>
<td>Warning</td>
<td><strong>Memory leak detected [Ref = 1, Handle = 0x0B1730B0]: Object created by</strong> clEnqueueNDRangeKernel</td>
</tr>
<tr>
<td>216</td>
<td>208</td>
<td>2268</td>
<td>Best Practices</td>
<td>clEnqueueNDRangeKernel: Work-group size is too small - [1,1,1]. Recommended work-group size is a multiple of 64.</td>
</tr>
<tr>
<td>270</td>
<td>319</td>
<td>2268</td>
<td>Best Practices</td>
<td>clEnqueueNDRangeKernel: Global work size is too small - [111], resulting in low GPU utilization.</td>
</tr>
<tr>
<td>144</td>
<td>482</td>
<td>1932</td>
<td>Error</td>
<td>clEnqueueNDRangeKernel returns CL_INVALID_KERNEL_ARGS.</td>
</tr>
</tbody>
</table>
From these summary pages, it is possible to determine whether an OpenCL™ application is bound by kernel execution or data transfer (Context Summary page). If the application is bound by kernel execution, you can determine which device is the bottleneck. From the Kernel Summary page, you can find the name of the kernel with the highest total execution time. From the Top 10 Kernel Summary page, you can find the individual kernel instance with the highest execution time. If the kernel execution on a GPU device is the bottleneck, the GPU performance counters can then be used to investigate the bottleneck inside the kernel.

If the application is bound by the data transfers, it is possible to determine the most expensive data transfer type (read, write, copy or map) in the application from the Context Summary page. You can then investigate whether you can minimize this type of data transfer by modifying the algorithm if necessary. With help from the Timeline View, you can investigate whether data transfers have been executed in the most efficient way (concurrently with a kernel execution).
The Code Viewer appears when you click on the name of a kernel in the first column of the **GPU Profiler Performance Counters Session** panel.

For OpenCL™ kernels, this panel shows the generated ISA or IL/HSAIL code of the kernel. It also shows the CL source code of the kernel, if the kernel source is available from the OpenCL™ runtime.

For DirectCompute kernels, this panel shows the DXASM code of the kernel.

For OpenCL™ kernels, you can select different modes in the combo box at the top of the panel to switch between displaying the IL code, ISA code, and CL source code.
GPU Profiler Kernel Occupancy Viewer
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Device Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Info</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device name</td>
<td>Beaver Creek</td>
<td></td>
</tr>
<tr>
<td>Number of compute units</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Max number of waves per compute unit</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Max number of work-groups per compute unit</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Wavefront size</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Kernel Info</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel name</td>
<td>computeIntersection</td>
<td></td>
</tr>
<tr>
<td>Vector GPR usage per work-item</td>
<td>16</td>
<td>240</td>
</tr>
<tr>
<td>LDS usage per work-group</td>
<td>0</td>
<td>128/64</td>
</tr>
<tr>
<td>Flattened work-group size</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Flattened global work size</td>
<td>111</td>
<td>16777216</td>
</tr>
<tr>
<td>Number of waves per work-group</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Kernel Occupancy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of waves limited by Vector GPR and Work-group size</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Number of waves limited by LDS and Work-group size</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Number of waves limited by Work-group size</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Limiting factor(s)</td>
<td>VPR</td>
<td></td>
</tr>
<tr>
<td>Estimated occupancy</td>
<td>37.5%</td>
<td></td>
</tr>
</tbody>
</table>
There are two ways to open the Kernel Occupancy panel:

- Click on the **kernel occupancy percentage** in the Kernel Occupancy column of the **GPU Profiler Performance Counters Session** panel (OpenCL™ and HSA).
- or

- Click on the **kernel occupancy percentage** in the Kernel Occupancy column of the API Trace in the **GPU Profiler Profiler Application Trace Session** panel (OpenCL™ only).

For kernels, this panel displays an HTML webpage which provides information about the occupancy of a particular kernel dispatch.

The top part of the page shows four graphs that provide a visual indication of how kernel resources affect the theoretical number of in-flight wavefronts on a compute unit. The graph representing the limiting resource has its title displayed in red text. More than one graph can have a red title if there is more than one limiting resource. In each graph, the actual usage of the particular resource being graphed is highlighted with an orange square. If you hover the mouse over a point in the graph, a popup hint is displayed showing you the current X and Y values at that location.

The first graph, titled **Number of waves limited by Workgroup size**, shows how the number of active wavefronts is affected by the size of the work-group for the dispatched kernel. In the screenshot above, you can see that the highest number of wavefronts is achieved when the work-group size is in the between 64 and 256.

The second graph, titled **Number of waves limited by VGPRs**, shows how the number of active wavefronts is affected by the number of vector GPRs used by the dispatched kernel. In the screenshot above, you can see that as the number of VGPRs used increases, and the number active wavefronts decreases, in steps. Note this graph shows that more than 62 VGPRs can be allocated, even though 62 is the maximum number of VGPRs that can be allocated, since the shader
compiler assumes the work-group size is 256 items by default (the largest possible work-group size). For the shader compiler to allocate more than 62 VGPRs, the kernel source code must be marked with the required_work_group_size kernel attribute. This attribute specifies to the shader compiler that the kernel is launched with a work-group size smaller than the maximum, allowing it to allocate more VGPRs. Thus, for X-axis values greater than 62, the VGPR graph shows the theoretical number of wavefronts that can be launched if the kernel specified a smaller work-group size using the attribute.

The third graph, titled **Number of waves limited by SGPRs**, shows how the number of active wavefronts is affected by the number of scalar GPRs used by the dispatched kernel. In the above screenshot, you can see that as the number of SGPRs used increases, the number active wavefronts decreases in steps.

The fourth graph, titled **Number of waves limited by LDS**, shows how the number of active wavefronts is affected by the amount of LDS used by the dispatched kernel. In the above screenshot, you can see that as the amount of LDS used increases, the number active wavefronts decreases in steps.

A table, below the four graphs, provides information about the device, the kernel, and the kernel occupancy. In the **Kernel Occupancy** section, you can see the limits imposed by each kernel resource, as well as which resource is currently limiting the number of waves for the kernel dispatch. This section also displays the **kernel occupancy** percentage.
This page provides an overview of the kernel occupancy calculation, providing the definition of the parameter and discussing the factors influencing the value and its interpretation.

Kernel occupancy is a measure of the use of the resources of a compute unit on a GPU, the use being measured by the number of in-flight wavefronts, for a given kernel, relative to the number of wavefronts that can be launched given the ideal kernel dispatch configuration (dependent on the work-group size and resource use in the kernel).

The number of wavefronts that are scheduled when a kernel is dispatched is constrained by three significant factors:

- the number of general purpose registers (GPR) required by each work-item,
- the amount of shared memory (LDS for local data store) used by each work-group, and
- the configuration of the work-group (the work-group size).

The basic definition of the occupancy (O) is given by:

\[ O = \frac{N_{WF}^A}{N_{WF}^T} \]

where \( N_{WF}^A \) is the number of in-flight wavefronts on the compute unit, and \( N_{WF}^T \) is the theoretical number of wavefronts that the compute unit can execute concurrently.

The first constraint is that work that is assigned to a compute unit is scheduled as groups of individual work-items, called
wavefronts, which have a fixed size defined by the hardware. The characteristic of a wavefront is that each work-item executes in step with the other work-items in the wavefront. The number of work-items that can be executed on a compute unit must be a multiple of a wavefront. In an ideal situation, the number of wavefronts that can be scheduled corresponds to the maximum number of wavefronts supported by the compute unit.

However, because there are resources that are shared among work-groups, which is the basic unit of processing on the compute unit, wavefronts are scheduled as part of a work-group. A work-group consists of a collection of work-items that make use of a common block of local data storage (LDS) that is shared among the members of the work-group, as well as registers. Each work-group consists of one or more wavefronts. Thus, the total number of wavefronts that can be launched on a compute unit is also constrained by the number of work-groups as this must correspond to an integral number of workgroups, even if the compute unit has capacity for additional wavefronts. In the ideal situation, the number of wavefronts that can be launched is an integral multiple of the number of wavefronts per work-group, which means that the maximum number of wavefronts the GPU is capable of allocating, can be achieved. When this is not the case, changing the size of the work-items in the work-group can change the number of wavefronts in the work-group.

**Kernel Occupancy for AMD Radeon™ HD 5000/6000 Series Based on VLIW5/VLIW4 Architecture**

1. **LDS limits on the number of in-flight**

In the case that the LDS is the only constraint on the number of in-flight wavefronts, the compute unit can support the launch of a number of in-flight work-groups given by:
where \( W_{G_{\text{max}}} \) is the maximum number of work-groups on a compute unit, \( L_{D_{S_{\text{CU}}}} \) is the shared memory available on the compute unit, and \( L_{D_{S_{\text{wg}}}} \) is the shared memory required by the work-group (based on the resources required by the kernel). The corresponding number of wavefronts is given as:

\[
WF_{\text{max}} = W_{G_{\text{max}}} \times WF_{\pi G}
\]

where \( WF_{\text{max}} \) is the maximum number of wavefronts, \( W_{G_{\text{max}}} \) is the maximum number of work-groups, and \( WF_{\pi G} \) is the number of wavefronts in a work-group.

There is also another constraint whereby a compute unit can only support a fixed number of work-groups, a hard limit of \( W_{G_{\text{max}}} = 8 \) (denoted by \( W_{G_{\text{max}}^{\text{CU}}} \)). This also limits the effectiveness of reducing the work-group size excessively, as the number of wavefronts is also limited by the maximum workgroup size. Currently, the maximum work-group size is 256 work-items, which means that the maximum number of wavefronts is 4 when the wavefront size is 64 (and 8 when the wavefront size is 32).

Thus, when the only limit to the number of wavefronts on a compute unit is set by the LDS usage (for a given kernel), then the maximum number of wavefronts, (LDS-limited) is given by:

\[
WF_{\text{LDS}} = \min(W_{G_{\text{max}}^{\text{CU}}} \times WF_{\pi G}, W_{G_{\text{max}}} \times WF_{\pi G})
\]

2. **GPR limits on the number of in-flight wavefronts**

Another limit on the number of in-flight wavefronts is the number of general-purpose registers (GPRs). Each compute unit has 16384 registers. These are divided among the work-items in a wavefront. Thus, the number of registers per work-item limits the number of wavefronts that can be launched. This can be expressed as:
\[ W_{\text{GPR}} = \frac{N_{\text{max}}^{\text{reg}}}{N_{\text{used}}^{\text{reg}}} \]

where \( N_{\text{reg}} \) is the number of registers per work-item; the superscripts \( \text{max} \) and \( \text{used} \) refer to the maximum number of registers per thread and the actual number of registers used.

The number of in-flight wavefronts being constrained by the work-group granularity, the number of GPR-limited wavefronts is given by:

\[ W_{\text{GPR}} = \text{floor} \left( \frac{W_{\text{GPR}}^{\text{cu}}}{W_{\text{WG}}^{\text{cu}}} \right) \times W_{\text{WG}} \]

### 3. Other constraints

Another limit on the number of in-flight wavefronts is the FCStack; however, this is really an insignificant constraint, so this is not considered here.

The final factor in the occupancy is the work-group size, as briefly discussed above. If there are no other constraints on the number of wavefronts on the compute unit, the maximum number of wavefronts is given by:

\[ W_{\text{WG}} = \min \left( \text{floor} \left( \frac{W_{\text{cu}}}{W_{\text{WG}}^{\text{cu}}} \right) \times W_{\text{WG}} \right) \]

where \( W_{\text{max}}^{\text{cu}} \) is the maximum number of wavefronts on the compute unit and \( W_{\text{WG}}^{\text{max}} \) is the maximum number of wavefronts on a compute unit when there are no other constraints than the work-group size.

This equation shows that having a workgroup size where the number of wavefronts divides the maximum number of wavefronts on the compute unit evenly generally yields the greatest number of in-flight wavefronts, while at the same time indicating that making the work-group size too small yields a reduced number of wavefronts. For example, setting a workgroup consisting of only 1 wavefront yields only 8 in-flight wavefronts, whereas (for example, given a maximum number of
wavefronts on the compute unit of 32), a work-group of 2 wavefronts will yield 16 wavefronts. Furthermore, having a single wavefront per work-group doubles the LDS usage relative to having 2 wavefronts per work-group as the LDS is only shared among the wavefronts in a same work-group (but not between work-groups).

Given these constraints, the maximum number of in-flight wavefronts is given by:

$$N_{\text{if}}^d = \min \left\{ \frac{W_F}{W_G}, \frac{W_F}{W_G} \right\}$$

Thus, the occupancy, O, is given by:

$$O = \frac{\min \left\{ \frac{W_F}{W_G}, \frac{W_F}{W_G} \right\}}{N_{\text{if}}^d}$$

The occupancy shown here is the estimated occupancy on a single compute unit. It is independent of the work-loads on the other compute units on the GPU because the occupancy is only really meaningful if there are sufficient work-items to require all the resources of at least one compute unit (and even then, ideally, there should be a sufficient work-load to ensure that more than one compute unit is needed to execute the work in order to gain the benefits of parallel operations). Higher occupancy allows for increased global memory latency hiding as it allows wavefronts to be swapped when there are global memory accesses. However, once there is a sufficient number of wavefronts on the compute unit to hide any global memory accesses, increasing occupancy may not increase performance.

**Kernel Occupancy for AMD Radeon™ HD 7000 Series or Newer, Based on Graphics Core Next Architecture**

There are a number of significant differences from the previous occupancy calculation due to the different architecture. In the Graphics Core Next architecture, each compute unit is actually made up of four SIMDs. While some features, such as the GPR,
are still computed on the basis of individual SIMDs, these must be scaled to the whole compute unit. On the other hand, workgroup limits must be computed over the whole compute unit. These are detailed below.

The first limit to the number of active wavefronts on the compute unit is the work-group size. Each Compute unit (CU), has up to 40 slots for wavefronts. If each work-group is exactly one wavefront, then the maximum number of wavefronts is:

\[ WF_{\text{max}} = 40 \]

Otherwise, if there is more than one wavefront (WF) per work-group (WG), there is an upper limit of 16 work-groups (WG) per compute unit (CU). Then, the maximum number of wavefronts on the compute unit is given by:

\[ WF_{WG}^{\text{max}} = \min(16 \times WF_{WG}, WF_{\text{max}}) \]

where \( WF_{WG} \) is the number of wavefronts per work group.

The second limit on the number of active wavefronts is the number of VGPR per SIMD.

\[ WF_{\text{VGPR}}^{\text{max}} = \text{floor}\left( \frac{VGPR_{\text{max}}}{VGPR_{\text{used}}} \right) \]

Where \( VGPR_{\text{max}} \) is maximum number of registers per work-item and \( VGPR_{\text{used}} \) is the actual number of registers used per work-item. However, we are interested in the total number of wavefronts per CU, so we have to scale this value by the number of CU.

\[ WF_{\text{VGPR}}^{\text{max}} = WF_{\text{VGPR}}^{\text{max}} \times \text{SIMD}_{\text{PER CU}} \]

At the same time, the number of wavefronts cannot exceed \( WF_{\text{max}} \), so

\[ WF_{\text{VGPR}}^{\text{max}} = \min(WF_{\text{VGPR}}^{\text{max}}, WF_{\text{max}}) \]

However, the wavefronts are constrained by work-group granularity, so the maximum number of wavefronts limited by the VGPR is given by

\[ WF_{\text{VGPR}}^{\text{max}} = \text{floor}\left( \frac{WF_{\text{VGPR}}^{\text{max}}}{WF_{WG}} \right) \times WF_{WG} \]
The third limit on the number of active wavefronts is the number of SGPR. Similar to VGPR, SGPR is calculated by

$$WF_{SGPR}^{max} = \text{floor} \left( \frac{\min(floor(SGPR_{max}) \cdot SIMD_{PERCU}, WF_{max})}{WF_{WG}} \right) \cdot WF_{WG}$$

The final limit on the number of active wavefronts is the LDS. The LDS limited number of wavefronts is given by:

$$WG_{max} = \text{floor} \left( \frac{LDS_{max}}{LDS_{used}} \right)$$

where $WG_{max}$ is the maximum number of work-groups determined by the LDS. Then, the maximum number of wavefronts is given by:

$$WF_{LDS}^{max} = WG_{max} \cdot WF_{WG}$$

Thus, the occupancy, $O$, is given by:

$$O = \frac{\min(WF_{LDS}^{max}, WF_{SGPR}^{max}, WF_{VGPR}^{max}, WF_{WG}^{max})}{WF_{max}}$$

© 2015 Advanced Micro Devices. All rights reserved.
GPU Profiling Project Settings

These Project Settings pages let you configure various aspects of the GPU Profiler for the active project.

The following pages contain the settings that can be configured:

Application Timeline Trace page
GPU Profile: Performance Counters page

© 2015 Advanced Micro Devices. All rights reserved.
Application Timeline Trace page

This page lets you configure the behavior of the Profiler when it performs an application timeline trace.

Application Timeline Trace settings page
Profile applications

Select the API to trace. When one of the OpenCL / HSA radio buttons is clicked, the project setting pages will...
<p>| that use | display the options relevant to the selected API. |
| Enable navigation to source code (high overhead) | When checked, the Profiler generates a symbol information file from an application's debugging information (the .pdb file on Windows), containing one entry for each called OpenCL™ API. This symbol information file lets you navigate from an item in the API Trace in the Application Timeline Trace Session panel to the source location of the API call. |
| Write trace data in intervals during program execution (ms) | When selected, the Profiler periodically writes all queued trace data to disk during program execution. The interval (in milliseconds) at which to write trace data is specified using the value following the checkbox. When checked, in addition to writing data periodically, the Profiler also writes all queued trace data when the <strong>clReleaseContext</strong> OpenCL™ API is called. However, if an application does not call <strong>clReleaseContext</strong>, or if it calls any OpenCL™ APIs after the final <strong>clReleaseContext</strong> call, then it is possible that not all trace data is written to the disk. When unchecked, all trace data is written to disk when the application terminates. On Linux, this is the default (and only supported) mode for writing trace data. Thus on Linux, the UI lets you |</p>
<table>
<thead>
<tr>
<th><strong>Maximum number of APIs to trace</strong></th>
<th>specify the interval but does not let you enable or disable writing the data in intervals.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>This controls how many APIs are traced over an application's lifetime. The default number of APIs to trace is 1 million. Limiting the number of APIs traced helps to prevent running out of memory while profiling. After the limit is reached, no additional APIs is traced, and the trace results do not include any additional information. Because of this, any information provided in the <a href="#">GPU Profiler Summary Pages</a> might not be correct, as a complete trace is required to provide a fully-accurate application summary.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Always show API error codes</strong></td>
<td>When checked, the Profiler reports the return codes for all OpenCL™ API calls. Some OpenCL™ API functions return an error code through a passed-in parameter. If the host application passes in NULL for that parameter, then the OpenCL™ runtime does not report an error code. The Profiler substitutes a non-null parameter in this case, and the API Trace can show the return code.</td>
</tr>
<tr>
<td></td>
<td>Some OpenCL™ applications wait for certain Enqueue API calls to complete by continuously checking the status of the event returned by the Enqueue API. These applications do this by calling <code>clGetEventInfo</code></td>
</tr>
</tbody>
</table>
Collapse consecutive identical clGetEventInfo calls

within a loop until the event status reaches a certain state (typically CL_COMPLETE). For these applications, the timeline and API trace can contain thousands of clGetEventInfo calls, making it difficult to easily analyze the timeline and trace data. To make analysis easier, the Profiler can collapse consecutive clGetEventInfo calls that have the same parameters and return values into a single entry in the timeline and API trace.

Generate occupancy information for each OpenCL kernel profiled

When checked, the Profiler generates kernel occupancy data for each OpenCL™ kernel dispatched to a GPU device.

Generate summary pages

When checked, the Profiler automatically generates GPU Profiler Summary Pages using the API trace and timeline data. You can further configure the summary pages by selecting rules to be used when generating the Warning(s)/Error(s) Summary page. The following table shows the currently supported rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect resource leaks</td>
<td>Tracks the reference count for all OpenCL™ objects not released</td>
</tr>
<tr>
<td>Detect deprecated API calls</td>
<td>Detects calls to OpenCL™ API functions of deprecated versions of OpenCL™</td>
</tr>
</tbody>
</table>

When checked, the Profiler automatically generates GPU Profiler Summary Pages using the API trace and timeline data. You can further configure the summary pages by selecting rules to be used when generating the Warning(s)/Error(s) Summary page. The following table shows the currently supported rules.
<table>
<thead>
<tr>
<th>Detect unnecessary blocking writes</th>
<th>Detects unnecessary blocking write operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect non-optimized work size</td>
<td>Detects <code>clEnqueueNDRangeKernel</code> calls with a size that is non-optimal for AMD Hardware</td>
</tr>
</tbody>
</table>
| Detect non-optimized data transfer | 1. Detects non-Fusion APU access to Device-Visible Host Memory directly  
2. Detects host-visible Device Memory read back to CPU directly |
| Detect redundant synchronization   | Detects redundant synchronization that results in low host and device use |
| Detect failed API calls           | Detects OpenCL™ API calls that do not return `CL_SUCCESS`.  
Detects HSA API calls that do not return `HSA_STATUS_SUCCESS`.  
Some of the return codes from OpenCL™ might not be detected unless the `Always show API error codes` option is checked |

- **APIs to trace** When checked, you can tell the Profiler which APIs you want traced. By limiting the APIs to trace, you can focus attention on particular APIs when analyzing trace data while also reducing the overhead of performing a trace. Because a full trace is required in order to generate the Summary pages, this option is mutually exclusive with the Generate summary pages option. Use the treeview below the option to select the APIs for the Profiler to trace.

© 2015 Advanced Micro Devices. All rights reserved.
This page lets you configure the behavior of the Profiler when it collects performance counters.
Settings
- **Measure kernel execution time** when checked, requires an additional pass during collection. (only applicable for OpenCL)
- **Generate occupancy information for each OpenCL™ or HSA kernel profiled** When checked, the Profiler generates kernel occupancy data for each OpenCL™ kernel dispatched to a GPU device.
- **Profile specific kernels** Profile only kernels that their names are specified.
- **Counter selection TreeView** This treeview displays the available GPU performance counters that can be enabled for a profile session. The performance counters are grouped by counter type. The counters shown depend on the type of GPU installed on the system. If the system has multiple GPU devices from multiple hardware families, the tree contains a top-level node for each available hardware family. For instance, if a system has both an AMD Radeon™ HD 7000 series GPU device (one based on Graphics Core Next Architecture) and an AMD Radeon™ HD 5000 series device, then the counter selection treeview includes counters supported by each device (see screenshot below).
- Some counter selection combinations require multi-pass collection. When profiling using multiple passes, any OpenCL kernels that use shared virtual memory or pipes as arguments will not be profiled.
- When more than one pass is required, the number of required passes will be displayed next to the device name.
- To load and save the counter selections to a file, click on the **Load Selection** and **Save Selection** buttons.

Below is a list and brief description of available counters. You also can use the cursor to hover over the counter names in the treeview to view the descriptions.
- The full set of counters for AMD Radeon™ HD 7000 series GPU devices or newer (based on Graphics Core
Next Architecture) are described in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wavefronts</td>
<td>Total wavefronts.</td>
</tr>
<tr>
<td>VALUInsts</td>
<td>The average number of vector ALU instructions executed per work-item (affected by flow control).</td>
</tr>
<tr>
<td>SALUInsts</td>
<td>The average number of scalar ALU instructions executed per work-item (affected by flow control).</td>
</tr>
<tr>
<td>VFetchInsts</td>
<td>The average number of vector fetch instructions from the video memory executed per work-item (affected by flow control).</td>
</tr>
<tr>
<td>SFetchInsts</td>
<td>The average number of scalar fetch instructions from the video memory executed per work-item (affected by flow control).</td>
</tr>
<tr>
<td>VWriteInsts</td>
<td>The average number of vector write instructions to the video memory executed per work-item (affected by flow control).</td>
</tr>
<tr>
<td>FlatVMemInsts</td>
<td>The average number of FLAT instructions that read from or write to the video memory executed per work item (affected by flow control).</td>
</tr>
<tr>
<td>GDSInsts</td>
<td>The average number of GDS read or GDS write instructions executed per work-item (affected by flow control).</td>
</tr>
<tr>
<td>VALUUtilization</td>
<td>The percentage of active vector ALU threads. Either more thread divergence in a wavefront or the work-group size is not a multiple of 64. Value range: 0% (bad), 100% (ideal).</td>
</tr>
<tr>
<td>VALUBusy</td>
<td>The percentage of GPUtil time vector ALU instructions executed. Value range: 0% (bad) to 100% (optimal).</td>
</tr>
<tr>
<td>SALUBusy</td>
<td>The percentage of GPUtil time scalar ALU instructions executed. Value range: 0% (bad) to 100% (optimal).</td>
</tr>
<tr>
<td>LDSInsts</td>
<td>The average number of LDS read or LDS write instructions executed per work-item (affected by flow control).</td>
</tr>
<tr>
<td>FlatLDSInsts</td>
<td>The average number of FLAT instructions that read from or write to the video memory executed per work item (affected by flow control).</td>
</tr>
<tr>
<td>LDSBankConflict</td>
<td>The percentage of GPUtil time LDS is stalled by bank conflicts. Value range: 0% (bad) to 100% (optimal).</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FetchSize</td>
<td>The total kilobytes fetched from the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.</td>
</tr>
<tr>
<td>WriteSize</td>
<td>The total kilobytes written to the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.</td>
</tr>
<tr>
<td>CacheHit</td>
<td>The percentage of fetch, write, atomic, and other instructions that hit the data cache. Value range: 0% (no hit) to 100% (optimal).</td>
</tr>
<tr>
<td>MemUnitBusy</td>
<td>The percentage of GPUtil time the memory unit is active. The result includes the stall time (MemUnitStalled). This is measured with all extra fetches and any cache or memory effects taken into account. Value range: 0% to 100% (fetch-bound).</td>
</tr>
<tr>
<td>MemUnitStalled</td>
<td>The percentage of GPUtil time the memory unit is stalled. Try reducing the number or size of fetches and writes if possible. Value range: 0% (optimal) to 100% (bad).</td>
</tr>
<tr>
<td>WriteUnitStalled</td>
<td>The percentage of GPUtil time the Write unit is stalled. Value range: 0% to 100% (bad).</td>
</tr>
</tbody>
</table>

© 2015 Advanced Micro Devices. All rights reserved.
Description of Output Files

SESSION_NAME.csv

This comma-delimited file is generated when a profile collects performance counters.

The file starts with a file header section (in comments) that indicates the Profiler version number and information about the application that was profiled. Following the file header is a line containing the list of the column headers shown in the GPU Profiler Performance Counters Session panel. Most items in this row represent the performance counters that were collected.

Each additional line contains data collected by the Profiler. There will be one line for each kernel dispatched by the profiled application.

SESSION_NAME.atp

This file is generated when performing a profile that collects an application timeline trace. The file starts with a file header section which contains the trace file version number, the Profiler version number, and information about the application that was profiled. Following the file header are several sections: the first section contains the API Trace data for the profile session; the second contains timestamp data for the profile session. For HSA traces that include HSA kernel dispatches, there will be a section containing the kernel dispatch timestamp data. If the option to Enable navigation to source code is checked on the Application Timeline Trace page, there will be a section containing the source code
information for the profile section.
The API Trace section contains one or more **thread blocks**.
An API Trace **thread block** consists of the following.

- A line giving the thread ID.
- A line giving the number of APIs for that thread, followed by a line for each API.

Each API is listed in the format: `ReturnValue = APIName ( ParameterList )`.

The **ParameterList** is a semi-colon delimited list of the parameters passed to the API.

The Timestamp section contains one or more **thread blocks**.
In the Timestamp section, all time counter data represents CPU-based time expressed in nanoseconds. A Timestamp **thread block** consists of the following.

- A line giving the thread ID.
- A line giving the number of APIs for that thread, followed by an **API line** for each API. An **API line** consists of at least 4 pieces of data:
  - An integer representing the API type.
  - A string showing the API name.
  - The time counter value for the start of the API.
  - The time counter value for the end of the API.

Most OpenCL™ Enqueue APIs contain the following additional data, appended to the end of the **API line**.

- An integer representing the enqueue command type.
- A string showing the enqueue command name.
- The time counter value for the time the command was queued by the host – this corresponds to `CL_PROFILING_COMMAND_QUEUED`.
- The time counter value for the time the command was submitted by the host to the target device – this corresponds to `CL_PROFILING_COMMAND_SUBMIT`.
- The time counter value for the time the command started executing on the target device – this corresponds to
CL_PROFILING_COMMAND_START.

- The time counter value for the time the command finished executing on the target device – this corresponds to CL_PROFILING_COMMAND_END.
- The unique numerical ID of the queue.
- The handle of the queue.
- The unique numerical ID of the context.
- The handle of the context.
- The device name.

OpenCL™ Kernel dispatch Enqueue commands contain the following additional data appended to the end of the API line:

- The handle of the kernel.
- The name of the kernel.
- The global work size for the kernel – one value is given for each work dimension.
- The work-group size for the kernel – one value is given for each work dimension.

OpenCL™ Data transfer Enqueue commands contain the data transfer size appended to the end of the API line.

The HSA Kernel Timestamp section contains the following information:

- A line giving the number of HSA kernel dispatches, followed by a Kernel Timestamp line for each kernel dispatched by the application. A Kernel Timestamp line consists of the following pieces of data:
  - A string showing the kernel symbol name.
  - The handle of the kernel.
  - The time counter value for the time the kernel started executing on the device.
  - The time counter value for the time the kernel finished executing on the device.
  - The name of the agent the where the kernel was dispatched.
  - The handle of the agent where the kernel was dispatched.
  - The zero-based index of the queue that was used to
dispatch the kernel.
- The handle of the queue that was used to dispatch the kernel.

The Source Code section contains one or more thread blocks. A Source Code thread block consists of the following.

- A line giving the thread ID.
- A line giving the number of APIs for that thread, followed by a Source Code line for each API. A Source Code line consists of the following 4 pieces of data:
  - A string showing the API name.
  - A string showing the name of the function that called the API (or an address if no debug information was found).
  - An integer representing the line number for the location of the API call.
  - A string showing the name of the file for the location of the API call (this is not shown if no debug information was found).

**SESSION_NAME.occupancy**

This comma-delimited file is generated when a profile collects kernel occupancy information.

The file starts with a file header section (in comments) that indicates the Profiler version number and information about the application that was profiled. Following the file header is a line containing the list of names of the data used in order to compute kernel occupancy.

Each additional line contains data collected by the Profiler. There will be one line for each kernel dispatched by the profiled application to a GPU device.
Description of Configuration Files

Format of counter configuration file (argument passed to --counterfile)

To specify a set of performance counters to enable when profiling from the command line, pass the name of a configuration file to the `--counterfile` option. You can generate a counter configuration file from within the Visual Studio client by using the "Save Counters" button on the GPU Profile: Performance Counters page of the Project Settings dialog. The format of this configuration file is one counter name per line. Counter names are case-sensitive. An example of the contents of this file is given below.

```
Wavefronts
VALUInsts
SALUInsts
VFetchInsts
SFetchInsts
VWriteInsts
LDSInsts
GDSInsts
VALUUtilization
VALUBusy
SALUBusy
FetchSize
WriteSize
CacheHit
MemUnitBusy
MemUnitStalled
WriteUnitStalled
LDSBankConflict
```

Format of kernel list configuration file (argument passed to --kernellistfile)

To specify a set of kernels to profile when collecting
performance counters from the command line, pass the name of a configuration file to the --kernellistfile option. The format of this configuration file is one kernel name per line. Kernel names are case-sensitive. When specified, any kernels dispatched by the application that are not contained in the kernel list configuration file will not be profiled. An example of the contents of this file is given below.

```
MatrixMultiplyKernel
binarySearch
binomial_options
```

**Format of API rules configuration file (argument passed to --apirulesfile)**

To specify a set of rules to use when generating the summary pages from a trace file when using the command line, pass the name of a configuration file to the --apirulesfile option. The format of this file is one rule per line in the NAME=VALUE format. An example of the contents of this file is given below. Note that the "VALUE" can be either "True" or "False".

```
APITrace.APIRules.RefTracker=True
APITrace.APIRules.BlockingWrite=False
APITrace.APIRules.BadWorkGroupSize=True
APITrace.APIRules.RetCodeAnalyzer=True
APITrace.APIRules.DataTransferAnalyzer=True
APITrace.APIRules.SyncAnalyzer=True
APITrace.APIRules.DeprecatedFunctionAnalyzer=True
```

**Format of API filter configuration file (argument passed to --apifilterfile)**

To ignore a set of APIs when collecting an API trace using the command line, pass the name of a configuration file to the --apifilterfile option. The format of this file is one API name per line. An example of the contents of this file for an OpenCL™ is given below.

```
clGetPlatformIDs
clGetPlatformInfo
clGetDeviceIDs
clGetDeviceInfo
clGetContextInfo
```
clGetCommandQueueInfo
cGetSupportedImageFormats
cGetMemObjectInfo
cGetImageInfo
cGetSamplerInfo
cGetProgramInfo
cGetProgramBuildInfo
cGetKernelInfo
cGetKernelWorkGroupInfo
cGetEventInfo
cGetEventProfilingInfo

**Format of environment variable file (argument passed to --envvarfile)**

To specify a set of environment variables to be defined for the application being profiled, pass the name of a configuration file to the **--envvarfile** option. The format of this file is one environment variable per line in the NAME=VALUE format. An example of the contents of this file is given below.

```plaintext
APPLICATION_DATA_DIR=c:\path\to\app\data
DEBUG_FLAG=True
LOG_FILE=c:\temp\logfile.log
```

**Format of occupancy display configuration file (argument passed to --occupancydisplay)**

To generate a Kernel Occupancy HTML display file using the command line, pass the name of a configuration file to the **--occupancydisplay** option. The format of this configuration file is one parameter per line in the NAME=VALUE format. An example of the contents of this file is given below. The "VALUES" are taken from a generated .occupancy file for a particular kernel.

```plaintext
ThreadID=3364
CallIndex=101
KernelName=reduce
DeviceName=Capeverde
ComputeUnits=10
MaxWavesPerComputeUnit=40
MaxWorkGroupPerComputeUnit=16
MaxVGPRs=256
MaxSGPRs=512
MaxLDS=32768
```
Using the Command Line Interface

1. Go to the location of the GPU Profiler binaries. The GPU Profiler binaries are located in either the `<x86>` or `<x64>` (<`x86_64` on Linux) subdirectory under the CodeXL installation directory. Alternatively, you can include the location of the Profiler binaries into the system's path environment variable.

2. Run the Profiler using the following instructions.

   **Usage:** `CodeXLGpuProfiler <options> InputApplication [InputApplication's command line arguments]

Note: When profiling JOCL (Java OpenCL) application the full path to the java VM must be provided. E.g `CodeXLGpuProfiler <options> /usr/bin/java -jar <jar_file>

Note: On some Linux systems which have an older version of the libstdc++ shared object, you may get an error when trying to profile an application from the command line. The error will say “Failed to generate profile result”. If you encounter this error, please try using the `CodeXLGpuProfilerRun` shell script to profile. This shell script will first set up the `LD_LIBRARY_PATH` environment variable to allow the profiler to find the correct version of the libstdc++ shared object. The command-line syntax for the CodeXLGpuProfilerRun shell script is identical to the syntax used for CodeXLGpuProfiler.

**General options:**

<p>| <code>-d</code> [ <code>--startdisabled</code> ] | Start the application with profiling disabled, <code>amdtStopProfiling</code> and <code>amdtResumeProfiling</code> are also disabled. |</p>
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-e [ --envvar ] arg</code></td>
<td>Environment variable that should be defined when running the profiler. The variable should be in the format NAME=VALUE.</td>
</tr>
<tr>
<td><code>-E [ --envvarfile ] arg</code></td>
<td>Path to a file containing a list of environment variables for the profiled application. The file should contain one line for each variable in the format NAME=VALUE.</td>
</tr>
<tr>
<td><code>-f [ --fullenv ]</code></td>
<td>The environment variables specified with the <code>--envvar</code> option will be added to the environment block. If not specified, then the environment variables represent additions or changes to the system environment block.</td>
</tr>
<tr>
<td><code>-l [ --list ]</code></td>
<td>Print the list of valid counter names.</td>
</tr>
<tr>
<td><code>-N [ --sessionname ] arg</code></td>
<td>Name of the generated session. If not specified, a default session name will be used.</td>
</tr>
<tr>
<td><code>-o [ --outputfile ] arg</code></td>
<td>Path to OutputFile. If not provided, the default location is the current user's Documents directory; for Linux, the default location is the current user's home directory.</td>
</tr>
<tr>
<td><code>-v [ --version ]</code></td>
<td>Print the CodeXL GpuProfiler version number.</td>
</tr>
<tr>
<td><code>-w [ --workingdirectory ] arg</code></td>
<td>Set the working directory. If not provided, the default working directory will be used.</td>
</tr>
<tr>
<td><code>-h [ --help ]</code></td>
<td>Show a help message.</td>
</tr>
</tbody>
</table>

**Profile mode options:**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-t [ --apitrace ]</code></td>
<td>Trace OpenCL™ application and generate CPU and GPU time stamps and detailed API call traces.</td>
</tr>
<tr>
<td><code>-p [ --perfcounter ]</code></td>
<td>Get the performance counters for each OpenCL™ or DirectCompute kernel dispatched by the application.</td>
</tr>
<tr>
<td><code>-A [ --hsatrace ]</code></td>
<td>Trace HSA application and generate CPU and GPU time stamps and detailed API call traces. (Linux only)</td>
</tr>
</tbody>
</table>

Get the performance counters for
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-C [ --hsapmc ]</td>
<td>each HSA kernel dispatched by the application.</td>
</tr>
<tr>
<td>-P [ --occupancydisplay ] arg</td>
<td>Path to configuration file to use to generate an occupancy display file. Specify the occupancy display file that is to be generated with --outputfile. See below for information about the configuration file format.</td>
</tr>
</tbody>
</table>

**Application Trace mode options (for --apitrace and --hsatrace):**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-F [ --apifilterfile ]</td>
<td>Path to the API filter file which contains a list of OpenCL™ APIs to be filtered out when performing an API trace. See below for information about the API filter file format.</td>
</tr>
<tr>
<td>-i [ --interval ] arg (=100)</td>
<td>Timeout interval. Ignored when not performing an API trace and using timeout mode.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>-m [ --timeout ]</code></td>
<td>Flush Trace data periodically, default timeout interval is 100 milliseconds (can be changed with -i option). Ignored when not performing an API trace. (Windows only, this is the default mode for Linux.)</td>
</tr>
<tr>
<td><code>-M [ --maxapicalls ] (=1000000)</code></td>
<td>Maximum number of API calls.</td>
</tr>
<tr>
<td><code>-n [ --nocollapse ]</code></td>
<td>Do not collapse consecutive identical clGetEventInfo calls into a single call in the trace output. Ignored when not performing an API trace.</td>
</tr>
<tr>
<td><code>-r [ --ret ]</code></td>
<td>Always include the OpenCL™ API return code in API trace, even if client application does not query it. Ignored when not performing an API trace.</td>
</tr>
<tr>
<td><code>-y [ --sym ]</code></td>
<td>Generate symbol information file (.st) for API trace, if available. Ignored when not performing an API trace.</td>
</tr>
</tbody>
</table>

**Performance Counter mode options (for --perfcounter and --hsapmc):**
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-c [ --counterfile ] arg</strong></td>
<td>Path to the counter file to enable selected counters (case-sensitive). If not provided, all counters are used. Ignored when performing an API trace. See below for information about the counter file format.</td>
</tr>
<tr>
<td><strong>-g [ --singlepass ]</strong></td>
<td>Only allow a single pass when collecting performance counters. Any counters that cannot fit into a single pass will be ignored. If specified, the GPUTime will not be collected, as a separate pass is required to query the GPUTime (OpenCL™ or DirectCompute only, this is the default for HSA).</td>
</tr>
<tr>
<td><strong>-G [ --nogputime ]</strong></td>
<td>Skip collection of GPUTime when profiling a kernel (GPUTime requires a separate pass) (OpenCL™ or DirectCompute only, this is the default for HSA).</td>
</tr>
<tr>
<td><strong>-K [ --kernellistfile ] arg</strong></td>
<td>Path to the kernel list file which contains a case-sensitive list of kernels to profile. If not provided, all kernels will be profiled. See below for information about the kernel list file format.</td>
</tr>
</tbody>
</table>
| **-k [ --kerneloutput ] arg** | Output the specified kernel file (OpenCL™ or DirectCompute only). Valid argument values are:  
  - **il**: output kernel IL files  
  - **isa**: output kernel ISA files  
  - **cl**: output kernel CL files  
  - **hsail**: output kernel HSAIL files  
  - **asm**: output DirectCompute shader ASM files  
  - **all**: output all files |
### Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-s [ --outputseparator ] arg</code></td>
<td>Character used to separate fields in the OutputFile. Ignored when performing an API trace.</td>
</tr>
<tr>
<td><code>-x [ --maxkernels ] arg (=100000)</code></td>
<td>Maximum number of kernels to profile.</td>
</tr>
</tbody>
</table>

### Trace Summary mode options (for `--tracesummary`):

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a [ --atpfile ] arg</code></td>
<td>Path to the .atp file from which to generate summary pages. Optional when performing an API trace. Required if -T is specified when not performing an API trace. The handle of the kernel.</td>
</tr>
<tr>
<td><code>-R [ --apirulesfile ] arg</code></td>
<td>Path to OpenCL™ API analyzer configuration file. If not specified, all rules are enabled. Ignored when -tracesummary is not specified. See below for information</td>
</tr>
</tbody>
</table>
Examples

- An example to collect OpenCL™ or DirectCompute performance counters:
  
  ```bash
  CodeXLGpuProfiler -o "/path/to/output.csv" -p "/path/to/app/working/directory" 
  
  
  
  
  "/path/to/app.exe" --device gpu
  ```

- An example to collect an OpenCL™ API trace:
  
  ```bash
  CodeXLGpuProfiler -o "/path/to/output.atp" -t -w "/path/to/app/working/directory" 
  "/path/to/app.exe" --device gpu
  ```

- An example to collect HSA performance counters:
  
  ```bash
  CodeXLGpuProfiler -o "/path/to/output.csv" -C -w "/path/to/app/working/directory" 
  
  
  "/path/to/app.exe"
  ```

- An example to collect an HSA API trace:
  
  ```bash
  CodeXLGpuProfiler -o "/path/to/output.atp" -A -w "/path/to/app/working/directory" 
  "/path/to/app.exe"
  ```

- An example to collect an OpenCL™ API trace with summary pages:
  
  ```bash
  CodeXLGpuProfiler -o "/path/to/output.atp" -t -T -w "/path/to/app/working/directory" 
  "/path/to/app.exe" --device gpu
  ```

- An example to generate summary pages from an .atp file:
  
  ```bash
  CodeXLGpuProfiler -a "/path/to/output.atp" -T
  ```

- An example to generate an occupancy display page:
  
  ```bash
  CodeXLGpuProfiler -P "/path/to/occupancy/params/file.txt" -o "path/to/output.html"
  ```

After you have used the command line to profile an application., you can view the results within CodeXL using the **Import Session** command in the **CodeXL Explorer**.

The format of the configuration files passed to the --counterfile, --apirulesfile, --apifilterfile, --envvarfile and --occupancydisplay
options can be found in the Description of Configuration Files topic.
The AMDT Activity Logger (previously named CLPerfMarkerAMD) library provides a simple host-code instrumentation API that can help you analyze your OpenCL applications.

It lets you instrument your code with calls to `amdtBeginMarker()` and `amdtEndMarker()`. These calls are then used by the GPU Profiler to annotate the host-code timeline in a hierarchical way.

The library also lets you instrument your code with calls to `amdtStopProfiling()` and `amdtResumeProfiling()` to control which parts of your application are profiled.

The following screenshot shows an application that has been instrumented with this API. In the image, the rows labeled `Smoke` and `TeapotOGL` under the `Host Thread 4864` branch represent performance markers added to the host code.
For more information on this API, see the AMDTActivityLogger.pdf file in the AMDTActivityLogger/Doc subdirectory under the CodeXL installation directory.
In Analyze mode, you can compile shaders and kernels for a variety of AMD GPUs and APUs, independent from the GPU/APU that is physically installed on your system, and generate AMD ISA, intermediate language and performance statistics for each target platform. CodeXL Analyzer supports the following inputs:

- OpenCL kernels
- DirectX shaders
- OpenGL and Vulkan programs

- **Switching to Analyze mode**
- **Creating a new project for Analysis**
- **Working with the new CodeXL Analyzer Explorer Tree**
- **Working with Programs**
- **Working with Folders**
- **Build Options - Defining OpenCL and DirectX build options**
- **Output Tab**
- **Kernel Statistics Tab**
- **Shader Statistics Tab**
- **Viewing compilation output: IL and ISA**
- **Export binaries**
- **Remove items from Project**
- **Static Analyze Toolbar - for OpenCL source files**
- **Static Analyze Toolbar - for DirectX source files**
- **CodeXLAnalyzer Command Line Interface**
Switching to Analyze mode

Option 1 - Analyze mode button:
Click on the Analyze Mode button in the CodeXL Mode toolbar:

Option 2 - Main menu:
Open the Analyze menu from menu bar and select the ‘Switch to Analyze Mode’ command:
After you switch to Analyze mode, you can also create a new project, open a previously saved project, or load the Teapot or Matrix Multiply samples.
Creating a new project for Analysis

Click on the “File->Create Project”, or use the Ctrl+N shortcut. The following CodeXL Project Settings dialog will appear:

Rename the project, and click on the OK.

After the new project has been created, the CodeXL Analyzer Explorer Tree should appear in the left pane:
If you are familiar with the former versions of the Analyzer, you probably noticed that the tree has a different structure than the one used in previous versions. Let’s examine the structure of the new CodeXL Analyzer Explorer:

1. **Programs and Folders:** before describing how to technically create Programs and Folders, let’s first discuss what those objects are, and why they can be useful.
   a. **Programs (OpenGL, Vulkan):**
      As of version 2.0, CodeXL can compile and link together multiple source files for OpenGL and Vulkan. This is especially important when different shaders have mutual impact on one another’s ISA and performance statistics. To provide that type of support, CodeXL Analyzer introduced the concept of a Program. There are two types of Programs in CodeXL 2.0:
      - Rendering Programs
      - Compute Programs

A **Rendering Program** represents a graphics pipeline, and can have a single shader attached to each of its stages:
- Vertex
- Tessellation Control
- Tessellation Evaluation
- Geometry
- Fragment

A **Compute Program** represents a compute pipeline, and can have a single compute shader attached to its single stage.

When you build a program that has multiple shaders attached to it, all shaders are being compiled and linked together. This way, you get more
accurate ISA and performance statistics than those generated using previous versions of CodeXL.

b. **Folders (OpenCL, DirectX):**

Folders are logical containers of source files. When you build a folder that has multiple source files attached to it, the source files are simply being built one after the other. Unlike programs, there is no kind of interdependency between the source files in a given folder: when a folder is being built, each source file is being compiled independently. Folders can be used to organize the project, by serving as a logical separator. They can also be used to ease the process of comparing build results, since now the build results are being maintained per-folder: you can create two different Folders, each containing the same source files, but have a different configuration (for example, create two DirectX Folders, each with a different shader model). After building the two Folders, you can toggle between the performance statistics of the two Folders to see the differences.

You may ask yourself why CodeXL does not support the concept of DirectX Programs, just like it does for OpenGL and Vulkan. This is a good point. Supporting DirectX Programs is at a high priority in the Analyzer’s roadmap, and we will do our best to add that feature in the upcoming versions of the product.

**Creating a new Program or Folder**

To create a new Program or a Folder, double-click on the “Create new program/folder” item in CodeXL Analyzer Explorer Tree:

Then, the following dialog would pop-up:
Select the Program/Folder type of choice, and click OK.

Then, the empty Program/Folder would appear in the Explorer Tree. For Example, if you choose an OpenGL Rendering Program, you will see an empty OpenGL Rendering Program created:
Working with Programs

After creating a new program, you will see that it contains an empty placeholder for every pipeline stage. Right-click on any stage to add an existing shader or create a new one:

Note: You can also double-click on a stage to create a new shader and automatically attach it to that Program’s stage.

To build the program, right-click on it and select the Build option, or use the F7 shortcut:
You can also select the Program and manually click on the Build button in the Analyzer toolbar:
Working with Folders

After creating a new OpenCL or DirectX Folder, an empty Folder would be listed in the Explorer Tree:

To create a new source file, and automatically add it to the Folder, double-click on the “Create new source file item...” item of the folder:
To add an existing source file, and automatically add it to the Folder, double-click on the “Add existing source file item…” item of the folder:

To configure the build properties of a source file under a specific Folder, click on that source file and use the Analyzer toolbar’s Type and Entry point drop-down lists. The first sets the type of the shader and the latter specifies the specific target shader (among the shaders in the source file). This
configuration is **Folder-specific**. That is, the same source file can be set with different properties under different Folders. CodeXL will remember those configurations for you.

To configure the build properties of the Folder, click on the Folder and adjust the enabled items in the Analyzer toolbar. For CodeXL 2.0, this is only relevant to the DX Shader Model property of DX Folders:

Once set, the DX Shader Model value will hold for all the shaders in the selected Folder. For example, if you choose 5_0 as the DX Shader Model, any D3D vertex shader in that Folder will be compiled using shader model vs_5_0.

To build the whole Folder, right-click on it and select the Build item:
Unlike the case with Programs, Folders are more flexible as they allow you to build selected source files, without being required to build the whole Folder. To build selected source files, click on the selected source files under the program, while holding the Ctrl key. Then, right-click on one of the selected files and select the build option:
Selecting target devices

CodeXL Analyzer can target a variety of devices, independent of the device that is physically installed on your system. To select the target devices, for which the build would be performed, first click on the Select Devices button in the Analyzer toolbar:

Then, the CodeXL Options dialog would pop-pup with its Analyze tab activated. The devices are grouped by generations. You can use the check boxes to select and remove devices:
Build Options- Defining OpenCL and DirectX build options

In the Static Analyze toolbar, you can define specific OpenCL or HLSL build options:

The Build Options box is a place to set compiler build flags such as \(-x\) clc++ or \(-o3\). Any compiler build flag can be placed in this box.

You can set the build options by typing the options directly in the designated text box or by using the OpenCL/HLSL Build Options dialog.

OpenCL Build Options Dialog

This dialog will help you choose the correct OpenCL build options for you and hopefully will prevent making spelling mistakes while typing the options manually.

To open the dialog, press The button. The dialog will be opened. You can switch between the “General & Optimization” tab and the “Other” tab to view all the available options. Once you choose an option, the option text is displayed in the “OpenCL Build Command Line” text box that appears below. This string will also appear in the menu bar after you click the OK button.
While typing a command in the “OpenCL Build Command Line” text box, you will notice that the relevant controls are being updated accordingly (for example, if you will type “-w”, you will be able to see that the “Disable all warnings” check box becomes checked).

Usage Example: build options

For building the tpAdvectFieldScalar.cl kernel from CodeXL’s
AMDTTeaPot sample project, enter the following options:

-D GRID_NUM_CELLS_X=64 -D GRID_NUM_CELLS_Y=64 -D GRID_NUM_CELLS_Z=64 -D GRID_INV_SPACING=1.000000f -D GRID_SPACING=1.000000f -D GRID_SHIFT_X=6 -D GRID_SHIFT_Y=6 -D GRID_SHIFT_Z=6 -D GRID_STRIDE_Y=64 -D GRID_STRIDE_SHIFT_Y=6 -D GRID_STRIDE_Z=4096 -D GRID_STRIDE_SHIFT_Z=12 -I path_to_example_src

On windows, path_to_example_src should be:

C:\Program Files\CodeXL\Examples\Teapot\res

On Linux, path_to_example_src should be:

/opt//CodeXL/bin/examples/Teapot/AMDTTeaPotLib/AMDTTeaPotLib/

Adding the option `-h` will dump the list of OpenCL compiler available options in the output tab. For additional details, ‘Compile Build Options’ Appendix.

**Build Options**

<table>
<thead>
<tr>
<th>General Options</th>
<th>Optimization Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>-D Predefined macros</td>
<td>-O0, -O1, -O2, -</td>
</tr>
<tr>
<td>-I Additional include directories</td>
<td>Optimization</td>
</tr>
<tr>
<td>-x clc, -x clc++</td>
<td></td>
</tr>
<tr>
<td>-w Disable all warnings</td>
<td></td>
</tr>
<tr>
<td>-Werror Treat any warning as an error</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>-cl-single-precision-constant</td>
<td>Treat double float-point constant as single one</td>
</tr>
<tr>
<td>-cl-denorms-are-zero</td>
<td>Flush denormalized floating point numbers as zeros</td>
</tr>
<tr>
<td>-cl-strict-aliasing</td>
<td>Compiler assumes the strict</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>-cl-mad-enable</code></td>
<td>Enable MAD</td>
</tr>
<tr>
<td><code>-cl-no-signed-zeros</code></td>
<td>Ignore the signedness of zero</td>
</tr>
<tr>
<td><code>-cl-unsafe-math-optimizations</code></td>
<td>Allow unsafe optimization</td>
</tr>
<tr>
<td><code>-cl-finite-math-only</code></td>
<td>Assume no NaN nor infinite</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-cl-fast-relaxed-math</td>
<td>Do aggressive Math Optimization only and -cl-unsafe-math-optimizations, allows optimizations for floating-point arithmetic that may violate the IEEE standard and the OpenCL numerical compliance requirements defined in section 7.4 for single-precision floating-point, section 9.3.9 for double-precision floating-point, and edge case behavior in section 7.5. This option allows the preprocessor macro <strong>FAST_RELAXED_MATH</strong> to be set for the OpenCL program.</td>
</tr>
<tr>
<td>-cl-fp32-correctly-rounded-divide-sqrt</td>
<td>The -cl-fp32-correctly-rounded-divide-sqrt option to clBuildProgram or clCompileProgram allows an application to specify that single-precision floating-point divide (x/y and 1/x) and sqrt used in the program source are correctly rounded. If this build option is not specified, the minimum numerical accuracy for single-precision floating-point divide and sqrt are defined in section 7.4 of the OpenCL specification. This build option can be specified if the CL_FP_CORRECTLY_ROUNDED is set in CL_DEVICE_SINGLE_FP_CONFIG (as defined in the table of allowed values for param_name for clGetDeviceInfo) for devices that the program is being build. clBuildProgram or clCompileProgram will fail to compile the program for a device if the -cl-fp32-correctly-rounded-divide-sqrt option is specified and CL_FP_CORRECTLY_ROUNDED is not set for the device.</td>
</tr>
</tbody>
</table>

**Other Options**
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-cl-std</strong></td>
<td>CL version supported</td>
</tr>
<tr>
<td></td>
<td>Determine the OpenCL C language version to use. A value for this option must be specified.</td>
</tr>
<tr>
<td></td>
<td>Valid values are:</td>
</tr>
<tr>
<td>CL1.1</td>
<td>Support all OpenCL C programs that use the OpenCL C features defined in section 6 of the</td>
</tr>
<tr>
<td></td>
<td>specification.</td>
</tr>
<tr>
<td>CL1.2</td>
<td>Support all OpenCL C programs that use the OpenCL C features defined in section 6 of the</td>
</tr>
<tr>
<td></td>
<td>specification.</td>
</tr>
<tr>
<td><strong>-cl-kernel-arg-info</strong></td>
<td>Kernel argument info</td>
</tr>
<tr>
<td></td>
<td>This option allows the compiler to store information about the arguments of a kernel(s)</td>
</tr>
<tr>
<td></td>
<td>in the program executable. The argument information stored includes the argument name,</td>
</tr>
<tr>
<td></td>
<td>its type, the address and access qualifiers used. Refer to the description of clGetKernelArgInfo</td>
</tr>
<tr>
<td></td>
<td>for information about how to query this information.</td>
</tr>
<tr>
<td><strong>-create-library</strong></td>
<td>Create library</td>
</tr>
<tr>
<td></td>
<td>Create a library of compiled binaries specified in input_programs argument to clLinkProgram.</td>
</tr>
<tr>
<td><strong>-enable-link-options</strong></td>
<td>Enable link options</td>
</tr>
<tr>
<td></td>
<td>Allows the linker to modify the library behavior based on one or more link options in</td>
</tr>
<tr>
<td></td>
<td>Program Linking Options, below, when this library is linked with a program executable.</td>
</tr>
<tr>
<td></td>
<td>This option must be specified with the library option.</td>
</tr>
<tr>
<td><strong>-g</strong></td>
<td>Produce debugging information</td>
</tr>
<tr>
<td></td>
<td>This is an experimental feature that lets you use the GNU project debugger, GDB, to debug</td>
</tr>
<tr>
<td></td>
<td>kernels on x86 CPUs running Linux, and under Windows, with cygwin/minGW. For details, see</td>
</tr>
<tr>
<td></td>
<td>Chapter 3, “Debugging OpenCL.”</td>
</tr>
<tr>
<td></td>
<td>This option does not affect the default optimization of the OpenCL code.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-fper-pointer-uav</td>
<td>UAV per pointer should be used (HD5XXX and HD6XXX series GPU's only)</td>
</tr>
<tr>
<td>-fno-per-pointer-uav</td>
<td></td>
</tr>
<tr>
<td>-fbin-bif30</td>
<td>Allow OpenCL binary to be BIF3.0 format</td>
</tr>
<tr>
<td>-fno-bin-bif30</td>
<td></td>
</tr>
<tr>
<td>-fbin-encrypt</td>
<td>Generate an encrypted OpenCL binary (not by default)</td>
</tr>
<tr>
<td>-fno-bin-encrypt</td>
<td></td>
</tr>
<tr>
<td>-save-temps</td>
<td>Store temporary files in current directory</td>
</tr>
<tr>
<td>-fuse-jit</td>
<td>Use JIT for CPU target</td>
</tr>
</tbody>
</table>
### HLSL Build Options Dialog

This dialog will help you choose the correct HLSL build options for you and hopefully will prevent making spelling mistakes while typing the options manually.

To open the dialog, press The Button. The dialog will be opened. Click the “HLSL Build Options” node to view the available options.

Once you choose an option, the option text is displayed in the “HLSL Build Command Line” text box that appears below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fno-use-jit</td>
<td>(disable if debugging is enabled)</td>
</tr>
<tr>
<td>-fforce-jit</td>
<td>Force use JIT for CPU target (even if debugging is enabled)</td>
</tr>
<tr>
<td>-ffno-force-jit</td>
<td></td>
</tr>
<tr>
<td>-fdisable-avx</td>
<td>Disable AVX code generation</td>
</tr>
<tr>
<td>-fno-disable-avx</td>
<td></td>
</tr>
<tr>
<td>-ffma-enable</td>
<td>Enable fma for a*b+c</td>
</tr>
<tr>
<td>-fno-fma-enable</td>
<td></td>
</tr>
<tr>
<td>-fuse-native</td>
<td>Replace math function calls with native version</td>
</tr>
</tbody>
</table>
This build option string will also appear in the toolbar’s build options box after you click the OK button.

As an alternative to selecting options through the radio buttons, it is possible to type a command in the “HLSL Build Command Line” text box. Build options types in the text box will automatically be translated to update of the relevant controls accordingly. For example, typing “D3DCOMPILE_DEBUG” in the lower text box automatically updates the “Debug” check box to be checked.
Build Options

The compilation of DirectX shaders can be executed either by directly referencing the D3D compiler DLL or by going through Microsoft’s FXC tool.

The CodeXL installation includes a copy of the Microsoft DirectX compiler DLL: d3dcompiler_47.dll. You may specify a different path if you want CodeXL to use a different d3dcompiler module. If you select the FXC compiler tool, you
must specify a path to the location of FXC.exe.

To select the path of the compiler module, click the ‘Browse...” option from the combo-box. When selecting Browse, a dialog box will open for selecting the compiler file.

- For D3D compiler – any file called d3compiler_*.dll can be selected.
- For FXC compiler – only files named FXC.exe can be selected.

<table>
<thead>
<tr>
<th>D3D compile command</th>
<th>FXC compile command</th>
</tr>
</thead>
<tbody>
<tr>
<td>-D</td>
<td></td>
</tr>
<tr>
<td>-I</td>
<td></td>
</tr>
<tr>
<td>Directive</td>
<td>Option</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>D3DCOMPILE_AVOID_FLOW_CONTROL</td>
<td>/Gfa</td>
</tr>
<tr>
<td>D3DCOMPILE_DEBUG</td>
<td>/Zi</td>
</tr>
<tr>
<td>D3DCOMPILE_ENABLE_BACKWARDS_COMPATIBILITY</td>
<td>/Gec</td>
</tr>
<tr>
<td>D3DCOMPILE_ENABLE STRICTNESS</td>
<td>/Ges</td>
</tr>
<tr>
<td>D3DCOMPILE_FORCE_PS_SOFTWARE_NO_OPT</td>
<td></td>
</tr>
<tr>
<td>D3DCOMPILE_FORCE_VS_SOFTWARE_NO_OPT</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Flag</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>D3DCOMPILE_IEEE_STRICTNESS</td>
<td>/Gis</td>
</tr>
<tr>
<td>D3DCOMPILE_NO_PRESHADER</td>
<td>/Op</td>
</tr>
</tbody>
</table>

* D3DCOMPILE_SKIP_OPTIMIZATION
* D3DCOMPILE_OPTIMIZATION_LEVEL0
* .. (no flag for default optimization)
* D3DCOMPILE_OPTIMIZATION_LEVEL2

* /Od
* /O0
* .. (for default optimization)
* /O1
| D3DCOMPILE_OPTIMIZATION_LEVEL3                   | * /O2  
|                                              | * /O3  
| D3DCOMPILE_PACK_MATRIX_COLUMN_MAJOR          | /Zpc  
<p>| D3DCOMPILE_PACK_MATRIX_ROW_MAJOR             | /Zpr  |</p>
<table>
<thead>
<tr>
<th>/Gpp</th>
<th>/Gfp</th>
<th>/res_may_alias</th>
<th>/Vd</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DCOMPILE_PARTIAL_PRECISION</td>
<td>D3DCOMPILE_PREFER_FLOW_CONTROL</td>
<td>D3DCOMPILE_RESOURCES_MAY_ALIAS</td>
<td>D3DCOMPILE_SKIP_VALIDATION</td>
</tr>
<tr>
<td>D3DCOMPILE_WARNINGS_ARE_ERRORS</td>
<td>/WX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/Lx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/Ni</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/No</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note: some of the flags are only relevant to the FXC tool.
Output Tab

The compiler output appears in the Output tab. The example below shows successful builds (no warnings or errors) for 4 devices.

```
--------- Build started: Building cIFolder02 (DCT_Kernels.cl, BitonicSort_Kernels.cl) for 3 devices. ---------
--------- Build started: Building C:\<temp>\AMD_DCT_Kernels.cl for 3 devices. ---------
1) DCT_Kernels for Bonaire:
   Building for Bonaire... succeeded.
2) DCT_Kernels for CapeVerde:
   Building for CapeVerde... succeeded.
3) DCT_Kernels for Cariris:
   Building for Cariris... succeeded.
--------- Build completed for 3 devices: 3 succeeded, 0 failed. ---------
--------- Build started: Building C:\<temp>\AMD_BitonicSort_Kernels.cl for 3 devices. ---------
1) BitonicSort_Kernels for Bonaire:
   Building for Bonaire... succeeded.
2) BitonicSort_Kernels for CapeVerde:
   Building for CapeVerde... succeeded.
3) BitonicSort_Kernels for Cariris:
   Building for Cariris... succeeded.
--------- Build completed for 3 devices: 3 succeeded, 0 failed. ---------
```

If there were errors, the output will display the error and the line where the error occurred:
Double clicking on an error navigates the user to the Source Code view, displaying the kernel source code:

```c
__global float4* curlU,  // Contains curlU
__constant SmokeSimConstants* p)
{
    int3 coord = (int3)(get_global_id(0), get_global_id(1),
                        int index = getIndex(coord);

    // Read in the vorticity vector at this location
    float3 ccU = curlU[index].xyz;
```
Kernel Statistics Tab

The statistics tab gives detailed statistics for the selected kernel for each target device.
To open the statistics tab, expand the desired kernel in the project tree, and double-click the statistics node:
You can see in this statistics output that Northern Islands devices do not use scalar registers (SGPRs and MaxSGPRs are N/A). Northern Islands VGPRs are quad-sized so the 20 VGPRs actually represent 80 float values. You can also see that Southern Islands devices use 44 scalar GPRs that are shared.
across the wavefront and 49 VGPRs that are used per thread equaling 49 float values.

This view puts the emphasis on giving the programmer the wave constraints based on the SGPRs, VGPRs and LDS size.

In the upper section there is a table that shows the current constraints based on the kernels information for the current selected device.

In the lower section, there is a reference table to help the programmer see the effect of the resources usage on the number of concurrent waves.

The LDS is constructed from the static & dynamic values. The dynamic value can be defined by the user to and its possible effect on the constraint can be immediately viewed. LDS is also affected by local workgroups size, so the values can be changed in order to see the impact on the performance of the kernel.

The middle table shows if the “ISA size” and “Scratch Registers” might have impact on the performance and what are the recommended values.
Shader Statistics Tab

The statistics tab gives detailed statistics about the selected shader for each target device. To open the statistics tab expand the desired shader in the project tree, and double-click the statistics node:

There is no support for shader statistics for v5 and earlier generation devices.

Statistics page for GCN devices (v6 and later generations):
The displayed information is explained in the following table:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGPRs</td>
<td>The number of scalar General Purpose Registers allocated by the shader</td>
</tr>
<tr>
<td>VGPRs</td>
<td>The number of vector General Purpose Registers allocated by the kernel. <code>ReqdWorkGroupX</code> - Required workgroup X size specified for kernel. N/A without optional <code>__attribute__((reqd_work_group_size(X, Y, Z)))</code></td>
</tr>
<tr>
<td>ISA size</td>
<td>Compiled code size</td>
</tr>
<tr>
<td>Scratch Registers</td>
<td>The number of scratch registers used by the kernel. If this value is bigger than 0, the shader may be incurring a performance penalty</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MaxSGPRs</td>
<td>The maximum number of scalar General Purpose Registers per kernel supported by the device</td>
</tr>
<tr>
<td>MaxVGPRs</td>
<td>The maximum number of vector General Purpose Registers per kernel supported by the device</td>
</tr>
</tbody>
</table>

This view puts the emphasis on giving the programmer the wave constraints based on the SGPRs and VGPRs.

In the upper section there is a table that shows the current constraints based on the shaders information for the current selected device.

In the lower section there is a reference table to help the programmer see the effect of the resources usage on the number of concurrent waves.

The middle table shows if the “ISA size” and “Scratch Registers” might have impact on the performance and what are the recommended values.
The performance statistics tab will be opened automatically when the build process is over. To view the compilation output, double click the node of the desired ASIC in the explorer tree, under the Program/Folder and configuration (32-bit or 64-bit): 

This will open a tab containing the source code, the IL and the ISA. The program source code and the IL code will be presented as standard text documents. The ISA will be presented in the “Enhanced ISA View” for GCN devices, and as a standard text document for pre-GCN devices.

The context menu also enables showing/hiding line numbers for each source code/IL/ISA tab.
Navigating through ISA code with the Enhanced ISA View

Using this view, you can inspect the ISA code of GCN devices and see the estimation for instruction cost in clock cycle. The view contains 5 columns:

- **Address**: the instruction’s offset within the program (in bytes)
- **Opcode**: the operation to be performed
- **Operands**: the data for the operation
- **Cycles**: the number of clock cycles which are required by a Compute Unit in order to process the instruction for a 64-thread Wavefront, while neglecting the system load and any other runtime-related factor.
- **Instruction Type**: the category of instructions to which the instruction belongs
- **Hex**: binary representation of the instruction, in hexadecimal format

**Notes:**

1. Note that code labels which appear in the Operands column are clickable. By clicking on a label link, you can navigate to the label’s spot in the code.

2. Note that this view is only available for GCN devices. For pre-GCN devices, the plain textual ISA view will be displayed.
Export binaries

You can export the binaries of the last build by right clicking on the folder or file in the explorer tree and selecting the “Export binaries...” option.

When you select the “Export Binaries...” option, an “Export Binaries” dialog will open where you can select the destination folder as well as which parts of the binaries will be included.

You can change the default base name by entering desired name into “Base file name” text field. All binary files will be created with the chosen base name followed by device name.

All the devices that were built will be exported.

* Notice: this option is only supported in OpenCL mode.
Remove items from Project

Programs, Folders and source files can be removed from the project. To remove an item from the project, right-click on it and select the Remove option.
## Static Analyze Toolbar – for OpenCL source files

<table>
<thead>
<tr>
<th>Build command</th>
<th><strong>Build command</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Build options</td>
<td><strong>Kernel Build options</strong></td>
</tr>
<tr>
<td>Opens the “Build Options” dialog</td>
<td><strong>Opens the “Build Options” dialog</strong></td>
</tr>
<tr>
<td>The bitness of the compilation process (for example, choose OpenCL, 64-bit to compile the files in using the 64-bit OpenCL compiler).</td>
<td><strong>The bitness of the compilation process (for example, choose OpenCL, 64-bit to compile the files in using the 64-bit OpenCL compiler).</strong></td>
</tr>
<tr>
<td>Displays the last selected kernel for the selected file. Jumps to the selected kernel in the active source if the source code is open.</td>
<td><strong>Displays the last selected kernel for the selected file. Jumps to the selected kernel in the active source if the source code is open.</strong></td>
</tr>
</tbody>
</table>

© 2015 Advanced Micro Devices. All rights reserved.
## Static Analyze Toolbar – for DirectX shaders

<table>
<thead>
<tr>
<th>Entry point: RenderPS</th>
<th>Build only command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shader Build options</td>
</tr>
<tr>
<td></td>
<td>Opens the “Build Options” dialog</td>
</tr>
<tr>
<td></td>
<td>The bitness of the compilation process (for example, choose OpenCL, 64-bit to compile the files in using the 64-bit OpenCL compiler).</td>
</tr>
<tr>
<td></td>
<td>The selected shader model. This parameter is set once at the Folder level, and holds for all the shaders in that Folder.</td>
</tr>
<tr>
<td></td>
<td>The type of the selected shader (Vertex, Pixel, Hull, etc.)</td>
</tr>
<tr>
<td></td>
<td>The target shader for the compilation process.</td>
</tr>
</tbody>
</table>

© 2015 Advanced Micro Devices. All rights reserved.
CodeXLAnalyzer Command Line Interface

CodeXLAnalyzer (formerly CodeXLKernelAnalyzer) enables compiling and generating performance statistics for OpenCL kernels, DirectX Shaders and OpenGL shaders. The compilation and statistics generation processes can be targeted at a variety of AMD GPUs and APUs, regardless to the actual GPU/APU type that is installed on your system. The application capabilities were expanded. OpenCL Kernels as well as DirectX shaders can now be compiled.

CodeXLAnalyzer can be executed from a shell window using the CodeXLAnalyzer console application which is located in the CodeXL installation directory.

**Note:** On Windows, CodeXLAnalyzer is available in both 32-bit and 64-bit versions. To execute the compilation and statistics generation in 64-bit, invoke CodeXLAnalyzer-x64.exe. On Linux, only a 64-bit version is available.

- **Using CodeXLAnalyzer Command Line Interface to compile OpenCL Kernels**
- **Using CodeXLAnalyzer Command Line Interface to compile DirectX shaders**
- **Using CodeXLAnalyzer Command Line Interface to compile OpenGL and Vulkan programs**
- **Generating and interpreting CodeXLAnalyzer CLI’s live register analysis report**

© 2015 Advanced Micro Devices. All rights reserved.
Using CodeXLAnalyzer Command Line Interface to compile OpenCL Kernels

OpenCL is the default language for CodeXLAnalyzer, so in order to compile OpenCL kernels specifying the input source code language is optional.

CodeXLAnalyzer uses the actual AMD OpenCL Driver installed on the computer, i.e. the Catalyst driver to perform offline compilation.

If no GPU is present, the OpenCL driver installed with APP SDK can be used.

Details of available commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h</code></td>
<td>View available options</td>
</tr>
<tr>
<td><code>CodeXLAnalyzer.exe -h</code></td>
<td></td>
</tr>
<tr>
<td><code>-s</code></td>
<td>Specify the source language for the compilation. “cl” is the default which means that for OpenCL kernel compilation there is no need to specify the <code>-s</code> switch</td>
</tr>
<tr>
<td><code>CodeXLAnalyzer.exe -s cl -l</code></td>
<td></td>
</tr>
<tr>
<td><code>-l [ --list-asics ]</code></td>
<td>List known ASIC targets.</td>
</tr>
<tr>
<td><code>CodeXLAnalyzer.exe -l</code></td>
<td></td>
</tr>
<tr>
<td><code>--verbose</code></td>
<td>View supported ASICs with detailed marketing names</td>
</tr>
<tr>
<td><code>CodeXLAnalyzer.exe -l --verbose</code></td>
<td></td>
</tr>
<tr>
<td><code>--version</code></td>
<td>Print version string.</td>
</tr>
<tr>
<td><code>CodeXLAnalyzer.exe --version</code></td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>-list-kernels</td>
<td>List the kernels functions available in the specify cl file</td>
</tr>
<tr>
<td>--isa arg</td>
<td>Path to output ISA disassembly file(s). This command requires compilation switches identifying the required kernel -kernel.</td>
</tr>
<tr>
<td>--il arg</td>
<td>Path to output IL file(s). Requires --kernel.</td>
</tr>
<tr>
<td>--debugil arg</td>
<td>Path to output Debug IL file(s).</td>
</tr>
<tr>
<td>--metadata arg</td>
<td>Path to output Metadata file(s). Requires --kernel.</td>
</tr>
<tr>
<td>-b [ --binary ] arg</td>
<td>Path to binary output file(s).</td>
</tr>
<tr>
<td>--suppress arg</td>
<td>Section to omit from binary output. Repeatable.</td>
</tr>
<tr>
<td>-k [ --kernel ] arg</td>
<td>Kernel to analyze or make IL or ISA.</td>
</tr>
<tr>
<td>--OpenCLoption</td>
<td>OpenCL compiler options. Repeatable.</td>
</tr>
<tr>
<td>arg</td>
<td>enable-mad --OpenCLoption -w</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>-D [ --define ] arg</td>
<td>Define symbol or symbol=value. Repeatable.</td>
</tr>
<tr>
<td>CodeXLAnalyzer foo.cl --kernel myKernel --isa foo.isa --D myDefine</td>
<td></td>
</tr>
<tr>
<td>--csv-separator arg</td>
<td>Override to default separator for analysis items.</td>
</tr>
<tr>
<td>CodeXLAnalyzer foo.cl --kernel myKernel --analysis foo.csv --csv-separator #</td>
<td></td>
</tr>
<tr>
<td>--livereg arg</td>
<td>Path to the live register analysis output file (note that &quot;--isa arg&quot; must be used in conjunction with the --livereg switch for live register analysis to be performed, since the live register analysis engine works by analyzing the ISA disassembly). Note: this is a beta feature of CodeXLAnalyzer CLI. You can find more info about it in the &quot;Generating and Interpreting CodeXLAnalyzer CLI’s Live Register Analysis Report&quot;</td>
</tr>
<tr>
<td>CodeXLAnalyzer --s cl --c Bonaire --kernel myKernel --isa foo.isa --livereg fooLiveRegFile.txt --il fooIl.il myClFile.cl</td>
<td></td>
</tr>
</tbody>
</table>

**Usage examples:**

Create binary files output/foo-ASIC.bin for foo.cl.

`CodeXLAnalyzer foo.cl --bin outdir/foo`

List the kernels available in foo.cl.

`CodeXLAnalyzer foo.cl --list-kernels`

Generate ISA and performance statistics for all ASICs for kernel myKernel which is defined in foo.cl

`CodeXLAnalyzer foo.cl --kernel myKernel --analysis foo.csv`

List the ASICs that the runtime supports.

`CodeXLAnalyzer --list-asics`

Generate ISA and IL code for Cypress (ASIC) for kernel myKernel which is defined in source file foo.cl
CodeXLAnalyzer foo.cl --kernel myKernel --il foo --isa foo --asic Cypress

Generate ISA code and live register analysis report for Iceland (ASIC), for kernel myKernel which is defined in source file foo.cl

CodeXLAnalyzer foo.cl --kernel myKernel --livereg livereg.txt --isa foo --asic Iceland

© 2015 Advanced Micro Devices. All rights reserved.
Using CodeXLAnalyzer Command Line Interface to compile DirectX shaders

CodeXLAnalyzer command line tool supports offline compilation and statistics generation for DirectX shaders. Naturally, it is supported on Windows only.

The Analyzer works in 2 stages:

1. By default, CodeXLAnalyzer compiles the shader using the D3D Compiler. CodeXL ships with a default compiler (d3dcompiler_47.dll). Unless the --DXLocation command line switch is specified, CodeXL will use the default compiler. The compilation can also go through Microsoft’s FXC tool instead of directly through the D3D compiler. To use FXC, you need to specify the –FXC command line switch with the location of FXC.exe.
2. CodeXLAnalyzer compiles the D3D ASM code generated by the D3D Compiler into AMD ISA, and generates statistics.

Regardless to the chosen compilation chain, all build errors and warnings will be printed in the command line window.

Details of available commands:

<table>
<thead>
<tr>
<th>Details of available commands:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-h</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>-s</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Option</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>-s hlsl -l</td>
</tr>
<tr>
<td>CodeXLAnalyzer.exe -s hlsl -l</td>
</tr>
<tr>
<td>-s hlsl -l --verbose</td>
</tr>
<tr>
<td>CodeXLAnalyzer.exe -s hlsl -l --verbose</td>
</tr>
<tr>
<td>CodeXLAnalyzer.exe -s hlsl -f VsMain -p vs_5_0 -p vs_5_0 c:/files/myShader.fx --isa c:/files/myShader.isa</td>
</tr>
<tr>
<td>--isa arg</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-c [ --asic ] arg</td>
</tr>
<tr>
<td>CodeXLAnalyzer.exe -s hlsl -f VsMain -p vs_5_0 c:/files/myShader.fx --isa c:/files/myShader.isa -c Hawaii -c Kaldini</td>
</tr>
<tr>
<td>-a [ --analysis ]</td>
</tr>
</tbody>
</table>
CodeXLAnalyzer.exe -s hls -f VsMain
c:/files/myShader.fx -p vs_5_0 -a c:/files/myShader.csv

-D

CodeXLAnalyzer.exe -s hls -f VsMain
c:/files/myShader.fx --isa c:/files/myShader.isa -p vs_5_0 --D MyDefine

-- DXFlags

Compile using DXFlags

CodeXLAnalyzer.exe -s hls -f VsMain
c:/files/myShader.fx --isa c:/files/myShader.isa -p vs_5_0 --DXFlagx 1

--version

View the Catalyst driver version installed

CodeXLAnalyzer_d.exe -s hls -version

--DXLocation

Compile with a specific D3Dcompiler DLL. Note that the path need to be in quotes if it contains spaces.

CodeXLAnalyzer.exe -s hls -f VsMain -p vs_5_0
c:/files/myShader.fx --isa c:/files/myShader.isa -c Hawaii --DXLocation "C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler_47.dll"

-s DXAsm

Compile from a blob. In this case, the application assumes D3DCompilation is done so it skips this and does the AMD DX Compilation only.

CodeXLAnalyzer.exe -f VsMain -s DXAsm -p vs_5_0
c:/files/myShader.obj --isa c:\temp\dxTest.isa

Compile using FXC. You need to provide the full FXC path and arguments, and need to use /Fo switch. Also, the output of the FXC
<table>
<thead>
<tr>
<th>--FXC</th>
<th>file should be the input file for KA. Use with the DXAsm switch.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CodeXLAnalyzer.exe -s DXAsm -f VsMain -p vs_5_0 c:/files/myShader.fx --isa c:\files\myShader.isa -c tahiti - -FXC &quot;&quot;C:\Program Files (x86)\Windows Kits\8.1\bin\x86\fxc.exe&quot; /E VsMain /T vs_5_0 /Fo c:/files/myShader.obj c:/files/myShader.fx&quot;</td>
</tr>
<tr>
<td>-s DXAsmT</td>
<td>Compile from a blob represented as text (DX ASM as Text). In this case, the application assumes D3DCompilation is done so it skips this and does the AMD DX Compilation only.</td>
</tr>
<tr>
<td></td>
<td>CodeXLAnalyzer.exe -s DXAsmT -f VsMain -p vs_5_0 c:/files/myShaderAsText.asm --isa c:\files\myShader.isa -c tahiti</td>
</tr>
<tr>
<td>--DumpMSIntermediate arg</td>
<td>When using this option, the MS Compilation output will be saved in the designated file</td>
</tr>
<tr>
<td></td>
<td>CodeXLAnalyzer.exe -s hlsl -f ps_main c:\temp\Pixel.psh --isa c:\temp\dxOutput.isa -c Tahiti -c Bonaire -p ps_5_0 --DumpMSIntermediate c:\temp\DumpMSIntermediate.txt</td>
</tr>
<tr>
<td>--livereg arg</td>
<td>Path to the live register analysis output file (note that &quot;--isa arg&quot; must be used in conjunction with the --livereg switch for live register analysis to be performed, since the live register analysis engine works by analyzing the ISA disassembly). Note: this is a beta feature of CodeXLAnalyzer CLI. You can find more info about it in the “Generating and Interpreting CodeXLAnalyzer CLI’s Live Register Analysis Report”</td>
</tr>
<tr>
<td></td>
<td>CodeXLAnalyzer.exe -s hlsl -c Fiji -f VSMain -p vs_5_0 --isa c:\temp\txt - -livereg c:\temp\reg.txt c:\temp\dx\BasicHLSL11_VS.hlsl</td>
</tr>
</tbody>
</table>
Usage examples:

1. Suppose that you would like to compile and generate the ISA code of a DirectX pixel shader (C:\Users\shaders\Render.hlsl), using a the default D3D compiler that ships with CodeXL, and would like the output files which contain the ISA code to be located at c:\temp\ and be named myISA-<device-name>.isa:

   CodeXLAnalyzer.exe -s hls1 -p ps_5_0 -f RenderPS --isa c:\temp\myISA.isa C:\Users\shaders\Render.hlsl

2. Suppose that you would like to compile and generate the ISA code of a DirectX pixel shader (C:\Users\shaders\Render.hlsl), using a specific D3D compiler (C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler_47.dll), and would like the output files which contain the ISA code to be located at c:\temp\ and be named myISA-<device-name>.isa:

   CodeXLAnalyzer.exe -s hls1 -p ps_5_0 -f RenderPS --DXLocation "C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler_47.dll" --isa c:\temp\myISA.isa C:\Users\shaders\Render.hlsl

3. Suppose that you would like to compile and generate the ISA code of a DirectX pixel shader (C:\Users\shaders\Render.hlsl) only for “Iceland”, using a specific D3D compiler (C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler_47.dll), and would like the output files which contain the ISA code to be located at c:\temp\ and be named myISA-<device-name>.isa:

   CodeXLAnalyzer.exe -s hls1 -p ps_5_0 -f RenderPS --c Iceland --DXLocation "C:\Program Files (x86)\Windows Kits\8.1\bin\x86\d3dcompiler_47.dll" --isa c:\temp\myISA.isa
Using CodeXLAnalyzer Command Line Interface to compile OpenGL and Vulkan programs

CodeXLAnalyzer.exe command line tool supports compilation and statistics generation OpenGL and Vulkan programs.

**Details of available commands:**

<table>
<thead>
<tr>
<th>Details of available commands:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>View available options</td>
</tr>
<tr>
<td>View available options</td>
<td>CodeXLAnalyzer.exe -h</td>
</tr>
<tr>
<td>-s</td>
<td>Specify the source platform for the compilation.</td>
</tr>
<tr>
<td>Specify the source platform for the compilation.</td>
<td>CodeXLAnalyzer.exe -s opengl</td>
</tr>
<tr>
<td></td>
<td>CodeXLAnalyzer.exe -s vulkan</td>
</tr>
<tr>
<td>-s &lt;platform&gt; -l</td>
<td>View supported ASICS for DirectX (also- this is default ASICS list for compilation)</td>
</tr>
<tr>
<td>View supported ASICS for DirectX (also- this is default ASICS list for compilation)</td>
<td>CodeXLAnalyzer.exe -s opengl -l</td>
</tr>
<tr>
<td>--isa arg</td>
<td>Path to output ISA disassembly file(s).</td>
</tr>
<tr>
<td>Path to output ISA disassembly file(s).</td>
<td></td>
</tr>
<tr>
<td>--vert &lt;arg&gt;</td>
<td>Full path to vertex shader’s location</td>
</tr>
<tr>
<td>Full path to vertex shader’s location</td>
<td></td>
</tr>
<tr>
<td>--tesc &lt;arg&gt;</td>
<td>Full path to tessellation control shader’s location</td>
</tr>
<tr>
<td>Full path to tessellation control shader’s location</td>
<td></td>
</tr>
<tr>
<td>--tese &lt;arg&gt;</td>
<td>Full path to tessellation evaluation shader’s location</td>
</tr>
<tr>
<td>Full path to tessellation evaluation shader’s location</td>
<td></td>
</tr>
<tr>
<td>--geom</td>
<td>Full path to geometry shader’s location</td>
</tr>
<tr>
<td>Full path to geometry shader’s location</td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--frag &lt;arg&gt;</td>
<td>Full path to fragment shader’s location</td>
</tr>
<tr>
<td>-c [ --asic ] arg</td>
<td>By default, compilation will be done to all default available devices. This option is repeatable.</td>
</tr>
<tr>
<td>-a [ --analysis ]</td>
<td>Path to the performance statistics output file. Requires compilation.</td>
</tr>
<tr>
<td>--version</td>
<td>View the Catalyst driver version installed</td>
</tr>
</tbody>
</table>

Usage examples:

1. To build an OpenGL program with a vertex shader and a fragment shader attached and generate ISA
   
   CodeXLAnalyzer.exe -s opengl --vert c:\shaders\glVertex.vert --geom c:\shaders\glGeom.geom --isa c:\output\myISA.txt

2. To build a Vulkan program with a vertex shader and a geometry shader attached, and generate ISA and performance statistics
   
   CodeXLAnalyzer.exe -s vulkan --vert c:\shaders\vkVertex.glsl --geom c:\shaders\vkGeom.geom --isa c:\output\myISA.isa --a c:\output\myStats.txt
Generating and Interpreting CodeXLAnalyzer CLI’s Live Register Analysis Report

Using CodeXLAnalyzer CLI’s live register analysis report, you can better understand the register usage of your HLSL shaders and OpenCL kernels throughout their execution. Live register analysis is a beta feature of CodeXLAnalyzer CLI, and it currently only fully supports HLSL shaders and partially supports OpenCL kernels.

Generating a live register analysis report for your kernel or shader:
As mentioned in the “Details of available commands” section above, in order to generate a live register analysis report, you need to make sure that your invocation command includes the following command line switches:

1. --isa <arg> which instructs CodeXLAnalyzer to generate ISA disassembly for your kernel/shader
2. --livereg <arg> which instructs CodeXLAnalyzer to perform a live register analysis of the generated ISA disassembly

Usage examples:
CodeXLAnalyzer.exe -s cl -c Fiji --kernel DCT --isa c:\output\isa --livereg c:\output\livereg.txt DCT_Kernels.cl

Let’s break down the above command to understand its structure:

1. “-s cl” instructs CodeXLAnalyzer to work in OpenCL mode
2. “-c Fiji” sets Fiji as the target ASIC
3. “--kernel DCT” sets DCT as the target kernel (this is the kernel to be analyzed; it is defined in DCT_Kernels.cl, which is the last argument in the above command)
4. “--isa c:\output\isa” instructs CodeXLAnalyzer to generate an ISA disassembly file and save it in c:\output with a “.isa” file extension. The output file name is generated automatically.
5. “--livereg c:\output\livereg.txt” instructs CodeXLAnalyzer to perform live register analysis, save the report in c:\output, and use “livereg.txt” as the report file name’s suffix and extension.

After running the above command, we see the following output files in c:\output (our
destination folder):  
Fiji_DCT.isa  
Fiji_DCT_livereg.txt  

The live register analysis report file is Fiji_DCT_livereg.txt.  

For HLSL, the usage is similar:  
CodeXLAnalyzer.exe -s hlsl -c Fiji -f VSMain -p vs_5_0 --isa c:\temp\txt --livereg c:\temp\lreg.txt c:\temp\dx\BasicHLSL11_VS.hlsl  
1. “-s hlsl” instructs CodeXLAnalyzer to work in HLSL mode  
2. “-c Fiji” sets Fiji as the target ASIC  
3. “-f VSMain” sets VSMain as the target shader  
4. “--isa c:\output\isa” instructs CodeXLAnalyzer to generate an ISA disassembly file and save it in c:\output with a “.isa” file extension. The output file name is generated automatically.  
5. “--livereg c:\output\livereg.txt” instructs CodeXLAnalyzer to perform live register analysis, save the report in c:\output, and use “livereg.txt” as the report file name’s suffix and extension.  

Report structure:  
If you open up the live register analysis report file, you will see that it is a plain textual file. Each line in the file gives a snapshot of the register usage when the PC is at that specific ISA line. Each line in the report is of the following format:  

```
<line number> | <number of live registers> | <list of registers + access type> | <ISA instruction>
```

Where:  
1. <line number> is the number of the current ISA disassembly line  
2. <number of live registers> is the number of live registers when the PC is at that ISA line  
3. <list of registers + access type> is a list of n columns. Each column (except for the first one) refers to a register:  
   a. ‘^’ indicates a register is written to  
   b. ‘v’ indicates a register is read  
   c. ‘x’ is used for a register which is written and read  
   d. ‘:’ is used for register where the contents must be preserved across this instruction (live register)  
   e. A blank means that the register is not used  
4. <ISA instruction> is the ISA disassembly of the relevant instruction  

At the end of the report, you will find a summary in the
following format:

Maximum # VGPR used <Max VGPR used>, # VGPR allocated: <Number of VGPR allocated>

Where:

1. <Max VGPR used> is the number of VGPRs actually used throughout the code
2. <Number of VGPR allocated> is the number of VGPRs that were allocated

Two things to remember when inspecting the live register analysis report are:

1. If the number of live registers is lower than the number of allocated registers, it indicates that the SC could reduce VGPRs without spilling by introducing moves.
2. If registers have a very long liveness range without read/write access, those registers could be likely spilled at low cost.

Here is a sample live register analysis report:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>label_basic_block_1: s_swapc_b64 s[2:3], s[2:3]</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_andn2_b32 s0, s9, 0x3fff0000</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_mov_b32 s1, s0</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_mov_b32 s2, s10</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_mov_b32 s3, s11</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_mov_b32 s0, s8</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_buffer_load_dwordx8 s[4:11], s[0:3], 0x00</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_buffer_load_dwordx8 s[12:19], s[0:3], 0x20</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_waitcnt lgkmcnt(0)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td>^</td>
<td></td>
<td></td>
<td>v_mul_f32 v0, s7, v7</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>^</td>
<td></td>
<td>^</td>
<td></td>
<td></td>
<td>v_mul_f32 v1, s11, v7</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
<td>^</td>
<td></td>
<td></td>
<td>v_mul_f32 v2, s15, v7</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td></td>
<td></td>
<td>^</td>
<td></td>
<td></td>
<td>v_mul_f32 v3, s19, v7</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td></td>
<td></td>
<td>x:::v:::</td>
<td></td>
<td></td>
<td>v_mac_f32 v0, s6, v6</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td></td>
<td></td>
<td>x:::v:::</td>
<td></td>
<td></td>
<td>v_mac_f32 v1, s10, v6</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td></td>
<td></td>
<td>x:::v:::</td>
<td></td>
<td></td>
<td>v_mac_f32 v2, s14, v6</td>
</tr>
<tr>
<td>17</td>
<td>12</td>
<td></td>
<td></td>
<td>x:::v:::</td>
<td></td>
<td></td>
<td>v_mac_f32 v3, s18, v6</td>
</tr>
<tr>
<td>18</td>
<td>11</td>
<td></td>
<td></td>
<td>x:::v:::</td>
<td></td>
<td></td>
<td>v_mac_f32 v0, s5, v5</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td></td>
<td></td>
<td>x:::v:::</td>
<td></td>
<td></td>
<td>v_mac_f32 v1, s9, v5</td>
</tr>
</tbody>
</table>
Maximum # VGPR used 13, # VGPR allocated: 14
CodeXL's Power Profiler is a powerful tool to help analyze the energy efficiency of systems based on AMD APUs and majority of the recent dGPU (discrete GPU).

Features of the Power Profiler include:

- Report the following data:
  - Estimated average power consumed by APU and supported dGPU subcomponents.
  - Average frequency of the CPU cores and the internal GPU and supported dGPU.
  - Thermal trend of the CPU compute-units and the internal GPU.
  - Thermal trend of supported dGPU.
  - CPU cores P-States.
- A command-line tool to for data collection and dump to text/binary format.
- The CodeXL graphic client provides these Power Profiling capabilities:
  - Real-time monitoring
  - Timeline view
  - Summary view
  - Offline review of session data
- Following hardware's are supported
  - AMD APUs: Carrizo, Kaveri, Mullins, Temash, Stoney, Bristol
  - AMD dGPUs: Graphics IP 7 GPUs, Radeon and FirePro models.

[Install the Power Profiler Linux Driver]

[Wider Linux power profiling support (DKMS)]

[Power Profiler’s Performance Counters]

[Power Profiler Command Line Interface]

[Power Profiler Project Settings]

[Power Profiler Counters Selection Dialog]

[Power Profiler Session Tree]

[Power Profiler Timeline View]
Legal Disclaimer

The reports generated from this Software may include thermal and power estimates which may contain errors. The report values may deviate from the actual values and/or the published specifications. Therefore, these reports should not be relied upon for any commercial or production systems and the user is responsible for obtaining accurate thermal and power measurements.

© 2015 Advanced Micro Devices. All rights reserved.
Installing the Power Profiler Linux Driver

On Linux systems, the CodeXL Debian and RPM packages perform the driver installation automatically. However, if you’ve downloaded the CodeXL tar archive, you have to install the Power Profiler’s Linux driver manually. This includes a simple step of running `AMDTPwrProfDriver.sh` script with root credentials.

Example:

```bash
$ tar -xf CodeXL_Linux_x86_64_2.0.XXXX.tar.gz
$ cd CodeXL_Linux_x86_64_2.0.XXXX
$ sudo ./AMDTPwrProfDriver.sh install
```

Installer will create a source tree for power profiler driver under `/usr/src/amdtPwrProf -<version number>`. All the source files required for module compilation is located in this directory are under MIT license.

To uninstall the driver run the following command:

```bash
$ $ cd <codexl-install-dir>
$ sudo ./AMDTPwrProfDriver.sh uninstall
```

© 2015 Advanced Micro Devices. All rights reserved.
Wider Linux power profiling support (DKMS)

On Linux machine Power Profiler driver can also be installed with Dynamic Kernel Module Support (DKMS) framework support. DKMS framework automatically upgrades the power profiler driver module whenever there is a change in the existing kernel. This saves user from manually upgrading the power profiler driver module.

The DKMS package needs to be installed on target machines before running the installation steps mentioned in the above section. AMDTPwrProfDriver.sh installer script will automatically takes care of DKMS related configuration if DKMS package is installed in the target machine.

Example (for Ubuntu system):
$ sudo apt-get install dkms
$ tar –xf CodeXL_Linux_x86_64_2.0.XXXX.tar.gz
$ cd CodeXL_Linux_x86_64_2.0.XXXX
$ sudo ./AMDPwrProfDriver.sh install

If the user upgrades the kernel version frequently it is recommended to use DKMS for installation.
## Power Profiler’s Performance Counters

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total APU Power</td>
<td></td>
<td>Average APU Power for the sampling period, reported in Watts. This is an estimated consumption value which is calculated based on APU activity levels.</td>
<td></td>
</tr>
<tr>
<td>CPU Compute Unit 0 Power</td>
<td></td>
<td>Average CPU Compute Unit Power for the sampling period, reported in Watts. This is an estimated consumption value which is calculated based on APU activity levels.</td>
<td></td>
</tr>
<tr>
<td>CPU Compute Unit 1 Power</td>
<td></td>
<td>Average Integrated-GPU Power for the sampling period, reported in</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>iGPU Power</td>
<td>Watts. This is an estimated consumption value which is calculated based on APU activity levels.</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCIe-Controller Power</td>
<td>Average PCIe-Controller Power for the sampling period, reported in Watts. This is an estimated consumption value which is calculated based on APU activity levels. This value does not include the power consumed by PCIe devices connected to the PCIe bus.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory-Controller Power</td>
<td>Average DDR Memory-Controller Power for the sampling period, reported in Watts. This is an estimated consumption value which is calculated based on APU activity levels. This value does not include</td>
<td></td>
</tr>
<tr>
<td>Display-Controller Power</td>
<td>the power consumed by the memory DIMMs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Display-Controller Power</td>
<td>Average Display-Controller Power for the sampling period, reported in Watts. This value refers to the APU's internal display controller which may be used in notebook and embedded configurations. This is an estimated consumption value which is calculated based on APU activity levels. This value does not include the power consumed by the display.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cumulative APU Power</td>
<td>The accumulated energy consumed by the APU throughout the profile session. Reported in Joules. Available only in the command line tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cumulative Compute Unit</td>
<td>The accumulated energy consumed by the CPU Available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
<td>Availability</td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>0 Power Cumulative Compute Unit 1 Power</td>
<td>Compute Unit throughout the profile session. Reported in Joules.</td>
<td>only in the command line tool</td>
<td></td>
</tr>
<tr>
<td>Cumulative iGPU Power</td>
<td>The accumulated energy consumed by the APU’s Internal GPU throughout the profile session. Reported in Joules.</td>
<td>Available only in the command line tool</td>
<td></td>
</tr>
<tr>
<td>dGPU power</td>
<td>Average Discrete-GPU Power for the sampling period, reported in Watts. This is an estimated consumption value which is calculated based on dGPU activity levels.</td>
<td>The dGPU family name is prefixed with this counter name.</td>
<td></td>
</tr>
<tr>
<td>CPU Core 0 Average Frequency</td>
<td>Average CPU Core Frequency for the sampling period, reported in MHz. This is the Core Effective Frequency (CEF). The core can go into various P-States within the sampling period,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Core 3 Average Frequency</td>
<td>each with its own frequency. The CEF is the average of the core frequencies over the sampling period.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iGPU Average Frequency</td>
<td>Average Integrated-GPU Frequency for the sampling period, reported in MHz.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dGPU Average Frequency</td>
<td>Average Discrete-GPU Frequency for the sampling period, reported in MHz. The dGPU family name is prefixed with this counter name.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Core 0 Frequency Histogram</td>
<td>Histogram of CPU Core Effective Frequency (average frequency for the sampling period). Available only in the command line tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Core 1 Frequency Histogram</td>
<td>CPU Core 2 Frequency Histogram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Core 3 Frequency Histogram</td>
<td>iGPU Frequency Histogram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram of Internal-GPU Effective Frequency (average</td>
<td>Available only in the command line tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Compute Unit 0</td>
<td>Measured Temperature</td>
<td>CPU Compute Unit 1</td>
<td>Measured Temperature</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------</td>
<td>--------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frequency for the</td>
<td></td>
<td>measured CPU</td>
<td></td>
</tr>
<tr>
<td>sampling period)</td>
<td></td>
<td>Compute Unit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average Temperature,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reported in</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Celsius. The</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reported value is</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>normalized and</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled, relative to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the specific</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>processor's</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>maximum operating</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>temperature. This</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>value can be used</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>to indicate rise</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>and decline of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>temperature.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Measured</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integrated-GPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average Temperature,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reported in</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Celsius. The</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reported value is</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>normalized and</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled, relative to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the specific</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>processor's</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>maximum operating</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>temperature. This</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>value can be used</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>to indicate rise</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>and decline of</td>
<td></td>
</tr>
<tr>
<td><strong>dGPU Measured Temperature</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measured Discrete-GPU Average Temperature, reported in Celsius. The reported value is normalized and scaled, relative to the specific processor's maximum operating temperature. This value can be used to indicate rise and decline of temperature.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>The dGPU family name is prefixed with this counter name.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CPU Core State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core 0 P-State</td>
</tr>
<tr>
<td>CPU Core 1 P-State</td>
</tr>
<tr>
<td>CPU Core 2 P-State</td>
</tr>
<tr>
<td>CPU Core 3 P-State</td>
</tr>
<tr>
<td>CPU Core P-State at the time when sampling was performed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Other counters</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0 Process ID</td>
</tr>
<tr>
<td>Core 1 Process ID</td>
</tr>
<tr>
<td>Core 2 Process ID</td>
</tr>
<tr>
<td>Core 3 Process ID</td>
</tr>
<tr>
<td>Process Id of the process that was executed by the CPU core at the point in time when the core was sampled.</td>
</tr>
<tr>
<td>Available only in the command line tool</td>
</tr>
</tbody>
</table>
Power Profiler Command Line Interface

CodeXL Power Profiler provides a command line interface utility for users who prefer to use command interpreters like cmd.exe on Windows and bash on Linux. This CLI utility can be used to collect and analyze the profile data. It can also be used from a batch file or a test script.

Usage:
On Windows:
   CodeXLPowerProfiler.exe <options>
On Linux:
   CodeXLPowerProfiler <options>

The following options are supported:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-P &lt;profile options&gt;</td>
<td>Specify what types of data will be collected during the profile session. The following data types are supported.</td>
</tr>
<tr>
<td></td>
<td><strong>power</strong> - collect all the available power counters</td>
</tr>
<tr>
<td></td>
<td><strong>temperature</strong> - collect all the available temperature counters</td>
</tr>
<tr>
<td></td>
<td><strong>frequency</strong> - collect all the available frequency counters</td>
</tr>
<tr>
<td></td>
<td><strong>cu_power</strong> - collect cpu compute-unit power counters</td>
</tr>
<tr>
<td></td>
<td><strong>cu_temperature</strong> - collect cpu compute-unit temperature counters</td>
</tr>
<tr>
<td></td>
<td><strong>gpu_power</strong> - collect gpu power counters</td>
</tr>
<tr>
<td><strong>gpu_temperature</strong></td>
<td>collect gpu temperature counters</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><strong>core</strong></td>
<td>collect core specific attributes. This includes core frequency, p-state and process id executing on the core.</td>
</tr>
<tr>
<td><strong>all</strong></td>
<td>collect all the supported counters</td>
</tr>
</tbody>
</table>

| **-l**               | List all the counters supported by the local hardware, and the hardware devices that the profiler recognizes. |
|                      | The counter IDs can be used with the ‘-e’ option. |
|                      | The hardware device IDs can be used with the ‘-D’ option. |

| **-e <counter,...>** | Specify the comma separated list of counter names to be collected. |
|                      | Use option ‘-l’ to get the supported counter names. |
|                      | Note: use any one of the options -P or -e. |

| **-D <counter,...>** | Specify the comma separated list of device ids. All the counters of these devices will be profiled and collected. |
|                      | Use option (-l) to get the supported devices. |

| **-T <sampling interval>** | Sampling interval in milli-seconds. The minimum value is 20ms. |

| **-d <duration>** | Profile duration in seconds. |

| **-o <path>** | Specify the output file path. The default path will be %Temp\Codexl-Power_<timestamp> on Windows and /tmp/Codexl-Power_<timestamp> on Linux. |

<table>
<thead>
<tr>
<th><strong>Define the output file format:</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>csv - Comma Separated Value text file</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>`-F &lt;csv</td>
<td>txt&gt;`</td>
</tr>
<tr>
<td><code>-C &lt;core mask&gt;</code></td>
<td>Specify core affinity mask for the application to be launched. Default affinity mask is all the available CPU cores.</td>
</tr>
<tr>
<td></td>
<td>0x1 = Core 0</td>
</tr>
<tr>
<td></td>
<td>0x2 = Core 1</td>
</tr>
<tr>
<td></td>
<td>0x4 = Core 2</td>
</tr>
<tr>
<td></td>
<td>0x8 = Core 3</td>
</tr>
<tr>
<td><code>-b</code></td>
<td>Terminate the launched application after the specified profile duration.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>Specify the working directory. Default will be the path of the launched application.</td>
</tr>
<tr>
<td><code>-h</code></td>
<td>Displays this help information.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Print version string.</td>
</tr>
<tr>
<td><code>-z &lt;db file output dir&gt;</code></td>
<td>Export results to a *.cxldb file which can be imported to CodeXL GUI application.</td>
</tr>
<tr>
<td><code>-M process</code></td>
<td>Process profiling is based on the IPC load. Collects power consumption of all running processes during their profile run. These power values may differ with the actual power consumption. These power values can be used to get a notion of power trend and relative power consumption among running processes.</td>
</tr>
<tr>
<td></td>
<td>Note: Currently PMC counters are supported only on windows operating system. Supported AMD platforms are Kaveri, Mullins, Temash and Carrizo. Please refer limitation section for further details.</td>
</tr>
</tbody>
</table>
Examples

- Collect power values for CPU Compute Units and GPU for the duration of 10 seconds, with sampling interval of 100 milliseconds:
  CodeXLPowerProfiler.exe –P cpu_power -P gpu_power -o c:\Temp\powerprof-out.txt –T 100 –d 10
- Collect all the supported counter values for the duration of 10 seconds with sampling interval of 100 milliseconds:
  CodeXLPowerProfiler.exe –P all -o c:\Temp\powerprof-out.txt –T 100 –d 10
- Collect all the supported counter values for the duration of 300 seconds with sampling interval of 100 milliseconds, and output the data to a binary file that can be imported into the graphic client:
  CodeXLPowerProfiler.exe –P all –C 0x3 –z c:\Temp\PowerOutput –T 100 –d 300
- Collect all running process with their energy consumption shares as well as all counters during a particular profile run. Where profile duration is set to 300 seconds and profile sampling period is set to 10 milliseconds.
  CodeXLPowerProfiler.exe –M process –e all c:\Temp\PowerOutput –T 10 –d 300
- Display help:
  CodeXLPowerProfiler.exe -h
- Display version string:
  CodeXLPowerProfiler.exe -v

Sample Text output files

Sample 1: Collecting non-cumulative counters

The following command collects all non-cumulative power counters for 10 seconds, sampling them every 500 milliseconds and dumping the results to a text file:

CodeXLPowerProfiler.exe -P power -d 10 -T 500 -o C:\temp\pwr_out.txt –F txt

When run on a Kaveri APU, the content of the result text file is as below:

CODEXL POWER PROFILE REPORT

PROFILE DETAILS

CPU Details: Family(0x15) Model(48)
CPU Core Mask: 0xf
Sampling Interval: 500 milli-seconds
Profile Start Time: Dec-09-2014_18-10-08
Profile Duration: 10 seconds

PROFILED COUNTERS
as	below:

When run on a Kaveri APU, the content of the result text file is

CodeXLPowerProfiler.exe -d 10 -o c:\temp\cumulative.txt -e 3,6,17,28 -F txt

When run on a Kaveri APU, the content of the result text file is as below:

Sample 2: Collecting cumulative counters

The following command collects the cumulative power counters for 10 seconds and dumping the results to a text file:

CodeXLPowerProfiler.exe -d 10 -o c:\temp\cumulative.txt -e 3,6,17,28 -F txt
PROFILE DETAILS
CPU Details: Family(0x15) Model(48)
CPU Core Mask: 0xf
Sampling Interval: 100 milli-seconds
Profile Start Time: Dec-11-2014_15-57-44
Profile Duration: 10 seconds

PROFILED COUNTERS
<table>
<thead>
<tr>
<th>COUNTER ID</th>
<th>NAME</th>
<th>CATEGORY</th>
<th>UNIT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>total-apu-power-cuml</td>
<td>Power</td>
<td>Joule</td>
<td>Cumulative APU Power, reported in Joules.</td>
</tr>
<tr>
<td>6.</td>
<td>cpu-cu0-power-cuml</td>
<td>Power</td>
<td>Joule</td>
<td>Cumulative CPU Compute Unit Power, reported in Joules.</td>
</tr>
<tr>
<td>17.</td>
<td>cpu-cu1-power-cuml</td>
<td>Power</td>
<td>Joule</td>
<td>Cumulative CPU Compute Unit Power, reported in Joules.</td>
</tr>
</tbody>
</table>

CUMULATIVE COUNTERS
<table>
<thead>
<tr>
<th>COUNTER</th>
<th>CUMULATIVE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>total-apu-power-cuml</td>
<td>228.11</td>
</tr>
<tr>
<td>cpu-cu0-power-cuml</td>
<td>48.85</td>
</tr>
<tr>
<td>cpu-cu1-power-cuml</td>
<td>50.30</td>
</tr>
<tr>
<td>igpu-power-cuml</td>
<td>78.65</td>
</tr>
</tbody>
</table>

Sample 3: Collecting histogram counters

The following command collects the frequency histogram counters for 10 seconds and dumping the results to a text file:

CodeXLPowerProfiler.exe -d 10 -o c:\temp\histogram.txt -e 10,14,21,25,31 –F txt

When run on a Kaveri APU, the content of the result text file is as below:

CODEXL POWER PROFILE REPORT

PROFILE DETAILS
CPU Details: Family(0x15) Model(48)
CPU Core Mask: 0xf
Sampling Interval: 100 milli-seconds
Profile Start Time: Dec-11-2014_16-00-35
Profile Duration: 10 seconds

**PROFILED COUNTERS**

<table>
<thead>
<tr>
<th>COUNTER ID</th>
<th>NAME</th>
<th>CATEGORY</th>
<th>UNIT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.</td>
<td>cpu-core0-frequency-hist</td>
<td>Frequency</td>
<td>MHz</td>
<td>Histogram of CPU Core Effective Frequency.</td>
</tr>
<tr>
<td>14.</td>
<td>cpu-core1-frequency-hist</td>
<td>Frequency</td>
<td>MHz</td>
<td>Histogram of CPU Core Effective Frequency.</td>
</tr>
<tr>
<td>21.</td>
<td>cpu-core2-frequency-hist</td>
<td>Frequency</td>
<td>MHz</td>
<td>Histogram of CPU Core Effective Frequency.</td>
</tr>
<tr>
<td>25.</td>
<td>cpu-core3-frequency-hist</td>
<td>Frequency</td>
<td>MHz</td>
<td>Histogram of CPU Core Effective Frequency.</td>
</tr>
<tr>
<td>31.</td>
<td>igpu-frequency-hist</td>
<td>Frequency</td>
<td>MHz</td>
<td>Histogram of Integrated-GPU Frequency.</td>
</tr>
</tbody>
</table>

**HISTOGRAMS OF COUNTERS**

**COUNTER** cpu-core0-frequency-hist

**HISTOGRAM**

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>600</td>
<td>800</td>
<td>16</td>
</tr>
<tr>
<td>800</td>
<td>1000</td>
<td>16</td>
</tr>
<tr>
<td>1000</td>
<td>1200</td>
<td>5</td>
</tr>
<tr>
<td>1200</td>
<td>1400</td>
<td>22</td>
</tr>
<tr>
<td>1400</td>
<td>1600</td>
<td>29</td>
</tr>
<tr>
<td>1600</td>
<td>1800</td>
<td>11</td>
</tr>
<tr>
<td>1800</td>
<td>2000</td>
<td>1</td>
</tr>
<tr>
<td>2000</td>
<td>2200</td>
<td>0</td>
</tr>
<tr>
<td>2200</td>
<td>2400</td>
<td>0</td>
</tr>
<tr>
<td>2400</td>
<td>2600</td>
<td>0</td>
</tr>
<tr>
<td>2600</td>
<td>2800</td>
<td>0</td>
</tr>
<tr>
<td>2800</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>3200</td>
<td>0</td>
</tr>
<tr>
<td>3200</td>
<td>3400</td>
<td>0</td>
</tr>
<tr>
<td>3400</td>
<td>3600</td>
<td>0</td>
</tr>
<tr>
<td>3600</td>
<td>3800</td>
<td>0</td>
</tr>
<tr>
<td>3800</td>
<td>4000</td>
<td>0</td>
</tr>
</tbody>
</table>

**COUNTER** cpu-core1-frequency-hist

**HISTOGRAM**

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td>low</td>
<td>high</td>
<td>count</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>200</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>600</td>
<td>0</td>
</tr>
<tr>
<td>600</td>
<td>800</td>
<td>6</td>
</tr>
<tr>
<td>800</td>
<td>1000</td>
<td>79</td>
</tr>
<tr>
<td>1000</td>
<td>1200</td>
<td>13</td>
</tr>
<tr>
<td>1200</td>
<td>1400</td>
<td>1</td>
</tr>
<tr>
<td>1400</td>
<td>1600</td>
<td>1</td>
</tr>
<tr>
<td>1600</td>
<td>1800</td>
<td>0</td>
</tr>
<tr>
<td>1800</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>2200</td>
<td>0</td>
</tr>
<tr>
<td>2200</td>
<td>2400</td>
<td>0</td>
</tr>
<tr>
<td>2400</td>
<td>2600</td>
<td>0</td>
</tr>
<tr>
<td>2600</td>
<td>2800</td>
<td>0</td>
</tr>
<tr>
<td>2800</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>3200</td>
<td>0</td>
</tr>
<tr>
<td>3200</td>
<td>3400</td>
<td>0</td>
</tr>
<tr>
<td>3400</td>
<td>3600</td>
<td>0</td>
</tr>
<tr>
<td>3600</td>
<td>3800</td>
<td>0</td>
</tr>
<tr>
<td>3800</td>
<td>4000</td>
<td>0</td>
</tr>
</tbody>
</table>
Sample 3: Collecting process profiling data

The following command collects the process profiling data for 100 seconds and dumping the results to a text file:
When run on a Carrizo APU, the content of the result text file is as below:

**CODEXL POWER PROFILE REPORT**

**PROFILE DETAILS**

- **CPU Details:** Family(0x15) Model(0x60)
- **CPU Core Mask:** 0xf
- **Sampling Interval:** 10 milli-seconds
- **Profile Start Time:** Mar-30-2016_00-20-44
- **Profile Duration:** 100 seconds

**PROCESS PROFILING DATA**

<table>
<thead>
<tr>
<th>SNo</th>
<th>PID</th>
<th>Samples</th>
<th>IPC</th>
<th>Power(Joules)</th>
<th>Power(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4956</td>
<td>11</td>
<td>3.34</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>CodeXLPowerProfiler.exe</td>
<td></td>
<td></td>
<td>\Program Files (x86)\CodeXL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>383503</td>
<td>31755</td>
<td>109.58</td>
<td>95.28</td>
</tr>
<tr>
<td></td>
<td>[System Process]</td>
<td></td>
<td></td>
<td>Unable to read path</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>648</td>
<td>1</td>
<td>0.38</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>lsass.exe</td>
<td></td>
<td></td>
<td>C:\Windows\System32</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>900</td>
<td>54</td>
<td>13.72</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>dwm.exe</td>
<td></td>
<td></td>
<td>C:\Windows\System32</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>224</td>
<td>96.76</td>
<td>0.24</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>System</td>
<td></td>
<td></td>
<td>Unable to read path</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2140</td>
<td>294</td>
<td>232.54</td>
<td>0.82</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>explorer.exe</td>
<td></td>
<td></td>
<td>C:\Windows</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>760</td>
<td>4</td>
<td>1.04</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>svchost.exe</td>
<td></td>
<td></td>
<td>C:\Windows\System32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PID</td>
<td>Ppid</td>
<td>CPU (ms)</td>
<td>Memory (MB)</td>
<td>App</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>7</td>
<td>1648</td>
<td>16</td>
<td>5.71</td>
<td>0.02</td>
<td>MsMpEng.exe</td>
</tr>
<tr>
<td>8</td>
<td>548</td>
<td>15</td>
<td>2.21</td>
<td>0.01</td>
<td>csrss.exe</td>
</tr>
<tr>
<td>9</td>
<td>4396</td>
<td>1066</td>
<td>710.18</td>
<td>3.86</td>
<td>iexplore.exe</td>
</tr>
<tr>
<td>10</td>
<td>680</td>
<td>178</td>
<td>170.15</td>
<td>0.29</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>11</td>
<td>5296</td>
<td>1</td>
<td>0.44</td>
<td>0.00</td>
<td>conhost.exe</td>
</tr>
<tr>
<td>12</td>
<td>2500</td>
<td>2</td>
<td>0.29</td>
<td>0.00</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>13</td>
<td>1320</td>
<td>6</td>
<td>2.57</td>
<td>0.01</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>14</td>
<td>1160</td>
<td>2</td>
<td>0.80</td>
<td>0.00</td>
<td>SearchIndexer.exe</td>
</tr>
<tr>
<td>15</td>
<td>456</td>
<td>35</td>
<td>15.70</td>
<td>0.04</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>16</td>
<td>1108</td>
<td>11</td>
<td>7.65</td>
<td>0.02</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>17</td>
<td>640</td>
<td>3</td>
<td>0.84</td>
<td>0.00</td>
<td>services.exe</td>
</tr>
<tr>
<td>18</td>
<td>936</td>
<td>2</td>
<td>1.00</td>
<td>0.00</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>19</td>
<td>1448</td>
<td>3</td>
<td>1.22</td>
<td>0.00</td>
<td>taskhostex.exe</td>
</tr>
<tr>
<td>20</td>
<td>720</td>
<td>2</td>
<td>0.46</td>
<td>0.00</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>21</td>
<td>1612</td>
<td>2</td>
<td>0.63</td>
<td>0.00</td>
<td>WUDFHost.exe</td>
</tr>
<tr>
<td>22</td>
<td>1296</td>
<td>4</td>
<td>1.42</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>
spoolsv.exe
23 2736 1 0.25 0.00
sppsvc.exe
Unable to read path

Profile Session Power Consumption: 115.00
Total PID record collected 385440

© 2015 Advanced Micro Devices. All rights reserved.
After switching to Profile mode, the Power Profiler Project Settings are accessible by any of these methods:

- In the Standalone application
  - Pressing CTRL + P
  - From the Profile menu, selecting “Profile Settings…”
- In Visual Studio
  - From the CodeXL menu, selecting “CodeXL Project Settings…”

The CodeXL Project Settings dialog appears. Navigate to the Power Profile sub-node under the Profile node in the options tree on the left.
The “Counters sampling interval (ms)” field sets the amount of time between every two consecutive sample operations of the profiler. The units used in this field are milliseconds. Setting an interval smaller than 100 milliseconds is discouraged as the overhead of the frequent sampling may skew the results and the design of the counters mechanism is not optimal for higher sample rates.
Power Profiler Counters Selection Dialog

The counter selection dialog allows you to choose which performance counters will be monitored in the next power profiling session. The available counters are divided into 4 categories: Power, Frequency, Temperature, and CPU Core State. At the right side of the dialog window you will find the list of Active Counters. These are the counters which will be monitored in the next power profiling session. To get the description of a specific counter, click on it in the Available Counters pane at the left side of the dialog window.
Note: The Total APU Power counter is enabled by default, and cannot be deactivated. Except for the Total APU Power counter, the activation of all other counters is optional.
Navigating and displaying power profiling sessions is performed using the CodeXL Explorer tree, same as for other types of CodeXL profiling sessions. Each power profiling session is represented by a node in the tree, containing two sub-nodes for the Timeline and Summary views. Double clicking a node opens its respective view in the MDI space.

A session can be imported into the CodeXL project by right-clicking on the Power Profiling node and selecting ‘Import Session’. The dialog that opens allows you to navigate to the location of the session to import, and displays a filtered list of files that correspond to the *.clxdb naming format.
By clicking the “New Power Session...” node a new session is created, but sampling of data does not yet begin. The Timeline and Summary view tab will appear, and you can select which counters will be collected during the profile session by either

- Double-clicking the last line in the timeline charts legend table “Double-click to add or remove counters...”
- Selecting the “Select Power Profiling Counters...” command from the menu. This command is located in the standalone CodeXL app’s Profile menu, and in the Visual Studio CodeXL menu.

To begin the sampling of data in the new session, click the “Start Profiling” toolbar button, or select “Start Profiling” from the Profile menu (standalone CodeXL) or CodeXL menu in Visual Studio.

Right-click on a session node in the explorer tree will display the session context menu which is identical to the context menu for other profile session types.
Power Profiler Timeline View

The Power Profiler Timeline View displays the measured values of the activated counters throughout the session. The horizontal axis of all charts represents the time that has passed since the beginning of the session. During profiling sessions, all of the charts in the Timeline View are being updated in real-time with the measured values which are streaming in. The uppermost ribbon (titled “Total APU Power”) displays the overall power consumption of the APU throughout the session.

The top chart has an adjustable range slider that controls the display of all the other timeline charts. By performing such actions as dragging the slider sideways, extending or retracting it, you set the scope of attention and the focus of the timeline charts. Each of the charts below displays only the data that was collected in the time range corresponding to the slider’s position and length. That is, the data in all timeline charts, except for the Total APU Power chart itself, is dictated by the time range which is selected by the Total APU Power chart’s range slider.

Below the Total APU Power ribbon, you will find additional ribbons containing more graphs, according to the set of activated counters: A Power chart which displays the power consumed by specific APU components (such as CPU cores or integrated GPU), a Frequency chart which displays the frequency of the selected components, a Temperature chart which displays the thermal trend of the selected components, and a CPU State chart which displays the CPU core states. The APU Power graph is always displayed, since the Total APU Power counter is activated by default and cannot be
deactivated. The other charts (frequency, temperature and CPU core state) are optional, and will only be displayed if the relevant counters were activated.

To the right side of each chart you will find a legend that displays the measured values at a specific point in time. To change the point in time for which the values are displayed, reposition the mouse cursor horizontally on one of the graphs. The list of counters in the legends is customizable, and specific counters can be removed/added between profile sessions.
Power Profiler Summary View

The Power Profiler Summary View displays an analysis of the values measured throughout the session. Similarly to the Timeline View, this view is updated in real-time when power profiling sessions are running. At the upper-left side of the summary view, you can see the session duration which is the amount of time that the profiling session was in progress. The uppermost chart (titled “Power”) shows the Total APU Power/Energy consumption. This chart has two modes: Cumulative and Average. The Cumulative mode displays the cumulative energy consumed by the APU components, measured in Joules. The Average mode displays the average power consumption of the APU components, measured in Watts. For each of these two modes, the Total APU Power Consumption value at the top of the summary view represents the value consumed by the whole APU.

If CPU or GPU frequency counters were activated for the session, you will find additional histogram graphs below the Power graph in the Summary View:

**CPU Frequency Graph**

This stacked histogram displays for each CPU core how much time it spent at each frequencies range.

**GPU Frequency Graph**

This histogram displays how much time the GPU spent at each frequencies range.
At the bottom of the Summary View you can find useful information about the current session:

- **Target Path**: The target application’s path. In system-wide sessions, this field will be left empty.
- **Working directory**: the target application’s working directory. In system-wide sessions, this field will be left empty.
- **Profile Session Directory**: the directory where CodeXL stored the session file.
- **Command Line Arguments**: the target application’s command line arguments (if any). In system-wide sessions, this field will be left empty.
- **Environment Variables**: the target application’s environment variables (if any). In system-wide sessions, this field will be left empty.
- **Profile Scope**: the scope of the current session (either system-wide, or system-wide with focus on an application).
- **Profile Session Type**: the type of this profiling session. This field will always be filled with the “Power” value for power profiling sessions.
- **Profile Start Time**: the time when the current session began.
- **Profile End Time**: the time when the current session ended.
Power Profiler Session Control

The control of power profiling sessions is very simple:

- To start a power profiling session, open the Counters Selection Dialog (in the CodeXL menu, click on Profile->Select Power Profiling Counters…) and select the counters which will be monitored during the session. Click OK, then click the ‘Start’ button on CodeXL’s session control toolbar.

- To stop a power profiling session, click the ‘Stop’ button on CodeXL’s session control toolbar.

© 2015 Advanced Micro Devices. All rights reserved.
Remote Power Profiling

For a detailed description of how to run a power profiling session on a remote machine, consult the Remote GPU Profiling, Power Profiling and Debugging section.
AMDTPowerProfileAPI Library

The CodeXL Power Profile API library is useful to analyze the energy efficiency of systems based on AMD CPUs, APUs and dGPUs (Discrete GPU). These APIs provide interface to read the power, thermal and frequency characteristics of APU/dGPU and their subcomponents.

These APIs are targeted for software developers who want to write their own application to sample the power counters based on their specific use case.

For detail information of these APIs, refer AMDTPowerProfileAPI.chm (on Windows platforms) or AMDTPowerProfileAPI.pdf (on Linux based platforms) file in the SDK/AMDTPowerProfile/doc subdirectory under the CodeXL installation directory.

AMDTPowerProfileAPI shared library has dependencies on AMDTBaseTools and AMDTOSWrappers shared libraries, so the corresponding .DLL (on Windows system) and .SO (on Linux system) should be added.

To build and execute a test application (test.cpp) following steps should be performed on Linux machine.

1. Assuming test.cpp is located at in /home/<user-dir>/samples
   $ cd /home/<user-dir>/samples
2. Set LD_LIBRARY_PATH
   $ export LD_LIBRARY_PATH=<codexl-install-dir>
3. Compile application code
$ g++ test.cpp -I<codexl-install-dir>/SDK/AMDTPowerProfile/inc -L<codexl-install-dir>/SDK/AMDTPowerProfile/bin/x86_64 -lAMDTPowerProfileAPI -L<codexl-install-dir> -lAMDTOSWrappers -lAMDTBaseTools -o test

4. Execute
   $ ./test
Limitations

- Multiple instance of CodeXL Power Profiler cannot be run simultaneously. If the CodeXL graphic client is conducting a power profile session then no other instance of the graphic client can perform a power profile and the command line tool cannot be run either, until the original profile session ends.
- ICELAND discrete GPU (Topaz-XT, Topaz PRO, Topaz XTL, Topaz LE) series is not support in 2.0 release.
- Please make sure you have latest catalyst driver installed before running power profiler. Newer version of discrete GPU may go to sleep (low power) state frequently if there is no activity in that GPU. In that case, power profiler may emits a warning AMDT_WARN_SMU_DISABLED. Counters may not be accessible during this state. It is advisable to bring discrete GPU to active state by running some openCL or openGL application, then run power profiling on that GPU.
- Process profiling is supported only with command line tool. If PMC (Performance Monitoring Counters) counters are not accessible and unable to calculate the IPC load, then compute unit power is distribute equally to each core. In that case power distribution to each process may not be accurate.
Remote GPU Profiling, Power Profiling and GPU Debugging

CodeXL provides remote profiling and debugging capabilities. Using these features, you can execute GPU profiling, power profiling and debugging sessions of applications that run on a remote machine. This is useful for working with tablets and headless server units.

- Running CodeXL Remote Agent
- The Agent’s Configuration File
- Performing Remote GPU Profiling
- Performing Remote Power Profiling
- Performing Remote GPU Debugging
As a first step, be sure that the remote machine has CodeXL Remote Agent installed. CodeXL Remote Agent ships with the CodeXL installer, and it is installed by default when installing CodeXL. You can also choose to install only CodeXL Remote Agent when using the installer.

CodeXL Remote Agent runs on the remote machine, and allows CodeXL clients located on other machines to connect the remote machine and execute GPU profiling and debugging sessions of applications on that machine.

When the remote is launched, it will output to the console a message in the following format:

```
CodeXL Remote Agent settings were extracted successfully
Agent version is: 1.4.4880.0
Read timeout is: Infinite Timeout
Write timeout is: Infinite Timeout
Agent IP:Port is: 10.20.0.144:27015
CodeXL Remote Agent is listening on <10.20.0.144:27015>
Press any key to terminate CodeXL Remote Agent. . .
```

In case the remote agent failed to launch successfully, it will output to the console a message describing the problem.

Please notice that by default the remote agent binds itself to the first valid IP address that it finds. In most cases, this
would be your desired behavior. However, if the remote machine has multiple IP addresses, and you would like to force CodeXL remote agent to bind itself to a particular address, you can use the --ip command line switch (--ip <ip_address>). For example, to force the remote agent to bind itself to 10.20.0.155, use the following command:

On Windows: CodeXLRemoteAgent --ip 10.20.0.155
On Linux: CodeXLRemoteAgent-bin --ip 10.20.0.155
The Agent’s Configuration File

CodeXL Remote Agent ships with a simple configuration file named CodeXLRemoteAgent.xml, which is located at the same folder as the CodeXLRemoteAgent executable. This configuration file defines several parameters which are being used by the agent. These parameters are being read by the agent just before it starts running. Therefore, if you change one of the values in the configuration file, you must rerun the agent for it to read the new values.

Here is a screenshot of the agent’s configuration file:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Settings for CodeXL Remote Agent -->
<Settings>
  <!-- Valid port number -->
  <PortNumber>27015</PortNumber>

  <!-- -1 for maximum possible timeout, or any positive integer -->
  <ReadTimeoutMs>-1</ReadTimeoutMs>
  <WriteTimeoutMs>-1</WriteTimeoutMs>

  <Version>public</Version>
</Settings>
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortNumber</td>
<td>The port number on which the remote agent listens to incoming connections.</td>
</tr>
<tr>
<td>ReadTimeoutMs</td>
<td>Read timeout to be used by the remote agent for incoming connections.</td>
</tr>
<tr>
<td>WriteTimeoutMs</td>
<td>Write timeout to be used by the remote agent for incoming connections.</td>
</tr>
</tbody>
</table>
Please pay special attention to the `<PortNumber>` parameter. This is the port on which the remote agent will listen to incoming connections. Prior to starting CodeXL remote Agent, you should verify that:

1. CodeXL Remote Agent is not blocked by the firewall on the remote machine.
2. The port number (27015 in the example above) is available (not being used by any other process), and that it is not blocked by a firewall. If there is a problem with the port, you can change this value and rerun the agent.

That’s it for the remote machine. Now, go back to the local machine where the CodeXL application is installed. The following sections describe how to perform remote GPU Profiling and Debugging.
Performing Remote GPU Profiling

1. Setting the remote target:

When you open the CodeXL project settings, look at the general tab:
<table>
<thead>
<tr>
<th>UI Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>These two radio buttons determine whether</td>
<td></td>
</tr>
<tr>
<td><strong>Local Host and Remote Host radio buttons</strong></td>
<td>we are in a remote or local session. If Remote Host is selected, all locations under the Target Application box will refer to paths on the remote machine. Otherwise, all locations will refer to paths on the local machine, as usual.</td>
</tr>
<tr>
<td><strong>Remote Host text box</strong></td>
<td>The IP address of the remote machine (the machine on which the remote agent is running). In the above screenshot, the remote host is 10.20.0.144. Alternatively, the host name of the remote machine can be used instead of its IP address, provided the name is recognized by the network DNS.</td>
</tr>
<tr>
<td><strong>Port text box</strong></td>
<td>The port (in the remote machine) on which the remote agent is listening. Note that the value of Port should match the PortNumber element value in CodeXLRemoteAgentConfig.xml which is located on the remote machine.</td>
</tr>
<tr>
<td><strong>Test Connection Button</strong></td>
<td>When this button is pressed, CodeXL will try to connect to the remote agent running on IP:Port (according to the text in the Remote Host Address and Port text boxes). This button can be used to verify that the remote agent is running on the remote machine and that it is reachable by the CodeXL client. Note that if the remote agent is not running on the remote machine or if a firewall blocks either CodeXL client or the remote agent - the connection test would fail.</td>
</tr>
</tbody>
</table>

2. For remote GPU performance counters profiling, have a look at the GPU Profile: Performance Counters tab (application trace settings act the same as in local sessions):
In a remote session, CodeXL does not know which counters are supported by the remote target’s GPU.
Therefore, all possible HW families are available. In case that you picked a counter that is not supported by the remote host, you will see a blank column in the result spreadsheet (can be easily eliminated by using the “Hide zero columns” feature).

3. To launch Remote GPU profiling, click OK. Then, start profiling as usual.
Performing Remote Power Profiling

1. Set the remote target as for remote GPU profiling (you can find a description of how to set the remote target in section 1 under Performing Remote GPU Profiling above). Make sure that the remote agent is running and is reachable via network by CodeXL client. Also, make sure that the remote machine supports power profiling. See the CodeXL System Requirements section for a list of AMD devices that support power profiling.

2. Open the Counter Selection Dialog and choose the desired set of counters, just like you do for local sessions. Note that a valid connection to the remote agent is required in order for CodeXL to populate the Counter Selection Dialog with the counters that are relevant to the specific remote machine.

3. Start the power profiling session by clicking the Start button, as you do for local sessions.
Performing Remote GPU Debugging

1. Setting the remote debugging ports:
   
   Go to Tools->Options, open the General tab and look at the Connection section. In the Remote Debugging Ports subsection, there are 4 ports which are being used by CodeXL on the local machine for remote debugging purposes. Please verify that the following requirements are fulfilled:

   a. All 4 port values are distinct.
   b. None of the 4 ports is blocked by a firewall.
   c. None of the 4 ports is being used by another process.

   If necessary, you can change the port values to fulfill the above requirements.

2. Setting the remote target:

   When you open the CodeXL project settings, look at the general tab:
<table>
<thead>
<tr>
<th>UI Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>Debug</td>
<td></td>
</tr>
<tr>
<td>Profile</td>
<td></td>
</tr>
<tr>
<td>&gt; CPU Profile</td>
<td></td>
</tr>
<tr>
<td>&gt; Application Timeline Trace</td>
<td></td>
</tr>
<tr>
<td>GPU Profile Perf. Counters</td>
<td></td>
</tr>
<tr>
<td>Power Profile</td>
<td></td>
</tr>
<tr>
<td>Frame Analysis</td>
<td></td>
</tr>
<tr>
<td>Analyze</td>
<td></td>
</tr>
<tr>
<td>&gt; Kernel/Shader Build Options</td>
<td></td>
</tr>
<tr>
<td>&gt; HLSL Build Options</td>
<td></td>
</tr>
<tr>
<td>Target Application</td>
<td></td>
</tr>
<tr>
<td>Application Type:</td>
<td>Desktop Application  Windows Store App</td>
</tr>
<tr>
<td>Executable Path:</td>
<td>C:/Users/Public/Documents/Code'Neill/examples/Teapot/Release/AMDITeapot.exe</td>
</tr>
<tr>
<td>Command Line Arguments:</td>
<td></td>
</tr>
<tr>
<td>Environment Variables:</td>
<td></td>
</tr>
<tr>
<td>Source Files</td>
<td></td>
</tr>
<tr>
<td>Source Files Directories:</td>
<td></td>
</tr>
<tr>
<td>Source Files Resolution Root Directory:</td>
<td></td>
</tr>
</tbody>
</table>
### Local Host and Remote Host radio buttons

These two radio buttons determine whether we are in a remote or local session. If Remote Host is selected, all locations under the Target Application box will refer to paths on the remote machine. Otherwise, all locations will refer to paths on the local machine, as usual.

### Remote Host text box

The IP address of the remote machine (the machine on which the remote agent is running). In the above screenshot, the remote host is 10.20.0.144. Alternatively, the host name of the remote machine can be used instead of its IP address, provided the name is recognized by the network DNS.

### Port text box

The port (in the remote machine) on which the remote agent is listening. Note that the value of Port should match the PortNumber element value in CodeXLRemoteAgentConfig.xml which is located on the remote machine.

3. Click OK, and start debugging as usual.
A compute or graphic memory allocated object is an OpenCL or OpenGL object created by a computer application, which is allocated on the memory of the OpenCL or OpenGL device. An allocated object can reside in the computer's main memory (as part of the driver's used virtual memory), in the device's internal memory, or in both. In multi-device OpenCL contexts and multi-renderer OpenGL contexts, an object's data can even reside in multiple devices at the same time. CodeXL currently supports the following allocated object types.

- OpenCL compute contexts
- OpenCL command queues
- OpenCL command events*
- OpenCL Image objects
- OpenCL buffer objects
- OpenCL computation programs
- OpenCL kernel objects*
- OpenCL sampler objects*
- OpenGL render contexts
- OpenGL render contexts' static buffers
- OpenGL texture objects
- OpenGL render buffers
- OpenGL framebuffer objects (FBOs)*
- OpenGL vertex buffer objects (VBOs)
- OpenGL shading programs*
- OpenGL shaders
- OpenGL display lists
- OpenGL pixel buffer objects (PBuffers)*

The memory consumption of these objects is insignificant, but their creation and deletion is still monitored by CodeXL.
**Function Types:** OpenCL functions can be divided into groups by their effect on the OpenCL implementation or by their outputs. Note that an OpenCL function can belong to more than one of these groups, or to none at all:

- Get Functions retrieve information from the OpenCL ICD.
- Buffer and Image Functions are functions related to buffer and image objects.
- Program and Kernel Functions are related to computation programs and kernels.
- Queue Functions are related to command queue objects.
- Synchronization Functions perform synchronization operations.

OpenGL functions can be grouped by their effect on the OpenGL implementation or by their outputs. Note that an OpenGL function can belong to more than one of these groups, or to none at all:

- Get Functions retrieve information from the OpenGL driver.
- State Change Functions change the values of OpenGL State Variables.
- Draw Functions can have a visible effect on the draw buffer.
- Raster Functions copy pixels from or to frame buffers.
- Programs and Shaders Functions are related to shaders and shading programs.
- Texture Functions create, delete, or manipulate textures.
- Matrix Functions are related to the matrix stacks (modelview matrix, projection matrix, color matrix, etc.).
- Name Functions are related to the OpenGL selection mode.
- Query Functions are related to OpenGL query items.
- Buffer Functions are related to OpenGL buffer objects.

To find out what groups a given function belongs to, look for its entry in the Function Calls Statistics tab (in the CodeXL Statistics view). The information is displayed under the "Function Type" column. The Statistics view's Function Types
Statistics tab shows details of your API usage by these groups, as well as more detailed information about the use of each function type.
Frames and Frame Terminators

**Computation Frame:** A computation frame is a set of OpenCL API calls, typically the largest set of calls an OpenCL compute context performs that can be considered a single logical operation. A computation frame is comparable to a render frame in a graphics engine. It is recommended for applications rendering graphics to define the computation frame as the set of commands used to calculate the data for a single render frame. Having a notion and boundary of what comprises a frame permits measurements such as frame times and frame rates, as well as API call statistics, which are useful in debugging and profiling.

**Frame Terminator:** Frame terminators are the functions that end your application computation and render frames. CodeXL uses frame terminators for frame-per-second calculations, statistics analysis, and other measurements, as well as for determining when to stop the execution after you press the Step Out button (Shift+F11). When choosing frame terminators for your project, ensure that at least one of the functions you choose is called for each frame you render. If you are using the OpenGL debug engine, you must select at least one render (OpenGL) frame terminator. A typical selection of Frame Terminators is clFinish for OpenCL and the SwapBuffers option for OpenGL.

© 2015 Advanced Micro Devices. All rights reserved.
**Vertex Batch**

**Vertex Batch:** A vertex batch is a group of vertices sent to the graphic driver in a single API call. In OpenGL, this effectively means the number of vertices sent to the graphic pipeline with a single function call. For example, the function `glVertex3f` always generates a 1-vertex batch, while `glDrawElements` creates a batch of the size determined by its count parameter value. The number of vertex batches sent to the driver represents the **cost** of drawing, while the batch size is the **benefit** of each function call. Drawing 200 1-vertex batches has the same benefit as drawing a single 200-vertex batch, for 200 times the cost. For this reason, it is best to draw as many vertices using as few batches as possible.
Bundled Sample Applications

CodeXL contains several bundled sample applications to demonstrate CodeXL usage scenarios and features.

The bundled sample applications are:

- Teapot
- Matrix Multiply
- D3DMultiThreading
The Teapot sample application draws a spinning teapot with steam coming out of its spout. The steam vapors movement is simulated using OpenCL. The rendering of the teapot and steam is implemented using OpenGL. In the Linux version of Teapot, the window, menu bar and UI widgets are implemented using the FLTK library.

The Teapot application can be used to demonstrate the features of CodeXL’s GPU Debugger, GPU Profiler and Static Analyzer.
Building the Teapot sample application

CodeXL includes a pre-built binary of the Teapot application. Follow the steps below if you wish to build it yourself.

On Windows

Follow these steps to build AMDTTeapot on Windows from the default install location:

· Launch Visual Studio with administrator privileges.
· From the CodeXL menu, select the ‘Open Teapot Sample Project’ command.
· Wait for the project to open and press F7.

On Linux

Follow these steps to build AMDTTeapot on Linux:

· Install the FLTK library
  
  o The bundled binaries of AMD Teapot were built with 64-bit binaries of FLTK 1.1.10.
  
  o Download FLTK sources from http://www.fltk.org

  o Extract and build the fltk libraries from the source files, then install the created fltk libraries

· Modifying The AMDTTeaPot Makefile

  o There are two makefiles in the Teapot sample folder. One is the /examples/Teapot/AMDTTeaPot/Makefile which you’ll need to edit, and the other is the /examples/Teapot/AMDTTeaPotLib/Makefile that does not need to be changed.
- Open /examples/Teapot/AMDTTeaPot/Makefile in a text editor.
- Replace the `-L"Replace with path to your local FLTK lib folder"` with the path of your local FLTK libraries, for example `-L/usr/lib64` in case you placed the libraries in the system folder and you are running a 64bit system.
- Replace the `-I"Replace with path to your local FLTK headers folder"` with the path of your local FLTK include files. Please note that the FLTK headers are placed in a folder named “FL” and the path you provide should point to the parent of the “FL” folder. Example `-I/user/include` if you placed them in the system include and not `-I/user/include/FL`

- Building the Teapot
  - Teapot consists of a library and an application so first the library needs to be built.
  - cd to the /examples/Teapot/AMDTTeaPotLib folder
  - make all
  - if everything went well you’ll see libAMDTTeaPot.a in that folder
  - Now the main application can be built.
  - cd to /examples/Teapot/AMDTTeaPot
  - make all
  - If successful the output should be in /examples/Teapot/release. Look for a file named AMDTTeaPot-bin.

Note that CodeXL RPM and Debian packages install CodeXL and is bundled sample applications under the /opt folder, which requires elevated privileges to write to on some Linux distributions.
Matrix Multiply

The matrix multiplication sample application performs multiplication of two matrices using 3 different implementations. All implementations are using ANSI C code.

The sample takes a command line argument that defines which of the matrix multiplication implementations will be executed. See the following table for descriptions of the 3 implementations and the command line argument that invokes each one.

This sample demonstrates how the CPU Profiler can be used to detect bottlenecks, identify problematic memory access patterns, and verify improved performance.

To select which implementation of the matrix multiplication will be performed, open the CodeXL Project Settings dialog, or in Visual Studio open the VS Project Settings dialog.
<table>
<thead>
<tr>
<th>#</th>
<th>Multiplication Implementation</th>
<th>Command line argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Inefficient</td>
<td></td>
<td>Inefficient implementation that performs redundant loop iterations.</td>
</tr>
<tr>
<td>2.</td>
<td>Classic</td>
<td>-c</td>
<td>Classic textbook implementation that uses naïve nested loops. The loops perform a sub-optimal memory access pattern.</td>
</tr>
<tr>
<td>3.</td>
<td>Improved</td>
<td>-i</td>
<td>Improved implementation that uses nested loops with continuous memory access pattern.</td>
</tr>
</tbody>
</table>
Project Settings for launching the classic implementation of matrix multiplication using the –c command line argument

Use Time-Based Profiling to detect the redundant loop iterations in the inefficient implementation (launched by not
supplying any command line argument).

Double clicking the inefficient_multiply_matrices(void) opens the source code view which demonstrates why this function is not efficient:

Use Event-Based Profiling with the Assess Performance session type to diagnose the problems in the classic implementation (launched by supplying “-c” as the command line argument):
Large number of cache misses in the internal nested loop non-sequential access to arrays

Finally, executing the improved implementation (launched by supplying “-i” as the command line argument) shows the number of cache misses is significantly reduced:
Building the Matrix Multiply sample application
Building the Matrix Multiply sample application

CodeXL includes a pre-built binary of the Matrix Multiply application. Follow the steps below if you wish to build it yourself.

**On Windows**

Follow these steps to build Matrix Multiply on Windows from the default install location:

- Launch Visual Studio with administrator privileges.
- From the CodeXL menu, select the ‘Open Matrix Multiply Sample Project’ command.
- Wait for the project to open and press F7.

**On Linux**

Follow these steps to build Matrix Multiply on Linux:

- Navigate to `codexl_folder_path/examples/ClassicMatMul/src`
  - `codexl_folder_path` is the folder in which you installed or unzipped CodeXL
- Type ‘make’ and hit Enter.

Note that CodeXL RPM and Debian packages install CodeXL and is bundled sample applications under the /opt folder, which requires elevated privileges to write to on some Linux distributions.

© 2015 Advanced Micro Devices. All rights reserved.
D3D12Multithreading

The D3D12Multithreading is a sample from the Microsoft D3D SDK, which is bundled with CodeXL.

The sample demonstrates the use of multiple thread with Direct3D 12.

The D3D12Multithreading sample can be used to get to know the Frame Analysis feature in CodeXL.
Building the D3D12Multithreading sample application

CodeXL includes a pre-built binary of the D3D12Multithreading application. Follow the steps below if you wish to build it yourself.

The D3D12Multithreading sample is a windows only sample, and can only be built in Visual Studio 2015, with the Windows 10 SDK installed.

On Windows

Follow these steps to build D3D12Multithreading on Windows from the default install location:

· Make sure that you have a Visual Studio 2015 IDE installed.

· Make sure that the Windows 10 SDK is installed on your system.

· Launch Visual Studio 2015

· From the CodeXL menu, select the ‘Open D3D12Multithreading Sample Project’ command.

· Wait for the project to open and press F7.
Tutorials

GPU Debugger Tutorial
CPU Profiler Tutorial
GPU Profiler Tutorial
Static Kernel Analyzer Tutorial

© 2015 Advanced Micro Devices. All rights reserved.
GPU Debugger Tutorial

- Host Debugging
- API-Level Debugging
- Kernel Debugging
- Additional Information

© 2015 Advanced Micro Devices. All rights reserved.
Host Debugging

Host Debugging is currently only available in the following configurations:
1. Linux CodeXL Standalone application
2. Windows CodeXL Visual Studio extension [native (C/C++) 32-bit target apps only]
Other configurations support only API-Level debugging.

In the supported configurations, CodeXL allows Host debugging of any application, regardless of any APIs used.

The following features are supported:
1. Host source code breakpoints
2. Host stepping (step-in, step-over, step-out)
3. Host break
4. Host locals display
5. Host expression evaluation (watch window)

CodeXL provides information directly from the system's native debugger, supporting any code and debug information that is available as those debuggers.

In the CodeXL standalone application, the debug controls are located in the Execution toolbar, next to the "Start Debugging" and "Stop Debugging" buttons. Setting source code breakpoints is as simple as opening a source file (Ctrl+O) and clicking the margins (F9). The Locals and Watch views are shared between Host Debugging and Kernel Debugging for a seamless experience.

In the CodeXL Visual Studio extension, Visual Studio's built-in views are used, providing a debugging experience as close to Visual Studio's built-in debug engines as possible.
API-Level Debugging

CodeXL allows API-level debugging of any OpenCL and / or OpenGL based application, regardless of implementation version and underlying hardware.

API Debugging Basics

CodeXL API-level debugging is how CodeXL displays the CPU (client) side of your OpenCL or OpenGL application. This means CodeXL can suspend the debugged application at any OpenCL and OpenGL API function, display the call stack for these function calls, display API calls history and give many analysis options for debugging this part of the application.

To run the debugged application, press the Go button (F5).

Once the application is running, you can suspend it by adding an API breakpoint or by pressing the "break" button.

Once the application is suspended, you can use the API steps drop-down to choose between API Step (Ctrl + F10 - run to the next API function call), Draw Step (Ctrl + Shift + F10 - run to the next OpenGL draw call) and Frame Step (Ctrl + F11 - run to the next frame terminator API function call).

The CodeXL Explorer tree view is an invaluable view in CodeXL. When the debugged application is suspended (by a breakpoint, a step command, a break command, etc.), this view fills with the hierarchy of all OpenCL and OpenGL objects allocated by the debugged application. Each context or category can be expanded to show its contained objects, and the CodeXL Properties view updates with the details of the selected object. Double-clicking any of the objects usually shows the contents of the object in some form.
View Compute and Graphic Memory Consumption

All of the compute and graphic memory information collected by CodeXL is displayed in the Memory view. To open this view, right-click on the object in the object tree, and select the "Memory analysis" option of the object. This view lists all the allocated objects of a given type (for example, context 1's buffers), as well as information regarding their memory consumption depending on the kind of objects listed. The graph view shows a breakdown of all the graphic memory consumed by the items in the current category. To see the scenario that led to an object's creation, use the Creation Calls Stack view. If the debugged application has debug information, double-clicking on a line in the Creation Calls Stack view opens the
location in the source file.

**Memory View**

<table>
<thead>
<tr>
<th>Name</th>
<th>Memory Size</th>
<th>Memory Flags</th>
<th>Reference Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer 1</td>
<td>4,096 KB</td>
<td>CL_MEM_READ_ONLY</td>
<td>...</td>
</tr>
<tr>
<td>Buffer 2</td>
<td>18,384 KB</td>
<td>CL_MEM_READ_WRITE</td>
<td></td>
</tr>
<tr>
<td>Buffer 3</td>
<td>18,384 KB</td>
<td>CL_MEM_READ_WRITE</td>
<td></td>
</tr>
<tr>
<td>Buffer 4</td>
<td>4,096 KB</td>
<td>CL_MEM_WRITE_ONLY</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>40,960 KB</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Find Unrecommended Function Calls**

Open the Statistics view. In the "Function Calls Statistics" tab, you might see warning icons next to some of the function names. Click any of these functions to display an explanation of why it is unrecommended, and what alternative is available.

**Statistics View**
View Buffers, Images, and Textures Information and Data

Double-clicking the main branch for buffers, images, or textures in the CodeXL Explorer (such as "CL Context 1 Images" or "GL Context 3 VBOs") opens a thumbnail view for those objects, to give a quick overview. Clicking an object in this view or double-clicking it in the CodeXL Explorer opens an Object view for it in the main views area. This view has a Data view spreadsheet showing the raw object data where each cell
in the spreadsheet shows the value of each pixel, and an Image view that visualizes the object. Clicking on a pixel samples it and keep its information in the "Selected Pixel" entry in the Information Panel on the right section of the view. Using the Images and Buffers toolbar, you can disable some or all the channels and zoom in and out on the image. You also can invert the image's colors, convert it to grayscale and rotate it. The Image Information panel on the right gives you controls over multi-imaged or multi-layered objects, such as 3D images, array textures, or mipmapped textures. Use the various sliders to change between the object elements.

**Objects View**
OpenCL buffers and OpenGL VBOs do not have the context of pixels, coordinates, and channels; thus, they are not visualized by the image view. Since they contain raw byte data, they even have no type or format information built-in into them. Instead, the Information panel has a combo-box letting you choose from the available data types and formats for use in their respective
View OpenCL Program and OpenGL Shader Source Code

To view an OpenCL program or OpenGL shader's source code, double-click it in the CodeXL Explorer. If you are using *.cl files
for OpenCL program source, adding the directory as the kernel source files folder in the Debug Settings dialog lets CodeXL to identify the source. This means you can set breakpoints inside the program before it was created and built at runtime, and you can persist breakpoints between debug sessions. Source viewing, of course, only applies to objects compiled at run time, and does not work with objects created with binaries. Double-clicking a kernel highlights the kernel function inside the program source. Applications that load program or shader source from strings or generate it at run time, have the sources saved to temporary files. Use the "File > Save As..." option to keep these files for comparison or analysis.
View OpenGL Program Pipeline and Sampler Objects

To view an OpenGL Program Pipeline or Sampler object, click on the object’s node in the CodeXL Explorer. This will display the object’s current state in the CodeXL Properties view. The state of Program Pipelines may contain references to OpenGL Shaders. These references will be displayed in the CodeXL Properties view as links. To view an OpenGL Shader’s source
code, just click on the link. Here is a snapshot of the CodeXL Properties view after clicking on a Program Pipeline object.

Here is a snapshot of the CodeXL Properties view displaying the state of an OpenGL Sampler object. Note that the “Bound to the following Texture Units” field may contain more than one texture (in a comma separated list between the curly brackets).
Work with API Function Breakpoints

Any CodeXL-supported OpenCL or OpenGL function can be set as a breakpoint in the Breakpoints dialog (Alt+Shift+B). The API Functions list contains all the supported functions. The Active Breakpoints list contains all the currently selected breakpoints. Each selected breakpoint has a checkbox next to it, representing whether the breakpoint is currently enabled. Disabling breakpoints instead of removing them saves the time of finding the function each time you want to enable the breakpoint. To add a function to the breakpoints list, select it in the API functions list and press Add or double-click it. To remove a breakpoint, select it in the breakpoints list and press Remove or double-click it. Using breakpoints can help you see the exact API status before the function is called, also allowing a comparison of the status before and after the function call (by pressing the Step Over button (F10) after the breakpoint hit). You also can use breakpoints to find where in your code each function is called, by looking at the Call Stack view after the breakpoint is hit. If you have debug information for the stack item, you can double-click it to see the exact location in the MDI source view.

Breakpoint Dialog
View Statistical Information About API Usage

Open the CodeXL Statistics view. The best way to overview your application's API usage is to collect the statistics for a few complete frames. Let your application run for a few frames,
then suspend it by pressing Break All (F6). If you are using frame terminators, press the Step Out button (Shift+F11) to advance to the end of the current frame. Alternatively, set a breakpoint on a frame terminator, and Continue (F5) past it a few times to collect enough information. This gives you the clearest statistical information about your API usage. To change between the statistics for different OpenCL and OpenGL contexts, choose them in the CodeXL Explorer.

The Function Types Statistics tab shows your OpenCL and OpenGL usage by function types. The percentages here do not always add up to 100% because an API function can belong to many or none of the categories. In the Function Calls Statistics tab, you can see details of specific functions, including the function types each function belongs to. In the Vertex Batch Statistics tab, you can see details of your application's vertex batches by size, giving an overview of the vertex sizes and occurrence, as well as the total number of vertices drawn with each batch size.
### Statistics - GL Context 1

<table>
<thead>
<tr>
<th>Function Type</th>
<th>% of Calls</th>
<th># of Calls</th>
<th>% of Calls (Avg. per Frame)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback Functions</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Synchronization Functions</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Buffer Functions</td>
<td>9.22</td>
<td>1,582</td>
<td>9.38</td>
</tr>
<tr>
<td>Query Functions</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Name Functions</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Matrix Functions</td>
<td>24.55</td>
<td>4,211</td>
<td>25.00</td>
</tr>
<tr>
<td>Texture Functions</td>
<td>21.51</td>
<td>3,690</td>
<td>21.88</td>
</tr>
<tr>
<td>Program and Shader Functions</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Raster Functions</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Draw Functions</td>
<td>21.46</td>
<td>3,682</td>
<td>21.88</td>
</tr>
<tr>
<td>Deprecated Functions</td>
<td>55.21</td>
<td>9,471</td>
<td>56.25</td>
</tr>
<tr>
<td>State Change Functions</td>
<td>42.97</td>
<td>7,371</td>
<td>43.75</td>
</tr>
<tr>
<td>Get Functions</td>
<td>7.90</td>
<td>1,356</td>
<td>6.25</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>17,155</strong></td>
<td></td>
<td><strong>32</strong></td>
</tr>
</tbody>
</table>

**Program and Shader Functions**

Functions that create, delete or manipulate the parameters of shaders and shading programs are considered Program and Shader functions.
### Statistics - GL Context 1

<table>
<thead>
<tr>
<th>Function Name</th>
<th>%</th>
<th># of Call</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>glTexCoord2f</td>
<td>12.26</td>
<td>2,104</td>
<td>Texture</td>
</tr>
<tr>
<td>glVertex3f</td>
<td>12.26</td>
<td>2,104</td>
<td>Draw</td>
</tr>
<tr>
<td>glMatrixMode</td>
<td>9.20</td>
<td>1,579</td>
<td>State Change</td>
</tr>
<tr>
<td>wglGetProcAddress</td>
<td>7.88</td>
<td>1,351</td>
<td>Get</td>
</tr>
<tr>
<td>glLoadIdentity</td>
<td>6.14</td>
<td>1,053</td>
<td>State Change</td>
</tr>
<tr>
<td>glBindBuffer - GL_.</td>
<td>6.14</td>
<td>1,053</td>
<td>State Change</td>
</tr>
<tr>
<td>glBindTexture - GL._</td>
<td>3.08</td>
<td>529</td>
<td>Texture</td>
</tr>
<tr>
<td>glDisable - GL_.</td>
<td>3.08</td>
<td>528</td>
<td>State Change</td>
</tr>
<tr>
<td>glBegin - GL_QUADS</td>
<td>3.07</td>
<td>527</td>
<td>Draw</td>
</tr>
<tr>
<td>glDisable - GL_LIG_</td>
<td>3.07</td>
<td>527</td>
<td>State Change</td>
</tr>
<tr>
<td>glDisable - GL_TEX_</td>
<td>3.07</td>
<td>527</td>
<td>State Change</td>
</tr>
<tr>
<td>glEnable - GL_TEX_</td>
<td>3.07</td>
<td>527</td>
<td>State Change</td>
</tr>
<tr>
<td>glViewport</td>
<td>3.07</td>
<td>527</td>
<td>State Change</td>
</tr>
<tr>
<td>glBindBuffer - GL_.</td>
<td>3.07</td>
<td>527</td>
<td>State Change</td>
</tr>
<tr>
<td>wglSyncBuffers</td>
<td>3.07</td>
<td>526</td>
<td>Draw</td>
</tr>
</tbody>
</table>

#### Warning: glBegin - GL_QUADS

Immediate mode rendering (glBegin-gEnd blocks) is considered an inefficient way to push primitive data into the graphic system. This mode requires making a large amount of function calls for describing the primitives' data.

Instead, consider using VBOs (Vertex Buffer Objects) or Vertex Arrays, which enable pushing large data chunks at once.
### Statistics - GL Context 1

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Deprecation Reason</th>
<th># of Calls</th>
<th>% of Calls</th>
<th>Deprecated at Ver</th>
<th>Removed at Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>glTexCoord2f</td>
<td>Completely Deprecated Function</td>
<td>2,104</td>
<td>12.25</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glVertex3f</td>
<td>Completely Deprecated Function</td>
<td>2,104</td>
<td>12.25</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glMatrixMode</td>
<td>Completely Deprecated Function</td>
<td>1,579</td>
<td>9.20</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td><strong>glLoadIdentity</strong></td>
<td>Completely Deprecated Function</td>
<td>1,053</td>
<td>6.14</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glBegin</td>
<td>Completely Deprecated Function</td>
<td>525</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glEnd</td>
<td>Completely Deprecated Function</td>
<td>525</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glOrtho</td>
<td>Completely Deprecated Function</td>
<td>525</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glLoadIdentity</td>
<td>Completely Deprecated Function</td>
<td>525</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>glMatrixMode</td>
<td>Completely Deprecated Function</td>
<td>525</td>
<td>3.07</td>
<td>OpenGL 3.0</td>
<td>OpenGL 3.1</td>
</tr>
<tr>
<td>Total (10 items)</td>
<td></td>
<td>9,471</td>
<td>55.21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### glLoadIdentity

**Deprecated Feature**: Fixed Pipeline Vertex Processing

The function **glLoadIdentity** was marked as deprecated in OpenGL version OpenGL 3.0 and was removed from OpenGL at version OpenGL 3.1.

**Warning**: It is recommended to avoid using deprecated functions, as their compatibility with future OpenGL versions is not assured.
Kernel Debugging

On AMD GPU hardware, CodeXL allows debugging OpenCL kernels in real time on AMD GPU hardware, with only a single machine and single device, and with hardware results.

Enter Kernel Debugging

There are three ways to enter kernel debugging:

- After starting debugging and suspending the process (with Break / F6 or an API function breakpoint), use the Step command or an API function breakpoint to get to an API call of clEnqueueNDRangeKernel or clEnqueueTask. Then, press Step Into (F11) to start debugging the kernel.

- Add the kernel function name as a kernel function
breakpoint in the breakpoints dialog (Alt+Shift+B), by 
switching to the Kernel Functions tab or by manually 
typing in the kernel name in the Active Breakpoints list. 
When a kernel matching the function name starts 
executing, the debugged process stops at the kernel’s 
beginning.

· From the CodeXL Explorer view, open the OpenCL 
program, and set a breakpoint inside the code. The 
application stops inside the kernel when it hits the 
breakpoint.

Kernel Debugging Control

While in kernel debugging, press step over (F10) to step to the 
next line in the same scope, step in (F11) to enter functions 
called inside the kernel, or step out (Shift+F11) to leave the 
current function or kernel.

You can also set kernel source breakpoints by selecting a line 
in the kernel and pressing Add Breakpoint (F9), or clicking in 
the margin next to the line. Press Continue (F5) to run up to 
the breakpoint. You can also set a breakpoint inside a function 
to enter that function (as with stepping in).

Focus On a Specific Work-Item

CodeXL's Current Work-Item toolbar lets you select a single 
work-item to focus on in CodeXL. If the toolbar is not open, 
right-click in the toolbars area, and select it from the dropdown 
menu. The combo-boxes for X, Y, and Z coordinates are enabled 
and offer values depending on the currently debugged kernel's 
N-Dimensional global work size. Selecting a work-item has the 
following effects:

· The locals and Watch views display variable values for 
  this work-item.

· When stepping through kernel debugging, "if" or "else"
clauses not entered by the selected work-item are skipped.

- The work-item is selected in the MultiWatch views.

![MultiWatch View]

**Inspect Variable Values**

Use the Local and Watch views to inspect variable values. The Locals view displays all variables that are currently in-scope. Use the Watch view to dereference pointers (*p) or access array members (a[150]).

The values shown in all these views are the at the current work-item, as selected by the current work-item toolbar.

**Compare Variable Values Across Work-Items and Work-Groups**

Use CodeXL Multi-Watch views to compare the values of variables across various work-items and work-groups. To open a Multi-Watch view, select it from the Views toolbar or menu, or right-click a variable and select "OpenCL Multi-Watch...". The view's variable selection combo-box offers all variables that can be viewed (similar to the Locals view list). Select a variable to view its values. The main view displays a spreadsheet or graphical representation of the values along a sliding scale ranging from the lowest value to the highest. Move the pointers in the Active Range slider to quickly highlight values outside a given range.

In Data (spreadsheet) view, the variable values are placed into cells according to their work-item coordinates. Work-groups
are alternately colored white and gray, letting you easily see the edge of a work-group.

In Image view, variables inside the range are colored on a gradient scale from black (lowest) to white (highest). Values below the range are red; values above the range are purple; work-items that are out of scope are yellow.

For three-dimensional work sizes, use the Z coordinate slider to navigate between the various XY planes in the global work.

The "hovered" controls display the location, value, and color for the mouse cursor's location. Click the view to set the current values to the "selected" control for an easy comparison against a single work-item. This also centers the Data view on the selected item.
# Keyboard Shortcuts

<table>
<thead>
<tr>
<th>Views and Dialogs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>New Project wizard</td>
<td>Ctrl+N</td>
</tr>
<tr>
<td>Debug Settings dialog</td>
<td>Ctrl+O</td>
</tr>
<tr>
<td>Breakpoints dialog</td>
<td>Alt+Shift+B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Common Debugging</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Go (Start / Continue Debugging)</td>
<td>F5</td>
</tr>
<tr>
<td>Stop Debugging</td>
<td>Shift+F5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>API-Level Debugging</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Break at next API function call</td>
<td>F6</td>
</tr>
<tr>
<td>Step to next API function call</td>
<td>F10</td>
</tr>
<tr>
<td>Step into kernel debugging</td>
<td>F11</td>
</tr>
<tr>
<td>Step to next frame terminator</td>
<td>Ctrl+F11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel Debugging</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Over</td>
<td>F10</td>
</tr>
<tr>
<td>Step Into function</td>
<td>F11</td>
</tr>
<tr>
<td>Step Out of function / kernel</td>
<td>Shift+F11</td>
</tr>
<tr>
<td>Set / remove breakpoint at kernel source</td>
<td>F9</td>
</tr>
</tbody>
</table>
The CPU Profiler component in CodeXL is used for performance analysis and tuning of applications running on CPU. It is used to identify the various factors affecting the performance of the profiled application or the entire system.

This tutorial provides step-by-step instructions for using CodeXL to analyze the performance of an application program. It consists of the following modules.

**Preparing an Application for CPU Profiling**

Create a CodeXL Project for CPU Profiling

Set the global options for CPU Profiling

Set the CPU Profile Session Options

Analysis with Time-based Profiling (TBP)

Analysis with Event-based Profiling (EBP)

Analysis with Instruction Based Sampling (IBS)

Analysis with Cache Line Utilization (CLU)

Settings for C++ Inline Functions

Settings for CPU Profiler to run on Virtual Machine

How to use CPU Profiler Control APIs

© 2015 Advanced Micro Devices. All rights reserved.
Preparing an Application for CPU Profiling

The CodeXL CPU Profiler uses the debug information generated by the compiler to show the correct function names in various analysis views and to attribute the collected samples to source statements in Source View. Otherwise, the results of the CPU Profiler would be less descriptive, displaying only the assembly code, and the functions would be reported as NO SYMBOL.

Generate Debug Information on Windows

When compiling an application in release mode, you can produce the debug information to enable CodeXL to perform an analysis. When using Microsoft Visual C++ to compile the application in release mode, set the following options to ensure that the debug information is generated and saved in a program database file(with a .pdb extension).

To set the compiler option to generate the debug information for a Win32 application in release mode:

1. Right click on the project and select Properties menu item.
2. In the Configuration list, select Active(Release).
3. In the Platform list, select Active(Win32).
4. In the project pane, expand the Configuration Properties folder, then expand the C/C++ folder.
5. Select General.
6. In the work pane, select Debug Information Format, and from the drop-down list select Program Database (Zi) or Program Database for Edit & Continue (/ZI).
7. In the project pane, expand the Linker folder; then expand the **Debugging** folder.
8. In the **Generate Debug Info** list, select **Yes** (/DEBUG).
Generate Debug Information on Linux

The application must be compiled with the -g option to enable the compiler to generate debug information. Modify either the Makefile or the respective build scripts accordingly.
Create a CodeXL Project for CPU Profiling

To start profiling their application, you must create a new CodeXL Project and set the appropriate settings. The following steps explain how to create a new CodeXL Project.

1. Once CodeXL is launched, click on Create New Project link in the main pane, or select File > New Project...
The Create a new CodeXL Project dialog box is displayed.
2. Select the General tree node on the left pane.
3. In the Executable Path box, click Browse, and select the application to be profiled.
4. By default, CodeXL uses the name of the application as the project name. To modify the project name, edit the CodeXL Project Name box.
5. Edit the other boxes that are required for executing the application to be profiled.
   · Working Directory
   · Command Line Arguments (optional)
   · Environment Variables (optional)
   · Source Files Directories (optional)
6. Click **OK**.
   A new CodeXL project is created and displayed in the Session Explorer window.
Creating CPU Profiling Project for Windows Store (Metro) Apps

1. Once CodeXL is launched, click on Create New Project link in the main pane, or select File > New Project...
   The Create a new CodeXL Project dialog box is displayed as shown in the previous screenshot.
2. Select the General tree node on the left pane.
3. Select “Application Type” as “Windows Store App”. Please note that Store Apps are only available on Windows 8.x or later versions.
4. In the Executable Path box, click Browse. The Select Windows Store app dialog will be displayed. Select the Store App to be profiled. Select OK.

1. Configure other settings as described in the previous section item 4, 5.
2. Click OK.
A new CodeXL project for Windows Store App is created and displayed in the Session Explorer window.
Set the global options for CPU Profiling

Specify the global CPU profile settings by modifying the CodeXL Options dialog box. The following steps explain how to do this.

1. Select the profile mode by selecting **Profile > Profile Mode - CPU: Time-based Sampling** or by clicking the Profile icon 🔄.
2. Select **Tools > Options...** menu item.
   The CodeXL Options dialog box is displayed.
3. Select the **General** tab, as shown in the following screenshot.
4. Select **Alert when a source file is missing** check box if you want CodeXL to pop up an alert message dialog box if source files are missing.

5. Select the **CPU Profile** tab, as shown in the screenshot in the below sub-section ‘Setting Symbol Server Path’.
Setting Symbol Server Path

1. If any additional debug symbol paths are to be specified to find the required .pdb files, select the **Additional debug symbol paths** check box, browse, and select the folder in which PDB files are kept.
2. Select **Symbol Server Directories** to enable the support of Microsoft symbol server technology. Set the download directory and select Microsoft symbol server link. Additional symbol server paths can be added to this list. (See the help file for a detailed description of the available options.)
3. Select the block size for disassembly fetch while navigating through disassembly code. The default size of 1024 is usually sufficient.
Set the General Profile Session Options

1. Click the **Profile > Profile Settings** menu item.
2. Click the **Profile** tree node in the left pane.
3. Select one of the CPU profile types in the profile type combo box. The corresponding description will appear.
4. Select the profile scope:
   - **Single Application Profile**: when this option is selected, CodeXL will launch the profiled application (classic.exe in this case), and will collect profile data only from this process.
   - **System-Wide Profile**: when this option is selected, CodeXL will not launch the profiled application, and will collect profile data from all the currently running processes.
   - **System-Wide Profile with focus on application**: when this option is selected, CodeXL will launch the profiled application (classic.exe). It will collect profiled events data both from classic.exe and other running processes on the machine. For classic.exe, CodeXL will also collect call stack information.
Set the CPU Profile Session Options

The CPU Profile view is used to control the profile data collection options. The selected settings persist and apply to future profile data collection sessions that are started within a project; they persist until the profile settings are modified.

1. Click the **Profile > Profile Settings** menu item.
2. Click the **Profile** tree node in the left pane. The **CPU Profile** settings page is displayed (see the following screenshot).
3. Select **Collect call stack details** to enable the collection of call stack data to analyze caller-called relationships and call frequency.

   **Note:** The collection of call stack is not supported for Java or CLR based applications. For those applications, this checkbox would be unchecked and disabled.

   - Set the value for **Collect for code executed in** drop box, to apply call stacks collection only to code sampled in User space, Kernel space or both (User space and Kernel space).
   - Set the value for **Collect Call Stack every** spin box, to specify how often a CSS sample is taken. If the selected value is N for example, then a CSS sample is taken every N regular samples. By default it is set to 1.
   - Set the value for **Call Stack Collection Depth**. Use Maximal for high accuracy, and Minimal for lower performance overhead.

   **Note:** This option is set separately for Time-Based Sampling and other CPU Profiling session types.

   - Check the **Reproduce missing call stack info** check box, to perform additional analysis to overcome frame-
pointer omission (FPO) in 32-bit apps and lack of unwind info in 64-bit. The profiler will store additional data during the profile session and require more time during post-session processing.

**Note:** This option is set separately for Time-Based Sampling and other CPU Profiling session types.

4. In the **Data Collection Schedule** section:
   - Select **Throughout entire duration** if you want the data to be collected throughout the entire execution of the profile session.
   - Select **Start profile with data collection paused** if you want to start the application and then decide when data collection should be started.
   - Select **Scheduled:** if you want to set the data collection start after and end after values. Once this option is checked, set the values for the start / end timing of the data collection.

5. Select / Unselect cores in the cores tree under **Profile Hardware Scope**, when you want to limit the execution of the program to the selected cores in a multicore system.
Analysis with Time-based Profiling (TBP)

**Time-based profiling (TBP)** identifies the hot spots in a program that are consuming the most time. Hot spots are good candidates for further investigation and optimization.

**Collecting TBP profile Data**
**Changing the TBP profile collection options**
**Analyzing the TBP profile data**
Collecting TBP profile Data

To collect the TBP data, do the following.

1. **Create a new CodeXL Project.**
2. In the toolbar, select Time-based profile in the profile configuration list, as shown in the following screenshot.

3. Set the profile session options as explained in **Set the CPU Profile Session settings**.
4. Click the Start button in the toolbar or select Profile > Start Profiling, as shown below.

CodeXL begins data collection and launches the specified application program. The start icon is disabled until profiling is in progress. The console window is open, and the launched application is running in the console window. The status bar shows the profile duration while profiling is running; this is the percentage completion of data translation during data translation phase.
Matrix multiplication sample

-----------------------------

Initializing matrices
Multiplying matrices

Invoke inefficient implementation of matrix multiplication

© 2015 Advanced Micro Devices. All rights reserved.
Changing the TBP profile collection options

CodeXL lets you modify the default profile collection settings. To do this:

1. Set the profile session options as explained in Set the CPU Profile Session settings.
2. By default, the sampling interval for TBP is 1 millisecond. To modify the sampling interval, click Profile > Profile Settings and click on the CPU Profile > Custom node in the left pane tree in the Edit CodeXL Project Settings dialog box. The Custom CPU Profile Monitored Events page will be displayed.
3. Click the Interval field and set the required sampling interval.
4. Click the OK button to save the modified profile settings.
Custom CPU Profile Monitored Events

Add an Event to the 'Monitored Events' list to include it in the Custom Profile session.

Available events:
- Assess Performance
- Cache Line Utilization
- Instruction-based Sampling
- Investigate Branching
- Investigate Data Access
- Investigate Instruction Access
- Investigate L2 Cache Access

Time-based Sampling:
- [E100] Timer Event

Samples are taken at discrete time intervals.
Analyzing the TBP profile data

This page explains how to analyze and navigate to the various profile views provided by the CodeXL CPU Profiler. It consists of the following modules.

- **Overview page**
- **Modules view**
- **Functions view**
- **Call Graph view**
- **Source Code view**
Overview page

When data collection is complete, CodeXL processes the data and displays the results.
Initially the **Profile Overview** is displayed with the following profile data.

- **5 Hottest Functions** table
- **5 Hottest Modules** table
- **Execution** section
- **Profile Details** section
From the overview page you can navigate to various pages to analyze the profile data.
Modules view

1. To navigate to Modules view, do one of the following:
   · In CodeXL Explorer pane, double-click on Modules node in the profile session tree entry created for this profile run.
     The Modules view is displayed.
   · In Overview page, right-click on 5 Hottest Modules table entry, and select Display in Modules View.
     The Modules view is displayed.

2. Click on the Display: All Data, All Modules link to set the Display Settings.
   The Display Settings dialog box is displayed to let you set the various display settings.
3. Click **OK** button to save the Display filter settings.
Functions view

1. To navigate to the Functions view, do one of the following.
   - In CodeXL Explorer pane, double-click on the **Functions** node in the profile session tree entry created for this profile run. The **Functions** view is displayed with function entries from all the profiled processes and the load modules.
   - In **Overview** page, right-click on the **5 Hottest Modules** table entry, and select **Display in Functions View**. The **Functions** view is displayed with function entries from the selected load module.
   - In **Overview** page, right-click on the **5 Hottest Functions** table entry, and select **Display in Functions View**. The **Functions** view is displayed with function entries from all the profiled processes and the load modules.
   - In **Modules** view, select a Process from the **Processes** table, and select **Display in Functions View**. This displays the functions from the selected process and all its load modules.
   - In **Modules** view, select a Module from the **Modules filtered by selected processes** table, and select **Display in Functions View**. This displays the functions from the selected load module.

2. Click on **Display: All Data, All Modules** link to set the display settings.
The **Display Settings** dialog box is displayed to let you set the various display settings.

3. Click **OK** button to save the Display filter settings.

4. Click on **modules shown, modules hidden** link to set the module filter settings to be used in the Functions view. The **CodeXL Modules Filter** dialog box is displayed to let you set the module filter settings.
5. Click the **OK** button to save the Modules filter settings.
Call Graph view

1. To navigate to Call Graph view, do any of the following:
   · In CodeXL Explorer pane, double-click on Call Graph node in the profile session tree entry created for this profile run.
     The Call Graph view is displayed with Callstack samples collected for the profiled application.
   · In Overview page, right-click on the 5 Hottest Functions table entry, and select Display in Call Graph View.
     The Functions view is displayed with Callstack samples collected for the profiled application.
   · In Functions view, select a function from the Functions table, and select Display in Call Graph View.
     This displays the Call Graph constructed from the callstack samples collected from the profiled application.

2. Click on the Display: All Modules link to enable Display system modules.
   This displays the callstack samples collected from the system modules.
3. Click the OK button to save the Display filter settings.
4. Click on the Process combo box to select the process for which you want to view the Call Graph.
## Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Self Samples</th>
<th>Deep Samples</th>
<th>% of Deep Samples</th>
<th>No. of Paths</th>
<th>Source File</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>_scct_common_main_seh</td>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td>eee_common_int(210)</td>
<td>AMDTC</td>
</tr>
<tr>
<td>main</td>
<td>308</td>
<td></td>
<td>95.67%</td>
<td>3</td>
<td>amdtdclassmatmul.cpp(160)</td>
<td>AMDTC</td>
</tr>
<tr>
<td>classic_multiply_matrices(void)</td>
<td>502</td>
<td>502</td>
<td>94.54%</td>
<td>2</td>
<td>amdtdclassmatmul.cpp(10)</td>
<td>AMDTC</td>
</tr>
<tr>
<td>initialize_matrices(void)</td>
<td>6</td>
<td>22</td>
<td>4.14%</td>
<td>10</td>
<td>amdtdclassmatmul.cpp(63)</td>
<td>AMDTC</td>
</tr>
<tr>
<td>rand</td>
<td>2</td>
<td>15</td>
<td>2.22%</td>
<td>8</td>
<td>ucrtbase</td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>5</td>
<td>14</td>
<td>2.64%</td>
<td>7</td>
<td>ucrtbase</td>
<td></td>
</tr>
<tr>
<td>RtlSetCurrentTransaction</td>
<td>6</td>
<td></td>
<td>1.13%</td>
<td>5</td>
<td>ntdll.dll</td>
<td></td>
</tr>
<tr>
<td>BaseThreadInitThunk</td>
<td>6</td>
<td></td>
<td>1.13%</td>
<td>5</td>
<td>kernel32</td>
<td></td>
</tr>
<tr>
<td>LdrSetAppContextDirRestrictionCallback</td>
<td>3</td>
<td></td>
<td>0.98%</td>
<td>3</td>
<td>ntdll.dll</td>
<td></td>
</tr>
</tbody>
</table>

## Immediate Parents and Children of Function: classic_multiply_matrices(void)

<table>
<thead>
<tr>
<th>Parents</th>
<th>Samples</th>
<th>% of samples</th>
<th>Module</th>
<th>Self + children</th>
<th>Samples</th>
<th>% of samples</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>502</td>
<td>100.00%</td>
<td>AMDTCClassicMatMul.exe</td>
<td>[set]</td>
<td>502</td>
<td>100.00%</td>
<td>AMDTCClassicMatMul.exe</td>
</tr>
</tbody>
</table>

## Paths

- **Function:** _scct_common_main_seh
  - Self samples: 500
  - % of Downstream samples: 99.60%
  - Module: AMDTCClassicMatMul.exe

- **Function:** main
  - Self samples: 500
  - % of Downstream samples: 99.60%
  - Module: AMDTCClassicMatMul.exe

- **Function:** classic_multiply_matrices(void)
  - Self samples: 500
  - % of Downstream samples: 96.60%
  - Module: AMDTCClassicMatMul.exe

- **Function:** RtlSetCurrentTransaction
  - Self samples: 2
  - % of Downstream samples: 0.40%
  - Module: ntdll.dll

- **Function:** RtlSetCurrentTransaction
  - Self samples: 2
  - % of Downstream samples: 0.40%
  - Module: ntdll.dll

**Search through call paths to find potential bottlenecks:** Functions downstream from your top level code.
Source Code view

1. To navigate to Call Graph view, do any of the following:
   - In the **Overview** page, right-click on the **5 Hottest Functions** table entry, and select **Open Source Code**. This displays the **Source Code** view for the selected function.
   - In the **Functions** view, select a function from the **Functions** table, right-click and select **Open Source Code** from the context menu. This displays the **Source Code** view for the selected function.
   - In **Call Graph** view, select a function from the **Functions** table, right-click and select **Open Source Code** from the context menu. This displays the **Source Code** view for the selected function.

2. To view the sources for the some other function, click on the **Function** combo-box, and select the function for which you want to view the source code.
void classic_multiply_matrices()
{
  // Multiply the two matrices
  for (int i = 0; i < ROWS; i++)
  {
    for (int j = 0; j < COLUMNS; j++)
    {
      float sum = 0.0;
      for (int k = 0; k < COLUMNS; k++)
      {
        sum = sum + matrix_a[i][k] * matrix_b[k][j];
      }
      matrix_c[i][j] = sum;
    }
  }
}

void improved_multiply_matrices()
{
  // Multiply the two matrices
  for (int i = 0; i < ROWS; i++)
  {
    for (int j = 0; j < COLUMNS; j++)
    {
      for (int k = 0; k < COLUMNS; k++)
      {
        // Code continues...
      }
    }
  }
}
Analysis with Event-based Profiling (EBP)

Event-based profiling (EBP) uses the hardware performance event counters to measure the number of specific kinds of events that occur during execution. Examples of events include processor clock cycles, retired instructions, data cache accesses, and data cache misses. The specific events to be measured are determined by the profile configuration that is used to set up data collection. This section comprises the following modules.

Collecting EBP profile data
Changing the EBP profile collection options
Analyzing the EBP profile data
Collecting EBP profile data

To collect the EBP data:

1. **Create a new CodeXL Project**
2. In the toolbar, select **CPU: Assess Performance** from the profile configuration list. This is a predefined Event-based profile configuration provided by CodeXL.

3. Set the profile session options as explained in [Set the CPU Profile Session settings](#).
4. Click the Start icon in the toolbar, or select **Profile > Start Profiling**, as shown below.

CodeXL begins data collection and launches the specified application program. The start icon has been clicked and is disabled while profiling is in progress. The console window is open, and the launched application is running in the console window. The status bar displays the profile duration while profiling is running; this is a percentage of completion of data translation during data translation phase.
Changing the EBP profile collection options

CodeXL lets you modify the default profile collection settings. To do this:

1. Set the profile session options as explained in Set the CPU Profile Session settings.
2. By default, the sampling interval for TBP is 1 millisecond. To modify the sampling interval, click Profile > Profile Settings and click the CPU Profile > Custom node in the left pane tree in the Edit CodeXL Project Settings dialog box. The Custom CPU Profile Monitored Events page will be displayed.
3. Click the Interval field and set the required sampling interval.
4. Click the OK button to save the modified profile settings.
Assess Performance

Use this configuration to get an overall assessment of performance and to find potential issues for investigation.
Analyzing the EBP profile data

This page explains how to analyze and navigate to the various profile views provided by the CodeXL CPU Profiler. Its modules are:

- **Overview page**
- **Modules view**
- **Functions view**
- **Call Graph view**
- **Source Code view**
Overview page

When data collection is complete, CodeXL processes the data and displays the results. Initially, the **Profile Overview** is displayed with the following profile data:

- 5 **Hottest Functions** table
- 5 **Hottest Modules** table
- **Execution** section
- **Profile Details** section
From the overview page, you can navigate to various pages to analyze the profile data.
Modules view

1. To navigate to the Modules view, do one of the following.
   · In CodeXL Explorer pane, double-click on the Modules node in the profile session tree entry created for this profile run.
     The Modules view is displayed.
   · In Overview page, right-click on the 5 Hottest Modules table entry, and select Display in Modules View.
     The Modules view is displayed.

2. Click on the Display: All Data, All Modules link to set the Display Settings.
   The Display Settings dialog box is displayed to let you set the various display settings.
3. Click OK button to save the Display filter settings.
Functions view

1. To navigate to Functions view, do one of the following:
   · In CodeXL Explorer pane, double-click on **Functions** node in the profile session tree entry created for this profile run.
     The **Functions** view is displayed with function entries from all the profiled processes and the load modules.
   · In the **Overview** page, right-click on the **5 Hottest Modules** table entry, and select **Display in Functions View**.
     The **Functions** view is displayed with function entries from the selected load module.
   · In the **Overview** page, right-click on the **5 Hottest Functions** table entry, and select **Display in Functions View**.
     The **Functions** view is displayed with function entries from all the profiled processes and the load modules.
   · In the **Modules** view, select a Process from the **Processes** table, and select **Display in Functions View**.
     This displays the functions from the selected process and all its load modules.
   · In the **Modules** view, select a Module from the **Modules filtered by selected processes** table, and select **Display in Functions View**.
     This displays the functions from the selected load module.

2. Click on the **Display: All Data, All Modules** link to set the display settings.
   The **Display Settings** dialog box is displayed to let you set...
the various display settings.
3. Click **OK** button to save the Display filter settings.
4. Click on the **modules shown, modules hidden** link to set the module filter settings to be used in the Functions view.
   The **CodeXL Modules Filter** dialog box is displayed to let you set the module filter settings.
5. Click **OK** button to save the Modules filter settings.

© 2015 Advanced Micro Devices. All rights reserved.
Call Graph view

1. To navigate to Call Graph view, do one of the following:
   · In the CodeXL Explorer pane, double-click on Call Graph node in the profile session tree entry created for this profile run. The Call Graph view is displayed with Callstack samples collected for the profiled application.
   · In the Overview page, right-click on the 5 Hottest Functions table entry, and select Display in Call Graph View. The Functions view is displayed with Callstack samples collected for the profiled application.
   · In Functions view, select a function from the Functions table, and select Display in Call Graph View. This displays the Call Graph constructed from the callstack samples collected from the profiled application.

2. Click on the Display: All Modules link to enable Display system modules. This displays the callstack samples collected from the system modules.
3. Click the OK button to save the Display filter settings.
4. Click on the Process combo box to select the process for which you want to view the Call Graph.
Source Code view

1. To navigate to Call Graph view, do one of the following:
   - In the Overview page, right-click on the 5 Hottest Functions table entry, and select Open Source Code. This displays the Source Code view for the selected function.
   - In the Functions view, select a function from the Functions table, right-click and select Open Source Code from the context menu. This displays the Source Code view for the selected function.
   - In the Call Graph view, select a function from the Functions table, right-click and select Open Source Code from the context menu. This displays the Source Code view for the selected function.

2. To view the sources for the some other function, click on the Function combo-box, and select the function for which you want to view the source code.
Analysis with Instruction Based Sampling (IBS)

**Instruction Based Sampling (IBS)** uses the hardware feature called IBS, to collect the IBS Fetch and IBS Op data.

*Collecting IBS profile data*

*Changing the IBS profile collection options*

*Analyzing the IBS profile data*
Collecting IBS profile data

To collect the IBS data:

1. **Create a new CodeXL Project**.
2. In the toolbar, select **CPU: Instruction-based Sampling** from the profile configuration list, as shown in the following screenshot.
   This is a predefined IBS profile configuration provided by CodeXL.
3. Set the profile session options as explained in **Set the CPU Profile Session settings**.

4. Click the Start icon in the toolbar, or select **Profile > Start Profiling**, as shown below.

![Profile settings](image)

CodeXL begins data collection and launches the specified application program. The start icon is disabled until profiling is in progress. The console window is open, and the launched application is running in the console window. The status bar displays the profile duration while profiling is running; this is the percentage completion of data translation during data translation phase.
Changing the IBS profile collection options

CodeXL lets you modify the default profile collection settings. To do this:

1. Set the profile session options as explained in Set the CPU Profile Session Options.
2. By default, the sampling interval for TBP is 1 millisecond. To modify the sampling interval, click Profile > Profile Settings and click on the CPU Profile > Custom node in the left pane tree. In the Edit CodeXL Project Settings dialog box. The Custom CPU Profile Monitored Events page will be displayed.
3. Click the Interval field and set the required sampling interval.
4. Click the OK button to save the modified profile settings.
Analyzing the IBS profile data

This page explains how to analyze and navigate to the various profile views provided by the CodeXL CPU Profiler. It is comprised of the following modules.

- Overview page
- Modules view
- Functions view
- Call Graph view
- Source Code view
When data collection is complete, CodeXL processes the data and displays the results.
Initially, the **Profile Overview** displays the following profile data.

- **5 Hottest Functions** table
- **5 Hottest Modules** table
- **Execution** section
- **Profile Details** section
From the overview page you can navigate to various pages to analyze the profile data.
Modules view

1. To navigate to Modules view, do one of the following.
   • In the CodeXL Explorer pane, double-click on the Modules node in the profile session tree entry created for this profile run.
     The Modules view is displayed.
   • In the Overview page, right-click on the 5 Hottest Modules table entry, and select Display in Modules View.
     The Modules view is displayed.

2. Click on the Display: All Data, All Modules link to set the Display Settings.
    The Display Settings dialog box is displayed to let you set the various display settings.
3. Click the **OK** button to save the Display filter settings.
Functions view

1. To navigate to Functions view, do one of the following.
   - In CodeXL Explorer pane, double-click on Functions node in the profile session tree entry created for this profile run.
     The Functions view is displayed with function entries from all the profiled processes and the load modules.
   - In the Overview page, right-click on the 5 Hottest Modules table entry, and select Display in Functions View.
     The Functions view is displayed with function entries from the selected load module.
   - In the Overview page, right-click on the 5 Hottest Functions table entry, and select Display in Functions View.
     The Functions view is displayed with function entries from all the profiled processes and the load modules.
   - In the Modules view, select a Process from the Processes table, and select Display in Functions View.
     This displays the functions from the selected process and all its load modules.
   - In the Modules view, select a Module from the Modules filtered by selected processes table, and select Display in Functions View.
     This displays the functions from the selected load module.

2. Click on the Display: All Data, All Modules link to set the display settings.
   The Display Settings dialog box is displayed to let you set the various display settings.
3. Click the **OK** button to save the Display filter settings.
4. Click on the **modules shown, modules hidden** link to set the module filter settings to be used in the Functions view.
   The **CodeXL Modules Filter** dialog box is displayed to let you set the module filter settings.
5. Click the **OK** button to save the Modules filter settings.
Call Graph view

1. To navigate to Call Graph view, do one of the following.
   - In the CodeXL Explorer pane, double-click on the Call Graph node in the profile session tree entry created for this profile run.
     The Call Graph view is displayed with Callstack samples collected for the profiled application.
   - In the Overview page, right-click on the 5 Hottest Functions table entry, and select Display in Call Graph View.
     The Functions view is displayed with Callstack samples collected for the profiled application.
   - In the Functions view, select a function from the Functions table, and select Display in Call Graph View.
     This displays the Call Graph constructed from the callstack samples collected from the profiled application.

2. Click on the Display: All Modules link to enable the Display system modules.
   This displays the callstack samples collected from the system modules.
3. Click the OK button to save the Display filter settings.
4. Click on the Process combo box to select the process for which you want to view the Call Graph.
1. To navigate to Call Graph view, do one of the following.
   · In the **Overview** page, right-click on the **5 Hottest Functions** table entry, and select **Open Source Code**. This displays the **Source Code** view for the selected function.
   · In the **Functions** view, select a function from the **Functions** table, right-click and select **Open Source Code** from the context menu. This displays the **Source Code** view for the selected function.
   · In the **Call Graph** view, select a function from the **Functions** table, right-click and select **Open Source Code** from the context menu. This displays the **Source Code** view for the selected function.

2. To view the sources for the some other function, click on the **Function** combo-box, and select the function for which you want to view the source code.
Analysis with Cache Line Utilization (CLU)

**Cache Line Utilization (CLU)** uses the Load and Store Instruction-Based Sampling (IBS) records to provide a measure of how efficiently an application utilizes the L1 data cache.

- Collecting CLU profile data
- Changing the CLU profile collection options
- Analyzing the CLU profile data

© 2015 Advanced Micro Devices. All rights reserved.
Collecting CLU profile data

To collect the CLU data, perform the following steps:

1. First [Create a new CodeXL Project](#)

2. In the toolbar, select **CPU: Cache Line Utilization** from the profile configuration list, as shown in the following figure. This is a predefined Cache Line Utilization profile configuration provided by CodeXL.

3. Set the profile session options as explained in [Set the CPU Profile Session settings](#)

4. Click the Start ▶ icon in the toolbar or select **Profile >**
Start Profiling as shown below.
Changing the CLU profile collection options

CodeXL lets you modify the default profile collection settings. To do this:

1. Set the profile session options as explained in Set the CPU Profile Session settings.
2. By default, the sampling interval for TBP is 1 millisecond. To modify the sampling interval, click Profile > Profile Settings and click on the CPU Profile > Custom node in the left pane tree. In the Edit CodeXL Project Settings dialog box. The Custom CPU Profile Monitored Events page will be displayed.
3. Click the Interval field and set the required sampling interval.
4. Click the OK button to save the modified profile settings.
Analyzing the CLU profile data

This page explains how to analyze and navigate to the various profile views provided by the CodeXL CPU Profiler.

- Overview page
- Modules view
- Functions view
- Source Code view
Overview page

When data collection is complete, CodeXL processes the raw profile data and displays the results.

Initially the **Profile Overview** is displayed. It displays the following profile data:

- 5 **Hottest Functions** table
- 5 **Hottest Modules** table
- 5 **Hottest Processes** table
- **Execution** section
- **Profile Details** section
From the overview page the user can navigate to various pages to analyze the profile data.
1. To navigate to Modules view, do any of the following:
   - In CodeXL Explorer pane, double click on Modules node in the profile session tree entry created for this profile run. The Modules view is displayed.
   - In Overview page, right click on 5 Hottest Modules table entry and select Display in Modules View. The Modules view is displayed.

2. Click on Display: All Data, All Modules link to set the Display Settings. The Display Settings dialog box is displayed to enable the users to set the various display settings.
Modules with a low cache line utilization usually indicate performance bottlenecks. Sort the table according to a specific metric to highlight potential bottle neck modules.
### Functions view

1. To navigate to Functions view, do any of the following:
   - In CodeXL Explorer pane, double click on **Functions** node in the profile session tree entry created for this profile run. The **Functions** view is displayed with function entries from all the profiled processes and the load modules.
   - In **Overview** page, right click on **5 Hottest Modules** table entry and select **Display in Functions View**. The **Functions** view is displayed with function entries from the selected load module.
   - In **Overview** page, right click on **5 Hottest Functions** table entry and select **Display in Functions View**. The **Functions** view is displayed with function entries from all the profiled processes and the load modules.
   - In **Modules** view, select a Process from the **Processes** table, and select **Display in Functions View**. This displays the functions from the selected process and all its load modules.
   - In **Modules** view, select a Module from the **Modules filtered by selected processes** table, and select **Display in Functions View**. This displays the functions from the selected load module.
Source Code view

1. To navigate to Source view, do any of the following:
   - In **Overview** page, right click on **5 Hottest Functions** table entry and select **Open Source Code**. This displays the **Source Code** view for the selected function.
   - In **Functions** view, select a function from the **Functions** table, right click and select **Open Source Code** from the context menu. This displays the **Source Code** view for the selected function.

2. To view the sources for the some other function, click on the **Function** combo box and select the function for which you want to view the source code.
Microsoft Visual Studio

Prior to Visual Studio 2013 Update 3, inline function information was not preserved in the PDB file. For CodeXL, to extract inline information from PDB file, the target application must be compiled with Visual Studio 2013 Update 3 or later. Use the below Visual Studio Configuration Properties if you want to enable inline function details in CodeXL CPU Profiler.

- Set **Configuration Properties > C/C++ > General > Debug Information Format** to **/Zi**. Without this flag, PDB file with debug info will not be generated.
Add `/Zo` to **Configuration Properties > C/C++ > Command Line > Additional Options**. Without this flag, additional debug info related to inline functions will not be generated. `/Zo` only recognized by VS2013 update 3 or later. The `/Zo` option is enabled by default in Visual Studio 2015 when you specify debugging information with `/Zi` or `/Z7`, hence this step is optional for VS2015.
- Enable Optimization by setting any one of: /O1, /O2, /Ox (Configuration Properties > C/C++ > Optimization > Optimization). Setting it to /Od will stop all optimizations including function inline.

- Enable Function inlining by setting one of: /Ob1, /Ob2 (Configuration Properties > C/C++ > Optimization > Inline Function Expansion). Setting it to /Ob0 will stop function inline.
On Visual Studio 2015, enable “Whole Program Optimization” (Configuration Properties > C/C++ > Optimization > Whole Program Optimization). This setting is not required for VS2013.
Disable data elimination by setting `/OPT:NOREF` (Configuration Properties > Linker > Optimization > References). Setting it to `/OPT:REF` eliminates unreferenced data/function during linking time. In Release build, by default REF is enabled, due to which inline function info gets removed from PDB file.

Disable COMDAT folding by setting `/OPT:NOICF` (Configuration Properties > Linker > Optimization > Enable COMDAT Folding). Setting it to `/OPT:ICF` combines identical functions. In Release build, by default ICF is enabled, due to which identical inline functions info gets removed from PDB file.
GCC/G++ Compiler

- Enable debug info using "-g" or "-gdwarf-version". Without this option, target application binary will not include any debugging information.

- Enable optimization using any one of: -O, -O1, -O2, or -O3. Without this option, compiler will disable any function inlining.
Session Views

Following screenshots for Overview, Call Graph view and Source Code view shows the inline functions for a sample target application.
Inline function "multiply_matrices(void)" is shown in the 5 Hottest Functions table.
Call Graph View
Inline function “multiply_matrices(void)” is shown in the Functions table and Paths table.
Source Code View
Samples are attributed to corresponding source lines for Inline function “multiply_matrices(void)”. 
Settings for CPU Profiler to run on Virtual Machine

CPU Profiler supports profiling on guest OS running on VMware workstation.
VMware Workstation

CPU Profiler supports TBP and EBP on guest OS running on VMware Workstation 11.0 or later. Recent AMD Carrizo processor is not yet supported by VMware Workstation 11.1.x.

To run TBP within guest OS, no additional configuration needed in host OS or guest OS.

To run EBP within guest OS, please ensure the following settings are done:
- Enable Virtualization or SVM (AMD-V) in BIOS settings before booting the host OS.
- Enable AMD-V in guest OS VM settings. Edit virtual machine settings > Hardware > Processors> Virtualization engine > Enable “Virtualize AMD-V/RVI”
- Enable vPMC in guest OS VM settings. Edit virtual machine settings > Hardware > Processors> Virtualization engine > Enable “Virtualize CPU performance counters”

**Known Issues on Windows 7 Host OS:**
- When CPU Profiler EBP is running on Windows 7 host OS and a Linux guest OS is launched, crash is observed on Windows 7 due to VMware driver.
- If EBP is performed on Windows 7 host OS and EBP is performed on Linux guest OS simultaneously, then crash is observed on Windows 7 due to VMware driver.

These scenarios work fine when host OS is Windows 8, 8.1 and 10.
How to use CPU Profiler Control APIs

CPU Profile control APIs provide the option to restrict the profiling to a specific part of the code. Follow the below steps to call, compile and invoke these APIs from your target program.
Required File Paths

All the paths mentioned below are with respect to CodeXL installation directory (CODEXL-INSTALL-DIR).

**API Documentation:**

- `<CODEXL-INSTALL-DIR>\SDK\AMDTActivityLogger\doc\AMDTActivityLogger.pdf` (on Windows platforms)
  OR
- `<CODEXL-INSTALL-DIR>/SDK/AMDTActivityLogger/doc/AMDTActivityLogger.pdf` (on Linux platforms)

**Header File:**

- `<CODEXL-INSTALL-DIR>\SDK\AMDTActivityLogger\include\AMDTActivityLogger.h` (on Windows platforms)
  OR
- `<CODEXL-INSTALL-DIR>/SDK/AMDTActivityLogger/include/AMDTActivityLogger.h` (on Linux platforms)

**LIB Files (on Windows platforms):**

- `<CODEXL-INSTALL-DIR>\SDK\AMDTActivityLogger\lib\x64\AMDTActivityLogger-x64.lib`
- `<CODEXL-INSTALL-DIR>\SDK\AMDTActivityLogger\lib\x86\AMDTActivityLogger.lib`

**DLL Files (on Windows platforms):**

- `<CODEXL-INSTALL-DIR>\SDK\AMDTActivityLogger\bin\x64\AMDTActivityLogger-`
x64.dll
<CODEXL-INSTALL-DIR>\SDK\AMDTActivityLogger\bin\x86\AMDTActivityLogger.dll

Shared Library (.so) Files (on Linux platforms):
<CODEXL-INSTALL-DIR>/SDK/AMDTActivityLogger/bin/x86_64/libAMDTActivityLogger.so
<CODEXL-INSTALL-DIR>/SDK/AMDTActivityLogger/bin/x86/libAMDTActivityLogger
Calling the APIs

Include the header file AMDTActivityLogger.h, and call the resume and pause APIs within the code. The code encapsulated within resume-pause API pair will be profiled by CPU Profiler.

These APIs can be called multiple times to profile different parts of the code. These APIs can be spread across multiple functions, i.e. resume called from one function and stop called from another function. These APIs can be spread across threads, i.e. resume called from one thread and stop called from another thread of the same target application.

In the below example, the CPU Profiler is restricted to the execution of multiply_matrices() function.

```c
#include <AMDTActivityLogger.h>

int main(int argc, char* argv[]) {
    // Initialize the matrices
    printf("Initializing matrices\n");
    initialize_matrices();

    // Multiply the matrices
    printf("Multiplying matrices\n");

    // Resume the CPU profiler
    // amdtrResumeProfiling(AMDT_CPU_PROFILING);
    amdtrResumeProfilingEx();

    multiply_matrices();

    // Stop the CPU Profiler
    // amdtrStopProfiling(AMDT_CPU_PROFILING);
    amdtrStopProfilingEx();

    return 0;
}
```
Compiling the target application

To compile the application on Microsoft Visual Studio, update the configuration properties to include the path of header file, DLL file, LIB file of AMDTActivityLogger.

To compile a C++ application on Linux using G++, use the following command:

```
$ g++ -std=c++0x <source-file.cpp> -L<CODEXL-INSTALL-DIR>/SDK/AMDTActivityLogger/bin/x86_64/ -L<CODEXL-INSTALL-DIR> -I<CODEXL-INSTALL-DIR>/SDK/AMDTActivityLogger/include -lAMDTBaseTools -lAMDTOSWrappers -lAMDTActivityLogger -lrt -ldl
```

You may choose to use 2011 C++ standard or newer C++ standard while compiling C++ target application with profile control APIs.

To compile a C application on Linux using GCC, use the following command:

```
```
After the compiling the target application, create a project in CodeXL, set the general profile session options, set the CPU profile session options. While setting the CPU profile session options, in step 4, in the Data Collection Schedule section, select Start profile with data collection paused.

Once all the settings done, start the CPU profiling. The CPU Profiler will be in pause state and target application execution begins. When the resume API gets called from target application, CPU Profile starts profiling till stop API gets called from target application. As soon as stop API is called in target application, CPU Profiler stops profiling and waits for next control API call.

To profile from CLI, option ‘-s 0’ should be used to start the profiler in pause state.

Sample command on Windows platforms:

```cmd
> CodeXL CPUProfiler.exe collect -m tpb -s 0 -o c:\Temp\cpuprof-tbp <target-application.exe>
```

Sample command on Linux platforms:

```bash
$ CodeXL CPUProfiler collect -m tpb -s 0 -o /tmp/cpuprof-tbp <target-application>
```
GPU Profiler Tutorial

The CodeXL GPU Profiler is a performance analysis tool that gathers data from the OpenCL run-time and AMD Radeon™ GPUs during the execution of an OpenCL application.

Use this information to discover bottlenecks in an application and to find ways to optimize the application's performance for AMD platforms. There are two modes of operation supported by the Profiler.

Collecting OpenCL Application Trace

Collecting OpenCL GPU Kernel Performance Counters

© 2015 Advanced Micro Devices. All rights reserved.
Run an Application Trace GPU Profile session

1. Open or create a CodeXL project.
2. Select the Application Timeline Trace profile mode.
3. Click the (Start Profiling) toolbar button to start profiling.
4. Stop the profiled application.
When the profiled application’s execution is over, CodeXL displays the session.
Timeline View
The Timeline View provides a visual representation of the execution of the application. Along the top of the timeline is the time grid, showing the total elapsed time (in milliseconds) of the application when fully zoomed out. Timing begins when the first OpenCL call is made by the application; it ends when the final OpenCL call is made. Directly below the time grid, each host (OS) thread that made at least one OpenCL call is listed. For each host thread, the OpenCL API calls are plotted along the time grid, showing the start time and duration of each call.

Below the host threads, the OpenCL tree shows all contexts and queues created by the application, along with data transfer operations and kernel execution operations for each queue. We can navigate in the Timeline View by zooming, panning, collapsing/expanding or selecting a region of interest. From the Timeline View, we can also navigate to the corresponding API call in the API Trace View and vice versa. The Timeline View can be useful for debugging your OpenCL application. Some examples are:
· You easily can confirm that the high-level structure of your application is correct. By examining the timeline, you can verify that the number of queues and contexts created match your expectations for the application.

· You also can confirm that synchronization has been performed properly in the application. For example, if kernel A execution is dependent on a buffer operation and outputs from kernel B execution, then kernel A execution appears after the completion of the buffer execution and kernel B execution in the time grid. It can be hard to find this type of synchronization error using traditional debugging techniques.

· You also can confirm that the application has been using the hardware efficiently: the timeline should show that non-dependent kernel executions and data transfer operations occur simultaneously.

**Summary Pages View**

The Summary Pages View shows various statistics for your OpenCL application. It can provide you with a general idea of the location of the application's bottlenecks. It also provides useful information, such as the number of buffers and images
created on each context, the most expensive kernel call, etc. The Summary Pages View provides access to the following individual pages:

- **API Summary page** — Shows statistics for all OpenCL API calls made in the application for API hotspot identification.
- **Context Summary page** — Shows the statistics for all the kernel dispatch and data transfer operations for each context. It also shows the number of buffers and images created for each context.
- **Kernel Summary page** — Shows statistics for all the kernels that are created in the application.
- **Top 10 Data Transfer Summary page** — Shows a sorted list of the ten most time-consuming individual data transfer operations.
- **Top 10 Kernel Summary page** — Shows a sorted list of the ten most time-consuming individual kernel execution operations. From these summary pages, it is possible to determine whether the application is bound by kernel execution or data transfer (Context Summary page). If the application is bound by kernel execution, you can determine which device is the bottleneck. From the Kernel Summary page, we can find the name of the kernel with the highest total execution time. Or, from the Top 10 Kernel Summary page, we can find the individual kernel instance with the highest execution time. If the kernel execution on a GPU device is the bottleneck, the GPU performance counters then can be used to investigate the bottleneck inside the kernel. See Collecting OpenCL GPU Kernel Performance Counters for more details. If the application is bound by the data transfers, it is possible to determine the most time-consuming data transfer type (read, write, copy or map) in the application from the Context Summary page. You can investigate whether it is possible to minimize this type of data transfer by modifying the algorithm. With help from the Timeline View, you can investigate whether data transfers have
been executed in the most efficient way (that is: concurrently with a kernel execution).

API Trace View

<table>
<thead>
<tr>
<th>Host Thread ID</th>
<th>Interface</th>
<th>Parameters</th>
<th>Result</th>
<th>Device Block</th>
<th>Kernel Occupancy</th>
<th>CPU Time</th>
<th>Device Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>26170</td>
<td>clReleaseEvent</td>
<td>0x0C54F600</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0215</td>
<td></td>
</tr>
<tr>
<td>26171</td>
<td>clFlush</td>
<td>0x0F493070</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0253</td>
<td></td>
</tr>
<tr>
<td>26172</td>
<td>clEnqueueWriteBuffer</td>
<td>0x0F5C36F0 (0x19575C4B, CL_FALSE, 0x320, 0x6E6F7D, 0xNULL, 0xNULL)</td>
<td>CL_SUCCESS</td>
<td></td>
<td>288 Byte WRITE..</td>
<td>0.0127</td>
<td>0.0083</td>
</tr>
<tr>
<td>26173</td>
<td>clEnqueueNDRangeKernel</td>
<td>0x0F5C36F0 (0x19575E40, 0x16D, 0x117, 0xNULL, 0xNULL, 0xNULL)</td>
<td>CL_SUCCESS</td>
<td></td>
<td>compute kernels..</td>
<td>0.0387</td>
<td>0.0026</td>
</tr>
<tr>
<td>26174</td>
<td>clFlush</td>
<td>0x0F5C36F0</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0039</td>
<td></td>
</tr>
<tr>
<td>26175</td>
<td>clEnqueueUnmapObjects</td>
<td>0x0F5C36F0 (0x19575C4B, 0xNULL, 0xNULL)</td>
<td>CL_SUCCESS</td>
<td></td>
<td>ACQUIRE_GL OBJ..</td>
<td>0.0018</td>
<td></td>
</tr>
<tr>
<td>26176</td>
<td>clEnqueueCopyBufferToImage</td>
<td>0x0F5C36F0 (0x19575D00, 0x1957595C, 0x60, 0x5C, 0x0, 0x4, 0x6, 0x5C, 0xNULL)</td>
<td>CL_SUCCESS</td>
<td></td>
<td>4.0 MB COPY BUF..</td>
<td>0.0132</td>
<td>0.0044</td>
</tr>
<tr>
<td>26177</td>
<td>clEnqueueUnmapObjects</td>
<td>0x0F5C36F0 (0x19575C4B, 0xNULL, 0xNULL)</td>
<td>CL_SUCCESS</td>
<td></td>
<td>RELEASE_GL OBJ..</td>
<td>0.0253</td>
<td></td>
</tr>
<tr>
<td>26178</td>
<td>clEnqueueMapBuffer</td>
<td>0x0F5C36F0 (0x19575D00, CL_FALSE, CL_MAP_READ, 0x19575C, 0xNULL, 0x19533F00, 0xCL_SUCCESS)</td>
<td>0x195400</td>
<td></td>
<td>31.2 KB MAP BUF..</td>
<td>0.0147</td>
<td>0.0024</td>
</tr>
<tr>
<td>26179</td>
<td>clFlush</td>
<td>0x0F5C36F0</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0040</td>
<td></td>
</tr>
<tr>
<td>26180</td>
<td>clFinish</td>
<td>0x0F5C36F0</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0517</td>
<td></td>
</tr>
<tr>
<td>26181</td>
<td>clReleaseEvent</td>
<td>0x0F5C36F0</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0207</td>
<td></td>
</tr>
<tr>
<td>26182</td>
<td>clEnqueueMapMemObject</td>
<td>0x0F5C36F0 (0x19575D00, 0x19575C, 0xNULL, 0xNULL)</td>
<td>CL_SUCCESS</td>
<td></td>
<td>UNMAP MEM OBJ..</td>
<td>0.0254</td>
<td>0.0108</td>
</tr>
<tr>
<td>26183</td>
<td>clFinish</td>
<td>0x0F5C36F0</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0254</td>
<td></td>
</tr>
<tr>
<td>26184</td>
<td>clFinish</td>
<td>0x0F5C36F0</td>
<td>CL_SUCCESS</td>
<td></td>
<td></td>
<td>0.0254</td>
<td></td>
</tr>
</tbody>
</table>

The API Trace View lists all the OpenCL API calls made by the application. Each host thread that makes at least one OpenCL call is listed in a separate tab. Each tab contains a list of all the API calls made by that particular thread. For each call, the list displays the index of the call (representing execution order), the name of the API function, a semi-colon delimited list of parameters passed to the function and the value returned by the function. When displaying parameters, the Profiler tries to dereference pointers and decode enumeration values to give as much information as possible about the data being passed in or returned from the function. Double-clicking an item in the API Trace View displays and zooms into that API call in the Host Thread row in the Timeline View.

This view lets you analyze and debug the input parameters and
output results for each API call. For example, you easily can check that all the API calls return CL_SUCCESS, or that all the buffers are created with the correct flags. This view also lets you identify redundant API calls.
Run a Performance Counters GPU Profile session

1. Open or create a CodeXL project.
2. Select the GPU: Performance Counters profile mode.
3. Click the (Start Profiling) toolbar button to start profiling.
4. Stop the profiled application when the part of the application that is under investigation has completed its
When the profiled application’s execution is done, CodeXL displays the session.

The GPU kernel performance counters can help find possible bottlenecks in the kernel execution. You can find the list of performance counters supported by AMD Radeon™ GPUs in the tool documentation. Once we have used the trace data to discover which kernel most requires optimization, we can collect the GPU performance counters to drill down into the
kernel execution on a GPU device. Using the performance counters, you can:

- Find the number of resources (general-purpose registers, local memory size, and flow-control stack size) allocated for the kernel. These resources affect the possible number of in-flight wavefronts in the GPU. A higher number better hides data latency.
- Determine the number of ALU, global, and local memory instructions executed by the GPU.
- Determine the number of bytes fetched from, and written to, the global memory.
- Determine the use of the SIMD engines and memory units in the system.
- View the efficiency of the shader compiler in packing ALU instructions into the VLIW instructions used by AMD GPUs.
- View any local memory (Local Data Share - LDS) bank conflicts. The Session View (see the screenshot above) shows the performance counters for a profile session. The output data is recorded in a comma-separated variable (.csv) format. You can also click on the kernel name entry in the "Method" column to view the OpenCL kernel source, AMD Intermediate Language (IL), GPU ISA, or CPU assembly code for that kernel.
Switching to Analyze mode

Option 1- Analyze mode button:

Click on the Analyze Mode button in the CodeXL Mode toolbar:

Option 2- Main menu:

Open the Analyze menu from menu bar and select the ‘Switch to Analyze Mode‘ command:

After you switch to Analyze mode, you can also create a new project, open a previously saved project, or load the Teapot or
Matrix Multiply samples.
Creating a new project for Analysis

Click on the “File->Create Project”, or use the Ctrl+N shortcut. The following CodeXL Project Settings dialog will appear:

Rename the project, and click on the OK.

After the new project has been created, the CodeXL Analyzer Explorer Tree should appear in the left pane:
If you are familiar with the former versions of the Analyzer, you probably noticed that the tree has a different structure than the one used in previous versions. Let’s examine the structure of the new CodeXL Analyzer Explorer:

2. **Programs and Folders:** before describing how to technically create Programs and Folders, let’s first discuss what those objects are, and why they can be useful.

   c. **Programs (OpenGL, Vulkan):**
      
      As of version 2.0, CodeXL can compile and link together multiple source files for OpenGL and Vulkan. This is especially important when different shaders have mutual impact on one another’s ISA and performance statistics. To provide that type of support, CodeXL Analyzer introduced the concept of a Program. There are two types of Programs in CodeXL 2.0:
      
      - Rendering Programs
      - Compute Programs

      A **Rendering Program** represents a graphics pipeline, and can have a single shader attached to each of its stages:
      
      - Vertex
      - Tessellation Control
      - Tessellation Evaluation
      - Geometry
      - Fragment

      A **Compute Program** represents a compute pipeline, and can have a single compute shader attached to its single stage.

      When you build a program that has multiple shaders attached to it, all shaders are being compiled and linked together. This way, you get more accurate ISA and performance statistics than those generated using
previous versions of CodeXL.

d. **Folders (OpenCL, DirectX):**

Folders are logical containers of source files. When you build a folder that has multiple source files attached to it, the source files are simply being built one after the other. Unlike programs, there is no kind of interdependency between the source files in a given folder: when a folder is being built, each source file is being compiled independently. Folders can be used to organize the project, by serving as a logical separator. They can also be used to ease the process of comparing build results, since now the build results are being maintained per-folder: you can create two different Folders, each containing the same source files, but have a different configuration (for example, create two DirectX Folders, each with a different shader model). After building the two Folders, you can toggle between the performance statistics of the two Folders to see the differences.

You may ask yourself why CodeXL does not support the concept of DirectX Programs, just like it does for OpenGL and Vulkan. This is a good point. Supporting DirectX Programs is at a high priority in the Analyzer’s roadmap, and we will do our best to add that feature in the upcoming versions of the product.

**Creating a new Program or Folder**

To create a new Program or a Folder, double-click on the “Create new program/folder” item in CodeXL Analyzer Explorer Tree:

Then, the following dialog would pop-up:
Select the Program/Folder type of choice, and click OK.

Then, the empty Program/Folder would appear in the Explorer Tree. For Example, if you choose an OpenGL Rendering Program, you will see an empty OpenGL Rendering Program created:
Working with Programs

After creating a new program, you will see that it contains an empty placeholder for every pipeline stage. Right-click on any stage to add an existing shader or create a new one:

![Image of CodeXL Explorer]

**Note**: You can also double-click on a stage to create a new shader and automatically attach it to that Program’s stage.

As you can see in the above screenshot, we attached SimpleVertexShader.vs as the vertex shader to our OpenGL Rendering Program, and it was also automatically added to the Source Files pool. We can now drag SimpleVertexShader.vs from the Source Files pool and drop it on the stage node of any Program that we may add to the project, to reuse
SimpleVertexShader.vs (there is no dependency in the build process between different Programs).

To build the program, right-click on it and select the Build option, or use the F7 shortcut:

You can also select the Program and manually click on the Build button in the Analyzer toolbar:
Working with Folders

After creating a new OpenCL or DirectX Folder, an empty Folder would be listed in the Explorer Tree:

To create a new source file, and automatically add it to the Folder, double-click on the “Create new source file item...” item of the folder:
To add an existing source file, and automatically add it to the Folder, double-click on the “Add existing source file item...” item of the folder:
To configure the build properties of a source file under a specific Folder, click on that source file and use the Analyzer toolbar’s Type and Entry point drop-down lists. The first sets the type of the shader and the latter specifies the specific target shader (among the shaders in the source file). This configuration is Folder-specific. That is, the same source file can be set with different properties under different Folders. CodeXL will remember those configurations for you.

To configure the build properties of the Folder, click on the Folder and adjust the enabled items in the Analyzer toolbar. For CodeXL 2.0, this is only relevant to the DX Shader Model property of DX Folders:
Once set, the DX Shader Model value will hold for all the shaders in the selected Folder. For example, if you choose 5_0 as the DX Shader Model, any D3D vertex shader in that Folder will be compiled using shader model vs_5_0.

To build the whole Folder, right-click on it and select the Build item:

Unlike the case with Programs, Folders are more flexible as they allow you to build selected source files, without being required to build the whole Folder. To build selected source files, click on the selected source files under the program, while holding the Ctrl key. Then, right-click on one of the selected files and select the build option:
Selecting target devices

CodeXL Analyzer can target a variety of devices, independent of the device that is physically installed on your system. To select the target devices, for which the build would be performed, first click on the Select Devices button in the Analyzer toolbar:

Then, the CodeXL Options dialog would pop-pup with its Analyze tab activated. The devices are grouped by generations. You can use the check boxes to select and remove devices:
In the Static Analyze toolbar, you can define specific OpenCL or HLSL build options:

The Build Options box is a place to set compiler build flags such as `-x clc++` or `-o3`. Any compiler build flag can be placed in this box.

You can set the build options by typing the options directly in the designated text box or by using the OpenCL/HLSL Build Options dialog.
OpenCL and DirectX build

In the Static Analyze toolbar, you can define specific OpenCL or DirectX build options. The Build Options box is a place to set compiler build flags. Any compiler build flag can be placed in the designated text box or by using the OpenCL/HLSL Build options by typing the options directly in.
The Statistics and Analysis Data

Kernels Statistic view

The kernel statistics tab gives detailed statistics for the selected kernel for each target device. To open the statistics tab, expand the desired kernel in the project tree, and double-click the Statistics node:

Statistics page for devices GCN devices:
This view is focused on giving the programmer the wave constraints based on the SGPRs, VGPRs and LDS size.

**Shaders Statistic view**

The shader statistics tab gives detailed statistics for the
selected shader for each target device. To open the statistics tab, expand the desired shader in the project tree, and double-click the Statistics node: **Note:** the statistics will be available only for V6 generation and later.
This view is focused on giving the programmer the wave constraints based on the SGPRs and VGPRs.
The performance statistics tab will be opened automatically when the build process is over. To view the compilation output, double click the node of the desired ASIC in the explorer tree, under the Program/Folder and configuration (32-bit or 64-bit):

This action will open a tab containing the source code, the AMD IL and the ISA:
Opening several build results of different devices for the same kernel/shader will open different IL/ISA tabs in the same “Source Code/IL/ISA” view as can be seen in the above image.
The context menu enables showing/hiding different parts of the view, enables showing/hiding line numbers, and enables different edit actions depending on the selection of where the context menu was opened and whether the view section is editable or not.
Build Options- Defining compilation options

In the Static Analyze toolbar, there is a space where you can define specific kernel/shader build options:

Build Options Dialog

This dialog will help you choose the correct kernel/shader build options for you and hopefully will prevent making spelling mistakes while typing the options manually.

To open the dialog, press The Button. The dialog will be opened.
For OpenCL build options - you can browse between the "General & Optimization" tab and the "Other" tab to view all the available options.
For DirectX build options – choose the “HLSL Build Options” tab.

Once you choose an option, the option text is displayed in the text box marked ".. Build Command Line" that appears below. This string will also appear in the menu bar after you click the OK button.

Typing the option in the text box will also mark it in the appropriate area in the dialog, or select it in the appropriate
drop box.
DISCLAIMER
The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of non-infringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. AMD, the AMD Arrow logo, AMD Radeon, AMD FirePro, gDEBugger, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Windows and DirectX are registered trademarks of Microsoft Corporation. PCIe and PCI Express are registered trademarks of PCI-SIG Corporation. OpenCL is a trademark of Apple Inc. Linux is a registered trademark of Linus Torvalds. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.
© 2016 Advanced Micro Devices, Inc. All rights reserved.

[AI] Please review this cross-reference once and let me know whether it is correct.

[US] This should point to the glossary page title of the same name.

© 2015 Advanced Micro Devices. All rights reserved.