Rocprofiler SDK

Release 1.0.0

Advanced Micro Devices, Inc.

Oct 14, 2025

INSTALL

ROCprofiler-SDK is a tooling infrastructure for profiling general-purpose GPU compute applications running on the ROCm software. It supports application tracing to provide a big picture of the GPU application execution and kernel counter collection to provide low-level hardware details from the performance counters. The ROCprofiler-SDK library provides runtime-independent APIs for tracing runtime calls and asynchronous activities such as GPU kernel dispatches and memory moves. The tracing includes callback APIs for runtime API tracing and activity APIs for asynchronous activity records logging.

In summary, ROCprofiler-SDK combines ROCProfiler and ROCTracer. You can utilize the ROCprofiler-SDK to develop a tool for profiling and tracing HIP applications on ROCm software.

ROCprofiler-SDK uses a companion library called AQLprofile that generates profiling command packets (AQL/PM4) for performance counters and SQ thread trace. See the AQLprofile docs for more info.

The code is open and hosted at https://github.com/ROCm/rocprofiler-sdk.

The documentation is structured as follows:

Install

- *Installing ROCprofiler-SDK*

How to

- *Samples*
- *Using rocprofv3*
- *Using rocprofv3-avail*
- *Using rocpd Output Format*
- *Using ROCTx*
- *Using rocprofv3 with MPI*
- *Using rocprofv3 with OpenMP*
- *Using PC sampling*
- *Using thread trace*

API reference

- *Tool library*
- *Runtime intercept tables*
- *Process attachment*
- *Buffered services*
- *Callback services*
- *Counter collection services*
- *PC sampling*
- *ROCprof Trace Decoder*
- *ROCprofiler-SDK API*
- *ROCTx API*

Conceptual

- *Comparing ROCprofiler-SDK to other ROCm profiling tools*

To contribute to the documentation, refer to Contributing to ROCm.

You can find licensing information on the Licensing page.

CHAPTER

ONE

INSTALLING ROCPROFILER-SDK

This document provides information required to install ROCprofiler-SDK from source.

1.1 Supported systems

ROCprofiler-SDK is supported only on Linux. The following distributions are tested:

- Ubuntu 20.04
- Ubuntu 22.04
- Ubuntu 24.04
- OpenSUSE 15.5
- OpenSUSE 15.6
- Red Hat 8.8
- Red Hat 8.9
- Red Hat 8.10
- Red Hat 9.2
- Red Hat 9.3
- Red Hat 9.4

ROCprofiler-SDK might operate as expected on other Linux distributions, but has not been tested.

1.2 Identifying the operating system

To identify the Linux distribution and version, see the `/etc/os-release` and `/usr/lib/os-release` files:

```
$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.4 LTS (Focal Fossa)"
ID=ubuntu
...
VERSION_ID="20.04"
...
```

The relevant fields are `ID` and the `VERSION_ID`.

1.3 Build requirements

To build ROCprofiler-SDK, install `CMake` as explained in the following section.

1.3.1 Install CMake

Install CMake version 3.21 (or later).

> **Note**
>
> If the `CMake` installed on the system is too old, you can install a new version using various methods. One of the easiest options is to use PyPi (Python's pip).

```
/usr/local/bin/python -m pip install --user 'cmake==3.22.0'
export PATH=${HOME}/.local/bin:${PATH}
```

1.4 Building ROCprofiler-SDK from source

```
git clone https://github.com/ROCm/rocprofiler-sdk.git rocprofiler-sdk-source
cmake                                                \
    -B rocprofiler-sdk-build                         \
    -D ROCPROFILER_BUILD_TESTS=ON                    \
    -D ROCPROFILER_BUILD_SAMPLES=ON                  \
    -D CMAKE_INSTALL_PREFIX=/opt/rocm                \
    rocprofiler-sdk-source

cmake --build rocprofiler-sdk-build --target all --parallel 8
```

1.5 Installing ROCprofiler-SDK

To install ROCprofiler-SDK from the `rocprofiler-sdk-build` directory, run:

```
cmake --build rocprofiler-sdk-build --target install
```

1.6 Testing ROCprofiler-SDK

To run the built tests, `cd` into the `rocprofiler-sdk-build` directory and run:

```
ctest --output-on-failure -O ctest.all.log
```

> **Note**
>
> Running a few of these tests require you to install pandas and pytest first.

```
/usr/local/bin/python -m pip install -r requirements.txt
```

1.7 Install using package manager

If you have ROCm version 6.2 or later installed, you can use the package manager to install a prebuilt copy of ROCprofiler-SDK.

Ubuntu

```
$ sudo apt install rocprofiler-sdk
```

Red Hat Enterprise Linux

```
$ sudo dnf install rocprofiler-sdk
```

SUSE Linux Enterprise Server

```
$ sudo zypper install rocprofiler-sdk
```

CHAPTER

TWO

ROCPROFILER-SDK SAMPLES

The samples are provided to help you see the profiler in action.

2.1 Finding samples

The ROCm installation provides sample programs and `rocprofv3` tool.

- Sample programs are installed here:

```
/opt/rocm/share/rocprofiler-sdk/samples
```

- `rocprofv3` tool is installed here:

```
/opt/rocm/bin
```

2.2 Building Samples

To build samples from any directory, run:

```
cmake -B build-rocprofiler-sdk-samples /opt/rocm/share/rocprofiler-sdk/samples -DCMAKE_
↪PREFIX_PATH=/opt/rocm
cmake --build build-rocprofiler-sdk-samples --target all --parallel 8
```

2.3 Running samples

To run the built samples, `cd` into the `build-rocprofiler-sdk-samples` directory and run:

```
ctest -V
```

The *-V* option enables verbose output, providing detailed information about the test execution.

CHAPTER

THREE

USING ROCPROFV3

`rocprofv3` is a CLI tool that helps you optimize applications and analyze the low-level kernel details without requiring any modification in the source code. It's backward compatible with its predecessor, rocprof, and provides enhanced features for application profiling with better accuracy.

The following sections demonstrate the use of `rocprofv3` for application tracing and kernel counter collection using various command-line options.

`rocprofv3` is installed with ROCm under `/opt/rocm/bin`. To use the tool from anywhere in the system, export the `PATH` variable:

```
export PATH=$PATH:/opt/rocm/bin
```

Before tracing or profiling your HIP application using `rocprofv3`, build it using:

```
cmake -B <build-directory> <source-directory> -DCMAKE_PREFIX_PATH=/opt/rocm
cmake --build <build-directory> --target all --parallel <N>
```

3.1 Command-line options

The following table lists the commonly used `rocprofv3` command-line options categorized according to their purpose.

Table 3.1: rocprofv3 options

Purpose	Option	Description
I/O options		
	`-i` INPUT | `--input` INPUT	Specifies the path to the input file. JSON and YAML formats support configuration of all command-line options for tracing and profiling whereas the text format supports only the specification of HW counters.
	`-o` OUTPUT_FILE | `--output-file` OUTPUT_FILE	Specifies output file name. If nothing is specified, the default path is `%hostname%/%pid%`.
	`-d` OUTPUT_DIRECTORY | `--output-directory` OUTPUT_DIRECTORY	Specifies the output path for saving the output files. If nothing is specified, the default path is `%hostname%/%pid%`.
	`--output-format {csv,json,pftrace,otf2,rocpd} [{csv,json,pftrace,otf2,rocpd} ...]`	Specifies output format. Supported formats: CSV, JSON, PFTrace, OTF2 and rocpd.
	`--log-level {fatal,error,warning,info,trace,env}`	Sets the desired log level.
	`-E` EXTRA_COUNTERS | `--extra-counters` EXTRA_COUNTERS	Specifies the path to a YAML file consisting of extra counter definitions.
Process attachment		
	`-p` PID | `--pid` PID | `--attach` PID	Attaches to a running process by process ID and profiles it dynamically. This enables profiling of applications that are already running without needing to restart them from the profiler. The profiler will instrument the target process and collect the specified tracing or counter data for the configured duration.
Aggregate tracing		
	`-r` [BOOL] | `--runtime-trace` [BOOL]	Collects tracing data for HIP runtime API, marker (ROCTx) API, RCCL API, memory operations (copies, scratch, and allocation), and kernel dispatches. Similar to `--sys-trace` but without HIP compiler API and the underlying HSA API tracing.
	`-s` [BOOL] | `--sys-trace` [BOOL]	Collects tracing data for HIP API, HSA API, marker (ROCTx) API, RCCL API, memory operations (copies, scratch, and allocations), and kernel dispatches.
PC sampling		
	`--pc-sampling-beta-enabled` [BOOL]	Enables PC sampling and sets the ROCPROFILER_PC_SAMPLING_BETA_ENABLED

To see exhaustive list of `rocprofv3` options:

```
rocprofv3 --help
```

3.2 Application tracing

Application tracing provides the big picture of a program's execution by collecting data on the execution times of API calls and GPU commands, such as kernel execution, async memory copy, and barrier packets. This information can be used as the first step in the profiling process to answer important questions, such as how much percentage of time was spent on memory copy and which kernel took the longest time to execute.

To use `rocprofv3` for application tracing, run:

```
rocprofv3 <tracing_option> -- <application_path>
```

> **Note**
>
> All the tracing examples below use the `--output-format csv` option to generate output in CSV format. However, the default output format is `rocpd` (SQLite3 database). You can simply omit the `--output-format` option to generate output in the default format. `rocpd` format can be converted to other formats such as CSV, OTF2, and PFTrace using the `rocpd` module. To understand how to convert `rocpd` output to other formats, see *Using rocpd Output Format*.

3.2.1 HIP trace

HIP trace comprises execution traces for the entire application at the HIP level. This includes HIP API functions and their asynchronous activities at the runtime level. In general, HIP APIs directly interact with the user program. It is easier to analyze HIP traces as you can directly map them to the program. Unlike previous iterations of `rocprof`, this does not enable kernel tracing, memory copy tracing, and so on. If you want to enable kernel tracing, memory copy tracing, they need to be provided explicitly.

To trace HIP runtime APIs, use:

```
rocprofv3 --hip-trace --output-format csv -- <application_path>
```

The preceding command generates a `hip_api_trace.csv` file prefixed with the process ID.

```
$ cat 238_hip_api_trace.csv
```

Here are the contents of `hip_api_trace.csv` file:

Table 3.2: HIP api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HIP_COMI	__hipRegisterFatBinary	15	15	1	1055015439953054	1055015439976484
HIP_COMI	__hipRegisterFunction	15	15	2	1055015439992584	1055015440011104
HIP_COMI	__hipRegisterFunction	15	15	3	1055015440011744	1055015440013824
HIP_COMI	__hipRegisterFunction	15	15	4	1055015440014244	1055015440014534
HIP_COMI	__hipRegisterFunction	15	15	5	1055015440014854	1055015440015524
HIP_RUNT	hipGetDeviceCount	15	15	6	1055015440617618	1055015539800733
HIP_RUNT	hipSetDevice	15	15	7	1055015539819503	1055015539821693
HIP_RUNT	hipDeviceSynchronize	15	15	8	1055015539832333	1055015539840903
HIP_RUNT	hipStreamCreateWithFlags	15	15	9	1055015539861673	1055015865247140
HIP_RUNT	hipHostMalloc	15	15	10	1055015865309761	1055015865849494
HIP_RUNT	hipHostMalloc	15	15	11	1055015865850944	1055015866265546
HIP_RUNT	hipHostMalloc	15	15	12	1055015866266646	1055015867082900
HIP_RUNT	hipMallocAsync	15	15	13	1055015867356542	1055015867662314
HIP_RUNT	hipMallocAsync	15	15	14	1055015867664174	1055015867937465
HIP_RUNT	hipMallocAsync	15	15	15	1055015867938815	1055015868219987
HIP_RUNT	hipMemcpyAsync	15	15	16	1055015868240137	1055015917307652
HIP_RUNT	hipMemcpyAsync	15	15	17	1055015917337263	1055015917360493

`rocprofv3` provides options to collect traces at more granular level. For HIP, you can collect traces for HIP compile time APIs and runtime APIs separately.

To collect HIP compile time API traces, use:

```
rocprofv3 --hip-compiler-trace --output-format csv -- <application_path>
```

The preceding command generates a `hip_api_trace.csv` file prefixed with the process ID.

```
$ cat 208_hip_api_trace.csv
```

Here are the contents of `hip_api_trace.csv` file:

Table 3.3: HIP compile time api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HIP_COMI	__hipRegisterFatBinary	15	15	1	1055015439953054	1055015439976484
HIP_COMI	__hipRegisterFunction	15	15	2	1055015439992584	1055015440011104
HIP_COMI	__hipRegisterFunction	15	15	3	1055015440011744	1055015440013824
HIP_COMI	__hipRegisterFunction	15	15	4	1055015440014244	1055015440014534
HIP_COMI	__hipRegisterFunction	15	15	5	1055015440014854	1055015440015524

To collect HIP runtime time API traces, use:

```
rocprofv3 --hip-runtime-trace --output-format csv -- <application_path>
```

The preceding command generates a `hip_api_trace.csv` file prefixed with the process ID.

```
$ cat 208_hip_api_trace.csv
```

Here are the contents of `hip_api_trace.csv` file:

Table 3.4: HIP runtime api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HIP_RUNT	hipGetDevicePropertiesR0600	238	238	1	1191915574691984	1191915687784011
HIP_RUNT	hipMalloc	238	238	2	1191915691312459	1191915691388696
HIP_RUNT	hipMalloc	238	238	3	1191915691390637	1191915691423279
HIP_RUNT	hipMemcpy	238	238	4	1191915691439107	1191916547828448
HIP_RUNT	hipLaunchKernel	238	238	5	1191916547842972	1191916548408842
HIP_RUNT	hipMemcpy	238	238	6	1191916548412677	1191916550217834
HIP_RUNT	hipFree	238	238	7	1191916562618151	1191916562789093
HIP_RUNT	hipFree	238	238	8	1191916562790923	1191916562836351

For the description of the fields in the output file, see *Output file fields*.

3.2.2 HSA trace

The HIP runtime library is implemented with the low-level HSA runtime. HSA API tracing is more suited for advanced users who want to understand the application behavior at the lower level. In general, tracing at the HIP level is recommended for most users. You should use HSA trace only if you are familiar with HSA runtime.

HSA trace contains the start and end time of HSA runtime API calls and their asynchronous activities.

```
rocprofv3 --hsa-trace --output-format csv -- <application_path>
```

The preceding command generates a `hsa_api_trace.csv` file prefixed with process ID. Note that the contents of this file have been truncated for demonstration purposes.

```
$ cat 197_hsa_api_trace.csv
```

Here are the contents of `hsa_api_trace.csv` file:

Table 3.5: HSA api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HSA_COR	hsa_system	197	197	1	1507843974724237	1507843974724947
HSA_COR	hsa_agent_	197	197	3	1507843974754471	1507843974755014
HSA_AMD	hsa_amd_m	197	197	5	1507843974761705	1507843974762398
HSA_AMD	hsa_amd_m	197	197	6	1507843974763901	1507843974764030
HSA_AMD	hsa_amd_m	197	197	7	1507843974765121	1507843974765224
HSA_AMD	hsa_amd_m	197	197	8	1507843974766196	1507843974766328
HSA_AMD	hsa_amd_m	197	197	9	1507843974767534	1507843974767641
HSA_AMD	hsa_amd_m	197	197	10	1507843974768639	1507843974768779
HSA_AMD	hsa_amd_a	197	197	4	1507843974758768	1507843974769238
HSA_COR	hsa_agent_	197	197	11	1507843974771091	1507843974771537

`rocprofv3` provides options to collect HSA traces at more granular level. HSA traces can be collected separately for four API domains: `HSA_AMD_EXT_API`, `HSA_CORE_API`, `HSA_IMAGE_EXT_API` and `HSA_FINALIZE_EXT_API`.

To collect HSA core API traces, use:

```
rocprofv3 --hsa-core-trace --output-format csv -- <application_path>
```

The preceding command generates a `hsa_api_trace.csv` file prefixed with process ID. Note that the contents of this file have been truncated for demonstration purposes.

```
$ cat 197_hsa_api_trace.csv
```

Here are the contents of `hsa_api_trace.csv` file:

Table 3.6: HSA core api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HSA_COR	hsa_system	57	57	1	1056813747808832	1056813747809252
HSA_COR	hsa_agent_	57	57	3	1056813747826572	1056813747826672
HSA_COR	hsa_agent_	57	57	4	1056813747837582	1056813747837622
HSA_COR	hsa_agent_	57	57	5	1056813747838542	1056813747838582
HSA_COR	hsa_agent_	57	57	6	1056813747839042	1056813747839082
HSA_COR	hsa_agent_	57	57	7	1056813747839512	1056813747839622
HSA_COR	hsa_iterate_	57	57	2	1056813747821012	1056813747839832
HSA_COR	hsa_agent_	57	57	8	1056813747843832	1056813747844132
HSA_COR	hsa_agent_	57	57	9	1056813747844482	1056813747844542
HSA_COR	hsa_agent_i	57	57	10	1056813747849402	1056813747850422
HSA_COR	hsa_isa_get	57	57	11	1056813747853542	1056813747875253
HSA_COR	hsa_isa_get	57	57	12	1056813747875883	1056813747878353
HSA_COR	hsa_agent_	57	57	13	1056813747886343	1056813747886403
HSA_COR	hsa_agent_	57	57	54	1056813748282015	1056813748282085
HSA_COR	hsa_system	57	57	55	1056813748282465	1056813748282505
HSA_COR	hsa_signal_	57	57	56	1056813749083419	1056813749085399
HSA_COR	hsa_agent_	57	57	57	1056813749741363	1056813749741443
HSA_COR	hsa_queue_	57	57	58	1056813749744053	1056813856914188
HSA_COR	hsa_signal_	57	57	59	1056813857149169	1056813857154109
HSA_COR	hsa_signal_	57	57	60	1056813857154929	1056813857155389
HSA_COR	hsa_signal_	57	57	61	1056813857155949	1056813857156429
HSA_COR	hsa_signal_	57	57	62	1056813857157169	1056813857157349
HSA_COR	hsa_executa	57	57	63	1056813965439362	1056813965466952
HSA_COR	hsa_code_o	57	57	64	1056813965476642	1056813965587493
HSA_COR	hsa_executa	57	57	65	1056813965592483	1056813965965295
HSA_COR	hsa_signal_	57	57	67	1056813966149786	1056813966151706
HSA_COR	hsa_signal_	57	57	68	1056813966156596	1056813966158646
HSA_COR	hsa_signal_	57	57	69	1056813966162276	1056813966163746
HSA_COR	hsa_executa	57	57	66	1056813965973105	1056813966778050
HSA_COR	hsa_executa	57	57	70	1056813966800070	1056813966801880
HSA_COR	hsa_executa	57	57	71	1056813966805750	1056813966805980
HSA_COR	hsa_executa	57	57	72	1056813966806300	1056813966806340

For the description of the fields in the output file, see *Output file fields*.

3.2.3 Marker trace

> **Note**
>
> To use `rocprofv3` for marker tracing, including and linking to old `ROCTx` works but it's recommended to switch to the new `ROCTx` to utilize new APIs. To use the new `ROCTx`, include header `"rocprofiler-sdk-roctx/roctx.h"` and link your application with `librocprofiler-sdk-roctx.so`. To see the complete list of `ROCTx` APIs, see public header file `"rocprofiler-sdk-roctx/roctx.h"`.
>
> To see usage of `ROCTx` or marker library, see *Using ROCTx*.

3.2.4 Kokkos trace

Kokkos is a C++ library for writing performance portable applications. Kokkos is widely used in scientific applications to write performance-portable code for CPUs, GPUs, and other accelerators. `rocprofv3` loads an inbuilt Kokkos Tools library, which emits roctx ranges with the labels passed using Kokkos APIs. For example, `Kokkos::parallel_for("MyParallelForLabel", ...)` calls `roctxRangePush` internally and enables the kernel renaming option to replace the highly templated kernel names with the Kokkos labels. To enable the inbuilt marker support, use the `kokkos-trace` option. Internally, this option automatically enables `marker-trace` and `kernel-rename`:

```
rocprofv3 --kokkos-trace --output-format csv -- <application_path>
```

The preceding command generates a `marker-trace` file prefixed with the process ID.

```
 $ cat 210_marker_api_trace.csv
"Domain","Function","Process_Id","Thread_Id","Correlation_Id","Start_Timestamp","End_
↪Timestamp"
"MARKER_CORE_API","Kokkos::Initialization Complete",4069256,4069256,1,56728499773965,
↪56728499773965
"MARKER_CORE_API","Kokkos::Impl::CombinedFunctorReducer<CountFunctor,␣
↪Kokkos::Impl::FunctorAnalysis<Kokkos::Impl::FunctorPatternInterface::REDUCE,␣
↪Kokkos::RangePolicy<Kokkos::Serial>, CountFunctor, long int>::Reducer, void>",4069256,
↪4069256,2,56728501756088,56728501764241
"MARKER_CORE_API","Kokkos::parallel_reduce: fence due to result being value, not view",
↪4069256,4069256,4,56728501767957,56728501769600
"MARKER_CORE_API","Kokkos::Finalization Complete",4069256,4069256,6,56728502054554,
↪56728502054554
```

3.2.5 Kernel trace

To trace kernel dispatch traces, use:

```
rocprofv3 --kernel-trace --output-format csv -- <application_path>
```

The preceding command generates a `kernel_trace.csv` file prefixed with the process ID.

```
$ cat 199_kernel_trace.csv
```

Here are the contents of `kernel_trace.csv` file:

Table 3.7: Kernel trace

Kin	Ag	Qu	Str	Thr	Dis pat	Ker nel	Ker nel	Co re- la- tior	Start_T	End_Ti	LD	Scr	VG	Ac- cur	SG	Wo gro	Wo gro	Wo gro	Gri	Gri	Gri	deSiz	Size_Z
KE NE	Ag 4	1	1	834	1	10	voi ad- di- tior floa con floa con int, int)	1	1550151	1550151	0	0	8	0	16	64	1	1	102	102	1		
KE NE	Ag 4	1	1	834	4	11	di- vid floa con floa con int, int)	4	1550151	1550151	0	0	12	4	16	64	1	1	102	102	1		
KE NE	Ag 4	1	1	834	3	12	mu ti- ply. floa con floa con int, int)	3	1550151	1550151	0	0	8	0	16	64	1	1	102	102	1		
KE NE	Ag 4	1	1	834	2	13	sub trac floa con floa con int, int)	2	1550151	1550151	0	0	8	0	16	64	1	1	102	102	1		
KE NE	Ag 4	2	2	834	5	10	voi ad- di- tior floa con floa con int, int)	5	1550151	1550151	0	0	8	0	16	64	1	1	102	102	1		
KE NE	Ag 4	2	2	834	6	13	sub trac floa con floa con int, int)	6	1550151	1550151	0	0	8	0	16	64	1	1	102	102	1		

For the description of the fields in the output file, see *Output file fields*.

3.2.6 Memory copy trace

Memory copy traces track `hipMemcpy` and `hipMemcpyAsync` functions, which use the `hsa_amd_memory_async_copy_on_engine` HSA functions internally. To trace memory moves across the application, use:

```
rocprofv3 --memory-copy-trace --output-format csv -- <application_path>
```

The preceding command generates a `memory_copy_trace.csv` file prefixed with the process ID.

```
$ cat 197_memory_copy_trace.csv
```

Here are the contents of `memory_copy_trace.csv` file:

Table 3.8: Memory copy trace

Kind	Direc-tion	Stream_I	Source_A	Des-tina-tion_Age	Cor-rela-tion_Id	Start_Timestamp	End_Timestamp
MEM-ORY_CO	MEM-ORY_CO	0	Agent 0	Agent 4	1	1057963336487172	1057963336564212
MEM-ORY_CO	MEM-ORY_CO	0	Agent 0	Agent 4	2	1057963336783973	1057963336859334
MEM-ORY_CO	MEM-ORY_CO	0	Agent 4	Agent 0	23	1057963497396292	1057963497471732
MEM-ORY_CO	MEM-ORY_CO	0	Agent 4	Agent 0	24	1057963498099125	1057963498200446

For the description of the fields in the output file, see *Output file fields*.

3.2.7 Memory allocation trace

Memory allocation traces track the HSA functions `hsa_memory_allocate`, `hsa_amd_memory_pool_allocate`, and `hsa_amd_vmem_handle_create`. The function `hipMalloc` calls these underlying HSA functions allowing memory allocations to be tracked.

In addition to the HSA memory allocation functions listed above, the corresponding HSA free functions `hsa_memory_free`, `hsa_amd_memory_pool_free`, and `hsa_amd_vmem_handle_release` are also tracked. Unlike the allocation functions, however, only the address of the freed memory is recorded. As such, the agent id and size of the freed memory are recorded as 0 in the CSV and JSON outputs. It should be noted that it is possible for some free functions to records a null pointer address of 0x0. This situation can occur when some HIP functions such as hipStreamDestroy call underlying HSA free functions with null pointers, even if the user never explicitly calls free memory functions with null pointer addresses.

To trace memory allocations during the application run, use:

```
rocprofv3 --memory-allocation-trace --output-format csv -- <application_path>
```

The preceding command generates a `memory_allocation_trace.csv` file prefixed with the process ID.

```
$ cat 6489_memory_allocation_trace.csv
```

Here are the contents of `memory_allocation_trace.csv` file:

Table 3.9: Memory allocation trace

Kind	Operation	Agent_Id	Allocation_Size	Address	Correlation_Id	Start_Timestamp	End_Timestamp
MEMORY_ALI	MEMORY_ALI	Agent 0	1024	0x7fb2d0(	11	3721742710532634	3721742710584854
MEMORY_ALI	MEMORY_ALI	Agent 0	0	0x7fb2d0(	12	3721742710596404	3721742710933366
MEMORY_ALI	MEMORY_ALI	Agent 0	1024	0x7fb2d0(	13	3721742710941416	3721742710960916
MEMORY_ALI	MEMORY_ALI	Agent 0	0	0x7fb2d0(	14	3721742710967236	3721742711197647
MEMORY_ALI	MEMORY_ALI	Agent 0	1024	0x7fb2d0(	15	3721742711204077	3721742711219717
MEMORY_ALI	MEMORY_ALI	Agent 0	0	0x7fb2d0(	16	3721742711225857	3721742711466018

For the description of the fields in the output file, see *Output file fields*.

3.2.8 Runtime trace

This is a shorthand option that targets the most relevant tracing options for a standard user by excluding traces for HSA runtime API and HIP compiler API.

The HSA runtime API is excluded because it is a lower-level API upon which HIP and OpenMP target are built and thus, tends to be an implementation detail irrelevant to most users. Similarly, the HIP compiler API is also excluded for being an implementation detail as these functions are automatically inserted during HIP compilation.

`--runtime-trace` traces the HIP runtime API, marker API, kernel dispatches, and memory operations (copies, allocations, and scratch).

```
rocprofv3 --runtime-trace --output-format csv -- <application_path>
```

Running the preceding command generates `hip_api_trace.csv`, `kernel_trace.csv`, `memory_copy_trace.csv`, `scratch_memory_trace.csv`, `memory_allocation_trace.csv`, and `marker_api_trace.csv` (if `ROCTx` APIs are specified in the application) files prefixed with the process ID.

3.2.9 System trace

This is an all-inclusive option to collect HIP, HSA, kernel, memory copy, memory allocation, and marker trace (if `ROCTx` APIs are specified in the application).

```
rocprofv3 --sys-trace --output-format csv -- <application_path>
```

Running the preceding command generates `hip_api_trace.csv`, `hsa_api_trace.csv`, `kernel_trace.csv`, `memory_copy_trace.csv`, `scratch_memory_trace.csv`, `memory_allocation_trace.csv`, and `marker_api_trace.csv` if `ROCTx` APIs are specified in the application.

3.2.10 Scratch memory trace

This option collects scratch memory operation traces. Scratch is an address space on AMD GPUs roughly equivalent to the local memory in NVIDIA CUDA. The local memory in CUDA is a thread-local global memory with interleaved addressing, which is used for register spills or stack space. This option helps to trace when the `rocr` runtime allocates, frees, and tries to reclaim scratch memory.

To trace scratch memory allocations during the application run, use:

```
rocprofv3 --scratch-memory-trace --output-format csv -- <application_path>
```

The preceding command generates a `scratch_memory_trace.csv` file prefixed with the process ID.

```
$ cat 100_scratch_memory_trace.csv
```

Here are the contents of `scratch_memory_trace.csv` file:

Table 3.10: Scratch memory trace

Kind	Operation	Agent_Id	Queue_Id	Thread_Id	Alloc_Flags	Start_Timestamp	End_Timestamp
SCRATCH	SCRATCH	Agent 4	1	113	0	1124926523146168	1124926554133606
SCRATCH	SCRATCH	Agent 4	1	113	0	1124926554522025	1124927132642186

For the description of the fields in the output file, see *Output file fields*.

3.2.11 RCCL trace

RCCL (pronounced "Rickle") is a stand-alone library of standard collective communication routines for GPUs. This option traces those communication routines.

```
rocprofv3 --rccl-trace --output-format csv -- <application_path>
```

The preceding command generates a `rccl_api_trace` file prefixed with the process ID.

```
$ cat 197_rccl_api_trace.csv
```

Here are the contents of `rccl_api_trace.csv` file:

Table 3.11: RCCL trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
RCCL_API	ncclGetVersion	1834151	1834151	416	18413845573432	18413845577374
RCCL_API	ncclGetUniqueId	1834151	1834151	1116	18413961300878	18413963267869
RCCL_API	ncclGetUniqueId	1834151	1834151	1481	18414166449182	18414166720831
RCCL_API	ncclGroupStart	1834151	1834151	1482	18414166723772	18414166726834
RCCL_API	ncclGroupEnd	1834151	1834151	1490	18414166823575	18414380520973
RCCL_API	ncclCommInitAll	1834151	1834151	1477	18414166402665	18414380522536
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89098	18414380660695	18414380661652
RCCL_API	ncclAllReduce	1834151	1834151	89097	18414380653860	18414380693574
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89108	18414380694631	18414380694659
RCCL_API	ncclAllReduce	1834151	1834151	89107	18414380694212	18414380704722
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89117	18414380706650	18414380706677
RCCL_API	ncclAllReduce	1834151	1834151	89116	18414380705574	18414380715055
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89126	18414380715749	18414380715774
RCCL_API	ncclAllReduce	1834151	1834151	89125	18414380715463	18414380723944
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89135	18414380724688	18414380724715
RCCL_API	ncclAllReduce	1834151	1834151	89134	18414380724395	18414380732209
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89154	18414380746383	18414380746411
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89157	18414380749863	18414380749889
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89160	18414380751671	18414380751696
RCCL_API	ncclCommGetAsyncError	1834151	1834151	89163	18414380753326	18414380753353

3.2.12 rocDecode trace

rocDecode is a high-performance video decode SDK for AMD GPUs. This option traces the rocDecode API.

```
rocprofv3 --rocdecode-trace --output-format csv -- <application_path>
```

The above command generates a `rocdecode_api_trace` file prefixed with the process ID.

```
$ cat 41688_rocdecode_api_trace.csv
```

Here are the contents of `rocdecode_api_trace.csv` file:

Table 3.12: rocDecode trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
ROCDECODE_API	rocDecCreateVideoParser	41688	41688	583	615449881677279	615449882001583
ROCDECODE_API	rocDecGetDecoderCaps	41688	41688	584	615449882016054	615449882163756
ROCDECODE_API	rocDecGetDecoderCaps	41688	41688	588	615449886038750	615449886050880
ROCDECODE_API	rocDecCreateDecoder	41688	41688	591	615449886084210	615450756910310
ROCDECODE_API	rocDecDecodeFrame	41688	41688	595	615450757036042	615450767147413
ROCDECODE_API	rocDecGetDecodeStatus	41688	41688	812	615450836779385	615450836779575

Perfetto will also show rocDecode API arguments. Pointers will not be dereferenced and only the address will be displayed.

3.2.13 rocJPEG trace

rocJPEG is a high-performance jpeg decode SDK for decoding jpeg images. This option traces the rocJPEG API.

```
rocprofv3 --rocjpeg-trace --output-format csv -- <application_path>
```

The above command generates a `rocjpeg_api_trace` file prefixed with the process ID.

```
$ cat 41688_rocjpeg_api_trace.csv
```

Here are the contents of `rocjpeg_api_trace.csv` file:

Table 3.13: rocJPEG trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
ROCJPEG_	rocJpegCreate	41884	41884	105	1286306029650499	1286306248201233
ROCJPEG_	rocJpegStreCreate	41884	41884	502	1286306248250747	1286306248268715
ROCJPEG_	rocJpegStreParse	41884	41884	503	1286306248421385	1286306248680757
ROCJPEG_	rocJpegGetImageInfo	41884	41884	504	1286306248684203	1286306248686556

3.2.14 Process Attachment

`rocprofv3` supports attaching to already running processes to profile them dynamically without requiring application restart. This is particularly useful for long-running applications, services, or when you need to profile an application that is already in a specific state.

Process attachment uses the `-p`, `--pid`, or `--attach` options (all equivalent) followed by the target process ID. The profiler will instrument the target process and collect the specified tracing or counter data for the configured duration.

Basic attachment syntax:

```
rocprofv3 -p <PID> <tracing_options>
# or
rocprofv3 --pid <PID> <tracing_options>
# or
rocprofv3 --attach <PID> <tracing_options>
```

Example: Attach to a running process and collect HIP traces:

```
# Find the process ID of your application
ps aux | grep my_application

# Attach to the process (replace 12345 with actual PID)
rocprofv3 --attach 12345 --hip-trace --output-format csv
```

Example: Attach with multiple tracing options:

```
rocprofv3 -p 12345 --hip-trace --kernel-trace --memory-copy-trace --output-format json
```

Example: Attach with counter collection:

```
rocprofv3 --pid 12345 --pmc SQ_WAVES GRBM_COUNT --output-format csv
```

Important considerations for process attachment:

- The target process must be running and actively using GPU resources for meaningful profiling data
- Attachment requires appropriate system permissions (may need elevated privileges depending on the target process)
- The profiler will collect data for the entire remaining lifetime of the process or until the configured collection period expires

- Use `--attach-duration-msec` to specify how long to profile the attached process (in milliseconds)

Example with duration control:

```
# Attach and profile for 5 seconds
rocprofv3 --attach 12345 --attach-duration-msec 5000 --sys-trace --output-format csv
```

The attachment functionality works with all tracing and profiling options available in `rocprofv3`, providing the same comprehensive analysis capabilities as standard application launching.

3.2.15 Post-processing tracing options

`rocprofv3` provides options to collect tracing summary or statistics after conclusion of a tracing session. These options are described here.

3.2.15.1 Stats

This option collects statistics for the enabled tracing types. For example, it collects statistics of HIP APIs, when HIP trace is enabled. The statistics help to determine the API or function that took the most amount of time.

```
rocprofv3 --stats --hip-trace --output-format csv -- <application_path>
```

The preceding command generates a `hip_api_stats.csv`, `domain_stats.csv` and `hip_api_trace.csv` file prefixed with the process ID.

```
$ cat hip_api_stats.csv
```

Here are the contents of `hip_api_stats.csv` file:

Table 3.14: HIP stats

Name	Calls	TotalDurationNs	AverageNs	Percentage	MinNs	MaxNs	StdDev
hipStreamCreateWithFlags	4	262497406	65624351.500000	85.15	3991286	24912184	12233253 1.343496
hipGetDeviceCount	1	32505687	32505687.000000	10.54	32505687	32505687	0.00000000e+00
hipHostMalloc	12	6096409	508034.083333	1.98	443793	548024	39236.753678
hipFree	12	1994421	166201.750000	0.6470	7790	1036046	299086.860470
hipMemcpyAsync	12	1368378	114031.500000	0.4439	2490	764044	249308.051619
hipMallocAsync	12	927255	77271.250000	0.3008	51540	107671	20487.475966
hipStreamSynchronize	12	870486	72540.500000	0.2824	140	866606	250065.900069
hipLaunchKernel	16	692734	43295.875000	0.2247	1000	670044	167133.656647
hipStreamDestroy	4	619905	154976.250000	0.2011	92901	339252	122852.320356
hipDeviceSynchronize	4	404252	101063.000000	0.1311	570	385212	189518.505401
hipHostFree	12	271202	22600.166667	0.0880	11950	34950	7480.268600
__hipRegisterFatBinary	1	9000	9000.000000	2.920e-03	9000	9000	0.00000000e+00
__hipRegisterFunction	4	6150	1537.500000	1.995e-03	230	5370	2555.091323
__hipPushCallConfiguration	16	2460	153.750000	7.980e-04	70	1140	267.503894
__hipPopCallConfiguration	16	2000	125.000000	6.488e-04	70	680	151.613544
hipGetLastError	16	1270	79.375000	4.120e-04	50	440	96.295985
hipSetDevice	1	660	660.000000	2.141e-04	660	660	0.00000000e+00

Here are the contents of `domain_stats.csv` file:

Table 3.15: Domain stats

Name	Calls	TotalDurationNs	AverageNs	Percentage	MinNs	MaxNs	StdDev
HIP_API	13	458514859	35270373.769231	100.00	2300	35227661	99315857.546240

For the description of the fields in the output file, see *Output file fields*.

3.2.15.2 Summary

This option displays a summary of tracing data for the enabled tracing type, after conclusion of the profiling session.

```
rocprofv3 -S --hip-trace -- <application_path>
```

```
ROCPROFV3 SUMMARY:
|                NAME                |   DOMAIN   |   CALLS   | DURATION (nsec) | AVERAGE (nsec) | PERCENT (INC) |  MIN (nsec)  |  MAX (nsec)  |    STDDEV     |
|------------------------------------|------------|-----------|-----------------|----------------|---------------|--------------|--------------|---------------|
| hipMemcpy                          | HIP_API    |         3 |        21208834 |       7.070e+07 |     65.244182 |         9288 |    211829985 |      1.222e+08 |
| hipGetDevicePropertiesR0600        | HIP_API    |         1 |       112327545 |       1.123e+08 |     34.555028 |    112327545 |    112327545 |      0.000e+00 |
| hipLaunchKernel                    | HIP_API    |         1 |          374897 |       3.749e+05 |      0.115329 |       374897 |       374897 |      0.000e+00 |
| hipFree                            | HIP_API    |         2 |          216804 |       1.084e+05 |      0.066695 |        83953 |       132851 |      3.458e+04 |
| hipMalloc                          | HIP_API    |         2 |           51981 |       2.599e+04 |      0.015991 |        10502 |        41479 |      2.190e+04 |
```

3.2.15.3 Summary per domain

This option displays a summary of each tracing domain for the enabled tracing type, after conclusion of the profiling session.

```
rocprofv3 -D --hsa-trace --hip-trace --output-format csv  -- <application_path>
```

The preceding command generates a `hip_trace.csv` and `hsa_trace.csv` file prefixed with the process ID along with displaying the summary of each domain.

3.2.15.4 Summary groups

This option displays a summary of multiple domains for the domain names specified on the command line. The summary groups can be separated using a pipe (|) symbol.

To see a summary for `MEMORY_COPY` domains, use:

```
rocprofv3 --summary-groups MEMORY_COPY --sys-trace  -- <application_path>
```

```
ROCPROFV3 MEMORY_COPY SUMMARY:
|                NAME                |   DOMAIN   |   CALLS   | DURATION (nsec) | AVERAGE (nsec) | PERCENT (INC) |  MIN (nsec)  |  MAX (nsec)  |    STDDEV     |
|------------------------------------|------------|-----------|-----------------|----------------|---------------|--------------|--------------|---------------|
| MEMORY_COPY_DEVICE_TO_HOST         | MEMORY_COPY |         1 |          109355 |       1.094e+05 |     51.617852 |       109355 |       109355 |      0.000e+00 |
| MEMORY_COPY_HOST_TO_DEVICE         | MEMORY_COPY |         1 |          102500 |       1.025e+05 |     48.382148 |       102500 |       102500 |      0.000e+00 |
```

To see a summary for `MEMORY_COPY` and `HIP_API` domains, use:

```
rocprofv3 --summary-groups 'MEMORY_COPY|HIP_API' --sys-trace -- <application_path>
```

```
ROCPROFV3 HIP_API + MEMORY_COPY SUMMARY:

|                NAME                |    DOMAIN     |     CALLS     | DURATION (nsec) | AVERAGE (nsec) | PERCENT (INC) |  MIN (nsec)   |  MAX (nsec)   |     STDDEV     |
|------------------------------------|---------------|---------------|-----------------|----------------|---------------|---------------|---------------|----------------|
| hipMemcpy                          | HIP_API       |             2 |       228708882 |    1.144e+08   |    70.476616  |       201413  |       228507469 |    1.614e+08   |
| hipGetDevicePropertiesR0600        | HIP_API       |             1 |        94988182 |    9.499e+07   |    29.270597  |        94988182 |        94988182 |    0.000e+00   |
| hipLaunchKernel                    | HIP_API       |             1 |          338689 |    3.387e+05   |     0.104367  |          338689 |          338689 |    0.000e+00   |
| hipFree                            | HIP_API       |             2 |          197402 |    9.870e+04   |     0.060829  |           82090 |          115312 |    2.349e+04   |
| MEMORY_COPY_DEVICE_TO_HOST         | MEMORY_COPY   |             1 |          109034 |    1.090e+05   |     0.033599  |          109034 |          109034 |    0.000e+00   |
| MEMORY_COPY_HOST_TO_DEVICE         | MEMORY_COPY   |             1 |          102059 |    1.021e+05   |     0.031449  |          102059 |          102059 |    0.000e+00   |
| hipMalloc                          | HIP_API       |             2 |           64612 |    3.231e+04   |     0.019910  |           20713 |           43899 |    1.639e+04   |
| __hipRegisterFatBinary             | HIP_API       |             1 |            4124 |    4.124e+03   |     0.001271  |            4124 |            4124 |    0.000e+00   |
| __hipRegisterFunction              | HIP_API       |             1 |            2877 |    2.877e+03   |     0.000887  |            2877 |            2877 |    0.000e+00   |
| __hipPushCallConfiguration         | HIP_API       |             1 |             788 |    7.880e+02   |     0.000243  |             788 |             788 |    0.000e+00   |
| __hipPopCallConfiguration          | HIP_API       |             1 |             753 |    7.530e+02   |     0.000232  |             753 |             753 |    0.000e+00   |
```

3.2.15.5 Summary output file

This option specifies the output file for the summary. By default, the summary is displayed on `stderr`. To specify another output file for summary, use:

```
rocprofv3 -S -D --summary-output-file filename --sys-trace -- <application_path>
```

The preceding command generates an output file named "filename" consisting of the summary for each domain. This also generates the files for the enabled tracing types under `-sys-trace` option.

```
ROCPROFV3 HSA_API SUMMARY:

|                    NAME                     |    DOMAIN     |      CALLS      | DURATION␣
↪(nsec) | AVERAGE (nsec)  | PERCENT (INC) |   MIN (nsec)    |   MAX (nsec)    |    ␣
↪STDDEV      |
|---------------------------------------------|---------------|-----------------|-----------
↪------|-----------------|---------------|-----------------|-----------------|----------
↪-------|
| hsa_queue_create                            | HSA_API       |               4 |      ␣
↪280077621 |       7.002e+07 |     75.372632 |        55026812 |       113288760 |     ␣
↪ 2.885e+07 |
| hsa_amd_memory_async_copy_on_engine         | HSA_API       |              24 |      ␣
↪55617052 |       2.317e+06 |     14.967292 |            7580 |        55195188 |     ␣
↪1.126e+07 |
| hsa_amd_memory_pool_allocate                | HSA_API       |              67 |      ␣
↪26428438 |       3.945e+05 |      7.112246 |            1510 |          857592 |     ␣
↪1.782e+05 |
| hsa_amd_memory_pool_free                    | HSA_API       |              72 |      ␣
↪5176173 |       7.189e+04 |      1.392977 |             290 |          170374 |     ␣
↪3.903e+04 |
| hsa_executable_freeze                       | HSA_API       |               2 |      ␣
↪964125 |       4.821e+05 |      0.259459 |          437471 |          526654 |       6.
↪306e+04 |
| hsa_signal_wait_scacquire                   | HSA_API       |              26 |      ␣
↪853122 |       3.281e+04 |      0.229587 |            2530 |          100782 |       3.
↪394e+04 |
| hsa_executable_load_agent_code_object       | HSA_API       |               2 |      ␣
↪616175 |       3.081e+05 |      0.165821 |          254476 |          361699 |       7.
↪582e+04 |
| hsa_amd_agents_allow_access                 | HSA_API       |              35 |      ␣
↪430680 |       1.231e+04 |      0.115902 |            4830 |           55182 |       9.
↪939e+03 |
| hsa_signal_store_screlease                  | HSA_API       |              56 |      ␣
```

(continues on next page)

(continued from previous page)

```
↪381491 |       6.812e+03 |      0.102664 |            1560 |           41831 |       7.
↪895e+03 |
| hsa_signal_create                           | HSA_API      |             107 |          ␣
↪160889 |       1.504e+03 |      0.043297 |              80 |            5650 |       1.
↪475e+03 |
| hsa_code_object_reader_create_from_memory | HSA_API      |               2 |          ␣
↪151314 |       7.566e+04 |      0.040721 |           32121 |          119193 |       6.
↪157e+04 |
| hsa_signal_load_relaxed                     | HSA_API      |            1296 |          ␣
↪137626 |       1.062e+02 |      0.037037 |              20 |            2930 |       2.
↪712e+02 |
| hsa_signal_destroy                          | HSA_API      |             618 |          ␣
↪111224 |       1.800e+02 |      0.029932 |              40 |            1540 |       2.
↪429e+02 |
| hsa_agent_get_info                          | HSA_API      |              65 |          ␣
↪77472 |       1.192e+03 |      0.020849 |              30 |           47121 |       6.
↪341e+03 |
| hsa_amd_signal_create                       | HSA_API      |             512 |          ␣
↪61290 |       1.197e+02 |      0.016494 |              40 |             930 |       1.
↪559e+02 |
| hsa_amd_signal_async_handler                | HSA_API      |              24 |          ␣
↪52641 |       2.193e+03 |      0.014166 |            1180 |            4020 |       9.
↪252e+02 |
| hsa_executable_iterate_symbols              | HSA_API      |              14 |          ␣
↪52521 |       3.752e+03 |      0.014134 |            2740 |            6940 |       1.
↪105e+03 |
| hsa_amd_memory_copy_engine_status           | HSA_API      |              18 |          ␣
↪47370 |       2.632e+03 |      0.012748 |             260 |            7990 |       2.
↪274e+03 |
| hsa_iterate_agents                          | HSA_API      |               1 |          ␣
↪41391 |       4.139e+04 |      0.011139 |           41391 |           41391 |       0.
↪000e+00 |
| hsa_executable_create_alt                   | HSA_API      |               2 |          ␣
↪40470 |       2.024e+04 |      0.010891 |            7530 |           32940 |       1.
↪797e+04 |
| hsa_isa_get_info_alt                        | HSA_API      |               2 |          ␣
↪30391 |       1.520e+04 |      0.008179 |            2490 |           27901 |       1.
↪797e+04 |
| hsa_signal_silent_store_relaxed             | HSA_API      |              48 |          ␣
↪24920 |       5.192e+02 |      0.006706 |              20 |            4570 |       7.
↪120e+02 |
| hsa_amd_agent_iterate_memory_pools          | HSA_API      |               5 |          ␣
↪20221 |       4.044e+03 |      0.005442 |            2561 |            8600 |       2.
↪574e+03 |
| hsa_queue_add_write_index_screlease         | HSA_API      |              56 |          ␣
↪ 7270 |       1.298e+02 |      0.001956 |              30 |            2310 |       3.
↪471e+02 |
| hsa_amd_profiling_set_profiler_enabled      | HSA_API      |               4 |          ␣
↪ 5600 |       1.400e+03 |      0.001507 |            1370 |            1470 |       4.
↪690e+01 |
| hsa_executable_symbol_get_info              | HSA_API      |             152 |          ␣
↪ 5470 |       3.599e+01 |      0.001472 |              30 |             340 |       3.
```

(continues on next page)

(continued from previous page)

```
↪563e+01 |
| hsa_queue_load_read_index_relaxed         | HSA_API      |              56 |          ␣
↪ 4560 |       8.143e+01 |      0.001227 |              20 |            1310 |       1.
↪863e+02 |
| hsa_executable_get_symbol_by_name         | HSA_API      |              14 |          ␣
↪ 4500 |       3.214e+02 |      0.001211 |             110 |            1510 |       4.
↪732e+02 |
| hsa_queue_load_read_index_scacquire       | HSA_API      |              56 |          ␣
↪ 3040 |       5.429e+01 |      0.000818 |              30 |             690 |       8.
↪705e+01 |
| hsa_amd_memory_pool_get_info              | HSA_API      |              43 |          ␣
↪ 1770 |       4.116e+01 |      0.000476 |              30 |             270 |       3.
↪640e+01 |
| hsa_system_get_info                       | HSA_API      |               4 |          ␣
↪ 1750 |       4.375e+02 |      0.000471 |              40 |             830 |       3.
↪544e+02 |
| hsa_amd_agent_memory_pool_get_info        | HSA_API      |              13 |          ␣
↪ 1140 |       8.769e+01 |      0.000307 |              30 |             640 |       1.
↪664e+02 |
| hsa_agent_iterate_isas                    | HSA_API      |               1 |          ␣
↪  700 |       7.000e+02 |      0.000188 |             700 |             700 |       0.
↪000e+00 |
| hsa_system_get_major_extension_table      | HSA_API      |               1 |          ␣
↪  190 |       1.900e+02 |      0.000051 |             190 |             190 |       0.
↪000e+00 |


ROCPROFV3 HIP_API SUMMARY:

|                  NAME                     |    DOMAIN    |      CALLS      | DURATION␣
↪(nsec) | AVERAGE (nsec)  | PERCENT (INC) |   MIN (nsec)    |   MAX (nsec)    |    ␣
↪STDDEV      |
|-------------------------------------------|--------------|-----------------|------------
↪-----|-----------------|---------------|-----------------|-----------------|-----------
↪------|
| hipStreamCreateWithFlags                  | HIP_API      |               8 |      ␣
↪406507215 |       5.081e+07 |     71.307804 |          735979 |       233800881 |     ␣
↪ 7.889e+07 |
| hipGetDeviceCount                         | HIP_API      |               1 |       ␣
↪76707894 |       7.671e+07 |     13.455780 |        76707894 |        76707894 |      ␣
↪0.000e+00 |
| hipMemcpyAsync                            | HIP_API      |              24 |       ␣
↪56109444 |       2.338e+06 |      9.842485 |           11640 |        55299811 |      ␣
↪1.128e+07 |
| hipHostMalloc                             | HIP_API      |              24 |       ␣
↪13007523 |       5.420e+05 |      2.281726 |          416631 |          866382 |      ␣
↪1.206e+05 |
| hipMallocAsync                            | HIP_API      |              24 |        ␣
↪7304847 |       3.044e+05 |      1.281386 |          275397 |          353719 |      ␣
↪2.207e+04 |
| hipHostFree                               | HIP_API      |              24 |        ␣
↪2786484 |       1.161e+05 |      0.488793 |           72242 |          221646 |      ␣
```

(continues on next page)

(continued from previous page)

```
↪4.606e+04 |
| hipStreamDestroy                          | HIP_API      |               8 |        ␣
↪2137924 |       2.672e+05 |      0.375026 |          221596 |          377469 |      ␣
↪5.489e+04 |
| hipLaunchKernel                           | HIP_API      |              32 |        ␣
↪2080214 |       6.501e+04 |      0.364902 |            8850 |         1608721 |      ␣
↪2.819e+05 |
| hipFree                                   | HIP_API      |              24 |        ␣
↪1572948 |       6.554e+04 |      0.275920 |            2130 |          186994 |      ␣
↪4.815e+04 |
| hipStreamSynchronize                      | HIP_API      |              24 |        ␣
↪1452706 |       6.053e+04 |      0.254828 |           20810 |          135803 |      ␣
↪3.469e+04 |
| __hipRegisterFunction                     | HIP_API      |               4 |        ␣
↪294207 |       7.355e+04 |      0.051609 |             210 |          291807 |       1.
↪455e+05 |
| hipDeviceSynchronize                      | HIP_API      |               4 |        ␣
↪50663 |       1.267e+04 |      0.008887 |             510 |           23621 |       9.
↪554e+03 |
| __hipRegisterFatBinary                    | HIP_API      |               1 |        ␣
↪43811 |       4.381e+04 |      0.007685 |           43811 |           43811 |       0.
↪000e+00 |
| __hipPushCallConfiguration                | HIP_API      |              32 |          ␣
↪6250 |       1.953e+02 |      0.001096 |              60 |            3640 |       6.
↪308e+02 |
| __hipPopCallConfiguration                 | HIP_API      |              32 |          ␣
↪4780 |       1.494e+02 |      0.000838 |              60 |            2520 |       4.
↪340e+02 |
| hipGetLastError                           | HIP_API      |              32 |          ␣
↪4471 |       1.397e+02 |      0.000784 |              60 |            2381 |       4.
↪092e+02 |
| hipSetDevice                              | HIP_API      |               1 |          ␣
↪2570 |       2.570e+03 |      0.000451 |            2570 |            2570 |       0.
↪000e+00 |


ROCPROFV3 KERNEL_DISPATCH SUMMARY:

|                                   NAME                                    |     DOMAIN␣
↪     |      CALLS      | DURATION (nsec) | AVERAGE (nsec)  | PERCENT (INC) |   MIN␣
↪(nsec)    |   MAX (nsec)    |     STDDEV      |
|---------------------------------------------------------------------------|-----------
↪-----|-----------------|-----------------|-----------------|---------------|-----------
↪------|-----------------|-----------------|
| void addition_kernel<float>(float*, float const*, float const*, int, int) | KERNEL_
↪DISPATCH |               8 |          184324 |       2.304e+04 |     40.681542 |      ␣
↪    11200 |           98802 |       3.062e+04 |
| divide_kernel(float*, float const*, float const*, int, int)               | KERNEL_
↪DISPATCH |               8 |           94482 |       1.181e+04 |     20.852811 |      ␣
↪    10240 |           13520 |       1.061e+03 |
| multiply_kernel(float*, float const*, float const*, int, int)             | KERNEL_
↪DISPATCH |               8 |           91763 |       1.147e+04 |     20.252709 |      ␣
```

(continues on next page)

(continued from previous page)

```
↪     9800 |           12800 |       9.417e+02 |
| subtract_kernel(float*, float const*, float const*, int, int)             | KERNEL_
↪DISPATCH |               8 |           82521 |       1.032e+04 |     18.212938 |      ␣
↪     8320 |           12920 |       1.436e+03 |


ROCPROFV3 MEMORY_COPY SUMMARY:

|                   NAME                    |    DOMAIN    |      CALLS      | DURATION␣
↪(nsec) | AVERAGE (nsec)  | PERCENT (INC) |   MIN (nsec)    |   MAX (nsec)    |    ␣
↪STDDEV      |
|-------------------------------------------|--------------|-----------------|------------
↪-----|-----------------|---------------|-----------------|-----------------|-----------
↪------|
| MEMORY_COPY_HOST_TO_DEVICE                | MEMORY_COPY  |              16 |        ␣
↪3691929 |       2.307e+05 |     85.494053 |           74842 |          284487 |      ␣
↪6.265e+04 |
| MEMORY_COPY_DEVICE_TO_HOST                | MEMORY_COPY  |               8 |        ␣
↪626417 |       7.830e+04 |     14.505947 |           74842 |           98603 |       8.
↪207e+03 |


ROCPROFV3 MEMORY_ALLOCATION SUMMARY:

|                   NAME                    |      DOMAIN       |      CALLS      |␣
↪DURATION (nsec) | AVERAGE (nsec)  | PERCENT (INC) |   MIN (nsec)    |   MAX (nsec)    ␣
↪|     STDDEV      |
|-------------------------------------------|-------------------|-----------------|-------
↪----------|-----------------|---------------|-----------------|-----------------|------
↪-----------|
| MEMORY_ALLOCATION_ALLOCATE                | MEMORY_ALLOCATION |              67 |      ␣
↪ 26314096 |       3.927e+05 |     83.661617 |             950 |          856812 |     ␣
↪ 1.785e+05 |
| MEMORY_ALLOCATION_FREE                    | MEMORY_ALLOCATION |              72 |      ␣
↪  5138913 |       7.137e+04 |     16.338383 |              20 |          166234 |     ␣
↪ 3.882e+04 |


ROCPROFV3 SUMMARY:

|                                  NAME                                     |     ␣
↪DOMAIN       |      CALLS      | DURATION (nsec) | AVERAGE (nsec)  | PERCENT (INC) |  ␣
↪MIN (nsec)    |   MAX (nsec)    |     STDDEV      |
|---------------------------------------------------------------------------|------------
↪-------|-----------------|-----------------|-----------------|---------------|---------
↪--------|-----------------|-----------------|
| hipStreamCreateWithFlags                                                  | HIP_API   ␣
↪       |               8 |       406507215 |       5.081e+07 |     41.569873 |        ␣
↪ 735979 |       233800881 |       7.889e+07 |
| hsa_queue_create                                                          | HSA_API   ␣
↪       |               4 |       280077621 |       7.002e+07 |     28.641044 |       ␣
↪55026812 |       113288760 |       2.885e+07 |
```

(continues on next page)

(continued from previous page)

```
| hipGetDeviceCount                                                    | HIP_API    ␣
↪       |               1 |        76707894 |       7.671e+07 |      7.844233 |       ␣
↪76707894 |        76707894 |       0.000e+00 |
| hipMemcpyAsync                                                       | HIP_API    ␣
↪       |              24 |        56109444 |       2.338e+06 |      5.737813 |       ␣
↪  11640 |        55299811 |       1.128e+07 |
| hsa_amd_memory_async_copy_on_engine                                  | HSA_API    ␣
↪       |              24 |        55617052 |       2.317e+06 |      5.687461 |       ␣
↪   7580 |        55195188 |       1.126e+07 |
| hsa_amd_memory_pool_allocate                                         | HSA_API    ␣
↪       |              67 |        26428438 |       3.945e+05 |      2.702601 |       ␣
↪   1510 |          857592 |       1.782e+05 |
| MEMORY_ALLOCATION_ALLOCATE                                           | MEMORY_
↪ALLOCATION |              67 |        26314096 |       3.927e+05 |      2.690908 |    ␣
↪        950 |          856812 |       1.785e+05 |
| hipHostMalloc                                                        | HIP_API    ␣
↪       |              24 |        13007523 |       5.420e+05 |      1.330164 |       ␣
↪ 416631 |          866382 |       1.206e+05 |
| hipMallocAsync                                                       | HIP_API    ␣
↪       |              24 |         7304847 |       3.044e+05 |      0.747002 |       ␣
↪ 275397 |          353719 |       2.207e+04 |
| hsa_amd_memory_pool_free                                             | HSA_API    ␣
↪       |              72 |         5176173 |       7.189e+04 |      0.529321 |       ␣
↪    290 |          170374 |       3.903e+04 |
| MEMORY_ALLOCATION_FREE                                               | MEMORY_
↪ALLOCATION |              72 |         5138913 |       7.137e+04 |      0.525511 |    ␣
↪         20 |          166234 |       3.882e+04 |
| MEMORY_COPY_HOST_TO_DEVICE                                           | MEMORY_
↪COPY       |              16 |         3691929 |       2.307e+05 |      0.377541 |    ␣
↪      74842 |          284487 |       6.265e+04 |
| hipHostFree                                                          | HIP_API    ␣
↪       |              24 |         2786484 |       1.161e+05 |      0.284949 |       ␣
↪  72242 |          221646 |       4.606e+04 |
| hipStreamDestroy                                                     | HIP_API    ␣
↪       |               8 |         2137924 |       2.672e+05 |      0.218626 |       ␣
↪ 221596 |          377469 |       5.489e+04 |
| hipLaunchKernel                                                      | HIP_API    ␣
↪       |              32 |         2080214 |       6.501e+04 |      0.212725 |       ␣
↪   8850 |         1608721 |       2.819e+05 |
| hipFree                                                              | HIP_API    ␣
↪       |              24 |         1572948 |       6.554e+04 |      0.160851 |       ␣
↪   2130 |          186994 |       4.815e+04 |
| hipStreamSynchronize                                                 | HIP_API    ␣
↪       |              24 |         1452706 |       6.053e+04 |      0.148555 |       ␣
↪  20810 |          135803 |       3.469e+04 |
| hsa_executable_freeze                                                | HSA_API    ␣
↪       |               2 |          964125 |       4.821e+05 |      0.098592 |       ␣
↪ 437471 |          526654 |       6.306e+04 |
| hsa_signal_wait_scacquire                                            | HSA_API    ␣
↪       |              26 |          853122 |       3.281e+04 |      0.087241 |       ␣
↪   2530 |          100782 |       3.394e+04 |
| MEMORY_COPY_DEVICE_TO_HOST                                           | MEMORY_
```

(continues on next page)

(continued from previous page)

```
↪COPY       |               8 |          626417 |       7.830e+04 |      0.064058 |␣
↪      74842 |           98603 |       8.207e+03 |
| hsa_executable_load_agent_code_object                                          | HSA_API   ␣
↪       |               2 |          616175 |       3.081e+05 |      0.063011 |␣
↪ 254476 |          361699 |       7.582e+04 |
| hsa_amd_agents_allow_access                                                    | HSA_API   ␣
↪       |              35 |          430680 |       1.231e+04 |      0.044042 |␣
↪   4830 |           55182 |       9.939e+03 |
| hsa_signal_store_screlease                                                     | HSA_API   ␣
↪       |              56 |          381491 |       6.812e+03 |      0.039012 |␣
↪   1560 |           41831 |       7.895e+03 |
| __hipRegisterFunction                                                          | HIP_API   ␣
↪       |               4 |          294207 |       7.355e+04 |      0.030086 |␣
↪    210 |          291807 |       1.455e+05 |
| void addition_kernel<float>(float*, float const*, float const*, int, int) | KERNEL_
↪DISPATCH   |               8 |          184324 |       2.304e+04 |      0.018849 |␣
↪      11200 |           98802 |       3.062e+04 |
| hsa_signal_create                                                              | HSA_API   ␣
↪       |             107 |          160889 |       1.504e+03 |      0.016453 |␣
↪     80 |            5650 |       1.475e+03 |
| hsa_code_object_reader_create_from_memory                                      | HSA_API   ␣
↪       |               2 |          151314 |       7.566e+04 |      0.015474 |␣
↪  32121 |          119193 |       6.157e+04 |
| hsa_signal_load_relaxed                                                        | HSA_API   ␣
↪       |            1296 |          137626 |       1.062e+02 |      0.014074 |␣
↪     20 |            2930 |       2.712e+02 |
| hsa_signal_destroy                                                             | HSA_API   ␣
↪       |             618 |          111224 |       1.800e+02 |      0.011374 |␣
↪     40 |            1540 |       2.429e+02 |
| divide_kernel(float*, float const*, float const*, int, int)                | KERNEL_
↪DISPATCH   |               8 |           94482 |       1.181e+04 |      0.009662 |␣
↪      10240 |           13520 |       1.061e+03 |
| multiply_kernel(float*, float const*, float const*, int, int)              | KERNEL_
↪DISPATCH   |               8 |           91763 |       1.147e+04 |      0.009384 |␣
↪       9800 |           12800 |       9.417e+02 |
| subtract_kernel(float*, float const*, float const*, int, int)              | KERNEL_
↪DISPATCH   |               8 |           82521 |       1.032e+04 |      0.008439 |␣
↪       8320 |           12920 |       1.436e+03 |
| hsa_agent_get_info                                                             | HSA_API   ␣
↪       |              65 |           77472 |       1.192e+03 |      0.007922 |␣
↪     30 |           47121 |       6.341e+03 |
| hsa_amd_signal_create                                                          | HSA_API   ␣
↪       |             512 |           61290 |       1.197e+02 |      0.006268 |␣
↪     40 |             930 |       1.559e+02 |
| hsa_amd_signal_async_handler                                                   | HSA_API   ␣
↪       |              24 |           52641 |       2.193e+03 |      0.005383 |␣
↪   1180 |            4020 |       9.252e+02 |
| hsa_executable_iterate_symbols                                                 | HSA_API   ␣
↪       |              14 |           52521 |       3.752e+03 |      0.005371 |␣
↪   2740 |            6940 |       1.105e+03 |
| hipDeviceSynchronize                                                           | HIP_API   ␣
↪       |               4 |           50663 |       1.267e+04 |      0.005181 |␣
```

(continues on next page)

(continued from previous page)

```
↪    510 |           23621 |       9.554e+03 |
| hsa_amd_memory_copy_engine_status                                    | HSA_API   ␣
↪       |              18 |           47370 |       2.632e+03 |      0.004844 |        ␣
↪    260 |            7990 |       2.274e+03 |
| __hipRegisterFatBinary                                               | HIP_API   ␣
↪       |               1 |           43811 |       4.381e+04 |      0.004480 |        ␣
↪  43811 |           43811 |       0.000e+00 |
| hsa_iterate_agents                                                   | HSA_API   ␣
↪       |               1 |           41391 |       4.139e+04 |      0.004233 |        ␣
↪  41391 |           41391 |       0.000e+00 |
| hsa_executable_create_alt                                            | HSA_API   ␣
↪       |               2 |           40470 |       2.024e+04 |      0.004139 |        ␣
↪   7530 |           32940 |       1.797e+04 |
| hsa_isa_get_info_alt                                                 | HSA_API   ␣
↪       |               2 |           30391 |       1.520e+04 |      0.003108 |        ␣
↪   2490 |           27901 |       1.797e+04 |
| hsa_signal_silent_store_relaxed                                      | HSA_API   ␣
↪       |              48 |           24920 |       5.192e+02 |      0.002548 |        ␣
↪     20 |            4570 |       7.120e+02 |
| hsa_amd_agent_iterate_memory_pools                                   | HSA_API   ␣
↪       |               5 |           20221 |       4.044e+03 |      0.002068 |        ␣
↪   2561 |           8600 |       2.574e+03 |
| hsa_queue_add_write_index_screlease                                  | HSA_API   ␣
↪       |              56 |            7270 |       1.298e+02 |      0.000743 |        ␣
↪     30 |            2310 |       3.471e+02 |
| __hipPushCallConfiguration                                           | HIP_API   ␣
↪       |              32 |            6250 |       1.953e+02 |      0.000639 |        ␣
↪     60 |            3640 |       6.308e+02 |
| hsa_amd_profiling_set_profiler_enabled                               | HSA_API   ␣
↪       |               4 |            5600 |       1.400e+03 |      0.000573 |        ␣
↪   1370 |            1470 |       4.690e+01 |
| hsa_executable_symbol_get_info                                       | HSA_API   ␣
↪       |             152 |            5470 |       3.599e+01 |      0.000559 |        ␣
↪     30 |             340 |       3.563e+01 |
| __hipPopCallConfiguration                                            | HIP_API   ␣
↪       |              32 |            4780 |       1.494e+02 |      0.000489 |        ␣
↪     60 |            2520 |       4.340e+02 |
| hsa_queue_load_read_index_relaxed                                    | HSA_API   ␣
↪       |              56 |            4560 |       8.143e+01 |      0.000466 |        ␣
↪     20 |            1310 |       1.863e+02 |
| hsa_executable_get_symbol_by_name                                    | HSA_API   ␣
↪       |              14 |            4500 |       3.214e+02 |      0.000460 |        ␣
↪    110 |            1510 |       4.732e+02 |
| hipGetLastError                                                      | HIP_API   ␣
↪       |              32 |            4471 |       1.397e+02 |      0.000457 |        ␣
↪     60 |            2381 |       4.092e+02 |
| hsa_queue_load_read_index_scacquire                                  | HSA_API   ␣
↪       |              56 |            3040 |       5.429e+01 |      0.000311 |        ␣
↪     30 |             690 |       8.705e+01 |
| hipSetDevice                                                         | HIP_API   ␣
↪       |               1 |            2570 |       2.570e+03 |      0.000263 |        ␣
↪   2570 |            2570 |       0.000e+00 |
```

(continues on next page)

(continued from previous page)

```
| hsa_amd_memory_pool_get_info                                                  | HSA_API    ␣
↪       |               43 |             1770 |       4.116e+01 |      0.000181 |        ␣
↪     30 |              270 |       3.640e+01 |
| hsa_system_get_info                                                           | HSA_API    ␣
↪       |                4 |             1750 |       4.375e+02 |      0.000179 |        ␣
↪     40 |              830 |       3.544e+02 |
| hsa_amd_agent_memory_pool_get_info                                            | HSA_API    ␣
↪       |               13 |             1140 |       8.769e+01 |      0.000117 |        ␣
↪     30 |              640 |       1.664e+02 |
| hsa_agent_iterate_isas                                                        | HSA_API    ␣
↪       |                1 |              700 |       7.000e+02 |      0.000072 |        ␣
↪    700 |              700 |       0.000e+00 |
| hsa_system_get_major_extension_table                                          | HSA_API    ␣
↪       |                1 |              190 |       1.900e+02 |      0.000019 |        ␣
↪    190 |              190 |       0.000e+00 |
```

3.2.16 Collecting traces using input file

The preceding sections describe how to collect traces by specifying the desired tracing type on the command line. You can also specify the desired tracing types in an input file in YAML (.yaml/.yml), or JSON (.json) format. You can supply any command-line option for tracing in the input file.

Here is a sample input.yaml file for collecting tracing summary:

```yaml
jobs:
  - output_directory: "@CMAKE_CURRENT_BINARY_DIR@/%env{ARBITRARY_ENV_VARIABLE}%"
    output_file: out
    output_format: [pftrace, json, otf2]
    log_level: env
    runtime_trace: true
    kernel_rename: true
    summary: true
    summary_per_domain: true
    summary_groups: ["KERNEL_DISPATCH|MEMORY_COPY"]
    summary_output_file: "summary"
```

Here is a sample input.json file for collecting tracing summary:

```json
{
  "jobs": [
    {
      "output_directory": "out-directory",
      "output_file": "out",
      "output_format": ["pftrace", "json", "otf2"],
      "log_level": "env",
      "runtime_trace": true,
      "kernel_rename": true,
      "summary": true,
      "summary_per_domain": true,
      "summary_groups": ["KERNEL_DISPATCH|MEMORY_COPY"],
      "summary_output_file": "summary"
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

Here is the input schema (properties) of JSON or YAML input files:

- **jobs** *(array)*: `rocprofv3` input data per application run.
  - **Items** *(object)*: Data for `rocprofv3`
    - **hip_trace** *(boolean)*
    - **hip_runtime_trace** *(boolean)*
    - **hip_compiler_trace** *(boolean)*
    - **marker_trace** *(boolean)*
    - **kernel_trace** *(boolean)*
    - **memory_copy_trace** *(boolean)*
    - **memory_allocation_trace** *(boolean)*
    - **scratch_memory_trace** *(boolean)*
    - **stats** *(boolean)*
    - **hsa_trace** *(boolean)*
    - **hsa_core_trace** *(boolean)*
    - **hsa_amd_trace** *(boolean)*
    - **hsa_finalize_trace** *(boolean)*
    - **hsa_image_trace** *(boolean)*
    - **sys_trace** *(boolean)*
    - **minimum-output-data** *(integer)*
    - **disable-signal-handlers** *(boolean)*
    - **mangled_kernels** *(boolean)*
    - **truncate_kernels** *(boolean)*
    - **output_file** *(string)*
    - **output_directory** *(string)*
    - **output_format** *(array)*
    - **log_level** *(string)*
    - **preload** *(array)*

For description of the options specified under job items, see *Command-line options*.

To supply the input file for collecting traces, use:

```
rocprofv3 -i input.yaml -- <application_path>
```

Please note that input file format must be a valid YAML or JSON file.

3.2.17 Disabling specific tracing options

When using aggregate tracing options like `--runtime-trace` or `--sys-trace`, you can disable specific tracing options by setting them to `False`. This allows fine-grained control over the traces to be collected.

```
rocprofv3 --runtime-trace --scratch-memory-trace=False -- <application_path>
```

The preceding command enables all traces included in `--runtime-trace` except for scratch memory tracing.

Similarly, for `--sys-trace`:

```
rocprofv3 --sys-trace --hsa-trace=False -- <application_path>
```

The preceding command enables all traces included in `--sys-trace` except for HSA API tracing.

To disable multiple specific tracing options, use:

```
rocprofv3 --sys-trace --hsa-trace=False --scratch-memory-trace=False -- <application_
↪path>
```

This feature is particularly useful to collect most traces excluding specific ones that might be unnecessary for your analysis or that generate excessive data.

3.3 Kernel counter collection

The application tracing functionality allows you to evaluate the duration of kernel execution but is of little help in providing insight into kernel execution details. The kernel counter collection functionality allows you to select kernels for profiling and choose the basic counters or derived metrics to be collected for each kernel execution, thus providing a greater insight into kernel execution.

AMDGPUs are equipped with hardware performance counters that can be used to measure specific values during kernel execution, which are then exported from the GPU and written into the output files at the end of the kernel execution. These performance counters vary according to the GPU. Therefore, it is recommended to examine the hardware counters that can be collected before running the profile.

There are two types of data available for profiling: hardware basic counters and derived metrics.

The derived metrics are the counters derived from the basic counters using mathematical expressions. Note that the basic counters and derived metrics are collectively referred as counters in this document.

To see the counters available on the GPU, use:

```
rocprofv3 --list-avail
```

You can also customize the counters according to the requirement. Such counters are named *Extra counters*.

For a comprehensive list of counters available on MI200, see MI200 performance counters and metrics.

3.3.1 Counter collection using input file

Input files can be in text (.txt), YAML (.yaml/.yml), or JSON (.json) format to specify the the desired counters for collection.

When using input file in text format, the line consisting of the counter names must begin with `pmc`. The number of counters that can be collected in one profiling run are limited by the GPU hardware resources. If too many counters are selected, the kernels need to be executed multiple times(multi-pass execution) to collect all the counters. For multi-pass execution, include multiple `pmc` rows in the input file. Counters in each `pmc` row can be collected in each application run.

Here is a sample input.txt file for specifying counters for collection:

```
$ cat input.txt

pmc: GPUBusy SQ_WAVES
pmc: GRBM_GUI_ACTIVE
```

While the input file in text format can only be used for counter collection, JSON and YAML formats support all the command-line options for profiling. The input file in YAML or JSON format has an array of profiling configurations called jobs. Each job is used to configure profiling for an application execution.

Here is the input schema (properties) of JSON or YAML input files:

- **jobs** *(array)*: `rocprofv3` input data per application run
  - **Items** *(object)*: Data for `rocprofv3`
    - **pmc** *(array)*: list of counters for collection
    - **kernel_include_regex** *(string)*
    - **kernel_exclude_regex** *(string)*
    - **kernel_iteration_range** *(string)*
    - **mangled_kernels** *(boolean)*
    - **truncate_kernels** *(boolean)*
    - **output_file** *(string)*
    - **output_directory** *(string)*
    - **output_format** *(array)*
    - **list_avail** *(boolean)*
    - **log_level** *(string)*
    - **preload** *(array)*
    - **minimum-output-data** *(integer)*
    - **disable-signal-handlers** *(boolean)*
    - **pc_sampling_unit** *(string)*
    - **pc_sampling_method** *(string)*
    - **pc_sampling_interval** *(integer)*
    - **pc_sampling_beta_enabled** *(boolean)*

For description of the options specified under job items, see *Command-line options*.

Here is a sample input.json file for specifying counters for collection along with the options to filter and control the output:

```
$ cat input.json

{
  "jobs": [
     {
        "pmc": ["SQ_WAVES", "GRBM_COUNT", "GRBM_GUI_ACTIVE"]
     },
```

(continues on next page)

(continued from previous page)

```
    {
       "pmc": ["FETCH_SIZE", "WRITE_SIZE"],
       "kernel_include_regex": ".*_kernel",
       "kernel_exclude_regex": "multiply",
       "kernel_iteration_range": "[1-2],[3-4]",
       "output_file": "out",
       "output_format": [
          "csv",
          "json"
       ],
       "truncate_kernels": true
    }
  ]
}
```

Here is a sample input.yaml file for counter collection:

```
jobs:
  - pmc: ["SQ_WAVES", "GRBM_COUNT", "GRBM_GUI_ACTIVE"]
  - pmc: ["FETCH_SIZE", "WRITE_SIZE"]
    kernel_include_regex: ".*_kernel"
    kernel_exclude_regex: "multiply"
    kernel_iteration_range: "[1-2],[3-4]"
    output_file: "out"
    output_format:
      - "csv"
      - "json"
    truncate_kernels: true
```

To supply the input file for kernel counter collection, use:

```
rocprofv3 -i input.yaml -- <application_path>
```

3.3.2 Counter collection using command line

You can also collect the desired counters by directly specifying them in the command line instead of using an input file.

To supply the counters in the command line, use:

```
rocprofv3 --pmc SQ_WAVES GRBM_COUNT GRBM_GUI_ACTIVE -- <application_path>
```

Note

- When specifying more than one counter, separate them using space or a comma.
- Job fails if the entire set of counters can't be collected in a single pass.

3.3.3 Extra counters

While the basic counters and derived metrics are available for collection by default, you can also define counters as per requirement. These user-defined counters with custom definitions are named extra counters.

You can define the extra counters in a YAML file as shown:

```
rocprofiler-sdk:
  counters-schema-version: 1
  counters:
    - name: GRBM_GUI_ACTIVE_SUM
      description: "Unit: cycles"
      properties: []
      definitions:
        - architectures:
            - gfx10
            - gfx1010
            - gfx1030
            - gfx1031
            - gfx1032
            - gfx11
            - gfx1100
            - gfx1101
            - gfx1102
            - gfx9
            - gfx906
            - gfx908
            - gfx90a
            - gfx942
          expression: reduce(GRBM_GUI_ACTIVE,max)*CU_NUM
    - name: CPC_CPC_STAT_BUSY
      description: CPC Busy.
      properties: []
      definitions:
        - architectures:
            - gfx940
            - gfx941
          block: CPC
          event: 25
```

Please note, the above sample uses the `CPC_CPC_STAT_BUSY` counter definition for the `gfx940` and `gfx941` architectures to demonstrate the YAML schema when counters have different architecture-specific definitions.

If this YAML is placed in a `extra_counters.yaml` file, to collect the extra counters defined in the `extra_counters.yaml` file, use the `-E` / `--extra-counters` option:

```
rocprofv3 -E <path-to-extra_counters.yaml> --pmc GRBM_GUI_ACTIVE_SUM --output-format csv␣
↪-- <application_path>
```

Where the option `--pmc` is used to specify the extra counters to be collected.

3.3.4 Kernel counter collection output

Using `rocprofv3` for counter collection using input file or command line generates a `./pmc_n/counter_collection.csv` file prefixed with the process ID. For each `pmc` row, a directory `pmc_n` containing a `counter_collection.csv` file is generated, where n = 1 for the first row and so on.

When using input file in JSON or YAML format, for each job, a directory `pass_n` containing a `counter_collection.csv` file is generated, where n = 1 for the first job and so on.

Each row of the CSV file is an instance of kernel execution. Here is a truncated version of the output file from `pmc_1`:

```
$ cat pmc_1/218_counter_collection.csv
```

Here are the contents of `counter_collection.csv` file:

Table 3.16: Counter collection

Correlation_	Dispatc	Agen	Que	Process	Thre	Grid	Kernel_	Kernel_	Workgrou	LDS	Scra	VGF	Accum	SGF	Cou	Cou	Star	End_Timestamp	Timestamp
1	1	Agen 1	1	1560	1560	1048	17	void addition_ float cons float cons int, int)	64	0	0	8	0	16	SQ_	1638	3200	3200	83232
2	2	Agen 1	1	1560	1560	1048	20	subtract_ float cons float cons int, int)	64	0	0	8	0	16	SQ_	1638	3200	3200	38794
3	3	Agen 1	1	1560	1560	1048	19	multiply_ float cons float cons int, int)	64	0	0	8	0	16	SQ_	1638	3200	3200	93025
4	4	Agen 1	1	1560	1560	1048	18	divide_ float cons float cons int, int)	64	0	0	12	0	16	SQ_	1638	3200	3200	27550

For the description of the fields in the output file, see *Output file fields*.

3.3.5 Iteration based counter multiplexing

Counter multiplexing allows a single run of the program to collect groups of counters. This is useful when the counters you want to collect exceed the hardware limits and you cannot run the program multiple times for collection.

This feature is available when using YAML (.yaml/.yml) or JSON (.json) input formats. Two new fields are introduced, `pmc_groups` and `pmc_group_interval`. The `pmc_groups` field is used to specify the groups of counters to be collected in each run. The `pmc_group_interval` field is used to specify the interval between each group of counters. Interval is per-device and increments per dispatch on the device (i.e. dispatch_id). When the interval is reached the next group is selected.

Here is a sample input.yaml file for specifying counter multiplexing:

```
jobs:
- pmc_groups: [["SQ_WAVES", "GRBM_COUNT"], ["GRBM_GUI_ACTIVE"]]
   pmc_group_interval: 4
```

This sample input will collect the first group of counters (`SQ_WAVES`, `GRBM_COUNT`) for the first 4 kernel executions on the device, then the second group of counters (`GRBM_GUI_ACTIVE`) for the next 4 kernel executions on the device, and so on.

An example of the interval period for this input is given below:

```
Device 1, <Kernel A>, Collect SQ_WAVES, GRBM_COUNT
Device 1, <Kernel A>, Collect SQ_WAVES, GRBM_COUNT
Device 1, <Kernel B>, Collect SQ_WAVES, GRBM_COUNT
Device 1, <Kernel C>, Collect SQ_WAVES, GRBM_COUNT
<Interval reached on Device 1, Swtiching Counters>
Device 1, <Kernel D>, Collect GRBM_GUI_ACTIVE
```

Here is the same sample in JSON format:

```
{
   "jobs": [
      {
            "pmc_groups": [["SQ_WAVES", "GRBM_COUNT"], ["GRBM_GUI_ACTIVE"]],
            "pmc_group_interval": 4
      }
   ]
}
```

3.4 Perfetto visualization

Perfetto is an open-source tracing tool that provides a detailed view of system performance. You can use Perfetto to visualize traces and performance counter data as explained in the following sections.

3.4.1 Perfetto visualization for traces

Perfetto helps you to visualize the collected traces in Perfetto viewer, which is a user-friendly interface that makes it easier to analyze and understand the performance characteristics of your application.

To generate a Perfetto trace file, use the `--output-format pftrace` option along with the desired tracing options. For example, to collect system traces and generate a Perfetto trace file, use:

```
rocprofv3 --sys-trace --output-format pftrace -- <application_path>
```

The generated Perfetto trace file can be opened in the Perfetto UI.

Figure 1: Generic perfetto visualization

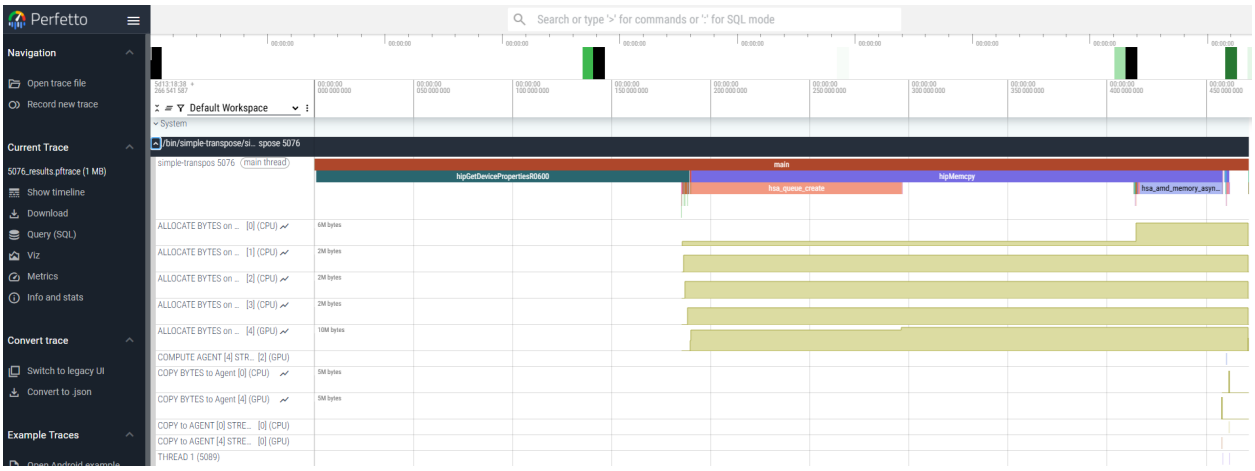
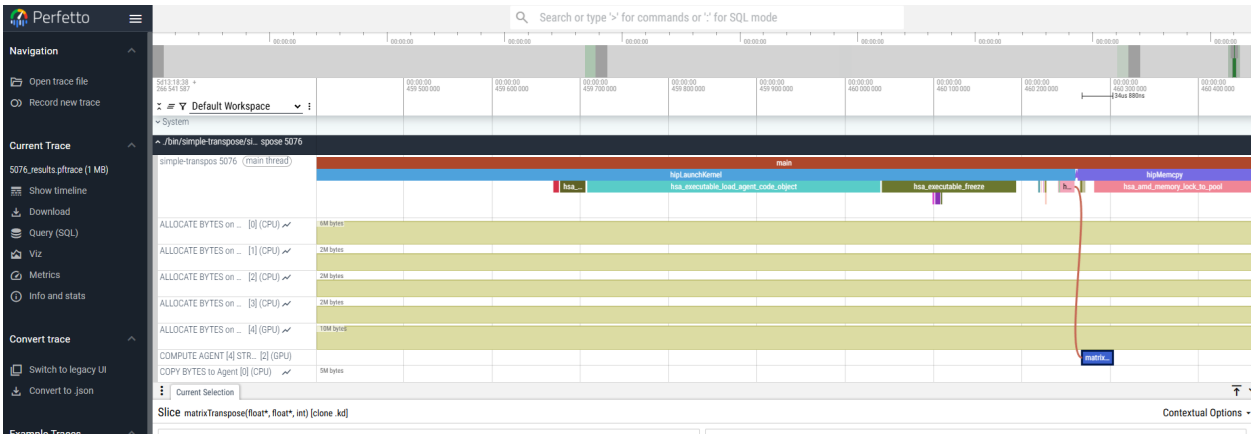


Figure 2: Visualization of ROCm flow data in Perfetto



3.4.2 Perfetto visualization for counter collection

When collecting performance counter data, you can visualize the counter tracks per agent in the Perfetto viewer by using the PFTrace output format. This helps you see how counter values change over time during kernel execution.

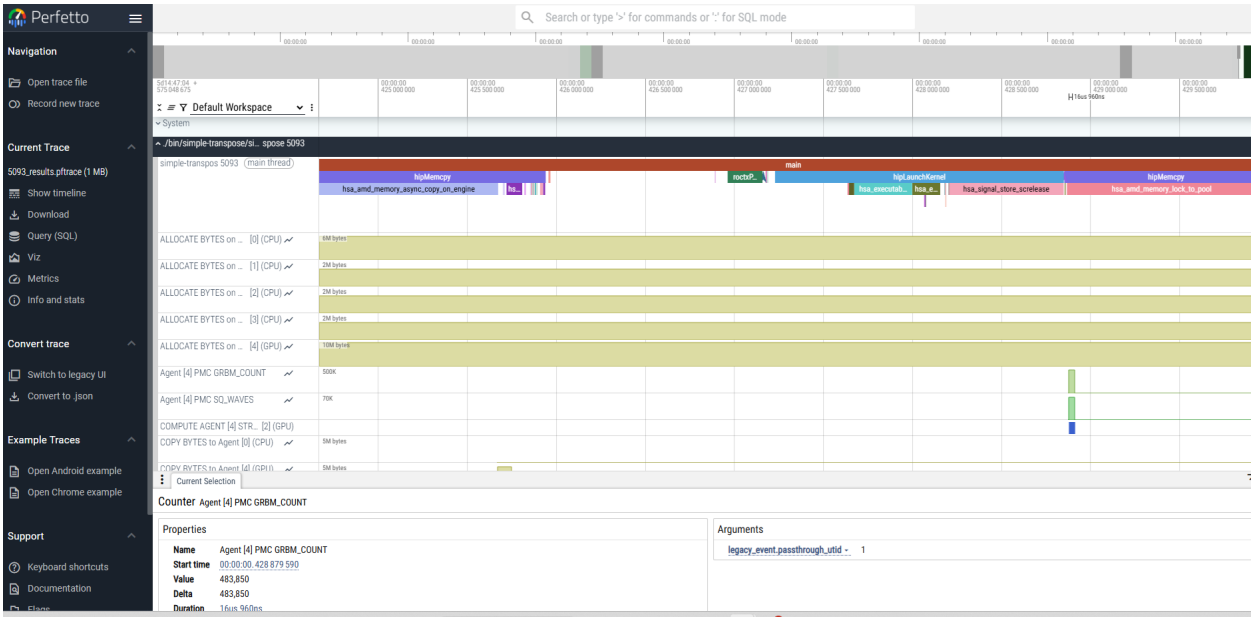
To generate a Perfetto trace file with counter data, use:

```
rocprofv3 --pmc SQ_WAVES GRBM_COUNT --output-format pftrace -- <application_path>
```

The generated Perfetto trace file can be opened in the Perfetto UI. In the viewer, performance counters will appear as counter tracks organized by agent, allowing you to visualize counter values changing over time alongside kernel executions and other traced activities.

You can also combine this with the system trace option to get a more comprehensive view of the system's performance. For example, you can use the following command to collect both system trace and performance counter data:

```
rocprofv3 --pmc SQ_WAVES GRBM_COUNT --sys-trace --output-format pftrace -- <application_
↪path>
```



3.5 Agent info

> **Note**
>
> All tracing and counter collection options generate an additional `agent_info.csv` file prefixed with the process ID.

The `agent_info.csv` file contains information about the CPU or GPU the kernel runs on.

```
$ cat 238_agent_info.csv

"Node_Id","Logical_Node_Id","Agent_Type","Cpu_Cores_Count","Simd_Count","Cpu_Core_Id_Base
↪","Simd_Id_Base","Max_Waves_Per_Simd","Lds_Size_In_Kb","Gds_Size_In_Kb","Num_Gws",
↪"Wave_Front_Size","Num_Xcc","Cu_Count","Array_Count","Num_Shader_Banks","Simd_Arrays_
↪Per_Engine","Cu_Per_Simd_Array","Simd_Per_Cu","Max_Slots_Scratch_Cu","Gfx_Target_
↪Version","Vendor_Id","Device_Id","Location_Id","Domain","Drm_Render_Minor","Num_Sdma_
↪Engines","Num_Sdma_Xgmi_Engines","Num_Sdma_Queues_Per_Engine","Num_Cp_Queues","Max_
↪Engine_Clk_Ccompute","Max_Engine_Clk_Fcompute","Sdma_Fw_Version","Fw_Version",
↪"Capability","Cu_Per_Engine","Max_Waves_Per_Cu","Family_Id","Workgroup_Max_Size","Grid_
↪Max_Size","Local_Mem_Size","Hive_Id","Gpu_Id","Workgroup_Max_Dim_X","Workgroup_Max_Dim_
↪Y","Workgroup_Max_Dim_Z","Grid_Max_Dim_X","Grid_Max_Dim_Y","Grid_Max_Dim_Z","Name",
↪"Vendor_Name","Product_Name","Model_Name"
0,0,"CPU",24,0,0,0,0,0,0,0,0,1,24,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3800,0,0,0,0,0,0,0,23,0,
↪0,0,0,0,0,0,0,0,0,0,"AMD Ryzen 9 3900X 12-Core Processor","CPU","AMD Ryzen 9 3900X 12-
↪Core Processor",""
1,1,"GPU",0,256,0,2147487744,10,64,0,64,64,1,64,4,4,1,16,4,32,90000,4098,26751,12032,0,
↪128,2,0,2,24,3800,1630,432,440,138420864,16,40,141,1024,4294967295,0,0,64700,1024,1024,
↪1024,4294967295,4294967295,4294967295,"gfx900","AMD","Radeon RX Vega","vega10"
```

3.6 Advanced options

`rocprofv3` provides the following miscellaneous functionalities for improved control and flexibility.

3.6.1 Agent index

The agent index is a unique identifier for each agent in the system. It is used to identify the agent in the output files. Since, each runtime or tool has an independent representation of the agent's indices, `rocprofv3` provides an option to configure the agent index in the output files.

- **absolute** == *node_id* - Absolute index of the agent, regardless of cgroups masking. This is a monotonically increasing number, which is incremented for every folder in `/sys/class/kfd/kfd/topology/nodes`. For example, Agent-0, Agent-2, Agent-4.
- **relative** == *logical_node_id* - Relative index of the agent accounting for cgroups masking. This is a monotonically increasing number, which is incremented for every folder in `/sys/class/kfd/kfd/topology/nodes/`, whose properties file is non-empty. For example, Agent-0, Agent-1, Agent-2.
- **type-relative** == *logical_node_type_id* - Relative index of the agent accounting for cgroups masking, where indexing starts at zero for each agent type. For example, CPU-0, GPU-0, GPU-1.

To set the agent index in the output files, use the `--agent-index` option. The default value is `relative`.

The following example shows how to set the agent index on a system with multiple GPUs and CPUs:

Here is the `rocm-smi` output:

```
========================================= ROCm System Management Interface␣
↪=========================================
================================================== Concise Info␣
↪==================================================
Device  Node  IDs              Temp        Power     Partitions          SCLK    MCLK    ␣
↪ Fan  Perf  PwrCap  VRAM%  GPU%
        (DID,     GUID)  (Junction)  (Socket)  (Mem, Compute, ID)
================================================================================================================
0       4     0x74a0,   50375  48.0°C      110.0W    NPS1, SPX, 0        98Mhz   1300Mhz␣
↪ 0%   auto  550.0W  0%     0%
1       5     0x74a0,   20890  53.0°C      113.0W    NPS1, SPX, 0        99Mhz   1200Mhz␣
↪ 0%   auto  550.0W  0%     0%
2       6     0x74a0,   44670  52.0°C      125.0W    NPS1, SPX, 0        100Mhz  1300Mhz␣
↪ 0%   auto  550.0W  0%     0%
3       7     0x74a0,   15139  47.0°C      115.0W    NPS1, SPX, 0        100Mhz  1300Mhz␣
↪ 0%   auto  550.0W  0%     0%
================================================================================================================
============================================= End of ROCm SMI Log␣
↪=============================================
```

To set the agent index to relative, use:

```
rocprofv3 --kernel-trace --agent-index=relative --output-format csv -- <application_path>
```

Here is the generated ouput file with `Agent_Id` as "Agent 7":

```
$ cat kernel_trace.csv

"Kind","Agent_Id","Queue_Id","Stream_Id","Thread_Id","Dispatch_Id","Kernel_Id","Kernel_
↪Name","Correlation_Id","Start_Timestamp","End_Timestamp","LDS_Block_Size","Scratch_Size
```

(continues on next page)

(continued from previous page)

```
↪","VGPR_Count","Accum_VGPR_Count","SGPR_Count","Workgroup_Size_X","Workgroup_Size_Y",
↪"Workgroup_Size_Z","Grid_Size_X","Grid_Size_Y","Grid_Size_Z"
"KERNEL_DISPATCH","Agent 7",17,26,847809,101,49,"void addition_kernel<float>(float*,␣
↪float const*, float const*, int, int)",101,1551401624448706,1551401624459226,0,0,8,0,
↪16,64,1,1,1024,1024,1
```

To set the agent index to type-relative, use:

```
rocprofv3 --kernel-trace --agent-index=type-relative --output-format csv -- <application_
↪path>
```

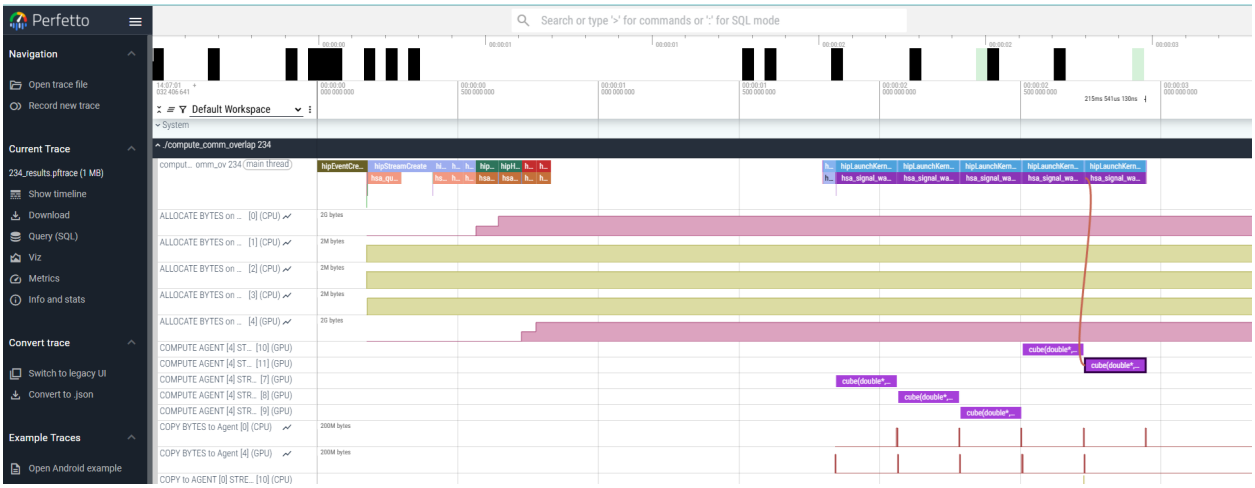
Here is the generated ouput file with `Agent_Id` as "GPU 3":

```
$ cat kernel_trace.csv

"Kind","Agent_Id","Queue_Id","Stream_Id","Thread_Id","Dispatch_Id","Kernel_Id","Kernel_
↪Name","Correlation_Id","Start_Timestamp","End_Timestamp","LDS_Block_Size","Scratch_Size
↪","VGPR_Count","Accum_VGPR_Count","SGPR_Count","Workgroup_Size_X","Workgroup_Size_Y",
↪"Workgroup_Size_Z","Grid_Size_X","Grid_Size_Y","Grid_Size_Z"
"KERNEL_DISPATCH","GPU 3",19,29,846827,113,49,"void addition_kernel<float>(float*, float␣
↪const*, float const*, int, int)",113,1551314943082302,1551314943092222,0,0,8,0,16,64,1,
↪1,1024,1024,1
```

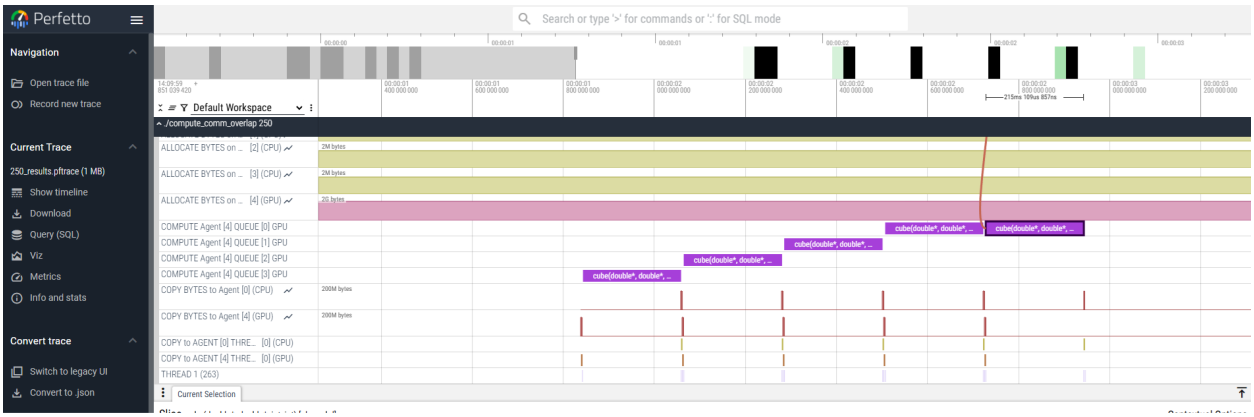
3.6.2 Group by queue

By default, `rocprofv3` shows the HIP streams to which the kernel and memory copy operations were submitted, when outputting a perfetto trace. Whereas, the `--group-by-queue` option displays the HSA queues to which these kernel and memory operations were submitted.




```
rocprofv3 -s --group-by-queue --output-format pftrace  -- <application_path>
```

The preceding command generates a `pftrace` file with the kernel and memory copy operations grouped into HSA queues instead of HIP streams.



3.7 Kernel naming and filtering

`rocprofv3` provides the following functionalities to configure the kernel name in the output file or to filter the kernels based on requirement.

3.7.1 Kernel name mangling

In `rocprofv3` output, by default, the kernel names are demangled to exclude the kernel arguments. This improves readability of the collected output.

To see the mangled kernel names, disable this feature by using the `--mangled-kernels` option.

Here is an example of kernel trace by default:

```
$ cat 123_kernel_trace.csv

"Kind","Agent_Id","Queue_Id","Stream_Id","Thread_Id","Dispatch_Id","Kernel_Id","Kernel_
↪Name","Correlation_Id","Start_Timestamp","End_Timestamp","LDS_Block_Size","Scratch_Size
↪","VGPR_Count","Accum_VGPR_Count","SGPR_Count","Workgroup_Size_X","Workgroup_Size_Y",
↪"Workgroup_Size_Z","Grid_Size_X","Grid_Size_Y","Grid_Size_Z"
"KERNEL_DISPATCH","Agent 4",1,1,852831,1,10,"void addition_kernel<float>(float*, float␣
↪const*, float const*, int, int)",1,1551874061244694,1551874061255734,0,0,8,0,16,64,1,1,
↪1024,1024,1
"KERNEL_DISPATCH","Agent 4",1,1,852831,2,13,"subtract_kernel(float*, float const*, float␣
↪const*, int, int)",2,1551874061259214,1551874061270254,0,0,8,0,16,64,1,1,1024,1024,1
"KERNEL_DISPATCH","Agent 4",1,1,852831,3,12,"multiply_kernel(float*, float const*, float␣
↪const*, int, int)",3,1551874061270254,1551874061279974,0,0,8,0,16,64,1,1,1024,1024,1
"KERNEL_DISPATCH","Agent 4",2,2,852831,8,11,"divide_kernel(float*, float const*, float␣
↪const*, int, int)",8,1551874061326294,1551874061335454,0,0,12,4,16,64,1,1,1024,1024,1
```

To disable kernel name demangling, use:

```
rocprofv3 --mangled-kernels --kernel-trace --output-format csv -- <application_path>
```

The preceding command generates the following `kernel_trace.csv` file with mangled kernel names:

```
$ cat 123_kernel_trace.csv

"Kind","Agent_Id","Queue_Id","Stream_Id","Thread_Id","Dispatch_Id","Kernel_Id","Kernel_
↪Name","Correlation_Id","Start_Timestamp","End_Timestamp","LDS_Block_Size","Scratch_Size
```

(continues on next page)

(continued from previous page)

```
↪","VGPR_Count","Accum_VGPR_Count","SGPR_Count","Workgroup_Size_X","Workgroup_Size_Y",
↪"Workgroup_Size_Z","Grid_Size_X","Grid_Size_Y","Grid_Size_Z"
"KERNEL_DISPATCH","Agent 4",1,1,850334,1,10,"_Z15addition_kernelIfEvPT_PKfS3_ii.kd",1,
↪1551636841670446,1551636841681606,0,0,8,0,16,64,1,1,1024,1024,1
"KERNEL_DISPATCH","Agent 4",1,1,850334,2,13,"_Z15subtract_kernelPfPKfS1_ii.kd",2,
↪1551636841686726,1551636841697606,0,0,8,0,16,64,1,1,1024,1024,1
"KERNEL_DISPATCH","Agent 4",1,1,850334,3,12,"_Z15multiply_kernelPfPKfS1_ii.kd",3,
↪1551636841701926,1551636841712806,0,0,8,0,16,64,1,1,1024,1024,1
"KERNEL_DISPATCH","Agent 4",2,2,850334,8,11,"_Z13divide_kernelPfPKfS1_ii.kd",8,
↪1551636841762926,1551636841774646,0,0,12,4,16,64,1,1,1024,1024,1
```

3.7.2 Kernel name truncation

The kernel name truncation feature allows you to limit the kernel name length in the output files. This is useful when dealing with long kernel names that can make the output files difficult to read.

To enable kernel name truncation, use the `--truncate-kernels` option:

```
rocprofv3 --truncate-kernels --kernel-trace --output-format csv -- <application_path>
```

The preceding command generates the following `kernel_trace.csv` file with truncated kernel names:

Table 3.17: Kernel trace truncated

Kin	Ag	Qu	Str	Thı	Dis pat	Ker nel	Ker nel	Co re- la- tior	Start_T	End_Ti	LD	Scı	VG	Ac· cur	SG	Wc gro	Wc gro	Wc gro	Gri	Gri	Gri	deSize_Z
KE NE	Ag 4	1	1	855	1	10	ad- di- tior	1	1552082	1552082	0	0	8	0	16	64	1	1	102	102	1	
KE NE	Ag 4	1	1	855	4	11	di- vid	4	1552082	1552082	0	0	12	4	16	64	1	1	102	102	1	
KE NE	Ag 4	1	1	855	3	12	mu ti- ply	3	1552082	1552082	0	0	8	0	16	64	1	1	102	102	1	
KE NE	Ag 4	1	1	855	2	13	sub trac	2	1552082	1552082	0	0	8	0	16	64	1	1	102	102	1	

3.7.3 Kernel filtering

Kernel filtering helps to include or exclude the kernels for profiling by specifying a filter using a regex string. You can also specify an iteration range for profiling the included kernels. If the iteration range is not provided, then all iterations of the included kernels are profiled.

Here is an input file with kernel filters:

```
$ cat input.yml
jobs:
    - pmc: [SQ_WAVES]
    kernel_include_regex: "divide"
    kernel_exclude_regex: ""
    kernel_iteration_range: "[1, 2, [5-8]]"
```

To collect counters for the kernels matching the filters specified in the preceding input file, run:

```
rocprofv3 -i input.yml --output-format csv -- <application_path>

$ cat pass_1/312_counter_collection.csv
"Correlation_Id","Dispatch_Id","Agent_Id","Queue_Id","Process_Id","Thread_Id","Grid_Size
↪","Kernel_Id","Kernel_Name","Workgroup_Size","LDS_Block_Size","Scratch_Size","VGPR_
↪Count","Accum_VGPR_Count","SGPR_Count","Counter_Name","Counter_Value","Start_Timestamp
↪","End_Timestamp"
1,1,4,1,225049,225049,1048576,10,"void addition_kernel<float>(float*, float const*,␣
↪float const*, int, int)",64,0,0,8,0,16,"SQ_WAVES",16384.000000,317095766765717,
↪317095766775957
2,2,4,1,225049,225049,1048576,13,"subtract_kernel(float*, float const*, float const*,␣
↪int, int)",64,0,0,8,0,16,"SQ_WAVES",16384.000000,317095767013157,317095767022957
3,3,4,1,225049,225049,1048576,11,"multiply_kernel(float*, float const*, float const*,␣
↪int, int)",64,0,0,8,0,16,"SQ_WAVES",16384.000000,317095767176998,317095767186678
4,4,4,1,225049,225049,1048576,12,"divide_kernel(float*, float const*, float const*, int,␣
↪int)",64,0,0,12,4,16,"SQ_WAVES",16384.000000,317095767380718,317095767390878
```

3.7.4 Kernel rename

The `roctxRangePush` and `roctxRangePop` also let you rename the enclosed kernel with the supplied message. In the legacy `rocprof`, this functionality was known as `--roctx-rename`.

See how to use `roctxRangePush` and `roctxRangePop` for renaming the enclosed kernel:

```
#include <rocprofiler-sdk-roctx/roctx.h>

roctxRangePush("HIP_Kernel-1");

// Launching kernel from host
hipLaunchKernelGGL(matrixTranspose, dim3(WIDTH/THREADS_PER_BLOCK_X, WIDTH/THREADS_PER_
↪BLOCK_Y), dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y), 0,0,gpuTransposeMatrix,
↪gpuMatrix, WIDTH);

// Memory transfer from device to host
roctxRangePush("hipMemCpy-DeviceToHost");

hipMemcpy(TransposeMatrix, gpuTransposeMatrix, NUM * sizeof(float),␣
↪hipMemcpyDeviceToHost);

roctxRangePop();  // for "hipMemcpy"
roctxRangePop();  // for "hipLaunchKernel"
roctxRangeStop(rangeId);
```

To rename the kernel, use:

```
rocprofv3 --marker-trace --kernel-rename --output-format csv -- <application_path>
```

The preceding command generates the following `marker-trace` file prefixed with the process ID:

```
 $ cat 210_marker_api_trace.csv
"Domain","Function","Process_Id","Thread_Id","Correlation_Id","Start_Timestamp","End_
↪Timestamp"
```

(continues on next page)

(continued from previous page)

```
"MARKER_CORE_API","roctxGetThreadId",315155,315155,2,58378843928406,58378843930247
"MARKER_CONTROL_API","roctxProfilerPause",315155,315155,3,58378844627184,58378844627502
"MARKER_CONTROL_API","roctxProfilerResume",315155,315155,4,58378844638601,58378844639267
"MARKER_CORE_API","pre-kernel-launch",315155,315155,5,58378844641787,58378844641787
"MARKER_CORE_API","post-kernel-launch",315155,315155,6,58378844936586,58378844936586
"MARKER_CORE_API","memCopyDth",315155,315155,7,58378844938371,58378851383270
"MARKER_CORE_API","HIP_Kernel-1",315155,315155,1,58378526575735,58378851384485
```

3.8 I/O control options

`rocprofv3` provides the following options to control the output.

3.8.1 Output prefix keys

Output prefix keys are useful in multiple use cases but are most helpful when dealing with multiple profiling runs or large MPI jobs. Here is the list of available keys:

String	Encoding
`%argv%`	Entire command-line condensed into a single string
`%argt%`	Similar to `%argv%` except basename of the first command-line argument
`%args%`	All command-line arguments condensed into a single string
`%tag%`	Basename of the first command-line argument
`%hostname%`	Hostname of the machine (`gethostname()`)
`%pid%`	Process identifier (`getpid()`)
`%ppid%`	Parent process identifier (`getppid()`)
`%pgid%`	Process group identifier (`getpgid(getpid())`)
`%psid%`	Process session identifier (`getsid(getpid())`)
`%psize%`	Number of sibling processes (reads `/proc/<PPID>/tasks/<PPID>/children`)
`%job%`	Value of `SLURM_JOB_ID` environment variable if exists, else 0
`%rank%`	Value of `SLURM_PROCID` environment variable if exists, else `MPI_Comm_rank`, or 0 for non-mpi
`%size%`	`MPI_Comm_size` or 1 for non-mpi
`%nid%`	`%rank%` if possible, otherwise `%pid%`
`%launch_time%`	Launch date and/or time according to `ROCPROF_TIME_FORMAT`
`%env{NAME}%`	Value of `NAME` environment variable (`getenv(NAME)`)
`$env{NAME}`	Alternative syntax to `%env{NAME}%`
`%p`	Shorthand for `%pid%`
`%j`	Shorthand for `%job%`
`%r`	Shorthand for `%rank%`
`%s`	Shorthand for `%size%`

3.8.2 Output directory

To specify the output directory, use `--output-directory` or `-d` option. If not specified, the default output path is `%hostname%/%pid%`.

```
rocprofv3 --hip-trace --output-directory output_dir --output-format csv -- <application_
↪path>
```

The preceding command generates an `output_dir/%hostname%/%pid%_hip_api_trace.csv` file.

The output directory option supports many placeholders such as:

- `%hostname%`: Machine host name
- `%pid%`: Process ID
- `%env{NAME}%`: Consistent with other output key formats (starts and ends with %)
- `$ENV{NAME}`: Similar to CMake
- `%q{NAME}%`: Compatibility with NVIDIA

To see the complete list, refer to *Output prefix keys*.

The following example shows how to use the output directory option with placeholders:

```
mpirun -n 2 rocprofv3 --hip-trace -d %h.%p.%env{OMPI_COMM_WORLD_RANK}% --output-format␣
↪csv -- <application_path>
```

The preceding command runs the application with `rocprofv3` and generates the trace file for each rank. The trace files are prefixed with hostname, process ID, and MPI rank.

Assuming the hostname as *ubuntu-latest* and the process IDs as 3000020 and 3000019, the output file names are:

```
ubuntu-latest.3000020.1/ubuntu-latest/3000020_agent_info.csv
ubuntu-latest.3000019.0/ubuntu-latest/3000019_agent_info.csv
ubuntu-latest.3000020.1/ubuntu-latest/3000020_hip_api_trace.csv
ubuntu-latest.3000019.0/ubuntu-latest/3000019_hip_api_trace.csv
```

3.8.3 Output file

To specify the output file name, use `--output-file` or `-o` option. If not specified, the output file is prefixed with the process ID by default.

```
rocprofv3 --hip-trace --output-file output --output-format csv -- <application_path>
```

The preceding command generates an `output_hip_api_trace.csv` file.

The output file name can also include placeholders such as `%hostname%` and `%pid%`. For example:

```
rocprofv3 --hip-trace --output-file %hostname%/%pid%_hip_api_trace --output-format csv --
↪ <application_path>
```

The preceding command generates an `%hostname%/%pid%_hip_api_trace.csv` file.

3.8.4 Collection period

The collection period is the time interval during which the profiling data is collected. You can specify the collection period using the `--collection-period` or `-P` option. You can also specify multiple configurations, each defined by a triplet in the format `start_delay:collection_time:repeat`.

The triplet is defined as follows:

- **Start delay time**: The time after which the profiling data collection starts.
- **Collection time**: The time period during which the profiling data is collected.
- **Repeat**: The number of times the cycle is repeated. A repeat value of 0 indicates that the cycle will repeat indefinitely.

```
rocprofv3 --collection-period 5:1:1 --hip-trace -- <application_path>
```

The preceding command collects the profiling data for 1 second, starting 5 seconds after the application starts, and this cycle will be repeated once.

The collection period can be specified in different units, such as seconds, milliseconds, microseconds, and nanoseconds. The default unit is "seconds". You can change the unit using the `--collection-period-unit` option.

The available time units are:

–collection-period-unit: *hour*, *min*, *sec*, *msec*, *usec*, *nsec*

To specify the time unit as milliseconds, use:

```
rocprofv3 --collection-period 5:1:0 --collection-period-unit msec --hip-trace --
↪<application_path>
```

3.8.5 Perfetto-specific options

The following options are specific to Perfetto tracing and are used to control the Perfetto data collection behavior:

- **–perfetto-buffer-fill-policy {discard,ring_buffer}**: Policy for handling new records when Perfetto reaches the buffer limit.
  - **RING_BUFFER (default)**: The buffer behaves like a ring buffer. Once full, writes wrap over and replace the oldest trace data in the buffer.
  - **DISCARD**: The buffer stops accepting data once full. Further write attempts are dropped.
- **–perfetto-buffer-size KB**: The buffer size for Perfetto output in KB. Default: 1 GB. If set, stops the tracing session after N bytes have been written. Used to cap the trace size.
- **–perfetto-backend {inprocess,system}**: Perfetto data collection backend. `system` mode requires starting traced and perfetto daemons. By default Perfetto keeps the full trace buffers in memory.
- **–perfetto-shmem-size-hint KB**: Perfetto shared memory size hint in KB. Default: 64 KB. This option gives you control over shared memory buffer sizing. You can tweak this option to avoid data losses when data is produced at a higher rate.

3.9 Output file fields

The following table lists the various fields or the columns in the output CSV files generated for application tracing and kernel counter collection:

Table 3.18: output file fields

Field	Description
Agent_Id	GPU identifier to which the kernel was submitted.
Correlation_Id	Unique identifier for correlation between HIP and HSA async calls during activity tracing.
Start_Timesta	Begin time in nanoseconds (ns) when the kernel begins execution.
End_Timestai	End time in ns when the kernel finishes execution.
Queue_Id	ROCm queue unique identifier to which the kernel was submitted.
Stream_Id	Identifies HIP stream ID to which kernel or memory copy operation was submitted. Defaults to 0 if the hip-stream-display option is not enabled
Private_Segmen	The amount of memory required in bytes for the combined private, spill, and arg segments for a work item.
Group_Segm	The group segment memory required by a workgroup in bytes. This does not include any dynamically allocated group segment memory that may be added when the kernel is dispatched.
Workgroup_Size	Size of the workgroup as declared by the compute shader.
Workgroup_Size_n	Size of the workgroup in the nth dimension as declared by the compute shader, where n = X, Y, or Z.
Grid_Size	Number of thread blocks required to launch the kernel.
Grid_Size_n	Number of thread blocks in the nth dimension required to launch the kernel, where n = X, Y, or Z.
LDS_Block_S	Thread block size for the kernel's Local Data Share (LDS) memory.
Scratch_Size	Kernel's scratch memory size.
SGPR_Count	Kernel's Scalar General Purpose Register (SGPR) count.
VGPR_Count	Kernel's Architected Vector General Purpose Register (VGPR) count.
Accum_VGPR_(	Kernel's Accumulation Vector General Purpose Register (Accum_VGPR/AGPR) count.

3.10 Output formats

- rocpd (SQLite3 Database (Default))
- CSV
- JSON (Custom format for programmatic analysis only)
- PFTrace (Perfetto trace for visualization with Perfetto)
- OTF2 (Open Trace Format for visualization with compatible third-party tools)

The default output format is `rocpd`. To know more about the rocpd format, see *Using rocpd Output Format*. To specify the particular output format, use the `--output-format` option followed by the desired format.

```
rocprofv3 -i input.txt --output-format json -- <application_path>
```

Format selection is case-insensitive and multiple output formats are supported. While `--output-format json` exclusively enables JSON output, `--output-format csv json pftrace otf2, rocpd` enables all four output formats for the run.

For PFTrace trace visualization, use the PFTrace format and open the trace in ui.perfetto.dev.

For OTF2 trace visualization, open the trace in vampir.eu or any supported visualizer.

Note

For large trace files (> 10GB), it's recommended to use OTF2 format.

3.10.1 JSON output schema

`rocprofv3` supports a custom JSON output format designed for programmatic analysis and **NOT** for visualization. The schema is optimized for size while factoring in usability.

Note

Perfetto UI doesn't accept this JSON output format.

To generate the JSON output, use `--output-format json` command-line option.

3.10.1.1 Properties

Here are the properties of the JSON output schema:

- **rocprofiler-sdk-tool *(array)*: rocprofv3 data per process (each element represents a process).**
  - **Items *(object)*: Data for rocprofv3.**
    - **metadata *(object, required)*: Metadata related to the profiler session.**
      - **pid** *(integer, required)*: Process ID.
      - **init_time** *(integer, required)*: Initialization time in nanoseconds.
      - **fini_time** *(integer, required)*: Finalization time in nanoseconds.
    - **agents *(array, required)*: List of agents.**
      - **Items *(object)*: Data for an agent.**

        size *(integer, required)*: Size of the agent data.

        id *(object, required)*: Identifier for the agent.

        handle *(integer, required)*: Handle for the agent.

        type *(integer, required)*: Type of the agent.

        cpu_cores_count *(integer)*: Number of CPU cores.

        simd_count *(integer)*: Number of SIMD units.

        mem_banks_count *(integer)*: Number of memory banks.

        caches_count *(integer)*: Number of caches.

        io_links_count *(integer)*: Number of I/O links.

        cpu_core_id_base *(integer)*: Base ID for CPU cores.

        simd_id_base *(integer)*: Base ID for SIMD units.

        max_waves_per_simd *(integer)*: Maximum waves per SIMD.

        lds_size_in_kb *(integer)*: Size of LDS in KB.

        gds_size_in_kb *(integer)*: Size of GDS in KB.

        num_gws *(integer)*: Number of GWS (global work size).

        wave_front_size *(integer)*: Size of the wave front.

num_xcc *(integer)*: Number of XCC (execution compute units).

cu_count *(integer)*: Number of compute units (CUs).

array_count *(integer)*: Number of arrays.

num_shader_banks *(integer)*: Number of shader banks.

simd_arrays_per_engine *(integer)*: SIMD arrays per engine.

cu_per_simd_array *(integer)*: CUs per SIMD array.

simd_per_cu *(integer)*: SIMDs per CU.

max_slots_scratch_cu *(integer)*: Maximum slots for scratch CU.

gfx_target_version *(integer)*: GFX target version.

vendor_id *(integer)*: Vendor ID.

device_id *(integer)*: Device ID.

location_id *(integer)*: Location ID.

domain *(integer)*: Domain identifier.

drm_render_minor *(integer)*: DRM render minor version.

num_sdma_engines *(integer)*: Number of SDMA engines.

num_sdma_xgmi_engines *(integer)*: Number of SDMA XGMI engines.

num_sdma_queues_per_engine *(integer)*: Number of SDMA queues per engine.

num_cp_queues *(integer)*: Number of CP queues.

max_engine_clk_ccompute *(integer)*: Maximum engine clock for compute.

max_engine_clk_fcompute *(integer)*: Maximum engine clock for F compute.

sdma_fw_version ***(object)*****: SDMA firmware version.**

  uCodeSDMA *(integer, required)*: SDMA microcode version.

  uCodeRes *(integer, required)*: Reserved microcode version.

fw_version ***(object)*****: Firmware version.**

  uCode *(integer, required)*: Microcode version.

  Major *(integer, required)*: Major version.

  Minor *(integer, required)*: Minor version.

  Stepping *(integer, required)*: Stepping version.

capability ***(object, required)*****: Agent capability flags.**

  HotPluggable *(integer, required)*: Hot pluggable capability.

  HSAMMUPresent *(integer, required)*: HSAMMU present capability.

  SharedWithGraphics *(integer, required)*: Shared with graphics capability.

  QueueSizePowerOfTwo *(integer, required)*: Queue size is power of two.

  QueueSize32bit *(integer, required)*: Queue size is 32-bit.

  QueueIdleEvent *(integer, required)*: Queue idle event.

  VALimit *(integer, required)*: VA limit.

WatchPointsSupported *(integer, required)*: Watch points supported.

WatchPointsTotalBits *(integer, required)*: Total bits for watch points.

DoorbellType *(integer, required)*: Doorbell type.

AQLQueueDoubleMap *(integer, required)*: AQL queue double map.

DebugTrapSupported *(integer, required)*: Debug trap supported.

WaveLaunchTrapOverrideSupported *(integer, required)*: Wave launch trap override supported.

WaveLaunchModeSupported *(integer, required)*: Wave launch mode supported.

PreciseMemoryOperationsSupported *(integer, required)*: Precise memory operations supported.

DEPRECATED_SRAM_EDCSupport *(integer, required)*: Deprecated SRAM EDC support.

Mem_EDCSupport *(integer, required)*: Memory EDC support.

RASEventNotify *(integer, required)*: RAS event notify.

ASICRevision *(integer, required)*: ASIC revision.

SRAM_EDCSupport *(integer, required)*: SRAM EDC support.

SVMAPISupported *(integer, required)*: SVM API supported.

CoherentHostAccess *(integer, required)*: Coherent host access.

DebugSupportedFirmware *(integer, required)*: Debug supported firmware.

Reserved *(integer, required)*: Reserved field.

* **counters *(array, required)*: Array of counter objects.**
  - **Items *(object)***

    agent_id *(object, required)*: Agent ID information.

    handle *(integer, required)*: Handle of the agent.

    id *(object, required)*: Counter ID information.

    handle *(integer, required)*: Handle of the counter.

    is_constant *(integer, required)*: Indicator if the counter value is constant.

    is_derived *(integer, required)*: Indicator if the counter value is derived.

    name *(string, required)*: Name of the counter.

    description *(string, required)*: Description of the counter.

    block *(string, required)*: Block information of the counter.

    expression *(string, required)*: Expression of the counter.

    dimension_ids *(array, required)*: Array of dimension IDs.

    Items *(integer)*: Dimension ID.

* **strings *(object, required)*: String records.**
  - **callback_records *(array)*: Callback records.**

    Items *(object)*

kind *(string, required)*: Kind of the record.

operations *(array, required)*: Array of operations.

Items *(string)*: Operation.

- **buffer_records *(array)*: Buffer records.**

  Items *(object)*

  kind *(string, required)*: Kind of the record.

  operations *(array, required)*: Array of operations.

  Items *(string)*: Operation.

- **marker_api *(array)*: Marker API records.**

  Items *(object)*

  key *(integer, required)*: Key of the record.

  value *(string, required)*: Value of the record.

- **counters *(object)*: Counter records.**

  dimension_ids *(array, required)*: Array of dimension IDs.

  Items *(object)*

  id *(integer, required)*: Dimension ID.

  instance_size *(integer, required)*: Size of the instance.

  name *(string, required)*: Name of the dimension.

- **pc_sample_instructions** *(array)*: Array of decoded instructions matching sampled PCs from pc_sample_host_trap section.

- **pc_sample_comments** *(array)*: Comments matching assembly instructions from pc_sample_instructions array. If debug symbols are available, comments provide instructions to source-line mapping. Otherwise, a comment is an empty string.

* **code_objects *(array, required)*: Code object records.**

  - **Items *(object)***

    size *(integer, required)*: Size of the code object.

    code_object_id *(integer, required)*: ID of the code object.

    rocp_agent *(object, required)*: ROCP agent information.

    handle *(integer, required)*: Handle of the ROCP agent.

    hsa_agent *(object, required)*: HSA agent information.

    handle *(integer, required)*: Handle of the HSA agent.

    uri *(string, required)*: URI of the code object.

    load_base *(integer, required)*: Base address for loading.

    load_size *(integer, required)*: Size for loading.

    load_delta *(integer, required)*: Delta for loading.

    storage_type *(integer, required)*: Type of storage.

    memory_base *(integer, required)*: Base address for memory.

memory_size *(integer, required)*: Size of memory.

* **kernel_symbols** ***(array, required)*****: Kernel symbol records.**
  - **Items** ***(object)***

    size *(integer, required)*: Size of the kernel symbol.

    kernel_id *(integer, required)*: ID of the kernel.

    code_object_id *(integer, required)*: ID of the code object.

    kernel_name *(string, required)*: Name of the kernel.

    kernel_object *(integer, required)*: Object of the kernel.

    kernarg_segment_size *(integer, required)*: Size of the kernarg segment.

    kernarg_segment_alignment *(integer, required)*: Alignment of the kernarg segment.

    group_segment_size *(integer, required)*: Size of the group segment.

    private_segment_size *(integer, required)*: Size of the private segment.

    formatted_kernel_name *(string, required)*: Formatted name of the kernel.

    demangled_kernel_name *(string, required)*: Demangled name of the kernel.

    truncated_kernel_name *(string, required)*: Truncated name of the kernel.

* **callback_records** ***(object, required)*****: Callback record details.**
  - **counter_collection** ***(array)*****: Counter collection records.**

    Items ***(object)***

      dispatch_data ***(object, required)*****: Dispatch data details.**

        size *(integer, required)*: Size of the dispatch data.

        correlation_id ***(object, required)*****: Correlation ID information.**

          internal *(integer, required)*: Internal correlation ID.

          external *(integer, required)*: External correlation ID.

        dispatch_info ***(object, required)*****: Dispatch information details.**

          size *(integer, required)*: Size of the dispatch information.

          agent_id ***(object, required)*****: Agent ID information.**

            handle *(integer, required)*: Handle of the agent.

          queue_id ***(object, required)*****: Queue ID information.**

            handle *(integer, required)*: Handle of the queue.

          kernel_id *(integer, required)*: ID of the kernel.

          dispatch_id *(integer, required)*: ID of the dispatch.

          private_segment_size *(integer, required)*: Size of the private segment.

          group_segment_size *(integer, required)*: Size of the group segment.

          workgroup_size ***(object, required)*****: Workgroup size information.**

            x *(integer, required)*: X dimension.

            y *(integer, required)*: Y dimension.

z *(integer, required)*: Z dimension.

grid_size *(object, required)*: Grid size information.

x *(integer, required)*: X dimension.

y *(integer, required)*: Y dimension.

z *(integer, required)*: Z dimension.

records *(array, required)*: Records.

Items *(object)*

counter_id *(object, required)*: Counter ID information.

handle *(integer, required)*: Handle of the counter.

value *(number, required)*: Value of the counter.

thread_id *(integer, required)*: Thread ID.

arch_vgpr_count *(integer, required)*: Count of Architected VGPRs.

accum_vgpr_count *(integer, required)*: Count of Accumulation VGPRs.

sgpr_count *(integer, required)*: Count of SGPRs.

lds_block_size_v *(integer, required)*: Size of LDS block.

* **pc_sample_host_trap *(array)*: Host Trap PC Sampling records.**
  - **Items *(object)***

hw_id *(object)*: Describes hardware part on which sampled wave was running.

chiplet *(integer)*: Chiplet index.

wave_id *(integer)*: Wave slot index.

simd_id *(integer)*: SIMD index.

pipe_id *(integer)*: Pipe index.

cu_or_wgp_id *(integer)*: Index of compute unit or workgroup processer.

shader_array_id *(integer)*: Shader array index.

shader_engine_id *(integer)*: Shader engine index.

workgroup_id *(integer)*: Workgroup position in the 3D.

vm_id *(integer)*: Virtual memory ID.

queue_id *(integer)*: Queue id.

microengine_id *(integer)*: ACE (microengine) index.

pc *(object)*: Encapsulates information about sampled PC. - **code_object_id** *(integer)*: Code object id. - **code_object_offset** *(integer)*: Offset within the object if the latter is known. Otherwise, virtual address of the PC.

exec_mask *(integer)*: Execution mask indicating active SIMD lanes of sampled wave.

timestamp *(integer)*: Timestamp.

dispatch_id *(integer)*: Dispatch id.

correlation_id *(object)*: Correlation ID information. - **internal** *(integer)*: Internal correlation ID. - **external** *(integer)*: External correlation ID.

rocprofiler_dim3_t *(object)*: Position of the workgroup in 3D grid.

x *(integer)*: Dimension x.

y *(integer)*: Dimension y.

z *(integer)*: Dimension z.

wave_in_group *(integer)*: Wave position within the workgroup (0-31).

* **buffer_records *(object, required)*: Buffer record details.**
  - **kernel_dispatch *(array)*: Kernel dispatch records.**

    Items *(object)*

    size *(integer, required)*: Size of the dispatch.

    kind *(integer, required)*: Kind of the dispatch.

    operation *(integer, required)*: Operation of the dispatch.

    thread_id *(integer, required)*: Thread ID.

    correlation_id *(object, required)*: Correlation ID information.

    internal *(integer, required)*: Internal correlation ID.

    external *(integer, required)*: External correlation ID.

    start_timestamp *(integer, required)*: Start timestamp.

    end_timestamp *(integer, required)*: End timestamp.

    dispatch_info *(object, required)*: Dispatch information details.

    size *(integer, required)*: Size of the dispatch information.

    agent_id *(object, required)*: Agent ID information.

    handle *(integer, required)*: Handle of the agent.

    queue_id *(object, required)*: Queue ID information.

    handle *(integer, required)*: Handle of the queue.

    kernel_id *(integer, required)*: ID of the kernel.

    dispatch_id *(integer, required)*: ID of the dispatch.

    private_segment_size *(integer, required)*: Size of the private segment.

    group_segment_size *(integer, required)*: Size of the group segment.

    workgroup_size *(object, required)*: Workgroup size information.

    x *(integer, required)*: X dimension.

    y *(integer, required)*: Y dimension.

    z *(integer, required)*: Z dimension.

    grid_size *(object, required)*: Grid size information.

    x *(integer, required)*: X dimension.

    y *(integer, required)*: Y dimension.

    z *(integer, required)*: Z dimension.

  - **hip_api *(array)*: HIP API records.**

Items ***(object)***

size *(integer, required)*: Size of the HIP API record.

kind *(integer, required)*: Kind of the HIP API.

operation *(integer, required)*: Operation of the HIP API.

correlation_id ***(object, required)*****: Correlation ID information.**

internal *(integer, required)*: Internal correlation ID.

external *(integer, required)*: External correlation ID.

start_timestamp *(integer, required)*: Start timestamp.

end_timestamp *(integer, required)*: End timestamp.

thread_id *(integer, required)*: Thread ID.

· **hsa_api** ***(array)*****: HSA API records.**

Items ***(object)***

size *(integer, required)*: Size of the HSA API record.

kind *(integer, required)*: Kind of the HSA API.

operation *(integer, required)*: Operation of the HSA API.

correlation_id ***(object, required)*****: Correlation ID information.**

internal *(integer, required)*: Internal correlation ID.

external *(integer, required)*: External correlation ID.

start_timestamp *(integer, required)*: Start timestamp.

end_timestamp *(integer, required)*: End timestamp.

thread_id *(integer, required)*: Thread ID.

· **marker_api** ***(array)*****: Marker (ROCTx) API records.**

Items ***(object)***

size *(integer, required)*: Size of the Marker API record.

kind *(integer, required)*: Kind of the Marker API.

operation *(integer, required)*: Operation of the Marker API.

correlation_id ***(object, required)*****: Correlation ID information.**

internal *(integer, required)*: Internal correlation ID.

external *(integer, required)*: External correlation ID.

start_timestamp *(integer, required)*: Start timestamp.

end_timestamp *(integer, required)*: End timestamp.

thread_id *(integer, required)*: Thread ID.

· **memory_copy** ***(array)*****: Async memory copy records.**

Items ***(object)***

size *(integer, required)*: Size of the Marker API record.

kind *(integer, required)*: Kind of the Marker API.

operation *(integer, required)*: Operation of the Marker API.

correlation_id ***(object, required)*****: Correlation ID information.**

internal *(integer, required)*: Internal correlation ID.

external *(integer, required)*: External correlation ID.

start_timestamp *(integer, required)*: Start timestamp.

end_timestamp *(integer, required)*: End timestamp.

thread_id *(integer, required)*: Thread ID.

dst_agent_id ***(object, required)*****: Destination Agent ID.**

handle *(integer, required)*: Handle of the agent.

src_agent_id ***(object, required)*****: Source Agent ID.**

handle *(integer, required)*: Handle of the agent.

bytes *(integer, required)*: Bytes copied.

· **memory_allocation** ***(array)*****: Memory allocation records.**

Items ***(object)***

size *(integer, required)*: Size of the Marker API record.

kind *(integer, required)*: Kind of the Marker API.

operation *(integer, required)*: Operation of the Marker API.

correlation_id ***(object, required)*****: Correlation ID information.**

internal *(integer, required)*: Internal correlation ID.

external *(integer, required)*: External correlation ID.

start_timestamp *(integer, required)*: Start timestamp.

end_timestamp *(integer, required)*: End timestamp.

thread_id *(integer, required)*: Thread ID.

agent_id ***(object, required)*****: Agent ID.**

handle *(integer, required)*: Handle of the agent.

address *(string, required)*: Starting address of allocation.

allocation_size *(integer, required)*: Size of allocation.

· **rocDecode_api** ***(array)*****: rocDecode API records.**

Items ***(object)***

size *(integer, required)*: Size of the rocDecode API record.

kind *(integer, required)*: Kind of the rocDecode API.

operation *(integer, required)*: Operation of the rocDecode API.

correlation_id ***(object, required)*****: Correlation ID information.**

internal *(integer, required)*: Internal correlation ID.

external *(integer, required)*: External correlation ID.

start_timestamp *(integer, required)*: Start timestamp.

end_timestamp *(integer, required)*: End timestamp.

thread_id *(integer, required)*: Thread ID.

CHAPTER

FOUR

USING ROCPD OUTPUT FORMAT

`rocprofv3` supports the following output formats:

- **rocpd** (SQLite3 Database, Default)
- **CSV**
- **JSON** (Custom format for programmatic analysis only)
- **PFTrace** (Perfetto trace for visualization with Perfetto)
- **OTF2** (Open Trace Format for visualization with compatible third-party tools)

The `rocpd` output format is the default for `rocprofv3`. It stores profiling results in a SQLite3 database, providing a structured and efficient way to analyze and post-process profiling data. This format allows users to query and manipulate profiling data using SQL, making it easy to extract specific information or perform complex analyses.

4.1 Features

- **Rich Data Model**: Stores all collected profiling data, including traces, counters, and metadata, in a single *.db* (SQLite3) file.
- **Programmatic Access**: Can be queried using standard SQL tools or libraries (e.g., *sqlite3* CLI, Python's *sqlite3* module).
- **Post-Processing**: Enables advanced analysis and visualization using custom scripts or third-party tools that support SQLite3.

4.2 Generating rocpd Output

To generate output in rocpd format, simply use:

```
rocprofv3 --hip-trace -- <application>
```

Or use the `--output-format` option with `rocpd`:

```
rocprofv3 --hip-trace --output-format rocpd -- <application>
```

The output will be saved as `%hostname%/%pid%_results.db`, where `%hostname%` is the name of the host machine and `%pid%` is the process ID of the application being profiled.

4.3 Converting rocpd to Other Formats

The `rocpd` output format can be converted to other formats for further analysis or visualization. First, ensure the `rocpd` Python module is available in your environment:

```
export PYTHONPATH=<install-path>/lib/pythonX.Y/site-packages:$PYTHONPATH
```

where `<install-path>` is the ROCm installation path (usually `/opt/rocm-<major.minor.patch>`), and `X.Y` is your Python version.

Once the `rocpd` module is available, use the `rocpd convert` command to convert the output to other formats.

Convert to CSV format:

```
python3 -m rocpd convert -i <input-file>.db --output-format csv
```

The converted CSV will be saved as `rocpd-output-data/out_hip_api_trace.csv` in the current working directory.

Convert to OTF2 format:

```
python3 -m rocpd convert -i <input-file>.db --output-format otf2
```

Convert to PFTrace format:

```
python3 -m rocpd convert -i <input-file>.db --output-format pftrace
```

4.4 rocpd convert Command-Line Options

```
usage: rocpd convert [-h] -i INPUT [INPUT ...] -f {csv,pftrace,otf2} [{csv,pftrace,otf2}␣
↪...]
                     [-o OUTPUT_FILE] [-d OUTPUT_PATH] [--kernel-rename]
                     [--agent-index-value {absolute,relative,type-relative}]
                     [--perfetto-backend {inprocess,system}]
                     [--perfetto-buffer-fill-policy {discard,ring_buffer}]
                     [--perfetto-buffer-size KB] [--perfetto-shmem-size-hint KB]
                     [--group-by-queue]
                     [--start START | --start-marker START_MARKER]
                     [--end END | --end-marker END_MARKER]
                     [--inclusive INCLUSIVE]
```

4.4.1 Options

Required Arguments:

- `-i INPUT [INPUT ...]`, `--input INPUT [INPUT ...]` Input path and filename to one or more database(s), separated by spaces.
- `-f {csv,pftrace,otf2} [{csv,pftrace,otf2} ...]`, `--output-format {csv,pftrace,otf2} [{csv,pftrace,otf2} ...]` Specify one or more output formats. Supported: `csv`, `pftrace`, `otf2`.

I/O Options:

- `-o OUTPUT_FILE`, `--output-file OUTPUT_FILE` Sets the base output file name (default: `out`).
- `-d OUTPUT_PATH`, `--output-path OUTPUT_PATH` Sets the output directory (default: `./rocpd-output-data`).

Kernel Naming Options:

- `--kernel-rename` Use ROCTx marker names instead of kernel names.

Generic Options:

- `--agent-index-value {absolute,relative,type-relative}` Device identification format in output:
  - `absolute`: Uses node_id (e.g., Agent-0, Agent-2, Agent-4), ignoring cgroups.
  - `relative`: Uses logical_node_id (e.g., Agent-0, Agent-1, Agent-2), considering cgroups. *(Default)*
  - `type-relative`: Uses logical_node_type_id (e.g., CPU-0, GPU-0, GPU-1), numbering resets for each device type.

Perfetto Trace (pftrace) Options:

- `--perfetto-backend {inprocess,system}` Perfetto data collection backend. `system` mode requires running `traced` and `perfetto` daemons (default: `inprocess`).
- `--perfetto-buffer-fill-policy {discard,ring_buffer}` Policy for handling new records when buffer is full (default: `discard`).
- `--perfetto-buffer-size KB` Buffer size for perfetto output in KB (default: 1 GB).
- `--perfetto-shmem-size-hint KB` Perfetto shared memory size hint in KB (default: 64 KB).
- `--group-by-queue` Display HIP streams that kernels and memory copy operations are submitted to, rather than HSA queues.

Time Window Options:

- `--start START` Start time as percentage or nanoseconds from trace file (e.g., `50%` or `781470909013049`).
- `--start-marker START_MARKER` Named marker event to use as window start point.
- `--end END` End time as percentage or nanoseconds from trace file (e.g., `75%` or `3543724246381057`).
- `--end-marker END_MARKER` Named marker event to use as window end point.
- `--inclusive INCLUSIVE` `True`: include events if START or END in window; `False`: only if BOTH in window (default: `True`).

Help:

- `-h`, `--help` Show help message and exit.

4.5 Examples

Convert one database to Perfetto trace:

```
python3 -m rocpd convert -i db1.db --output-format pftrace
```

Convert two databases to Perfetto trace, set output path and filename, and limit to last 70% of trace:

```
python3 -m rocpd convert -i db1.db db2.db --output-format pftrace -d "./output/" -o
↪"twoFileTraces" --start 30% --end 100%
```

Convert six databases to CSV and Perfetto trace formats:

```
python3 -m rocpd convert -i db{0..5}.db --output-format csv pftrace -d "~/output_folder/
↪" -o "sixFileTraces"
```

Convert two databases to CSV, OTF2, and Perfetto trace formats:

```
python3 -m rocpd convert -i db{3,4}.db --output-format csv otf2 pftrace
```

CHAPTER

FIVE

USING ROCPROFV3-AVAIL

`rocprofv3-avail` is a CLI tool that helps you to query the features supported by the hardware and Rocprofiler SDK.

The following sections demonstrate the use of `rocprofv3-avail` for querying features using various command-line options.

`rocprofv3-avail` is installed with ROCm under `/opt/rocm/bin`. To use the tool from anywhere in the system, export `PATH` variable:

```
export PATH=$PATH:/opt/rocm/bin
```

5.1 Command-line options

The following table lists `rocprofv3-avail` command-line options categorized according to their purpose.

Table 5.1: rocprofv3-avail options

Purpose	Option	Description
avail-aptions commands		

	`info`
	`list`
	`pmc-check`
		Info options for detailed information of counters, agents, and pc-sampling configurations.
		List options for hw counters, agents and pc-sampling support".
		Checking counters collection support on agents.

5.1.1 Available Hardware Counters

```
rocprofv3-avail -d 0
```

The preceding command selects a device with logical node type id as 0 in the node. The option is applied to further sub commands and options

```
rocprofv3-avail list
```

The preceding command generates an output listing agents and hardware counters

```
rocprofv3-avail list --agent
```

The preceding command generates an output listing basic info for all agents, if used with -d only basic info for device -d is listed.

```
rocprofv3-avail list --pmc
```

The preceding command generates an output listing counters for all agents, if used with -d only counters on the the -d device is listed

```
rocprofv3-avail list --pc-sampling
```

The preceding command generates an output listing agents that supports any kind of PC Sampling. -d option is not applicable here. .. code-block:: bash

    rocprofv3-avail info

The preceding command generates an output with agent information and listing all counters supported on each

```
rocprofv3-avail info --pmc
```

The preceding command generates an output with the pmc info, if used with -d information of pmc for device -d is generated.

```
rocprofv3-avail info --pc-sampling
```

The preceding command generates list of supported PC sampling configurations for each agent that supports PC sampling. -d option is not applicable here.

```
rocprofv3-avail pmc-check  [pmc [pmc...]]
```

The preceding command checks if the pmc can be collected together

```
 rocprofv3-avail pmc-check -d 0 <pmc1> <pmc2> <pmc3>:device=1

The preceding command checks if the pmc1 and pmc2 can be collected together on agent 0
and pmc3 on agent 1
```

Note

The above command writes the ouptut to the standard output.

CHAPTER

SIX

USING ROCTX

ROCTx is an AMD tools extension library, a cross platform API for annotating code with markers and ranges. The ROCTx API is written in C++. In certain situations, such as debugging performance issues in large-scale GPU programs, API-level tracing might be too fine-grained to provide an overview of the program execution. In such cases, it is helpful to define specific tasks to be traced. To specify the tasks for tracing, enclose the respective source code with the API calls provided by the ROCTx library. This process is also known as instrumentation.

6.1 ROCTx annotations

ROCTx provides two types of annotations: markers and ranges.

6.1.1 Markers

Markers are used to insert a marker in the code with a message. Creating markers helps you see when a line of code is executed.

6.1.2 Ranges

Ranges are used to define the scope of code for instrumentation using enclosing API calls. A range is a programmer-defined task that has a well-defined start and end code scope. You can further refine the scope specified within a range using nested ranges. `rocprofv3` also reports the timelines for these nested ranges.

These are the two types of ranges:

- **Push and Pop:** These can be nested to form a stack. The Pop call is automatically associated with a prior Push call on the same thread.
- **Start and End:** These may overlap with other ranges arbitrarily. The Start call returns a handle that must be passed to the End call. These ranges can start and end on different threads.

6.1.3 ROCTx APIs

Here is the list of useful APIs for code instrumentation:

- `roctxMark`: Inserts a marker in the code with a message. Creating marks help you see when a line of code is executed.
- `roctxRangeStart`: Starts a range. Different threads can start ranges.
- `roctxRangePush`: Starts a new nested range.
- `roctxRangePop`: Stops the current nested range.
- `roctxRangeStop`: Stops the given range.
- `roctxProfilerPause`: Requests any currently running profiling tool to stop data collection.

- `roctxProfilerResume`: Requests any currently running profiling tool to resume data collection.
- `roctxGetThreadId`: Retrieves the ID for the current thread identical to the ID received using `rocprofiler_get_thread_id(rocprofiler_thread_id_t*)`.
- `roctxNameOsThread`: Labels the current CPU OS thread in the profiling tool output with the provided name.
- `roctxNameHsaAgent`: Labels the given HSA agent in the profiling tool output with the provided name.
- `roctxNameHipDevice`: Labels the HIP device ID in the profiling tool output with the provided name.
- `roctxNameHipStream`: Labels the given HIP stream in the profiling tool output with the provided name.

6.2 Using ROCTx in the application

The following sample code from the MatrixTranspose application shows the usage of ROCTx APIs:

```
#include <rocprofiler-sdk-roctx/roctx.h>

roctxMark("before hipLaunchKernel");
int rangeId = roctxRangeStart("hipLaunchKernel range");
roctxRangePush("hipLaunchKernel");

// Launching kernel from host
hipLaunchKernelGGL(matrixTranspose, dim3(WIDTH/THREADS_PER_BLOCK_X, WIDTH/THREADS_PER_
↪BLOCK_Y), dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y), 0,0,gpuTransposeMatrix,
↪gpuMatrix, WIDTH);

roctxMark("after hipLaunchKernel");

// Memory transfer from device to host
roctxRangePush("hipMemcpy");

hipMemcpy(TransposeMatrix, gpuTransposeMatrix, NUM * sizeof(float),␣
↪hipMemcpyDeviceToHost);

roctxRangePop();  // for "hipMemcpy"
roctxRangePop();  // for "hipLaunchKernel"
roctxRangeStop(rangeId);
```

To trace the API calls enclosed within the range, use:

```
rocprofv3 --marker-trace --output-format csv -- <application_path>
```

Running the preceding command generates a `marker_api_trace.csv` file prefixed with the process ID.

```
$ cat 210_marker_api_trace.csv
```

Here are the contents of `marker_api_trace.csv` file:

Table 6.1: Marker api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
MARKER_	before hipLaunchKernel	717	717	1	1520113899312225	1520113899312225
MARKER_	after hipLaunchKernel	717	717	4	1520113900128482	1520113900128482
MARKER_	hipMemcpy	717	717	5	1520113900141100	1520113901483408
MARKER_	hipLaunchKernel	717	717	3	1520113899684965	1520113901491622
MARKER_	hipLaunchKernel range	717	0	2	1520113899682208	1520113901495882

For the description of the fields in the output file, see *Output file fields*.

`roctxProfilerPause` and `roctxProfilerResume` can be used to hide the calls between them. This is useful when you want to hide the calls that are not relevant to your profiling session.

```
#include <rocprofiler-sdk-roctx/roctx.h>

// Memory transfer from host to device
HIP_API_CALL(hipMemcpy(gpuMatrix, Matrix, NUM * sizeof(float), hipMemcpyHostToDevice));

auto tid = roctx_thread_id_t{};
roctxGetThreadId(&tid);
roctxProfilerPause(tid);
// Memory transfer that should be hidden by profiling tool
HIP_API_CALL(
    hipMemcpy(gpuTransposeMatrix, gpuMatrix, NUM * sizeof(float),␣
↪hipMemcpyDeviceToDevice));
roctxProfilerResume(tid);

// Launching kernel from host
hipLaunchKernelGGL(matrixTranspose,
                   dim3(WIDTH / THREADS_PER_BLOCK_X, WIDTH / THREADS_PER_BLOCK_Y),
                   dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y),
                   0,
                   0,
                   gpuTransposeMatrix,
                   gpuMatrix,
                   WIDTH);

// Memory transfer from device to host
HIP_API_CALL(
    hipMemcpy(TransposeMatrix, gpuTransposeMatrix, NUM * sizeof(float),␣
↪hipMemcpyDeviceToHost));
```

To trace the preceding code, use:

```
rocprofv3 --marker-trace --hip-trace --output-format csv -- <application_path>
```

The preceding command generates a `hip_api_trace.csv` file prefixed with the process ID. The file contains two `hipMemcpy` calls with the in-between `hipMemcpyDeviceToHost` call hidden .

```
"Domain","Function","Process_Id","Thread_Id","Correlation_Id","Start_Timestamp","End_
↪Timestamp"
"HIP_COMPILER_API","__hipRegisterFatBinary",1643920,1643920,1,320301257609216,
↪320301257636427
"HIP_COMPILER_API","__hipRegisterFunction",1643920,1643920,2,320301257650707,
↪320301257678857
"HIP_RUNTIME_API","hipGetDevicePropertiesR0600",1643920,1643920,4,320301258114239,
↪320301337764472
"HIP_RUNTIME_API","hipMalloc",1643920,1643920,5,320301338073823,320301338247374
"HIP_RUNTIME_API","hipMalloc",1643920,1643920,6,320301338248284,320301338399595
"HIP_RUNTIME_API","hipMemcpy",1643920,1643920,7,320301338410995,320301631549262
"HIP_COMPILER_API","__hipPushCallConfiguration",1643920,1643920,10,320301632131175,
↪320301632134215
"HIP_COMPILER_API","__hipPopCallConfiguration",1643920,1643920,11,320301632137745,
↪320301632139735
"HIP_RUNTIME_API","hipLaunchKernel",1643920,1643920,12,320301632142615,320301632898289
"HIP_RUNTIME_API","hipMemcpy",1643920,1643920,14,320301632901249,320301633934395
"HIP_RUNTIME_API","hipFree",1643920,1643920,15,320301643320908,320301643511479
"HIP_RUNTIME_API","hipFree",1643920,1643920,16,320301643512629,320301643585639
```

6.3 Resource naming

`ROCTx` provides APIs to rename certain resources in the output generated by the profiling tool. You can pass the desired label for a specific resource in the output as an argument to the API. Note that ROCprofiler-SDK doesn't provide any explicit support for how profiling tools handle this request. Support for this capability is tool-specific.

The following table lists the APIs available for labeling the given resources:

Table 6.2: resource naming

Resource	API	Description
OS thread	`roctxNameOsThread(const char* name)`	Labels the current CPU OS thread with the given name in the output. Note that ROCTx does NOT rename the thread using `pthread_setname_np`.
HIP runtime		
	`roctxNameHipDevice(const char* name, int device_id)`	Labels the given HIP device ID with the given name in the output.
	`roctxNameHipStream(const char* name, const struct ihipStream_t* stream)`	Labels the given HIP stream ID with the given name in the output.
HSA runtime	`roctxNameHsaAgent(const char* name, const struct hsa_agent_s*)`	Labels the given HSA agent with the given name in the output.

6.4 Using ROCTx in the python application

ROCTx APIs can be used in a python application using the `roctx` module. The APIs are available as functions in the module. The API names are prefixed with `roctx` to avoid name conflicts with other libraries.

The following sample code from the MatrixTranspose application shows the usage of ROCTx APIs in a python application:

```
import os
import roctx
import random
from roctx.context_decorators import RoctxRange

_prefix = os.path.basename(__file__)

@RoctxRange("matrix_transpose")
def matrix_transpose(matrix):
    nrows = len(matrix)
    ncols = len(matrix[0]) if nrows > 0 else 0
    with RoctxRange(f"transpose(nrows={nrows}, ncols={ncols})"):
        # Transpose the matrix
        transposed = [[matrix[j][i] for j in range(nrows)] for i in range(ncols)]
        return transposed

def generate_matrix(rows, cols):
    with RoctxRange(f"generate_matrix(rows={rows}, cols={cols})"):
        return [[random.randint(0, 100) for _ in range(cols)] for _ in range(rows)]

def run(rows, cols):
    idx = roctx.rangeStart(f"run(rows={rows}, cols={cols})")
    matrix = generate_matrix(rows, cols)
    transposed = matrix_transpose(matrix)
    roctx.rangeStop(idx)
    return matrix, transposed

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument("-r", "--rows", type=int, default=4, help="Number of rows")
    parser.add_argument("-c", "--cols", type=int, default=5, help="Number of columns")
    args = parser.parse_args()

    roctx.mark(f"MatrixTranspose: rows={args.rows}, cols={args.cols}")
    with RoctxRange("main"):
        matrix, transposed = run(args.rows, args.cols)
        print(f"[{_prefix}] Original matrix:")
        for row in matrix:
            print(row)
        print(f"\n[{_prefix}] Transposed matrix:")
        for row in transposed:
            print(row)
```

Before using the `roctx` module for python application, ensure that the `roctx` module is built, installed and available

in your python environment.

An example to build and install `roctx` module is as follows:

```
cmake -B build-sdk -DCMAKE_INSTALL_PREFIX=/opt/rocm -DROCPROFILER_PYTHON_VERSIONS="3.10"␣
↪-DCMAKE_PREFIX_PATH=/opt/rocm
```

If you are using a different python version, replace `3.10` with the appropriate version in the above command. Multiple python versions can be specified in the `ROCPROFILER_PYTHON_VERSIONS` variable. The roctx module will be built and installed for all the specified python versions.

```
``cmake -B build-sdk -DCMAKE_INSTALL_PREFIX=/opt/rocm -DROCPROFILER_PYTHON_VERSIONS="3.8;
↪3.9;3.10;3.11;3.12" -DCMAKE_PREFIX_PATH=/opt/rocm``
```

Based on the python major.minor version and the roctx module install path ("/opt/rocm" in above example), set the `PYTHONPATH` environment variable to include the path to the `roctx` module.

```
export PYTHONPATH="<install-path>/lib/pythonX.Y/site-packages:$PYTHONPATH"
```

Above example will install the roctx module in `/opt/rocm/lib/python3.10/site-packages`, set the `PYTHONPATH` as follows:

```
export PYTHONPATH=/opt/rocm/lib/python3.10/site-packages:$PYTHONPATH
```

Once the `PYTHONPATH` is set, user should be able to import the *roctx* package:

```
python3 -c "import roctx"
```

User can profile the python application which is annotated with ROCTx markers using `rocprofv3` as follows:

```
rocprofv3 --marker-trace --output-format csv -- $(which python) <python_application_path>
```

The preceding command generates a `marker_api_trace.csv` file prefixed with the process ID.

Table 6.3: Marker api trace for python application

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
MARKER_	Matrix-Transpose: rows=4, cols=5	15964	15964	1	1392141764711512	1392141764711512
MARKER_	generate_matrix(cols=5)	15964	15964	4	1392141765500646	1392141765523276
MARKER_	transpose(nrows ncols=5)	15964	15964	7	1392141765527536	1392141765531786
MARKER_	matrix_transpo	15964	15964	6	1392141765525156	1392141765532696
MARKER_	run(rows=4 cols=5)	15964	15964	3	1392141765498106	1392141765534186
MARKER_	main	15964	15964	2	1392141765494795	1392141765574506

CHAPTER

SEVEN

USING ROCPROFV3 WITH MPI

Message Passing Interface (MPI) is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. MPI is widely used for developing parallel applications and is considered the de facto standard for communication in high-performance computing (HPC) environments. MPI applications are parallel programs that run across multiple processes, which can be distributed over one or more nodes.

For MPI applications or other job launchers such as SLURM, place `rocprofv3` inside the job launcher. The following example demonstrates how to use `rocprofv3` with MPI:

```
mpirun -n 4 rocprofv3 --hip-trace --output-format csv -- <application_path>
```

The preceding command runs the application with `rocprofv3` and generates the trace file for each rank. The trace files are prefixed with the process ID.

```
2293213_agent_info.csv
2293213_hip_api_trace.csv
2293214_agent_info.csv
2293214_hip_api_trace.csv
2293212_agent_info.csv
2293212_hip_api_trace.csv
2293215_agent_info.csv
2293215_hip_api_trace.csv
```

Since the data collection is performed in-process, it's ideal to collect data from within the processes launched by MPI. When `rocprofv3` is run outside of `mpirun`, the tool library is loaded into the *mpirun* executable.. Collecting data outside of `mpirun` works but fetches agent info for the `mpirun` process too. For example:

```
rocprofv3 --hip-trace -d %h.%p.%env{OMPI_COMM_WORLD_RANK}% --output-format csv -- mpirun␣
↪-n 2  <application_path>
```

In the preceding example, an extra agent info file is generated for the `mpirun` process. The trace files are prefixed with the hostname, process ID, and the MPI rank.

```
ubuntu-latest.3000020.1/3000020_agent_info.csv
ubuntu-latest.3000020.0/3000019_agent_info.csv
ubuntu-latest.3000020.1/3000020_hip_api_trace.csv
ubuntu-latest.3000020.0/3000019_hip_api_trace.csv
```

7.1 ROCTx annotations

For an MPI application, you can use ROCTx annotations to mark the start and end of the MPI code region. The following example demonstrates how to use ROCTx annotations with MPI:

```
#include <roctx.h>
#include <mpi.h>
...

void run(int rank, int tid, int dev_id, int argc, char** argv)
{
    auto roctx_run_id = roctxRangeStart("run");

    const auto mark = [rank, tid, dev_id](std::string_view suffix) {
        auto _ss = std::stringstream{};
        _ss << "run/rank-" << rank << "/thread-" << tid << "/device-" << dev_id << "/" <
↪< suffix;
        roctxMark(_ss.str().c_str());
    };

    mark("begin");

    constexpr unsigned int M = 4960 * 2;
    constexpr unsigned int N = 4960 * 2;

    unsigned long long nitr = 0;
    unsigned long long nsync = 0;

    if(argc > 2) nitr = atoll(argv[2]);
    if(argc > 3) nsync = atoll(argv[3]);

    hipStream_t stream = {};

    printf("[transpose] Rank %i, thread %i assigned to device %i\n", rank, tid, dev_id);
    HIP_API_CALL(hipSetDevice(dev_id));
    HIP_API_CALL(hipStreamCreate(&stream));

    auto_lock_t _lk{print_lock};
    std::cout << "[transpose][" << rank << "][" << tid << "] M: " << M << " N: " << N <<␣
↪std::endl;
    _lk.unlock();

    std::default_random_engine         _engine{std::random_device{}() * (rank + 1) *␣
↪(tid + 1)};
    std::uniform_int_distribution<int> _dist{0, 1000};

    ...

    auto t1 = std::chrono::high_resolution_clock::now();
    for(size_t i = 0; i < nitr; ++i)
    {
        roctxRangePush("run/iteration");
        transpose<<<grid, block, 0, stream>>>(in, out, M, N);
```

(continues on next page)

(continued from previous page)

```
        check_hip_error();
        if(i % nsync == (nsync - 1))
        {
            roctxRangePush("run/iteration/sync");
            HIP_API_CALL(hipStreamSynchronize(stream));
            roctxRangePop();
        }
        roctxRangePop();
    }
    auto t2 = std::chrono::high_resolution_clock::now();
    HIP_API_CALL(hipStreamSynchronize(stream));
    HIP_API_CALL(hipMemcpyAsync(out_matrix, out, size, hipMemcpyDeviceToHost, stream));
    double time = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1).
↪count();
    float  GB   = (float) size * nitr * 2 / (1 << 30);

    print_lock.lock();
    std::cout << "[transpose][" << rank << "][" << tid << "] Runtime of transpose is " <
↪< time
              << " sec\n";
    std::cout << "[transpose][" << rank << "][" << tid
              << "] The average performance of transpose is " << GB / time << " GBytes/
↪sec"
              << std::endl;
    print_lock.unlock();

    ...

    mark("end");

    roctxRangeStop(roctx_run_id);
}
```

This preceding sample generates output similar to the following:

```
"MARKER_CORE_API","run/rank-0/thread-0/device-0/begin",2936128,2936128,5,432927100747635,
↪432927100747635
"MARKER_CORE_API","run/rank-0/thread-1/device-1/begin",2936128,2936397,7,432927100811475,
↪432927100811475
"MARKER_CORE_API","run/iteration",2936128,2936397,22,432928615598809,432928648197081
"MARKER_CORE_API","run/iteration",2936128,2936397,61,432928648229081,432928648234041
"MARKER_CORE_API","run/iteration",2936128,2936397,67,432928648234701,432928648239621
"MARKER_CORE_API","run/iteration",2936128,2936397,73,432928648239971,432928648244141
"MARKER_CORE_API","run/iteration/sync",2936128,2936397,84,432928648249791,432928664871094
...

"MARKER_CORE_API","run/iteration",2936128,2936128,6313,432929397644269,432929397648369
"MARKER_CORE_API","run/iteration/sync",2936128,2936128,6324,432929397653119,
↪432929401455250
"MARKER_CORE_API","run/iteration",2936128,2936128,6319,432929397648779,432929401455640
"MARKER_CORE_API","run/rank-0/thread-1/device-1/end",2936128,2936397,6339,
↪432929527301990,432929527301990
```

(continues on next page)

(continued from previous page)

```
"MARKER_CORE_API","run",2936128,2936397,6,432927100787035,432929527313480
"MARKER_CORE_API","run/rank-0/thread-0/device-0/end",2936128,2936128,6342,
↪432929612438185,432929612438185
"MARKER_CORE_API","run",2936128,2936128,4,432927100729745,432929612448285
```

7.2 Output format features

To collect the profiles of the individual MPI processes, use `rocprofv3` with output directory option to send output to unique files.

```
mpirun -n 2 rocprofv3 --hip-trace -d %h.%p.%env{OMPI_COMM_WORLD_RANK}% --output-format␣
↪csv --  <application_path>
```

To see the placeholders supported by the output directory option, see *output directory placeholders*.

Assuming the hostname as *ubuntu-latest*, the process IDs as 3000020 and 3000019, the generated output file names are:

```
ubuntu-latest.3000020.1/ubuntu-latest/3000020_agent_info.csv
ubuntu-latest.3000019.0/ubuntu-latest/3000019_agent_info.csv
ubuntu-latest.3000020.1/ubuntu-latest/3000020_hip_api_trace.csv
ubuntu-latest.3000019.0/ubuntu-latest/3000019_hip_api_trace.csv
```

CHAPTER

EIGHT

USING ROCPROFV3 WITH OPENMP

rocprofv3 does not provide native support for profiling CPU-side OpenMP code. However, when OpenMP is used to offload computations to AMD GPUs (for example, via OpenMP target offload), *rocprofv3* can capture and profile GPU activities initiated by these offloaded regions. Note that profiling of CPU-side OpenMP parallel regions is not supported.

8.1 Example: Vector Addition Using OpenMP Offload on AMD GPUs

The following example demonstrates how to perform vector addition using OpenMP target offload, enabling execution of the workload on AMD GPUs.

Key Steps:

- Initialize input arrays on the host.
- Offload the vector addition computation to the GPU using OpenMP directives.
- Retrieve and verify the results on the host.

```
#include <stdio.h>
#include <omp.h>

#define N 1024

int main() {
    float a[N], b[N], c[N];

    // Initialize input arrays
    for (int i = 0; i < N; ++i) {
        a[i] = i * 1.0f;
        b[i] = (N - i) * 1.0f;
    }

    // Offload vector addition to GPU
    #pragma omp target teams distribute parallel for map(to: a[0:N], b[0:N]) map(from:␣
↪c[0:N])
    for (int i = 0; i < N; ++i) {
        c[i] = a[i] + b[i];
    }

    // Verify results
    int errors = 0;
    for (int i = 0; i < N; ++i) {
```

(continues on next page)

(continued from previous page)

```
        if (c[i] != N * 1.0f) {
            errors++;
        }
    }

    if (errors == 0) {
        printf("Vector addition successful!\\n");
    } else {
        printf("Vector addition failed with %d errors.\\n", errors);
    }

    return 0;
}
```

8.2 Building the OpenMP Offload Application

To compile the application for AMD GPU offload, use the following command:

```
amdclang++ -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -L/opt/rocm/lib --offload-
↪arch=gfx9xx -o vector_add <application>
```

8.3 Profiling the Application with rocprofv3

To profile the GPU activity during execution, run the application with *rocprofv3*:

```
rocprofv3 -s --output-format csv -- ./vector_add
```

Upon execution, *rocprofv3* will generate several CSV trace files, such as:

- *<pid>_kernel_trace.csv*
- *<pid>_hsa_api_trace.csv*
- *<pid>_memory_copy_trace.csv*
- *<pid>_memory_allocation_trace.csv*
- *<pid>_scratch_memory_trace.csv*

These files contain detailed profiling information about GPU kernel execution, HSA API calls, memory operations, and more, enabling comprehensive analysis of the offloaded workload.

CHAPTER

NINE

USING PC SAMPLING

PC (Program Counter) sampling service for GPU profiling is a profiling technique to periodically sample the program counter during GPU kernel execution. PC sampling helps in understanding code execution patterns and identifying hotspot(s).

Here are the benefits of using PC sampling:

- Identify performance bottlenecks
- Understand kernel execution behavior
- Analyze code coverage
- Find heavily executed code paths

To try out the PC sampling feature, you can use the command-line tool `rocprofv3` or the ROCprofiler-SDK library on *ROCm 6.4* or later.

> **Note**
>
> PC sampling is ONLY supported on AMD GPUs with architectures gfx90a and later.

9.1 PC sampling availability and configuration

To check if the GPU supports PC sampling, use:

```
rocprofv3 -L
```

Or

```
rocprofv3 --list-avail
```

The output lists if `rocprofv3` supports PC sampling on the GPU and the supported configuration.

```
List available PC Sample Configurations for node_id     11
Method: ROCPROFILER_PC_SAMPLING_METHOD_HOST_TRAP
Unit:   ROCPROFILER_PC_SAMPLING_UNIT_TIME
Minimum_Interval:       1
Maximum_Interval:       18446744073709551615
```

The preceding output shows that the GPU supports PC sampling with the `ROCPROFILER_PC_SAMPLING_METHOD_HOST_TRAP` method and the `ROCPROFILER_PC_SAMPLING_UNIT_TIME` unit. The minimum and maximum intervals are also displayed.

> **Note**
>
> Important firmware fixes to host-trap and stochastic PC-sampling for AMD Instinct MI300X have been made in ROCm 7.0. To ensure that you have the latest fixes, check if you have the correct firmware versions installed:
>
> For host-trap PC-sampling on MI300X: PSP TOS Firmware >= version 00.36.02.59 or 0x00360259 For stochastic PC-sampling on MI300X as described in the following section: MEC Firmware feature version: 50, firmware version >= 0x0000001a

To check the firmware versions, use:

```
# To check PSP TOS Firmware:
sudo cat /sys/kernel/debug/dri/0/amdgpu_firmware_info | grep SOS

# To check MEC Firmware:
sudo cat /sys/kernel/debug/dri/1/amdgpu_firmware_info | grep MEC
```

Based on the available PC-sampling configurations, use the following command to profile the application using PC-sampling:

```
rocprofv3 --pc-sampling-beta-enabled --pc-sampling-method host_trap --pc-sampling-unit␣
↪time --pc-sampling-interval 1 --output-format csv -- <application_path>
```

The preceding command enables PC sampling with the `host_trap` method, `time` unit, and an interval of 1 s (microsecond). Replace `<application_path>` with the path to the application you want to profile.

This generates two files, `agent_info.csv` and `pc_sampling_host_trap.csv`. Both files are prefixed with the process ID.

Here are the contents of `pc_sampling_host_trap.csv` file generated for MatrixTranspose sample application:

Table 9.1: PC sampling host trap

Sample_Timestamp	Exec_Mask	Dispatch_Id	Instruction	Instruction_Comm	Correlation_Id
3464444413017201	65535	1	s_endpgm		1
3464444413017201	65535	1	s_waitcnt vmcnt(0)		1
3464444413018481	65535	1	s_waitcnt vmcnt(0)		1
3464444413018481	65535	1	s_endpgm		1
3464444413018481	65535	1	s_waitcnt vmcnt(0)		1
3464444413018481	65535	1	s_waitcnt vmcnt(0)		1
3464444413018481	65535	1	s_endpgm		1
3464444413018481	65535	1	s_endpgm		1
3464444413019601	65535	1	s_waitcnt vmcnt(0)		1
3464444413019761	65535	1	s_load_dwor s8, s[4:5], 0x24		1
3464444413019761	65535	1	s_waitcnt vmcnt(0)		1

continues on next page

Table 9.1 – continued from previous page

Sample_Timestamp	Exec_Mask	Dis-patch_Id	Instruction	Instruc-tion_Comm	Correlation_Id
3464444413019761	65535	1	s_endpgm		1
3464444413019761	65535	1	s_load_dwor s8, s[4:5], 0x24		1
3464444413019761	65535	1	s_endpgm		1
3464444413019761	65535	1	s_endpgm		1
3464444413020881	65535	1	s_endpgm		1
3464444413020881	65535	1	s_endpgm		1
3464444413020881	65535	1	s_endpgm		1
3464444413020881	65535	1	s_waitcnt lgkmcnt(0)		1
3464444413020881	65535	1	v_addc_co_u v5, vcc, v1, v5, vcc		1
3464444413020881	65535	1	s_endpgm		1
3464444413020881	65535	1	s_waitcnt vmcnt(0)		1
3464444413020881	65535	1	s_endpgm		1
3464444413020881	65535	1	s_waitcnt vmcnt(0)		1
3464444413021041	65535	1	s_endpgm		1
3464444413020881	65535	1	v_bfe_u32 v0, v0, 10, 10		1
3464444413021041	65535	1	s_endpgm		1
3464444413021041	65535	1	s_endpgm		1
3464444413021041	65535	1	s_waitcnt vmcnt(0)		1
3464444413021041	65535	1	s_endpgm		1
3464444413021041	65535	1	s_waitcnt vmcnt(0)		1
3464444413021041	65535	1	s_endpgm		1
3464444413022001	65535	1	s_waitcnt vmcnt(0)		1
3464444413022001	65535	1	s_endpgm		1
3464444413022001	65535	1	s_endpgm		1
3464444413022001	65535	1	s_endpgm		1
3464444413022001	65535	1	s_endpgm		1
3464444413022001	65535	1	s_waitcnt vmcnt(0)		1
3464444413022001	65535	1	s_endpgm		1
3464444413022001	65535	1	s_waitcnt vmcnt(0)		1
3464444413022001	65535	1	s_waitcnt lgkmcnt(0)		1
3464444413022161	65535	1	s_endpgm		1
3464444413022161	65535	1	s_waitcnt vmcnt(0)		1
3464444413022161	65535	1	s_endpgm		1

continues on next page

Table 9.1 – continued from previous page

Sample_Timestamp	Exec_Mask	Dispatch_Id	Instruction	Instruction_Comm	Correlation_Id
3464444413022161	65535	1	s_load_dwor s8, s[4:5], 0x24		1
3464444413022161	65535	1	global_store_ v[0:1], v3, off		1
3464444413022161	65535	1	s_endpgm		1
3464444413022161	65535	1	s_endpgm		1
3464444413022161	65535	1	s_waitcnt vmcnt(0)		1
3464444413022161	65535	1	s_endpgm		1
3464444413022161	65535	1	s_endpgm		1
3464444413022161	65535	1	s_waitcnt vmcnt(0)		1
3464444413022161	65535	1	s_endpgm		1
3464444413022321	65535	1	s_load_dwor s[0:3], s[4:5], 0x0		1
3464444413022161	65535	1	s_waitcnt vmcnt(0)		1
3464444413022321	65535	1	s_endpgm		1
3464444413022161	65535	1	s_waitcnt vmcnt(0)		1
3464444413023281	65535	1	s_endpgm		1
3464444413023281	65535	1	s_endpgm		1
3464444413023281	65535	1	v_ashrrev_i3 v1, 31, v0		1
3464444413024561	65535	1	s_waitcnt vmcnt(0)		1
3464444413023281	65535	1	s_endpgm		1
3464444413024561	65535	1	s_endpgm		1
3464444413023761	65535	1	s_waitcnt vmcnt(0)		1
3464444413026321	65535	1	s_waitcnt vmcnt(0)		1
3464444413024401	65535	1	global_store_ v[0:1], v3, off		1
3464444413027121	65535	1	s_waitcnt vmcnt(0)		1
3464444413025041	65535	1	v_add_co_u3 v0, vcc, s0, v0		1
3464444413027761	65535	1	s_waitcnt vmcnt(0)		1
3464444413025361	65535	1	s_endpgm		1
3464444413027601	65535	1	s_waitcnt vmcnt(0)		1

continues on next page

Table 9.1 – continued from previous page

Sample_Timestamp	Exec_Mask	Dispatch_Id	Instruction	Instruction_Comm	Correlation_Id
3464444413026321	65535	1	s_waitcnt vmcnt(0)		1
3464444413028401	65535	1	s_waitcnt vmcnt(0)		1
3464444413026481	65535	1	s_waitcnt vmcnt(0)		1
3464444413028881	65535	1	s_waitcnt vmcnt(0)		1
3464444413026641	65535	1	s_waitcnt vmcnt(0)		1
3464444413028401	65535	1	s_load_dwor s8, s[4:5], 0x24		1
3464444413027281	65535	1	s_waitcnt vmcnt(0)		1
3464444413029681	65535	1	s_endpgm		1

For description of the fields in the output file, see *PC sampling fields*.

If you find the `Instruction_Comment` field in the output file to be empty, populate this field by compiling your application with debug symbols. Enabling debug symbols while compiling the application maps back to the source line. This helps in understanding the code execution pattern and hotspots.

Table 9.2: PC sampling host trap with debug symbols

Sample_Timestamp	Exec_Mask	Dispatch_Id	Instruction	Instruction_Comm	Correlation_Id	
54155306462675	65535	1	s_waitcnt lgkmcnt(0)	/opt/rocm/inc	1	_runtime.h:275
54155306462715	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306462755	65535	1	s_endpgm	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306462755	65535	1	s_endpgm	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306462955	65535	1	s_endpgm	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306463035	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306463235	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306463315	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306463515	65535	1	s_endpgm	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306463755	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306463875	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306464075	65535	1	v_mov_b32_ v2, s4	/opt/rocm/inc	1	_runtime.h:275
54155306464155	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306464155	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306464275	65535	1	s_endpgm	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306464395	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306464515	65535	1	s_waitcnt lgkmcnt(0)	/opt/rocm/inc	1	_runtime.h:275
54155306464555	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306464595	65535	1	s_waitcnt vmcnt(0)	/opt/rocm-6.4.0/share/h	1	atrixTranspose/Ma
54155306464595	65535	1	v_mov_b32_ v2, s6	/opt/rocm/inc	1	_runtime.h:275
54155306464595	65535	1	s_waitcnt lgkmcnt(0)	/opt/rocm/inc	1	_runtime.h:275

The preceding output shows the `Instruction_Comment` field populated with the source-line information.

9.2 PC sampling fields

Here are the fields in the output file generated by PC sampling:

- `Sample_Timestamp`: Timestamp when sample is generated
- `Exec_Mask`: Active SIMD lanes when sampled
- `Dispatch_Id`: Originating kernel dispatch ID
- `Instruction`: Assembly instruction such as `s_load_dword s8, s[1:2], 0x10`
- `Instruction_Comment`: Instruction comment that maps back to the source-line if debug symbols were enabled when application was compiled
- `Correlation_Id`: API launch call ID that matches dispatch ID

  To dump samples in a more comprehensive format, use JSON through `--output-format json`:

```
rocprofv3 --pc-sampling-beta-enabled --pc-sampling-method host_trap --pc-sampling-unit␣
↪time --pc-sampling-interval 1 --output-format json -- <application_path>
```

The preceding command generates a JSON file with the comprehensive output. Here is a trimmed down output with multiple records:

```
{
  "pc_sample_host_trap": [
    {
      "record": {
        "hw_id": {
          "chiplet": 0,
          "wave_id": 0,
          "simd_id": 2,
          "pipe_id": 0,
          "cu_or_wgp_id": 1,
          "shader_array_id": 0,
          "shader_engine_id": 2,
          "workgroup_id": 0,
          "vm_id": 3,
          "queue_id": 2,
          "microengine_id": 1
        },
        "pc": {
          "code_object_id": 1,
          "code_object_offset": 20228
        },
        "exec_mask": 18446744073709551615,
        "timestamp": 51040126667689,
        "dispatch_id": 1,
        "corr_id": {
          "internal": 1,
          "external": 0
        },
        "wrkgrp_id": {
          "x": 182,
          "y": 0,
          "z": 0
```

(continues on next page)

(continued from previous page)


```
        },
        "wave_in_grp": 1
      },
      "inst_index": 0
    },
    {
      "record": {
        "hw_id": {
          "chiplet": 0,
          "wave_id": 0,
          "simd_id": 2,
          "pipe_id": 0,
          "cu_or_wgp_id": 0,
          "shader_array_id": 0,
          "shader_engine_id": 2,
          "workgroup_id": 0,
          "vm_id": 3,
          "queue_id": 2,
          "microengine_id": 1
        },
        "pc": {
          "code_object_id": 1,
          "code_object_offset": 20236
        },
        "exec_mask": 18446744073709551615,
        "timestamp": 51040126667689,
        "dispatch_id": 1,
        "corr_id": {
          "internal": 1,
          "external": 0
        },
        "wrkgrp_id": {
          "x": 158,
          "y": 0,
          "z": 0
        },
        "wave_in_grp": 2
      },
      "inst_index": 1
    }
  ]
}
```


For description of the fields in the JSON output, see *Output file fields*.

9.3 Hardware-Based (Stochastic) PC Sampling Method

The new `ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC` has been introduced for gfx942 architecture. It employs a specific hardware for probing waves actively running on GPU. Beside information already provided with `ROCPROFILER_PC_SAMPLING_METHOD_HOST_TRAP` useful for determining hot-spots within the kernel, it delivers additional information that tells whether a sampled wave issued an instruction represented with particular PC. If not, it provides the reason for not issuing the instruction (stall reason). This type of information is particularly useful for

understanding stalls during the kernel execution.

To use this method on gfx942, we recommend listing available PC sampling configurations to verify if the latest ROCm stack is installed on the system by running:

```
rocprofv3 -L
```

Output similar to the following indicates that the `ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC` method is available:

```
Method: ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC
Unit:   ROCPROFILER_PC_SAMPLING_UNIT_CYCLES
Minimum_Interval:       256
Maximum_Interval:       2147483648
```

Please note that on gfx942, *ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC* requires intervals to be specified in cycles, whose values are powers of 2

To profile a gfx942 accelerated application with `ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC` PC sampling, one can use the following command:

```
rocprofv3 --pc-sampling-beta-enabled --pc-sampling-method stochastic --pc-sampling-unit␣
↪cycles --pc-sampling-interval 1048576 --output-format csv, json -- <application_path>
```

The previous command serializes samples in both CSV and JSON output formats in the `pc_sampling_stochastic.csv` and `out_results.json` files, respectively.

Comparing the `pc_sampling_stochastic.csv` to `pc_sampling_host_trap` from previous section, one can notice that the `ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC` method generates additional fields: - `Wave_Issued_Instruction`: Indicates whether the wave issued an instruction (value 1) represented with particular PC or not (value 0) - `Instruction_Type`: If the value of `Wave_Issued_Instruction` is 1, this fields indicates the type of the issued instruction. Otherwise, this fields irrelevant. - `Stall_Reason`: If the value of `Wave_Issued_Instruction` is 0, this fields indicates the reason for not issuing the instruction (stall reason). Otherwise, this field is irrelevant. - `Wave_Count`: Total number of waves actively running on a compute unit when the sample was generated.

Table 9.3: PC sampling stochastic with debug symbols

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390705261841	184467	24	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	24	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390705261924	184467	29	v_max(v1, v2, v0	/home/ sdk-interna operati ops.cp	29	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390705694732	184467	53	v_mad v[0:1], s[2:3], v0, s2, v[2:3]	/home/ sdk-interna operati ops.cp	53	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390705694744	184467	54	v_lshl_ v[0:1], s[4:5], 0, v[0:1]	/home/ sdk-interna operati ops.cp	54	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390705694769	184467	56	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	56	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390705694772	184467	56	s_waitc lgkm-cnt(0)	/usr/in	56	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390705694810	184467	58	v_cmp vcc, s2, v1	/home/ sdk-interna operati ops.cp	58	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390705694820	184467	59	s_waitc vm-cnt(1)	/home/ sdk-interna operati ops.cp	59	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390705694840	184467	60	s_and_ s5, s4, 0xffff	/usr/in	60	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390705694856	184467	61	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	61	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706112944	184467	65	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	65	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	7
390706112965	184467	66	global_ v[0:1], v2, off	/home/ sdk-interna operati ops.cp	66	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706112966	18446	66	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	66	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706112966	18446	66	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	66	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706112967	18446	66	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	66	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706112971	18446	66	s_load_ s[4:7], s[0:1], 0x0	/usr/in	66	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706112984	18446	67	s_waitc vm-cnt(1)	/home/ sdk-interna operati ops.cp	67	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706112988	18446	67	s_waitc vm-cnt(1)	/home/ sdk-interna operati ops.cp	67	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113000	18446	68	v_add_ v0, s3, v0	/home/ sdk-interna operati ops.cp	68	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113004	18446	68	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	68	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113053	18446	69	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	69	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	7

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706113059	184467	69	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	69	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113080	184467	70	s_load_ s[4:7], s[0:1], 0x0	/usr/in	70	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113097	184467	71	s_waitc vm-cnt(1)	/home/ sdk-interna operati ops.cp	71	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113101	184467	71	s_waitc vm-cnt(1)	/home/ sdk-interna operati ops.cp	71	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113111	184467	72	v_sub_ v4, v2, v3	/home/ sdk-interna operati ops.cp	72	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113115	184467	72	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	72	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113134	184467	73	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	73	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	7
390706113147	184467	74	s_waitc lgkm-cnt(0)	/home/ sdk-interna operati ops.cp	74	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113149	184467	74	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	74	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706113153	184467	74	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	74	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113179	184467	76	s_waitc lgkm-cnt(0)	/usr/in	76	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	2
390706113184	184467	76	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	76	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113184	184467	76	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	76	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113206	184467	77	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	77	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	10
390706113209	184467	77	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	77	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	11
390706113220	184467	78	s_load_ s4, s[0:1], 0x2c	/usr/in	78	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	1
390706113221	184467	78	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	78	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113227	184467	78	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	78	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113234	184467	79	s_waitc vm-cnt(1)	/home/ sdk-interna operati ops.cp	79	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706113235(	184467	79	s_load_ s[0:1], s[0:1], 0x10	/usr/in(	79	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113236(	184467	79	s_and_ s[2:3], vcc	/home/ sdk-interna operati ops.cp	79	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113250(	184467	80	s_wait(lgkm-cnt(0)	/usr/in(	80	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	1
390706113253(	184467	80	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	80	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113255(	184467	80	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	80	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113257:	184467	80	v_add_ v2, 1.0, v2	/home/ sdk-interna operati ops.cp	80	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113270	184467	81	s_wait(lgkm-cnt(0)	/usr/in(	81	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113278:	184467	81	s_load_ s2, s[0:1], 0x18	/usr/in(	81	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	8
390706113292'	184467	82	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	82	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113301	184467	83	s_wait(lgkm-cnt(0)	/usr/in(	83	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	1
390706113301(	184467	83	s_and_ s[2:3], vcc	/home/ sdk-interna operati ops.cp	83	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3

continues on next page

Table 9.3 – continued from previous page

Sample_Timestam	Exec_	Dispatch_	Instruction	Instruction_C	Correlation_Id	Wave_	Instruction_Type	Stall_Reason	Wave_Count
390706113303	184467	83	s_and_s[2:3], vcc	/home/sdk-interna operati ops.cp	83	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113305	184467	83	v_lshlr v[0:1], 2, v[0:1]	/home/sdk-interna operati ops.cp	83	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113317	184467	84	v_div_ v3, v3, v6, v7	/home/sdk-interna operati ops.cp	84	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113318	184467	84	s_wait vm-cnt(0)	/home/sdk-interna operati ops.cp	84	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113336	184467	85	global_ v2, v[2:3], off	/home/sdk-interna operati ops.cp	85	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	11
390706113351	184467	86	s_mul_ s2, s2, s5	/home/sdk-interna operati ops.cp	86	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113352	184467	86	s_wait vm-cnt(0)	/home/sdk-interna operati ops.cp	86	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113369	184467	87	s_wait vm-cnt(1)	/home/sdk-interna operati ops.cp	87	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113373	184467	87	s_wait vm-cnt(1)	/home/sdk-interna operati ops.cp	87	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113387	184467	88	s_wait lgkm-cnt(0)	/usr/in	88	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	1

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706113390	184467	88	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	88	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113408	184467	89	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	89	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706113409	184467	89	v_add_ v2, v2, v3	/home/ sdk-interna operati ops.cp	89	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	10
390706113411	184467	89	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	89	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	9
390706113411	184467	89	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	89	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	7
390706113412	184467	89	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	89	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	7
390706113415	184467	89	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	89	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	9
390706113424	184467	90	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	90	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113443	184467	91	v_add_ v2, -1.0, v2	/home/ sdk-interna operati ops.cp	91	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	2

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706113444	184467	91	v_add_ v3, -1.0, v3	/home/ sdk- interna operati ops.cp	91	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113459	184467	92	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	92	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113459	184467	92	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	92	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113462	184467	92	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	92	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706113480	184467	93	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	15
390706113484	184467	93	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	7
390706113496	184467	93	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113500	184467	93	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	9
390706113506	184467	93	s_wait vm-cnt(0)	/home/ sdk- interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	14

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706113506	184467	93	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706113508	184467	93	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	17
390706113522	184467	93	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	93	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	15
390706113561	184467	94	s_wait(lgkm-cnt(0)	/home/ sdk-interna operati ops.cp	94	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3
390706113573	184467	95	s_wait(vm-cnt(1)	/home/ sdk-interna operati ops.cp	95	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706526501	184467	112	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	112	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706526582	184467	117	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	117	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	8
390706526594	184467	118	s_wait(vm-cnt(0)	/home/ sdk-interna operati ops.cp	118	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706526629	184467	120	s_wait(lgkm-cnt(0)	/home/ sdk-interna operati ops.cp	120	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5

continues on next page

Table 9.3 – continued from previous page

Sam-ple_Timestam	Exec_	Dis-patch_	In-struc-tion	In-struc-tion_C	Correla-tion_Id	Wave_	Instruc-tion_Type	Stall_Reason	Wave_Count
390706526629	18446	120	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	120	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706526633	18446	120	v_mul_ v7, v3, v6	/home/ sdk-interna operati ops.cp	120	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706526634	18446	120	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	120	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706526665	18446	122	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	122	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4
390706526677	18446	123	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	123	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6
390706526695	18446	124	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	124	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706526809	18446	126	v_and_ v2, 0x7ffff v2	/opt/ro 6.4.0/li	126	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706526810	18446	126	s_waitc vm-cnt(0)	/home/ sdk-interna operati ops.cp	126	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5
390706526821	18446	127	s_waitc vm-cnt(1)	/home/ sdk-interna operati ops.cp	127	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5

Similarly, `ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC` method delivers additional information to every sample in the JSON output. The following snippet shows one sample from `out_results.json` file.

```
{
  "record": {
    "flags": {
      "has_mem_cnt": 0
    },
    "hw_id": {
      "chiplet": 4,
      "wave_id": 0,
      "simd_id": 2,
      "pipe_id": 3,
      "cu_or_wgp_id": 1,
      "shader_array_id": 0,
      "shader_engine_id": 3,
      "workgroup_id": 0,
      "vm_id": 3,
      "queue_id": 2,
      "microengine_id": 1
    },
    "pc": {
      "code_object_id": 2,
      "code_object_offset": 13880
    },
    "exec_mask": 18446744073709551615,
    "timestamp": 390705261924637,
    "dispatch_id": 29,
    "corr_id": {
      "internal": 29,
      "external": 0
    },
    "wrkgrp_id": {
      "x": 9,
      "y": 489,
      "z": 0
    },
    "wave_in_grp": 0,
    "wave_issued": 1,
    "inst_type": "ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_VALU",
    "wave_cnt": 6,
    "snapshot": {
      "stall_reason": "ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_OTHER_WAIT",
      "dual_issue_valu": 0,
      "arb_state_issue_valu": 1,
      "arb_state_issue_matrix": 0,
      "arb_state_issue_lds": 0,
      "arb_state_issue_lds_direct": 0,
      "arb_state_issue_scalar": 0,
      "arb_state_issue_vmem_tex": 0,
      "arb_state_issue_flat": 0,
      "arb_state_issue_exp": 0,
      "arb_state_issue_misc": 0,
      "arb_state_issue_brmsg": 0,
      "arb_state_stall_valu": 0,
      "arb_state_stall_matrix": 0,
```


(continues on next page)

(continued from previous page)

```
      "arb_state_stall_lds": 0,
      "arb_state_stall_lds_direct": 0,
      "arb_state_stall_scalar": 0,
      "arb_state_stall_vmem_tex": 0,
      "arb_state_stall_flat": 0,
      "arb_state_stall_exp": 0,
      "arb_state_stall_misc": 0,
      "arb_state_stall_brmsg": 0
    }
  },
  "inst_index": 1
},
```

Fields starting with `arb_state_` are of particular interest as they indicate the state of the arbiter at the time of sampling. Namely, `arb_state_issue_` fields indicate what type of instructions arbiter issued at the time of sampling. On the other hand, `arb_state_stall_` fields indicate what type of instructions were stalled at the time of sampling. This information is useful for understanding how many instructions per cycle (IPC) are issued.

CHAPTER

TEN

USING THREAD TRACE

Thread trace is a shader execution tracing technique capable of profiling wavefronts at the instruction timing level. This is a low-level tracing and profiling feature that targets a single or a few kernel executions.

Thread trace features include:

- Near cycle-accurate instruction tracing
- Exact thread or wave execution path
- Wave scheduling and stall timing analysis
- Instruction and source level hotspots
- Extremely fast and granular counter collection (AMD Instinct)

Supported devices:

- AMD Instinct: MI200 and MI300 series
- AMD Radeon: gfx10, gfx11 and gfx12

Thread trace profiling is performed in the following steps:

1. Tracing (data collection) - Uses ROCprofiler-SDK thread trace service API
2. Decoding (analysis) - Uses ROCprof Trace Decoder API
3. Visualization - Requires ROCprof Compute Viewer

Tracing and decoding is handled by `rocprofv3` while visualization is handled by the ROCprof Compute Viewer.

10.1 Prerequisites

- aqlprofile:
  - ROCm 7.x build, or
  - Early release can be built from source
  - Otherwise, `rocprofv3` throws error "INVALID_SHADER_DATA" or "Agent not supported".
- Installation of ROCprof Trace Decoder component:
  - For binary files, see ROCprof trace decoder release page.
  - Default install location is `/opt/rocm/lib`
  - For custom location, use:
    - Parameter `--att-library-path`, or
    - Environment variable `ROCPROF_ATT_LIBRARY_PATH`

10.2 rocprofv3 parameters for thread tracing

To collect thread trace with default parameters, use:

```
rocprofv3 --att -d <output_dir> -- <application_path>
```

The following table lists the parameters relevant to thread tracing:

Parameter	Type	Range	Typical	Description
att-target-cu	Integer	0 - 15	1	Defines the CU used to gather detail tokens (WGP on Navi)
att-shader-engine-mask	Bitmask	1 - ~0u	0x1	Defines the Shader Engines (SE) to be traced. Max 2^32 - 1
att-simd-select	Integer	0 - 0xF	gfx9: 0xF Navi: 0x0	Defines one or more SIMDs to be traced, out of four. Bitmask on GFX9 and SIMD_ID[0,3] on Navi.
kernel-iteration-range	List			Defines dispatch iteration of the kernel to be profiled
kernel-include-regex	Strin	Any		Profiles kernel names matching the regex
kernel-exclude-regex	Strin	Any		Doesn't profile kernel names matching the regex
att-buffer-size	Bytes	1MB 2GB	96MB	Specifies the trace buffer size. This is shared for all SEs. Increase this value if the buffer tends to get full.
att-serialize-all	Bool		False	If set to "True", turns on serialization for untraced kernels
att-perfcounte ctrl	Integer	1 - 32	2~8	Available only in gfx9. Streams SQ performance counters to the thread trace buffer in the given relative period. As this uses high bandwidth, a value too low can cause or worsen "Data Lost" events and warnings.
att-perfcounte	Strin	SQ-only		Available only in gfx9. Specifies the list of SQ counters. To list all counters, use "rocprofv3 –list-avail``.
att-activity	Integer	1 - 16	5~10	Available only in gfx9. Shorthand for att-perfcounter-ctrl and the att-perfcounters related to compute unit activity such as VALU, SALU, etc.

For AMD Instinct accelerators, enable perfmon streaming using:

```
rocprofv3 --att --att-activity 8 -- <application_path>
```

For AMD Radeon, the `simd-select` parameter is a SIMD ID defaulting to 3. For some applications it's best to use:

```
rocprofv3 --att --att-simd-select 0x0 -- <application_path>
```

10.3 Using input file

As explained in the preceding section, you can specify parameters on the command line or use a JSON input file:

```
{
    "jobs": [
        {
            "advanced_thread_trace": true,
            "att_target_cu": 1,
            "att_shader_engine_mask": "0x1",
            "att_simd_select": "0xF",
            "att_buffer_size": "0x6000000"
        }
    ]
}
```

10.4 Thread tracing for multiple kernel instances

By default, `rocprofv3` enables thread trace only once per kernel instance. This implies that if an application launches the same kernel multiple times, only the first instance will be traced. To enable thread trace for multiple kernel instances, use the `kernel-iteration-range` parameter. It's recommended to use `kernel-include-regex` parameter to filter the desired kernel names instead of tracing everything.

10.5 rocprofv3 output files

After the application finishes executing, ROCprof Trace Decoder runs automatically and the following output files are generated:

- stats_*.csv files:
  - Contains a summary of instruction latency per kernel.
- ui_output_agent_{agent_id}_dispatch_{dispatch_id} directory:
  - Contains detailed tracing information in the form of .json files.
  - This directory can be opened using the ROCprof Compute Viewer.
- Raw files:
  - .att - Raw SQTT data. Can be used with the ROCprof Trace Decoder for further analysis.
  - .out - Code object binaries (executable). Can be used with ISA analysis tools.

10.5.1 Stats CSV

Here is a sample stats_*.csv file that is generated by the rocprofv3 tool.

Codeobj	Vaddr	Instruction	Hit-count	La-tency	Stall	Idle	Source
11	5888	s_load_dwordx4 s[40:43], s[0:1], 0x18	48	276	96	48	ker-nel.py:391
11	5896	s_load_dwordx2 s[38:39], s[0:1], 0x28	48	192	0	0	ker-nel.py:391
11	5904	s_ashr_i32 s3, s2, 31	48	260	0	0	ker-nel.py:395
11	5908	s_add_i32 s7, s2, s3	48	196	0	0	ker-nel.py:395

The columns of the stats_*.csv file are described here:

- **Codeobj:** The code object load ID assigned by ROCprofiler-SDK.
- **Vaddr:** ELF vaddr.
- **Hitcount:** The number of times a particular instruction is executed while adding all the traced waves.
- **Latency:** Total latency in cycles, defined as "Stall time + Issue time" for gfx9 or "Stall time + Execute time" for gfx10+.
- **Stall:** The total number of cycles the hardware pipe couldn't issue an instruction.
  - Usually caused when the hardware unit is busy, such as TCP or LDS backpressure.
- **Idle:** The total time gap between the completion of previous instruction and the beginning of the current instruction. The idle time can be caused by:
  - Arbiter loss
  - Source or destination register dependency
  - Instruction cache miss
- **Source:** The original source line of code assigned by the compiler.
  - Requires compiling with debug symbols.

10.6 Troubleshooting

For some applications, stats_*.csv file could be empty even for a valid kernel dispatch. Thread trace is limited to a single CU per SE (`att-target-cu`). If a kernel dispatch doesn't launch enough waves to populate the whole GPU, there's a possibility of no wave getting assigned to the `target_cu`. In such cases, there's nothing to be traced. Here are some options to handle this:

- Launch more waves.
- Swap the `target_cu`.
- Set the `--att-shader-engine-mask` to 0x11111111, or possibly to 0xFFFFFFFF
  - A number too high can cause packet losses and/or lead to a full buffer.
- Set the `HSA_CU_MASK` to mask out all CUs but the target. For more details, see setting CUs.
  - If only the `target_cu` (or a few CUs) are not masked out, then all or most waves will be assigned to the `target_cu`.
  - This can potentially cause low performance in high-demanding kernels.

CHAPTER

ELEVEN

ROCPROFILER-SDK TOOL LIBRARY

The tool library utilizes APIs from `rocprofiler-sdk` and `rocprofiler-register` libraries for profiling and tracing HIP applications. This document provides information to help you design a tool by utilizing the `rocprofiler-sdk` and `rocprofiler-register` libraries efficiently. The command-line tool `rocprofv3` is also built on `librocprofiler-sdk-tool.so.X.Y.Z`, which uses these libraries.

11.1 ROCm runtimes design

The ROCm runtimes are designed to directly communicate with a helper library named `rocprofiler-register` during initialization. This library performs cursory checks to find if a tool requires ROCprofiler-SDK services. This detection is based on the presence of one or more instances of `rocprofiler_configure` in the tool or `ROCP_TOOL_LIBRARIES` environment variable. This design provides drastic improvement over previous designs, which relied solely on a tool racing to set runtime-specific environment variables like `HSA_TOOLS_LIB` before the runtime initialization.

11.2 Tool library design

When ROCprofiler-SDK detects `rocprofiler_configure` in a tool's symbol table, ROCprofiler-SDK invokes `rocprofiler_configure` with parameters such as ROCprofiler-SDK version that invokes the function, number of tools already invoked, and a unique identifier for the tool. The tool returns a pointer to a `rocprofiler_tool_configure_result_t` struct, which, if non-null, provides ROCprofiler-SDK with:

- Function to be called for tool initialization, which is also the opportunity for context creation.
- Function to be called when ROCprofiler-SDK is finalized.
- A pointer to data to be provided to the tool when ROCprofiler-SDK calls the initialization and finalization functions.

ROCprofiler-SDK provides a `rocprofiler-sdk/registration.h` header file, which forward declares the `rocprofiler_configure` function with the necessary compiler function attributes to ensure that the `rocprofiler_configure` symbol is publicly visible.

```
#include <rocprofiler-sdk/registration.h>

namespace
{
struct ToolData
{
    uint32_t                                version;
    const char*                             runtime_version;
    uint32_t                                priority;
```

(continues on next page)

(continued from previous page)

```
    rocprofiler_client_id_t               client_id;
};

int
tool_init(rocprofiler_client_finalize_t fini_func,
          void* tool_data_v);

void
tool_fini(void* tool_data_v);
}

extern "C"
{
rocprofiler_tool_configure_result_t*
rocprofiler_configure(uint32_t                 version,
                      const char*              runtime_version,
                      uint32_t                 priority,
                      rocprofiler_client_id_t* client_id)
{
    //If not the first tool to register, indicate that the tool doesn't want to do␣
↪anything
    if(priority > 0) return nullptr;

    // (optional) Provide a name for this tool to rocprofiler
    client_id->name = "ExampleTool";

    // (optional) create configure data
    static auto data = ToolData{ version,
                                 runtime_version,
                                 priority,
                                 client_id };

    // construct configure result
    static auto cfg =
        rocprofiler_tool_configure_result_t{ sizeof(rocprofiler_tool_configure_result_t),
                                             &tool_init,
                                             &tool_fini,
                                             static_cast<void*>(&data) };

    return &cfg;
}
```

Note

ROCprofiler-SDK does NOT support calls to any runtime function (HSA, HIP, and so on) during tool initialization. Invoking any functions from the runtimes results in a deadlock.

For each tool that contains a `rocprofiler_configure` function and returns a non-null pointer to a `rocprofiler_tool_configure_result_t` struct, ROCprofiler-SDK invokes the `initialize` callback after completing the scan for all `rocprofiler_configure` symbols. In other words, ROCprofiler-SDK collects all `rocprofiler_tool_configure_result_t` instances before invoking the `initialize` member of any of these in-

stances. When ROCprofiler-SDK invokes `initialize` function in a tool, this is the opportunity to create contexts:

```
#include <rocprofiler-sdk/rocprofiler.h>

namespace
{
int
tool_init(rocprofiler_client_finalize_t fini_func,
          void* data_v)
{
    // create a context
    auto ctx = rocprofiler_context_id_t{0};
    rocprofiler_create_context(&ctx);

    // ... associate services with context ...

    // start the context (optional)
    rocprofiler_start_context(ctx);

    return 0;
}
}
```

Although not mandatory, it is recommended that tools store the context handles to control the data collection for the services associated with the context.

11.3 Tool finalization

When the *initialize* callback is invoked in the tool, ROCprofiler-SDK provides a function pointer of type *rocprofiler_client_finalize_t*. The tool can invoke this function pointer to explicitly invoke the *finalize* callback from the *rocprofiler_tool_configure_result_t* instance:

```
#include <rocprofiler-sdk/rocprofiler.h>

namespace
{
    int
    tool_init(rocprofiler_client_finalize_t fini_func,
              void* data_v)
    {
        // ... see initialization section ...

        // function, which
        auto explicit_finalize = [](rocprofiler_client_finalize_t finalizer,
                                    rocprofiler_client_id_t* client_id)
        {
            std::this_thread::sleep_for(std::chrono::seconds{ 10 });
            finalizer(client_id);
        };

        // start the context
        rocprofiler_start_context(ctx);
```

(continues on next page)

(continued from previous page)

```
        // dispatch a background thread to explicitly finalize after 10 seconds
        std::thread{ explicit_finalize, fini_func, static_cast<ToolData*>(data_v)->
↪client_id }.detach();

        return 0;
    }
}
```

Otherwise, ROCprofiler-SDK invokes the *finalize* callback via an *atexit* handler.

11.4 Full rocprofiler_configure sample

All the code snippets from the previous sections are combined here to demonstrate complete ROCProfiler configuration.

```
#include <rocprofiler-sdk/registration.h>

namespace
{
struct rocp_tool_data
{
    uint32_t                              version;
    const char*                           runtime_version;
    uint32_t                              priority;
    rocprofiler_client_id_t               client_id;
    rocprofiler_client_finalize_t         finalizer;
    std::vector<rocprofiler_context_id_t> contexts;
};

void
tool_tracing_callback(rocprofiler_callback_tracing_record_t record,
                      rocprofiler_user_data_t*              user_data,
                      void*                                 callback_data);

int
tool_init(rocprofiler_client_finalize_t fini_func,
          void* tool_data_v)
{
    rocp_tool_data* tool_data = static_cast<rocp_tool_data*>(tool_data_v);

    // Save the finalizer function
    tool_data->finalizer = fini_func;

    // create a context
    auto ctx = rocprofiler_context_id_t{0};
    rocprofiler_create_context(&ctx);

    // Save your contexts
    tool_data->contexts.emplace_back(ctx);

    // Associate code object tracing with this context
    rocprofiler_configure_callback_tracing_service(
        ctx,
```

(continues on next page)

(continued from previous page)

```
        ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT,
        nullptr,
        0,
        tool_tracing_callback,
        tool_data);

    // ... Associate services with contexts ...

    return 0;
}

void
tool_fini(void* tool_data);
}

extern "C"
{
rocprofiler_tool_configure_result_t*
rocprofiler_configure(uint32_t                 version,
                      const char*              runtime_version,
                      uint32_t                 priority,
                      rocprofiler_client_id_t* client_id)
{
    // (optional) Provide a name for this tool to rocprofiler
    client_id->name = "ExampleTool";

    // Info provided back to tool_init and tool_fini
    auto* my_tool_data = new rocp_tool_data{ version,
                                             runtime_version,
                                             priority,
                                             client_id,
                                             nullptr };

    // Create configure data
    static auto cfg =
        rocprofiler_tool_configure_result_t{ sizeof(rocprofiler_tool_configure_result_t),
                                             &tool_init,
                                             &tool_fini,
                                             my_tool_data };

    return &cfg;
}
```

CHAPTER

TWELVE

RUNTIME INTERCEPT TABLES

While tools commonly leverage the callback or buffer tracing services for tracing the HIP, HSA, and ROCTx APIs, ROCprofiler-SDK also provides access to the raw API dispatch tables.

12.1 Forward declaration of public C API function

All the aforementioned APIs are designed similar to the following sample:

```
extern "C"
{
// forward declaration of public C API function
int
foo(int) __attribute__((visibility("default")));
}
```

12.2 Internal implementation of API function

```
namespace impl
{
int
foo(int val)
{
    // real implementation
    return (2 * val);
}
}
```

12.3 Dispatch table implementation

```
namespace impl
{
struct dispatch_table
{
    int (*foo_fn)(int) = nullptr;
};

// Invoked once: populates the dispatch_table with function pointers to implementation
dispatch_table*&
```

(continues on next page)

(continued from previous page)

```
construct_dispatch_table()
{
    static dispatch_table* tbl = new dispatch_table{};
    tbl->foo_fn                = impl::foo;

    // In between, ROCprofiler-SDK gets passed the pointer
    // to the dispatch table and has the opportunity to wrap the function
    // pointers for interception

    return tbl;
}

// Constructs dispatch table and stores it in static variable
dispatch_table*
get_dispatch_table()
{
    static dispatch_table*& tbl = construct_dispatch_table();
    return tbl;
}
}  // namespace impl
```

12.4 Implementation of public C API function

```
extern "C"
{
// implementation of public C API function
int
foo(int val)
{
    return impl::get_dispatch_table()->foo_fn(val);
}
}
```

12.5 Dispatch table chaining

ROCprofiler-SDK can save the original values of the function pointers such as `foo_fn` in `impl::construct_dispatch_table()` and install its own function pointers in its place. This results in the public C API function `foo` calling into the ROCprofiler-SDK function pointer, which in turn, calls the original function pointer to `impl::foo`. This phenomenon is named chaining. Once ROCprofiler-SDK makes necessary modifications to the dispatch table, tools requesting access to the raw dispatch table via `rocprofiler_at_intercept_table_registration` are provided the pointer to the dispatch table.

For examples on dispatch table chaining, see samples/intercept_table.

CHAPTER

THIRTEEN

IMPLEMENTING PROCESS ATTACHMENT TOOLS

13.1 Overview

This document provides the technical details needed to implement a process attachment tool similar to `rocprofv3 --attach`. Process attachment allows profiling tools to dynamically attach to running GPU applications without requiring application restart.

The implementation uses specific exported C functions and involves low-level process manipulation using ptrace, environment variable injection, library loading, and coordination with the ROCprofiler-SDK registration system.

13.2 Exported C Functions for Attachment

The attachment functionality provides the following exported C functions that tools can use:

13.2.1 ROCprofiler-Attach Functions

These functions are exported from the `rocprofiler-attach` binary:

```
extern "C" {
    // Start attachment to a target process
    void attach(uint32_t pid) ROCPROFILER_EXPORT;

    // Detach from target process and cleanup
    void detach() ROCPROFILER_EXPORT;
}
```

Function Details:

- ``**attach(uint32_t pid)**``: Main entry point for starting attachment to a process - Takes the target process ID as parameter - Initiates ptrace-based attachment sequence - Spawns background thread for ptrace operations
- ``**detach()**``: Entry point for detaching from the target process - Cleans up attachment resources and terminates profiling - Joins ptrace thread and releases resources

13.2.2 ROCprofiler-Register Functions

These functions are exported from the `librocprofiler-register.so` library and are called via ptrace:

```
extern "C" {
    // Activate profiling in target process (called via ptrace)
    rocprofiler_register_error_code_t
    rocprofiler_register_attach(const char* environment_buffer, const char* tool_lib_
```

(continues on next page)

(continued from previous page)

```
↪path)
        ROCPROFILER_REGISTER_PUBLIC_API;

    // Deactivate profiling in target process (called via ptrace)
    rocprofiler_register_error_code_t
    rocprofiler_register_detach()
        ROCPROFILER_REGISTER_PUBLIC_API;

    // Reattach to previously attached process (experimental)
    rocprofiler_register_error_code_t
    rocprofiler_register_invoke_reattach()
        ROCPROFILER_REGISTER_PUBLIC_API;

    // Client callback functions for reattachment support
    void rocprofiler_call_client_reattach(void)
        ROCPROFILER_REGISTER_PUBLIC_API;
    void rocprofiler_call_client_detach(void)
        ROCPROFILER_REGISTER_PUBLIC_API;
}
```

Function Details:

- ``**rocprofiler_register_attach(const char* environment_buffer, const char* tool_lib_path)**``: - Called via ptrace from the attachment system - Receives serialized environment variables for profiling configuration - Receives the tool library path to load (defaults to "librocprofiler-sdk-tool.so" if NULL) - Loads the specified tool library and activates profiling services - Returns `rocprofiler_register_error_code_t` status
- ``**rocprofiler_register_detach()**``: - Called via ptrace to stop profiling in the target process - Calls the tool's detach function and cleans up resources - Returns `rocprofiler_register_error_code_t` status
- ``**rocprofiler_register_invoke_reattach()**``: (EXPERIMENTAL) - Called to reattach profiling to a previously attached process - Invokes client reattach callbacks without full re-initialization - Used for resuming profiling after temporary detachment - Returns `rocprofiler_register_error_code_t` status
- ``**rocprofiler_call_client_reattach()** and ``**rocprofiler_call_client_detach()**``: - C wrapper functions for client tool reattachment callbacks - Automatically resolved and called by the registration system - Enable tools to handle dynamic attach/detach cycles

13.3 Function Call Sequence

13.3.1 Initial Attachment Sequence

The initial attachment process follows this sequence:

```
Tool Implementation
     |
     v
attach(pid) ← Your tool calls this
     |
     v
Ptrace attachment & environment setup
     |
     v
rocprofiler_register_attach(env_buffer) ← Called via ptrace in target
```

(continues on next page)

(continued from previous page)

```
     |
     v
Profiling active in target process
     |
     v
[Profiling data collection...]
     |
     v
rocprofiler_register_detach() ← Called via ptrace in target
     |
     v
detach() ← Your tool calls this
     |
     v
Cleanup complete
```

13.3.2 Reattachment Sequence (Experimental)

For reattachment to a previously attached process:

```
Tool Implementation
     |
     v
attach(pid) ← Your tool calls this again
     |
     v
Ptrace attachment & environment setup
     |
     v
rocprofiler_register_attach(env_buffer) ← Detects previous attachment
     |
     v
rocprofiler_register_invoke_reattach() ← Calls client reattach callbacks
     |
     v
Profiling resumed in target process
     |
     v
[Continued profiling data collection...]
     |
     v
rocprofiler_register_detach() ← Called via ptrace in target
     |
     v
detach() ← Your tool calls this
     |
     v
Cleanup complete
```

13.4 Using the Attachment Functions

Here's how to use these functions in your own attachment tool:

13.4.1 Basic Attachment Tool Implementation

```
#include <dlfcn.h>
#include <iostream>
#include <thread>
#include <chrono>

class ROCprofilerAttachmentTool {
private:
    void* attach_lib_handle = nullptr;
    void (*attach_func)(uint32_t) = nullptr;
    void (*detach_func)() = nullptr;

public:
    bool initialize() {
        // Load the rocprofiler-attach library/binary
        attach_lib_handle = dlopen("librocprofiler-attach.so", RTLD_NOW);
        if (!attach_lib_handle) {
            std::cerr << "Failed to load rocprofiler-attach: " << dlerror() << std::endl;
            return false;
        }

        // Get the attachment function pointers
        attach_func = (void(*)(uint32_t))dlsym(attach_lib_handle, "attach");
        detach_func = (void(*)())dlsym(attach_lib_handle, "detach");

        if (!attach_func || !detach_func) {
            std::cerr << "Failed to find attachment functions" << std::endl;
            return false;
        }

        return true;
    }

    bool attach_to_process(pid_t pid, uint32_t duration_ms = 0) {
        // Validate the target process
        if (kill(pid, 0) != 0) {
            std::cerr << "Target process " << pid << " is not accessible" << std::endl;
            return false;
        }

        std::cout << "Attaching to process " << pid << std::endl;

        // Start attachment - this will handle all ptrace operations
        attach_func(pid);

        if (duration_ms > 0) {
            // Profile for specified duration
            std::cout << "Profiling for " << duration_ms << " milliseconds..." <<␣
```

(continues on next page)

(continued from previous page)

```
↪std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(duration_ms));

            // Stop profiling
            detach_func();
        } else {
            std::cout << "Profiling until process ends or manual detach..." << std::endl;
            // Monitor process or wait for external signal to detach
            while (kill(pid, 0) == 0) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            detach_func();
        }

        std::cout << "Profiling completed" << std::endl;
        return true;
    }

    ~ROCprofilerAttachmentTool() {
        if (attach_lib_handle) {
            dlclose(attach_lib_handle);
        }
    }
};
```

13.4.2 Complete Tool Example

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <PID> [duration_ms]" << std::endl;
        std::cerr << "  PID: Process ID to attach to" << std::endl;
        std::cerr << "  duration_ms: Optional profiling duration in milliseconds" <<␣
↪std::endl;
        return 1;
    }

    pid_t target_pid = std::stoi(argv[1]);
    uint32_t duration = (argc > 2) ? std::stoi(argv[2]) : 0;

    // Set up profiling environment variables before attachment
    setenv("ROCP_TOOL_ATTACH", "1", 1);

    // Note: The attachment system now uses the hardcoded default tool library path
    // "librocprofiler-sdk-tool.so" and no longer uses environment variables for tool␣
↪selection
```

(continues on next page)

(continued from previous page)

```
    setenv("ROCPROF_HIP_API_TRACE", "1", 1);
    setenv("ROCPROF_KERNEL_TRACE", "1", 1);
    setenv("ROCPROF_MEMORY_COPY_TRACE", "1", 1);
    setenv("ROCPROF_OUTPUT_PATH", "./attachment-output", 1);
    setenv("ROCPROF_OUTPUT_FILE_NAME", "attached_profile", 1);

    // Initialize and run attachment tool
    ROCprofilerAttachmentTool tool;
    if (!tool.initialize()) {
        std::cerr << "Failed to initialize attachment tool" << std::endl;
        return 1;
    }

    if (!tool.attach_to_process(target_pid, duration)) {
        std::cerr << "Attachment failed" << std::endl;
        return 1;
    }

    std::cout << "Attachment completed successfully" << std::endl;
    return 0;
}
```

13.5 Experimental Reattachment API

ROCprofiler-SDK now provides experimental support for reattachment, allowing tools to handle dynamic attach/detach cycles more efficiently.

13.5.1 Tool Configuration for Reattachment

Tools that support reattachment should implement the experimental configuration structure:

```
#include <rocprofiler-sdk/registration.h>

// Experimental reattachment callbacks
void tool_reattach(void* tool_data) {
    // Reinitialize contexts and resume profiling
    // This is called when reattaching to a previously profiled process
}

void tool_detach(void* tool_data) {
    // Suspend profiling operations temporarily
    // This is called during detachment, but contexts may be preserved
}

extern "C" rocprofiler_tool_configure_result_experimental_t*
rocprofiler_configure_experimental(uint32_t                 version,
                                   const char*              runtime_version,
                                   uint32_t                 prio,
                                   rocprofiler_client_id_t* client_id)
{
    static auto cfg = rocprofiler_tool_configure_result_experimental_t {
```

(continues on next page)

(continued from previous page)

```
        .size = sizeof(rocprofiler_tool_configure_result_experimental_t),
        .initialize = &tool_init,
        .finalize = &tool_fini,
        .tool_data = nullptr,
        .tool_reattach = &tool_reattach,  // Experimental reattachment support
        .tool_detach = &tool_detach       // Experimental detachment support
    };

    return &cfg;
}
```

13.5.2 Client Callback Functions

The registration system automatically provides C wrapper functions:

```
// These are automatically generated and called by rocprofiler-register
extern "C" void rocprofiler_call_client_reattach(void) {
    // Calls the tool's reattach callback with stored tool_data
}

extern "C" void rocprofiler_call_client_detach(void) {
    // Calls the tool's detach callback with stored tool_data
}
```

13.5.3 Reattachment Environment Variables

When using reattachment, set this additional environment variable:

```
// Indicates that the tool was loaded via attachment (not LD_PRELOAD)
setenv("ROCPROFILER_REGISTER_TOOL_ATTACHED", "1", 1);
```

This helps the registration system differentiate between initial attachment and reattachment cycles.

13.6 Environment Variable Configuration

Before calling the attachment functions, set up environment variables that will be injected into the target process:

13.6.1 Required Variables

```
// Essential for attachment functionality
setenv("ROCP_TOOL_ATTACH", "1", 1);
```

13.6.2 Tool Library Configuration

The attachment system now uses a hardcoded default tool library path:

```
// The attachment system automatically uses "librocprofiler-sdk-tool.so"
// No environment variable configuration is needed or supported
```

13.6.3 Tracing Options

```
// Enable different types of tracing
setenv("ROCPROF_HIP_API_TRACE", "1", 1);           // HIP API calls
setenv("ROCPROF_HSA_API_TRACE", "1", 1);           // HSA API calls
setenv("ROCPROF_KERNEL_TRACE", "1", 1);            // Kernel dispatches
setenv("ROCPROF_MEMORY_COPY_TRACE", "1", 1);       // Memory operations
setenv("ROCPROF_MEMORY_ALLOCATION_TRACE", "1", 1); // Memory allocations
setenv("ROCPROF_SCRATCH_MEMORY_TRACE", "1", 1);    // Scratch memory
setenv("ROCPROF_MARKER_TRACE", "1", 1);            // ROCTx markers
```

13.6.4 Output Configuration

```
// Control output location and format
setenv("ROCPROF_OUTPUT_PATH", "/path/to/output", 1);
setenv("ROCPROF_OUTPUT_FILE_NAME", "profile_name", 1);
setenv("ROCPROF_OUTPUT_FORMAT", "csv", 1);  // or "json", "pftrace", etc.
```

13.7 Build Configuration

To build a tool using the attachment functions:

13.7.1 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)
project(my_rocprofiler_attach_tool)

set(CMAKE_CXX_STANDARD 17)

# Find ROCprofiler SDK (for headers and linking)
find_package(rocprofiler-sdk REQUIRED)

add_executable(my_attach_tool
    main.cpp
    attachment_tool.cpp
)

# Link with required libraries
target_link_libraries(my_attach_tool
    rocprofiler-sdk::rocprofiler-sdk
    dl  # for dlopen/dlsym operations
)

# Set capabilities for ptrace operations
add_custom_command(TARGET my_attach_tool POST_BUILD
    COMMAND sudo setcap cap_sys_ptrace+ep $<TARGET_FILE:my_attach_tool>
    COMMENT "Setting ptrace capability"
)
```

13.8 Error Handling

When using the attachment functions, handle these common error conditions:

```
class AttachmentErrorHandler {
public:
    static bool validate_target_process(pid_t pid) {
        // Check if process exists
        if (kill(pid, 0) != 0) {
            std::cerr << "Process " << pid << " not found or not accessible" <<␣
↪std::endl;
            return false;
        }

        // Check if it's a GPU application
        std::string maps_path = "/proc/" + std::to_string(pid) + "/maps";
        std::ifstream maps(maps_path);
        std::string line;

        bool has_gpu_libs = false;
        while (std::getline(maps, line)) {
            if (line.find("libamdhip64.so") != std::string::npos ||
                line.find("libhsa-runtime64.so") != std::string::npos) {
                has_gpu_libs = true;
                break;
            }
        }

        if (!has_gpu_libs) {
            std::cerr << "Process " << pid << " does not appear to use GPU APIs" <<␣
↪std::endl;
            return false;
        }

        return true;
    }

    static void handle_attachment_errors() {
        // Check for common permission issues
        if (geteuid() != 0) {
            std::cerr << "Warning: Not running as root. Ensure CAP_SYS_PTRACE capability␣
↪is set." << std::endl;
        }

        // Check if rocprofiler libraries are available
        if (getenv("LD_LIBRARY_PATH") == nullptr ||
            std::string(getenv("LD_LIBRARY_PATH")).find("/opt/rocm/lib") ==␣
↪std::string::npos) {
            std::cerr << "Warning: /opt/rocm/lib may not be in LD_LIBRARY_PATH" <<␣
↪std::endl;
        }
    }
};
```

13.9 Architecture Overview

Process attachment consists of several cooperating components:

```
Attachment Tool (your implementation)
     |
     v
1. Process Discovery & Validation
     |
     v
2. Ptrace Attachment & Control
     |
     v
3. Environment Variable Injection
     |
     v
4. Library Loading (rocprofiler-register)
     |
     v
5. Profiling Service Activation
     |
     v
6. Data Collection & Management
     |
     v
7. Detachment & Cleanup
```

13.10 Theoretical Implementation Details

13.11 Core Implementation Components

13.11.1 1. Process Discovery and Validation

Target Process Requirements:

```cpp
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

bool validate_target_process(pid_t pid) {
    // Check if process exists and is accessible
    if (kill(pid, 0) != 0) {
        return false;  // Process doesn't exist or no permission
    }

    // Verify it's a GPU application by checking loaded libraries
    std::string maps_path = "/proc/" + std::to_string(pid) + "/maps";
    std::ifstream maps(maps_path);
    std::string line;

    bool has_hip = false, has_hsa = false;
    while (std::getline(maps, line)) {
```

(continues on next page)

(continued from previous page)

```
        if (line.find("libamdhip64.so") != std::string::npos) has_hip = true;
        if (line.find("libhsa-runtime64.so") != std::string::npos) has_hsa = true;
    }

    return has_hip || has_hsa;  // Must use HIP or HSA
}
```

13.11.2 2. Ptrace-Based Process Control

Core Ptrace Operations:

```
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/user.h>

class ProcessAttachment {
private:
    pid_t target_pid;
    bool attached = false;

public:
    bool attach(pid_t pid) {
        target_pid = pid;

        // Attach to the target process
        if (ptrace(PTRACE_ATTACH, target_pid, nullptr, nullptr) == -1) {
            perror("ptrace PTRACE_ATTACH failed");
            return false;
        }

        // Wait for the process to stop
        int status;
        if (waitpid(target_pid, &status, 0) == -1) {
            perror("waitpid failed");
            detach();
            return false;
        }

        if (!WIFSTOPPED(status)) {
            fprintf(stderr, "Process did not stop after attach\n");
            detach();
            return false;
        }

        attached = true;
        return true;
    }

    bool detach() {
        if (!attached) return true;

        // Detach and allow process to continue
```

(continues on next page)

(continued from previous page)

```
        if (ptrace(PTRACE_DETACH, target_pid, nullptr, nullptr) == -1) {
            perror("ptrace PTRACE_DETACH failed");
            return false;
        }

        attached = false;
        return true;
    }
};
```

13.11.3 3. Environment Variable Injection

Environment Variable Management:

```
#include <fstream>
#include <vector>

class EnvironmentInjector {
public:
    struct EnvironmentVar {
        std::string name;
        std::string value;
    };

    // Prepare environment variables for profiling
    std::vector<EnvironmentVar> prepare_profiling_env(
        const std::vector<std::string>& trace_options,
        const std::string& output_path,
        const std::string& output_file) {

        std::vector<EnvironmentVar> env_vars;

        // Essential attachment variable
        env_vars.push_back({"ROCP_TOOL_ATTACH", "1"});

        // Configure tracing based on options
        for (const auto& option : trace_options) {
            if (option == "hip-trace") {
                env_vars.push_back({"ROCPROF_HIP_API_TRACE", "1"});
            }
            if (option == "kernel-trace") {
                env_vars.push_back({"ROCPROF_KERNEL_TRACE", "1"});
            }
            if (option == "hsa-trace") {
                env_vars.push_back({"ROCPROF_HSA_API_TRACE", "1"});
            }
            if (option == "memory-copy-trace") {
                env_vars.push_back({"ROCPROF_MEMORY_COPY_TRACE", "1"});
            }
        }

        // Output configuration
```

(continues on next page)

(continued from previous page)

```
        env_vars.push_back({"ROCPROF_OUTPUT_PATH", output_path});
        env_vars.push_back({"ROCPROF_OUTPUT_FILE_NAME", output_file});

        return env_vars;
    }

    // Serialize environment for injection
    std::vector<uint8_t> serialize_environment(const std::vector<EnvironmentVar>& vars) {
        std::vector<uint8_t> buffer(4);  // Start with count
        uint32_t count = vars.size();

        // Store count in first 4 bytes
        buffer[0] = count & 0xFF;
        buffer[1] = (count >> 8) & 0xFF;
        buffer[2] = (count >> 16) & 0xFF;
        buffer[3] = (count >> 24) & 0xFF;

        // Add each variable as null-terminated name and value
        for (const auto& var : vars) {
            // Add variable name
            for (char c : var.name) {
                buffer.push_back(c);
            }
            buffer.push_back(0);  // Null terminate name

            // Add variable value
            for (char c : var.value) {
                buffer.push_back(c);
            }
            buffer.push_back(0);  // Null terminate value
        }

        return buffer;
    }
};
```

13.11.4 4. Memory Manipulation and Library Loading

Remote Memory Operations:

```
#include <sys/mman.h>

class RemoteMemoryManager {
private:
    pid_t target_pid;

public:
    RemoteMemoryManager(pid_t pid) : target_pid(pid) {}

    // Allocate memory in remote process
    void* remote_mmap(size_t length, int prot, int flags) {
        // Find a suitable location for injection
```

(continues on next page)

(continued from previous page)

```
    struct user_regs_struct regs;
    if (ptrace(PTRACE_GETREGS, target_pid, nullptr, &regs) == -1) {
        return nullptr;
    }

    // Save original registers
    struct user_regs_struct orig_regs = regs;

    // Set up mmap syscall
    regs.rax = 9;  // __NR_mmap
    regs.rdi = 0;  // addr (let kernel choose)
    regs.rsi = length;
    regs.rdx = prot;
    regs.r10 = flags;
    regs.r8 = -1;  // fd
    regs.r9 = 0;   // offset

    if (ptrace(PTRACE_SETREGS, target_pid, nullptr, &regs) == -1) {
        return nullptr;
    }

    // Execute syscall
    if (ptrace(PTRACE_SYSCALL, target_pid, nullptr, nullptr) == -1) {
        return nullptr;
    }

    // Wait for syscall completion
    int status;
    waitpid(target_pid, &status, 0);

    // Get result
    if (ptrace(PTRACE_GETREGS, target_pid, nullptr, &regs) == -1) {
        return nullptr;
    }

    void* result = (void*)regs.rax;

    // Restore original registers
    ptrace(PTRACE_SETREGS, target_pid, nullptr, &orig_regs);

    return (result == (void*)-1) ? nullptr : result;
}

// Write data to remote process memory
bool write_memory(void* addr, const void* data, size_t size) {
    const uint8_t* bytes = static_cast<const uint8_t*>(data);
    size_t written = 0;

    while (written < size) {
        long word = 0;
        size_t to_copy = std::min(sizeof(long), size - written);
```

(continues on next page)

(continued from previous page)

```
            // For partial words, read existing content first
            if (to_copy < sizeof(long)) {
                errno = 0;
                word = ptrace(PTRACE_PEEKDATA, target_pid,
                              (uint8_t*)addr + written, nullptr);
                if (errno != 0) return false;
            }

            // Copy new data into word
            memcpy(&word, bytes + written, to_copy);

            // Write word to remote process
            if (ptrace(PTRACE_POKEDATA, target_pid,
                       (uint8_t*)addr + written, word) == -1) {
                return false;
            }

            written += to_copy;
        }

        return true;
    }
};
```

13.11.5 5. Library Injection and Symbol Resolution

Dynamic Library Loading:

```
#include <dlfcn.h>
#include <link.h>

class LibraryInjector {
private:
    pid_t target_pid;
    RemoteMemoryManager memory_manager;

public:
    LibraryInjector(pid_t pid) : target_pid(pid), memory_manager(pid) {}

    // Inject rocprofiler-register library
    bool inject_register_library() {
        const char* lib_path = "/opt/rocm/lib/librocprofiler-register.so";

        // Find dlopen in target process
        void* dlopen_addr = find_function_address("dlopen");
        if (!dlopen_addr) {
            fprintf(stderr, "Could not find dlopen in target process\n");
            return false;
        }

        // Allocate memory for library path
        void* path_addr = memory_manager.remote_mmap(
```

(continues on next page)

(continued from previous page)

```
            strlen(lib_path) + 1,
            PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS);

        if (!path_addr) return false;

        // Write library path to remote memory
        if (!memory_manager.write_memory(path_addr, lib_path, strlen(lib_path) + 1)) {
            return false;
        }

        // Call dlopen in target process
        return call_remote_function(dlopen_addr,
            {(uint64_t)path_addr, RTLD_NOW | RTLD_GLOBAL});
    }

    void* find_function_address(const char* function_name) {
        // Parse /proc/PID/maps to find loaded libraries
        std::string maps_path = "/proc/" + std::to_string(target_pid) + "/maps";
        std::ifstream maps(maps_path);
        std::string line;

        while (std::getline(maps, line)) {
            if (line.find("libc.so") != std::string::npos) {
                // Extract base address of libc
                size_t dash = line.find('-');
                std::string base_addr_str = line.substr(0, dash);
                void* base_addr = (void*)std::stoull(base_addr_str, nullptr, 16);

                // Open libc and find function offset
                void* handle = dlopen("libc.so.6", RTLD_LAZY);
                if (handle) {
                    void* func_addr = dlsym(handle, function_name);
                    if (func_addr) {
                        // Calculate actual address in target process
                        return (uint8_t*)base_addr + ((uint8_t*)func_addr - (uint8_
↪t*)dlsym(RTLD_DEFAULT, "main"));
                    }
                    dlclose(handle);
                }
            }
        }
        return nullptr;
    }
};
```

13.11.6 6. ROCprofiler-Register Communication Protocol

Attachment Protocol Implementation:

```
extern "C" {
    // Function signatures from rocprofiler-register
```

(continues on next page)

(continued from previous page)

```
    typedef void (*attach_func_t)(uint32_t pid);
    typedef void (*detach_func_t)();
}

class ROCprofilerAttachment {
private:
    pid_t target_pid;
    void* register_handle = nullptr;
    attach_func_t attach_func = nullptr;
    detach_func_t detach_func = nullptr;

public:
    bool initialize() {
        // Load rocprofiler-register library
        register_handle = dlopen("/opt/rocm/lib/librocprofiler-register.so", RTLD_NOW);
        if (!register_handle) {
            fprintf(stderr, "Failed to load rocprofiler-register: %s\n", dlerror());
            return false;
        }

        // Get attachment functions
        attach_func = (attach_func_t)dlsym(register_handle, "attach");
        detach_func = (detach_func_t)dlsym(register_handle, "detach");

        if (!attach_func || !detach_func) {
            fprintf(stderr, "Failed to find attachment functions\n");
            return false;
        }

        return true;
    }

    bool attach_to_process(pid_t pid, const std::vector<uint8_t>& env_buffer) {
        target_pid = pid;

        // Set up environment for rocprofiler-register
        // This involves injecting the environment buffer into the target process

        // Call the attach function
        attach_func(pid);

        return true;
    }

    void detach_from_process() {
        if (detach_func) {
            detach_func();
        }
    }
};
```

13.12 Complete Attachment Tool Implementation

Main Attachment Tool Structure:

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>
#include <thread>

class ROCprofilerAttachTool {
private:
    ProcessAttachment process_control;
    EnvironmentInjector env_injector;
    LibraryInjector lib_injector;
    ROCprofilerAttachment rocprof_attachment;

public:
    struct AttachmentConfig {
        pid_t target_pid;
        std::vector<std::string> trace_options;
        std::string output_path = "./rocprof-attachment-output";
        std::string output_filename = "attached_profile";
        uint32_t duration_msec = 0;  // 0 = until process ends
    };

    bool attach_and_profile(const AttachmentConfig& config) {
        // 1. Validate target process
        if (!validate_target_process(config.target_pid)) {
            std::cerr << "Invalid or inaccessible target process: " << config.target_pid
↪<< std::endl;
            return false;
        }

        // 2. Initialize rocprofiler attachment system
        if (!rocprof_attachment.initialize()) {
            std::cerr << "Failed to initialize rocprofiler attachment system" <<␣
↪std::endl;
            return false;
        }

        // 3. Attach to target process
        if (!process_control.attach(config.target_pid)) {
            std::cerr << "Failed to attach to process " << config.target_pid <<␣
↪std::endl;
            return false;
        }

        // 4. Prepare environment variables
        auto env_vars = env_injector.prepare_profiling_env(
            config.trace_options,
            config.output_path,
            config.output_filename);
```

(continues on next page)

(continued from previous page)

```
        auto env_buffer = env_injector.serialize_environment(env_vars);

        // 5. Inject rocprofiler-register library
        LibraryInjector injector(config.target_pid);
        if (!injector.inject_register_library()) {
            std::cerr << "Failed to inject rocprofiler-register library" << std::endl;
            process_control.detach();
            return false;
        }

        // 6. Activate profiling
        if (!rocprof_attachment.attach_to_process(config.target_pid, env_buffer)) {
            std::cerr << "Failed to activate profiling" << std::endl;
            process_control.detach();
            return false;
        }

        // 7. Allow process to continue with profiling active
        if (!process_control.detach()) {
            std::cerr << "Warning: Failed to detach cleanly" << std::endl;
        }

        // 8. Wait for specified duration or until process ends
        if (config.duration_msec > 0) {
            std::cout << "Profiling for " << config.duration_msec << " milliseconds..." <
↪< std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(config.duration_msec));

            // Re-attach to stop profiling
            rocprof_attachment.detach_from_process();
        } else {
            std::cout << "Profiling until process ends..." << std::endl;
            // Monitor process and wait for it to end
            while (kill(config.target_pid, 0) == 0) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
        }

        std::cout << "Profiling completed. Output saved to: "
                  << config.output_path << "/" << config.output_filename << std::endl;
        return true;
    }
};

// Example usage
int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <PID> [options]" << std::endl;
        return 1;
    }

    ROCprofilerAttachTool::AttachmentConfig config;
```

(continues on next page)

(continued from previous page)

```
    config.target_pid = std::stoi(argv[1]);
    config.trace_options = {"hip-trace", "kernel-trace", "memory-copy-trace"};
    config.duration_msec = 5000;  // 5 seconds

    ROCprofilerAttachTool tool;
    if (!tool.attach_and_profile(config)) {
        std::cerr << "Attachment and profiling failed" << std::endl;
        return 1;
    }

    return 0;
}
```

13.13 Required System Permissions and Setup

Permission Requirements:

```
# Your attachment tool will need:

# 1. Ptrace permissions (may require root or capabilities)
sudo setcap cap_sys_ptrace+ep your_attachment_tool

# 2. Access to /proc filesystem
# Usually available by default

# 3. Ability to load shared libraries
# Ensure ROCm libraries are in LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/opt/rocm/lib:$LD_LIBRARY_PATH
```

Build Requirements:

```
# CMakeLists.txt for your attachment tool
cmake_minimum_required(VERSION 3.16)
project(rocprofiler_attach_tool)

set(CMAKE_CXX_STANDARD 17)

find_package(rocprofiler-sdk REQUIRED)

add_executable(rocprofiler_attach_tool
    main.cpp
    process_attachment.cpp
    environment_injection.cpp
    library_injection.cpp
)

target_link_libraries(rocprofiler_attach_tool
    rocprofiler-sdk::rocprofiler-sdk
    dl  # for dlopen/dlsym
)
```

13.14 Error Handling and Debugging

Common Issues and Solutions:

1. **Ptrace Permissions**: Use `strace` to debug ptrace failures
2. **Library Loading**: Check `/proc/PID/maps` to verify library injection
3. **Environment Variables**: Validate environment buffer format
4. **Process State**: Monitor target process status during attachment

Debugging Techniques:

```
// Enable debug logging
setenv("ROCPROF_LOGGING_LEVEL", "trace", 1);

// Monitor attachment progress
bool debug_attachment(pid_t pid) {
    std::cout << "Target process memory maps:" << std::endl;
    std::string cmd = "cat /proc/" + std::to_string(pid) + "/maps";
    system(cmd.c_str());

    std::cout << "Target process environment:" << std::endl;
    cmd = "cat /proc/" + std::to_string(pid) + "/environ | tr '\\0' '\\n'";
    system(cmd.c_str());

    return true;
}
```

This implementation guide provides the foundation needed to build a complete process attachment tool for ROCprofiler-SDK. The actual rocprofv3 implementation uses similar techniques with additional optimizations and error handling.

CHAPTER

FOURTEEN

ROCPROFILER-SDK BUFFERED SERVICES

In the buffered approach, the internal (background) thread sends callbacks for batches of records. Supported buffer record categories are enumerated in `rocprofiler_buffer_category_t` category field and supported buffer tracing services are enumerated in `rocprofiler_buffer_tracing_kind_t`. Configuring a buffered tracing service requires buffer creation. Flushing the buffer implicitly or explicitly invokes a callback to the tool, which provides an array of one or more buffer records. To flush a buffer explicitly, use `rocprofiler_flush_buffer` function.

14.1 Subscribing to buffer tracing services

During tool initialization, the tool configures callback tracing using `rocprofiler_configure_buffer_tracing_service` function. However, before invoking `rocprofiler_configure_buffer_tracing_service`, the tool must create a buffer for the tracing records as shown in the following section.

14.2 Creating a buffer

```
rocprofiler_status_t
rocprofiler_create_buffer(rocprofiler_context_id_t        context,
                          size_t                          size,
                          size_t                          watermark,
                          rocprofiler_buffer_policy_t     policy,
                          rocprofiler_buffer_tracing_cb_t callback,
                          void*                           callback_data,
                          rocprofiler_buffer_id_t*        buffer_id);
```

Here are the parameters required to create a buffer:

- `size`: Size of the buffer in bytes, which is rounded up to the nearest memory page size (defined by `sysconf(_SC_PAGESIZE)`). The default memory page size on Linux is 4096 bytes (4 KB).
- `watermark`: Specifies the number of bytes at which the buffer should be flushed. To flush the buffer, the records in the buffer must invoke the `callback` parameter to deliver the records to the tool. For example, for a buffer of size 4096 bytes with the watermark set to 48 bytes, six 8-byte records can be placed in the buffer before `callback` is invoked. However, every 64-byte record that is placed in the buffer will trigger a flush. It is safe to set the `watermark` to any value between zero and the buffer size.
- `policy`: Specifies the behavior when a record is larger than the amount of free space in the current buffer. For example, for a buffer of size 4000 bytes with the watermark set to 4000 bytes and 3998 bytes populated with records, the `policy` dictates how to handle an incoming record greater than 2 bytes. If the environment variable `ROCPROFILER_BUFFER_POLICY_DISCARD` is enabled, all records greater than 2 bytes are dropped until the tool _explicitly_ flushes the buffer using `rocprofiler_flush_buffer` function call whereas, if the environment variable `ROCPROFILER_BUFFER_POLICY_LOSSLESS` is enabled, the current buffer is swapped out for an empty buffer and placed in the new buffer while the former (full) buffer is _implicitly_ flushed.

- `callback`: Invoked to flush the buffer.
- `callback_data`: Value passed as one of the arguments to the `callback` function.
- `buffer_id`: Output parameter for the function call to contain a non-zero handle field after successful buffer creation.

14.3 Creating a dedicated thread for buffer callbacks

By default, all buffers use the same (default) background thread created by ROCprofiler-SDK to invoke their callback. However, ROCprofiler-SDK provides an interface to allow the tools to create an additional background thread for one or more of their buffers.

To create callback threads for buffers, use `rocprofiler_create_callback_thread` function:

```
rocprofiler_status_t
rocprofiler_create_callback_thread(rocprofiler_callback_thread_t* cb_thread_id);
```

To assign buffers to that callback thread, use `rocprofiler_assign_callback_thread` function:

```
rocprofiler_status_t
rocprofiler_assign_callback_thread(rocprofiler_buffer_id_t       buffer_id,
                                   rocprofiler_callback_thread_t cb_thread_id);
```

Example:

```
{
    // create a context
    auto context_id = rocprofiler_context_id_t{0};
    rocprofiler_create_context(&context_id);

    // create a buffer associated with the context
    auto buffer_id  = rocprofiler_buffer_id_t{};
    rocprofiler_create_buffer(context_id, ..., &buffer_id);

    // specify that a new callback thread should be created and provide
    // and assign the identifier for it to the "thr_id" variable
    auto thr_id = rocprofiler_callback_thread_t{};
    rocprofiler_create_callback_thread(&thr_id);

    // assign the buffer callback to be delivered on this thread
    rocprofiler_assign_callback_thread(buffer_id, thr_id);
}
```

14.4 Configuring buffer tracing services

To configure buffer tracing services, use:

```
rocprofiler_status_t
rocprofiler_configure_buffer_tracing_service(rocprofiler_context_id_t          context_
↪id,
                                             rocprofiler_buffer_tracing_kind_t kind,
                                             rocprofiler_tracing_operation_t* ␣
↪operations,
```

(continues on next page)

(continued from previous page)

```
                                                size_t                          ␣
↪operations_count,
                                                rocprofiler_buffer_id_t          buffer_
↪id);
```

Here are the parameters required to configure buffer tracing services:

- `kind`: A high-level specification of the services to be traced. This parameter is also known as "domain". Domain examples include, but not limited to, the HIP API, HSA API, and kernel dispatches.
- `operations`: For each domain, there are often various `operations` that can be used to restrict the callbacks to a subset within the domain. For domains corresponding to APIs, the `operations` are the functions composing the API. To trace all operations in a domain, set the `operations` and `operations_count` parameters to `nullptr` and `0` respectively. To restrict the tracing domain to a subset of operations, the tool library must specify a C-array of type `rocprofiler_tracing_operation_t` for `operations` and size of the array for the `operations_count` parameter.

Similar to the `rocprofiler_configure_callback_tracing_service`, `rocprofiler_configure_buffer_tracing_service` returns an error if a buffer service for the specified context and domain is configured more than once.

Example:

```
{
    auto ctx = rocprofiler_context_id_t{};
    // ... creation of context, etc. ...

    // buffer parameters
    constexpr auto KB          = 1024;  // 1024 bytes
    constexpr auto buffer_size = 16 * KB;
    constexpr auto watermark   = 15 * KB;
    constexpr auto policy      = ROCPROFILER_BUFFER_POLICY_LOSSLESS;

    // buffer handle
    auto buffer_id = rocprofiler_buffer_id_t{};

    // create a buffer associated with the context
    rocprofiler_create_buffer(
        context_id, buffer_size, watermark, policy, callback_func, nullptr, &buffer_id);

    // configure HIP runtime API function records to be placed in buffer
    rocprofiler_configure_buffer_tracing_service(
        ctx, ROCPROFILER_BUFFER_TRACING_HIP_RUNTIME_API, nullptr, 0, buffer_id);

    // configure kernel dispatch records to be placed in buffer
    // (more than one service can use the same buffer)
    rocprofiler_configure_buffer_tracing_service(
        ctx, ROCPROFILER_BUFFER_TRACING_KERNEL_DISPATCH, nullptr, 0, buffer_id);

    // ... etc. ...
}
```

14.5 Buffer tracing callback function

Here is the buffer tracing callback function:

```
typedef void (*rocprofiler_buffer_tracing_cb_t)(rocprofiler_context_id_t      context,
                                                rocprofiler_buffer_id_t       buffer_id,
                                                rocprofiler_record_header_t** headers,
                                                size_t                        num_
↪headers,
                                                void*                         data,
                                                uint64_t                      drop_
↪count);
```

The `rocprofiler_record_header_t` data type contains the following information:

- `category` (`rocprofiler_buffer_category_t`): The `category` is used to classify the buffer record. For all services configured via `rocprofiler_configure_buffer_tracing_service`, the `category` is equal to the value of `ROCPROFILER_BUFFER_CATEGORY_TRACING`. The other available categories are `ROCPROFILER_BUFFER_CATEGORY_PC_SAMPLING` and `ROCPROFILER_BUFFER_CATEGORY_COUNTERS`.
- `kind`: The `kind` field is dependent on the `category`. For example, for `category` `ROCPROFILER_BUFFER_CATEGORY_TRACING`, the value of `kind` depicts the tracing type such as HSA core API in `ROCPROFILER_BUFFER_TRACING_HSA_CORE_API`.
- `payload`: The `payload` is casted after the category and kind have been determined.

```
{
    if(header->category == ROCPROFILER_BUFFER_CATEGORY_TRACING &&
        header->kind == ROCPROFILER_BUFFER_TRACING_HIP_RUNTIME_API)
    {
        auto* record =
            static_cast<rocprofiler_buffer_tracing_hip_api_record_t*>(header->payload);

        // ... etc. ...
    }
}
```

Example:

```
void
buffer_callback_func(rocprofiler_context_id_t      context,
                     rocprofiler_buffer_id_t       buffer_id,
                     rocprofiler_record_header_t** headers,
                     size_t                        num_headers,
                     void*                         user_data,
                     uint64_t                      drop_count)
{
    for(size_t i = 0; i < num_headers; ++i)
    {
        auto* header = headers[i];

        if(header->category == ROCPROFILER_BUFFER_CATEGORY_TRACING &&
           header->kind == ROCPROFILER_BUFFER_TRACING_HIP_RUNTIME_API)
        {
            auto* record =
```

(continues on next page)

(continued from previous page)

```
                static_cast<rocprofiler_buffer_tracing_hip_api_record_t*>(header->
↪payload);

            // ... etc. ...
        }
        else if(header->category == ROCPROFILER_BUFFER_CATEGORY_TRACING &&
                header->kind == ROCPROFILER_BUFFER_TRACING_KERNEL_DISPATCH)
        {
            auto* record =
                static_cast<rocprofiler_buffer_tracing_kernel_dispatch_record_t*>(header-
↪>payload);

            // ... etc. ...
        }
        else
        {
            throw std::runtime_error{"unhandled record header category + kind"};
        }
    }
}
```

14.6 Buffer tracing record

Unlike callback tracing records, there is no common set of data for each buffer tracing record. However, many buffer tracing records contain a `kind` and an `operation` field. You can obtain the value for the `kind` of tracing using `rocprofiler_query_buffer_tracing_kind_name` function and the value for the `operation` specific to a tracing kind using the `rocprofiler_query_buffer_tracing_kind_operation_name` function. You can also iterate over all the buffer tracing `kinds` and `operations` for each tracing kind using the `rocprofiler_iterate_buffer_tracing_kinds` and `rocprofiler_iterate_buffer_tracing_kind_operations` functions.

The buffer tracing record data types are available in the `rocprofiler-sdk/buffer_tracing.h` header.

CHAPTER

FIFTEEN

ROCPROFILER-SDK CALLBACK TRACING SERVICES

Callback tracing services provide immediate callbacks to a tool on the current CPU thread on the occurrence of an event. For example, when tracing an API function such as `hipSetDevice`, callback tracing invokes a user-specified callback before and after the traced function executes on the thread invoking the API function.

15.1 Subscribing to callback tracing services

During tool initialization, tools configure callback tracing using:

```
rocprofiler_status_t
rocprofiler_configure_callback_tracing_service(rocprofiler_context_id_t           ␣
↪context_id,
                                               rocprofiler_callback_tracing_kind_t kind,
                                               rocprofiler_tracing_operation_t*   ␣
↪operations,
                                               size_t                             ␣
↪operations_count,
                                               rocprofiler_callback_tracing_cb_t  ␣
↪callback,
                                               void*                              ␣
↪callback_args);
```

Here are the parameters required to configure callback tracing services:

- `kind`: A high-level specification of the services to be traced. This parameter is also known as "domain". Domain examples include, but not limited to, the HIP API, HSA API, and kernel dispatches.
- `operations`: For each domain, there are often various `operations` that can be used to restrict the callbacks to a subset within the domain. For domains corresponding to APIs, the `operations` are the functions composing the API. To trace all operations in a domain, set the `operations` and `operations_count` parameters to `nullptr` and `0` respectively. To restrict the tracing domain to a subset of operations, the tool library must specify a C-array of type `rocprofiler_tracing_operation_t` for `operations` and size of the array for the `operations_count` parameter.

`rocprofiler_configure_callback_tracing_service` returns an error if a callback service for the specified context and domain is configured more than once.

Example: To trace only two functions within the HIP runtime API, `hipGetDevice` and `hipSetDevice`:

```
{
    auto ctx = rocprofiler_context_id_t{};
    // ... creation of context, etc. ...
```

(continues on next page)

(continued from previous page)

```
    // array of operations (i.e. API functions)
    auto operations = std::array<rocprofiler_tracing_operation_t, 2>{
        ROCPROFILER_HIP_RUNTIME_API_ID_hipSetDevice,
        ROCPROFILER_HIP_RUNTIME_API_ID_hipGetDevice
    };

    rocprofiler_configure_callback_tracing_service(ctx,
                                                   ROCPROFILER_CALLBACK_TRACING_HIP_
↪RUNTIME_API,
                                                   operations.data(),
                                                   operations.size(),
                                                   callback_func,
                                                   nullptr);
    // ... etc. ...
}
```

The following code returns error ROCPROFILER_STATUS_ERROR_SERVICE_ALREADY_CONFIGURED as the callback service is already configured:

```
{
    auto ctx = rocprofiler_context_id_t{};
    // ... creation of context, etc. ...

    // array of operations (i.e. API functions)
    auto operations = std::array<rocprofiler_tracing_operation_t, 2>{
        ROCPROFILER_HIP_RUNTIME_API_ID_hipSetDevice,
        ROCPROFILER_HIP_RUNTIME_API_ID_hipGetDevice
    };

    for(auto op : operations)
    {
        // after the first iteration, returns ROCPROFILER_STATUS_ERROR_SERVICE_ALREADY_
↪CONFIGURED
        rocprofiler_configure_callback_tracing_service(ctx,
                                                       ROCPROFILER_CALLBACK_TRACING_HIP_
↪RUNTIME_API,
                                                       &op,
                                                       1,
                                                       callback_func,
                                                       nullptr);
    }

    // ... etc. ...
}
```

15.2 Callback tracing callback function

Here is the callback tracing callback function:

```
typedef void (*rocprofiler_callback_tracing_cb_t)(rocprofiler_callback_tracing_record_t␣
↪record,
```

(continues on next page)

(continued from previous page)

```
                                                rocprofiler_user_data_t*                ␣
↪user_data,
                                                void* callback_data)
```

The parameters `record` and `user_data` are discussed here:

- `record`: Contains the information to uniquely identify a tracing record type. Here is the definition:

```
typedef struct rocprofiler_callback_tracing_record_t
{
  rocprofiler_context_id_t            context_id;
  rocprofiler_thread_id_t             thread_id;
  rocprofiler_correlation_id_t        correlation_id;
  rocprofiler_callback_tracing_kind_t kind;
  uint32_t                            operation;
  rocprofiler_callback_phase_t        phase;
  void*                               payload;
} rocprofiler_callback_tracing_record_t;
```

  The underlying type of `payload` field is typically unique to a domain and, less frequently, an operation. For example, for the `ROCPROFILER_CALLBACK_TRACING_HIP_RUNTIME_API` and `ROCPROFILER_CALLBACK_TRACING_HIP_COMPILER_API`, the payload must be casted to `rocprofiler_callback_tracing_hip_api_data_t*`, which contains the arguments to the function and the return value when exiting the function. The payload field is a valid pointer only during the invocation of the callback function(s).

- `user_data`: Stores data in between callback phases. This value is unique for every instance of an operation. For example, for a tool library to store the timestamp of the `ROCPROFILER_CALLBACK_PHASE_ENTER` phase for the ensuing `ROCPROFILER_CALLBACK_PHASE_EXIT` callback, the data can be stored using:

```
void
callback_func(rocprofiler_callback_tracing_record_t record,
              rocprofiler_user_data_t*               user_data,
              void*                                  cb_data)
{
    auto ts = rocprofiler_timestamp_t{};
    rocprofiler_get_timestamp(&ts);

    if(record.phase == ROCPROFILER_CALLBACK_PHASE_ENTER)
    {
        user_data->value = ts;
    }
    else if(record.phase == ROCPROFILER_CALLBACK_PHASE_EXIT)
    {
        auto delta_ts = (ts - user_data->value);
        // ... etc. ...
    }
    else
    {
        // ... etc. ...
    }
}
```

  The *callback_data* is passed to *rocprofiler_configure_callback_tracing_service* as the value of *callback_args* to

subscribe to callback tracing services.

15.3 Callback tracing record

To obtain the name of the `kind` of tracing, you can use `rocprofiler_query_callback_tracing_kind_name` function and to obtain the name of an `operation` specific to a tracing kind, use `rocprofiler_query_callback_tracing_kind_operation_name` function. To iterate over all the callback tracing kinds and operations for each tracing kind, use `rocprofiler_iterate_callback_tracing_kinds` and `rocprofiler_iterate_callback_tracing_kind_operations` functions.

Lastly, for a specified `rocprofiler_callback_tracing_record_t` object, ROCprofiler-SDK supports generically iterating over the arguments of the payload field for many domains. Within the `rocprofiler_callback_tracing_record_t` object, the domain-specific information is available in an opaque `void* payload`. The data types generally follow the naming convention of `rocprofiler_callback_tracing_<DOMAIN>_data_t`. For example, for the tracing kinds `ROCPROFILER_BUFFER_TRACING_HSA_{CORE,AMD_EXT,IMAGE_EXT,FINALIZE_EXT}_API`, cast the payload to `rocprofiler_callback_tracing_hsa_api_data_t*`:

```
void
callback_func(rocprofiler_callback_tracing_record_t record,
              rocprofiler_user_data_t*              user_data,
              void*                                 cb_data)
{
    static auto hsa_domains = std::unordered_set<rocprofiler_buffer_tracing_kind_t>{
        ROCPROFILER_BUFFER_TRACING_HSA_CORE_API,
        ROCPROFILER_BUFFER_TRACING_HSA_AMD_EXT_API,
        ROCPROFILER_BUFFER_TRACING_HSA_IMAGE_EXT_API,
        ROCPROFILER_BUFFER_TRACING_HSA_FINALIZER_API};

    if(hsa_domains.count(record.kind) > 0)
    {
        auto* payload = static_cast<rocprofiler_callback_tracing_hsa_api_data_t*>(record.
↪payload);

        hsa_status_t status = payload->retval.hsa_status_t_retval;
        if(record.phase == ROCPROFILER_CALLBACK_PHASE_EXIT && status != HSA_STATUS_
↪SUCCESS)
        {
            const char* _kind = nullptr;
            const char* _operation = nullptr;

            rocprofiler_query_callback_tracing_kind_name(record.kind, &_kind, nullptr);
            rocprofiler_query_callback_tracing_kind_operation_name(
                record.kind, record.operation, &_operation, nullptr);

            // message that
            fprintf(stderr, "[domain=%s] %s returned a non-zero exit code: %i\n", _kind,␣
↪_operation, status);
        }
    }
    else if(record.phase == ROCPROFILER_CALLBACK_PHASE_EXIT)
    {
        auto delta_ts = (ts - user_data->value);
```

(continues on next page)

(continued from previous page)

```
        // ... etc. ...
    }
    else
    {
        // ... etc. ...
    }
}
```

Example: Iterating over all the callback tracing kinds and operations for each tracing kind using `rocprofiler_iterate_callback_tracing_kind_operation_args`:

```
int
print_args(rocprofiler_callback_tracing_kind_t domain_idx,
           uint32_t                            op_idx,
           uint32_t                            arg_num,
           const void* const                   arg_value_addr,
           int32_t                             arg_indirection_count,
           const char*                         arg_type,
           const char*                         arg_name,
           const char*                         arg_value_str,
           int32_t                             arg_dereference_count,
           void*                               data)
{
    if(arg_num == 0)
    {
        const char* _kind      = nullptr;
        const char* _operation = nullptr;

        rocprofiler_query_callback_tracing_kind_name(domain_idx, &_kind, nullptr);
        rocprofiler_query_callback_tracing_kind_operation_name(
            domain_idx, op_idx, &_operation, nullptr);

        fprintf(stderr, "\n[%s] %s\n", _kind, _operation);
    }

    char* _arg_type = abi::__cxa_demangle(arg_type, nullptr, nullptr, nullptr);

    fprintf(stderr, "    %u: %-18s %-16s = %s\n", arg_num, _arg_type, arg_name, arg_
↪value_str);

    free(_arg_type);

    // unused in example
    (void) arg_value_addr;
    (void) arg_indirection_count;
    (void) arg_dereference_count;
    (void) data;

    return 0;
}

void
```

(continues on next page)

(continued from previous page)

```
callback_func(rocprofiler_callback_tracing_record_t record,
              rocprofiler_user_data_t*              user_data,
              void*                                 cb_data)
{
    if(record.phase == ROCPROFILER_CALLBACK_PHASE_EXIT &&
       record.kind == ROCPROFILER_CALLBACK_TRACING_HIP_RUNTIME_API &&
       (record.operation == ROCPROFILER_HIP_RUNTIME_API_ID_hipLaunchKernel ||
        record.operation == ROCPROFILER_HIP_RUNTIME_API_ID_hipMemcpyAsync))
    {
        rocprofiler_iterate_callback_tracing_kind_operation_args(
                            record, print_args, record.phase, nullptr));
    }
}
```

Sample output:

```
[HIP_RUNTIME_API] hipLaunchKernel
    0: void const*        function_address = 0x219308
    1: rocprofiler_dim3_t numBlocks        = {z=1, y=310, x=310}
    2: rocprofiler_dim3_t dimBlocks        = {z=1, y=32, x=32}
    3: void**             args             = 0x7ffe6d8dd3c0
    4: unsigned long      sharedMemBytes   = 0
    5: hipStream_t*      stream           = 0x17b40c0

[HIP_RUNTIME_API] hipMemcpyAsync
    0: void*              dst              = 0x7f06c7bbb010
    1: void const*        src              = 0x7f0698800000
    2: unsigned long      sizeBytes        = 393625600
    3: hipMemcpyKind      kind             = DeviceToHost
    4: hipStream_t*      stream           = 0x25dfcf0
```

15.4 Code object tracing

The code object tracing service is a critical component for obtaining information regarding asynchronous activity on the GPU. The `rocprofiler_callback_tracing_code_object_load_data_t` payload (kind=``ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT``, operation=``ROCPROFILER_CODE_OBJECT_LOAD``) provides a unique identifier for a bundle of one or more GPU kernel symbols that are loaded for a specific GPU agent. For example, if your application leverages a multi-GPU system consisting of four Vega20 GPUs and four MI100 GPUs, at least eight code objects will be loaded: one code object for each GPU. Each code object will be associated with a set of kernel symbols. The `rocprofiler_callback_tracing_code_object_kernel_symbol_register_data_t` payload (kind=``ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT``, operation=``ROCPROFILER_CODE_OBJECT_DEVICE_KERNEL_SYMBOL_REGISTER``) provides a globally unique identifier for the specific kernel symbol along with the kernel name and several other static properties of the kernel such as scratch size, scalar general purpose register count, and so on.

> **Note**
>
> The kernel identifiers for two identical kernel symbols with the same properties (kernel name, scratch size, and so on) that are part of similar code objects loaded for different GPU agents will still be unique. Furthermore, the identifier for a code object and its kernel symbols after being unloaded and then reloaded, will also be unique.

Here is the general sequence of events when a code object is loaded and unloaded:

1. Callback: load code object - kind= `ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT` - operation= `ROCPROFILER_CODE_OBJECT_LOAD` - phase= `ROCPROFILER_CALLBACK_PHASE_LOAD`
2. Callback: load kernel symbol - kind= `ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT` - operation= `ROCPROFILER_CODE_OBJECT_DEVICE_KERNEL_SYMBOL_REGISTER` - phase= `ROCPROFILER_CALLBACK_PHASE_LOAD` - Repeats for each kernel symbol in code object
3. Execute application
4. Callback: unload kernel symbol - kind= `ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT` - operation= `ROCPROFILER_CODE_OBJECT_DEVICE_KERNEL_SYMBOL_REGISTER` - phase= `ROCPROFILER_CALLBACK_PHASE_UNLOAD` - Repeats for each kernel symbol in code object
5. Callback: unload code object - kind= `ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT` - operation= `ROCPROFILER_CODE_OBJECT_LOAD` - phase= `ROCPROFILER_CALLBACK_PHASE_UNLOAD`

Note

ROCprofiler-SDK doesn't provide an interface to query information outside of the code object tracing service. If you wish to associate kernel names with kernel tracing records, the tool must be configured to create a copy of the relevant information when the code objects and kernel symbol are loaded. However, any constant string fields like `const char* kernel_name` don't need to be copied as these are guaranteed to be valid pointers until after ROCprofiler-SDK finalization. If a tool decides to delete its copy of the data associated with a code object or kernel symbol identifier when the code object and kernel symbols are unloaded, it is highly recommended to flush all buffers that might contain references to that code object or kernel symbol identifier before deleting the associated data.

For a sample of code object tracing, see samples/code_object_tracing.

CHAPTER

SIXTEEN

ROCPROFILER-SDK COUNTER COLLECTION SERVICES

There are two modes of counter collection service:

- **Dispatch counting**: In this mode, counters are collected on a per-kernel launch basis. This mode is useful for collecting highly detailed counters for a specific kernel execution in isolation. Note that dispatch counting allows only a single kernel to execute in hardware at a time.
- **Device counting**: In this mode, counters are collected on a device level. This mode is useful for collecting device level counters not tied to a specific kernel execution, which encompasses collecting counter values for a specific time range.

This topic explains how to setup dispatch and device counting and use common counter collection APIs. For details on the APIs including the less commonly used counter collection APIs, see the API library. For fully functional examples of both dispatch and device counting, see Samples.

16.1 Definitions

Profile Config: A configuration to specify the counters to be collected on an agent. This must be supplied to various counter collection APIs to initiate collection of counter data. Profiles are agent-specific and can't be used on different agents.

Counter ID: Unique Id (per-architecture) that specifies the counter. The counter Id can be used to fetch counter information such as its name or expression.

Instance ID: Unique record Id that encodes the counter Id and dimension for a collected value.

Dimension: Dimensions help to provide context to the raw counter values by specifying the hardware register that is the source of counter collection such as a shader engine. All counter values have dimension data encoded in their instance Id, which allows you to extract the values for individual dimensions using functions in the counter interface. The following dimensions are supported:

```
ROCPROFILER_DIMENSION_XCC,            ///< XCC dimension of result
ROCPROFILER_DIMENSION_AID,            ///< AID dimension of result
ROCPROFILER_DIMENSION_SHADER_ENGINE,  ///< SE dimension of result
ROCPROFILER_DIMENSION_AGENT,          ///< Agent dimension
ROCPROFILER_DIMENSION_SHADER_ARRAY,   ///< Number of shader arrays
ROCPROFILER_DIMENSION_WGP,            ///< Number of workgroup processors
ROCPROFILER_DIMENSION_INSTANCE,       ///< From unspecified hardware register
```

16.2 Using the Counter Collection Service

The setup for dispatch and device counting is similar with only minor changes needed to adapt code from one to another. Here are the steps required to configure the counter collection services:

16.2.1 tool_init() setup

Similar to tracing services, you must create a context and a buffer to collect the output when initializing the tool.

```
rocprofiler_context_id_t ctx{0};
rocprofiler_buffer_id_t buff;
ROCPROFILER_CALL(rocprofiler_create_context(&ctx), "context creation failed");
ROCPROFILER_CALL(rocprofiler_create_buffer(ctx,
                                           4096,
                                           2048,
                                           ROCPROFILER_BUFFER_POLICY_LOSSLESS,
                                           buffered_callback, // Callback to process␣
↪data
                                           user_data,
                                           &buff),
                 "buffer creation failed");
```

After creating a context and buffer to store results in `tool_init`, it is highly recommended but not mandatory for you to construct the profiles for each agent, containing the counters for collection. Profile creation should be avoided in the time critical dispatch counting callback as it involves validating if the counters can be collected on the agent. After profile setup, you can set up the collection service for dispatch or device counting. To set up either dispatch or device counting (only one can be used at a time), use:

```
/* For Dispatch Counting */
// Setup the dispatch profile counting service. This service will trigger the dispatch_
↪callback
// when a kernel dispatch is enqueued into the HSA queue. The callback will specify what
// counters to collect by returning a profile config id.
ROCPROFILER_CALL(rocprofiler_configure_buffer_dispatch_counting_service(
                     ctx, buff, dispatch_callback, nullptr),
                 "Could not setup buffered service");

/* For Agent Counting */
// set_profile is a callback that is use to select the profile to use when
// the context is started. It is called at every rocprofiler_ctx_start() call.
ROCPROFILER_CALL(rocprofiler_configure_device_counting_service(
                     ctx, buff, agent_id, set_profile, nullptr),
                 "Could not setup buffered service");
```

16.3 Profile Setup

1. The first step in constructing a counter collection profile is to find the GPU agents on the machine. You must create a profile for each set of counters to be collected on every agent on the machine. You can use `rocprofiler_query_available_agents` to find agents on the system. The following example collects all GPU agents on the device and stores them in the vector agents:

```
std::vector<rocprofiler_agent_v0_t> agents;
```

(continues on next page)

(continued from previous page)

```
// Callback used by rocprofiler_query_available_agents to return
// agents on the device. This can include CPU agents as well. We
// select GPU agents only (i.e. type == ROCPROFILER_AGENT_TYPE_GPU)
rocprofiler_query_available_agents_cb_t iterate_cb = [](rocprofiler_agent_version_t␣
↪agents_ver,
                                                        const void**               ␣
↪agents_arr,
                                                        size_t                      num_
↪agents,
                                                        void*                      ␣
↪udata) {
    if(agents_ver != ROCPROFILER_AGENT_INFO_VERSION_0)
        throw std::runtime_error{"unexpected rocprofiler agent version"};
    auto* agents_v = static_cast<std::vector<rocprofiler_agent_v0_t>*>(udata);
    for(size_t i = 0; i < num_agents; ++i)
    {
        const auto* agent = static_cast<const rocprofiler_agent_v0_t*>(agents_arr[i]);
        if(agent->type == ROCPROFILER_AGENT_TYPE_GPU) agents_v->emplace_back(*agent);
    }
    return ROCPROFILER_STATUS_SUCCESS;
};

// Query the agents, only a single callback is made that contains a vector
// of all agents.
ROCPROFILER_CALL(
    rocprofiler_query_available_agents(ROCPROFILER_AGENT_INFO_VERSION_0,
                                       iterate_cb,
                                       sizeof(rocprofiler_agent_t),
                                       const_cast<void*>(static_cast<const void*>(&
↪agents))),
    "query available agents");
```

2. To identify the counters supported by an agent, query the available counters with `rocprofiler_iterate_agent_supported_counters`. Here is an example of a single agent returning the available counters in `gpu_counters`:

```
std::vector<rocprofiler_counter_id_t> gpu_counters;

// Iterate all the counters on the agent and store them in gpu_counters.
ROCPROFILER_CALL(rocprofiler_iterate_agent_supported_counters(
                     agent,
                     [](rocprofiler_agent_id_t,
                        rocprofiler_counter_id_t* counters,
                        size_t                    num_counters,
                        void*                     user_data) {
                         std::vector<rocprofiler_counter_id_t>* vec =
                             static_cast<std::vector<rocprofiler_counter_id_t>*>(user_
↪data);
                         for(size_t i = 0; i < num_counters; i++)
                         {
                             vec->push_back(counters[i]);
                         }
```

(continues on next page)

(continued from previous page)

```
                        return ROCPROFILER_STATUS_SUCCESS;
                    },
                    static_cast<void*>(&gpu_counters)),
                "Could not fetch supported counters");
```

3. `rocprofiler_counter_id_t` is a handle to a counter. To fetch information about the counter such as its name, use `rocprofiler_query_counter_info`:

```
for(auto& counter : gpu_counters)
{
    // Contains name and other attributes about the counter.
    // See API documentation for more info on the contents of this struct.
    rocprofiler_counter_info_v0_t info;
    ROCPROFILER_CALL(
        rocprofiler_query_counter_info(
            counter, ROCPROFILER_COUNTER_INFO_VERSION_0, static_cast<void*>(&info)),
        "Could not query info for counter");
}
```

4. After identifying the counters to be collected, construct a profile by passing a list of these counters to `rocprofiler_create_counter_config`.

```
// Create and return the profile
rocprofiler_counter_config_id_t profile;
ROCPROFILER_CALL(rocprofiler_create_counter_config(
                    agent, counters_array, counters_array_count, &profile),
                "Could not construct profile cfg");
```

5. You can use the created profile for both dispatch and agent counter collection services.

Note

Points to note on profile behavior:

- Profile created is *only valid* for the agent it was created for.
- Profiles are immutable. To collect a new counter set, construct a new profile.
- A single profile can be used multiple times on the same agent.
- Counter Ids supplied to `rocprofiler_create_counter_config` are *agent-specific* and can't be used to construct profiles for other agents.

16.4 Dispatch Counting Callback

When a kernel is dispatched, a dispatch callback is issued to the tool to allow selection of counters to be collected for the dispatch by supplying a profile.

```
void
dispatch_callback(rocprofiler_dispatch_counting_service_data_t dispatch_data,
                  rocprofiler_counter_config_id_t*             config,
                  rocprofiler_user_data_t* user_data,
                  void* /*callback_data_args*/)
```

`dispatch_data` contains information about the dispatch being launched such as its name. `config` is used by the tool to specify the profile, which allows counter collection for the dispatch. If no profile is supplied, no counters are collected for this dispatch. `user_data` contains user data supplied to `rocprofiler_configure_buffered_dispatch_profile_counting_service`.

16.5 Agent Set Profile Callback

This callback is invoked after the context starts and allows the tool to specify the profile to be used.

```
void
set_profile(rocprofiler_context_id_t                context_id,
            rocprofiler_agent_id_t                  agent,
            rocprofiler_device_counting_agent_cb_t set_config,
            void*)
```

The profile to be used for this agent is specified by calling `set_config(agent, profile)`.

16.5.1 Buffered callback

Data from collected counter values is returned through a buffered callback. The buffered callback routines are similar for dispatch and device counting except that some data such as kernel launch Ids is not available in device counting mode. Here is a sample iteration to print out counter collection data:

```
for(size_t i = 0; i < num_headers; ++i)
{
    auto* header = headers[i];
    if(header->category == ROCPROFILER_BUFFER_CATEGORY_COUNTERS &&
       header->kind == ROCPROFILER_COUNTER_RECORD_PROFILE_COUNTING_DISPATCH_HEADER)
    {
        // Print the returned counter data.
        auto* record =
            static_cast<rocprofiler_dispatch_counting_service_record_t*>(header->
↪payload);
        ss << "[Dispatch_Id: " << record->dispatch_info.dispatch_id
           << " Kernel_ID: " << record->dispatch_info.kernel_id
           << " Corr_Id: " << record->correlation_id.internal << ")]\n";
    }
    else if(header->category == ROCPROFILER_BUFFER_CATEGORY_COUNTERS &&
            header->kind == ROCPROFILER_COUNTER_RECORD_VALUE)
    {
        // Print the returned counter data.
        auto* record = static_cast<rocprofiler_counter_record_t*>(header->payload);
        rocprofiler_counter_id_t counter_id = {.handle = 0};

        rocprofiler_query_record_counter_id(record->id, &counter_id);

        ss << "  (Dispatch_Id: " << record->dispatch_id << " Counter_Id: " << counter_id.
↪handle
           << " Record_Id: " << record->id << " Dimensions: [";

        for(auto& dim : counter_dimensions(counter_id))
        {
            size_t pos = 0;
```

(continues on next page)

(continued from previous page)

```
            rocprofiler_query_record_dimension_position(record->id, dim.id, &pos);
            ss << "{" << dim.name << ": " << pos << "},";
        }
        ss << "] Value [D]: " << record->counter_value << "),";
    }
}
```

16.6 Counter Definitions

Counters are defined in yaml format in the `counter_defs.yaml` file. The counter definition has the following format:

```
counter_name:       # Counter name
  architectures:
    gfx90a:         # Architecture name
      block:        # Block information (SQ/etc)
      event:        # Event ID (used by AQLProfile to identify counter register)
      expression:   # Formula for the counter (if derived counter)
      description:  # Per-arch description (optional)
    gfx1010:
      ...
  description:      # Description of the counter
```

You can separately define the counters for different architectures as shown in the preceding example for gfx90a and gfx1010. If two or more architectures share the same block, event, or expression definition, they can be specified together using "/" delimiter ("gfx90a/gfx1010:"). Hardware metrics have the elements block, event, and description defined. Derived metrics have the element expression defined and can't have block or event defined.

16.7 Derived Metrics

Derived metrics are expressions performing computation on collected hardware metrics. These expressions produce result similar to a real hardware counter.

```
GPU_UTIL:
  architectures:
    gfx942/gfx941/gfx10/gfx1010/gfx1030/gfx1031/gfx11/gfx1032/gfx1102/gfx906/gfx1100/
↪gfx1101/gfx940/gfx908/gfx90a/gfx9:
      expression: 100*GRBM_GUI_ACTIVE/GRBM_COUNT
  description: Percentage of the time that GUI is active
```

In the preceding example, `GPU_UTIL` is a derived metric that uses a mathematic expression to calculate the utilization rate of the GPU using values of two GRBM hardware counters `GRBM_GUI_ACTIVE` and `GRBM_COUNT`. Expressions support the standard set of math operators (/,*,-,+) along with a set of special functions such as reduce and accumulate.

16.7.1 Reduce Function

```
Expression: 100*reduce(GL2C_HIT,sum)/(reduce(GL2C_HIT,sum)+reduce(GL2C_MISS,sum))
```

The reduce function reduces counter values across all dimensions such as shader engine, SIMD, and so on, to produce a single output value. This helps to collect and compare values across the entire device. Here are the common reduction operations:

- `sum`: Sums to create a single output. For example, `reduce(GL2C_HIT,sum)` sums all `GL2C_HIT` hardware register values.
- `avr`: Calculates the average across all dimensions.
- `min`: Selects minimum value across all dimensions.
- `max`: Selects the maximum value across all dimensions.

```
expression: reduce(X,sum,[DIMENSION_XCC])
```

Reduce() also supports dimension wise reduction, when provided dimensions in 3rd parameter. In the expression above, if `X` has two dimensions `DIMENSION_XCC`, `DIMENSION_SHADER_ARRAY`, and `DIMENSION_WGP`, the reduce happens across counter values where `DIMENSION_SHADER_ARRAY` and `DIMENSION_WGP` dimensions are same as shown below.

Let's say DIM sizes of XCC, SHADER_ARRAY(SH), WGP be 2, 4, 4 respectively.

Raw Counter Data in 3D space:

XCC[0]:

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
| SH[0] |   1  |   2  |   3  |   4  |
| SH[1] |   5  |   6  |   7  |   8  |
| SH[2] |   9  |   10 |   11 |   12 |
| SH[3] |   13 |   14 |   15 |   16 |
```

XCC[1]:

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
| SH[0] |   1  |   2  |   3  |   4  |
| SH[1] |   5  |   6  |   7  |   8  |
| SH[2] |   9  |   10 |   11 |   12 |
| SH[3] |   13 |   14 |   15 |   16 |
```

Reducing XCC dim with sum, results to 2D space with only WGP and SH.

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
| SH[0] |  2   |   4  |   6  |   8  |
| SH[1] |  10  |   12 |   14 |   16 |
| SH[2] |  18  |   20 |   22 |   24 |
| SH[3] |  26  |   28 |   30 |   32 |
```

similarly, for `reduce(X,sum,[DIMENSION_XCC,DIMENSION_SHADER_ARRAY])` results in only WGP dimension.

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
|       |  56  |  64  |  72  |  80  |
```

16.7.2 Select Function

```
expression: select(Y, [DIMENSION_XCC=[0],DIMENSION_SHADER_ENGINE=[2]])
```

select() only returns counter values which match the dimension indexes provided by the user in expression. This operation is to allow a user to state they only want to select specific dimensions index. Supported dimensions include `DIMENSION_XCC`, `DIMENSION_AID`, `DIMENSION_SHADER_ENGINE`, `DIMENSION_AGENT`, `DIMENSION_SHADER_ARRAY`, `DIMENSION_WGP`, `DIMENSION_INSTANCE`. For example `select(Y, [DIMENSION_XCC=[0],DIMENSION_SHADER_ENGINE=[2]])` gives counter values which are from DIMENSION_XCC= 0 and DIMENSION_SHADER_ENGINE= 2 for Y Metric.

Let's say Y has XCC, SHADER_ENGINE (SE), WGP dimensions with sizes 2, 4, 4 respectively.

Raw Counter Data in 3D space:

XCC[0]:

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
| SE[0] |   1  |   2  |   3  |   4  |
| SE[1] |   5  |   6  |   7  |   8  |
| SE[2] |   9  |   10 |   11 |   12 |
| SE[3] |   13 |   14 |   15 |   16 |
```

XCC[1]:

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
| SE[0] |   17 |   18 |   19 |   20 |
| SE[1] |   21 |   22 |   23 |   24 |
| SE[2] |   25 |   26 |   27 |   28 |
| SE[3] |   29 |   30 |   31 |   32 |
```

Selecting at XCC=0 results to 2D space with WGP and SH dimensions, as shown below.

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
| SE[0] |   1  |   2  |   3  |   4  |
| SE[1] |   5  |   6  |   7  |   8  |
| SE[2] |   9  |   10 |   11 |   12 |
| SE[3] |   13 |   14 |   15 |   16 |
```

similarly, for `select(Y, [DIMENSION_XCC=[0],DIMENSION_SHADER_ENGINE=[2]])` results in only WGP dimension with XCC=0 and SE=2.

```
|       |WGP[0]|WGP[1]|WGP[2]|WGP[3]|
|-------|------|------|------|------|
|       |  9   |  10  |  11  |  12  |
```

16.8 Accumulate Function

```
Expression: accumulate(<basic_level_counter>, <resolution>)
```

- The accumulate function sums the values of a basic level counter over the specified number of cycles. The `resolution` parameter allows you to control the frequency of the following summing operation:
  - `HIGH_RES`: Sums up the basic level counter every clock cycle. Captures the value every cycle for higher accuracy, which helps in fine-grained analysis.

  - LOW_RES: Sums up the basic level counter every four clock cycles. Reduces the data points and provides less detailed summing, which helps in reducing data volume.
  - NONE: Does nothing and is equivalent to collecting basic level counter. Outputs the value of the basic level counter without performing any summing operation.

Example:

```
MeanOccupancyPerCU:
  architectures:
    gfx942/gfx941/gfx940:
      expression: accumulate(SQ_LEVEL_WAVES,HIGH_RES)/reduce(GRBM_GUI_ACTIVE,max)/CU_NUM
  description: Mean occupancy per compute unit.
```

<metric name="MeanOccupancyPerCU" expr=accumulate(SQ_LEVEL_WAVES,HIGH_RES)/reduce(GRBM_GUI_ACTIVE,max)/CU
descr="Mean occupancy per compute unit."></metric>

- MeanOccupancyPerCU: In the preceding example, the MeanOccupancyPerCU metric calculates the mean occupancy per compute unit. It uses the accumulate function with HIGH_RES to sum the SQ_LEVEL_WAVES counter every clock cycle. This sum is then divided by the maximum value of GRBM_GUI_ACTIVE and the number of compute units CU_NUM to derive the mean occupancy.

16.9 Kernel Serialization

Counter collection in *dispatch counting* mode requires serialized execution of kernels on a target device. Kernel serialization isolates kernel executions, which helps to collect performance counter data. However, for applications requiring two kernels to execute on the same device simultaneously (co-dependent kernels), kernel serialization leads to deadlock in dispatch counter collection mode. To avoid deadlock in such applications, opt for any of the following options:

- Avoid co-dependent kernels in application.
- Don't collect performance data for co-dependent kernels by using kernel filtration methods in the rocprofv3's input configuration PMC file.
- Use ROCprofiler-SDK's device-wide counter collection mode to collect performance data. You can use tools such as RDC and PAPI to collect information. Note that the device-wide counter collection captures data for all executions on the device and not specific to the kernels.

CHAPTER

SEVENTEEN

ROCPROFILER-SDK PC SAMPLING METHOD

Program Counter (PC) sampling is a profiling method that uses statistical approximation of the kernel execution by sampling GPU program counters. Furthermore, this method periodically chooses an active wave in a round robin manner and snapshots its PC. This process takes place on every compute unit simultaneously, making it device-wide PC sampling. The outcome is the histogram of samples, explaining how many times each kernel instruction was sampled.

Warning

Risk acknowledgment: The PC sampling feature is under development and might not be completely stable. Use this beta feature cautiously. It may affect your system's stability and performance. Proceed at your own risk.

By activating this feature through `ROCPROFILER_PC_SAMPLING_BETA_ENABLED` environment variable, you acknowledge and accept the following potential risks:

- Hardware freeze: This beta feature could cause your hardware to freeze unexpectedly.
- Need for cold restart: In the event of a hardware freeze, you might need to perform a cold restart (turning the hardware off and on) to restore normal operations.

17.1 ROCprofiler-SDK PC sampling service

This section describes how to use ROCProfiler-SDK PC sampling API to configure and use PC sampling service. For fully functional examples, see Samples.

17.1.1 tool_init() setup

Here are the steps to set up `tool_init()`:

```
rocprofiler_context_id_t ctx{0};
rocprofiler_buffer_id_t buff;
ROCPROFILER_CALL(rocprofiler_create_context(&ctx), "context creation failed");
ROCPROFILER_CALL(rocprofiler_create_buffer(ctx,
                                           8192,
                                           2048,
                                           ROCPROFILER_BUFFER_POLICY_LOSSLESS,
                                           pc_sampling_callback, // Callback to process␣
↪PC samples
                                           user_data,
                                           &buff),
                 "buffer creation failed");
```

For more details on buffer creation, see *ROCprofiler-SDK buffered services*.

The PC sampling service is tied to a GPU agent. To extract the list of available agents, use the `rocprofiler_query_available_agents` as shown in the following code snippet:

```
std::vector<rocprofiler_agent_v0_t> agents;

// Callback used by rocprofiler_query_available_agents to return
// agents on the device. This can include CPU agents as well.
// Select GPU agents only (type == ROCPROFILER_AGENT_TYPE_GPU)
rocprofiler_query_available_agents_cb_t iterate_cb = [](rocprofiler_agent_version_t␣
↪agents_ver,
                                                        const void**              ␣
↪agents_arr,
                                                        size_t                     num_
↪agents,
                                                        void*                     ␣
↪udata) {
    if(agents_ver != ROCPROFILER_AGENT_INFO_VERSION_0)
        throw std::runtime_error{"unexpected rocprofiler agent version"};
    auto* agents_v = static_cast<std::vector<rocprofiler_agent_v0_t>*>(udata);
    for(size_t i = 0; i < num_agents; ++i)
    {
        const auto* agent = static_cast<const rocprofiler_agent_v0_t*>(agents_arr[i]);
        if(agent->type == ROCPROFILER_AGENT_TYPE_GPU) agents_v->emplace_back(*agent);
    }
    return ROCPROFILER_STATUS_SUCCESS;
};

// Query the agents. Only a single callback is made that contains a vector
// of all agents.
ROCPROFILER_CALL(
    rocprofiler_query_available_agents(ROCPROFILER_AGENT_INFO_VERSION_0,
                                       iterate_cb,
                                       sizeof(rocprofiler_agent_t),
                                       const_cast<void*>(static_cast<const void*>(&
↪agents))),
    "query available agents");
```

Only newer GPU architectures (MI200 onwards) support this feature. To determine whether an agent with `agent_id` supports the PC sampling and the available configurations (`rocprofiler_pc_sampling_configuration_t`), use the *rocprofiler_query_pc_sampling_agent_configurations*.

```
std::vector<rocprofiler_pc_sampling_configuration_t> available_configurations;

auto cb = [](const rocprofiler_pc_sampling_configuration_t* configs,
             size_t                                         num_config,
             void*                                          user_data) {
    auto* avail_configs = static_cast<avail_configs_vec_t*>(user_data);
    for(size_t i = 0; i < num_config; i++)
    {
        avail_configs->emplace_back(configs[i]);
    }
    return ROCPROFILER_STATUS_SUCCESS;
```

(continues on next page)

(continued from previous page)

```
};

auto status = rocprofiler_query_pc_sampling_agent_configurations(
    agent_id, cb, &available_configurations);
```

Assuming the *available_configurations* contain a single element:

```
rocprofiler_pc_sampling_configuration_t {
    .method = ROCPROFILER_PC_SAMPLING_METHOD_HOST_TRAP,
    .unit = ROCPROFILER_PC_SAMPLING_UNIT_TIME,
    .min_interval = 1,
    .max_interval = 10000
};
```

Configure the PC sampling service on an agent with `agent_id` to generate samples every 1000 micro-seconds as shown here:

```
auto status = rocprofiler_configure_pc_sampling_service(ctx,
                                                        agent_id,
                                                        picked_cfg->method,
                                                        picked_cfg->unit,
                                                        1000,  // 1000 us
                                                        buffer_id,
                                                        0);
if (status == ROCPROFILER_STATUS_SUCCESS)
{
    // PC Sampling service has been configured successfully.
}
else
{
    // code for error handling
}
```

Note

Multiple processes can share the same GPU agent simultaneously, so the following A->B->A problem is possible on shared systems. For example, process A can query available configurations and opt to configure the service with configuration CA. However, if process B manages to finish configuring the service with configuration CB, then process A will fail. Thus, it is advisable for process A to repeat the querying process to observe configuration CB and reuse it for configuring the PC sampling service. For more details, refer to the Samples.

17.2 Processing PC samples

The PC sampling service asynchronously delivers samples via a dedicated callback (`pc_sampling_callback`). The following code snippet outlines the process of iterating over samples.

```
void
pc_sampling_callback(rocprofiler_context_id_t ctx,
                     rocprofiler_buffer_id_t buff,
                     rocprofiler_record_header_t** headers,
```

(continues on next page)

(continued from previous page)

```
                    size_t num_headers,
                    void* data,
                    uint64_t drop_count)
{
    for(size_t i = 0; i < num_headers; i++)
    {
        auto* cur_header = headers[i];

        if(cur_header->category == ROCPROFILER_BUFFER_CATEGORY_PC_SAMPLING)
        {
            if(cur_header->kind == ROCPROFILER_PC_SAMPLING_RECORD_HOST_TRAP_V0_SAMPLE)
            {
                auto* pc_sample = static_cast<rocprofiler_pc_sampling_record_host_trap_
↪v0_t*>(
                    cur_header->payload);

                // Processing a single sample...
            }
            else
            {
                // ...
            }
        }
    }
}
```

For more information on the data comprising a single sample, see pc_sampling.h.

Note

A user can synchronously flush buffers via `rocprofiler_buffer_flush` that triggers `pc_sampling_callback`.

CHAPTER

EIGHTEEN

ROCPROF TRACE DECODER AND THREAD TRACE APIS

Thread trace is a profiling method that provides fine-grained insight into GPU kernel execution by collecting detailed traces of shader instructions executed by the GPU. This feature captures GPU occupancy, instruction execution times, fast performance counters, and other detailed performance data. Thread trace utilizes GPU hardware instrumentation to record events as they happen, resulting in precise timing information about wave (threads) execution behavior.

ROCprofiler-SDK provides wrapper APIs for the ROCprof Trace Decoder, a library to decode thread trace data.

Note

Thread trace can generate large amounts of data, especially when profiling complex applications or longer execution runs. This might require handling potentially high volumes of trace data, so it's recommended to implement appropriate filtering strategies to focus on the specific parts of interest in your application.

Note

ROCprof Trace Decoder is a binary-only library and can be found here.

18.1 Thread trace service API

This section describes how to use the ROCprofiler-SDK thread trace API to configure and use the thread trace service. For fully functional examples, see Samples.

18.1.1 tool_init() setup

Here are the steps to set up `tool_init()` for thread trace:

1. Configure callback tracing for code objects to get disassembly information:

```
rocprofiler_context_id_t ctx{0};
ROCPROFILER_CALL(rocprofiler_create_context(&ctx), "context creation failed");

ROCPROFILER_CALL(
    rocprofiler_configure_callback_tracing_service(ctx,
                                                   ROCPROFILER_CALLBACK_TRACING_CODE_
↪OBJECT,
                                                   nullptr,
                                                   0,
                                                   tool_codeobj_tracing_callback,
```

(continues on next page)

(continued from previous page)

```
                                                    nullptr),
    "code object tracing service configure");
```

2. The thread trace service is tied to a GPU agent. To extract the list of available agents, use the `rocprofiler_query_available_agents` function:

```
std::vector<rocprofiler_agent_id_t> agents{};

ROCPROFILER_CALL(
    rocprofiler_query_available_agents(
        ROCPROFILER_AGENT_INFO_VERSION_0,
        [](rocprofiler_agent_version_t, const void** _agents, size_t _num_agents, void* _
↪data) {
            auto* agent_v = static_cast<std::vector<rocprofiler_agent_id_t>*>(_data);
            for(size_t i = 0; i < _num_agents; ++i)
            {
                auto* agent = static_cast<const rocprofiler_agent_v0_t*>(_agents[i]);
                if(agent->type == ROCPROFILER_AGENT_TYPE_GPU)
                    agent_v->emplace_back(agent->id);
            }
            return ROCPROFILER_STATUS_SUCCESS;
        },
        sizeof(rocprofiler_agent_v0_t),
        &agents),
    "Failed to iterate agents");
```

3. Optionally, specify the configuration parameters:

```
std::vector<rocprofiler_thread_trace_parameter_t> params{};

params.push_back({ROCPROFILER_THREAD_TRACE_PARAMETER_SHADER_ENGINE_MASK, 0xF});

params.push_back({ROCPROFILER_THREAD_TRACE_PARAMETER_TARGET_CU, 0});

params.push_back({ROCPROFILER_THREAD_TRACE_PARAMETER_SIMD_SELECT, 0xF});

params.push_back({ROCPROFILER_THREAD_TRACE_PARAMETER_BUFFER_SIZE, 1u<<30}); // 1 GB
```

The configuration parameters are described here:

- ROCPROFILER_THREAD_TRACE_PARAMETER_SHADER_ENGINE_MASK: Configures the Shader Engine (SE) mask, which determines the SEs to be traced. This is a bitmask where each bit corresponds to a SE. For MI3xx, each hex digit corresponds to an XCD. It's highly recommended to trace only one SE at a time to avoid data loss.
- ROCPROFILER_THREAD_TRACE_PARAMETER_TARGET_CU: Configures the target Compute Unit (CU) or WGP. Instruction tracing can only operate on a single CU or WGP at a time. The same target is used for all SEs in `ROCPROFILER_THREAD_TRACE_PARAMETER_SHADER_ENGINE_MASK`.
- ROCPROFILER_THREAD_TRACE_PARAMETER_SIMD_SELECT: Configures SIMD selection. For gfx9, this is a bitmask where each bit corresponds to a SIMD lane. For example, 0xF selects all SIMD lanes in the `target_cu`. For gfx10, gfx11, and gfx12, this selects a single SIMD ID to trace. Results are taken mod4 for compatibility with gfx9 so 0xF selects SIMD3 of the target WGP.
- ROCPROFILER_THREAD_TRACE_PARAMETER_BUFFER_SIZE: Configures the

buffer size. This buffer is shared among all SEs specified in ROCPROFILER_THREAD_TRACE_PARAMETER_SHADER_ENGINE_MASK. There is a minimal side effect to specifying a larger buffer size, except for increased VRAM usage.

The thread trace can be configured in two primary modes: device-wide or per-dispatch, as described in the following sections.

18.1.2 Device thread trace

To enable thread trace service asynchronously or independently of kernel dispatches on a device, use:

```
// For device thread trace, it's recommended to create a separate context just to enable␣
↪and disable the service independently.
for(auto agent_id : agents)
{
    ROCPROFILER_CALL(
        rocprofiler_configure_device_thread_trace_service(
            ctx,
            agent_id,
            params.data(),
            params.size(),
            shader_data_callback,
            nullptr),
        "thread trace service configure");
}
```

18.1.3 Dispatch thread trace

To enable selective thread trace based on specific kernel dispatches, use the dispatch-based configuration. By default, only the traced kernels are serialized in dispatch tracing. An optional parameter is provided to serialize all kernels, which ensures no parallel kernel execution during tracing.

```
// (Optional) To serialize ALL kernels, not just the traced ones
// This ensures no parallel kernel execution during tracing
params.push_back({ROCPROFILER_THREAD_TRACE_PARAMETER_SERIALIZE_ALL, 1});

// Define dispatch callback to control thread trace
rocprofiler_thread_trace_control_flags_t
dispatch_callback(rocprofiler_agent_id_t agent_id,
                  rocprofiler_queue_id_t queue_id,
                  rocprofiler_async_correlation_id_t correlation_id,
                  rocprofiler_kernel_id_t kernel_id,
                  rocprofiler_dispatch_id_t dispatch_id,
                  void* userdata,
                  rocprofiler_user_data_t* dispatch_userdata)
{
    // Trace only the desired kernels
    if(target_kernel_id == kernel_id)
        return ROCPROFILER_THREAD_TRACE_CONTROL_START_AND_STOP;

    return ROCPROFILER_THREAD_TRACE_CONTROL_NONE;
}

// Configure dispatch-based thread trace
```

(continues on next page)

(continued from previous page)

```
for(auto agent_id : agents)
{
    ROCPROFILER_CALL(
        rocprofiler_configure_dispatch_thread_trace_service(
            ctx,
            agent_id,
            params.data(),
            params.size(),
            dispatch_callback,
            shader_data_callback,
            nullptr),
        "thread trace service configure");
}
```

For device-wide thread trace, starting the context automatically begins data capture. Some application warmup is recommended before starting the device thread trace. For the dispatch thread trace, this step is not necessary as tracing doesn't start automatically.

To start the context after all services are configured, use:

```
auto status = rocprofiler_start_context(ctx);

// Run your application workload here.
```

To stop the context to end data collection for device-wide thread trace, use:

```
status = rocprofiler_stop_context(ctx);
```

18.2 ROCprof Trace Decoder API

The thread trace functionality requires you to install the ROCprof Trace Decoder package separately. This package provides the necessary decoder library for processing thread trace data. Ensure to install this package on your system before using the thread trace feature.

18.2.1 Trace Decoder setup

To decode the raw thread trace data, create and initialize a Trace Decoder:

```
rocprofiler_thread_trace_decoder_handle_t decoder{};

// Create the Trace Decoder with the path to the decoder library
ROCPROFILER_CALL(
    rocprofiler_thread_trace_decoder_create(&decoder, "/opt/rocm/lib"),
    "thread trace decoder creation");

// Adds code object load information, reported by the code object tracing service
ROCPROFILER_CALL(rocprofiler_thread_trace_decoder_codeobj_load(decoder,
                                                               code_object_id,
                                                               load_delta,
                                                               load_size,
                                                               data,
```

(continues on next page)

(continued from previous page)

```
                                                                datasize),
                "code object load");
```

18.2.2 Code object tracking

To properly decode instruction addresses, track the code object information:

```
void
tool_codeobj_tracing_callback(rocprofiler_callback_tracing_record_t record,
                              rocprofiler_user_data_t* /* user_data */,
                              void* /* userdata */)
{
    if(record.kind != ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT ||
       record.operation != ROCPROFILER_CODE_OBJECT_LOAD)
        return;

    // Optionally, ROCPROFILER_CALLBACK_PHASE_UNLOAD can be handled by calling
    // rocprofiler_thread_trace_decoder_codeobj_unload(decoder, data->code_object_id);
    if(record.phase != ROCPROFILER_CALLBACK_PHASE_LOAD) return;

    auto* data = static_cast<rocprofiler_callback_tracing_code_object_load_data_t*>
↪(record.payload);
    // TODO: Handle file storage types
    if(data->storage_type == ROCPROFILER_CODE_OBJECT_STORAGE_TYPE_FILE) return;

    auto* memorybase = reinterpret_cast<const void*>(data->memory_base);

    // Register code object with Trace Decoder
    ROCPROFILER_CALL(
        rocprofiler_thread_trace_decoder_codeobj_load(
            decoder,
            data->code_object_id,
            data->load_delta,
            data->load_size,
            memorybase,
            data->memory_size),
        "code object loading to decoder");
}
```

18.3 Processing thread trace data

> **Note**
>
> In the provided samples, thread trace data is processed immediately within the shader data callbacks for simplicity. In practice, it's recommended to save the data to a file or buffer and process it after the application completes. The rate at which thread trace generates data tends to be higher (GB/s) than the rate at which it can be processed (MB/s). Deferred processing is strongly recommended to avoid performance bottlenecks.

The thread trace service asynchronously delivers raw trace data via a dedicated callback `shader_data_callback`. This data must be processed using the Trace Decoder to generate useful information:

```
void
shader_data_callback(rocprofiler_agent_id_t agent,
                     int64_t shader_engine_id,
                     void* data,
                     size_t data_size,
                     rocprofiler_user_data_t userdata)
{
    // Process shader callback data using the Trace Decoder.
    auto status = rocprofiler_trace_decode(decoder_handle,
                                           trace_decoder_callback,
                                           data,
                                           data_size,
                                           userdata);
}
```

18.3.1 Decoder callback

The trace decoder provides decoded information through a callback:

```
// Callback for decoded thread trace data
void
trace_decoder_callback(rocprofiler_thread_trace_decoder_record_type_t record_type,
                       void* trace_events,
                       uint64_t trace_size,
                       void* userdata)
{
    switch(record_type)
    {
        case ROCPROFILER_THREAD_TRACE_DECODER_RECORD_WAVE:
        {
            // Process wave information
            auto* waves = static_cast<rocprofiler_thread_trace_decoder_wave_t*>(trace_
↪events);
            for(uint64_t i = 0; i < trace_size; ++i)
            {
                // Process wave data (timeline, instruction execution, etc.)
            }
            break;
        }

        // Handle other record types as needed
    }
}
```

18.3.2 Trace Decoder info events

The Trace Decoder provides important information about the quality and comprehensiveness of the trace data through `ROCPROFILER_THREAD_TRACE_DECODER_RECORD_INFO` events. It is important to handle these events to understand potential issues with your trace data:

- ROCPROFILER_THREAD_TRACE_DECODER_INFO_DATA_LOST

  This event indicates that part of the trace data was dropped either due to hardware bandwidth limitations or buffer overflows. Receiving this event implies that portions of your trace might be missing or unreliable, which can

affect the accuracy of any analysis based on the trace data.

Possible causes:

- The trace buffer size was too small for the workload
- Memory bandwidth was exceeded

Recommended actions:

- Increase buffer sizes if possible
- Reduce the number of SEs or SIMD lanes being traced
- Disable `ROCPROFILER_THREAD_TRACE_PARAMETER_PERFCOUNTER` or increase `ROCPROFILER_THREAD_TRACE_PARAMETER_PERFCOUNTERS_CTRL` if enabled

- ROCPROFILER_THREAD_TRACE_DECODER_INFO_STITCH_INCOMPLETE

  This event indicates that the Trace Decoder was unable to find the PC (Program Counter) address for one or more traced instructions. Affected instructions will have their "pc" field set to zero.

  Possible causes:

  - The trace was started in the middle of a kernel execution:
    - If the trace was started after the kernel execution began, the Trace Decoder might not have received the necessary context to find the PC for all instructions.
    - Subsequent dispatches function normally.
  - Missing code object registration
  - Runtime kernels present in the trace: These are not always reported in the code object tracing callbacks
  - The `ROCPROFILER_THREAD_TRACE_DECODER_INFO_DATA_LOST` event was triggered. If parts of the trace were missing, important information might not have been available to the decoder.
  - There is a possible bug in the Trace Decoder. If you suspect this, report it to the ROCprofiler team.

For more information about the data structures and functions available for thread trace decoding, see the following headers:

- trace_decoder.h
- trace_decoder_types.h
- core.h
- dispatch.h
- agent.h

CHAPTER

NINETEEN

ROCPROFILER-SDK API LIBRARY

This ROCprofiler-SDK API topic covers:

- *Modules*
- *Global Data structures, topics, files*

19.1 Modules

The ROCprofiler-SDK API is organized into the following modules based on functionality:

- *Agent Information*
- *Buffer handling*
- *Buffer tracing*
- *Callback tracing*
- *Context management*
- *Counters*
- *Counter config*
- *Device counting service*
- *Dispatch counting service*
- *External correlation*
- *Intercept table*
- *Internal threading management*
- *OMPT Registration*
- *PC Sampling service*
- *Thread trace*
- *Tool registration*

19.1.1 Agent Information

enum **rocprofiler_agent_version_t**

Enumeration ID for version of the rocprofiler_agent_v*_t struct in rocprofiler_i.

Values:

enumerator **ROCPROFILER_AGENT_INFO_VERSION_NONE**

enumerator **ROCPROFILER_AGENT_INFO_VERSION_0**

enumerator **ROCPROFILER_AGENT_INFO_VERSION_LAST**

typedef *rocprofiler_agent_v0_t* **rocprofiler_agent_t**

Typedef for the current *rocprofiler_agent_version_t*.

typedef *rocprofiler_status_t* (***rocprofiler_query_available_agents_cb_t**)(*rocprofiler_agent_version_t* version, const void **agents, unsigned long num_agents, void *user_data)

Callback function type for querying the available agents.

If callback is invoked, returns the *rocprofiler_status_t* value returned from callback

Param version
: **[in]** Enum specifying the version of agent info

Param agents
: **[in]** Array of pointers to agents

Param num_agents
: **[in]** Number of agents in array

Param user_data
: **[in]** Data pointer passback

Retval ROCPROFILER_STATUS_ERROR_INCOMPATIBLE_ABI
: size of the agent struct in application is larger than the agent struct for rocprofiler-sdk

Retval ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT
: Invalid *rocprofiler_agent_version_t* value

Return
: *rocprofiler_status_t*

rocprofiler_status_t **rocprofiler_query_available_agents**(*rocprofiler_agent_version_t* version, *rocprofiler_query_available_agents_cb_t* callback, unsigned long agent_size, void *user_data)

Receive synchronous callback with an array of available agents at moment of invocation.

Parameters
- **version** – **[in]** Enum value specifying the struct type of the agent info
- **callback** – **[in]** Callback function accepting list of agents
- **agent_size** – **[in]** Should be set to sizeof(rocprofiler_agent_t)
- **user_data** – **[in]** Data pointer provided to callback

Returns
: *rocprofiler_status_t*

struct **rocprofiler_agent_cache_t**

#include <rocprofiler-sdk/agent.h> Cache information for an agent.

struct **rocprofiler_agent_io_link_t**

#include <rocprofiler-sdk/agent.h> IO link information for an agent.

struct **rocprofiler_agent_mem_bank_t**

#include <rocprofiler-sdk/agent.h> Memory bank information for an agent.

struct **rocprofiler_agent_runtime_visiblity_t**

#include <rocprofiler-sdk/agent.h> Provides an *estimate* about the runtime visibility of an agent based on the environment variables (ROCR_VISIBLE_DEVICES, HIP_VISIBLE_DEVICES, GPU_DEVICE_ORDINAL, CUDA_VISIBLE_DEVICES). Reference: https://rocm.docs.amd.com/en/latest/conceptual/gpu-isolation.html.

struct **rocprofiler_agent_v0_t**

#include <rocprofiler-sdk/agent.h> Stores the properties of an agent (CPU, GPU, etc.)

The `node_id` member is the KFD topology node id. It should be considered the "universal" indexing number. It is equivalent to the HSA-runtime HSA_AMD_AGENT_INFO_DRIVER_NODE_ID property of a `hsa_agent_t`. The `const char*` fields (`name`, `vendor_name`, etc.) are guaranteed to be valid pointers to null-terminated strings during tool finalization. Pointers to the agents via

See also

rocprofiler_query_available_agents are constant and will not be deallocated until after tool finalization. Making copies of the agent struct is also valid.

19.1.2 Buffer handling

typedef void (***rocprofiler_buffer_tracing_cb_t**)(*rocprofiler_context_id_t* context, *rocprofiler_buffer_id_t* buffer_id, *rocprofiler_record_header_t* **headers, unsigned long num_headers, void *data, uint64_t drop_count)

Async callback function.

```
for(size_t i = 0; i < num_headers; ++i)
{
    rocprofiler_record_header_t* hdr = headers[i];
    if(hdr->kind == ROCPROFILER_RECORD_KIND_PC_SAMPLE)
    {
        auto* data = static_cast<rocprofiler_pc_sample_t*>(&hdr->payload);
        ...
    }
}
```

rocprofiler_status_t **rocprofiler_create_buffer**(*rocprofiler_context_id_t* context, unsigned long size, unsigned long watermark, *rocprofiler_buffer_policy_t* policy, *rocprofiler_buffer_tracing_cb_t* callback, void *callback_data, *rocprofiler_buffer_id_t* *buffer_id)

Create buffer.

Parameters

- **context** – **[in]** Context identifier associated with buffer
- **size** – **[in]** Size of the buffer in bytes
- **watermark** – **[in]** - watermark size, where the callback is called, if set to 0 then the callback will be called on every record
- **policy** – **[in]** Behavior policy when buffer is full
- **callback** – **[in]** Callback to invoke when buffer is flushed/full
- **callback_data** – **[in]** Data to provide in callback function
- **buffer_id** – **[out]** Identification handle for buffer

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_destroy_buffer**(*rocprofiler_buffer_id_t* buffer_id)

Destroy buffer.

Note: This will destroy the buffer even if it is not empty. The user can call *rocprofiler_flush_buffer* before it to make sure the buffer is empty.

Parameters

buffer_id – **[in]**

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_flush_buffer**(*rocprofiler_buffer_id_t* buffer_id)

Flush buffer.

Parameters

buffer_id – **[in]**

Returns

rocprofiler_status_t

19.1.3 Buffer tracing

typedef int (***rocprofiler_buffer_tracing_kind_cb_t**)(*rocprofiler_buffer_tracing_kind_t* kind, void *data)

Callback function for mapping *rocprofiler_buffer_tracing_kind_t* ids to string names.

See also

rocprofiler_iterate_buffer_trace_kind_names.

typedef int (***rocprofiler_buffer_tracing_kind_operation_cb_t**)(*rocprofiler_buffer_tracing_kind_t* kind, *rocprofiler_tracing_operation_t* operation, void *data)

Callback function for mapping the operations of a given *rocprofiler_buffer_tracing_kind_t* to string names.

See also

rocprofiler_iterate_buffer_trace_kind_operation_names.

typedef int (***rocprofiler_buffer_tracing_operation_args_cb_t**)(*rocprofiler_buffer_tracing_kind_t* kind, *rocprofiler_tracing_operation_t* operation, uint32_t arg_number, const void *const arg_value_addr, int32_t arg_indirection_count, const char *arg_type, const char *arg_name, const char *arg_value_str, void *data)

Callback function for iterating over the function arguments to a traced function. This function will be invoked for each argument.

See also

rocprofiler_iterate_buffer_tracing_record_args

Param kind
[in] domain
Param operation
[in] associated domain operation
Param arg_number
[in] the argument number, starting at zero
Param arg_value_addr
[in] the address of the argument stored by rocprofiler.
Param arg_indirection_count
[in] the total number of indirection levels for the argument, e.g. int == 0, int* == 1, int** == 2
Param arg_type
[in] the typeid name of the argument (not demangled)
Param arg_name
[in] the name of the argument in the prototype (or rocprofiler union)
Param arg_value_str
[in] conversion of the argument to a string, e.g. operator<< overload
Param data
[in] user data

rocprofiler_status_t **rocprofiler_configure_buffer_tracing_service**(*rocprofiler_context_id_t* context_id, *rocprofiler_buffer_tracing_kind_t* kind, const *rocprofiler_tracing_operation_t* *operations, unsigned long operations_count, *rocprofiler_buffer_id_t* buffer_id)

Configure Buffer Tracing Service.

Parameters
- **context_id** – **[in]** Associated context to control activation of service
- **kind** – **[in]** Buffer tracing category
- **operations** – **[in]** Array of specific operations (if desired)
- **operations_count** – **[in]** Number of specific operations (if non-null set of operations)
- **buffer_id** – **[in]** Buffer to store the records in

Return values
- **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED** – *rocprofiler_configure* initialization phase has passed
- **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND** – context is not valid
- **ROCPROFILER_STATUS_ERROR_SERVICE_ALREADY_CONFIGURED** – Context has already been configured for the *rocprofiler_buffer_tracing_kind_t* kind
- **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND** – Invalid *rocprofiler_buffer_tracing_kind_t*
- **ROCPROFILER_STATUS_ERROR_OPERATION_NOT_FOUND** – Invalid operation id for *rocprofiler_buffer_tracing_kind_t* kind was found

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_buffer_tracing_kind_name**(*rocprofiler_buffer_tracing_kind_t* kind, const char **name, uint64_t *name_len)

Query the name of the buffer tracing kind. The name retrieved from this function is a string literal that is encoded in the read-only section of the binary (i.e. it is always "allocated" and never "deallocated").

Parameters
- **kind** – **[in]** Buffer tracing domain

- **name** – **[out]** If non-null and the name is a constant string that does not require dynamic allocation, this paramter will be set to the address of the string literal, otherwise it will be set to nullptr
- **name_len** – **[out]** If non-null, this will be assigned the length of the name (regardless of the name is a constant string or requires dynamic allocation)

Return values

- **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND** – Returned if the domain id is not valid
- **ROCPROFILER_STATUS_SUCCESS** – Returned if a valid domain, regardless if there is a constant string or not.

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_buffer_tracing_kind_operation_name**(*rocprofiler_buffer_tracing_kind_t* kind, *rocprofiler_tracing_operation_t* operation, const char **name, uint64_t *name_len)

Query the name of the buffer tracing kind. The name retrieved from this function is a string literal that is encoded in the read-only section of the binary (i.e. it is always "allocated" and never "deallocated").

Parameters

- **kind** – **[in]** Buffer tracing domain
- **operation** – **[in]** Enumeration id value which maps to a specific API function or event type
- **name** – **[out]** If non-null and the name is a constant string that does not require dynamic allocation, this paramter will be set to the address of the string literal, otherwise it will be set to nullptr
- **name_len** – **[out]** If non-null, this will be assigned the length of the name (regardless of the name is a constant string or requires dynamic allocation)

Return values

- **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND** – An invalid domain id
- **ROCPROFILER_STATUS_ERROR_OPERATION_NOT_FOUND** – The operation number is not recognized for the given domain
- **ROCPROFILER_STATUS_ERROR_NOT_IMPLEMENTED** – Rocprofiler does not support providing the operation name within this domain
- **ROCPROFILER_STATUS_SUCCESS** – Valid domain and operation, regardless of whether there is a constant string or not.

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_iterate_buffer_tracing_kinds**(*rocprofiler_buffer_tracing_kind_cb_t* callback, void *data)

Iterate over all the buffer tracing kinds and invokes the callback for each buffer tracing kind.

This is typically used to invoke *rocprofiler_iterate_buffer_tracing_kind_operations* for each buffer tracing kind.

Parameters

- **callback** – **[in]** Callback function invoked for each enumeration value in *rocprofiler_buffer_tracing_kind_t* with the exception of the NONE and LAST values.
- **data** – **[in]** User data passed back into the callback

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_iterate_buffer_tracing_kind_operations**(*rocprofiler_buffer_tracing_kind_t* kind, *rocprofiler_buffer_tracing_kind_operation_cb_t* callback, void *data)

Iterates over all the operations for a given *rocprofiler_buffer_tracing_kind_t* and invokes the callback with the kind and operation id. This is useful to build a map of the operation names during tool initialization instead of querying rocprofiler everytime in the callback hotpath.

Parameters

- **kind** – **[in]** which buffer tracing kind operations to iterate over
- **callback** – **[in]** Callback function invoked for each operation associated with *rocprofiler_buffer_tracing_kind_t* with the exception of the NONE and LAST values.
- **data** – **[in]** User data passed back into the callback

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_iterate_buffer_tracing_record_args**(*rocprofiler_record_header_t* record, *rocprofiler_buffer_tracing_operation_args_cb_t* callback, void *user_data)

struct **rocprofiler_buffer_tracing_hsa_api_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer HSA API Tracer Record.

struct **rocprofiler_buffer_tracing_hip_api_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer HIP API Tracer Record.

struct **rocprofiler_buffer_tracing_hip_api_ext_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer HIP API Tracer Record.

struct **rocprofiler_buffer_tracing_ompt_target_t**

#include <rocprofiler-sdk/buffer_tracing.h> Additional trace data for OpenMP target routines.

struct **rocprofiler_buffer_tracing_ompt_target_data_op_t**

struct **rocprofiler_buffer_tracing_ompt_target_kernel_t**

struct **rocprofiler_buffer_tracing_ompt_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer OMPT Tracer Record.

struct **rocprofiler_buffer_tracing_marker_api_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Marker Tracer Record.

struct **rocprofiler_buffer_tracing_rccl_api_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer RCCL API Record.

struct **rocprofiler_buffer_tracing_rocdecode_api_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer rocDecode API Record.

struct **rocprofiler_buffer_tracing_rocdecode_api_ext_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> An extended ROCProfiler rocDecode API Tracer Record which includes function arguments. Pointers are not dereferenced.

struct **rocprofiler_buffer_tracing_rocjpeg_api_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer rocJPEG API Record.

struct **rocprofiler_buffer_tracing_memory_copy_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Memory Copy Tracer Record.

struct **rocprofiler_buffer_tracing_memory_allocation_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Memory Allocation Tracer Record.

struct **rocprofiler_buffer_tracing_kernel_dispatch_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Kernel Dispatch Tracer Record.

struct **rocprofiler_buffer_tracing_kfd_event_page_migrate_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Page Migration event record from KFD.

struct **rocprofiler_buffer_tracing_kfd_event_page_fault_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Page Fault event record from KFD.

struct **rocprofiler_buffer_tracing_kfd_event_queue_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Queue event record from KFD.

struct **rocprofiler_buffer_tracing_kfd_event_unmap_from_gpu_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Unmap of memory from GPU event record from KFD.

struct **rocprofiler_buffer_tracing_kfd_event_dropped_events_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Dropped events event record, for when KFD reports that it has dropped some events.

struct **rocprofiler_buffer_tracing_kfd_page_migrate_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Page Migration (paired) record from KFD.

struct **rocprofiler_buffer_tracing_kfd_page_fault_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Page Fault (paired) record from KFD.

struct **rocprofiler_buffer_tracing_kfd_queue_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Queue suspend (paired) record from KFD.

struct **rocprofiler_buffer_tracing_scratch_memory_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Scratch Memory Tracer Record.

struct **rocprofiler_buffer_tracing_correlation_id_retirement_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Correlation ID Retirement Tracer Record.

struct **rocprofiler_buffer_tracing_runtime_initialization_record_t**

#include <rocprofiler-sdk/buffer_tracing.h> ROCProfiler Buffer Runtime Initialization Tracer Record.

rocprofiler_buffer_tracing_ompt_record_t.__unnamed1__

Public Members

rocprofiler_buffer_tracing_ompt_target_t **target**

rocprofiler_buffer_tracing_ompt_target_data_op_t **target_data_op**

rocprofiler_buffer_tracing_ompt_target_kernel_t **target_kernel**

uint64_t **reserved**[5]

19.1.4 Callback tracing

enum **rocprofiler_code_object_storage_type_t**

ROCProfiler Enumeration for code object storage types (identical values to `hsa_ven_amd_loader_code_object_storage_type_t` enumeration)

Values:

enumerator **ROCPROFILER_CODE_OBJECT_STORAGE_TYPE_NONE**

enumerator **ROCPROFILER_CODE_OBJECT_STORAGE_TYPE_FILE**

enumerator **ROCPROFILER_CODE_OBJECT_STORAGE_TYPE_MEMORY**

enumerator **ROCPROFILER_CODE_OBJECT_STORAGE_TYPE_LAST**

typedef void (***rocprofiler_callback_tracing_cb_t**)(*rocprofiler_callback_tracing_record_t* record, *rocprofiler_user_data_t* *user_data, void *callback_data)

API Tracing callback function. This function is invoked twice per API function: once before the function is invoked and once after the function is invoked. The external correlation id value within the record is assigned the value at the top of the external correlation id stack. It is permissible to invoke *rocprofiler_push_external_correlation_id* within the enter phase; when a new external correlation id is pushed during the enter phase, rocprofiler will use that external correlation id for any async events and provide the new external correlation id during the exit callback… In other words, pushing a new external correlation id within the enter callback will result in that external correlation id value in the exit callback (which may or may not be different from the external correlation id value in the enter callback). If a tool pushes new external correlation ids in the enter phase, it is recommended to pop the external correlation id in the exit callback.

Param record

[in] Callback record data

Param user_data
[inout] This paramter can be used to retain information in between the enter and exit phases.
Param callback_data
[in] User data provided when configuring the callback tracing service

typedef int (***rocprofiler_callback_tracing_kind_cb_t**)(*rocprofiler_callback_tracing_kind_t* kind, void *data)

Callback function for mapping *rocprofiler_callback_tracing_kind_t* ids to string names.

See also

rocprofiler_iterate_callback_tracing_kind_names.

typedef int (***rocprofiler_callback_tracing_kind_operation_cb_t**)(*rocprofiler_callback_tracing_kind_t* kind, *rocprofiler_tracing_operation_t* operation, void *data)

Callback function for mapping the operations of a given *rocprofiler_callback_tracing_kind_t* to string names.

See also

rocprofiler_iterate_callback_tracing_kind_operation_names.

typedef int (***rocprofiler_callback_tracing_operation_args_cb_t**)(*rocprofiler_callback_tracing_kind_t* kind, *rocprofiler_tracing_operation_t* operation, uint32_t arg_number, const void *const arg_value_addr, int32_t arg_indirection_count, const char *arg_type, const char *arg_name, const char *arg_value_str, int32_t arg_dereference_count, void *data)

Callback function for iterating over the function arguments to a traced function. This function will be invoked for each argument.

See also

rocprofiler_iterate_callback_tracing_operation_args

Param kind
[in] domain
Param operation
[in] associated domain operation
Param arg_number
[in] the argument number, starting at zero
Param arg_value_addr
[in] the address of the argument stored by rocprofiler.
Param arg_indirection_count
[in] the total number of indirection levels for the argument, e.g. int == 0, int* == 1, int** == 2
Param arg_type
[in] the typeid name of the argument

Param arg_name
[in] the name of the argument in the prototype (or rocprofiler union)
Param arg_value_str
[in] conversion of the argument to a string, e.g. operator<< overload
Param arg_dereference_count
[in] the number of times the argument was dereferenced when it was converted to a string
Param data
[in] user data

rocprofiler_status_t **rocprofiler_configure_callback_tracing_service**(*rocprofiler_context_id_t* context_id, *rocprofiler_callback_tracing_kind_t* kind, const *rocprofiler_tracing_operation_t* *operations, unsigned long operations_count, *rocprofiler_callback_tracing_cb_t* callback, void *callback_args)

Configure Callback Tracing Service. The callback tracing service provides two synchronous callbacks around an API function on the same thread as the application which is invoking the API function. This function can only be invoked once per *rocprofiler_callback_tracing_kind_t* value, i.e. it can be invoked once for the HSA API, once for the HIP API, and so on but it will fail if it is invoked for the HSA API twice. Please note, the callback API does have the potentially non-trivial overhead of copying the function arguments into the record. If you are willing to let rocprofiler record the timestamps, do not require synchronous notifications of the API calls, and want to lowest possible overhead, use the.

See also

Asynchronous Tracing Service.

Parameters
- **context_id** – **[in]** Context to associate the service with
- **kind** – **[in]** The domain of the callback tracing service
- **operations** – **[in]** Array of operations in the domain (i.e. enum values which identify specific API functions). If this is null, all API functions in the domain will be traced
- **operations_count** – **[in]** If the operations array is non-null, set this to the size of the array.
- **callback** – **[in]** The function to invoke before and after an API function
- **callback_args** – **[in]** Data provided to every invocation of the callback function

Return values
- **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED** – Invoked outside of the initialization function in *rocprofiler_tool_configure_result_t* provided to rocprofiler via *rocprofiler_configure* function
- **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND** – The provided context is not valid/registered
- **ROCPROFILER_STATUS_ERROR_SERVICE_ALREADY_CONFIGURED** – if the same *rocprofiler_callback_tracing_kind_t* value is provided more than once (per context) — in other words, we do not support overriding or combining the operations in separate function calls.

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_callback_tracing_kind_name**(*rocprofiler_callback_tracing_kind_t* kind, const char **name, uint64_t *name_len)

Query the name of the callback tracing kind. The name retrieved from this function is a string literal that is encoded in the read-only section of the binary (i.e. it is always "allocated" and never "deallocated").

Parameters

- **kind** – **[in]** Callback tracing domain
- **name** – **[out]** If non-null and the name is a constant string that does not require dynamic allocation, this paramter will be set to the address of the string literal, otherwise it will be set to nullptr
- **name_len** – **[out]** If non-null, this will be assigned the length of the name (regardless of the name is a constant string or requires dynamic allocation)

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_callback_tracing_kind_operation_name**(*rocprofiler_callback_tracing_kind_t* kind, *rocprofiler_tracing_operation_t* operation, const char **name, uint64_t *name_len)

Query the name of the callback tracing kind. The name retrieved from this function is a string literal that is encoded in the read-only section of the binary (i.e. it is always "allocated" and never "deallocated").

Parameters

- **kind** – **[in]** Callback tracing domain
- **operation** – **[in]** Enumeration id value which maps to a specific API function or event type
- **name** – **[out]** If non-null and the name is a constant string that does not require dynamic allocation, this paramter will be set to the address of the string literal, otherwise it will be set to nullptr
- **name_len** – **[out]** If non-null, this will be assigned the length of the name (regardless of the name is a constant string or requires dynamic allocation)

Return values

- **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND** – Domain id is not valid
- **ROCPROFILER_STATUS_SUCCESS** – Valid domain provided, regardless if there is a constant string or not.

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_iterate_callback_tracing_kinds**(*rocprofiler_callback_tracing_kind_cb_t* callback, void *data)

Iterate over all the mappings of the callback tracing kinds and get a callback for each kind.

Parameters

- **callback** – **[in]** Callback function invoked for each enumeration value in *rocprofiler_callback_tracing_kind_t* with the exception of the NONE and LAST values.
- **data** – **[in]** User data passed back into the callback

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_iterate_callback_tracing_kind_operations**(*rocprofiler_callback_tracing_kind_t* kind, *rocprofiler_callback_tracing_kind_operation_cb_t* callback, void *data)

Iterates over all the mappings of the operations for a given *rocprofiler_callback_tracing_kind_t* and invokes the

callback with the kind id, operation id, and user-provided data.

Parameters

- **kind** – **[in]** which tracing callback kind operations to iterate over
- **callback** – **[in]** Callback function invoked for each operation associated with *rocprofiler_callback_tracing_kind_t* with the exception of the NONE and LAST values.
- **data** – **[in]** User data passed back into the callback

Return values

- **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND** – Invalid domain id
- **ROCPROFILER_STATUS_SUCCESS** – Valid domain

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_iterate_callback_tracing_kind_operation_args**(*rocprofiler_callback_tracing_record_t* record, *rocprofiler_callback_tracing_operation_args_* callback, int32_t max_dereference_count, void *user_data)

ROCPROFILER_CODE_OBJECT_ID_NONE

The NULL value of a code object id. Used when code object is unknown.

struct **rocprofiler_callback_tracing_hsa_api_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler HSA API Callback Data.

struct **rocprofiler_callback_tracing_hip_api_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler HIP runtime and compiler API Tracer Callback Data.

struct **rocprofiler_callback_tracing_ompt_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler OMPT Callback Data.

struct **rocprofiler_callback_tracing_marker_api_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Marker Tracer Callback Data.

struct **rocprofiler_callback_tracing_rccl_api_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler RCCL API Callback Data.

struct **rocprofiler_callback_tracing_rocdecode_api_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler rocDecode API Callback Data.

struct **rocprofiler_callback_tracing_rocjpeg_api_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler rocJPEG API Callback Data.

struct **rocprofiler_callback_tracing_code_object_load_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Code Object Load Tracer Callback Record.

struct **rocprofiler_callback_tracing_code_object_kernel_symbol_register_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Code Object Kernel Symbol Tracer Callback Record.

struct **rocprofiler_callback_tracing_code_object_host_kernel_symbol_register_data_t**

struct **rocprofiler_callback_tracing_kernel_dispatch_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Kernel Dispatch Callback Tracer Record.

struct **rocprofiler_callback_tracing_memory_copy_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Memory Copy Callback Tracer Record.

The timestamps in this record will only be non-zero in the *ROCPROFILER_CALLBACK_PHASE_EXIT* callback

struct **rocprofiler_callback_tracing_memory_allocation_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Memory Allocation Tracer Record.

struct **rocprofiler_callback_tracing_scratch_memory_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Scratch Memory Callback Data.

struct **rocprofiler_callback_tracing_runtime_initialization_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Runtime Initialization Data.

struct **rocprofiler_callback_tracing_hip_stream_data_t**

#include <rocprofiler-sdk/callback_tracing.h> ROCProfiler Stream Handle Callback Data.

rocprofiler_callback_tracing_code_object_load_data_t.__unnamed3__

Public Members

rocprofiler_agent_id_t **rocp_agent**

Deprecated. Renamed to agent_id.

rocprofiler_agent_id_t **agent_id**

The agent on which this loaded code object is loaded.

rocprofiler_callback_tracing_code_object_load_data_t.__unnamed5__

Public Members

struct rocprofiler_callback_tracing_code_object_load_data_t

struct rocprofiler_callback_tracing_code_object_load_data_t

rocprofiler_callback_tracing_code_object_load_data_t.__unnamed5__.__unnamed7__

rocprofiler_callback_tracing_code_object_load_data_t.__unnamed5__.__unnamed9__

19.1.5 Context management

rocprofiler_status_t **rocprofiler_create_context**(*rocprofiler_context_id_t* *context_id)

Create context.

Parameters

context_id – **[out]** Context identifier

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_start_context**(*rocprofiler_context_id_t* context_id)

Start context.

Parameters

context_id – **[in]** Identifier for context to be activated

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_stop_context**(*rocprofiler_context_id_t* context_id)

Stop context.

Parameters

context_id – **[in]** Identifier for context to be deactivated

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_context_is_active**(*rocprofiler_context_id_t* context_id, int *status)

Query whether context is currently active.

Parameters

- **context_id** – **[in]** Context identifier for the query
- **status** – **[out]** If context is active, this will be a nonzero value

Return values

- **ROCPROFILER_STATUS_SUCCESS** – The input context id identified a registered context
- **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND** – The input context id did not identify a registered context

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_context_is_valid**(*rocprofiler_context_id_t* context_id, int *status)

Query whether the context is valid.

Parameters

- **context_id** – **[in]** Context identifier for the query
- **status** – **[out]** If context is invalid, this will be a nonzero value

Returns

rocprofiler_status_t

ROCPROFILER_CONTEXT_NONE

The NULL Context handle.

19.1.6 Counter config

rocprofiler_status_t **rocprofiler_create_counter_config**(*rocprofiler_agent_id_t* agent_id, *rocprofiler_counter_id_t* *counters_list, unsigned long counters_count, *rocprofiler_counter_config_id_t* *config_id)

(experimental) Create Counter Configuration. A config is bound to an agent but can be used across many contexts. The config has a fixed set of counters that are collected (and specified by counter_list). The available counters for

an agent can be queried using *rocprofiler_iterate_agent_supported_counters*. An existing config may be supplied via config_id to use as a base for the new config. All counters in the existing config will be copied over to the new config. The existing config will remain unmodified and usable with the new config id being returned in config_id.

Parameters

- **agent_id** – **[in]** Agent identifier
- **counters_list** – **[in]** List of GPU counters
- **counters_count** – **[in]** Size of counters list
- **config_id** – **[inout]** Identifier for GPU counters group. If an existing config is supplied, that profiles counters will be copied over to a new config (returned via this id)

Return values

- **ROCPROFILER_STATUS_SUCCESS** – if config created
- **ROCPROFILER_STATUS_ERROR** – if config could not be created

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_destroy_counter_config**(*rocprofiler_counter_config_id_t* config_id)

(experimental) Destroy Profile Configuration.

Parameters

config_id – **[in]**

Return values

- **ROCPROFILER_STATUS_SUCCESS** – if config destroyed
- **ROCPROFILER_STATUS_ERROR** – if config could not be destroyed

Returns

rocprofiler_status_t

19.1.7 Counters

typedef *rocprofiler_status_t* (***rocprofiler_available_counters_cb_t**)(*rocprofiler_agent_id_t* agent_id, *rocprofiler_counter_id_t* *counters, unsigned long num_counters, void *user_data)

(experimental) Callback that gives a list of counters available on an agent. The counters variable is owned by rocprofiler and should not be free'd.

Param agent_id

[in] Agent ID of the current callback

Param counters

[in] An array of counters that are avialable on the agent *rocprofiler_iterate_agent_supported_counters* was called on.

Param num_counters

[in] Number of counters contained in counters

Param user_data

[in] User data supplied by *rocprofiler_iterate_agent_supported_counters*

rocprofiler_status_t **rocprofiler_query_record_counter_id**(*rocprofiler_counter_instance_id_t* id, *rocprofiler_counter_id_t* *counter_id)

(experimental) Query counter id information from record_id.

Parameters

- **id** – **[in]** record id from *rocprofiler_counter_record_t*
- **counter_id** – **[out]** counter id associated with the record

Return values

ROCPROFILER_STATUS_SUCCESS – if id decoded

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_record_dimension_position**(*rocprofiler_counter_instance_id_t* id, *rocprofiler_counter_dimension_id_t* dim, unsigned long *pos)

(experimental) Query dimension position from record_id. If the dimension does not exist in the counter, the return will be 0.

Parameters

- **id** – **[in]** record id from *rocprofiler_counter_record_t*
- **dim** – **[in]** dimension for which positional info is requested (currently only 0 is allowed, i.e. flat array without dimension).
- **pos** – **[out]** value of the dimension in id

Return values

ROCPROFILER_STATUS_SUCCESS – if dimension decoded

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_counter_info**(*rocprofiler_counter_id_t* counter_id, *rocprofiler_counter_info_version_id_t* version, void *info)

(experimental) Query Counter info such as name or description.

Parameters

- **counter_id** – **[in]** counter to get info for
- **version** – **[in]** Version of struct in info, see *rocprofiler_counter_info_version_id_t* for available types
- **info** – **[out]** rocprofiler_counter_info_{version}_t struct to write info to.

Return values

- **ROCPROFILER_STATUS_SUCCESS** – if counter found
- **ROCPROFILER_STATUS_ERROR_COUNTER_NOT_FOUND** – if counter not found
- **ROCPROFILER_STATUS_ERROR_INCOMPATIBLE_ABI** – Version is not supported

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_iterate_agent_supported_counters**(*rocprofiler_agent_id_t* agent_id, *rocprofiler_available_counters_cb_t* cb, void *user_data)

(experimental) Query Agent Counters Availability.

Parameters

- **agent_id** – **[in]** GPU agent identifier
- **cb** – **[in]** callback to caller to get counters
- **user_data** – **[in]** data to pass into the callback

Return values

- **ROCPROFILER_STATUS_SUCCESS** – if counters found for agent
- **ROCPROFILER_STATUS_ERROR** – if no counters found for agent

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_create_counter**(const char *name, unsigned long name_len, const char *expr, unsigned long expr_len, const char *description, unsigned long description_len, *rocprofiler_agent_id_t* agent, *rocprofiler_counter_id_t* *counter_id)

(experimental) Creates a new counter based on a derived metric provided. The counter will only be available for counter collection profiles created after the addition of this counter. Due to the regeneration of internal ASTs and dimension cache, this call may be slow and should generally be avoided in performance sensitive code blocks (i.e. dispatch callbacks).

Parameters

- **name** – **[in]** The name of the new counter.
- **name_len** – **[in]** The length of the counter name.
- **expr** – **[in]** The counter expression, formatted identically to YAML counter definitions.
- **expr_len** – **[in]** The length of the expression.
- **agent** – **[in]** The *rocprofiler_agent_id_t* specifying the agent for which to create the counter.
- **description** – **[in]** The description of the new counter (optional).
- **description_len** – **[in]** The length of the description.
- **counter_id** – **[out]** The *rocprofiler_counter_id_t* of the created counter.

Return values

- **ROCPROFILER_STATUS_SUCCESS** – if the counter was successfully created.
- **ROCPROFILER_STATUS_ERROR_AST_GENERATION_FAILED** – if the counter could not be created.
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – if a counter argument is incorrect
- **ROCPROFILER_STATUS_ERROR_AGENT_NOT_FOUND** – if the agent is not found

Returns

rocprofiler_status_t

struct **rocprofiler_counter_info_v0_t**

#include <rocprofiler-sdk/counters.h> (experimental) Counter info struct version 0

struct **rocprofiler_counter_dimension_info_t**

#include <rocprofiler-sdk/counters.h> (experimental) Represents metadata about a single dimension of a counter instance.

This structure provides the name of the dimension (e.g., "XCC", "SE", etc.) and the index indicating the position of a specific counter instance within that dimension.

struct **rocprofiler_counter_record_dimension_instance_info_t**

#include <rocprofiler-sdk/counters.h> (experimental) Describes a specific counter instance and its position across multiple dimensions.

This structure provides the unique instance ID, associated counter ID, number of dimensions for the instance, and a pointer to an array of metadata describing each dimension's name and index.

struct **rocprofiler_counter_info_v1_t**

#include <rocprofiler-sdk/counters.h> (experimental) Counter info struct version 1. Combines information from *rocprofiler_counter_info_v0_t* with the dimension information.

19.1.8 Device counting service

typedef *rocprofiler_status_t* (*__rocprofiler_device_counting_agent_cb_t__)(*rocprofiler_context_id_t* context_id, *rocprofiler_counter_config_id_t* config_id)

(experimental) Callback to set the profile config for the agent.

Param context_id

[in] context id

Param config_id

[in] Profile config detailing the counters to collect for this kernel

Retval ROCPROFILER_STATUS_ERROR_PROFILE_NOT_FOUND

Returned if the config_id is not found

Retval ROCPROFILER_STATUS_ERROR_CONTEXT_INVALID
Returned if the ctx is not valid
Retval ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED
Returned if attempting to make this call outside of context startup.
Retval ROCPROFILER_STATUS_ERROR_AGENT_MISMATCH
Agent of profile does not match agent of the context.
Retval ROCPROFILER_STATUS_SUCCESS
Returned if succesfully configured
Return
rocprofiler_status_t

typedef void (*`rocprofiler_device_counting_service_cb_t`)(*rocprofiler_context_id_t* context_id, *rocprofiler_agent_id_t* agent_id, *rocprofiler_device_counting_agent_cb_t* set_config, void *user_data)

(experimental) Configure Profile Counting Service for agent. Called when the context is started. Selects the counters to be used for agent profiling.
Param context_id
[in] context id
Param agent_id
[in] agent id
Param set_config
[in] Function to call to set the profile config (see rocprofiler_device_counting_agent_cb_t)
Param user_data
[in] Data supplied to rocprofiler_configure_device_counting_service

rocprofiler_status_t `rocprofiler_configure_device_counting_service`(*rocprofiler_context_id_t* context_id, *rocprofiler_buffer_id_t* buffer_id, *rocprofiler_agent_id_t* agent_id, *rocprofiler_device_counting_service_cb_t* cb, void *user_data)

(experimental) Configure Device Counting Service for agent. There may only be one counting service configured per agent in a context and can be only one active context that is profiling a single agent at a time. Multiple agent contexts can be started at the same time if they are profiling different agents.
Parameters
- `context_id` – **[in]** context id
- `buffer_id` – **[in]** id of the buffer to use for the counting service. When rocprofiler_sample_device_counting_service is called, counter data will be written to this buffer. If the input buffer id is null (i.e. *rocprofiler_buffer_id_t*`{.handle = 0}`), the counter data will not be written to a buffer and will only be returned in the output_records of rocprofiler_sample_device_counting_service
- `agent_id` – **[in]** agent to configure profiling on.
- `cb` – **[in]** Callback called when the context is started for the tool to specify what counters to collect (*rocprofiler_counter_config_id_t*).
- `user_data` – **[in]** User supplied data to be passed to the callback cb when triggered

Return values
- `ROCPROFILER_STATUS_ERROR_CONTEXT_INVALID` – Returned if the context does not exist.
- `ROCPROFILER_STATUS_ERROR_BUFFER_NOT_FOUND` – Returned if the buffer is not found.
- `ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT` – Returned if context already has agent profiling configured for agent_id.
- `ROCPROFILER_STATUS_SUCCESS` – Returned if succesfully configured

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_sample_device_counting_service**(*rocprofiler_context_id_t* context_id, *rocprofiler_user_data_t* user_data, *rocprofiler_counter_flag_t* flags, *rocprofiler_counter_record_t* *output_records, unsigned long *rec_count)

(experimental) Trigger a read of the counter data for the agent profile. The counter data will be written to the buffer specified in rocprofiler_configure_device_counting_service. The data in *rocprofiler_user_data_t* will be written to the buffer along with the counter data. flags can be used to specify if this call should be performed asynchronously (default is synchronous).

Parameters

- **context_id** – **[in]** context id
- **user_data** – **[in]** User supplied data, included in records outputted to buffer.
- **flags** – **[in]** Flags to specify how the counter data should be collected (defaults to sync).
- **output_records** – **[in]** (Optional) Provides the values immediately instead of outputting to buffer. Must be allocated by caller.
- **rec_count** – **[in]** (Optional) On entry, this is the maximum number of records rocprof can store in output_records. On exit, contains the number of actual records.

Return values

- **ROCPROFILER_STATUS_ERROR_CONTEXT_INVALID** – Returned if the context does not exist or the context is not configured for agent profiling.
- **ROCPROFILER_STATUS_ERROR_CONTEXT_ERROR** – Returned if another operation is in progress (start/stop ctx or another read).
- **ROCPROFILER_STATUS_ERROR** – Returned if HSA has not been initialized yet.
- **ROCPROFILER_STATUS_ERROR_OUT_OF_RESOURCES** – Returned output_records is set but size is too small to store results
- **ROCPROFILER_STATUS_SUCCESS** – Returned if read request was successful.
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – Returned If ASYNC is being used while output_records is not null.

Returns

rocprofiler_status_t

19.1.9 Dispatch counting service

typedef void (***rocprofiler_dispatch_counting_service_cb_t**)(*rocprofiler_dispatch_counting_service_data_t* dispatch_data, *rocprofiler_counter_config_id_t* *config, *rocprofiler_user_data_t* *user_data, void *callback_data_args)

(experimental) Kernel Dispatch Callback. This is a callback that is invoked before the kernel is enqueued into the HSA queue. What counters to collect for a kernel are set via passing back a profile config (config) in this callback. These counters will be collected and emplaced in the buffer with *rocprofiler_buffer_id_t* used when setting up this callback.

See also

rocprofiler_dispatch_counting_service_data_t

Param dispatch_data
[in]

Param config
[out] Profile config detailing the counters to collect for this kernel
Param user_data
[out] User data unique to this dispatch. Returned in record callback
Param callback_data_args
[in] Callback supplied via buffered_dispatch_counting_service

typedef void (***rocprofiler_dispatch_counting_record_cb_t**)(*rocprofiler_dispatch_counting_service_data_t* dispatch_data, *rocprofiler_counter_record_t* *record_data, unsigned long record_count, *rocprofiler_user_data_t* user_data, void *callback_data_args)

(experimental) Counting record callback. This is a callback is invoked when the kernel execution is complete and contains the counter profile data requested in *rocprofiler_dispatch_counting_service_cb_t*. Only used with *rocprofiler_configure_callback_dispatch_counting_service*.

See also

rocprofiler_dispatch_counting_service_data_t

Param dispatch_data
[in]
Param record_data
[in] Counter record data.
Param record_count
[in] Number of counter records.
Param user_data
[in] User data instance from dispatch callback
Param callback_data_args
[in] Callback supplied via buffered_dispatch_counting_service

rocprofiler_status_t **rocprofiler_configure_buffer_dispatch_counting_service**(*rocprofiler_context_id_t* context_id, *rocprofiler_buffer_id_t* buffer_id, *rocprofiler_dispatch_counting_service_cb_t* callback, void *callback_data_args)

(experimental) Configure buffered dispatch profile Counting Service. Collects the counters in dispatch packets and stores them in a buffer with `buffer_id`. The buffer may contain packets from more than one dispatch (denoted by correlation id). Will trigger the callback based on the parameters setup in buffer_id_t.

NOTE: Interface is up for comment as to whether restrictions on agent should be made here (limiting the CB based on agent) or if the restriction should be performed by the tool in *rocprofiler_dispatch_counting_service_cb_t* (i.e. tool code checking the agent param to see if they want to profile it).

Interface is up for comment as to whether restrictions on agent should be made here (limiting the CB based on agent) or if the restriction should be performed by the tool in *rocprofiler_dispatch_counting_service_cb_t* (i.e. tool code checking the agent param to see if they want to profile it).

Parameters
- **context_id** – **[in]** context id
- **buffer_id** – **[in]** id of the buffer to use for the counting service
- **callback** – **[in]** callback to perform when dispatch is enqueued
- **callback_data_args** – **[in]** callback data

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_configure_callback_dispatch_counting_service**(*rocprofiler_context_id_t* context_id, *rocprofiler_dispatch_counting_service_cb_t* dispatch_callback, void *dispatch_callback_args, *rocprofiler_dispatch_counting_record_cb_t* record_callback, void *record_callback_args)

(experimental) Configure buffered dispatch profile Counting Service. Collects the counters in dispatch packets and calls a callback with the counters collected during that dispatch.

Parameters
- **context_id** – **[in]** context id
- **dispatch_callback** – **[in]** callback to perform when dispatch is enqueued
- **dispatch_callback_args** – **[in]** callback data for dispatch callback
- **record_callback** – **[in]** Record callback for completed profile data
- **record_callback_args** – **[in]** Callback args for record callback

Returns
rocprofiler_status_t

struct **rocprofiler_dispatch_counting_service_data_t**

#include <rocprofiler-sdk/dispatch_counting_service.h> (experimental) Kernel dispatch data for profile counting callbacks.

struct **rocprofiler_dispatch_counting_service_record_t**

#include <rocprofiler-sdk/dispatch_counting_service.h> (experimental) ROCProfiler Profile Counting Counter Record Header Information

This is buffer equivalent of *rocprofiler_dispatch_counting_service_data_t*

19.1.10 External correlation

enum **rocprofiler_external_correlation_id_request_kind_t**

(experimental) ROCProfiler External Correlation ID Operations.

These kinds correspond to callback and buffered tracing kinds (

See also

rocprofiler_callback_tracing_kind_t and *rocprofiler_buffer_tracing_kind_t*) which generate correlation IDs. Typically, rocprofiler-sdk uses the most recent external correlation ID on the current thread set via *rocprofiler_push_external_correlation_id*; however, this approach can be problematic if a new external correlation ID should be set before the *ROCPROFILER_CALLBACK_PHASE_ENTER* callback or if relevant external correlation IDs are desired when the buffered tracing is used. Thus, rocprofiler-sdk provides a way for tools to get a callback whenever an external correlation ID is needed. However, this can add significant overhead for those who only need these callbacks for, say, kernel dispatches while the HSA API is being traced (i.e. lots of callbacks for HSA API functions). The enumeration below is provided to ensure that tools can default

to using the external correlation IDs set via the push/pop methods where the external correlation ID value is not important while also getting a request for an external correlation ID for other tracing kinds.

Values:

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_NONE**

Unknown kind.

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_HSA_CORE_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_HSA_AMD_EXT_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_HSA_IMAGE_EXT_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_HSA_FINALIZE_EXT_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_HIP_RUNTIME_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_HIP_COMPILER_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_MARKER_CORE_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_MARKER_CONTROL_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_MARKER_NAME_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_MEMORY_COPY**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_KERNEL_DISPATCH**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_SCRATCH_MEMORY**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_RCCL_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_OMPT**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_MEMORY_ALLOCATION**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_ROCDECODE_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_ROCJPEG_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_MARKER_CORE_RANGE_API**

enumerator **ROCPROFILER_EXTERNAL_CORRELATION_REQUEST_LAST**

typedef int (***rocprofiler_external_correlation_id_request_cb_t**)(*rocprofiler_thread_id_t* thread_id, *rocprofiler_context_id_t* context_id, *rocprofiler_external_correlation_id_request_kind_t* kind, *rocprofiler_tracing_operation_t* operation, uint64_t internal_corr_id_value, *rocprofiler_user_data_t* *external_corr_id_value, void *data)

(experimental) Callback requesting a value for the external correlation id.

Param thread_id
: **[in]** Id of the thread making the request

Param context_id
: **[in]** Id of the context making the request

Param kind
: **[in]** Origin of the external correlation id request

Param operation
: **[in]** Regardless of whether callback or buffer tracing is being used, the operation value will be the same, i.e., regardless of whether callback kind is *ROCPROFILER_CALLBACK_TRACING_HSA_CORE_API* or the buffer record kind is *ROCPROFILER_BUFFER_TRACING_HSA_CORE_API*, the data/record for `hsa_init` will have an operation value of ROCPROFILER_HSA_CORE_API_ID_hsa_init.

Param internal_corr_id_value
: **[in]** Current internal correlation ID value for the request

Param external_corr_id_value
: **[out]** Set this value to the desired external correlation ID value

Param data
: **[in]** The `callback_args` value passed to *rocprofiler_configure_external_correlation_id_request_service*.

Retval 0
: Used to indicate the tool had zero issues setting the external correlation ID field

Retval 1
: (or any other non-zero number) Used to indicate the callback did not set an external correlation ID value and the thread-local value for the most recently pushed external correlation ID should be used instead

Return
: int

rocprofiler_status_t **rocprofiler_configure_external_correlation_id_request_service**(*rocprofiler_context_id_t* context_id, const *rocprofiler_external_correlation_id_req* *kinds, unsigned long kinds_count, *rocprofiler_external_correlation_id_req* callback, void *callback_args)

(experimental) Configure External Correlation ID Request Service.

Parameters
- **context_id** – **[in]** Context to associate the service with
- **kinds** – **[in]** Array of *rocprofiler_external_correlation_id_request_kind_t* values. If

this parameter is null, all tracing operations will invoke the callback to request an external correlation ID.
- **kinds_count** – **[in]** If the kinds array is non-null, set this to the size of the array.
- **callback** – **[in]** The function to invoke for an external correlation ID request
- **callback_args** – **[in]** Data provided to every invocation of the callback function

Return values
- **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED** – Invoked outside of the initialization function in *rocprofiler_tool_configure_result_t* provided to rocprofiler via *rocprofiler_configure* function
- **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND** – The provided context is not valid/registered
- **ROCPROFILER_STATUS_ERROR_SERVICE_ALREADY_CONFIGURED** – if the same *rocprofiler_callback_tracing_kind_t* value is provided more than once (per context) — in other words, we do not support overriding or combining the kinds in separate function calls.

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_external_correlation_id_request_kind_name**(*rocprofiler_external_correlation_id_*
kind, const char **name, uint64_t *name_len)

Query the name of the external correlation request kind. The name retrieved from this function is a string literal that is encoded in the read-only section of the binary (i.e. it is always "allocated" and never "deallocated").

Parameters
- **kind** – **[in]** External correlation id request domain
- **name** – **[out]** If non-null and the name is a constant string that does not require dynamic allocation, this paramter will be set to the address of the string literal, otherwise it will be set to nullptr
- **name_len** – **[out]** If non-null, this will be assigned the length of the name (regardless of the name is a constant string or requires dynamic allocation)

Return values
- **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND** – Returned if the domain id is not valid
- **ROCPROFILER_STATUS_SUCCESS** – Returned if a valid domain, regardless if there is a constant string or not.

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_push_external_correlation_id**(*rocprofiler_context_id_t* context, *rocprofiler_thread_id_t* tid, *rocprofiler_user_data_t* external_correlation_id)

Push default value for `external` field in *rocprofiler_correlation_id_t* onto stack.

External correlation ids are thread-local values. However, if rocprofiler internally requests an external correlation id on a non-main thread and an external correlation id has not been pushed for this thread, the external correlation ID will default to the latest external correlation id on the main thread — this allows tools to push an external correlation id once on the main thread for, say, the MPI rank or process-wide UUID and this value will be used by all subsequent child threads.

See also

rocprofiler_get_thread_id

Parameters

- **context** – **[in]** Associated context
- **tid** – **[in]** thread identifier.
- **external_correlation_id** – **[in]** User data to place in external field in *rocprofiler_correlation_id_t*

Return values

- **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND** – Context does not exist
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – Thread id is not valid

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_pop_external_correlation_id**(*rocprofiler_context_id_t* context, *rocprofiler_thread_id_t* tid, *rocprofiler_user_data_t* *external_correlation_id)

Pop default value for `external` field in *rocprofiler_correlation_id_t* off of stack.

See also

rocprofiler_get_thread_id

Parameters

- **context** – **[in]** Associated context
- **tid** – **[in]** thread identifier.
- **external_correlation_id** – **[out]** Correlation id data popped off the stack

Return values

- **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND** – Context does not exist
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – Thread id is not valid

Returns

rocprofiler_status_t

19.1.11 Intercept table

typedef void (***rocprofiler_intercept_library_cb_t**)(*rocprofiler_intercept_table_t* type, uint64_t lib_version, uint64_t lib_instance, void **tables, uint64_t num_tables, void *user_data)

(experimental) Callback type when a new runtime library is loaded.

See also

rocprofiler_at_intercept_table_registration

Param type

[in] Type of API table

Param lib_version
[in] Major, minor, and patch version of library encoded into single number similar to ROCPROFILER_VERSION
Param lib_instance
[in] The number of times this runtime library has been registered previously
Param tables
[in] An array of pointers to the API tables
Param num_tables
[in] The size of the array of pointers to the API tables
Param user_data
[in] The pointer to the data provided to rocprofiler_at_intercept_table_registration

rocprofiler_status_t **rocprofiler_query_intercept_table_name**(*rocprofiler_intercept_table_t* kind, const char **name, uint64_t *name_len)

(experimental) Query the name of the intercept table. The name retrieved from this function is a string literal that is encoded in the read-only section of the binary (i.e. it is always "allocated" and never "deallocated").

Parameters
- **kind** – **[in]** Intercept table kind
- **name** – **[out]** If non-null and the name is a constant string that does not require dynamic allocation, this paramter will be set to the address of the string literal, otherwise it will be set to nullptr
- **name_len** – **[out]** If non-null, this will be assigned the length of the name (regardless of the name is a constant string or requires dynamic allocation)

Return values
- **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND** – Returned if the domain id is not valid
- **ROCPROFILER_STATUS_SUCCESS** – Returned if a valid domain, regardless if there is a constant string or not.

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_at_intercept_table_registration**(*rocprofiler_intercept_library_cb_t* callback, int libs, void *data)

19.1.12 Internal threading management

typedef void (***rocprofiler_internal_thread_library_cb_t**)(*rocprofiler_runtime_library_t*, void*)

(experimental) Callback type before and after internal thread creation.

See also

rocprofiler_at_internal_thread_create

rocprofiler_status_t **rocprofiler_at_internal_thread_create**(*rocprofiler_internal_thread_library_cb_t* precreate, *rocprofiler_internal_thread_library_cb_t* postcreate, int libs, void *data)

(experimental) Invoke this function to receive callbacks before and after the creation of an internal thread by a library which as invoked on the thread which is creating the internal thread(s).

Use the *rocprofiler_runtime_library_t* enumeration for specifying which libraries you want callbacks before

and after the library creates an internal thread. These callbacks will be invoked on the thread that is about to create the new thread (not on the newly created thread). In thread-aware tools that wrap pthread_create, this can be used to disable the wrapper before the pthread_create invocation and re-enable the wrapper afterwards. In many cases, tools will want to ignore the thread(s) created by rocprofiler since these threads do not exist in the normal application execution, whereas the internal threads for HSA, HIP, etc. are created in normal application execution; however, the HIP, HSA, etc. internal threads are typically background threads which just monitor kernel completion and are unlikely to contribute to any performance issues. Please note that the postcreate callback is guaranteed to be invoked after the underlying system call to create a new thread but it does not guarantee that the new thread has been started. Please note, that once these callbacks are registered, they cannot be removed so the caller is responsible for ignoring these callbacks if they want to ignore them beyond a certain point in the application.

Parameters

- **precreate** – **[in]** Callback invoked immediately before a new internal thread is created
- **postcreate** – **[in]** Callback invoked immediately after a new internal thread is created
- **libs** – **[in]** Bitwise-or of libraries, e.g. ROCPROFILER_LIBRARY | ROCPROFILER_MARKER_LIBRARY means the callbacks will be invoked whenever rocprofiler and/or the marker library create internal threads but not when the HSA or HIP libraries create internal threads.
- **data** – **[in]** Data shared between callbacks

Return values

ROCPROFILER_STATUS_SUCCESS – There are currently no conditions which result in any other value, even if internal threads have already been created

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_create_callback_thread**(*rocprofiler_callback_thread_t* *cb_thread_id)

(experimental) Create a handle to a unique thread (created by rocprofiler) which, when associated with a particular buffer, will guarantee those buffered results always get delivered on the same thread. This is useful to prevent/control thread-safety issues and/or enable multithreaded processing of buffers with non-overlapping data

Parameters

cb_thread_id – **[in]** User-provided pointer to a *rocprofiler_callback_thread_t*

Return values

- **ROCPROFILER_STATUS_SUCCESS** – Successful thread creation
- **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED** – Thread creation is no longer available post-initialization
- **ROCPROFILER_STATUS_ERROR** – Failed to create thread

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_assign_callback_thread**(*rocprofiler_buffer_id_t* buffer_id, *rocprofiler_callback_thread_t* cb_thread_id)

(experimental) By default, all buffered results are delivered on the same thread. Using *rocprofiler_create_callback_thread*, one or more buffers can be assigned to deliever their results on a unique, dedicated thread.

Parameters

- **buffer_id** – **[in]** Buffer identifier
- **cb_thread_id** – **[in]** Callback thread identifier via *rocprofiler_create_callback_thread*

Return values

- **ROCPROFILER_STATUS_SUCCESS** – Successful assignment of the delivery thread for the given buffer
- **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED** – Thread assignment is no longer available post-initialization
- **ROCPROFILER_STATUS_ERROR_THREAD_NOT_FOUND** – Thread identifier did not match

any of the threads created by rocprofiler
- **ROCPROFILER_STATUS_ERROR_BUFFER_NOT_FOUND** – Buffer identifier did not match any of the buffers registered with rocprofiler

Returns
rocprofiler_status_t

struct **rocprofiler_callback_thread_t**

#include <rocprofiler-sdk/internal_threading.h> (experimental) opaque handle to an internal thread identifier which delivers callbacks for buffers

See also

rocprofiler_create_callback_thread

19.1.13 OMPT Registration

rocprofiler_status_t **rocprofiler_ompt_is_initialized**(int *status)

(experimental) Query whether rocprofiler-sdk OMPT implementation has been initialized by OpenMP runtime.

Parameters
status – **[out]** Set to 0 if rocprofiler OMPT has not been initialized. Otherwise, set to 1.

Return values
ROCPROFILER_STATUS_SUCCESS – Always returns this value

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_ompt_is_finalized**(int *status)

(experimental) Query whether rocprofiler-sdk OMPT implementation has invoked ompt_finalize function.

Parameters
status – **[out]** Set to 0 if rocprofiler OMPT has not been finalized. Otherwise, set to 1.

Return values
ROCPROFILER_STATUS_SUCCESS – Always returns this value

Returns
rocprofiler_status_t

ompt_start_tool_result_t ***rocprofiler_ompt_start_tool**(unsigned int omp_version, const char *runtime_version)

(experimental) If a tool which contains a "ompt_start_tool" function which is invoked by the OpenMP runtime but the tool wishes to defer to rocprofiler-sdk to be the OMPT tool, it should invoke this function from it's `ompt_start_tool` implementation.

Parameters
- **omp_version** – **[in]** Refer to OpenMP OMPT docs for more information
- **runtime_version** – **[in]** Refer to OpenMP OMPT docs for more information

Returns
ompt_start_tool_result_t*

19.1.14 PC Sampling service

enum **rocprofiler_pc_sampling_configuration_flags_t**

(experimental) Enumeration describing values of flags of *rocprofiler_pc_sampling_configuration_t*.

Values:

enumerator **ROCPROFILER_PC_SAMPLING_CONFIGURATION_FLAGS_NONE**

enumerator **ROCPROFILER_PC_SAMPLING_CONFIGURATION_FLAGS_INTERVAL_POW2**

enum **rocprofiler_pc_sampling_instruction_type_t**

(experimental) Enumeration describing type of sampled issued instruction.

Values:

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_NONE**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_VALU**

vector ALU instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_MATRIX**

matrix instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_SCALAR**

scalar (memory) instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_TEX**

texture memory instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_LDS**

LDS memory instruction.

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_LDS_DIRECT**

LDS direct memory instruction.

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_FLAT**

flat memory instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_EXPORT**

export instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_MESSAGE**

message instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_BARRIER**

barrier instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_BRANCH_NOT_TAKEN**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_BRANCH_TAKEN**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_JUMP**
jump instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_OTHER**
other types of instruction

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_NO_INST**
no instruction issued

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_DUAL_VALU**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_TYPE_BRANCH_TAKEN**

enum **rocprofiler_pc_sampling_instruction_not_issued_reason_t**
(experimental) Enumeration describing reason for not issuing an instruction.
Values:

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_NONE**

enumerator
ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_NO_INSTRUCTION_AVAILABLE

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_ALU_DEPENDENCY**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_WAITCNT**
waitcnt dependency

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_INTERNAL_INSTRUCTION**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_BARRIER_WAIT**
waiting on a barrier

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_ARBITER_NOT_WIN**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_ARBITER_WIN_EX_STALL**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_OTHER_WAIT**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_SLEEP_WAIT**
wave was sleeping

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_ALU_DEPENDENCY**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_INTERNAL_INSTRUCTION**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_ARBITER_NOT_WIN**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_ARBITER_WIN_EX_STALL**

enumerator **ROCPROFILER_PC_SAMPLING_INSTRUCTION_NOT_ISSUED_REASON_OTHER_WAIT**

typedef *rocprofiler_status_t* (***rocprofiler_available_pc_sampling_configurations_cb_t**)(const *rocprofiler_pc_sampling_configuration_t* *configs, unsigned long num_config, void *user_data)

(experimental) Rocprofiler SDK's callback function to deliver the list of available PC sampling configurations upon the call to the *rocprofiler_query_pc_sampling_agent_configurations*.

Param configs
: **[out]** - The array of PC sampling configurations supported by the agent at the moment of invoking *rocprofiler_query_pc_sampling_agent_configurations*.

Param num_config
: **[out]** - The number of configurations contained in the underlying array `configs`. In case the GPU agent does not support PC sampling, the value is 0.

Param user_data
: **[in]** - client's private data passed via *rocprofiler_query_pc_sampling_agent_configurations*

Return
: *rocprofiler_status_t*

rocprofiler_status_t **rocprofiler_configure_pc_sampling_service**(*rocprofiler_context_id_t* context_id, *rocprofiler_agent_id_t* agent_id, *rocprofiler_pc_sampling_method_t* method, *rocprofiler_pc_sampling_unit_t* unit, uint64_t interval, *rocprofiler_buffer_id_t* buffer_id, int flags)

(experimental) Function used to configure the PC sampling service on the GPU agent with `agent_id`.

Prerequisites are the following:

- The client must create a context and supply its `context_id`. By using this context, the client can start/stop PC sampling on the agent. For more information, please

  See also

  rocprofiler_start_context/rocprofiler_stop_context.

- The user must create a buffer and supply its `buffer_id` . Rocprofiler-SDK uses the buffer to deliver the PC samples to the client. For more information about the data delivery, please

  Before calling this function, we recommend querying PC sampling configurations supported by the GPU agent via the

  Rocprofiler-SDK checks whether the requsted configuration is actually supported at the moment of calling this function. If the answer is yes, it returns the

  There are a few constraints a client's code needs to be aware of.

See also

rocprofiler_create_buffer and

See also

rocprofiler_buffer_tracing_cb_t.

See also

rocprofiler_query_pc_sampling_agent_configurations. The client chooses the `method`, `unit`, and `interval` to match one of the available configurations. Note that the `interval` must belong to the range of values [available_config.min_interval, available_config.max_interval], where available_config is the instance of the

See also

rocprofiler_pc_sampling_configuration_s supported/available at the moment.

See also

ROCPROFILER_STATUS_SUCCESS. Otherwise, it notifies the client about the rejection reason via the returned status code. For more information about the status codes, please

See also

rocprofiler_status_t.

Constraint1: A GPU agent can be configured to support at most one running PC sampling configuration at any time, which implies some of the consequences described below. After the tool configures the PC sampling with one of the available configurations, rocprofiler-SDK guarantees that this configuration will be valid for the tool's lifetime. The tool can start and stop the configured PC sampling service whenever convenient.

Constraint2: Since the same GPU agent can be used by multiple processes concurrently, Rocprofiler-SDK cannot guarantee the exclusive access to the PC sampling capability. The consequence is the following scenario. The tool TA that belongs to the process PA, calls the

Constraint3: Rocprofiler-SDK allows only one context to contain the configured PC sampling service within the process, that implies that at most one of the loaded tools can use PC sampling. One context can contains multiple PC sampling services configured for different GPU agents.

See also

rocprofiler_query_pc_sampling_agent_configurations that returns the two supported configurations CA and CB by the agent. Then the tool TB of the process PB, configures the PC sampling on the same agent by using

the configuration CB. Subsequently, the TA tries configuring the CA on the agent, and it fails. To point out that this case happened, we introduce a special status code

See also

ROCPROFILER_STATUS_ERROR_NOT_AVAILABLE. When this status code is observed by the tool TA, it queries all available configurations again by calling

See also

rocprofiler_query_pc_sampling_agent_configurations, that returns only CB this time. The tool TA can choose CB, so that both TA and TB use the PC sampling capability in the separate processes. Both TA and TB receives samples generated by the kernels launched by the corresponding processes PA and PB, respectively.

Constraint4: PC sampling feature is not available within the ROCgdb.

Constraint5: PC sampling service cannot be used simultaneously with counter collection service.

Parameters

- **context_id** – **[in]** - id of the context used for starting/stopping PC sampling service
- **agent_id** – **[in]** - id of the agent on which caller tries using PC sampling capability
- **method** – **[in]** - the type of PC sampling the caller tries to use on the agent.
- **unit** – **[in]** - The unit appropriate to the PC sampling type/method.
- **interval** – **[in]** - frequency at which PC samples are generated
- **buffer_id** – **[in]** - id of the buffer used for delivering PC samples
- **flags** – **[in]** - for future use

Return values

- **ROCPROFILER_STATUS_SUCCESS** – PC sampling service configured successfully
- **ROCPROFILER_STATUS_ERROR_NOT_AVAILABLE** – One of the scenarios is present:
  a. PC sampling is already configured with configuration different than requested,
  b. PC sampling is requested from a process that runs within the ROCgdb.
  c. HSA runtime does not support PC sampling.
  d. GPU device does not support requested PC sampling method.
- **ROCPROFILER_STATUS_ERROR_INCOMPATIBLE_KERNEL** – the amdgpu driver installed on the system does not support the PC sampling feature
- **ROCPROFILER_STATUS_ERROR** – a general error caused by the amdgpu driver
- **ROCPROFILER_STATUS_ERROR_CONTEXT_CONFLICT** – counter collection service already setup in the context
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – function invoked with an invalid argument

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_query_pc_sampling_agent_configurations**(*rocprofiler_agent_id_t* agent_id, *rocprofiler_available_pc_sampling_configurations_cb_* cb, void *user_data)

(experimental) Query PC Sampling Configuration.

Lists PC sampling configurations a GPU agent with `agent_id` supports at the moment of invoking the function. Delivers configurations via `cb`. In case the PC sampling is configured on the GPU agent, the `cb` delivers

information about the active PC sampling configuration. In case the GPU agent does not support PC sampling capability, the cb delivers none PC sampling configurations.

Parameters

- **agent_id** – **[in]** - id of the agent for which available configurations will be listed
- **cb** – **[in]** - User callback that delivers the available PC sampling configurations
- **user_data** – **[in]** - passed to the cb

Return values

- **ROCPROFILER_STATUS_ERROR_NOT_AVAILABLE** – One of the scenarios is present:
  a. PC sampling is requested from a process that runs within the ROCgdb.
  b. HSA runtime does not support PC sampling.
- **ROCPROFILER_STATUS_ERROR_INCOMPATIBLE_KERNEL** – the amdgpu driver installed on the system does not support the PC sampling feature.
- **ROCPROFILER_STATUS_ERROR** – a general error caused by the amdgpu driver
- **ROCPROFILER_STATUS_SUCCESS** – cb successfully finished

Returns

rocprofiler_status_t

const char *rocprofiler_get_pc_sampling_instruction_type_name(*rocprofiler_pc_sampling_instruction_type_t* instruction_type)

(experimental) Return the string encoding of *rocprofiler_pc_sampling_instruction_type_t* value

Parameters

instruction_type – **[in]** instruction type enum value

Returns

Will return a nullptr if invalid/unsupported *rocprofiler_pc_sampling_instruction_type_t* value is provided.

const char *rocprofiler_get_pc_sampling_instruction_not_issued_reason_name(*rocprofiler_pc_sampling_instruction_not* not_issued_reason)

(experimental) Return the string encoding of *rocprofiler_pc_sampling_instruction_not_issued_reason_t* value

Parameters

not_issued_reason – **[in]** no issue reason enum value

Returns

Will return a nullptr if invalid/unsupported *rocprofiler_pc_sampling_instruction_not_issued_reason_t* value is provided.

struct **rocprofiler_pc_sampling_configuration_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) PC sampling configuration supported by a GPU agent.

struct **rocprofiler_pc_sampling_hw_id_v0_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) Information about the GPU part where wave was executing at the moment of sampling.

struct **rocprofiler_pc_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) Sampled program counter.

struct **rocprofiler_pc_sampling_record_host_trap_v0_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) ROCProfiler Host-Trap PC Sampling Record.

struct **rocprofiler_pc_sampling_record_stochastic_header_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) The header of the *rocprofiler_pc_sampling_record_stochastic_v0_t*, indicating what fields of the *rocprofiler_pc_sampling_record_stochastic_v0_t* instance are meaningful for the sample.

struct **rocprofiler_pc_sampling_snapshot_v0_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) Data provided by stochastic sampling hardware.

struct **rocprofiler_pc_sampling_memory_counters_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) Counters of issued but not yet completed instructions.

struct **rocprofiler_pc_sampling_record_stochastic_v0_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) ROCProfiler Stochastic PC Sampling Record.

struct **rocprofiler_pc_sampling_record_invalid_t**

#include <rocprofiler-sdk/pc_sampling.h> (experimental) Record representing an invalid PC Sampling Record.

19.1.15 Thread trace

enum **rocprofiler_thread_trace_parameter_type_t**

Types of Thread Trace parameters.

Values:

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_TARGET_CU**

Select the Target CU or WGP.

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_SHADER_ENGINE_MASK**

Bitmask of shader engines.

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_BUFFER_SIZE**

Size of combined GPU buffer for ATT.

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_SIMD_SELECT**

Bitmask (GFX9) or ID (Navi) of SIMDs.

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_PERFCOUNTERS_CTRL**

Period [1,32] or disable (0) perfmon.

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_PERFCOUNTER**

Perfmon ID and SIMD mask.

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_SERIALIZE_ALL**

Serializes kernels not under thread trace.

enumerator **ROCPROFILER_THREAD_TRACE_PARAMETER_LAST**

enum **rocprofiler_thread_trace_control_flags_t**

Values:

enumerator **ROCPROFILER_THREAD_TRACE_CONTROL_NONE**

enumerator **ROCPROFILER_THREAD_TRACE_CONTROL_START_AND_STOP**

enum **rocprofiler_thread_trace_decoder_info_t**

Describes the type of info received.

Values:

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INFO_NONE**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INFO_DATA_LOST**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INFO_STITCH_INCOMPLETE**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INFO_LAST**

enum **rocprofiler_thread_trace_decoder_wstate_type_t**

Wave state type.

Values:

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_WSTATE_EMPTY**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_WSTATE_IDLE**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_WSTATE_EXEC**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_WSTATE_WAIT**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_WSTATE_STALL**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_WSTATE_LAST**

enum **rocprofiler_thread_trace_decoder_inst_category_t**

Instruction type.

Values:

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_NONE**

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_SMEM**

Scalar memory op.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_SALU**
Scalar ALU op.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_VMEM**
Vector memory op.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_FLAT**
Flat addressing vmem or lds.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_LDS**
Local Data Share op.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_VALU**
Vector ALU op.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_JUMP**
Branch taken.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_NEXT**
Branch not taken.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_IMMED**
Internal operation.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_CONTEXT**
Wave context switch.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_MESSAGE**
MSG types.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_BVH**
Raytrace op.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_INST_LAST**

enum **rocprofiler_thread_trace_decoder_record_type_t**
Defines the type of payload received by rocprofiler_thread_trace_decoder_callback_t.
Values:

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_RECORD_GFXIP**
Record is gfxip_major, type size_t.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_RECORD_OCCUPANCY**
rocprofiler_thread_trace_decoder_occupancy_t*

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_RECORD_PERFEVENT**

rocprofiler_thread_trace_decoder_perfevent_t*

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_RECORD_WAVE**

rocprofiler_thread_trace_decoder_wave_t*

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_RECORD_INFO**

rocprofiler_thread_trace_decoder_info_t*

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_RECORD_DEBUG**

Debug.

enumerator **ROCPROFILER_THREAD_TRACE_DECODER_RECORD_LAST**

typedef void (***rocprofiler_thread_trace_shader_data_callback_t**)(*rocprofiler_agent_id_t* agent, int64_t shader_engine_id, void *data, unsigned long data_size, *rocprofiler_user_data_t* userdata)

Callback to be triggered every time some ATT data is generated by the device.

See also

rocprofiler_agent_id_t)

Param agent
[in] Identifier for the target agent (
Param shader_engine_id
[in] ID of shader engine, as enabled by SE_MASK
Param data
[in] Pointer to the buffer containing the ATT data
Param data_size
[in] Number of bytes in "data"
Param userdata
[in] Passed back to user from *rocprofiler_thread_trace_dispatch_callback_t()*

typedef *rocprofiler_thread_trace_control_flags_t* (***rocprofiler_thread_trace_dispatch_callback_t**)(*rocprofiler_agent_id_t* agent_id, *rocprofiler_queue_id_t* queue_id, *rocprofiler_async_correlation_id_t* correlation_id, *rocprofiler_kernel_id_t* kernel_id, *rocprofiler_dispatch_id_t* dispatch_id, void *userdata_config, *rocprofiler_user_data_t* *userdata_shader)

Callback to be triggered every kernel dispatch, indicating to start and/or stop ATT.
Param agent_id
[in] agent_id.
Param queue_id
[in] queue_id.
Param correlation_id
[in] internal correlation id.
Param kernel_id
[in] kernel_id.
Param dispatch_id
[in] dispatch_id.

Param userdata_config
[in] Userdata passed back from rocprofiler_configure_dispatch_thread_trace_service.
Param userdata_shader
[out] Userdata to be passed in shader_callback

typedef void (*`rocprofiler_thread_trace_decoder_callback_t`)(*rocprofiler_thread_trace_decoder_record_type_t* record_type_id, void *trace_events, uint64_t trace_size, void *userdata)

Callback for rocprof-trace-decoder to return decoder traces back to user.
Param record_type_id
[in] One of *rocprofiler_thread_trace_decoder_record_type_t*
Param trace_events
[in] A pointer to sequence of events, of size trace_size.
Param trace_size
[in] The number of events in the trace.
Param userdata
[in] Arbitrary data pointer to be sent back to the user via callback.

rocprofiler_status_t `rocprofiler_configure_device_thread_trace_service`(*rocprofiler_context_id_t* context_id, *rocprofiler_agent_id_t* agent_id, *rocprofiler_thread_trace_parameter_t* *parameters, unsigned long num_parameters, *rocprofiler_thread_trace_shader_data_callback_t* shader_callback, *rocprofiler_user_data_t* callback_userdata)

Configure Thread Trace Service for agent. There may only be one agent profile configured per context and can be only one active context that is profiling a single agent at a time. Multiple agent contexts can be started at the same time if they are profiling different agents.
Parameters
- `context_id` – **[in]** context id
- `parameters` – **[in]** List of ATT-specific parameters.
- `num_parameters` – **[in]** Number of parameters. Zero is allowed.
- `agent_id` – **[in]** agent to configure profiling on.
- `shader_callback` – **[in]** Callback fn where the collected data will be sent to.
- `callback_userdata` – **[in]** Passed back to user in shader_callback.

Return values
- `ROCPROFILER_STATUS_SUCCESS` – on success
- `ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED` – for configuration locked
- `ROCPROFILER_STATUS_ERROR_CONTEXT_INVALID` – for conflicting configurations in the same ctx
- `ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND` – for invalid context id
- `ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT` – for invalid *rocprofiler_thread_trace_parameter_t*

Returns
rocprofiler_status_t

rocprofiler_status_t **rocprofiler_configure_dispatch_thread_trace_service**(*rocprofiler_context_id_t* context_id, *rocprofiler_agent_id_t* agent_id, *rocprofiler_thread_trace_parameter_t* *parameters, unsigned long num_parameters, *rocprofiler_thread_trace_dispatch_callback_t* dispatch_callback, *rocprofiler_thread_trace_shader_data_callback_t* shader_callback, void *callback_userdata)

Enables the thread trace service for dispatch-based tracing. The tool has an option to enable/disable thread trace on every dispatch callback. This service serializes all traced kernels, and optionally all non-traced kernels.

Parameters

- **context_id** – **[in]** id of the context used for start/stop thread_trace.
- **agent_id** – **[in]** *rocprofiler_agent_id_t* to configure thread trace.
- **parameters** – **[in]** List of ATT-specific parameters.
- **num_parameters** – **[in]** Number of parameters. Zero is allowed.
- **dispatch_callback** – **[in]** Control fn which decides when TT starts/stop collecting.
- **shader_callback** – **[in]** Callback fn where the collected data will be sent to.
- **callback_userdata** – **[in]** Passed back to user in dispatch_callback.

Return values

- **ROCPROFILER_STATUS_SUCCESS** – on success
- **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED** – for configuration locked
- **ROCPROFILER_STATUS_ERROR_CONTEXT_INVALID** – for conflicting configurations in the same ctx
- **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND** – for invalid context id
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – for invalid *rocprofiler_thread_trace_parameter_t*
- **ROCPROFILER_STATUS_ERROR_SERVICE_ALREADY_CONFIGURED** – if already configured

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_thread_trace_decoder_create**(*rocprofiler_thread_trace_decoder_handle_t* *handle, const char *path)

Initializes Trace Decoder library with a library search path.

Parameters

- **handle** – **[out]** Handle to created decoder instance.
- **path** – **[in]** Path to trace decoder library location (e.g. /opt/rocm/lib).

Return values

- **ROCPROFILER_STATUS_ERROR_NOT_AVAILABLE** – Library not found
- **ROCPROFILER_STATUS_ERROR_INCOMPATIBLE_ABI** – Library found but version not supported
- **ROCPROFILER_STATUS_SUCCESS** – Handle created

Returns

rocprofiler_status_t

void **rocprofiler_thread_trace_decoder_destroy**(*rocprofiler_thread_trace_decoder_handle_t* handle)

Deletes handle created by *rocprofiler_thread_trace_decoder_create*.

Parameters

handle – **[in]** Handle to destroy

rocprofiler_status_t **rocprofiler_thread_trace_decoder_codeobj_load**(*rocprofiler_thread_trace_decoder_handle_t* handle, uint64_t load_id, uint64_t load_addr, uint64_t load_size, const void *data, uint64_t size)

Loads a code object binary to match with Thread Trace. The size, data and load_* are reported by rocprofiler-sdk's code object tracing service. Used for the decoder library to know what code objects to look into when decoding shader data. Not all application code objects are required to be reported here, only the ones containing code executed at the time the shader data was collected by thread_trace services. If a code object not reported here is encountered while decoding shader data, a record of type INFO_STITCH_INCOMPLETE will be generated and instructions will not be reported with a PC address.

Parameters

- **handle** – **[in]** Handle to decoder instance.
- **load_id** – **[in]** Code object load ID.
- **load_addr** – **[in]** Code object load address.
- **load_size** – **[in]** Code object load size.
- **data** – **[in]** Code object binary data.
- **size** – **[in]** Code object binary data size.

Return values

- **ROCPROFILER_STATUS_ERROR** – Unable to load code object.
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – Invalid handle
- **ROCPROFILER_STATUS_SUCCESS** – Code object loaded

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_thread_trace_decoder_codeobj_unload**(*rocprofiler_thread_trace_decoder_handle_t* handle, uint64_t load_id)

Unloads a code object binary.

Parameters

- **handle** – **[in]** Handle to decoder instance.
- **load_id** – **[in]** Code object load ID to remove.

Return values

- **ROCPROFILER_STATUS_ERROR** – Code object not loaded.
- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – Invalid handle
- **ROCPROFILER_STATUS_SUCCESS** – Code object unloaded

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_trace_decode**(*rocprofiler_thread_trace_decoder_handle_t* handle, *rocprofiler_thread_trace_decoder_callback_t* callback, void *data, uint64_t size, void *userdata)

Decodes shader data returned by *rocprofiler_thread_trace_shader_data_callback_t*. Use *rocprofiler_thread_trace_decoder_codeobj_load* to add references to loaded code objects during the trace. A *rocprofiler_thread_trace_decoder_callback_t* returns decoded data back to user. The first record is always of type *ROCPROFILER_THREAD_TRACE_DECODER_RECORD_GFXIP*.

Parameters

- **handle** – **[in]** Decoder handle
- **callback** – **[in]** Decoded trace data returned to user.
- **data** – **[in]** Thread trace binary data.
- **size** – **[in]** Thread trace binary size.
- **userdata** – **[in]** Userdata passed back to caller via callback.

Return values

- **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT** – invalid argument
- **ROCPROFILER_STATUS_ERROR_AGENT_ARCH_NOT_SUPPORTED** – arch not supported
- **ROCPROFILER_STATUS_ERROR** – generic error

- **ROCPROFILER_STATUS_SUCCESS** – on success

Returns

rocprofiler_status_t

const char ***rocprofiler_thread_trace_decoder_info_string**(*rocprofiler_thread_trace_decoder_handle_t* handle, *rocprofiler_thread_trace_decoder_info_t* info)

Returns the string description of a *rocprofiler_thread_trace_decoder_info_t* record.

Parameters

- **handle** – **[in]** Decoder handle
- **info** – **[in]** The decoder info received

Return values

null – terminated string as description of "info".

struct **rocprofiler_thread_trace_parameter_t**

#include <rocprofiler-sdk/experimental/thread-trace/core.h> Thread Trace parameter specification.

struct **rocprofiler_thread_trace_decoder_handle_t**

#include <rocprofiler-sdk/experimental/thread-trace/trace_decoder.h> Handle containing a loaded rocprof-trace-decoder and a decoder state.

struct **rocprofiler_thread_trace_decoder_pc_t**

#include <rocprofiler-sdk/experimental/thread-trace/trace_decoder_types.h> Describes a PC address.

struct **rocprofiler_thread_trace_decoder_perfevent_t**

#include <rocprofiler-sdk/experimental/thread-trace/trace_decoder_types.h> Describes four performance counter values.

struct **rocprofiler_thread_trace_decoder_occupancy_t**

#include <rocprofiler-sdk/experimental/thread-trace/trace_decoder_types.h> Describes an occupancy event (wave started or wave ended).

struct **rocprofiler_thread_trace_decoder_wave_state_t**

#include <rocprofiler-sdk/experimental/thread-trace/trace_decoder_types.h> A wave state change event.

struct **rocprofiler_thread_trace_decoder_inst_t**

#include <rocprofiler-sdk/experimental/thread-trace/trace_decoder_types.h> Describes an instruction execution event.

The duration is measured as stall+issue time (gfx9) or stall+execution time (gfx10+). Time + duration marks the issue (gfx9) or execution (gfx10+) completion time. Time + stall marks the successful issue time. Duration - stall is the issue time (gfx9) or execution time (gfx10+).

struct **rocprofiler_thread_trace_decoder_wave_t**

#include <rocprofiler-sdk/experimental/thread-trace/trace_decoder_types.h> Struct describing a wave during it's lifetime. This record is only generated for waves executing in the target_cu and target_simd, selected by ROCPROFILER_THREAD_TRACE_PARAMETER_TARGET_CU and ROCPROFILER_THREAD_TRACE_PARAMETER_SIMD_SELECT.

instructions_array contains a time-ordered list of all (traced) instructions by the wave.

rocprofiler_thread_trace_parameter_t.__unnamed12__

Public Members

uint64_t **value**

struct rocprofiler_thread_trace_parameter_t

rocprofiler_thread_trace_parameter_t.__unnamed12__.__unnamed14__

19.1.16 Tool registration

typedef void (***rocprofiler_client_finalize_t**)(*rocprofiler_client_id_t*)

Prototype for the function pointer provided to tool in *rocprofiler_tool_initialize_t*. This function can be used to explicitly invoke the client tools *rocprofiler_tool_finalize_t* finalization function prior to the `atexit` handler which calls it.

If this function pointer is invoked explicitly, rocprofiler will disable calling the *rocprofiler_tool_finalize_t* functioin pointer during it's `atexit` handler.

typedef int (***rocprofiler_tool_initialize_t**)(*rocprofiler_client_finalize_t* finalize_func, void *tool_data)

Prototype for the initialize function where a tool creates contexts for the set of rocprofiler services used by the tool.

Param finalize_func
[in] Function pointer to explicitly invoke the finalize function for the client

Param tool_data
[in] `tool_data` field returned from *rocprofiler_configure* in *rocprofiler_tool_configure_result_t*.

typedef void (***rocprofiler_tool_finalize_t**)(void *tool_data)

Prototype for the finalize function where a tool does any cleanup or output operations once it has finished using rocprofiler services.

Param tool_data
[in] `tool_data` field returned from *rocprofiler_configure* in *rocprofiler_tool_configure_result_t*.

typedef *rocprofiler_tool_configure_result_t* *(***rocprofiler_configure_func_t**)(uint32_t version, const char *runtime_version, uint32_t priority, *rocprofiler_client_id_t* *client_id)

Function pointer typedef for *rocprofiler_configure* function.

Param version
[in] The version of rocprofiler: `(10000 * major) + (100 * minor) + patch`

Param runtime_version
[in] String descriptor of the rocprofiler version and other relevant info.

Param priority
[in] How many client tools were initialized before this client tool

Param client_id
[inout] tool identifier value.

rocprofiler_status_t **rocprofiler_is_initialized**(int *status)

Query whether rocprofiler has already scanned the binary for all the instances of *rocprofiler_configure* (or is currently scanning). If rocprofiler has completed it's scan, clients can directly register themselves with rocprofiler.

Parameters

status – **[out]** 0 indicates rocprofiler has not been initialized (i.e. configured), 1 indicates rocprofiler has been initialized, -1 indicates rocprofiler is currently initializing.

Return values

ROCPROFILER_STATUS_SUCCESS – Returned unconditionally

Returns

rocprofiler_status_t

rocprofiler_status_t **rocprofiler_is_finalized**(int *status)

Query rocprofiler finalization status.

Parameters

status – **[out]** 0 indicates rocprofiler has not been finalized, 1 indicates rocprofiler has been finalized, -1 indicates rocprofiler is currently finalizing.

Return values

ROCPROFILER_STATUS_SUCCESS – Returned unconditionally

Returns

rocprofiler_status_t

rocprofiler_tool_configure_result_t ***rocprofiler_configure**(uint32_t version, const char *runtime_version, uint32_t priority, *rocprofiler_client_id_t* *client_id)

This is the special function that tools define to enable rocprofiler support. The tool should return a pointer to *rocprofiler_tool_configure_result_t* which will contain a function pointer to (1) an initialization function where all the contexts are created, (2) a finalization function (if necessary) which will be invoked when rocprofiler shutdown and, (3) a pointer to any data that the tool wants communicated between the rocprofiler_tool_configure_result_t::initialize and rocprofiler_tool_configure_result_t::finalize functions. If the user.

```
#include <rocprofiler-sdk/registration.h>

static rocprofiler_client_id_t       my_client_id;
static rocprofiler_client_finalize_t my_fini_func;
static int                           my_tool_data = 1234;

static int my_init_func(rocprofiler_client_finalize_t fini_func,
                        void* tool_data)
{
     my_fini_func = fini_func;

     assert(*static_cast<int*>(tool_data) == 1234 && "tool_data is wrong");

     rocprofiler_context_id_t ctx{0};
     rocprofiler_create_context(&ctx);

     if(int valid_ctx = 0;
        rocprofiler_context_is_valid(ctx, &valid_ctx) != ROCPROFILER_STATUS_SUCCESS␣
↪||
        valid_ctx != 0)
     {
         // notify rocprofiler that initialization failed
```

(continues on next page)

(continued from previous page)

```
        // and all the contexts, buffers, etc. created
        // should be ignored
        return -1;
    }

    if(rocprofiler_start_context(ctx) != ROCPROFILER_STATUS_SUCCESS)
    {
        // notify rocprofiler that initialization failed
        // and all the contexts, buffers, etc. created
        // should be ignored
        return -1;
    }

    // no errors
    return 0;
}

static int my_fini_func(void* tool_data)
{
    assert(*static_cast<int*>(tool_data) == 1234 && "tool_data is wrong");
}

rocprofiler_tool_configure_result_t*
rocprofiler_configure(uint32_t version,
                      const char* runtime_version,
                      uint32_t priority,
                      rocprofiler_client_id_t* client_id)
{
    // only activate if main tool
    if(priority > 0) return nullptr;

    // set the client name
    client_id->name = "ExampleTool";

    // make a copy of client info
    my_client_id = *client_id;

    // compute major/minor/patch version info
    uint32_t major = version / 10000;
    uint32_t minor = (version % 10000) / 100;
    uint32_t patch = version % 100;

    // print info
    printf("Configuring %s with rocprofiler-sdk (v%u.%u.%u) [%s]\n",
           client_id->name, major, minor, patch, runtime_version);

    // create configure data
    static auto cfg = rocprofiler_tool_configure_result_t{ &my_init_func,
                                                           &my_fini_func,
                                                           &my_tool_data };

    // return pointer to configure data
```

(continues on next page)

(continued from previous page)

```
    return &cfg;
}
```

Parameters

- **version** – **[in]** The version of rocprofiler: `(10000 * major) + (100 * minor) + patch`
- **runtime_version** – **[in]** String descriptor of the rocprofiler version and other relevant info.
- **priority** – **[in]** How many client tools were initialized before this client tool
- **client_id** – **[inout]** tool identifier value.

Returns

rocprofiler_tool_configure_result_t*

rocprofiler_status_t **rocprofiler_force_configure**(*rocprofiler_configure_func_t* configure_func)

Function for explicitly registering a configuration with rocprofiler. This can be invoked before any ROCm runtimes (lazily) initialize and context(s) can be started before the runtimes initialize.

Parameters

configure_func – **[in]** Address of *rocprofiler_configure* function. A null pointer is acceptable if the address is not known

Return values

- **ROCPROFILER_STATUS_SUCCESS** – Registration was successfully triggered.
- **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED** – Returned if rocprofiler has already been configured, or is currently being configured

Returns

rocprofiler_status_t

struct **rocprofiler_client_id_t**

#include <rocprofiler-sdk/registration.h> (experimental) A client refers to an individual or entity engaged in the configuration of ROCprofiler services. e.g: any third party tool like PAPI or any internal tool (Omnitrace). A pointer to this data structure is provided to the client tool initialization function. The name member can be set by the client to assist with debugging (e.g. rocprofiler cannot start your context because there is a conflicting context started by <name> — at least that is the plan). The handle member is a unique identifer assigned by rocprofiler for the client and the client can store it and pass it to the *rocprofiler_client_finalize_t* function to force finalization (i.e. deactivate all of it's contexts) for the client.

struct **rocprofiler_tool_configure_result_t**

#include <rocprofiler-sdk/registration.h> Data structure containing a initialization, finalization, and data.

After rocprofiler has retrieved all instances of *rocprofiler_tool_configure_result_t* from the tools — via either *rocprofiler_configure* and/or *rocprofiler_force_configure*, rocprofiler will invoke all non-null `initialize` functions and provide the user a function pointer which will explicitly invoke the `finalize` function pointer. Both the `initialize` and `finalize` functions will be passed the value of the `tool_data` field. The `size` field is used for ABI reasons, in the event that rocprofiler changes the *rocprofiler_tool_configure_result_t* struct and it should be set to `sizeof(rocprofiler_tool_configure_result_t)`

19.2 Global Data structures, topics, files

This ROCprofiler-SDK API topic covers:

- *Global Basic Data Types*
- *Topics*

- *Data Structures*
- *File List*

19.2.1 Global Basic Data Types

enum **rocprofiler_status_t**

Status codes.

Values:

enumerator **ROCPROFILER_STATUS_SUCCESS**

No error occurred.

enumerator **ROCPROFILER_STATUS_ERROR**

Generalized error.

enumerator **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_FOUND**

No valid context for given context id.

enumerator **ROCPROFILER_STATUS_ERROR_BUFFER_NOT_FOUND**

No valid buffer for given buffer id.

enumerator **ROCPROFILER_STATUS_ERROR_KIND_NOT_FOUND**

Kind identifier is invalid.

enumerator **ROCPROFILER_STATUS_ERROR_OPERATION_NOT_FOUND**

Operation identifier is invalid for domain.

enumerator **ROCPROFILER_STATUS_ERROR_THREAD_NOT_FOUND**

No valid thread for given thread id.

enumerator **ROCPROFILER_STATUS_ERROR_AGENT_NOT_FOUND**

Agent identifier not found.

enumerator **ROCPROFILER_STATUS_ERROR_COUNTER_NOT_FOUND**

Counter identifier does not exist.

enumerator **ROCPROFILER_STATUS_ERROR_CONTEXT_ERROR**

Generalized context error.

enumerator **ROCPROFILER_STATUS_ERROR_CONTEXT_INVALID**

Context configuration is not valid.

enumerator **ROCPROFILER_STATUS_ERROR_CONTEXT_NOT_STARTED**

Context was not started (e.g., atomic swap into active array failed)

enumerator **ROCPROFILER_STATUS_ERROR_CONTEXT_CONFLICT**

Context operation failed due to a conflict with another context.

enumerator **ROCPROFILER_STATUS_ERROR_CONTEXT_ID_NOT_ZERO**

Context ID is not initialized to zero.

enumerator **ROCPROFILER_STATUS_ERROR_BUFFER_BUSY**

buffer operation failed because it currently busy handling another request (e.g. flushing)

enumerator **ROCPROFILER_STATUS_ERROR_SERVICE_ALREADY_CONFIGURED**

service has already been configured in context

enumerator **ROCPROFILER_STATUS_ERROR_CONFIGURATION_LOCKED**

Function call is not valid outside of rocprofiler configuration (i.e. function called post-initialization)

enumerator **ROCPROFILER_STATUS_ERROR_NOT_IMPLEMENTED**

Function is not implemented.

enumerator **ROCPROFILER_STATUS_ERROR_INCOMPATIBLE_ABI**

Data structure provided by user is incompatible with current version of rocprofiler.

enumerator **ROCPROFILER_STATUS_ERROR_INVALID_ARGUMENT**

Function invoked with one or more invalid arguments.

enumerator **ROCPROFILER_STATUS_ERROR_METRIC_NOT_VALID_FOR_AGENT**

Invalid metric supplied to agent.

enumerator **ROCPROFILER_STATUS_ERROR_FINALIZED**

invalid because rocprofiler has been finalized

enumerator **ROCPROFILER_STATUS_ERROR_HSA_NOT_LOADED**

Call requires HSA to be loaded before performed.

enumerator **ROCPROFILER_STATUS_ERROR_DIM_NOT_FOUND**

Dimension is not found for counter.

enumerator **ROCPROFILER_STATUS_ERROR_PROFILE_COUNTER_NOT_FOUND**

Profile could not find counter for GPU agent.

enumerator **ROCPROFILER_STATUS_ERROR_AST_GENERATION_FAILED**

AST could not be generated correctly.

enumerator **ROCPROFILER_STATUS_ERROR_AST_NOT_FOUND**

AST was not found.

enumerator **ROCPROFILER_STATUS_ERROR_AQL_NO_EVENT_COORD**

Event coordinate was not found by AQL profile.

enumerator **ROCPROFILER_STATUS_ERROR_INCOMPATIBLE_KERNEL**

A service depends on a newer version of KFD (amdgpu kernel driver). Check logs for service that report incompatibility.

enumerator **ROCPROFILER_STATUS_ERROR_OUT_OF_RESOURCES**

The given resources are insufficient to complete operation.

enumerator **ROCPROFILER_STATUS_ERROR_PROFILE_NOT_FOUND**

Could not find the counter profile.

enumerator **ROCPROFILER_STATUS_ERROR_AGENT_DISPATCH_CONFLICT**

Cannot enable both agent and dispatch counting in the same context.

enumerator **ROCPROFILER_STATUS_INTERNAL_NO_AGENT_CONTEXT**

No agent context found, may not be an error.

enumerator **ROCPROFILER_STATUS_ERROR_SAMPLE_RATE_EXCEEDED**

Sample rate exceeded.

enumerator **ROCPROFILER_STATUS_ERROR_NO_PROFILE_QUEUE**

Profile queue creation failed.

enumerator **ROCPROFILER_STATUS_ERROR_NO_HARDWARE_COUNTERS**

No hardware counters were specified.

enumerator **ROCPROFILER_STATUS_ERROR_AGENT_MISMATCH**

Agent mismatch between profile and context.

enumerator **ROCPROFILER_STATUS_ERROR_NOT_AVAILABLE**

The service is not available. Please refer to API functions that return this status code for more information.

enumerator **ROCPROFILER_STATUS_ERROR_EXCEEDS_HW_LIMIT**

Exceeds hardware limits for collection.

enumerator **ROCPROFILER_STATUS_ERROR_AGENT_ARCH_NOT_SUPPORTED**

Agent HW architecture not supported.

enumerator **ROCPROFILER_STATUS_ERROR_PERMISSION_DENIED**

Permission denied.

enumerator **ROCPROFILER_STATUS_LAST**

enum **rocprofiler_buffer_category_t**

Buffer record categories. This enumeration type is encoded in *rocprofiler_record_header_t* category field.

Values:

enumerator **ROCPROFILER_BUFFER_CATEGORY_NONE**

enumerator **ROCPROFILER_BUFFER_CATEGORY_TRACING**

enumerator **ROCPROFILER_BUFFER_CATEGORY_PC_SAMPLING**

enumerator **ROCPROFILER_BUFFER_CATEGORY_COUNTERS**

enumerator **ROCPROFILER_BUFFER_CATEGORY_LAST**

enum **rocprofiler_agent_type_t**

Agent type.

Values:

enumerator **ROCPROFILER_AGENT_TYPE_NONE**

Agent type is unknown.

enumerator **ROCPROFILER_AGENT_TYPE_CPU**

Agent type is a CPU.

enumerator **ROCPROFILER_AGENT_TYPE_GPU**

Agent type is a GPU.

enumerator **ROCPROFILER_AGENT_TYPE_LAST**

enum **rocprofiler_callback_phase_t**

Service Callback Phase.

Values:

enumerator **ROCPROFILER_CALLBACK_PHASE_NONE**

Callback has no phase.

enumerator **ROCPROFILER_CALLBACK_PHASE_ENTER**

Callback invoked prior to function execution.

enumerator **ROCPROFILER_CALLBACK_PHASE_LOAD**

Callback invoked prior to code object loading.

enumerator **ROCPROFILER_CALLBACK_PHASE_EXIT**

Callback invoked after to function execution.

enumerator **ROCPROFILER_CALLBACK_PHASE_UNLOAD**

Callback invoked prior to code object unloading.

enumerator **ROCPROFILER_CALLBACK_PHASE_LAST**

enum **rocprofiler_callback_tracing_kind_t**

Service Callback Tracing Kind.

See also

rocprofiler_configure_callback_tracing_service.

Values:

enumerator **ROCPROFILER_CALLBACK_TRACING_NONE**

enumerator **ROCPROFILER_CALLBACK_TRACING_HSA_CORE_API**

See also

rocprofiler_hsa_core_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_HSA_AMD_EXT_API**

See also

rocprofiler_hsa_amd_ext_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_HSA_IMAGE_EXT_API**

See also

rocprofiler_hsa_image_ext_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_HSA_FINALIZE_EXT_API**

See also

rocprofiler_hsa_finalize_ext_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_HIP_RUNTIME_API**

> **See also**
>
> rocprofiler_hip_runtime_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_HIP_COMPILER_API**

> **See also**
>
> rocprofiler_hip_compiler_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_MARKER_CORE_API**

> **See also**
>
> rocprofiler_marker_core_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_MARKER_CONTROL_API**

> **See also**
>
> rocprofiler_marker_control_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_MARKER_NAME_API**

> **See also**
>
> rocprofiler_marker_name_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT**

> **See also**
>
> *rocprofiler_code_object_operation_t*

enumerator **ROCPROFILER_CALLBACK_TRACING_SCRATCH_MEMORY**

See also

rocprofiler_scratch_memory_operation_t

enumerator **ROCPROFILER_CALLBACK_TRACING_KERNEL_DISPATCH**

Callbacks for kernel dispatches.

enumerator **ROCPROFILER_CALLBACK_TRACING_MEMORY_COPY**

See also

rocprofiler_memory_copy_operation_t

enumerator **ROCPROFILER_CALLBACK_TRACING_RCCL_API**

RCCL tracing.

enumerator **ROCPROFILER_CALLBACK_TRACING_OMPT**

See also

rocprofiler_ompt_operation_t

enumerator **ROCPROFILER_CALLBACK_TRACING_MEMORY_ALLOCATION**

See also

rocprofiler_memory_allocation_operation_t

enumerator **ROCPROFILER_CALLBACK_TRACING_RUNTIME_INITIALIZATION**

Callback notifying that a runtime library has been initialized.

enumerator **ROCPROFILER_CALLBACK_TRACING_ROCDECODE_API**

rocDecode API Tracing

enumerator **ROCPROFILER_CALLBACK_TRACING_ROCJPEG_API**

rocJPEG API Tracing

enumerator **ROCPROFILER_CALLBACK_TRACING_HIP_STREAM**

See also

rocprofiler_hip_stream_operation_t

enumerator **ROCPROFILER_CALLBACK_TRACING_MARKER_CORE_RANGE_API**

> **See also**
>
> rocprofiler_marker_core_range_api_id_t

enumerator **ROCPROFILER_CALLBACK_TRACING_LAST**

enum **rocprofiler_buffer_tracing_kind_t**

Service Buffer Tracing Kind.

> **See also**
>
> *rocprofiler_configure_buffer_tracing_service*.

Values:

enumerator **ROCPROFILER_BUFFER_TRACING_NONE**

enumerator **ROCPROFILER_BUFFER_TRACING_HSA_CORE_API**

> **See also**
>
> rocprofiler_hsa_core_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_HSA_AMD_EXT_API**

> **See also**
>
> rocprofiler_hsa_amd_ext_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_HSA_IMAGE_EXT_API**

> **See also**
>
> rocprofiler_hsa_image_ext_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_HSA_FINALIZE_EXT_API**

See also

rocprofiler_hsa_finalize_ext_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_HIP_RUNTIME_API**

See also

rocprofiler_hip_runtime_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_HIP_COMPILER_API**

See also

rocprofiler_hip_compiler_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_MARKER_CORE_API**

See also

rocprofiler_marker_core_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_MARKER_CONTROL_API**

See also

rocprofiler_marker_control_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_MARKER_NAME_API**

See also

rocprofiler_marker_name_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_MEMORY_COPY**

See also

rocprofiler_memory_copy_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KERNEL_DISPATCH**

Buffer kernel dispatch info.

enumerator **ROCPROFILER_BUFFER_TRACING_SCRATCH_MEMORY**

Buffer scratch memory reclaimation info.

enumerator **ROCPROFILER_BUFFER_TRACING_CORRELATION_ID_RETIREMENT**

Correlation ID in no longer in use.

enumerator **ROCPROFILER_BUFFER_TRACING_RCCL_API**

RCCL tracing.

enumerator **ROCPROFILER_BUFFER_TRACING_OMPT**

See also

rocprofiler_ompt_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_MEMORY_ALLOCATION**

See also

rocprofiler_memory_allocation_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_RUNTIME_INITIALIZATION**

Record indicating a runtime library has been initialized.

See also

rocprofiler_runtime_initialization_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_ROCDECODE_API**

rocDecode tracing

enumerator **ROCPROFILER_BUFFER_TRACING_ROCJPEG_API**

rocJPEG tracing

enumerator **ROCPROFILER_BUFFER_TRACING_HIP_STREAM**

See also

rocprofiler_hip_stream_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_HIP_RUNTIME_API_EXT**

enumerator **ROCPROFILER_BUFFER_TRACING_HIP_COMPILER_API_EXT**

enumerator **ROCPROFILER_BUFFER_TRACING_ROCDECODE_API_EXT**

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_EVENT_PAGE_MIGRATE**

See also

rocprofiler_kfd_event_page_migrate_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_EVENT_PAGE_FAULT**

See also

rocprofiler_kfd_event_page_fault_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_EVENT_QUEUE**

See also

rocprofiler_kfd_event_queue_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_EVENT_UNMAP_FROM_GPU**

See also

rocprofiler_kfd_event_unmap_from_gpu_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_EVENT_DROPPED_EVENTS**

See also

rocprofiler_kfd_event_dropped_events_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_PAGE_MIGRATE**

See also

rocprofiler_kfd_page_migrate_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_PAGE_FAULT**

> **See also**
>
> rocprofiler_kfd_page_fault_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_KFD_QUEUE**

> **See also**
>
> rocprofiler_kfd_queue_operation_t

enumerator **ROCPROFILER_BUFFER_TRACING_MARKER_CORE_RANGE_API**

> **See also**
>
> rocprofiler_marker_core_range_api_id_t

enumerator **ROCPROFILER_BUFFER_TRACING_LAST**

enumerator **ROCPROFILER_BUFFER_TRACING_HIP_RUNTIME_API_EXT**

enumerator **ROCPROFILER_BUFFER_TRACING_HIP_COMPILER_API_EXT**

enumerator **ROCPROFILER_BUFFER_TRACING_ROCDECODE_API_EXT**

enum **rocprofiler_code_object_operation_t**

ROCProfiler Code Object Tracer Operations.

Values:

enumerator **ROCPROFILER_CODE_OBJECT_NONE**

Unknown code object operation.

enumerator **ROCPROFILER_CODE_OBJECT_LOAD**

Code object containing kernel symbols.

enumerator **ROCPROFILER_CODE_OBJECT_DEVICE_KERNEL_SYMBOL_REGISTER**

Kernel symbols - Device.

enumerator **ROCPROFILER_CODE_OBJECT_HOST_KERNEL_SYMBOL_REGISTER**

Kernel symbols - Host.

enumerator ROCPROFILER_CODE_OBJECT_LAST

enum rocprofiler_hip_stream_operation_t

ROCProfiler HIP Stream Operations. These operations can be used to associate subsequent information with a HIP stream.

Values:

enumerator ROCPROFILER_HIP_STREAM_NONE

Unknown stream handle operation.

enumerator ROCPROFILER_HIP_STREAM_CREATE

A stream handle is created.

enumerator ROCPROFILER_HIP_STREAM_DESTROY

A stream handle is destroyed.

enumerator ROCPROFILER_HIP_STREAM_SET

enumerator ROCPROFILER_HIP_STREAM_LAST

enumerator ROCPROFILER_HIP_STREAM_SET

enum rocprofiler_memory_copy_operation_t

Memory Copy Operations.

Values:

enumerator ROCPROFILER_MEMORY_COPY_NONE

Unknown memory copy direction.

enumerator ROCPROFILER_MEMORY_COPY_HOST_TO_HOST

Memory copy from host to host.

enumerator ROCPROFILER_MEMORY_COPY_HOST_TO_DEVICE

Memory copy from host to device.

enumerator ROCPROFILER_MEMORY_COPY_DEVICE_TO_HOST

Memory copy from device to host.

enumerator ROCPROFILER_MEMORY_COPY_DEVICE_TO_DEVICE

Memory copy from device to device.

enumerator ROCPROFILER_MEMORY_COPY_LAST

enum **rocprofiler_memory_allocation_operation_t**

Memory Allocation Operation.

Values:

enumerator **ROCPROFILER_MEMORY_ALLOCATION_NONE**

Unknown memory allocation function.

enumerator **ROCPROFILER_MEMORY_ALLOCATION_ALLOCATE**

Allocate memory function.

enumerator **ROCPROFILER_MEMORY_ALLOCATION_VMEM_ALLOCATE**

Allocate vmem memory handle.

enumerator **ROCPROFILER_MEMORY_ALLOCATION_FREE**

Free memory function.

enumerator **ROCPROFILER_MEMORY_ALLOCATION_VMEM_FREE**

Release vmem memory handle.

enumerator **ROCPROFILER_MEMORY_ALLOCATION_LAST**

enum **rocprofiler_kernel_dispatch_operation_t**

ROCProfiler Kernel Dispatch Tracing Operation Types.

Values:

enumerator **ROCPROFILER_KERNEL_DISPATCH_NONE**

Unknown kernel dispatch operation.

enumerator **ROCPROFILER_KERNEL_DISPATCH_ENQUEUE**

enumerator **ROCPROFILER_KERNEL_DISPATCH_COMPLETE**

enumerator **ROCPROFILER_KERNEL_DISPATCH_LAST**

enumerator **ROCPROFILER_KERNEL_DISPATCH_ENQUEUE**

enumerator **ROCPROFILER_KERNEL_DISPATCH_COMPLETE**

enum **rocprofiler_pc_sampling_method_t**

PC Sampling Method.

Values:

enumerator **ROCPROFILER_PC_SAMPLING_METHOD_NONE**

Unknown sampling type.

enumerator ROCPROFILER_PC_SAMPLING_METHOD_STOCHASTIC

Stochastic sampling (MI300+)

enumerator ROCPROFILER_PC_SAMPLING_METHOD_HOST_TRAP

Interval sampling (MI200+)

enumerator ROCPROFILER_PC_SAMPLING_METHOD_LAST

enum rocprofiler_pc_sampling_unit_t

PC Sampling Unit.

Values:

enumerator ROCPROFILER_PC_SAMPLING_UNIT_NONE

Sample interval has unspecified units.

enumerator ROCPROFILER_PC_SAMPLING_UNIT_INSTRUCTIONS

Sample interval is in instructions.

enumerator ROCPROFILER_PC_SAMPLING_UNIT_CYCLES

Sample interval is in cycles.

enumerator ROCPROFILER_PC_SAMPLING_UNIT_TIME

Sample internval is in nanoseconds.

enumerator ROCPROFILER_PC_SAMPLING_UNIT_LAST

enum rocprofiler_buffer_policy_t

Actions when Buffer is full.

Values:

enumerator ROCPROFILER_BUFFER_POLICY_NONE

No policy has been set.

enumerator ROCPROFILER_BUFFER_POLICY_DISCARD

Drop records when buffer is full.

enumerator ROCPROFILER_BUFFER_POLICY_LOSSLESS

Block when buffer is full.

enumerator ROCPROFILER_BUFFER_POLICY_LAST

enum rocprofiler_scratch_memory_operation_t

Scratch event kind.

Values:

enumerator **ROCPROFILER_SCRATCH_MEMORY_NONE**

Unknown scratch operation.

enumerator **ROCPROFILER_SCRATCH_MEMORY_ALLOC**

Scratch memory allocation event.

enumerator **ROCPROFILER_SCRATCH_MEMORY_FREE**

Scratch memory free event.

enumerator **ROCPROFILER_SCRATCH_MEMORY_ASYNC_RECLAIM**

Scratch memory asynchronously reclaimed.

enumerator **ROCPROFILER_SCRATCH_MEMORY_LAST**

enum **rocprofiler_runtime_library_t**

Enumeration for specifying runtime libraries supported by rocprofiler. This enumeration is used for thread creation callbacks.

See also

Internal Thread Handling.

Values:

enumerator **ROCPROFILER_LIBRARY**

enumerator **ROCPROFILER_HSA_LIBRARY**

enumerator **ROCPROFILER_HIP_LIBRARY**

enumerator **ROCPROFILER_MARKER_LIBRARY**

enumerator **ROCPROFILER_RCCL_LIBRARY**

enumerator **ROCPROFILER_ROCDECODE_LIBRARY**

enumerator **ROCPROFILER_ROCJPEG_LIBRARY**

enumerator **ROCPROFILER_LIBRARY_LAST**

enum **rocprofiler_intercept_table_t**

Enumeration for specifying intercept tables supported by rocprofiler. This enumeration is used for intercept tables.

See also

Intercept table for runtime libraries.

Values:

enumerator **ROCPROFILER_HSA_TABLE**

enumerator **ROCPROFILER_HIP_RUNTIME_TABLE**

enumerator **ROCPROFILER_HIP_COMPILER_TABLE**

enumerator **ROCPROFILER_MARKER_CORE_TABLE**

enumerator **ROCPROFILER_MARKER_CONTROL_TABLE**

enumerator **ROCPROFILER_MARKER_NAME_TABLE**

enumerator **ROCPROFILER_RCCL_TABLE**

enumerator **ROCPROFILER_ROCDECODE_TABLE**

enumerator **ROCPROFILER_ROCJPEG_TABLE**

enumerator **ROCPROFILER_TABLE_LAST**

enum **rocprofiler_runtime_initialization_operation_t**

ROCProfiler Runtime Initialization Tracer Operations.

Values:

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_NONE**

Unknown runtime initialization.

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_HSA**

Application loaded HSA runtime.

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_HIP**

Application loaded HIP runtime.

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_MARKER**

Application loaded Marker (ROCTx) runtime.

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_RCCL**

Application loaded RCCL runtime.

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_ROCDECODE**

Application loaded rocDecoder runtime.

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_ROCJPEG**

Application loaded rocJPEG runtime.

enumerator **ROCPROFILER_RUNTIME_INITIALIZATION_LAST**

enum **rocprofiler_counter_info_version_id_t**

Enumeration for specifying the counter info struct version you want.

Values:

enumerator **ROCPROFILER_COUNTER_INFO_VERSION_NONE**

enumerator **ROCPROFILER_COUNTER_INFO_VERSION_0**

See also

rocprofiler_counter_info_v0_t

enumerator **ROCPROFILER_COUNTER_INFO_VERSION_1**

See also

rocprofiler_counter_info_v1_t

enumerator **ROCPROFILER_COUNTER_INFO_VERSION_LAST**

enum **rocprofiler_counter_record_kind_t**

Enumeration for distinguishing different buffer record kinds within the ROCPROFILER_BUFFER_CATEGORY_COUNTERS category.

Values:

enumerator **ROCPROFILER_COUNTER_RECORD_NONE**

enumerator **ROCPROFILER_COUNTER_RECORD_PROFILE_COUNTING_DISPATCH_HEADER**

rocprofiler_dispatch_counting_service_record_t

enumerator **ROCPROFILER_COUNTER_RECORD_VALUE**

enumerator **ROCPROFILER_COUNTER_RECORD_LAST**

enum **rocprofiler_counter_flag_t**

Enumeration of flags that can be used with some counter api calls.

Values:

enumerator **ROCPROFILER_COUNTER_FLAG_NONE**

enumerator **ROCPROFILER_COUNTER_FLAG_ASYNC**

Do not wait for completion before returning.

enumerator **ROCPROFILER_COUNTER_FLAG_APPEND_DEFINITION**

Append the counter definition to the system provided counter definition file.

enumerator **ROCPROFILER_COUNTER_FLAG_LAST**

enum **rocprofiler_pc_sampling_record_kind_t**

Enumeration for distinguishing different buffer record kinds within the ROCPROFILER_BUFFER_CATEGORY_PC_SAMPLING category.

Values:

enumerator **ROCPROFILER_PC_SAMPLING_RECORD_NONE**

enumerator **ROCPROFILER_PC_SAMPLING_RECORD_INVALID_SAMPLE**

rocprofiler_pc_sampling_record_invalid_t

enumerator **ROCPROFILER_PC_SAMPLING_RECORD_HOST_TRAP_V0_SAMPLE**

rocprofiler_pc_sampling_record_host_trap_v0_t

enumerator **ROCPROFILER_PC_SAMPLING_RECORD_STOCHASTIC_V0_SAMPLE**

rocprofiler_pc_sampling_record_stochastic_v0_t

enumerator **ROCPROFILER_PC_SAMPLING_RECORD_LAST**

typedef uint64_t **rocprofiler_timestamp_t**

ROCProfiler Timestamp.

typedef uint64_t **rocprofiler_thread_id_t**

Thread ID. Value will be equivalent to `syscall(__NR_gettid)`

typedef int32_t **rocprofiler_tracing_operation_t**

Tracing Operation ID. Depending on the kind, operations can be determined. If the value is equal to zero that means all operations will be considered for tracing. Detailed API tracing operations can be found at associated header file for that partiular operation. i.e: For ROCProfiler enumeration of HSA AMD Extended API tracing operations, look at source/include/rocprofiler-sdk/hsa/amd_ext_api_id.h.

typedef uint64_t **rocprofiler_kernel_id_t**

Kernel identifier type.

typedef uint64_t **rocprofiler_dispatch_id_t**

typedef uint64_t **rocprofiler_counter_instance_id_t**

Unique record id encoding both the counter and dimensional values (positions) for the record.

typedef uint64_t **rocprofiler_counter_dimension_id_t**

A dimension for counter instances. Some example dimensions include XCC, SM (Shader), etc. This value represents the dimension beind described or queried about.

typedef *rocprofiler_counter_record_dimension_info_t* **rocprofiler_record_dimension_info_t**

typedef *rocprofiler_counter_record_t* **rocprofiler_record_counter_t**

static inline uint64_t **rocprofiler_record_header_compute_hash**(uint32_t category, uint32_t kind)

Function for computing the unsigned 64-bit hash value in *rocprofiler_record_header_t* from a category and kind (two unsigned 32-bit values)

Parameters

- **category** – **[in]** a value from *rocprofiler_buffer_category_t*
- **kind** – **[in]** depending on the category, this is the domain value, e.g., *rocprofiler_buffer_tracing_kind_t* value

Returns

uint64_t hash value of category and kind

ROCPROFILER_CORRELATION_ID_INTERNAL_NONE

The NULL value of an internal correlation ID.

ROCPROFILER_CORRELATION_ID_ANCESTOR_NONE

The NULL value of an ancestor correlation ID.

union **rocprofiler_user_data_t**

#include <rocprofiler-sdk/fwd.h> User-assignable data type.

Public Members

uint64_t **value**

usage example: set to process id, thread id, etc.

void ***ptr**

usage example: set to address of data allocation

union **rocprofiler_address_t**

#include <rocprofiler-sdk/fwd.h> Stores memory address for profiling.

Public Members

uint64_t **handle**

compatability

uint64_t **value**

usage example: store address in uint64_t format

const void ***ptr**

usage example: generic form of address

struct **rocprofiler_uuid_t**

#include <*rocprofiler-sdk/fwd.h*> Stores UUID for devices.

struct **rocprofiler_version_triplet_t**

#include <*rocprofiler-sdk/fwd.h*> Versioning info.

struct **rocprofiler_context_id_t**

#include <*rocprofiler-sdk/fwd.h*> Context ID.

struct **rocprofiler_queue_id_t**

#include <*rocprofiler-sdk/fwd.h*> Queue ID.

struct **rocprofiler_stream_id_t**

#include <*rocprofiler-sdk/fwd.h*> Stream ID.

struct **rocprofiler_correlation_id_t**

#include <*rocprofiler-sdk/fwd.h*> ROCProfiler Record Correlation ID.

struct **rocprofiler_async_correlation_id_t**

#include <*rocprofiler-sdk/fwd.h*> ROCProfiler Correlation ID record for async activity.

struct **rocprofiler_buffer_id_t**

#include <*rocprofiler-sdk/fwd.h*> Buffer ID.

struct **rocprofiler_agent_id_t**

#include <*rocprofiler-sdk/fwd.h*> Agent Identifier.

struct **rocprofiler_counter_id_t**

#include <*rocprofiler-sdk/fwd.h*> Counter ID.

struct **rocprofiler_counter_config_id_t**

#include <*rocprofiler-sdk/fwd.h*> Profile Configurations.

See also

rocprofiler_create_counter_config for how to create.

struct **rocprofiler_dim3_t**

#include <rocprofiler-sdk/fwd.h> Multi-dimensional struct of data used to describe GPU workgroup and grid sizes.

struct **rocprofiler_callback_tracing_record_t**

#include <rocprofiler-sdk/fwd.h> Tracing record.

struct **rocprofiler_record_header_t**

#include <rocprofiler-sdk/fwd.h> Generic record with type identifier(s) and a pointer to data. This data type is used with buffered data.

```
void
tool_tracing_callback(rocprofiler_record_header_t** headers,
                      size_t                        num_headers)
{
    for(size_t i = 0; i < num_headers; ++i)
    {
        rocprofiler_record_header_t* header = headers[i];

        if(header->category == ROCPROFILER_BUFFER_CATEGORY_TRACING &&
           header->kind == ROCPROFILER_BUFFER_TRACING_HSA_API)
        {
            // cast to rocprofiler_buffer_tracing_hsa_api_record_t which
            // is type associated with this category + kind
            auto* record =
                static_cast<rocprofiler_buffer_tracing_hsa_api_record_t*>(header->
↪payload);

            // trivial test
            assert(record->start_timestamp <= record->end_timestamp);
        }
    }
}
```

struct **rocprofiler_kernel_dispatch_info_t**

#include <rocprofiler-sdk/fwd.h> ROCProfiler kernel dispatch information.

struct **rocprofiler_counter_record_dimension_info_t**

#include <rocprofiler-sdk/fwd.h> (experimental) Details for the dimension, including its size, for a counter record.

struct **rocprofiler_counter_record_t**

#include <rocprofiler-sdk/fwd.h> (experimental) ROCProfiler Profile Counting Counter Record per instance.

rocprofiler_record_header_t.__unnamed16__

Public Members

struct rocprofiler_record_header_t

uint64_t **hash**

generic identifier. You can compute this via: `uint64_t hash = category | ((uint64_t)(kind) << 32)`

rocprofiler_record_header_t.__unnamed16__.__unnamed18__

19.2.2 Topics

19.2.3 Data Structures

19.2.4 File List

CHAPTER

TWENTY

ROCTX API

20.1 Introduction

ROCTx is a comprehensive library that implements the AMD code annotation API. It provides essential functionality for:

- Event annotation and marking
- Code range tracking and management
- Profiler control and customization
- Thread and device naming capabilities

Key features:

- Nested range tracking with push/pop functionality
- Process-wide range management
- Thread-specific and global profiler control
- Device and stream naming support
- HSA agent and HIP device management

The API is divided into several main components:

1. **Markers** - For single event annotations
2. **Ranges** - For tracking code execution spans
3. **Profiler Control** - For managing profiling tool behavior
4. **Naming Utilities** - For labeling threads, devices, and streams

Thread Safety:

- Range operations are thread-local and thread-safe
- Marking operations are thread-safe
- Profiler control operations are process-wide

Integration:

- Compatible with HIP runtime
- Supports HSA (Heterogeneous System Architecture)
- Provides both C and C++ interfaces

Performance Considerations:

- Minimal overhead for marking and range operations
- Thread-local storage for efficient range stacking
- Lightweight profiler control mechanisms

Note

All string parameters must be null-terminated.

Warning

Proper nesting of range push/pop operations is the user's responsibility.

This ROCTx API topic broadly covers:

- *Modules*
- *Global Data structures, topics, files*

20.2 Modules

The ROCprofiler-SDK-ROCTx API is organized into the following modules based on functionality:

- *Markers Information*
- *Ranges Information*
- *Profiler Control Information*
- *Naming Information*

20.2.1 Markers Information

void **roctxMarkA**(const char *message)

Mark an event.

Parameters

message – **[in]** The message associated with the event.

20.2.2 Ranges Information

int **roctxRangePushA**(const char *message)

Start a new nested range.

Nested ranges are stacked and local to the current CPU thread.

Parameters

message – **[in]** The message associated with this range.

Returns

Returns the level this nested range is started at. Nested range levels are 0 based.

int **roctxRangePop**()

Stop the current nested range.

Stop the current nested range, and pop it from the stack. If a nested range was active before the last one was started, it becomes again the current nested range.

Returns
Returns the level the stopped nested range was started at, or a negative value if there was no nested range active.

roctx_range_id_t **roctxRangeStartA**(const char *message)

Starts a process range.

Start/stop ranges can be started and stopped in different threads. Each timespan is assigned a unique range ID.

Parameters
message – **[in]** The message associated with this range.

Returns
Returns the ID of the new range.

void **roctxRangeStop**(*roctx_range_id_t* id)

Stop a process range.

Parameters
id – **[in]** *roctx_range_id_t* returned from *roctxRangeStartA* to stop

20.2.3 Profiler Control Information

int **roctxProfilerPause**(*roctx_thread_id_t* tid)

Request any currently running profiling tool that is should stop collecting data.

Within a profiling tool, it is recommended that the tool cache all active contexts at the time of the request and then stop them. By convention, the application should pass zero to indicate a global pause of the profiler in the current process. If the application wishes to pause only the current thread, the application should obtain the thread ID via *roctxGetThreadId*.

Parameters
tid – **[in]** Zero for all threads in current process or non-zero for a specific thread

Returns
int A profiling tool may choose to set this value to a non-zero value to indicate a failure while executing the request or lack of support. If the profiling tool supports pausing but is already paused, the tool should ignore the request and return zero.

int **roctxProfilerResume**(*roctx_thread_id_t* tid)

Request any currently running profiling tool that is should resume collecting data.

Within a profiling tool, it is recommended that the tool re-activated the active contexts which were cached when the pause request was issued. By convention, the application should pass zero to indicate a global pause of the profiler in the current process. If the application wishes to pause only the current thread, the application should obtain the thread ID via *roctxGetThreadId*.

Parameters
tid – **[in]** Zero for all threads in current process or non-zero for a specific thread

Returns
int A profiling tool may choose to set this value to a non-zero value to indicate a failure while executing the request or lack of support. If the profiling tool is supports resuming but is already active, the tool should ignore the request and return zero.

20.2.4 Naming Information

int **roctxNameOsThread**(const char *name)

Indicate to a profiling tool that, where possible, you would like the current CPU OS thread to be labeled by the provided name in the output of the profiling tool.

Rocprofiler does not provide explicit support for how profiling tools handle this request: support for this capability is tool specific. ROCTx does NOT rename the thread via `pthread_setname_np`.

Parameters
name – **[in]** Name for the current OS thread
Returns
int A profiling tool may choose to set this value to a non-zero value to indicate a failure while executing the request or lack of support

int **roctxNameHsaAgent**(const char *name, const struct hsa_agent_s *agent)

Indicate to a profiling tool that, where possible, you would like the given HSA agent to be labeled by the provided name in the output of the profiling tool.

Rocprofiler does not provide any explicit support for how profiling tools handle this request: support for this capability is tool specific.

Parameters
- **name** – **[in]** Name for the specified agent
- **agent** – **[in]** Pointer to a HSA agent identifier

Returns
int A profiling tool may choose to set this value to a non-zero value to indicate a failure while executing the request or lack of support

int **roctxNameHipDevice**(const char *name, int device_id)

Indicate to a profiling tool that, where possible, you would like the given HIP device id to be labeled by the provided name in the output of the profiling tool.

Rocprofiler does not provide any explicit support for how profiling tools handle this request: support for this capability is tool specific.

Parameters
- **name** – **[in]** Name for the specified device
- **device_id** – **[in]** HIP device ordinal

Returns
int A profiling tool may choose to set this value to a non-zero value to indicate a failure while executing the request or lack of support

int **roctxNameHipStream**(const char *name, const struct ihipStream_t *stream)

Indicate to a profiling tool that, where possible, you would like the given HIP stream to be labeled by the provided name in the output of the profiling tool.

Rocprofiler does not provide any explicit support for how profiling tools handle this request: support for this capability is tool specific.

Parameters
- **name** – **[in]** Name for the specified stream
- **stream** – **[in]** A `hipStream_t` value (hipStream_t == ihipStream_t*)

Returns
int A profiling tool may choose to set this value to a non-zero value to indicate a failure while executing the request or lack of support

int **roctxGetThreadId**(*roctx_thread_id_t* *tid)

Retrieve a id value for the current thread which will be identical to the id value a profiling tool gets via `rocprofiler_get_thread_id(rocprofiler_thread_id_t*)`

Parameters
tid – [out] Pointer to where the value should be placed
Returns
int A profiling tool may choose to set this value to a non-zero value to indicate a failure while executing the request or lack of support

20.3 Global Data structures, topics, files

This ROCprofiler-SDK-ROCTx API topic covers:

- *Global Basic Data Types*
- *Topics*
- *File List*

20.3.1 Global Basic Data Types

typedef uint64_t **roctx_range_id_t**

ROCTx range ID.

ROCTx range ID.

This is the range ID used to identify start/end ranges.

This is the range ID used to identify start/end ranges.

typedef uint64_t **roctx_thread_id_t**

ROCTx thread ID.

This is the thread ID used to identify OS threads

20.3.2 Topics

20.3.3 File List

CHAPTER

TWENTYONE

COMPARING ROCPROFILER-SDK TO OTHER ROCM PROFILING TOOLS

ROCprofiler-SDK is an improved version of ROCm profiling tools that enables more efficient implementations and better thread safety while avoiding problems that plague the former implementations of ROCProfiler and ROCTracer. Here are the distinct ROCprofiler-SDK features, which also highlight the improvements over ROCProfiler and ROCTracer:

- Improved tool initialization
- Support for simultaneous use of the same services by multiple tools
- Simplified control of one or more data collection services
- Improved error checking and logging
- Backward ABI compatibility
- PC sampling (beta implementation)

The former implementations allow a tool to access any of the services provided by ROCProfiler or ROCTracer, such as API tracing and kernel tracing, by calling `roctracer_init()` when an ROCm runtime is initially loaded. As the calling tool is not required to specify during initialization, the services it needs to use, the libraries must be effectively prepared for any service to be available anytime. This behavior introduces unnecessary overhead and makes thread-safe data management difficult, as tools generally don't use all the available services. For example, ROCTracer always installs wrappers around every runtime API and adds indirection overhead through the ROCTracer library to check for the current service configuration in a thread-safe manner.

ROCprofiler-SDK introduces *context* to solve the preceding issues. Contexts are effectively bundles of service configurations. ROCprofiler-SDK provides a single opportunity for a tool to create as many contexts as required. A tool can group all services into one context, create one context per service, or choose a mix. This change in the design allows ROCprofiler-SDK to be aware of the services that might be requested by a tool at any given time. The design change empowers ROCprofiler-SDK to:

- Avoid unnecessary preparation for services that are never used. If no registered contexts request HSA API tracing, no wrappers need to be generated.
- Perform more extensive checks during service specification and inform a tool about potential issues early.
- Allow multiple tools to use certain services simultaneously.
- Improve thread safety without introducing parallel bottlenecks.
- Manage internal data and allocations more efficiently.

CHAPTER

TWENTYTWO

COMPARING COMMAND-LINE TOOL OPTIONS: ROCPROFILER(ROCPROF, ROCPROFV2) AND ROCPROFILER-SDK(ROCPROFV3)

ROCprofiler-SDK introduces a new command-line tool, *rocprofv3*, which is a more efficient and flexible version of the ROCprofiler tool.

Table 22.1: Comparison of ROCprofiler Command-Line Tool's options

Category	Feature	rocprof	rocprofv2	rocprofv3	Improve-ments	Notes
Basic tracing options	HIP Trace	*–hip-trace*	*–hip-api*, *–hip-trace*	*–hip-trace*	No change	rocprof and rocprofv2 *–hip-trace* options include kernel dispatches and memory copy activities, which is not the case in rocprofv3
Basic tracing options	HSA Trace	*–hsa-trace*	*–hsa-trace*	*–hsa-trace*	No change	rocprof and rocprofv2 *–hsa-trace* options include kernel dispatches and memory copy activities, which is not the case in rocprofv3

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Basic tracing options	Scratch Memory Trace	*Not Available*	*Not Available*	*–scratch-memory-trace*	New option to trace scratch memory operations	
Basic tracing options	Marker Trace (ROCTx)	*–roctx-trace*	*–roctx-trace*	*–marker-trace*	Improved ROCTx library with more features	
Basic tracing options	Memory Copy Trace	Part of HIP and HSA Traces	Part of HIP and HSA Traces	*–memory-copy-trace*	Provides granularity for memory move operations	
Basic tracing options	Memory allocation Trace	*Not Available*	*Not Available*	*–memory-allocation-trace*	New option for collecting Memory Allocation Traces. Displays starting address, allocation size, and agent where allocation occurred.	
Basic tracing options	Kernel Trace	*–kernel-trace*	*–kernel-trace*	*–kernel-trace*	Performance improvement.	
Granular tracing options	HIP runtime trace	Part of *–hip-trace* option	Part of *–hip-trace* option	*–hip-runtime-trace*	For collecting HIP Runtime API Traces, e.g. public HIP API functions starting with 'hip' (i.e. hipSetDevice).	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Granular tracing options	HIP compiler trace	*Not Available*	*Not Available*	*–hip-compiler-trace*	For collecting HIP Compiler generated code Traces, e.g. HIP API functions starting with '__hip' (i.e. __hipRegisterFatBinary).	
Granular tracing options	HSA core API trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-core-trace*	New option for collecting only HSA API Traces (core API), e.g. HSA functions prefixed with only *hsa_* (i.e. hsa_init)	
Granular tracing options	HSA AMD trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-amd-trace*	For collecting HSA API Traces (AMD-extension API), e.g. HSA function prefixed with *hsa_amd_* (i.e. hsa_amd_coher	
Granular tracing options	HSA Image Extension trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-image-trace*	New option for collecting HSA API Traces (Image-extension API), e.g. HSA functions prefixed with only *hsa_ext_image_* (i.e. hsa_ext_image_	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Granular tracing options	HSA Finalizer trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-finalizer-trace*	New option for collecting HSA API Traces (Finalizer-extension API), e.g. HSA functions prefixed with only *hsa_ext_progra* (i.e. hsa_ext_progra	
Advanced tracing options	Kokkos trace	*Not Available*	*Not Available*	*–kokkos-trace*	New option to enable built-in Kokkos Tools support (implies –marker-trace and –kernel-rename)	
Advanced tracing options	RCCL trace	*Not Available*	*Not Available*	*–rccl-trace*	For collecting RCCL (ROCm Communication Collectives Library. Also pronounced as 'Rickle') Traces	
Advanced tracing options	Scratch memory trace	*Not Available*	*Not Available*	*–scratch-memory-trace*	Collecting scratch memory event traces.	
Advanced tracing options	rocDecode trace	*Not Available*	*Not Available*	*–rocdecode-trace*	Tracing rocDecode library.	
Advanced tracing options	rocJPEG trace	*Not Available*	*Not Available*	*–rocjpeg-trace*	Tracing rocJPEG library.	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Aggregate tracing options	Sys Trace	*–sys-trace* [hip-trace|hsa-trace|roctx-trace|kernel-trace]	*–sys-trace* [hip-trace|hsa-trace|roctx-trace|kernel-trace]	` -s, –sys-trace` [hip-trace|hsa-trace|scratch-trace|memory-copy-trace|roctx-trace|kernel-trace]	Extends the sys trace options with more features	
Aggregate tracing options	Runtime Trace	*Not available*	*Not available*	` -r, –runtime-trace` [hip-runtime-trace|scratch-trace|memory-copy-trace|roctx-trace|kernel-trace]	New option to aggregate trace operations	
Kernel naming options	Kernel Name Mangling	*Not Available*	*Not Available*	*-M*, *–mangled-kernels*	New option for mangled kernel names	
Kernel naming options	Kernel Name Truncation	*–basenames* <*on*|*off*>	*–basenames*	*-T*, *–truncate-kernels*	New option for truncating the demangled kernel names	
Kernel naming options	Kernel Rename	*–roctx-rename*	*Not available*	*–kernel-rename*	New option to use region names defined by roctxRangePush/roctxRang regions to rename the kernels	
Post-processing tracing options	Statistics	–stats	*Not Available*	–stats	Statistics for the collected traces	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Post-processing tracing options	Summary	*Not available*	*Not available*	*-S, –summary*	New option to output a single summary of tracing data after the profiling session	*rocprof* generated the post-processing step's summary, stats, JSON, and database files with much less information.
Post-processing tracing options	Summary Per Domain	*Not available*	*Not available*	*-D, –summary-per-domain*	New option to output summary for each tracing domain after the profiling session	*rocprof –stats* option had less number of domains in the summary reports than *rocprofv3*
Post-processing tracing options	Summary Groups	*Not available*	*Not available*	*–summary-groups REGULAR_EXPRESS*	New option to output a summary for each set of domains matching the regular expression, e.g. 'KERNEL_DISPAT(will generate a summary from all the tracing data in the KERNEL_DISPAT(and MEMORY_COPY domains	COPY'
Summary options	Summary Output File	*Not available*	*Not available*	*–summary-output-file SUMMARY_OUTPU*	New option to output summary to a file, stdout, or stderr (default: stderr)	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Summary options	Summary Units	*Not available*	*Not available*	*-u , –summary-units*	New option to output summary in desired time units {sec,msec,usec	
Display options	List available basic and derived metrics and PC sampling configurations	*–list-basic*, *–list-derived*	*–list-counters*	*-L*, *–list-avail*	A valid YAML is supported for this option now	
Perfetto-specific options	Perfetto data collection backend	*Not available*	*Not available*	*–perfetto-backend* {in-process,system	New option for perfetto data collection backend. 'system' mode requires starting traced and perfetto daemons	*rocprofv2* used only in-process collection for perfetto plugin, However, *rocprofv3* gives the user the option.
Perfetto-specific options	Perfetto Buffer Size	*Not available*	Setting env variable *rocprofiler_PERFETT* to the desired buffer size	*–perfetto-buffer-size* {KB}	New option to define size of buffer for perfetto output in KB. default: 1 GB	
Perfetto-specific options	Perfetto Buffer fill Policy	*Not available*	*Not available*	*–perfetto-buffer-fill-policy* {discard,ring_buffe	New option or handling new records when perfetto has reached the buffer limit	*rocprofv2* always used *TraceConfig_BufferConfig_FillPolicy_RIN* fill policy.
Perfetto-specific options	Perfetto shared memory size	*Not available*	*Not available*	*–perfetto-shmem-size-hint* KB	New option to define perfetto shared memory size hint in KB. default: 64 KB	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Filtering options	Kernel Filtration options for Counter Collection	Supported in input.xml file (supports range, gpu and kernel filtration)	kernel: <kernel_name> (can only be provided in input.txt file)	*–kernel-include-regex*, *–kernel-exclude-regex*, *–kernel-iteration-range*	Extensive control over output options using regular expressions	
I/O options	Output Directory	*-d* <data directory>	*-d* | *–output-directory*	*-d* OUTPUT_DIRECT(*–output-directory* OUTPUT_DIRECT(	rocprofv3 supports special keys for runtime values, e.g. %pid% gets replaced by the process ID	
I/O options	Output File	*-o* <output file>	*-o* | *–output-file-name*	*-o* OUTPUT_FILE, *–output-file* OUTPUT_FILE	rocprofv3 supports special keys for runtime values, e.g. %pid% gets replaced by the process ID	
I/O options	Logging	Minimal logging via environment variable	Minimal logging via environment variable	–log-level {fatal,error,warnin	Extensive logging options	
I/O options	Plugins	*Not Available*	plugin support for different output formats	Replaced by *–output-format* option	Not needed as rocprofv3 supports multiple output formats	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
I/O options	Output Formats	CSV, JSON (Chrome-Tracing format)	CSV, JSON (Chrome-Tracing format), Perfetto, CTF	CSV, JSON (custom schema), Perfetto, OTF2	# Multiple output formats can be supported in single run. # OTF2 can visualize larger trace files compared to perfetto.	The Perfetto UI does not accept the JSON output format produced by rocprofv3. Perfetto is dropping support for the JSON Chrome tracing format in favor of the binary Perfetto protobuf format (`.pftrace` extension), which is supported by rocprofv3.
I/O options	Counter Collection	Supports input text and XML format	Only supports input text format	Input support for text, YAML and JSON formats	# It's not possible to check for valid text file. Hence rocprofv3 supports strongly typed input formats. # YAML and JSON formats are more readable and easy to maintain. # Allows flexibility to add more features for the tool input	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
I/O options	Command-line Counter Collection	*Not Available*	*Not Available*	*–pmc*	New option to collect performance counters from command line. Counters should be comma OR space separated in case of more than 1 counters	
I/O options	Providing Custom metrics file	*-m* <metric file>	*-m* <metric file>	*-E* <metric file> –pmc <counter>	In rocprofv3, this option has changed to provide a file with custom metrics and collect performance counters from the command line using –pmc option	
Advanced options	Preload	*Not Available*	*Not Available*	–preload	Libraries to prepend to LD_PRELOAD (usually for sanitizers)	
Trace Control options	Trace Period	*–trace-period*	*-tp* | *–trace-period*	*-P* |*–collection-period*,`–collec period-unit`	Users can specify multiple configurations, each defined by a triplet in the format *start_delay:col* :*eat*, with the ability to change the unit of time in the given configurations.	
Trace Control options	Trace start	*–trace-start <on|off>*	*Not available*	*Not available*	Not yet in rocprofv3	

continues on next page

Table 22.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Trace Control options	Flush Interval	*–flush-rate*	*–flush-interval*	*Not available*	Not applicable for rocprofv3	
Trace Control options	Merge Traces	*–merge-traces*	*Not available*	*Not available*	Not yet in rocprofv3	
PC Sampling options	PC Sampling`	*Not available*	*Not available*	*–pc-sampling-beta-enabled*	Enable pc sampling support; beta version.	
Legacy options	Timestamp On/Off	*–timestamp <on|off>*	*Not available*	*Not available*	Not applicable for rocprofv3	
Legacy options	Context wait	*–ctx-wait*	*Not available*	*Not available*	Not applicable for rocprofv3	
Legacy options	Context Limit	*–ctx-limit <max number>*	*Not available*	*Not available*	Not applicable for rocprofv3	
Legacy options	Code Object Tracking	*–obj-tracking <on|off>*	Always `ON` in rocprofv2	Always `ON` in rocprofv3		
Legacy options	Heartbeat	*–heartbeat <rate sec>*	*Not available*	*Not available*	Not applicable for rocprofv3	

CHAPTER

TWENTYTHREE

TIMING DIFFERENCE BETWEEN ROCPROFV3 AND ROCPROFV1/V2

`rocprofv3` has improved the accuracy of timing information by reducing the tool overhead required to collect data and reducing the interference to the timing of the kernel being measured. The result of this work is a reduction in variance of kernel times received for the same kernel execution and more accurate timing in general. These changes have not been backported (and will not be backported) to rocprofv1/v2, so there can be substantial (20%) differences in execution time reported by v1/v2 vs v3 for a single kernel execution. Over a large number of samples of the same kernel, the difference in average execution time is in the low single digit percentage time with a much tighter variance of results on rocprofv3. We have included testing in the test suite to verify the timing information outputted by rocprofv3 to ensure that the values we are returning are accurate.

CHAPTER

TWENTYFOUR

DEFAULT RUN OF ROCPROFV3 AND ROCPROFV1/V2

`rocprofv3` has a different default behavior than rocprofv1/v2 when being run without any option. The default behavior of rocprofv3 is to collect all available agents on the system and to output it in `csv` format. The default behavior of rocprofv1/v2 was to output the *kernel traces* in CSV format. In rocprofv3, kernel traces can be obtained by using `--kernel-trace` option.

CHAPTER

TWENTYFIVE

LICENSE

INDEX