## GPU Profiler

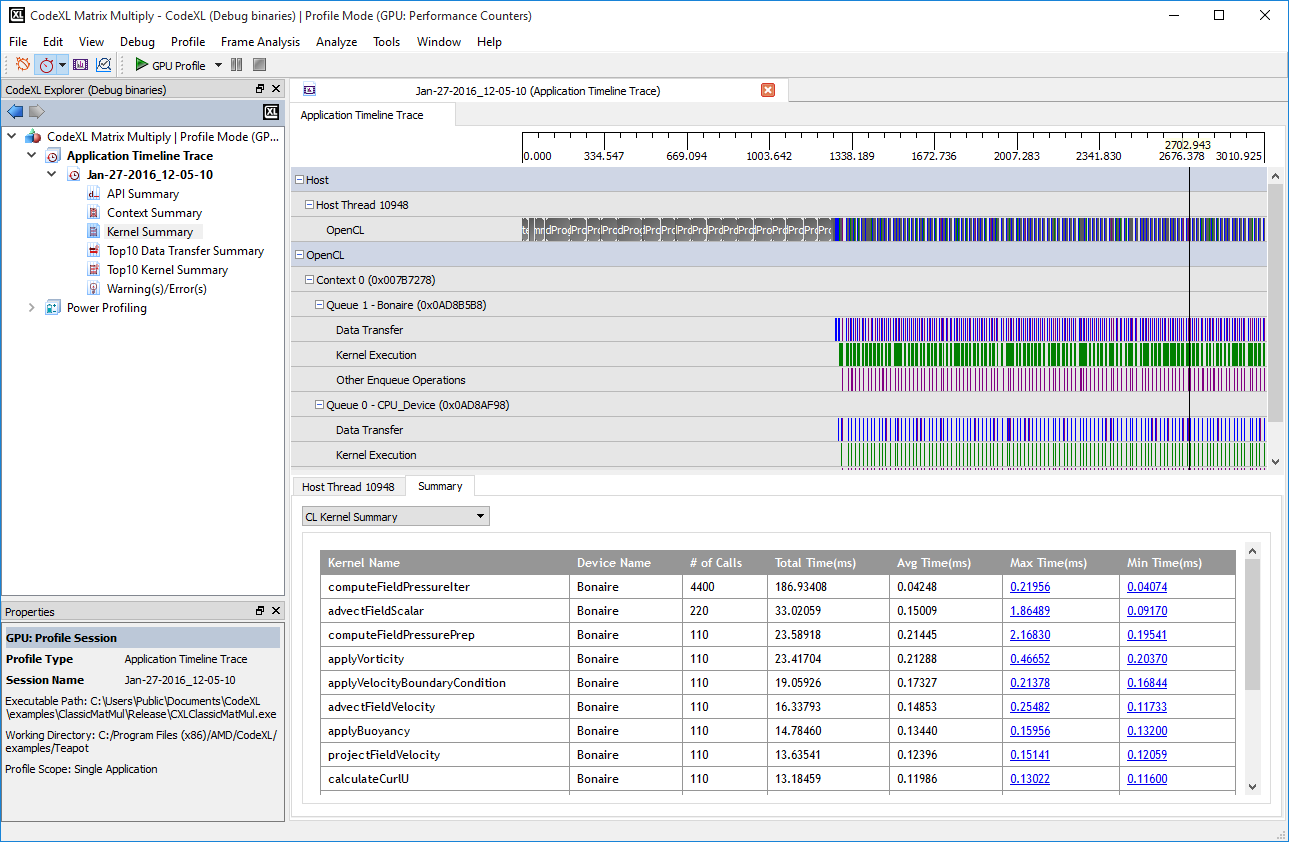
The GPU Profiler component in CodeXL is a performance analysis tool that gathers data from the API run-time and GPU for OpenCL™ or ROCm/HSA applications.

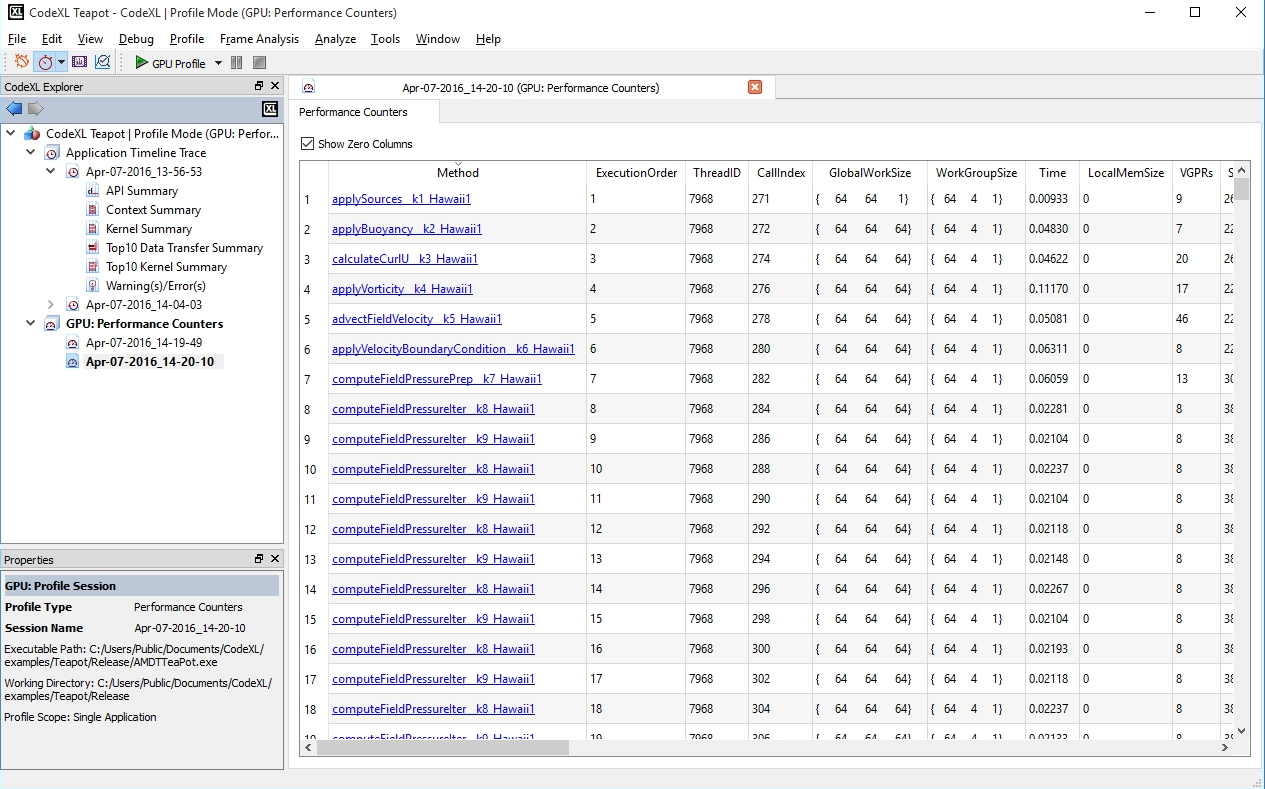
This information can be used by developers to discover bottlenecks in the application and find ways to optimize the application's performance. The GPU Profiler can also be used as a command-line tool.

Features of the GPU Profiler include:

* Measure the execution time of an OpenCL™ or ROCm/HSA 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.
* 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 an OpenCL™ or HSA application. It displays: 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 called by the host as well as data transfers (OpenCL™ only) and kernels executing on a device. 3. A set of Summary Pages, providing a set of statistics for the application, as well as the results of detailed analysis of the application. For more information, see: [**Application Timeline Trace Session**](#_topic_GPUProfilerProfilerApplicationTr) [**Application Timeline Trace Summary Pages**](#_topic_GPUProfilerSummaryPages) |
| Performance Counters | This mode collects performance counters from the AMD GPU or APU for each kernel dispatched to the device by either an OpenCL™ or ROCm/HSA application. It also displays statistics from the shader compiler for each kernel dispatched. The performance counters and statistics can be used to discover bottlenecks in a particular kernel. This mode also can also display the kernel source code, the generated IL code, and the compiled ISA code for an OpenCL™ kernel dispatched to a GPU. For more information, see: [**GPU Profiler Performance Counters Session**](#_topic_GPUProfilerPerformanceCountersSe) [**GPU Profiler Code Viewer**](#_topic_GPUProfilerCodeViewer). |

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**](#_topic_GPUProfilerKernelOccupancyViewer) and [**GPU Profiler Kernel Occupancy**](#_topic_GPUProfilerKernelOccupancy)

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**](#_topic_ExecutionToolbar) 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**](#_topic_CodeXLExplorer). 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**](#_topic_GPUProfilerProfilerApplicationTr)

[**GPU Profiler Performance Counters Session**](#_topic_GPUProfilerPerformanceCountersSe)

[**GPU Profiler Summary Pages**](#_topic_GPUProfilerSummaryPages)

[**GPU Profiler Code Viewer**](#_topic_GPUProfilerCodeViewer)

[**GPU Profiler Kernel Occupancy Viewer**](#_topic_GPUProfilerKernelOccupancyViewer)

[**GPU Profiler Kernel Occupancy**](#_topic_GPUProfilerKernelOccupancy)

[**GPU Profile Project Settings**](#_topic_GPUProfilingProjectSettings)

[**Description of Output Files**](#_topic_DescriptionofOutputFiles)

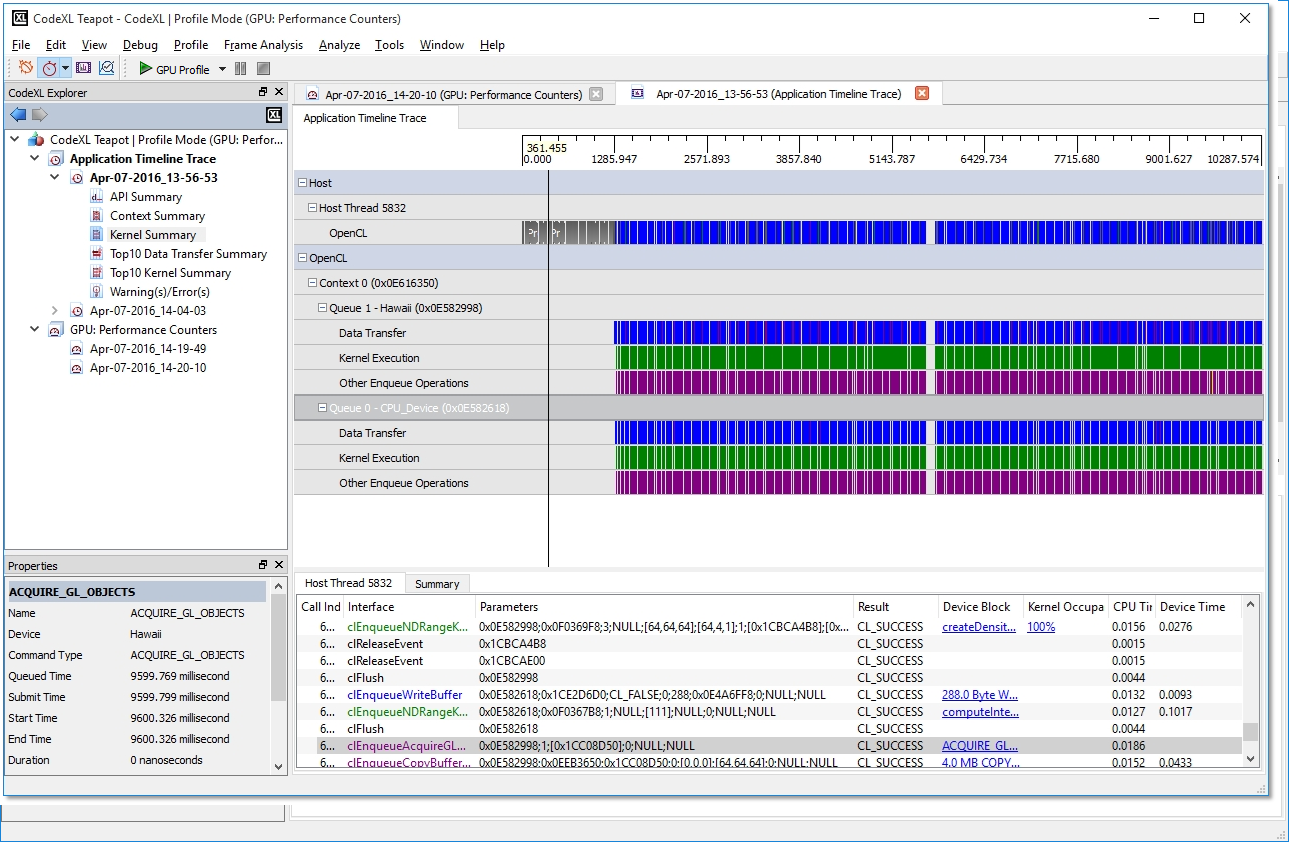
[**Description of Configuration Files**](#_topic_DescriptionofConfigurationFiles)

[**Using the Command-Line Interface**](#_topic_UsingtheCommandLineInterface)

**AMDTActivityLogger Library**

#### Application Timeline Trace Session

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**](#_topic_CodeXLExplorer) , and select "Open Containing Folder" from the menu. See [**Description of Output Files**](#_topic_DescriptionofOutputFiles) 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™ of 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**](#_topic_GPUProfilerKernelOccupancyViewer), which provides more information about the [**kernel occupancy**](#_topic_GPUProfilerKernelOccupancy).

If the option to Enable navigation to source code is checked on the [**Application Timeline Trace page**](#_topic_GPUProfileApplicationTracepage), 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

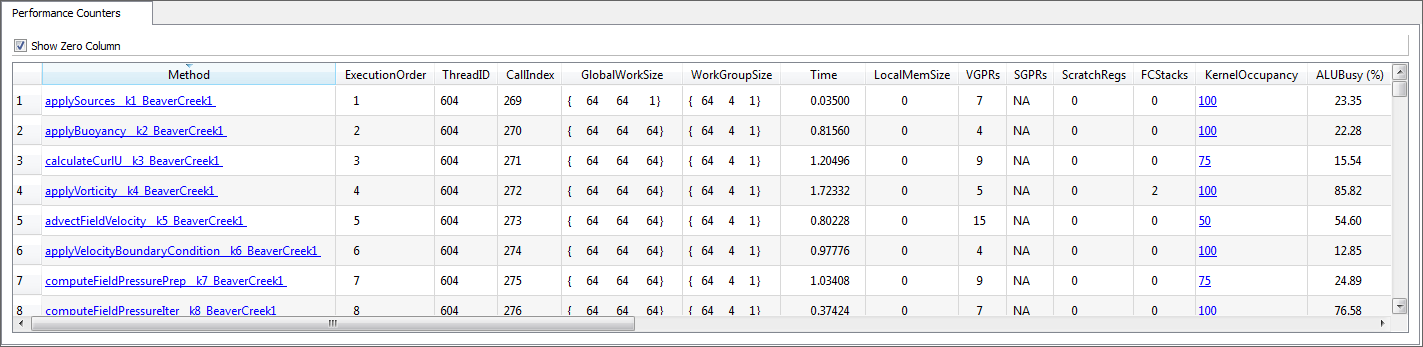
Colors are used in both the application timeline and the API Trace to help distinguish data transfer and kernel dispatch. Green is used on kernel dispatch items for both the OpenCL™ Enqueue calls from the host and the kernels themselves on the device. Shades of blue are used to color OpenCL™ data transfer 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**](#_topic_GPUProfileApplicationTracepage) (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.

#### GPU Profiler Performance Counters Session

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**](#_topic_CodeXLExplorer)](#_topic_CodeXLExplorer), and select "Open Containing Folder" from the menu. See [**Description of Output Files**](#_topic_DescriptionofOutputFiles) 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.

|  |  |
| --- | --- |
| **Name** | **Description** |
| Method | The kernel name (appended with \_\_k[KernelID]\_[DeviceName][DeviceID] to differentiate unique kernels with the same name). |
| ExecutionOrder | The order of execution for the kernel dispatch operations in the program. |
| ThreadID | The thread ID of the host thread that made the OpenCL™ API call that initiated the kernel dispatch operation. |
| CallIndex | The call index of the OpenCL™ API call that initiated the kernel dispatch operation. |
| GlobalWorkSize | The global work-item size of the kernel. |
| WorkGroupSize | The work-group size of the kernel. |
| Time | The time spent (in milliseconds) executing the kernel. This does not include the kernel set-up time. |
| LocalMemSize | The amount of local memory (LDS for GPU) in bytes being used by the kernel. |
| VGPRs | The number of general-purpose vector registers used by the kernel (valid only for GPU devices). |
| SGPRs | The number of general-purpose scalar registers used by the kernel (valid only for GPU devices). |
| ScratchRegs | The number of scratch registers used by the kernel (valid only for GPU devices). If non zero, this typically is the main bottleneck. To reduce this number, reduce the number of GPRs used by the kernel. |
| KernelOccupancy | The kernel occupancy (valid only for GPU devices). This is an estimate of 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 table gives a description of these columns for an HSA application.

|  |  |
| --- | --- |
| **Name** | **Description** |
| Method | The kernel name (appended with the Device Name). |
| ExecutionOrder | The order of execution for the kernel dispatch operations in the program. |
| ThreadID | The thread ID of the host thread that made the HSA API call that initiated the kernel dispatch operation. |
| GlobalWorkSize | The global work-item size of the kernel. |
| WorkGroupSize | The work-group size of the kernel. |
| LocalMemSize | The amount of local memory (LDS) in bytes being used by the kernel. |
| VGPRs | The number of general-purpose vector registers used by the kernel. |
| SGPRs | The number of general-purpose scalar registers used by the kernel. |
| KernelOccupancy | The kernel occupancy. This is an estimate of 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 [**GPU Profile: Performance Counters page**](#_topic_GPUProfilePerformanceCounterspag) of the [**GPU Profiling Project Settings**](#_topic_GPUProfilingProjectSettings) 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, click on the kernel name in the first column to open the [**GPU Profiler Code Viewer**](#_topic_GPUProfilerCodeViewer).

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**](#_topic_GPUProfilerKernelOccupancy) , 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**](#_topic_GPUProfilerKernelOccupancy) figure for an OpenCL™ kernel, click on the percentage value in the **Kernel Occupancy** column to open the [**GPU Profiler Kernel Occupancy Viewer**](#_topic_GPUProfilerKernelOccupancyViewer).

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

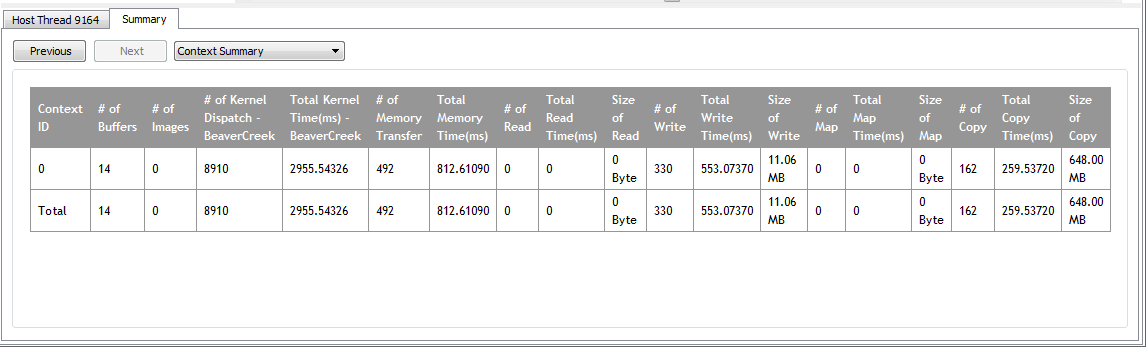
When collecting performance counters, it is strongly recommended that no other workloads (i.e. graphics workloads) are running on the GPU. Performance counters on AMD Radeon™ GPUs are global in nature, meaning that graphics workloads running on the GPU concurrently with a compute workload that is being 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.

#### Application Timeline Trace Summary Pages

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**](#_topic_GPUProfileApplicationTracepage). To generate summary pages from the command line, see [**Using the Command Line Interface**](#_topic_UsingtheCommandLineInterface). 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **API Name** | **# of Calls** | **Total Time(ms)** | **Ave Time(ms)** | **Max Time(ms)** | **Min Time(ms)** |
| clSetKernelArg | 60884 | 106.42239 | 0.00175 | 0.07284 | 0.00147 |
| clGetKernelInfo | 4332 | 8.29252 | 0.00191 | 0.02347 | 0.00147 |
| clEnqueueNDRangeKernel | 4332 | 178.09443 | 0.04111 | 0.19556 | 0.00831 |
| clReleaseMemObject | 4265 | 537.25810 | 0.12597 | 1.07458 | 0.00147 |
| clCreateBuffer | 4265 | 13.41322 | 0.00314 | 0.05622 | 0.00195 |
| clEnqueueReadBuffer | 938 | 2403.70320 | 2.56258 | 18.78409 | 0.60280 |
| clEnqueueWriteBuffer | 231 | 318.01591 | 1.37669 | 10.39231 | 0.78809 |
| clEnqueueWriteBufferRect | 63 | 1286.34994 | 20.41825 | 63.08036 | 13.78178 |
| clEnqueueReadBufferRect | 63 | 4288.33441 | 68.06880 | 134.18926 | 45.09560 |
| clReleaseKernel | 13 | 0.03764 | 0.00290 | 0.00440 | 0.00244 |
| clCreateKernel | 13 | 0.06600 | 0.00508 | 0.00880 | 0.00293 |
| clGetDeviceInfo | 11 | 0.03422 | 0.00311 | 0.00391 | 0.00244 |
| clGetPlatformInfo | 5 | 0.02396 | 0.00479 | 0.00635 | 0.00293 |
| clGetContextInfo | 4 | 0.02396 | 0.00599 | 0.01613 | 0.00196 |
| clReleaseContext | 2 | 0.00831 | 0.00415 | 0.00587 | 0.00244 |
| clCreateContextFromType | 2 | 0.06844 | 0.03422 | 0.04547 | 0.02298 |
| clReleaseProgram | 1 | 2.08413 | 2.08413 | 2.08413 | 2.08413 |
| clReleaseCommandQueue | 1 | 97.29672 | 97.29672 | 97.29672 | 97.29672 |
| clFinish | 1 | 0.06307 | 0.06307 | 0.06307 | 0.06307 |
| clCreateProgramWithBinary | 1 | 0.58276 | 0.58276 | 0.58276 | 0.58276 |
| clCreateCommandQueue | 1 | 227.11189 | 227.11189 | 227.11189 | 227.11189 |
| clBuildProgram | 1 | 1492.02943 | 1492.02943 | 1492.02943 | 1492.02943 |

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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Context ID** | **# of Buffers** | **# of Images** | **# of Kernel Dispatch - CPU\_Device** | **Total Kernel Time(ms) - CPU\_Device** | **# of Kernel Dispatch - Juniper** | **Total Kernel Time(ms) - Juniper** | **# of Memory Transfer** | **Total Memory Time(ms)** | **# of Read** | **Total Read Time(ms)** | **Size of Read** | **# of Write** | **Total Write Time(ms)** | **Size of Write** | **# of Map** | **Total Map Time(ms)** | **Size of Map** | **# of Copy** | **Total Copy Time(ms)** | **Size of Copy** |
| 0 | 2 | 0 | 1 | 69.07170 | 1 | 4.10271 | 2 | 1.08503 | 0 | 0 | 0 Byte | 2 | 1.08503 | 512.00 KB | 0 | 0 | 0 Byte | 0 | 0 | 0 Byte |
| 1 | 2 | 0 | 1 | 35.53000 | NA | NA | 1 | 0.21951 | 0 | 0 | 0 Byte | 1 | 0.21951 | 256.00 KB | 0 | 0 | 0 Byte | 0 | 0 | 0 Byte |
| 2 | 2 | 0 | NA | NA | 1 | 3.41856 | 1 | 0.68449 | 0 | 0 | 0 Byte | 1 | 0.68449 | 256.00 KB | 0 | 0 | 0 Byte | 0 | 0 | 0 Byte |
| 3 | 2 | 0 | 1 | 35.73143 | NA | NA | 1 | 0.17307 | 0 | 0 | 0 Byte | 1 | 0.17307 | 256.00 KB | 0 | 0 | 0 Byte | 0 | 0 | 0 Byte |
| 4 | 2 | 0 | NA | NA | 1 | 1.38896 | 1 | 1.45957 | 0 | 0 | 0 Byte | 1 | 1.45957 | 256.00 KB | 0 | 0 | 0 Byte | 0 | 0 | 0 Byte |
| Total | 10 | 0 | 3 | 140.33313 | 3 | 8.91023 | 6 | 3.62166 | 0 | 0 | 0 Byte | 6 | 3.62166 | 1.50 MB | 0 | 0 | 0 Byte | 0 | 0 | 0 Byte |

Kernel Summary Page

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Kernel Name** | **Device Name** | **# of Calls** | **Total Time(ms)** | **Avg Time(ms)** | **Max Time(ms)** | **Min Time(ms)** |
| multiDeviceKernel | CPU\_Device | 3 | 140.33313 | 46.77771 | 69.07170 | 35.53000 |
| multiDeviceKernel | Juniper | 3 | 8.91023 | 2.97008 | 4.10271 | 1.38896 |

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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Command Type** | **Context ID** | **Command Queue ID** | **Duration(ms)** | **Transfer Size** | **Transfer Rate(MB/s)** | **Thread ID** | **Sequence ID** |
| WRITE\_BUFFER | 4 | 5 | 1.45957 | 256.00 KB | 171.284 | 3496 | 108 |
| WRITE\_BUFFER | 0 | 1 | 0.94618 | 256.00 KB | 264.220 | 3496 | 16 |
| WRITE\_BUFFER | 2 | 3 | 0.68449 | 256.00 KB | 365.237 | 3496 | 56 |
| WRITE\_BUFFER | 1 | 2 | 0.21951 | 256.00 KB | 1138.895 | 3496 | 55 |
| WRITE\_BUFFER | 3 | 4 | 0.17307 | 256.00 KB | 1444.527 | 3496 | 107 |
| WRITE\_BUFFER | 0 | 0 | 0.13885 | 256.00 KB | 1800.569 | 3496 | 15 |

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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Kernel Name** | **Context ID** | **Command Queue ID** | **Device Name** | **Duration(ms)** | **Global Work Size** | **Work-Group Size** | **Thread ID** | **Sequence ID** |
| multiDeviceKernel | 0 | 0 | CPU\_Device | 69.07170 | {65536} | {64} | 3496 | 21 |
| multiDeviceKernel | 3 | 4 | CPU\_Device | 35.73143 | {65536} | {64} | 1968 | 0 |
| multiDeviceKernel | 1 | 2 | CPU\_Device | 35.53000 | {65536} | {64} | 3496 | 61 |
| multiDeviceKernel | 0 | 1 | Juniper | 4.10271 | {65536} | {64} | 3496 | 23 |
| multiDeviceKernel | 2 | 3 | Juniper | 3.41856 | {65536} | {64} | 3496 | 63 |
| multiDeviceKernel | 4 | 5 | Juniper | 1.38896 | {65536} | {64} | 2892 | 0 |

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.

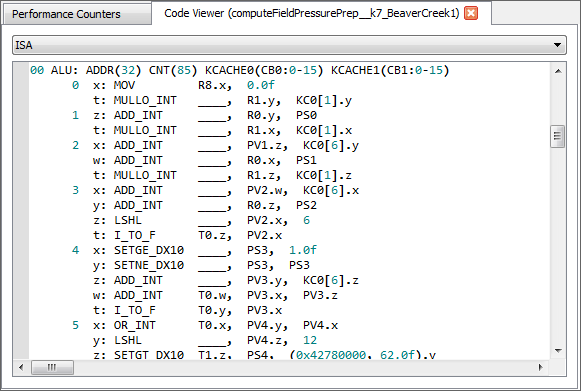
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **Call Index** | **Thread ID** | **Type** | **Message** |
| 0 | 542 | 2268 | Warning | Memory leak detected [Ref = 1, Handle = 0x0B1730B0]: Object created by clEnqueueNDRangeKernel |
| 216 | 208 | 2268 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1,1]. Recommended work-group size is a multiple of 64. |
| 270 | 319 | 2268 | Best Practices | clEnqueueNDRangeKernel: Global work size is too small - [111], resulting in low GPU utilization. |
| 144 | 482 | 1932 | Error | clEnqueueNDRangeKernel returns CL\_INVALID\_KERNEL\_ARGS. |

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).

#### GPU Profiler Code Viewer

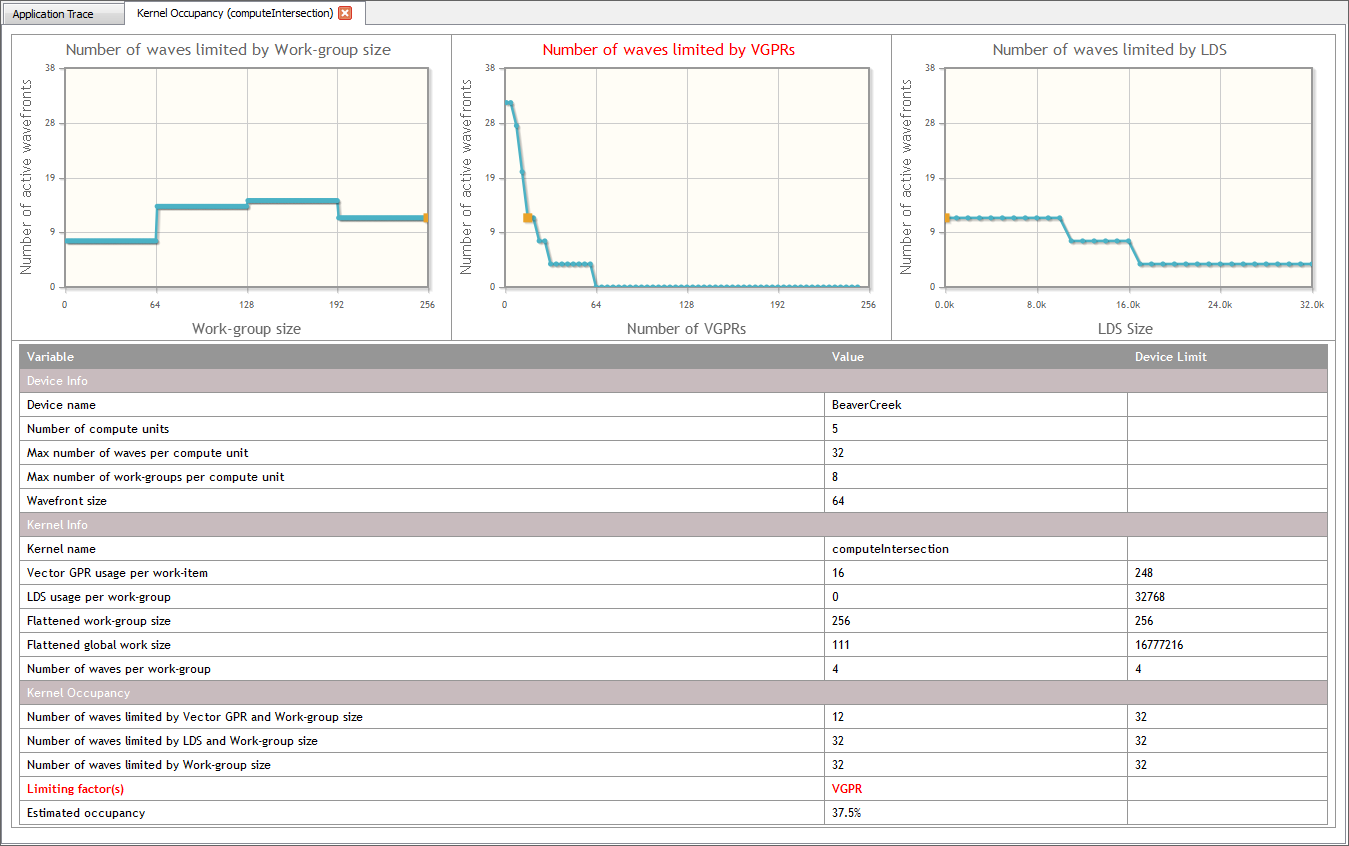
The Code Viewer appears when you click on the name of a kernel in the first column of the [**GPU Profiler Performance Counters Session**](#_topic_GPUProfilerPerformanceCountersSe) panel.

****

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 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

****

There are two ways to open the Kernel Occupancy panel:

* Click on the [**kernel occupancy percentage**](#_topic_GPUProfilerKernelOccupancy) in the Kernel Occupancy column of the [**GPU Profiler Performance Counters Session**](#_topic_GPUProfilerPerformanceCountersSe) panel (OpenCL™ and HSA).  
  or
* Click on the [**kernel occupancy percentage**](#_topic_GPUProfilerKernelOccupancy) in the Kernel Occupancy column of the API Trace in the [**GPU Profiler Profiler Application Trace Session**](#_topic_GPUProfilerProfilerApplicationTr) 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 Work-group 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**](#_topic_GPUProfilerKernelOccupancy) percentage.

#### GPU Profiler Kernel Occupancy

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:



where NWA is the number of in-flight wavefronts on the compute unit, and NWT 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 WGmax is the maximum number of work-groups on a compute unit, LDSCU is the shared memory available on the compute unit, and LDSwg 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:



where WFmax is the maximum number of wavefronts, WGmax is the maximum number of work-groups, and WFWG 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 WGmax=8 (denoted by WGmaxCU). 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:



1. 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:



where Nreg is the number of registers per work-item; the superscripts max and 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:



1. 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:



where WFmaxCU is the maximum number of wavefronts on the compute unit and WFWGmax 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:



Thus, the occupancy, O, is given by:



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, work-group 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:



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:



where WFWG is the number of wavefronts per work group.

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



Where VGPRmax is maximum number of registers per work-item and VGPRused 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.



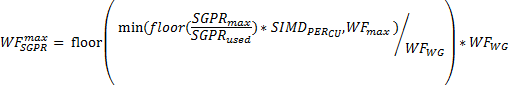
At the same time, the number of wavefronts cannot exceed WFmax, so



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



The third limit on the number of active wavefronts is the number of SGPR. Similar to VGPR, SGPR is calculated by



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



where WGmax is the maximum number of work-groups determined by the LDS. Then, the maximum number of wavefronts is given by:



Thus, the occupancy, O, is given by:



#### 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:

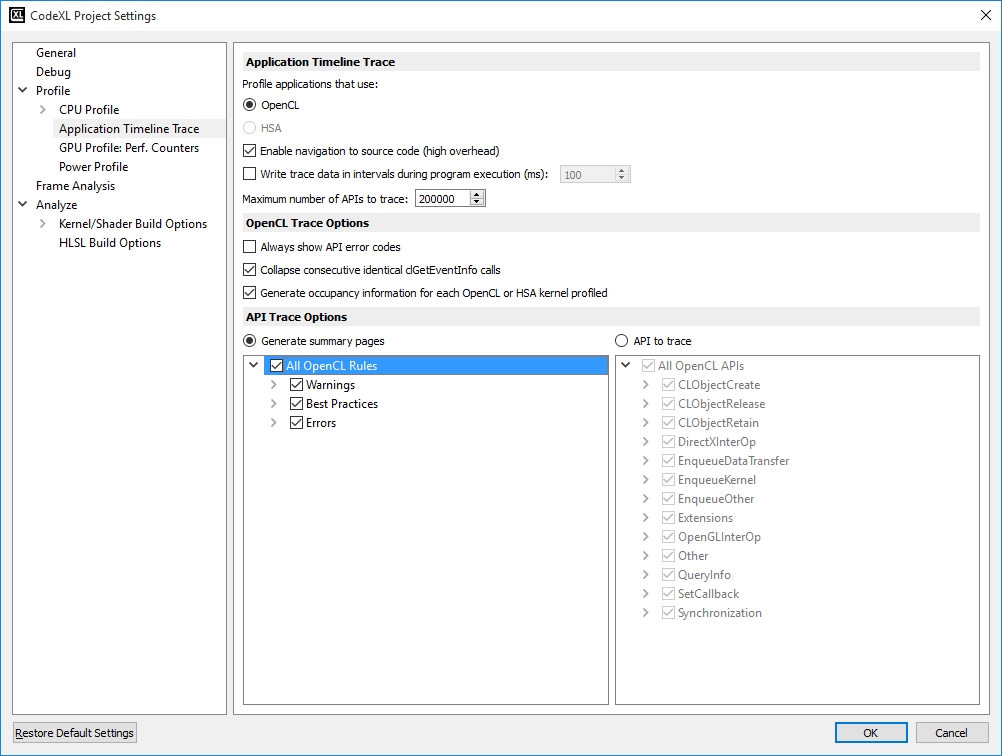
[**Application Timeline Trace page**](#_topic_GPUProfileApplicationTracepage)

[**GPU Profile: Performance Counters page**](#_topic_GPUProfilePerformanceCounterspag)

##### 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 that use | Select the API to trace. When one of the OpenCL / HSA radio buttons is clicked, the project setting pages will 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**](#_topic_GPUProfilerProfilerApplicationTr) 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 **clReleaseContext** OpenCL™ API is called. However, if an application does not call **clReleaseContext**, or if it calls any OpenCL™ APIs after the final **clReleaseContext** 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 specify the interval but does not let you enable or disable writing the data in intervals. |
| Maximum number of APIs to trace | 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 [**GPU Profiler Summary Pages**](#_topic_GPUProfilerSummaryPages) might not be correct, as a complete trace is required to provide a fully-accurate application summary. |
| Always show API error codes | 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. |
| Collapse consecutive identical clGetEventInfo calls | 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 clGetEventInfo 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**](#_topic_GPUProfilerKernelOccupancy) data for each OpenCL™ kernel dispatched to a GPU device. |

Generate summary pages

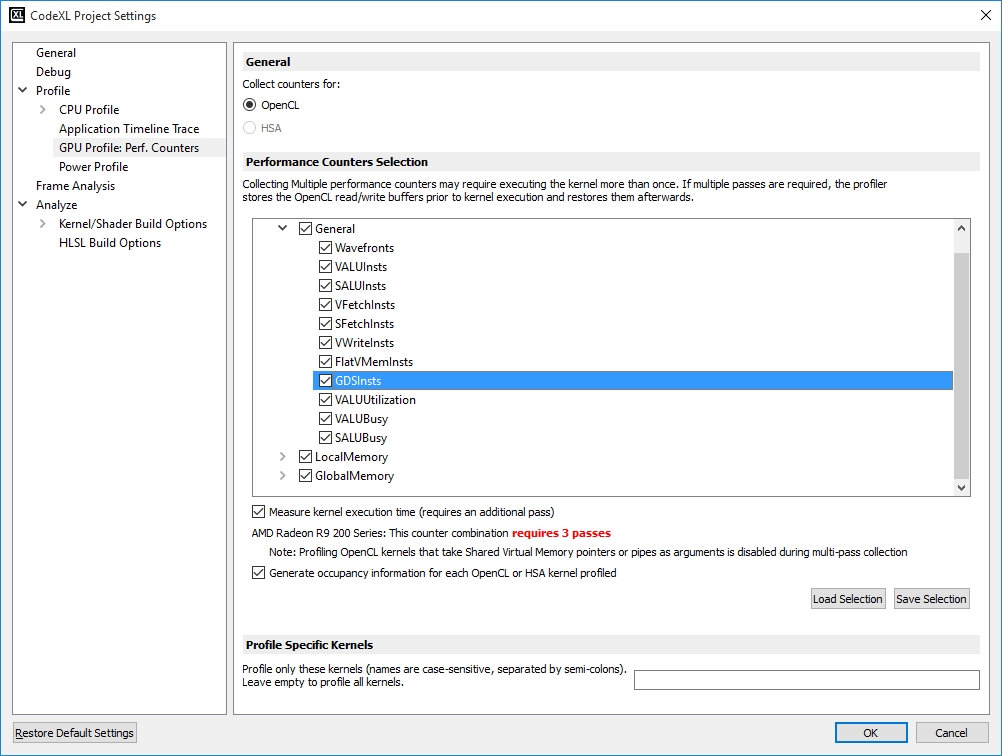
When checked, the Profiler automatically generates [**GPU Profiler Summary Pages**](#_topic_GPUProfilerSummaryPages) using the API trace and timeline data. You can further configure the summary pages by selecting rules to be used when generating the Warning(s)/Error(s) Summary page. The following table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

|  |  |  |
| --- | --- | --- |
| **Rule** | **Description** | **API** |
| Detect resource leaks | Tracks the reference count for all OpenCL™ or HSA objects, and reports any objects not released | OpenCL / HSA |
| Detect deprecated API calls | Detects calls to OpenCL™ API functions that have been deprecated in recent versions of OpenCL™ | OpenCL |
| Detect unnecessary blocking writes | Detects unnecessary blocking write operations | OpenCL |
| Detect non-optimized work size | Detects clEnqueueNDRangeKernel calls that specify a global or local workgroup size that is non-optimal for AMD Hardware | OpenCL |
| 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 | OpenCL |
| Detect redundant synchronization | Detects redundant synchronization that results in low host and device use | OpenCL |
| 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™ APIs might not be detected unless the **Always show API error codes** option is checked | OpenCL / HSA |

* **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.

##### GPU Profile: Performance Counters page

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**](#_topic_GPUProfilerKernelOccupancy) 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.

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

#### 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**](#_topic_GPUProfilerPerformanceCountersSe) 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**](#_topic_GPUProfileApplicationTracepage), 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**](#_topic_GPUProfilePerformanceCounterspag) 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

clGetSupportedImageFormats

clGetMemObjectInfo

clGetImageInfo

clGetSamplerInfo

clGetProgramInfo

clGetProgramBuildInfo

clGetKernelInfo

clGetKernelWorkGroupInfo

clGetEventInfo

clGetEventProfilingInfo

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.

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.

ThreadID=3364

CallIndex=101

KernelName=reduce

DeviceName=Capeverde

ComputeUnits=10

MaxWavesPerComputeUnit=40

MaxWorkGroupPerComputeUnit=16

MaxVGPRs=256

MaxSGPRs=512

MaxLDS=32768

UsedVGPRs=11

UsedSGPRs=20

UsedLDS=4096

WavefrontSize=64

WorkGroupSize=256

WavesPerWorkGroup=4

MaxWorkGroupSize=256

MaxWavesPerWorkGroup=4

GlobalWorkSize=256

MaxGlobalWorkSize=16777216

WavesLimitedByVGPR=40

WavesLimitedBySGPR=40

WavesLimitedByLDS=32

WavesLimitedByWorkgroup=40

Occupancy=80

#### 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**:

|  |  |
| --- | --- |
| **--startdisabled** | Start the application with profiling disabled. This is useful for applications that call amdtStopProfiling and amdtResumeProfiling from the AMDTActivityLogger library. |
| **-d [startdelay] arg** | Starts the application with profiling disabled and enables it after the provided delay in milliseconds |
| **-D [profileduration] arg** | Profiles the application for only given ‘arg’ milliseconds |
| **-e [ --envvar ] arg** | Environment variable that should be defined when running the profiled application. Argument should be in the format NAME=VALUE. |
| **-E [ --envvarfile ] arg** | Path to a file containing a list of environment variables that should be defined when running the profiled application. The file should contain one line for each variable in the format NAME=VALUE. |
| **-f [ --fullenv ]** | The environment variables specified with the envvar switch represent the full environment block. If not specified, then the environment variables represent additions or changes to the system environment block. |
| **-l [ --list ]** | Print the list of valid counter names. |
| **-L [listdetailed]** | Prints the list of valid counters along with its description. |
| **-N [ --sessionname ] arg** | Name of the generated session. If not specified, the name is the parent directory of the OutputFile. |
| **--maxpassperfile arg** | This option is used along with -- list and -- outputfile options to generate multiple counter files containing set of counters which can fit in the ‘arg’ pass. |
| **--numberofpass** | This option displays the number of passes required for default counter set or provided counter file with –counterfile option |
| **-o [ --outputfile ] arg** | Path to OutputFile. If not provided, the default is Session1.csv in an "CodeXL" directory under the current user's Documents directory; when performing an API trace, the default is apitrace.atp in the same location. For Linux, the default location is the current user’s home directory. |
| **--hsanokerneldemangle** | Disable the demangling of the kernel name in HSA trace or performance counter mode. (Linux only) |
| **-v [ --version ]** | Print the CodeXLGpuProfiler version number. |
| **-w [ --workingdirectory ] arg** | Set the working directory. If not provided, the default is the app binary's path. |
| **-h [ --help ]** | Show a help message. |

**Profile mode options**:

|  |  |
| --- | --- |
| **-t [ --apitrace ]** | Trace OpenCL™ application and generate CPU and GPU time stamps and detailed API call traces. |
| **-p [ --perfcounter ]** | Get the performance counters for each OpenCL™ kernel dispatched by the application. |
| **-A [ --hsatrace]** | Trace HSA application and generate CPU and GPU time stamps and detailed API call traces. (Linux only) |
| **-C [ --hsapmc]** | Get the performance counters for each HSA kernel dispatched by the application. (Linux only) |
| **--hsaaqlpackettrace** | Enable HSA AQL Packet tracing. This enhances the --hsatrace output by adding information about all AQL packets processed by the application. (Linux only) |
| **-O [ --occupancy ]** | Generate kernel occupancy information file (.occupancy). |
| **-P [ --occupancydisplay ] arg** | 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. |
| **-T [ --tracesummary ]** | Generate summary page from an input .atp file. |

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

|  |  |
| --- | --- |
| **-F [ --apifilterfile ]** | Path to the API filter file which contains a list of OpenCL™ or HSA APIs to be filtered out when performing an API trace. See below for information about the API filter file format. |
| **-i [ --interval ] arg (=100)** | Timeout interval in milliseconds. Ignored when not performing an API trace and using timeout mode. |
| **-m [ --timeout ]** | 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.) |
| **-M [ --maxapicalls ] (=1000000)** | Maximum number of API calls. |
| **-n [ --nocollapse ]** | Do not collapse consecutive identical clGetEventInfo calls into a single call in the trace output. Ignored when not performing an API trace. |
| **-r [ --ret ]** | 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. |
| **-y [ --sym ]** | Generate symbol information file (.st) for API trace, if available. Ignored when not performing an API trace. |

**Performance Counter mode options (for --perfcounter and --hsapmc):**

|  |  |
| --- | --- |
| **-c [ --counterfile ] arg** | 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. |
| **-g [ --singlepass ]** | 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™ only, this is the default for HSA). |
| **-G [ --nogputime ]** | Skip collection of GPUTime when profiling a kernel (GPUTime requires a separate pass) (OpenCL™ only, this is the default for HSA). |
| **-k [ --kerneloutput ] arg** | Output the specified kernel file (OpenCL™ 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  **all**: output all files |
| **-K [ --kernellistfile ] arg** | 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. |
| **-s [ --outputseparator ] arg** | Character used to separate fields in the OutputFile. Ignored when performing an API trace. |
| **-x [ --maxkernels] arg (=100000)** | Maximum number of kernels to profile. |
| **--xinitthreads** | Call XInitThreads at application startup. This can be a workaround for an assert that occurs when collecting performance counters . (Linux only) |

**Trace Summary mode options (for --tracesummary):**

|  |  |
| --- | --- |
| **-a [ --atpfile ] arg** | 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. |
| **-R [ --apirulesfile ] arg** | Path to OpenCL™ API analyzer configuration file. If not specified, all rules are enabled. Ignored when –tracesummary is not specified. See below for information about the configuration file format. |

Examples

* An example to collect OpenCL™ performance counters:

CodeXLGpuProfiler -o "/path/to/output.csv" -p -w "/path/to/app/working/directory" "/path/to/app.exe" --device gpu

* An example to collect an OpenCL™ API trace:

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:

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:

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:

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:

CodeXLGpuProfiler -a "/path/to/output.atp" -T

* An example to generate an occupancy display page:

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**](#_topic_CodeXLExplorer) .

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_DescriptionofConfigurationFiles) topic.

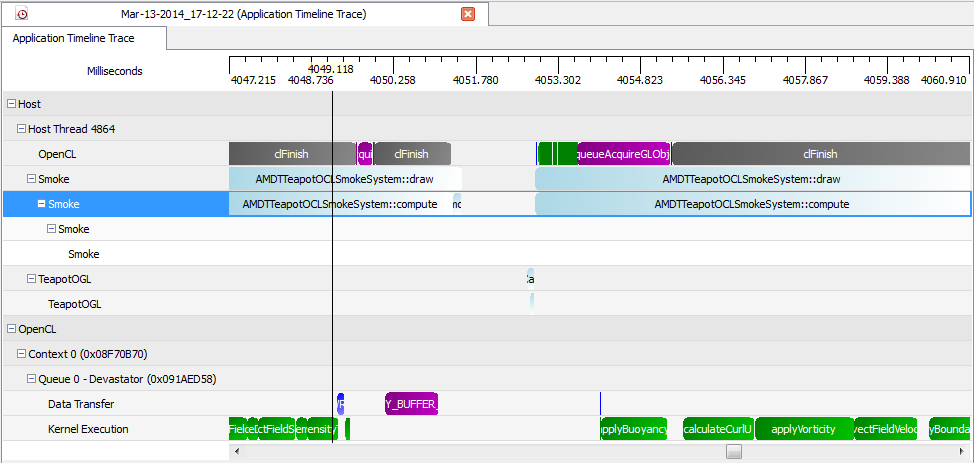
#### AMDTActivityLogger Library

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.