|  |  |
| --- | --- |
|  | AMD_E_Blk_RGB.png |

# AMD CodeXL White Paper

Version 1.7

## Purpose

The purpose of this document is to provide an “under-the-hood” view of some of the CodeXL components.

For additional documentation, check out the CodeXL Getting Started guide and the CodeXL User Guide on the [CodeXL Web Page](http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/).

## CodeXL Components and Features

AMD CodeXL contains the following executables:

* CodeXL graphic standalone application
* CodeXL Visual Studio Extension (Windows only)
* CodeXL Remote Agent application
* CodeXL command line tools:
* SProfile (GPU Profiler)
* CodeXLAnalyzer
* CodeXLPowerProfiler
* CodeXLCpuProfiler

AMD CodeXL provides the following components:

* GPU Debugger
* GPU Profiler
* CPU Profiler
* Static Analyzer
* Power Profiler

AMD CodeXL supports Windows and Linux. For system requirements see the CodeXL Release Notes on the [CodeXL Web Page](http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/).

For a detailed list of features see the CodeXL User Guide on the [CodeXL Web Page](http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/).

## CodeXL Usage Scenarios

Local Scenario #1: Use the Standalone application or the VS Extension to debug/profile/analyze on the local station and view the results in the GUI.

Local Scenario #2: Use the command line tools to debug/profile/analyze on the local station. Use text editors to view the results or import them into the CodeXL standalone application or VS Extension.

Remote Scenario #1: Run the CodeXL Remote Agent on the target platform, and run the Standalone application or the VS Extension on the development station to debug/profile on the target platform and view the results in the GUI.

Remote Scenario #2: Log in to the target platform over the network, and use the CodeXL command line tools to profile on the remote station. Copy the session files to the local development station and import them into CodeXL standalone application or VS Extension to view the results in the GUI.



Figure Debugging/profiling on a remote target platform with CodeXL

## CodeXL Architecture



Figure CodeXL components and interaction during local debug/profile scenarios



Figure CodeXL components and interaction during remote debug/profile scenarios

## CodeXL Framework

The CodeXL components make use of framework of classes for OS abstraction, UI functionality, logging, assertion handlers, wrappers for OpenCL/OpenGL runtime API and general programming purposes.



Figure Modules hierarchy of CodeXL components and framework

## GPU Debugger

The GPU Debugger performs OpenCL and OpenGL API interception by DLL replacement - forcing the debugged application to load the debugger’s custom OpenCL.dll and opengl32.dll modules (or libOpenCL.so and libGL.so on Linux) instead of the system modules. These modules expose an API identical to that of the original OpenCL and OpenGL runtimes. When the debugged application makes an API call, it is actually calling the debugger’s API wrapper implementation which notifies the debugger front-end of the call, stores the parameters and other information for later reference, and then propagates the call to the ‘real’ OpenCL or OpenGL dll. If the user set a breakpoint on this API then the debugger suspends execution when the API interception occurs.



Figure 5 GPU Debugger interception of OpenCL/OpenGL API calls, when debugging 64-bit Windows applications

Depending on the operating system and the architecture of the debugged application, the debugger uses a different control chain that may include the CodeXL Remote Debugging Server or gdb or direct communication with the debugger backend, as shown in the table below:

|  |  |  |
| --- | --- | --- |
| **Debugged app runs on** | **Debugged app architecture** | **Debugger control chain** |
| Local Windows Station | 32-bit | CodeXL Debugger Front-End  ⇩  CodeXL Debugger Backend inside the debugged application |
| Local Windows Station | 64-bit | CodeXL Debugger Front-End  ⇩  CodeXL Remote Debugging Server  ⇩  CodeXL Debugger Backend inside the debugged application |
| Remote Windows Station | 32/64-bit | CodeXL Debugger Front-End  ⇩  CodeXL Remote Debugging Server  ⇩  CodeXL Debugger Backend inside the debugged application |
| Local Linux Station | 32/64-bit | CodeXL Debugger Front-End  ⇩  gdb  ⇩  CodeXL Debugger Backend inside the debugged application |
| Remote Linux Station | 32/64-bit | CodeXL Debugger Front-End  ⇩  gdb  ⇩  CodeXL Debugger Backend inside the debugged application |

### OpenCL Kernel Debugging

The GPU Debugger backend intercepts the compilation of OpenCL kernels inside the debugged application during the debug session, and forces them to be compiled with debug information. When the user sets a breakpoint on a kernel source line, or steps through a kernel, the debugger rebuilds the kernel and employs IL Patching to suspend the kernel execution at the breakpoint location.

Here is how the IL Patching is performed. Assume that the kernel source code includes 20 lines of code, and the user has set a breakpoint at line 15.

Step I: Generate IL  
The debugger first calls the high-level compiler to generate Intermediate Language (IL) representation of the kernel, using the -g flag to force the generation of debug information

Step II: Patch the IL  
The debugger uses the debug information provided by the compiler to identify the IL instruction that corresponds to the location of the user breakpoint in the OpenCL source code. At this location in the IL block the debugger inserts additional IL instructions that will dump the values stored in its registers to a special debug buffer and then end the kernel execution.

Step III: Generate ISA  
The debugger passes the patched IL block to the finalizer which generates ISA from it. The generated ISA includes ISA instructions that correspond to IL instructions inserted by the debugger.

Step IV: Execute the kernel  
The debugger now enqueues the patched kernel for execution. The kernel execution dumps its register values to a debug buffer when it reaches the instructions inserted by the debugger, and then the execution ends.

Step V: Display debug information relevant to the breakpoint location  
The debugger collects the register values from the debug buffer, and using the debug information generated by the compiler it matches each high level source code variable with the register that contains its value. The debugger then displays the values stored for each of the OpenCL variable.

Step VI: Continue execution  
If the user makes another step through he kernel source code, the debugger repeats the process of Steps II till V, removing the IL patch f the previous step and patching the kernel IL at the new step location. It then executes the patched kernel which ends execution at the new step location. To the user this creates an illusion of one step forward in the kernel when in fact the whole kernel was executed from scratch and execution was allowed to continue one step further then in the previous step.



Figure Example of GPU Debugger performs IL Patching

## GPU Profiler

The GPU Profiler provides 2 types of profiling sessions: Application Timeline Trace and Performance Counters.

### GPU Profiling: Application Timeline Trace

The application timeline provides a visual representation of the execution of the application. This session type displays a time grid which shows the total elapsed time, in milliseconds, of the application, and all OpenCL activity: every API call, transfer operations and kernel execution. Timing begins when the first OpenCL™ call is made by the application; it ends when the final OpenCL™ call is made.

The profiler intercepts the API calls made by the profiled application, stores the details of the call and then calls the “real” implementation of the API, in the OpenCL/DirectX/DirectCompute module.

### GPU Profiling: Performance Counters

The CodeXL GPU Profiler’s Performance Counters session executes each kernel multiple times (up to 3 times, see next paragraph) during a profile session to collect the profiling data regarding that kernel. The profiler stores the values of kernel input parameters and restores them after each execution except the last execution.

The reason the profiler performs multiple executions is that the collection of counters in a single pass has limited capacity. To collect a large set of counters requires multiple passes. To collect all the supported counters and measure the kernel execution time requires 3 passes. The good news is you can control the number of passes that the profiler will perform. The bad news is that fewer passes means less collected data. That’s the tradeoff.

To control the number of passes, open the CodeXL Project Settings dialog (click CTRL+P), select the Profile->”GPU Profile Perf. Counters” node. The text line below the tree displays the number of passes the profiler will perform to collect the selected counters. Uncheck counters in the counter tree and the “Measure kernel execution time (requires an additional pass)” check box below the tree to reduce the number of passes.

#### Kernels that use SVM and pipes

If the number of passes is 2 or more then the profiler will not collect data for kernels that use Shared Virtual Memory or take pipes as input. The profiler will also not execute these kernels multiple times. This is because it is impossible to restore the input of these kernels to its original state before performing additional passes.

### Intercepting API calls inside the profiled application

API Interception is performed in 2 different ways:

* For OpenCL and HSA, the profiler uses the agent mechanism that is an internal AMD interface in the OpenCL/HSA runtime implementations.
* For DirectCompute, the profiler uses Microsoft Detours to intercept the API calls.

Here’s how the agent mechanism works in the AMD OpenCL runtime:

When the runtime is initialized, it checks an environment variable called CL\_AGENT.  The value of that environment variable is a string containing one or more comma-delimited DLL/SO names.  The runtime will load those libraries and call an exported function from each library (the function is called clAgent\_OnLoad). It passes a cl\_agent pointer to the clAgent\_OnLoad function.  Using this pointer, you can query the current ICD dispatch table (a table of function pointers, with one entry for each OpenCL API).  Using that table, you can override the function pointers to point to your own instances of the API call.  Once you set up the table, there is then an API call where you can give the modified dispatch table back to the runtime.

The runtime will now use the modified dispatch table whenever the application calls an API function.

Here’s how the profiler backend uses this:

Before launching the process to profile, sprofile will set the environment variable to one or more of its agents.  The two typical cases used by CodeXL are:

1. To set it to both the API Trace agent and the Kernel Occupancy agent, or
2. To set it to the Performance Counters agent and the Kernel Occupancy agent.

It then launches the application to profile.  When the application makes its first OpenCL API call, the OpenCL runtime is initialized, and it checks the environment variable and calls clAgent\_OnLoad in each specified agent module.

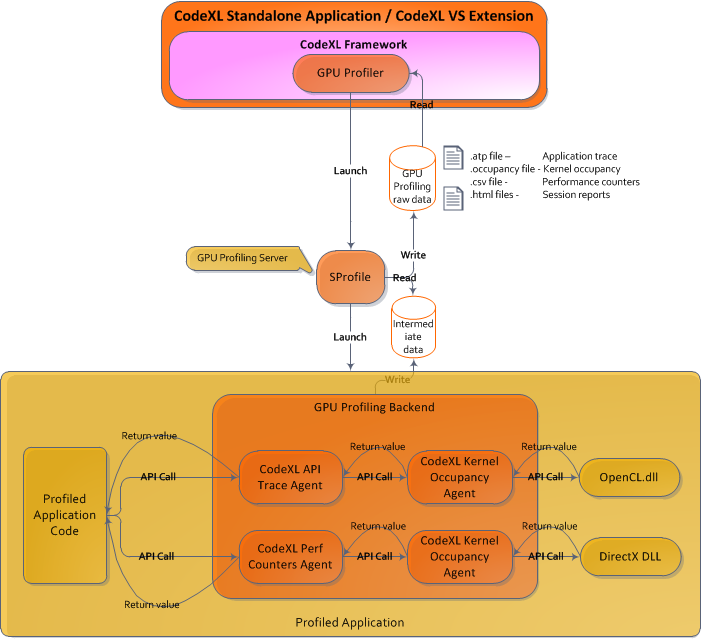
Each agent stores the dispatch table it gets from the runtime (these are the “real” functions that are called from within the intercepted APIs), and then overwrites the function pointers for the API calls it cares about. Both the trace agent and the perf counter agent override all APIs, while the kernel occupancy agent only overrides clEnqueuNDRangeKernel and clReleaseContext. The agents then pass the modified dispatch table back to the runtime.  If more than one agent is specified, when the second agent queries the current dispatch table, it will actually get a dispatch table containing the modified table that the first agent passed back to the runtime (this mechanism is what allows multiple agents to work correctly).

In order to ensure a properly running application, each agent must call the “real” function from within an intercepted API (there is one exception to this rule on the HSA side, but that is an implementation detail).

It’s effectively a daisy-chain of agents. For example, when two agents are used:

1. The Application calls an OpenCL API
2. The runtime directs that call to the dispatch table which is an intercepted function in the first agent
3. The first agent does what it needs to do and then calls the “real” function
4. The “real” function that the first agent sees is actually the modified function in the second agent, so the second agent does what it needs to do and then calls the “real” function
5. The “real” function the second agent sees is now the true “real” implementation in the runtime.

Some of the mechanics are slightly different for HSA, but the concept is quite similar.



## Static Analyzer

The CodeXL Static Analyzer uses the OpenCL, DirectX and OpenGL offline compilers available in the Catalyst package. The term ‘offline compiler’ means that the build is performed for a target device that may or may not be installed on the local station.

For OpenCL and OpenGL, CodeXL passes the high-level source code (OpenCL or GLSL), build options that the user specified and a list of target devices to the relevant offline compiler. The compiler returns:

* Build result  
  Is the build successful or not and warning and error messages for the built source
* If the build is successful, the compiler also returns a binary ELF package that contains
  + Intermediate representation AMDIL for OpenCL 1.2, HSAIL for future paths
  + The generated final ISA
* ISA Statistics

For DirectX the path is slightly different and contains an extra link in the chain. CodeXL passes the high-level source code (HLSL) and the build options that the user specified to the Microsoft DirectX compiler which returns a blob of DirectX assembly if the build is successful, warnings and error messages if the build failed. If the build is successful CodeXL passes the blob to the AMD DXX compiler which returns an ELF package with ISA and ISA statistics.

CodeXL performs disassembly of the ISA to generate a textual view that can be displayed, and also parses the ISA and analyzes it.

CodeXL uses reference tables to display the implications of the ISA statistics on the maximum number of wavefronts that can be executed concurrently for the built kernel/shader.



## CPU Profiler

Driver collects data and dumps it to raw data files on local hard-disk

Frontend reads the data, performs post-processing and displays it to the user.